UML

Arquitectura de aplicaciones en Java, C++ y Python

2ª edición



Desde **www.ra-ma.es** podrá descargar material adicional.

Carlos Jiménez de Parga

Ra-Ma®

UML
Arquitectura de aplicaciones en Java, C++ y Python

2ª edición

Dr. Carlos Jiménez de Parga

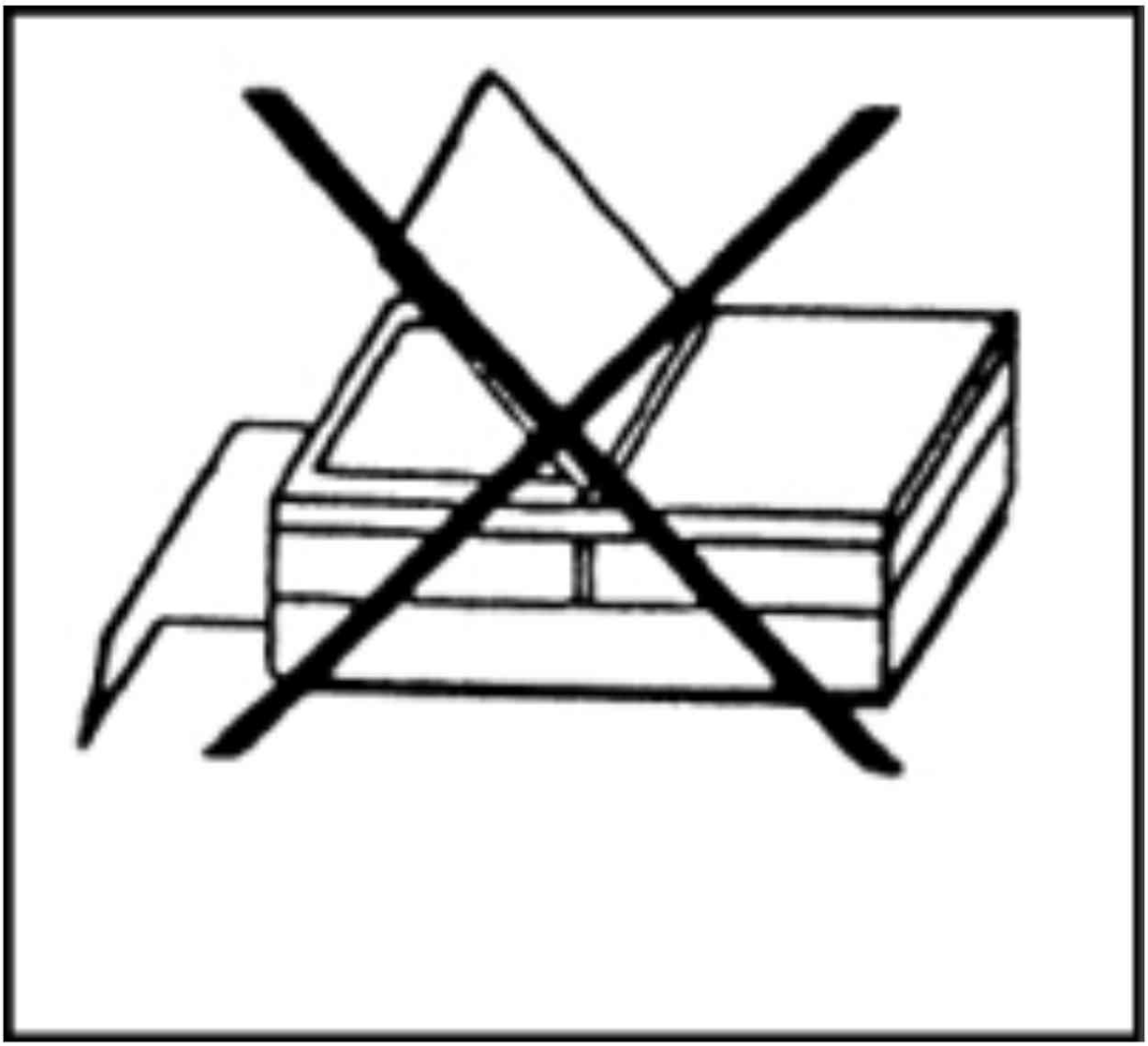
Ingeniero Informático / Ingeniero de Software

Revisión Técnica:

Dr. Manuel Arias Calleja



Ra-Ma®



La ley prohíbe

fotocopiar este libro

UML. Arquitectura de aplicaciones en Java, C++ y Python. 2ª Edición

© Carlos Jiménez de Parga

© De la edición: Ra-Ma 2021

MARCAS COMERCIALES. Las designaciones utilizadas por las empresas para distinguir sus productos (hardware, software, sistemas operativos, etc.) suelen ser marcas registradas. RA-MA ha intentado a lo largo de este libro distinguir las marcas comerciales de los términos descriptivos, siguiendo el estilo que utiliza el fabricante, sin intención de infringir la marca y solo en beneficio del propietario de la misma. Los datos de los ejemplos y pantallas

son ficticios a no ser que se especifique lo contrario.
RA-MA es marca comercial registrada.
Se ha puesto el máximo empeño en ofrecer al lector una información completa y precisa. Sin embargo, RA-MA Editorial no asume ninguna responsabilidad derivada de su uso ni tampoco de cualquier violación de patentes ni otros derechos de terceras partes que pudieran ocurrir. Esta publicación tiene por objeto proporcionar unos conocimientos precisos y acreditados sobre el tema tratado. Su venta no supone para el editor ninguna forma de asistencia legal, administrativa o de ningún otro tipo. En caso de precisarse asesoría legal u otra forma de ayuda experta, deben buscarse los servicios de un profesional competente.
Reservados todos los derechos de publicación en cualquier idioma.
Según lo dispuesto en el Código Penal vigente, ninguna parte de este libro puede ser reproducida, grabada en sistema de almacenamiento o transmitida en forma alguna ni por cualquier procedimiento, ya sea electrónico, mecánico, reprográfico, magnético o cualquier otro sin autorización previa y por escrito de RA-MA; su contenido está protegido por la ley vigente, que establece penas de prisión y/o multas a quienes, intencionadamente, reprodujeren o plagiaren, en todo o en parte, una obra literaria, artística o científica.
Editado por:
RA-MA Editorial
Calle Jarama, 3A, Polígono Industrial Igarsa
28860 PARACUELLOS DE JARAMA, Madrid
Teléfono: 91 658 42 80
Fax: 91 662 81 39
Correo electrónico: *editorial@ra-ma.com*
Internet: *www.ra-ma.es* y *www.ra-ma.com*
ISBN: 978-84-9964-977-1

Depósito legal: M-17905-2020
Maquetación: Antonio García Tomé
Diseño de portada: Antonio García Tomé
Imagen de portada: Antonio Jiménez de Parga
Filmación e impresión: Safekat
Impreso en España en marzo de 2021

A mis padres,a los que estaré eternamente agradecido por todo.
En memoria de mi tío Juanjo.
A usted, lector, y su familia.

Índice

prólogo 17

¿A QUIÉN VA DIRIGIDO ESTE LIBRO? 17
ESTRUCTURA DEL LIBRO 18
NOVEDADES EN LA SEGUNDA EDICIÓN 18
ESTILOS DE PROGRAMACIÓN 19
DIAGRAMAS Y PROGRAMAS DE EJEMPLO 19
FEEDBACK 20
AGRADECIMIENTOS ACADÉMICOS 20

1. UML en el contexto de la ingeniería del software 21

1. la necesidad de un modelo unificado 21
2. ¿por qué modelar? 22
3. Modelos de proceso software 23

1. Modelo en cascada 24
2. Modelo incremental 25
3. Modelo en espiral 26
4. Relación de las fases con los diagramas UML 27

4. metodologías de desarrollo 30

1. Metodologías ágiles 30
2. Metodologías pesadas 35

5. calidad del software 36

1. Definición y conceptos 36
2. Dimensiones de la calidad 37

3. Crisis del software y necesidad de una medida (métricas) 38
4. Normalización y certificación 38
5. La norma ISO/IEC 9126 38
6. Aseguramiento de la calidad del software (SQA) 39

6. desarrollo de software seguro 41

1. Introducción 41
2. Análisis de software seguro 42
3. Análisis de riesgos 43

7. mda 44

1. Introducción 44
2. Características de MDA 45

2. diagramas de casos de uso 49

1. introducción 49
2. actores 50
3. CASOS DE USO 50
4. Especificación de los casos de uso 52
5. Flujo principal 53

1. Sentencia condicional “si” 53
2. Sentencia de control “para” 54
3. Sentencia de control “mientras” 55

6. uso de <<include>> y <<extend>> 56
7. CASO de ESTUDIO: mercurial 60

8. CASO de ESTUDIO: servicio de cifrado remoto 64

3. diagramas de robustez 71

1. conceptos básicos 71
2. importancia de los diagramas de robustez 74
3. caso de estudio: ajedrez 75

1. Caso de uso “Hacer jugada” 75
2. Caso de uso “Configurar parámetros” 78
3. Caso de uso “Validar usuario” 79

4. caso de estudio: mercurial 79

1. Caso de uso “Listar ficheros” 79
2. Caso de uso “Clonar repositorio” 80

5. caso de estudio: servicio de cifrado remoto 80

1. Caso de uso “Encriptar” 80
2. Caso de uso “Desencriptar” 81

4. modelo del dominio 83

1. extracción de conceptos 83
2. identificar asociaciones 84
3. establecer los atributos 85
4. Herencia 86
5. agregación y composición 87
6. ejemplo de modelado: “spindizzy” 88
7. caso de estudio: ajedrez 92

8. caso de estudio: mercurial 94
9. caso de estudio: servicio de cifrado remoto 96

1. Cliente 96
2. Servidor 97

5. diagramas orientados a la arquitectura 99

1. taxonomía de estilos arquitectónicos 100

1. Tuberías y filtros 100
2. Por capas 101
3. Repositorios 102
4. Intérprete 102
5. Basadas en eventos 103
6. Distribuidas 103
7. Programa principal / subprograma 104

2. diagramas de componentes 104

1. Interfaces 105
2. Componentes 105
3. Puertos 107
4. Dependencias 108
5. Caso de estudio: Ajedrez 109
6. Caso de estudio: Mercurial 110
7. Caso de estudio: Servicio de cifrado remoto 111

3. diagramas de despliegue 112

1. Nodos 112

2. Artefactos 114
3. Caso de estudio: Ajedrez 114
4. Caso de estudio: Mercurial 114
5. Caso de estudio: Servicio de cifrado remoto 115

4. diagramas de paquetes 116

1. Paquetes 116
2. Generalización 118
3. Relaciones de dependencia 118
4. Caso de estudio: Ajedrez 119
5. Caso de estudio: Mercurial 120
6. Caso de estudio: Servicio de cifrado remoto 121

6. diagramas de clases 123

1. clases 123
2. asociaciones 128

1. Multiplicidad 131
2. Agregación y composición 132

3. herencia 133
4. Interfaces 135
5. dependencias 136
6. excepciones 137
7. clases parametrizables 138
8. ejemplo de modelado: “spindizzy” 139
9. caso de estudio: ajedrez 142
10. caso de estudio: mercurial 144

11. caso de estudio: servicio de cifrado remoto 146

1. Lado cliente 146
2. Lado servidor 148

7. diagramas de secuencias 151

1. conceptos preliminares 151
2. estructura básica 152

1. Línea de vida 153
2. Activación 154
3. Mensajes síncronos y asíncronos 154
4. Creación, destrucción e invocación recursiva 156

3. ejemplo de modelado: "servidor ftp" 157

1. Un solo cliente 157
2. Dos clientes 158

4. fragmentos combinados 159

1. Saltos condicionales 159
2. Iteraciones 161
3. Paralelismo 163

5. Parametrización 164
6. caso de estudio: ajedrez 165

1. Turno del jugador humano 165
2. Turno de la IA 167

7. caso de estudio: mercurial 169
8. caso de estudiO: servicio de cifrado remoto 171

8. diagramas de comunicación 173

1. estructura básica 174
2. saltos condicionales 176
3. iteraciones 177
4. caso de estudio: ajedrez 178
5. caso de estudio: mercurial 179
6. caso de estudio: servicio de cifrado remoto 180

9. patrones de diseño 181

1. ¿por qué patrones? 182
2. tipos de patrones 183
3. patrones arquitectónicos 183

    1. Sistemas genéricos 183
    2. Sistemas distribuidos 185
    3. Sistemas interactivos 187

4. patrones de diseño GoF 187

    1. Descripción de un patrón de diseño 188
    2. Patrones de creación 189
    3. Patrones estructurales 195
    4. Patrones de comportamiento 197

5. caso de estudio: ajedrez 206

1. Patrón Facade 206
2. Patrón Observer 207
3. Patrón Strategy 208

6. caso de estudio: mercurial 209

1. Patrón Facade e Interpreter 209
2. Patrón Composite 210

7. caso de estudio: servicio de cifrado remoto 211

1. Lado cliente 211
2. Lado servidor 212

10. buenas prácticas de programación 215

1. patrones de diseño grasp 215

1. ¿Qué es una responsabilidad? 215
2. Experto en información 216
3. Creador 217
4. Bajo acoplamiento 218
5. Alta cohesión 219
6. Controlador 219
7. Polimorfismo 220
8. Fabricación Pura 221
9. Indirección 221
10. Variaciones Protegidas (VP) 222

2. caso de estudio: ajedrez 224

3. caso de estudio: mercurial 226
4. caso de estudio: servicio de cifrado remoto 228

1. Lado cliente 228
2. Lado servidor 230

11. diagramas de estado 233

1. conceptos básicos 234
2. estructura de un estado 235
3. estructura de las transiciones 235
4. tipos de nodos 236

1. Nodos inicial y final 236
2. Nodos de interconexión 237
3. Nodos condicionales 238

5. eventos 240

1. Eventos de llamada 240
2. Eventos de tiempo 241

6. estados compuestos 242

1. Estados compuestos simples 242
2. Estados compuestos ortogonales 243

7. sincronización de submáquinas 246
8. simplificación del diagrama de estados 247
9. historial 248

1. Historia superficial 248
2. Historia profunda 249

10. caso de estudio: ajedrez 251
11. caso de estudio: mercurial 253
12. caso de estudio: servicio de cifrado remoto 254

12. diagramas de actividad 255

1. estructura básica 255
2. estructuras de control 258

1. Nodos de decisión 258
2. Nodos de concurrencia y paralelismo 260

3. eventos de tiempo 261
4. nodos objeto 262
5. nodos de datos 262
6. particiones 263
7. parametrización 264
8. regiones 265

1. Regiones de expansión 265
2. Regiones interrumpibles 267
3. Regiones “if” 268
4. Regiones “loop” 269

9. Manejo de excepciones 270
10. conectores 271
11. señales y eventos 271
12. múltiples flujos 273

13. streaming 274
14. multicasting 274
15. caso de estudio: ajedrez 275
16. caso de estudio: mercurial 276
17. caso de estudio: servicio de cifrado remoto 278

13. Diagramas de estructura compuesta 281

1. estructura básica 282

1. Puertos 284

2. colaboraciones 286
3. uso de la colaboración 287
4. caso de estudio: ajedrez 288

1. Diagrama de estructura compuesta 288
2. Colaboración 289
3. Uso de la colaboración 290

5. caso de estudio: mercurial 290

1. Diagrama de estructura compuesta 290
2. Colaboración 291
3. Uso de la colaboración 291

6. caso de estudio: servicio de cifrado remoto (cliente) 292

1. Diagrama de estructura compuesta 292
2. Colaboración 292
3. Uso de la colaboración 293

7. caso de estudio: servicio de cifrado remoto (servidor) 294

1. Diagrama de estructura compuesta 294
2. Colaboración 295
3. Uso de la colaboración 295

14. OCL (Object constraint language) 297

1. estructura básica 297
2. tipos y operadores 298
3. modelo de referencia: "academia" 300
4. expresiones de restricción 301

1. Expresión inv: 301
2. Expresión pre 302
3. Expresión post 302
4. Operador @pre 302

5. expresiones de definición 303

1. Expresión body 303
2. Expresión init 304
3. Expresión def 304
4. Expresión let 305
5. Expresión derive 305

6. Colecciones 305

1. Nociones básicas 305
2. Operaciones básicas 306

7. navegación 317
8. ocl en los diagramas de estado 318
9. caso de estudio: ajedrez 319
10. caso de estudio: mercurial 320
11. caso de estudio: servicio de cifrado Remoto 321

15. ingeniería directa en java 323

1. diagramas de clases 324

1. Representación de una clase 324
2. Asociaciones 325
3. Herencia 326
4. Agregación 327
5. Composición 328
6. Clases abstractas 330
7. Clases internas 331
8. Clases asociación 332
9. Asociación calificada 336

2. diagramas de secuencias 337

1. Interacción básica 337
2. Creación, destrucción, automensajes y recursividad 338
3. Saltos condicionales 339
4. Iteraciones 341

3. diagramas de estado 343
4. caso de estudio: mercurial 349

16. ingeniería directa en c++ 359

1. diagramas de clases 360

1. Representación de una clase 360
2. Asociaciones 361
3. Herencia simple 363
4. Herencia múltiple 364
5. Agregación 364
6. Composición 365
7. Clases abstractas e interfaces 368
8. Clases internas o anidadas 371
9. Clases asociación 371
10. Asociación calificada 377

2. diagramas de secuencias 379

1. Interacción básica 379
2. Creación, destrucción, automensajes y recursividad 379
3. Saltos condicionales 382
4. Iteraciones 384

3. diagramas de estado 385
4. caso de estudio: ajedrez 394

17. ingeniería directa en python 411

1. diagramas de clases 412

1. Representación de una clase 412
2. Asociaciones 413
3. Herencia simple 414
4. Herencia múltiple 414

5. Implementación de interface 415
6. Agregación 415
7. Composición 416
8. Clases abstractas 418
9. Clases internas o anidadas 419
10. Clases asociación 419
11. Asociación calificada 422

2. diagramas de secuencias 423

1. Interacción básica 423
2. Creación, destrucción, automensajes y recursividad 423
3. Saltos condicionales 425
4. Iteraciones 426

3. diagramas de estado 428
4. caso de estudio: servicio de cifrado remoto 431

1. Lado cliente 432
2. Lado servidor 438

anexo a. programación orientada a objetos 445

A.1 breve reseña histórica 445
A.2 características de la poo 446
A.3 clases y objetos 447
A.4 programación orientada a objetos en c++ 447
A.4.1 Clases y objetos 447
A.4.2 Herencia 450
A.4.3 Polimorfismo 453

A.5 Programación orientada a objetos en java 459
A.5.1 Clases y objetos 459
A.5.2 Paquetes 461
A.5.3 Herencia 462
A.5.4 Interfaces 464
A.5.5 Polimorfismo 465
A.6 Programación orientada a objetos en python 469
A.6.1 Clases y objetos 469
A.6.2 Paquetes 471
A.6.3 Herencia 472
A.6.4 Interfaces 473
A.6.5 Polimorfismo 474
anexo b. resumen de notación uml 477
B.1 diagrama de casos de uso 477
B.2 diagrama de robustez 478
B.3 diagrama de componentes 478
B.4 diagrama de despliegue 478
B.5 diagrama de paquetes 479
B.6 diagrama de clases 479
B.7 diagrama de secuencias 480
B.8 diagrama de comunicación 481
B.9 diagrama de estados 481
B.10 diagrama de actividades 482
B.11 diagrama de estructura compuesta 483
bibliografía y referencias web 485
Referencias web 487
material adicional.....489

índice alfabéti-
co.....491

prólogo

¿A QUIÉN VA DIRIGIDO ESTE LIBRO?

La edición del ejemplar que tiene en sus manos ha sido fruto de cuatro años de trabajo y de investigación en el campo de la *Ingeniería del Software* y *del Lenguaje Unificado de Modelado* (UML). Como especialista en esta disciplina, y después de una larga trayectoria de desarrollos en diversos lenguajes de programación, he intentado aplicar todos mis conocimientos a esta obra, otorgándole cierto rigor académico y sobre todo un aspecto atractivo para el lector. Para ello me he servido de ejemplos basados en desarrollos multimedia, de sistemas y como no, de software de gestión. El presente libro no intenta ser una obra de referencia en el campo, sino más bien un libro de ayuda al estudiante y profesional de UML, con el firme propósito de transmitir la importancia que tiene la formación arquitectónica del software después de la algorítmica.

La lectura de los contenidos le será más amena si posee ciertos conocimientos a nivel de *Ingeniería Técnica* o *Ciclo Superior de FP en Informática*. De esta forma, las nociones de *Sistemas Operativos, Algorítmica, Redes, Bases de Datos, Autómatas* y *Programación Orientada a Objetos* serán de una valiosa utilidad para seguir las indicaciones dadas en los capítulos.

Finalmente, si usted posee conocimientos avanzados de los lenguajes Java, C++ y/o Python, le resultará más fácil el aprendizaje de los conceptos aquí explicados. En caso contrario le podrá ser de utilidad el anexo A y las referencias bibliográficas sobre el tema que podrá encontrar al final del volumen. Es por tanto importante que revise estos contenidos antes de comenzar a leer.

ESTRUCTURA DEL LIBRO

Los contenidos de la presente obra se han estructurado de acuerdo a un ciclo de vida secuencial de un desarrollo software. La sucesión de las explicaciones teóricas de los capítulos está acordes al avance de un proyecto real.

Con la idea de explicar la asociación existente entre UML y un desarrollo real, se presentan tres proyectos que van evolucionando al ritmo del ciclo de vida, es decir, desde la adquisición de requisitos hasta la implementación. Dichos ejemplos nos irán guiando en los diferentes diagramas UML del proyecto y sus fases de construcción. De esta forma el libro se ha estructurado en el siguiente orden metodológico:

- Parte I: Introducción teórica.
- Parte II: Análisis de requisitos.
- Parte III: Diseño arquitectónico.
- Parte IV: Diseño detallado.
- Parte V: Implementación.

Se ha intentado no descuidar los aspectos teóricos ni los prácticos, pretendiendo así conjugar a lo largo del libro ambos matices.

NOVEDADES EN LA SEGUNDA EDICIÓN

Se presenta aquí la segunda edición de la obra y, como es lógico en una nueva edición, se han corregido las erratas de la primera edición y se ha actualizado el contenido con las últimas tendencias en *Ingeniería de Software* y lenguajes de programación más demandados en la industria. Por esa razón, en el primer capítulo hallará una ampliación con el ciclo de vida incremental, metodologías de desarrollo software, calidad para la normalización/estandarización, así como teoría sobre el desarrollo de software seguro. También se ha incluido un nuevo patrón de diseño GOF y se ha tenido en consideración, por su importancia, incluir un nuevo capítulo dedicado a patrones GRASP con el fin de dar a conocer las técnicas que facilitan buenas prácticas de programación orientada a objetos actualmente.

La gran novedad de la presente edición recae en la incorporación de un nuevo caso de estudio para el abordaje de los conocimientos sobre análisis, diseño y programación mediante el lenguaje multiplataforma y multiparadigma Python, debido principalmente a su éxito en el mundo empresarial y de investigación.

Por todo ello, espero que esta obra cubra con completitud los conocimientos teóricos y prácticos que todo desarrollador aspira a alcanzar en aras de la construcción de software complejo que tanto necesitan las compañías y los investigadores de todo el mundo. Saber que este libro le ha servido para lograr el citado propósito sería una de mis grandes satisfacciones.

ESTILOS DE PROGRAMACIÓN

Como corresponde a cualquier proyecto informático basado en programación, un buen código fuente no es solo el que realiza sus funciones correctamente, sino también el que es legible y mantenible a lo largo del tiempo. Por este motivo, en el libro se ha intentado seguir ciertos criterios estilísticos en relación a cada lenguaje. Así para el código Java se ha utilizado las guías de estilo oficiales de *Sun/Oracle* publicadas en su sitio Web, mientras que para C++ el estándar elegido ha sido el *Joint Strike Fighter - Air Vehicle - C++ Coding Standards* recomendado por *Bjarne Stroustrup*. Respecto a Python, se ha aplicado el estilo de codificación indicado en su página de referencia: www.python.org.

DIAGRAMAS Y PROGRAMAS DE EJEMPLO

Para diseñar los diagramas de ejemplo del libro se han utilizado las aplicaciones *StarUML* y *MagicDraw* 15.0. La primera de ellas es una aplicación gratuita escrita inicialmente en Object-Pascal (*Borland Delphi*) y posteriormente en Java/Eclipse que puede ser descargada en la siguiente URL: http://staruml.sourceforge.net. La segunda es una aplicación comercial, pero puede descargarse una versión de prueba en: http://www.nomagic.com con el fin de evaluar los proyectos propuestos en el libro.

Las aplicaciones y ejemplos de Java aquí descritos han sido compilados y ejecutados directamente desde la línea de comandos en Windows, mientras que los ejemplos de C++ han sido programados sobre Visual C++ 2017. Si está interesado en la última actualización, puede descargarse una versión gratuita desde la Web de Microsoft en https://visualstudio.microsoft.com/es/downloads/ En cuanto a los proyectos realizados en Python, usted podrá descargar el intérprete del lenguaje desde su sitio web: https://www.python.org/downloads/

Finalmente, si desea descargar tanto los diagramas UML como los códigos fuente debe acceder a la web del propio libro en: www.ra-ma.com

Le recomiendo que lea primeramente el fichero *leame.txt* que se encuentra en el directorio con las indicaciones. Este fichero le guiará en la instalación de los ejemplos y le informará de la estructura de los archivos.

FEEDBACK

Cualquier crítica constructiva será siempre bienvenida. Éstas me servirán para mejorar mi trabajo de cara a una futura edición; por lo que cualquier errata, error, inconsistencia o fallo de presentación serán bien recibidas por mi parte.

Para cualquier consulta o informe de fallos puede contactarme por e-mail: cjimeneztau@gmail.com

AGRADECIMIENTOS ACADÉMICOS

Quisiera expresar mi agradecimiento al profesor *Dr. Manuel Arias Calleja* por su amabilidad al revisar y dirigir algunos contenidos vía e-mail y telefónica desde Madrid; al *Departamento de Ingeniería de Software y Sistemas Informáticos* de la UNED por los conocimientos transmitidos en su Máster en Investigación y al profesor *Dr. Jesús García Molina*, de la *Universidad de Murcia*, por su segunda revisión y sabios consejos.

Murcia

El autor

Enero de 2021.

I

UML en el contexto de la ingeniería del software

«Siempre imaginé que el Paraíso sería algún tipo de biblioteca».

(Jorge Luis Borges)

Comenzamos la explicación de los conceptos relacionados con el mundo de la construcción del software haciendo una breve síntesis de la historia y evolución de las técnicas de modelado y cómo ha sido posible la llegada de UML (*Unified Modeling Language*) al mundo de la IS (*Ingeniería del Software*). En este capítulo también se hará una reseña a los diferentes diagramas que se tratarán a lo largo del libro y la relación que tienen con los diferentes ciclos de vida del software. Para finalizar se introducirá la última tendencia en el desarrollo automático de software mediante modelado con las técnicas de MDA (*Model Driven Architecture*).

la necesidad de un modelo unificado

UML comienza a gestarse cuando la empresa de software *Rational,* fundada por *Grady Booch,* contrata a *James Rumbaugh* en 1994 que por entonces trabajaba en la *General Electric*. UML nació como la fusión de la metodología OMT (*Object-Modeling Technique*) de *Rumbaugh* y el método de *Grady Booch*. Al poco tiempo (1995) se unió a Rational *Ivar Jacobson,* inventor del método OOSE (*Object-Oriented Software Engineering*) y de algunos conceptos de otros lenguajes de modelado. Al grupo de los tres inventores se le llamó familiarmente como los "*Tres Amigos*" a causa de sus frecuentes debates sobre UML. En enero de 1997 fue propuesto el primer borrador de UML 1.0 a la OMG[1] (*Object-Management Group*) a través de un consorcio llamado *UML Partners*. Más tarde, en noviembre de 1997, y después de la creación de la *Semantic Task Force* por *UML Partners* para estandarizar UML y dotarla de contenido semántico, UML 1.1 fue adoptada por la OMG. Desde la versión 1.1 hasta hoy han existido varias revisiones menores de la especificación como la 1.2, 1.3 y 1.4. La versión 2.5.1 de UML, lanzada en diciembre de 2017, es la última a fecha de escritura de la presente edición.

Actualmente UML goza de un reconocido prestigio en el campo de la IS, aplicándose de forma extensa en una gran variedad de campos del desarrollo de software (incluidos la Inteligencia Artificial) y con un éxito sin precedentes en los proyectos de software. La versión de UML utilizada en este libro es la 2.x y data de julio 2005.

¿por qué modelar?

Un modelo es una abstracción de un problema de la realidad. Con esta idea surge el concepto de modelar, que consiste en abstraer las características esenciales de un problema real a una representación útil para un propósito determinado. En el caso de la ingeniería convencional como la aeronáutica o la mecánica, los ingenieros construyen modelos para asegurarse que el producto final funcionará. La implicación directa del modelo es que es posible su validación y comprobación, por lo que no tiene sentido construir modelos para luego no realizar pruebas. Obviamente los modelos son más baratos que los productos acabados y además permiten verificar su correcto funcionamiento.

En el mundo del software, el ingeniero construye modelos para probar si sus proyectos tendrán éxito y para comunicar a otros ingenieros y programadores las ideas de lo que intenta modelar. En el campo de la IS predomina el uso de UML y no sería lógico utilizar otra técnica para organizar la estructura estática o dinámica de una aplicación, ya que el lenguaje de modelado UML es bastante maduro para este fin. Con el uso de UML el ingeniero puede crear un modelo más factible e inteligible que el código fuente complejo, intercambiable entre expertos y que permite probar fácilmente la aplicación. Las pruebas desde UML son un campo todavía en experimentación al que se dirige la tecnología MDA (*Model Driven Architecture*) y que aún no es fidedigna del todo. Actualmente ya es posible la conversión directa de un modelo UML a una aplicación ejecutable y distribuible, aunque cuenta con la desventaja del excesivo coste económico de estas herramientas.

Es siempre más factible dedicar un tiempo prudencial a analizar y diseñar la aplicación para comprobar que funciona, ya que el hecho de realizar esta acción nos asegura evitar muchos errores de diseño y la construcción de programas erróneos. En este sentido, nuestro trabajo siempre podrá ser

entendido, discutido y comunicado con otros colegas de otras compañías o nacionalidades de una manera estándar, lo que implica también una forma de documentar productos de software para terceros desarrolladores.

En resumen, UML no se utilizará para probar si nuestro programa funcionará correctamente, sino para discutir con otros, documentar, aplicar la ingeniería directa o simplemente representar una idea. Por último, UML no es adecuado para sistemas en tiempo real, para tales sistemas existen notaciones específicas que no están dentro del ámbito de este libro.

Modelos de proceso software

De igual forma que sucede en otras actividades de ingeniería, en la que existe un ciclo de vida desde que se detecta la necesidad de construir el producto hasta que está en uso, la IS también requiere de tiempo y esfuerzo para su desarrollo y debe permanecer en uso durante un tiempo mucho mayor. Por este motivo se requiere de un ciclo de construcción y mantenimiento del software que se repita de forma secuencial o de otras maneras y permita el establecimiento de fases de desarrollo. Al conjunto de fases delimitadas con sus entradas y salidas se les denomina comúnmente "ciclo de vida" y permiten la elaboración de software de una manera metódica y ordenada.

Las fases principales son comunes en todos los ciclos de vida y se repiten de manera secuencial, incremental o en espiral. Las fases principales en cualquier ciclo de vida son:

1. *Análisis*: Es el proceso de reunión de requisitos para construir el software. El analista del software debe comprender el dominio de información del software y construir el modelo de análisis.
2. *Diseño*: Esta fase se compone de muchos pasos en los que se encuentran la deducción de estructuras de datos, la arquitectura del software, la interfaz de usuario y el diseño algorítmico. Algunos autores dividen esta etapa en el *diseño global o arquitectónico*y *diseño detallado*. El primero consiste en transformar los requisitos en una arquitectura de alto nivel, definir las pruebas, generar la documentación y planificar la integración. El diseño detallado consiste en refinar el diseño para cada módulo, definiendo sus requisitos y la documentación.
3. *Codificación*: Se transforma el diseño en código ejecutable para la construcción del sistema. Las herramientas Computer-Aided

Software Engineering (CASE) actuales permiten la conversión de un modelo UML para generar una parte esquemática del código de la aplicación, aunque no toda completamente.

4. *Pruebas*: Después de generar el código se procede a probar el programa. El proceso de pruebas consiste en el análisis de los procesos lógicos internos del software (comprobando que las sentencias y las bifurcaciones se cumplen), o en el análisis de los procesos externos funcionales (se ejecuta el programa con una serie de entradas y condiciones para comprobar que se ejecuta correctamente). A las primeras se les denomina pruebas de "caja blanca", mientras que a las últimas se les denomina pruebas de "caja negra", por la opacidad de la visión del código fuente.
5. *Mantenimiento*: Se produce después de entregar el producto al cliente.Es la parte que más tiempo consume. Se debe comprobar que el software sigue funcionando correctamente, y que se adapta a los nuevos requisitos y condiciones externas. Por ejemplo, un cambio de sistema operativo o de un dispositivo o periférico.

Estas etapas o fases se dividen en tareas. La *documentación* es una tarea importante que se realiza en todas las fases y donde UML cumple un papel excelente para los desarrolladores.

Modelo en cascada

El ciclo de vida en cascada fue propuesto por *Winston W. Royce* en 1970 y posteriormente revisada por *Barry Boehm* en 1980 e *Ian Sommerville* en 1985. Se basa en una serie de etapas o fases que se ejecutan en el proyecto de forma iterativa. Es decir, partiendo del *análisis de requisitos* se transita por el resto de las fases secuencialmente hasta llegar al *mantenimiento*. Una característica de este ciclo de vida es que podemos volver hacia atrás en el ciclo si tuviéramos que realizar cambios en algunas de las etapas anteriores. Por ejemplo, si en la etapa de *pruebas* fuera necesario volver al *diseño*, tendríamos obligatoriamente que pasar de nuevo por la *codificación*.

Entre las ventajas de este modelo destacan la facilidad de utilización y la existencia de una gran cantidad de herramientas CASE que lo soportan. Entre las desventajas se incluye la exigencia de tener todos los requisitos al comienzo del proyecto y que no genera ningún producto hasta que se hayan finalizado todas las fases (ciclo). En general, este tipo de modelo se utilizará en proyectos de corta duración y fáciles que no requieran de cambios frecuentes en los requisitos.

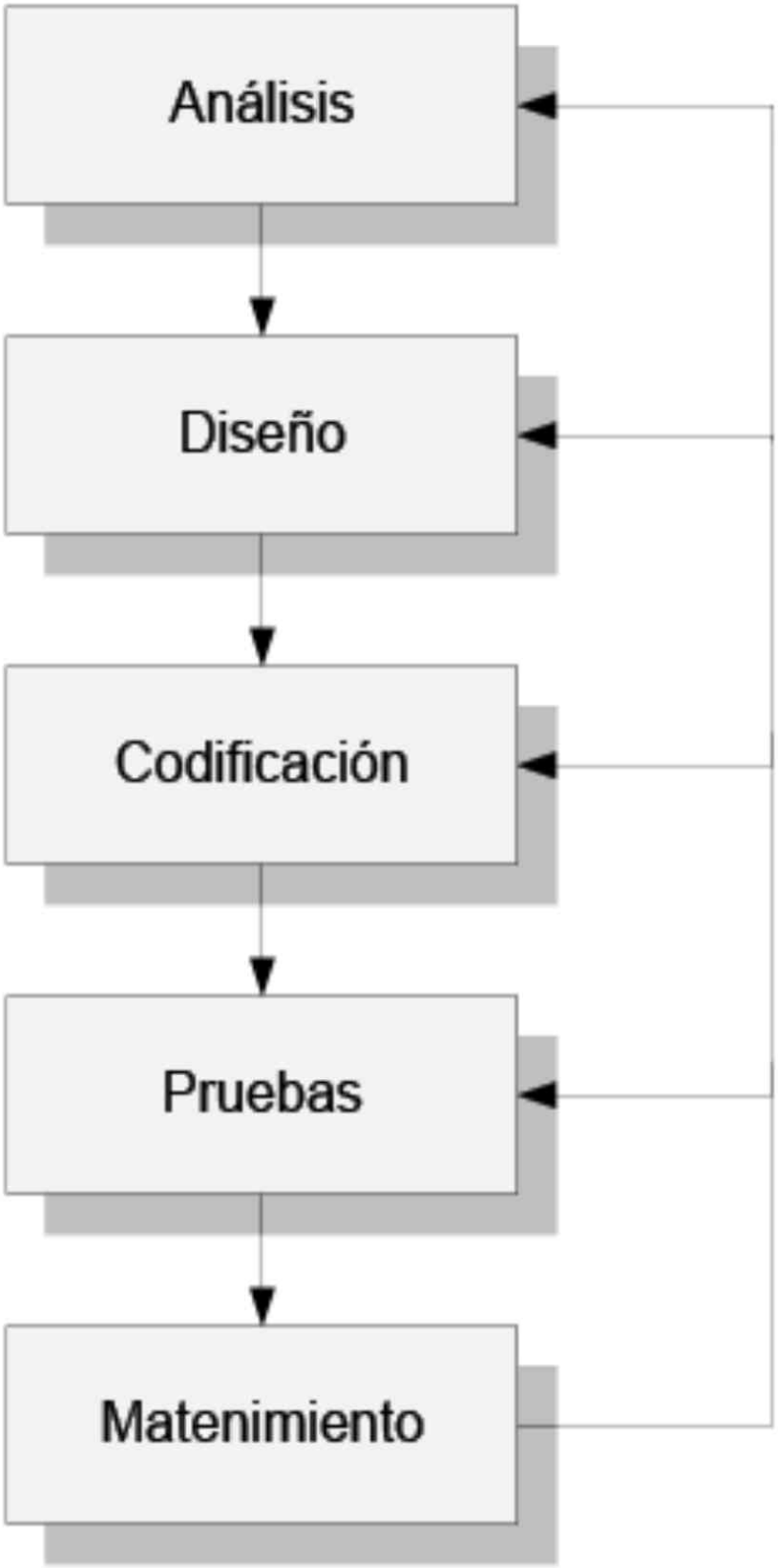

Análisis
Diseño
Codificación
Pruebas
Matenimiento

Figura 1.1. Ciclo de vida en cascada

Modelo incremental

Según [Pressman02], el modelo incremental corresponde a la gama de modelos evolutivos de proceso de software, ya que combina las ventajas del modelo en cascada con la filosofía de construcción de prototipos. Como veremos a continuación, la metodología RUP (Rational Unified Process) utilizará un esquema de ciclo de vida muy similar. En la figura 1.2 se muestra el diagrama de vida del citado modelo donde se puede apreciar los diferentes incrementos que producirán versiones incompletas del producto basadas en los incrementos anteriores, hasta llegar a una versión completa y estable. Cada incremento permite al usuario evaluar sobre un producto parcial y operacional las funcionalidades requeridas, de forma que facilite el desarrollo progresivo en caso de insuficiencia de personal con vistas a una fecha límite de entrega estricta. Una de las ventajas que se adquiere utilizando este método es la no necesidad de tener un conjunto completo de requisitos al principio del proyecto; no obstante, la detección de errores en este modelo de proceso se produce tarde [Arias07].

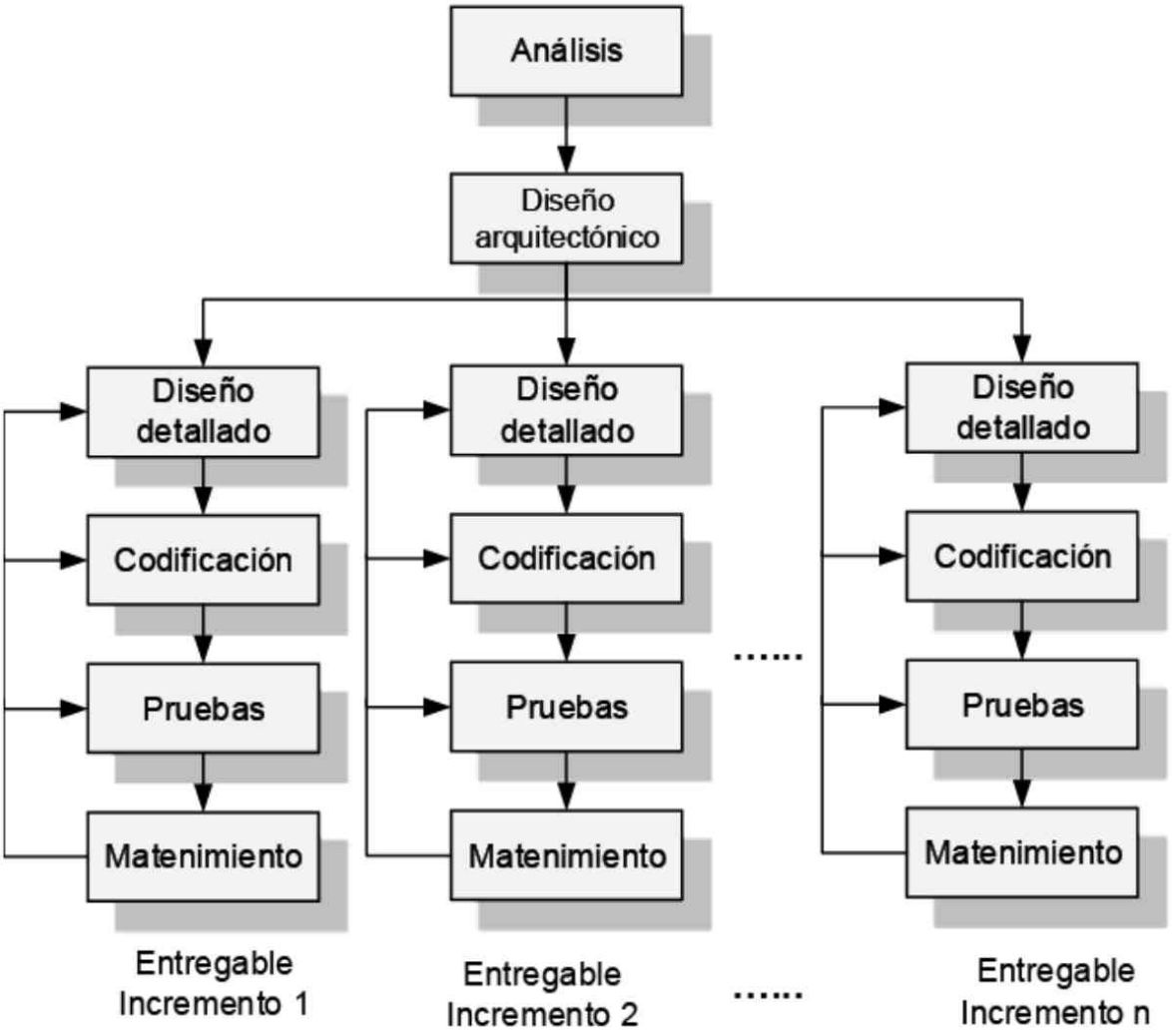


Figura 1.2. Ciclo de vida incremental

Modelo en espiral

Este ciclo de vida fue propuesto inicialmente por *Barry Boehm* en 1986 y consiste en una serie de ciclos con hitos que se repiten un número determinado de veces propiciando una evolución del producto hasta su conclusión final. Una característica importante de este ciclo de vida es la introducción del análisis del riesgo en cada una de las evoluciones, intentando optar por el más asumible.

Entre las ventajas de este modelo se incluye la capacidad para reducir riesgos, mejorar la calidad y la no necesidad de tener todos los requisitos al principio. Entre las desventajas se incluye la dificultad de adopción de riesgos y su coste. Sin embargo, debido a su naturaleza evolutiva y recurrente permite ser utilizado en proyectos complejos, críticos y de larga duración.

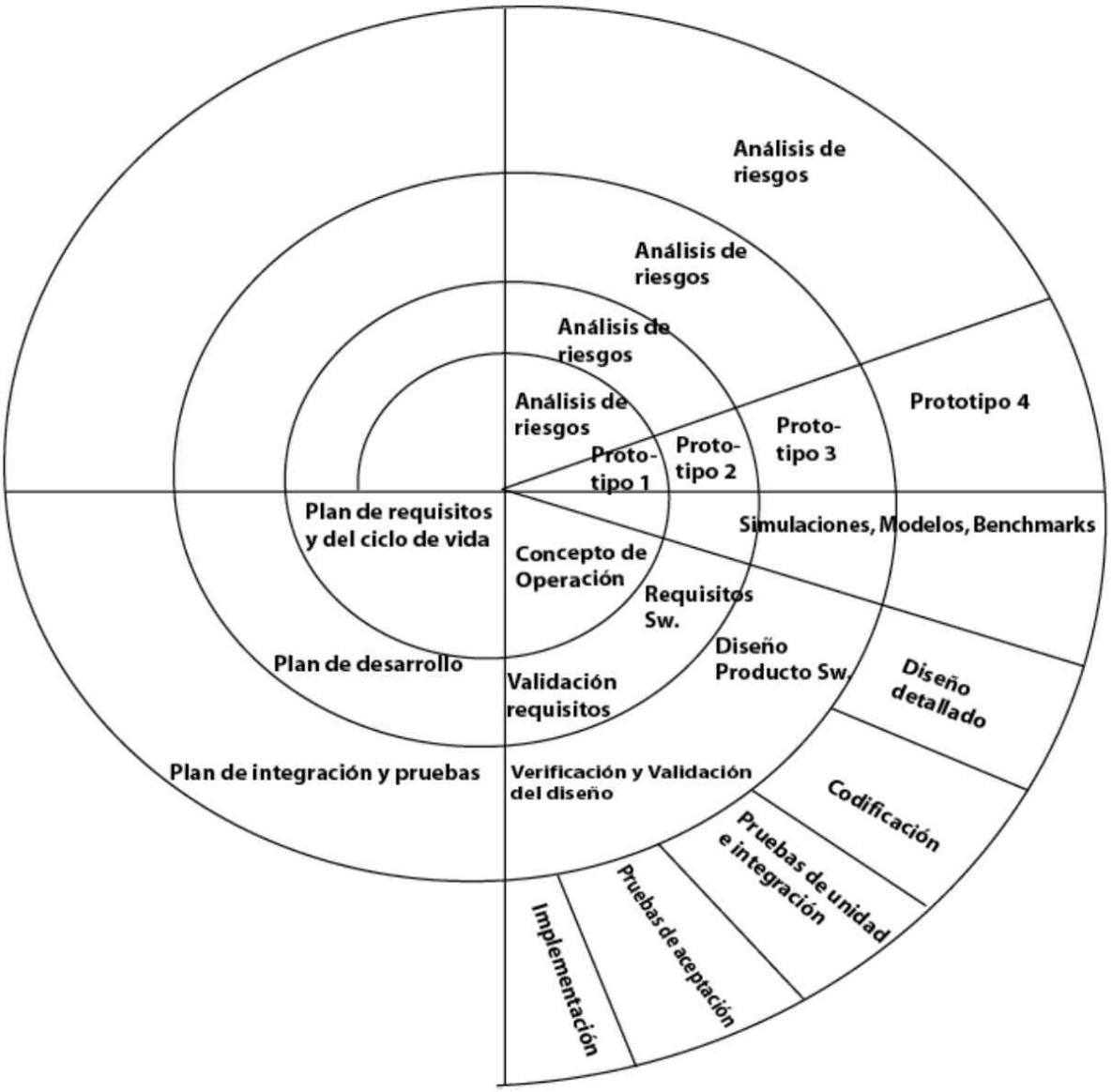


Figura 1.3. Ciclo de vida en espiral

Relación de las fases con los diagramas UML

En lo que respecta a los diferentes tipos de diagramas relacionados con el contenido del presente libro, se han estudiado aquellos que están involucrados en la parte de construcción de alto nivel donde se aplica la visión del ingeniero de software, es decir, los diagramas de representación de requisitos ya elicitados, análisis, diseño arquitectónico y diseño detallado. Cada diagrama se usa en una o más fases específicas del ciclo de vida.

Para comprender mejor esta relación se expone a continuación el siguiente esquema donde se desglosan los diferentes tipos de diagramas en relación a su fase dentro del ciclo de vida:

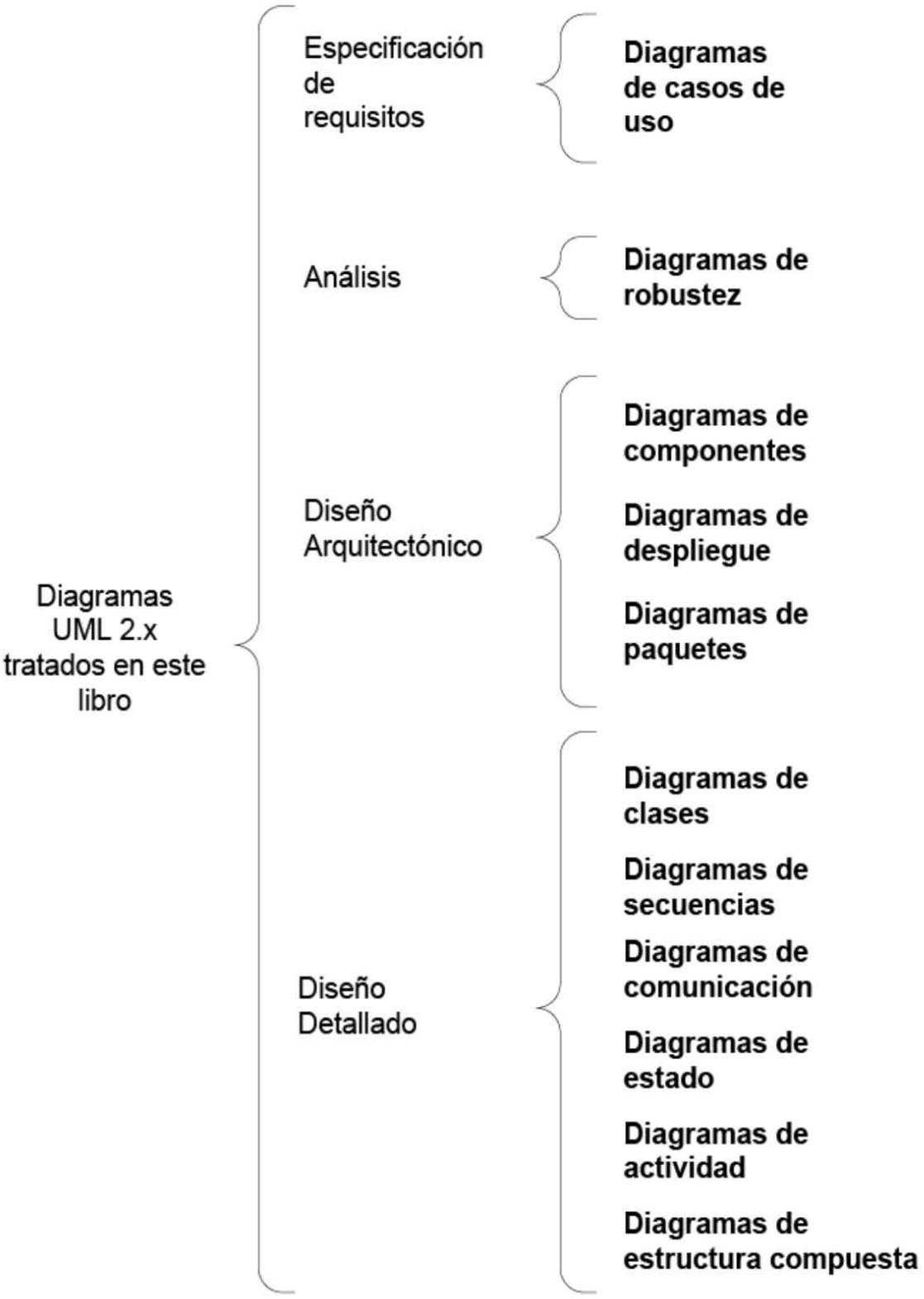


Figura 1.4. Esquema de los diferentes diagramas UML

En lo que respecta a la fase de Especificación de requisitos, nos encontramos

con el *Diagrama de casos de uso*, que permiten representar el comportamiento del sistema desde el punto de vista del usuario y cómo éste interactúa con el sistema para llevar a cabo los requerimientos de la aplicación. En la fase de Análisis se hallan los *Diagramas de robustez* con el fin de analizar los pasos de un caso de uso para validar la lógica de negocio, en otras palabras –permiten comprobar la robustez de un caso de uso y su conformidad con los requerimiento del sistema en construcción–.

Dentro de la fase de *Diseño arquitectónico* hallamos los *Diagramas de componentes* que son los responsables de modelar la visión estática del sistema en el nivel de implementación. Muestra la organización y las dependencias entre componentes software: librerías, *frameworks*, interfaces de usuario, ficheros de cabecera, etc. También, y dentro de esta fase, se sitúa el *Diagrama de despliegue* con la idea de representar la organización del hardware y los principales artefactos de nuestro sistema. En el *Diagrama de paquetes* se muestra la división del sistema en unidades lógicas y muestra la dependencia entre ellas. Normalmente el paquete está representado como un directorio y es de gran utilidad especialmente en la organización de la aplicación tanto en C++ como en Java y Python[2].

En la fase de *Diseño detallado* es donde se ubica el *Diagrama de clases* que permite representar estáticamente el sistema, indicando la relación entre las diferentes clases, su estructura jerárquica y cómo se organizan. Seguidamente estarán los *Diagramas de interacción* que permiten visualizar detalladamente cómo los diferentes objetos involucrados en el sistema interactúan para ejecutar una tarea. La aplicación principal de los diagramas de interacción es mostrar el funcionamiento de un caso de uso y los pasos para completarlo. Por ello, en los *Diagramas de secuencias* y *comunicación* se modelan las interacciones entre los objetos una vez han sido instanciados en el sistema.

Los *Diagramas de estados* son los que representan la vista dinámica del sistema en cuanto que modelan la evolución de un sistema a lo largo del tiempo. En UML representan el comportamiento de las instancias de las clases, componentes o casos de uso. De forma similar se encuentran los *Diagramas de actividad*, que están basados en las *redes de Petri* y modelan fundamentalmente el paralelismo y el comportamiento del sistema haciendo énfasis en la participación de los objetos en ese comportamiento. Finalmente, los *Diagramas de estructura compuesta* permiten representar una visión estructural de los principales clasificadores utilizados en el modelo estático del sistema.

Como hemos podido observar, estos diagramas que se han explicado brevemente son de importancia capital en cualquier proyecto software basado en alguna metodología. A lo largo de los siguientes capítulos trataremos cada uno de ellos de forma más pausada.

metodologías de desarrollo

Por *metodología de desarrollo de software* se entiende como el conjunto de roles o personal implicado en las fases de construcción del producto, los artefactos o recursos y materiales utilizados en el desarrollo del mismo y el ciclo de vida concreto utilizado. En esta sección se expondrán las dos formas de abordar una metodología de desarrollo actualmente, la cual puede ser ligera (ágil) o pesada.

Metodologías ágiles

De acuerdo a [Pressman14], una metodología ágil promueve la satisfacción del cliente como resultado de las prontas entregas incrementales de software gracias a equipos humanos muy motivados que aplican un trabajo ingeniería de software simple y ligero. No obstante, no todas las metodologías ágiles pueden ser aplicadas a todos los tipos de proyectos y escenarios. El auge y éxito de las metodologías ágiles actualmente es uno de los factores importantes a tener en cuenta en el momento de crear una empresa de desarrollo de software.

Por *ágiles* entenderemos metodologías *ligeras* que son fácilmente adaptables a la dinámica de los negocios y a los cambios, fomentando la comunicación entre clientes y desarrolladores por medio de grupos de trabajo con dotes comunicativas y bien motivados. La experiencia de décadas de desarrollo confirma que el coste de utilizar una metodología tradicional evoluciona casi exponencialmente con el tiempo; mientras que una metodología ágil lo hace "idealmente" de forma logarítmica. Por ello, las claves de una metodología ágil residen en ser *adaptables* a los imprevistos relacionados con los requerimientos del cliente y a los imprevistos del análisis, diseño, la implementación y pruebas. Por ello, un método ágil debe adaptarse incrementalmente a estas contingencias por medio de equipos que mantengan una relación constante con el cliente y realicen entregas operativas de prototipos en periodos cortos de tiempo.

extreme programming (xp)

Extreme Programming (XP) es un conjunto de principios de desarrollo descrito por *Kent Beck* en su libro sobre esta materia [Beck99]; por tanto, no es exactamente una metodología, sino un conjunto de principios que derivan en una serie de prácticas. En la actualidad ha perdido importancia, aunque suele tener presencia frente a metodologías clásicas debido a su simplicidad e

interacción con el cliente. Según [Pressman14], [Arias07] las principales actividades que lo componen (ver figura 1.5) son:

1. **Planificación**: Se crean las *historias de usuario* que son unos documentos cortos escritos por los usuarios pero sin vocabulario técnico. Cada historia especifica un requisito del sistema y debe durar "idealmente" entre una y tres semanas, conociendo de antemano que nuevas historias pueden ser escritas en cualquier momento. El equipo XP colabora conjuntamente con los clientes ordenando las diferentes historias en categorías dependiendo de su urgencia y riesgo. El cliente puede añadir historias, modificarlas, dividirlas o eliminarlas definitivamente a su gusto. Una desventaja de este modelo es que hay que tener disponible al cliente desde el comienzo al fin del desarrollo.
2. **Diseño**: Siguiendo el principio de ser lo más simple posible, se realizará el diseño evitando aproximaciones demasiado complejas. En esta actividad se utilizarán las tarjetas CRC (Clase, Responsabilidades, Colaboraciones)[3] y se producirá un prototipo operacional correspondiente al incremento del diseño. Además, XP se basa en la utilización de *refactorización* ("refactoring") y otras técnicas adicionales que no se comentarán aquí y que permiten cambiar la estructura interna del código fuente sin afectar a la funcionalidad o comportamiento externo de la aplicación.
3. **Implementación**: Una vez que las historias de usuario se han obtenido y el diseño preliminar ha finalizado, el equipo de desarrollo prepara las *pruebas de unidad*. Una vez que las pruebas de unidad han sido creadas, el programador puede enfocar mejor su atención sobre lo que debe ser implementado para pasar las pruebas. Cuando

el código se ha implementado pueden realizarse las pruebas de unidad. Una de las características de XP es la programación por pares ("pair-programming"), ya que es una práctica recomendada, en la que un programador escribe código y el otro lo prueba y después se intercambian los papeles. Una vez que cada pareja ha finalizado, su trabajo puede ser integrado con el trabajo de otros equipos. XP garantiza una integración diaria por un equipo especializado. Una noción importante en XP es que el diseño y la implementación están a menudo entrelazados.

4. **Pruebas**: Las pruebas de unidad se automatizan y se aplican en esta fase, lo que facilita las *pruebas de regresión*. Las pruebas de validación y la integración son frecuentes durante esta actividad. Finalmente se realizan los *test de aceptación* que son realizados por los clientes sobre el sistema y se basan en las historias de usuario creadas al comienzo.

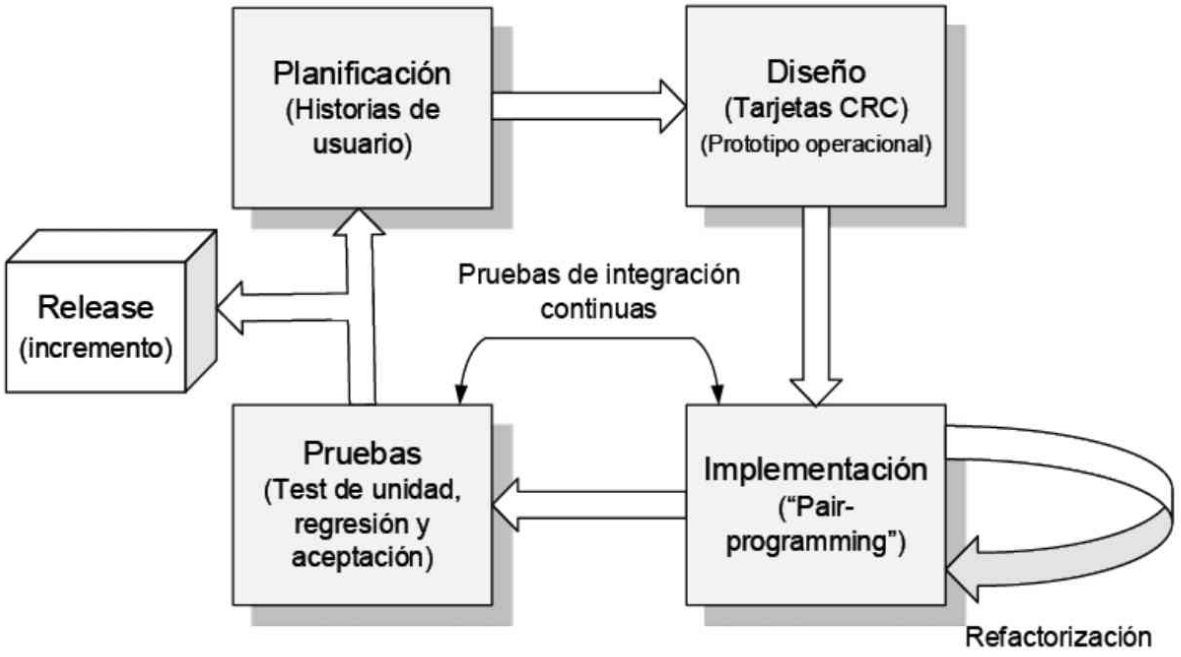


Figura 1.5. Metodología Extreme Programming (XP)

scrum

Scrum es otra metodología ágil utilizada ampliamente para productos complejos que comprende desde software y hardware hasta vehículos autónomos, escuelas y un amplio rango de organizaciones. Fue creada a comienzos de 1990 por Jeff Sutherland y Ken Schwaber [Sutherland17] con el propósito de definir un marco de trabajo para tratar con la complejidad. Scrum está compuesto por los siguientes elementos relacionados:

- Roles (*Roles*)
- Eventos (*Events*)
- Artefactos (*Artifacts*)
- Reglas (*Rules*), que relacionan a los anteriores elementos.

Equipo Scrum

El equipo scrum *(Scrum Team*) está compuesto por el Propietario del Producto (*Product Owner*), el Equipo de Desarrollo (*Development Team*) y un Scrum Master.

- **Propietario del Producto (*Product Owner*):** Es la persona encargada de gestionar la Pila del Producto (*Product Backlog*) de forma que realice acciones de ordenación de elementos para alcanzar los objetivos, optimizar el valor del trabajo que realiza el Equipo de Desarrollo. Adicionalmente, toda la organización debe respetar sus decisiones.
- **Equipo de Desarrollo (*Development Team*):** Se compone de profesionales autoorganizados y multifuncionales encargados de realizar un incremento del producto. Es importante la sinergia que resulte del equipo ya que ayuda a optimizar su eficiencia y efectividad. El tamaño ideal del Equipo de Desarrollo es de menos de tres

miembros, puesto que reduce la interacción y resulta en ganancias de productividad más pequeñas. No es aconsejable equipos de más de nueve miembros por requerir demasiada coordinación.

- **Scrum Master (*Scrum Master*):** Es el encargado de promocionar y apoyar la Guía de Scrum [Sutherland17]. El Scrum Master es un líder que está al servicio del Equipo Scrum, en concreto del Propietario del producto, del Equipo de Desarrollo y de la Organización.

Eventos Scrum

En Scrum existen diferentes tipos de eventos que se definen como períodos de tiempo con una duración máxima.

- **Sprint (*Sprint*):** Es el corazón de Scrum y consiste en un período de tiempo (time-box) de un mes de duración durante el cual se crea un incremento de producto "Terminado" (*Done*)[4], utilizable y potencialmente desplegable.
- **Planificación del Sprint (*Sprint Planning*):** Se decide el trabajo a realizar durante el Sprint (objetivos, elementos de la Pila del Producto, etc.) y tiene una duración máxima de ocho horas para un Sprint de un mes. El Sprint puede ser cancelado, aunque no es recomendable debido a que consume recursos al reagruparse estos en otra planificación de Sprint y por tanto suele ser traumático.
- **Scrum Diario (*Daily Scrum*):** Consiste en una reunión diaria con una duración de tiempo de quince minutos para el Equipo de Desarrollo donde se planea el trabajo de las siguientes veinticuatro horas, lo que optimiza la colaboración y el desempeño del equipo.
- **Revisión del Sprint (*Sprint Review*):** Es una reunión informal de no más de cuatro horas para un Sprint de un mes que se realiza a su

término, con el fin de revisar el incremento y adaptar la Pila del Producto, lo que facilita la retroalimentación de información y el fomento de la colaboración.

- **Retrospectiva del Sprint (*Sprint Retrospective*):** Se trata de una reunión de como máximo tres horas para un Sprint de un mes en el que se inspecciona el último Sprint en cuanto a personas, relaciones, procesos y herramientas. Se identifican y ordenan los elementos más importantes y se crea un plan para implementar las mejoras de forma que el Equipo Scrum desempeñe su trabajo.

Artefactos

Representan el trabajo en diversas formas útiles en favor de la transparencia y oportunidades para la inspección y adaptación.

- **Pila del Producto (*Product Backlog*):** Contiene todo lo conocido que podría ser necesario en el producto y los requisitos (enumera características, funcionalidades, mejoras y correcciones que constituyen cambios a realizar sobre entregas futuras del producto). Es un artefacto dinámico que crece y se adapta para que el producto sea competitivo y útil. Su existencia está supeditada al ciclo de vida del producto.
- **Pila del Sprint (*Sprint Backlog*):** Este artefacto incluye los elementos de la Pila del Producto utilizados en el Sprint, junto con un plan para entregar el Incremento de producto y conseguir el objetivo del Sprint. Es una predicción hecha por el Equipo de Desarrollo que contiene la funcionalidad del siguiente Incremento y el trabajo necesario para entregar dicha funcionalidad al finalizar el Incremento.
- **Incremento (*Increment*):** Es el conjunto de todos los elementos de la Pila del Producto realizados durante el Sprint y los Sprint anteriores y

supone un paso hacia una meta. Cuando se finaliza un Sprint este debe hallarse en el estado "Terminado" (*Done*) y en condiciones de utilizarse por si se decide liberarlo.

Metodologías pesadas

rational unified process (rup)

Según [Arias07] y [Jacobson00], el Proceso Unificado de Rational es una metodología orientada a objetos desarrollada por los mismos creadores de UML. Las características de RUP son las siguientes:

- **Dirigido por casos de uso:** Representan los requisitos funcionales del sistema desde el punto de vista del usuario y servirán como guía a RUP durante su ciclo de vida. Los casos de uso se explicarán en el Capítulo 2.
- **Ciclo de vida iterativo e incremental:** El proyecto se divide en mini-proyectos que representarán una iteración.
- **Estructura del ciclo de vida:** El proceso RUP consiste en una serie de ciclos, donde cada ciclo se compone de las siguientes fases: inicio, elaboración, construcción y transición que terminan con un hito. Al final de cada ciclo se tiene una versión del producto.

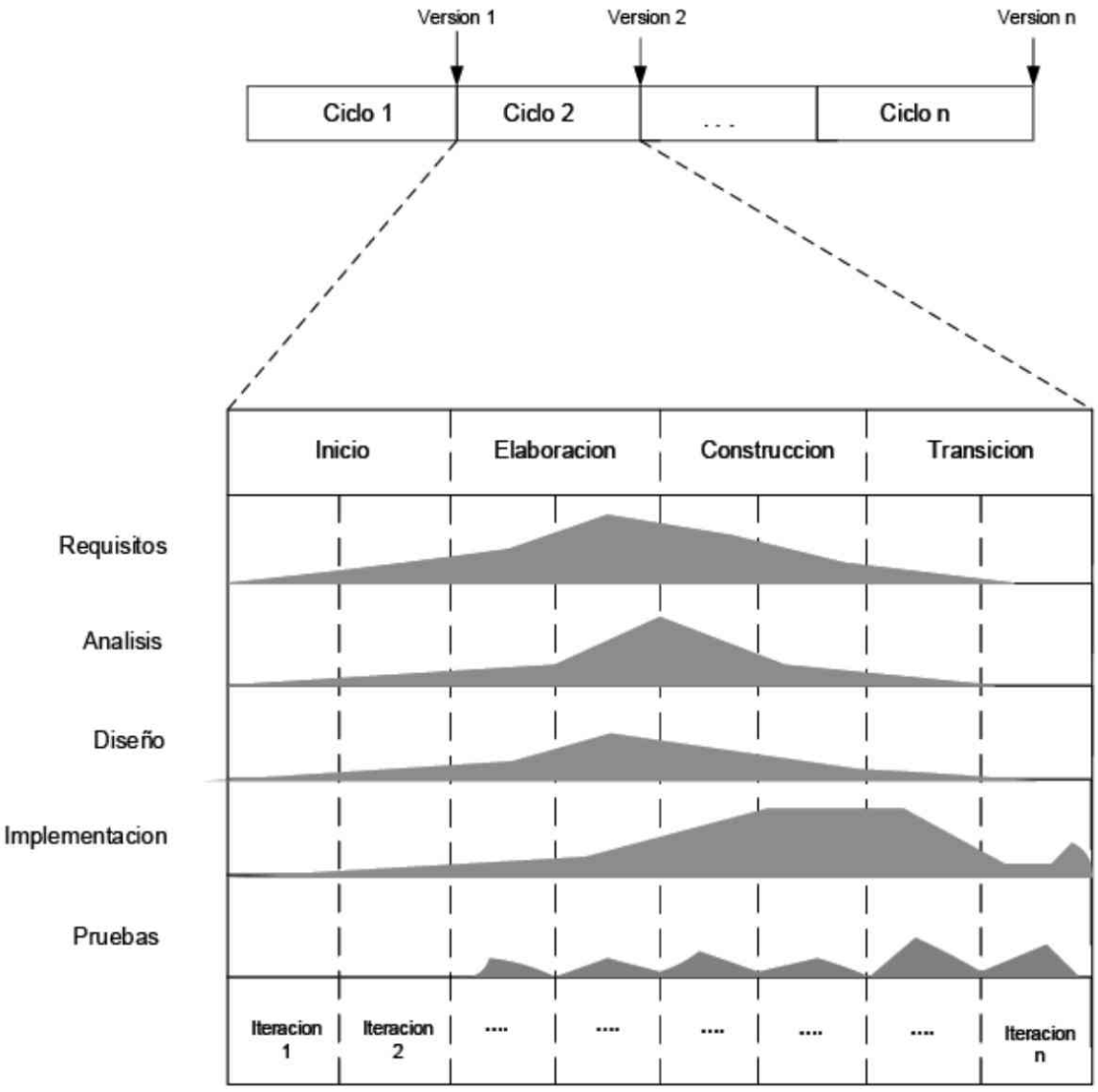


Figura 1.6. Metodología Rational Unified Process (RUP)[Arias07]

Según [Larman02] RUP organiza el trabajo de acuerdo a las cuatro fases anteriormente citadas:

- *Inicio*: Se da una visión aproximada, se realiza un análisis del negocio y se estudia el alcance.
- *Elaboración*: Se realiza una versión más refinada y una implementación iterativa del núcleo central de la arquitectura del sistema. Se identifican riesgos altos, alcance y requisitos.
- *Construcción*: Se implementan iterativamente el resto de los requisitos

fáciles y de menor riesgo. Se prepara el despliegue.

- *Transición*: Se liberan pruebas beta.

El producto terminado en RUP debe incluir el código ejecutable, requisitos, casos de uso, especificaciones no funcionales y casos de prueba.
RUP está basado en UML y es soportado por muchas herramientas CASE.
Además de las características enumeradas arriba, RUP construye en componentes conectados entre sí a través de interfaces, que darán lugar a una arquitectura en la cual se fundamenta esta metodología de software (Capítulo 5).

calidad del software

Definición y conceptos

El concepto de calidad siempre ha sido difícil de definir debido a su alto grado de subjetividad. Así, la calidad se convierte en un concepto totalmente relativo: lo que algo es de valor para unas personas puede no serlo para otras. Según el Diccionario de la Real Academia de la Lengua (DRAE) en su 23ª edición, el vocablo *calidad* tiene las siguientes acepciones: 1. f. Propiedad o conjunto de propiedades inherentes a algo, que permiten juzgar su valor. Esta tela es de buena calidad. 2. f. Buena calidad, superioridad o excelencia. La calidad de ese aceite ha conquistado los mercados. 3. f. Adecuación de un producto o servicio a las características especificadas. Control de la calidad de un producto. 4. f. Carácter, genio, índole. 5. f. Condición o requisito que se pone en un contrato. 6. f. Estado de una persona, naturaleza, edad y demás circunstancias y condiciones que se requieren para un cargo o dignidad. 7. f. Nobleza del linaje. 8. f. Importancia o gravedad de algo. 9. f. pl. Prendas personales. 10. f. pl. Condiciones que se ponen en algunos juegos de naipes.

En lo que respecta al domino del software, una definición ampliamente aceptada es la del estándar IEEE 610-1991: "*El grado con el que un sistema, componente o proceso cumple los requisitos especificados y cubre las necesidades o expectativas del cliente o usuario*". Obviamente, el concepto de calidad en el software difiere de cualquier otro producto de la industria, debido a su naturaleza abstracta e intangible, entre otros aspectos.

Dimensiones de la calidad

Aunque existen otras clasificaciones de las dimensiones de la calidad del software, como las de David Garvin [Gar87], aquí se presentarán las de Sebastián, Bargueño y Novo [Mingueto3]:

- **Prestaciones**: Se entenderá por prestaciones aquellas características operativas y principales de un producto que son fundamentalmente medibles, como por ejemplo: el tamaño en GB de un una unidad de estado sólido, el número de núcleos de un procesador, etc. La relación entre calidad y prestaciones puede variar dependiendo de las preferencias del usuario. Así, un usuario puede preferir la velocidad de procesamiento a la capacidad de almacenamiento de un equipo y viceversa.
- **Diferenciación**: Son las características secundarias del producto que le proporcionan un valor añadido, por ejemplo, un navegador GPS en un automóvil.
- **Fiabilidad**: Se define como la probabilidad de que un producto no falle en un determinado intervalo de tiempo, por ejemplo, software en tiempo-real tolerante a fallos en un quirófano. Es una medida objetiva.
- **Conformidad**: Se define como el grado en el que el diseño y las prestaciones del producto satisface los estándares establecidos, en general relacionados con la tasa de defectos, número de reclamaciones o reparaciones en período de garantía. También es una medida objetiva.
- **Duración**: Con duración nos referiremos a la estimación de la vida del producto antes de su deterioro sin posibilidad de reparación.
- **Asistencia técnica**: Es uno de los factores que más afecta a la calidad

actualmente y se refiere a la atención técnica y de reparaciones de una forma competente y rápida. Algunos aspectos pueden medirse objetivamente.

- **Estética**: Es la dimensión más subjetiva y está supeditada a los sentidos.

Crisis del software y necesidad de una medida (métricas)

A mediados de la década de los sesenta y a medida que los programas se volvían más grandes y más complejos, surgió la necesidad de migrar de las técnicas artesanales a un modo de trabajo disciplinado y ordenado. La magnitud del problema llegó a tal extremo que fue necesario convocar al Comité Científico de la OTAN donde expertos informáticos, basándose en las disciplinas de otras de ingenierías tradicionales, propusieron el concepto de *Ingeniería del Software*. Más tarde, en los años noventa, y a consecuencia de este fenómeno, se reconoció también la necesidad de una medida del software mediante el establecimiento de estimaciones formales, algorítmicas y matemáticas que pudieran, mediante pruebas empíricas, determinar la calidad del software. Así, surgen modelos de medición como COCOMO (COnstructive COst MOdel), SLIM (Software, LIfe Cycle Management) y los conocidos *Puntos Función*. Más tarde se incorporarán métricas para medir la productividad, la complejidad, la fiabilidad, los modelos de datos, la eficacia de los algoritmos criptográficos o la seguridad de la red.

Normalización y certificación

Las metodologías anteriormente explicadas y los modelos de proceso de gestión del software como CMM (Modelo de la Madurez de la Capacidad), la norma ISO 15504 y la metodología METRICA Versión 3 del Ministerio de Administraciones Públicas se consideran estrategias para mejorar la calidad del software [Mingueto3].

Además de los conceptos de metodología y modelo, surge el concepto de norma. Dichas normas son establecidas por organizaciones nacionales o internacionales que adoptan los citados modelos y metodologías. En muchas ocasiones las empresas deciden certificar su grado de cumplimiento respecto a una determinada normativa, como por ejemplo las normativas de la ISO (Organización Internacional para la Estandarización). Su homólogo nacional se denomina AENOR (Asociación Española de Normalización) que es el organismo responsable de inspeccionar las empresas y expedir el correspondiente certificado de calidad. En España, la certificación más extendida es la asociada a la norma de ISO 9001:2000 [Mingueto3] que especifica los requisitos para un Sistema de Gestión de la Calidad (SGC) en las empresas y asegurar la satisfacción del cliente de sus productos [Cuevas03].

La norma ISO/IEC 9126

La norma ISO/IEC 9126 nace como norma ISO de medida de calidad del software interna y externa en forma de descomposición jerárquica en árbol. Está basada en los modelos de calidad de McCall y Bohem, autores que sentaron un precedente en el campo de la calidad del software. A continuación se expone la tabla sinóptica de características y subcaracterísticas de la norma[5]:

Calidad del software (interna y externa)					
Funcionalidad	**Fiabilidad**	**Facilidad de uso**	**Eficiencia**	**Mantenimiento**	**Movilidad**
Idoneidad	Madurez	Fácil comprensión	Comportamiento frente al tiempo	Facilidad de análisis	Adaptabilidad
Exactitud	Tolerancia a fallos	Fácil aprendizaje	Uso de recursos	Capacidad para cambios	Facilidad de instalación
Interoperatividad	Capacidad de recuperación	Operatividad	Adherencia a normas	Estabilidad	Coexistencia
Seguridad	Adherencia a normas	Software atractivo		Facilidad para pruebas	Facilidad de reemplazo
Adherencia a normas		Adherencia a normas		Adherencia a normas	Adherencia a normas

Figura 1.7. La norma ISO/IEC 9126

Aseguramiento de la calidad del software (SQA)

El software se encuentra cada vez más presente en nuestras vidas y en muchos elementos de nuestro entorno. Por este motivo, la implicación en el riesgo del software propenso a errores es una cuestión estrechamente ligada a la calidad. Según [Cuevas03], "El objeto del SQA es proporcionar a la dirección una perspectiva adecuada del proceso que utiliza el proyecto software y de los productos a crear. El Aseguramiento de la Calidad Software implica la revisión y la inspección de los productos y las actividades del software, con objeto de verificar que se cumplen los procedimientos y estándares aplicables, y de proporcionar a los responsables del proyecto software y a otros directivos los resultados de dichas revisiones e inspecciones".

Por tanto, la misión del SQA es:

- Planificar las actividades de aseguramiento de la calidad.
- Verificar que los productos y el proceso software cumplen los requisitos, estándares y procedimientos establecidos.
- Informar a la dirección y a las personas relacionadas con el proyecto sobre los resultados del aseguramiento de la calidad.
- Elevar a la dirección los aspectos que no pueden solucionarse en el contexto del proyecto.

Las actividades que recaen sobre el grupo SQA son:

- Realizar el *plan*[6] SQA en las primeras fases del proyecto.
- Establecer las actividades del personal SQA respecto al plan SQA.
- Participar en la definición y revisión del plan del proyecto. El grupo SQA asesora en cuanto elaboración y revisión de planes, normas y procedimientos de desarrollo software.
- Revisar las actividades de desarrollo software con el fin de verificar su

cumplimiento.

- Inspeccionar los productos software con el propósito de verificar su corrección (antes de entregar el producto se evalúa en función a estándares, procedimientos y requisitos del contrato).
- Informar periódicamente al equipo de desarrollo.
- Documentar las desviaciones de las actividades y productos software.
- Revisar periódicamente las propias actividades del grupo SQA con el personal SQA del cliente.

Las actividades de gestión, aseguramiento y control de la calidad se realizan mediante técnicas para detectar defectos en el software principalmente. Las técnicas se dividen en *estáticas* o *dinámicas*. Las técnicas estáticas se realizan sin ejecutar el software, como las auditorías; mientras que las técnicas dinámicas se realizan mediante casos de prueba que suponen la ejecución del software real y el análisis manual o automatizado del código fuente. Estos casos de prueba pueden ser de *caja blanca* (cobertura de sentencias, decisiones, ramificaciones, condiciones y caminos) y *caja negra* (particiones de equivalencia, análisis de valores límite, conjetura de errores, gráficas de causa-efecto, etc.).
[Cuevas03].

Introducción

La utilización de software en entornos críticos como el militar, transportes, gubernamental o médicos implica la protección de datos personales tales como bases de datos, ficheros, aplicaciones, memoria del sistema y unidades de procesamiento. Por ello, uno de los aspectos del SQA es la de velar por la seguridad del software con el objetivo de satisfacer factores de calidad como integridad, disponibilidad y fiabilidad.

Es ampliamente asumida la importancia de la seguridad en sistemas computacionales en lo concerniente a redes sociales, aplicaciones móviles, la nube o en software ubicuo. Los ataques a dichos sistemas aprovechan las vulnerabilidades del software para conseguir acceso a través de técnicas como el *phishing*, el *malware* y el espionaje de datos por *hackers* o entidades no autorizadas. Por este motivo es necesario la construcción de software confiable que garantice al usuario su utilización segura. Para ello es imprescindible un diseño con una arquitectura robusta que prevenga, repela y se recupere de los ataques maliciosos. Puesto que las consecuencias de un fallo o un ataque pueden concernir vidas humanas o costes millonarios, es de suma importancia anticipar e identificar las condiciones o amenazas que puedan causar daños o vulnerabilidades. A este proceso se le denomina *análisis de amenazas*.

Las actividades recomendadas para la construcción de software seguro recomiendan no aplicar métodos *ad hoc* o sobre la marcha para ir parcheando errores y fallas de seguridad, pues es ineficiente y muy costoso.

Una de las recomendaciones es realizar revisiones de código antes de las pruebas para evitar fallos potenciales y mejorar la calidad del software. Durante la planificación del proyecto debe tenerse en cuenta el presupuesto y temporización de los objetivos de seguridad. En el análisis de riesgos se debe estimar el coste asociado a la perdida de datos o recursos producidos por fallos de software o ataques malintencionados. Así mismo, la identificación de

amenazas al sistema a menudo se retrasa hasta que los requisitos de un incremento software son trasladados a sus requisitos de diseño. Las revisiones de código centradas en aspectos de seguridad deben incluirse en la implementación y deben basarse en los objetivos y amenazas de seguridad identificados en las actividades de diseño.

Respecto a la fase de pruebas, el aseguramiento de la seguridad debe demostrar que se ha construido un sistema seguro que inspira confianza. Por este motivo, una de las actividades del aseguramiento de la seguridad es la realización de verificaciones que proporcionen evidencias de que el software cumple los requisitos. El aseguramiento de la seguridad se compone de una serie de artefactos auditables, llamados *casos de aseguramiento*, que confirman que el software satisface las demandas de seguridad. Como casos de aseguramiento pueden utilizarse, aunque no es suficiente, las pruebas formales de software y herramientas de análisis de vulnerabilidades como RATS (*Rough Auditing Tool for Security)*, ITS4 para C++ y SLAM desarrollado por Microsoft. [Pressman14].

Análisis de software seguro

análisis de requisitos

Es importante saber que los requisitos de seguridad son requerimientos no funcionales que influyen fuertemente en la posterior arquitectura del software. Una vez que se los requisitos de seguridad han sido analizados mediante el modelo de amenazas y el análisis de riesgos, se procede a crear un conjunto de políticas de seguridad. Una política de seguridad proporciona una definición de seguridad que incluye requisitos clave de seguridad y una serie de reglas que describen cómo aplicar la seguridad durante el funcionamiento del software. Puede que los requisitos de seguridad entren en conflicto con otros requisitos convencionales del software, tales como seguridad y usabilidad. No es algo infrecuente. Según [Marick02] el análisis de riesgos tiene que considerar los siguientes preceptos:

- Las necesidades de los usuarios respecto a la seguridad.
- La seguridad del diseño arquitectónico para que se ajuste a un buen diseño de interfaz de usuario.
- Una interfaz de usuario segura pero que al mismo tiempo posibilite aspectos como efectividad y eficiencia.

Durante el análisis de requisitos, el analista puede utilizar *patrones de ataque* que facilitan en gran medida la reutilización de escenarios de vulnerabilidad al modo problema → solución, como por ejemplo *phishing, SQL injection*, etc.

modelado de seguridad

Un modelo de seguridad es una descripción formal de una política de seguridad y puede ser una valiosa guía durante el diseño, implementación y el proceso de revisión. Un modelo de seguridad suele representarse de forma

textual o mediante formalismos gráficos como las máquinas de estado (Capítulo 11). El ingeniero de software debe asegurarse que las transiciones de la máquina de estados finita comiencen en un estado seguro y que finalicen en otro estado seguro. Otros formalismos gráficos utilizados en el modelado se seguridad están basados en el lenguaje **UMLsec** que comprende una extensión de UML utilizando estereotipos y restricciones y GRL que es otro lenguaje que permite la captura de requisitos no funcionales del sistema.

métricas de seguridad

Un software seguro debe cumplir los siguientes tres preceptos[7]:

1. Operar en situaciones hostiles.
2. El software no se comportará de una forma maliciosa.
3. Funcionar en cualquier situación comprometida.

Las citadas tres propiedades deben ser tenidas en cuenta por cualquier métrica de seguridad. Dichas métricas deben proporcionar a los desarrolladores una medida del nivel de confidencialidad de los datos y la integridad del sistema que podría estar en riesgo.

comprobaciones de seguridad

Las comprobaciones de seguridad deben realizarse durante todo el ciclo de vida del software utilizando auditorías, inspecciones y casos de prueba, pero fundamentalmente al comienzo del proceso de desarrollo. En las actividades del SQA se incluyen los estándares de seguridad y guías de desarrollo seguro para ser utilizados durante el modelado y la construcción del software. Las actividades de verificación comprueban que los casos de prueba de seguridad se han realizado correctamente y pueden ser trazados hasta los requisitos del sistema. Los datos recogidos durante las comprobaciones son analizados en

los casos de aseguramiento descritos en la sección 1.6.1. [Pressman14].

Análisis de riesgos

El análisis de riesgos es una actividad crítica en la planificación del proyecto. El método del *modelado de amenazas* se utiliza en el análisis de riesgos con el propósito de identificar posibles daños al software y debe realizarse en las primeras fases del ciclo de vida utilizando los requisitos y los modelos de análisis. *El modelo de amenazas* debe identificar los componentes clave del sistema, descomponerlo, clasificar las amenazas de cada componente del software por nivel de riesgo y desarrollar estrategias de migración. [Pressman14].

mda

En el año 2000 el *Object Management Group* (OMG) publicó su informe sobre la *Arquitectura Dirigida por Modelos* MDA **(***Model-Driven Architecture***)**, un paradigma de construcción de aplicaciones que innovaría por completo el panorama actual de las herramientas CASE de última generación para el desarrollo de aplicaciones. La idea principal en la que se basa MDA es motivar a desarrollar principalmente con modelos, con el fin de independizarse de la plataforma y el lenguaje de implementación. Muy recientemente ha surgido el paradigma MDE **(***Model-Driven Engineering***)** que es más avanzado y se basa en modelos abstractos del dominio más que en algorítmica computacional (léase [Brambilla17]).

Introducción

Desde siempre, la construcción de software complejo se ha vuelto económicamente muy costosa. La demanda de software continúa en auge y en los últimos años ha habido un incremento muy significativo en la industria del software. Las técnicas tradicionales de programación se han quedado obsoletas para abordar la complejidad y el coste del desarrollo de los futuros sistemas informáticos. Esto ha suscitado la aparición de enfoques, como el MDA, orientados a aumentar el nivel de abstracción en el desarrollo.

En los años 50 el desarrollo se realizaba únicamente introduciendo los códigos binarios directamente a la máquina, posteriormente, en los años 60 se inventó el lenguaje ensamblador que permitiría asociar códigos nemotécnicos a un conjunto de bits que representaban una instrucción. Con la llegada de los primeros lenguajes de alto nivel (Algol, Fortran, C), se pudo dar un paso más en el aumento del nivel de abstracción, convirtiendo sentencias sintácticas complejas en instrucciones en ensamblador. Finalmente, en los 80 emergió la Programación Orientada a Objetos que agilizó enormemente la productividad del software. El último paso en la evolución del desarrollo de software ha sido convertir los modelos textuales o gráficos en código ejecutable gracias a compiladores basados en MDA.

Además del nivel de abstracción, otro de los objetivos que persigue MDA es la posibilidad de aumentar el nivel de reutilización, permitiendo la construcción de software con bloques de diseño reutilizables. La otra gran ventaja que permite esta tecnología es la *Interoperabilidad de Tiempo-Diseño*, o lo que es lo mismo, la facilidad de desarrollar una aplicación independiente de su implementación lo que permite recombinar tecnologías.

Aunque la reutilización ha conseguido un gran avance en el mundo de la computación, aún existe un problema: hay poca reutilización de aplicaciones. Las librerías y los *frameworks* han conseguido un desarrollo espectacular, aun

así se los considera muy cercanos a nivel de la máquina.

Características de MDA

Modelos: Con los modelos permitimos elaborar una descripción abstracta del mundo. MDA permite modelar la aplicación con entidades que representan objetos del mundo concreto o abstracto; instrumentos como la abstracción y la clasificación permiten jerarquizar y ordenar los conceptos componentes del sistema. Para conseguir todas estas cosas, MDA cuenta con el uso de UML que es el estándar oficial para el modelado de software; aunque actualmente se están imponiendo los DSL **(***Domain Specific Languages* o Lenguajes específicos del dominio) que ya son una alternativa a UML y pueden ser textuales o gráficos.

Metamodelos: Un metamodelo es simplemente un modelo de un lenguaje de modelado como UML. Así, por ejemplo, un metamodelo para UML describe cómo éste crea las clases, sus relaciones y jerarquías en otro modelo más completo que el anterior y que recoge detalladamente cada característica del modelo que describe.

El metamodelo de UML se expresa mediante MOF **(***Meta Object Facility***)**, una herramienta estandarizada por la OMG. Un ejemplo es XMI, que es el acrónimo de (XML Metadata Interchange) y permite definir cómo se describen e intercambian los modelos. Muchas de las herramientas CASE actuales como Rational Rose, MagicDraw o StarUML utilizan este formato XML para expresar sus metamodelos.

Transformaciones entre modelos: La idea que se persigue con la transformación de modelos es poder convertir un modelo en otro similar, ya sea por refinamiento o abstracción. En este sentido sería posible transformar lo que se denomina PIM *(Platform Indepenent Model)* en un PSM *(Platform Specific Model)*. Para ello se utilizan las denominadas reglas de transformación y las marcas que permiten asociar y ajustar el nivel de detalle en dichas transformaciones. Se puede concebir las marcas como una especie de filtro, en el que

por un lado se introduce la transformación a realizar y por otro lado obtenemos la transformación realizada. El objetivo de estos es transformar un modelo de entrada en otro modelo de salida en el que, por ejemplo, se haya realizado una transformación para obtener un mayor nivel de refinamiento. Piense, por ejemplo, en una transformación entre un DSL textual para el lenguaje ensamblador al código específico de una determinada arquitectura hardware.

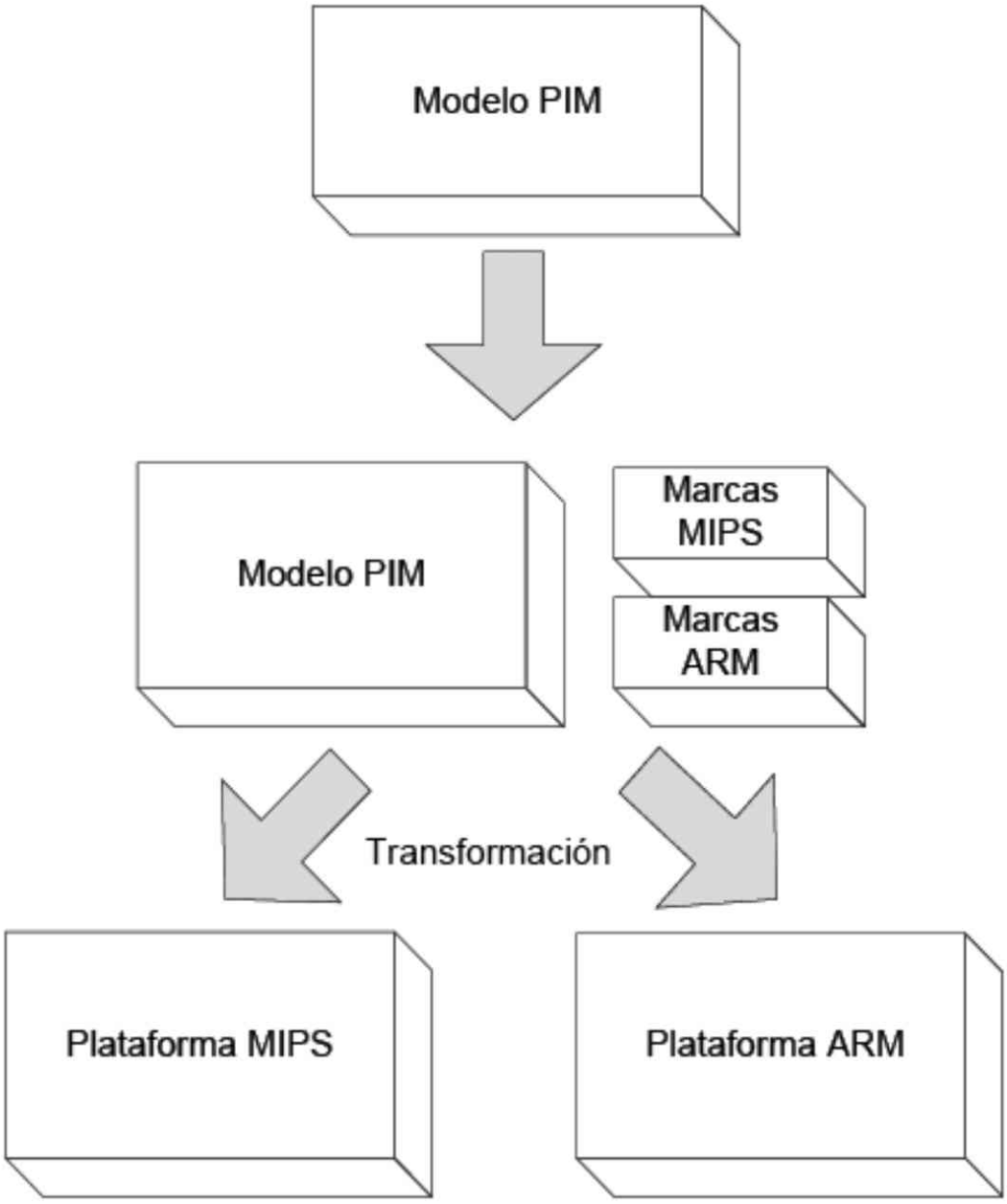


Figura 1.8. Transformación del PIM al PSM

Utilización de lenguajes: Una de las características más interesantes de MDA es la posibilidad de extender las capacidades de UML por medio de los perfiles (profiles). Estos se adaptan *"ad-hoc"* a cualquier dominio, plataforma o método que se elija, proporcionando una versatilidad y una potencia que alcanza una gran variedad de lenguajes y tecnologías. De esta forma es posible

encontrar herramientas MDA en el mercado que ofrecen perfiles tan variados como EJB, C++, Ruby, CORBA, etc. Una de las maneras de ampliar la potencialidad de los lenguajes ofrecidos por una herramienta de estas características es por medio de los *estereotipos*, los cuales permiten extender el vocabulario de UML, así como las *restricciones* (constraints) que especifican condiciones que deben cumplirse dentro de un modelo para su correcto funcionamiento.

Modelos ejecutables y MDA ágil: La idea subyacente de los modelos ejecutables es que se comporten directamente como código, facilitando la interacción con el dominio del cliente. La finalidad de este objetivo MDA es la independencia de la plataforma consiguiendo, entre otros aspectos, que tanto los modelos como el código sean una única entidad y que puedan ser ejecutados tan pronto como finalice el diseño de los mismos, proporcionando una realimentación con los clientes y expertos del dominio.

Para entender estas ideas supongamos que tuviera que desarrollar una aplicación con tres posibles sistemas operativos, tres bases de datos y tres implementaciones de *Servicios Web* (SOA). Se tendría en total $3 \times 3 \times 3 = 27$ implementaciones diferentes, lo que se concluye que la reutilización a nivel de código es multiplicativa, no aditiva.

Aplicación
Base de Datos (SGBD)
SOA
Plataforma (Sistema Operativo + Hardware)

Figura 1.9. Problema de la reutilización multiplicativa

En la figura 1.9 podemos observar el problema anteriormente citado. Para conseguir una solución aditiva que permita reutilizar una capa independientemente de las otras se deben interconectar con mecanismos que sean independientes de los contenidos de las otras capas. Esto supone un gran

avance, pues MDA impone la arquitectura únicamente en el último momento. En general las ventajas que se consiguen son muchas, entre ella la posibilidad de no tener que modificar los modelos para generar el código final en caso de cambiar alguna capa, el coste bajo del desarrollo multiplataforma, una productividad y un mantenimiento barato.

---

1 La OMG es un consorcio sin ánimo de lucro formado por varias empresas tecnológicas. Su finalidad es la gestión y estandarización de tecnologías orientadas a objetos como metodologías, bases de datos, CORBA, etc.

2 En estos dos últimos se le denomina paquete.

3 Cada tarjeta representa un objeto. En la cabecera se escribe el nombre de la clase que instancia, en la parte izquierda se indican las responsabilidades y, en la derecha, las clases con las que colabora junto a cada responsabilidad.

4 El equipo Scrum debe tener un entendimiento compartido de lo que significa que el trabajo esté completado con el fin de asegurar la transparencia.

5 Para más información consultar [Mingue03].

6 Para más información del plan SQA consultar [Cuevas03] y el estándar IEEE 730-1989 allí descrito.

7 *https://www.us-cert.gov/bsi*

diagramas de casos de uso

«No olvides que es comedia nuestra vida y teatro de farsa el mundo todo que muda el aparato por instantes y que todos en él somos farsantes;».
(Francisco de Quevedo: El Epicteto y Phocílides en español con consonantes).

Comenzamos este capítulo con el primero de los modelos dentro del ciclo de vida de un proyecto real. Los diagramas de casos de uso están determinados dentro de la fase de especificación de requisitos y permiten la representación del sistema desde el punto de vista del usuario o de agentes externos para así representar los requerimientos funcionales. Su utilización nos facilitará capturar mejor los requisitos del cliente.

introducción

Los modelos de casos de uso permiten tener una visión externa de cómo los usuarios interactúan con las funcionalidades del sistema. Permiten modelar los requisitos funcionales y los agentes involucrados en su uso para facilitar la validación del producto y la planificación de las fases del ciclo de vida. Dichos agentes (ya sean humanos o no) se les denomina "actores" y pueden ser por ejemplo: un usuario que solicita la compra de un producto, un estudiante que realiza un cuestionario, un dispositivo que envía un mensaje al sistema o un agente inteligente que genera acciones sobre la aplicación.

Los casos de uso se utilizarán en posteriores fases del proyecto para modelar el dominio y los diagramas de comportamiento (diagrama de secuencias y comunicación).

actores

Un actor representa un rol de algo externo al sistema y que interactúa con él. La mayor parte de las veces son usuarios, y a veces otros sistemas, e incluso *timers* que generan eventos hacia el sistema como un reloj del hardware que consulte el estado de un periférico. La diferencia principal entre un *actor primario* y uno *secundario* estriba en el que el primario inicia o pide la funcionalidad al sistema, mientras que en el secundario es el sistema el que inicia la acción y el actor el que la recibe.

Un actor se representa en UML como un muñeco de palo:



Figura 2.1. Ejemplo de actor en UML

Ejemplos de actores: un cliente, un vendedor, un estudiante, un operador, un

administrador del sistema, un jugador, una empresa virtual o real, un sistema, un servicio, un periférico, un agente, etc.

CASOS DE USO

Los casos de uso son normalmente implementados por un conjunto de sentencias de ejecución. Los actores son los que directamente interactúan con los casos de uso que se representan en UML con una elipse a la que se enlaza el actor.

Para identificar los casos de uso debemos averiguar qué funciones debe realizar el actor con el sistema, por ejemplo: iniciar una compra, apagar el sistema, enviar una señal a un periférico, mover un personaje en un laberinto, etc.

Para ello debemos tratar cuidadosamente los aspectos funcionales del sistema e intentar indagar qué funciones importantes deben realizarse por los actores principales de la aplicación.

La característica de modelado de casos de uso debe abstraer en gran medida lo fundamental del sistema, por muy grande que éste sea. Por tanto, cuando se modela un caso de uso debe asociarse con un conjunto de operaciones que a su vez serán implementadas en varias o muchas líneas de código.

Los casos de uso se representan con una elipse que enlaza con el actor o actores que interactúan con él mediante una línea llamada asociación.



Figura 2.2. Ejemplo de caso de uso

Como hemos visto, en la figura 2.2 se representa el caso de uso "*Hacer jugada*" para un hipotético actor de un escenario de un videojuego de ajedrez.

Los casos de uso mostrarán las situaciones en las que los actores externos al sistema, tales como usuarios, clientes, estudiantes, operadores, administradores, agentes software/hardware etc. interactúan con las funcionalidades básicas del sistema. Estas funcionalidades se describen a un alto nivel que luego se detallará y siempre contendrán otras tareas que debe realizar la aplicación a más bajo nivel de abstracción. Como veremos más adelante, los casos de uso se especificarán descomponiéndolos en acciones más pequeñas y que explican de manera más elemental los pasos que lo definen más en detalle.

A modo de ejemplo se presenta una parte elemental del diagrama de casos de uso para el escenario del juego de ajedrez:

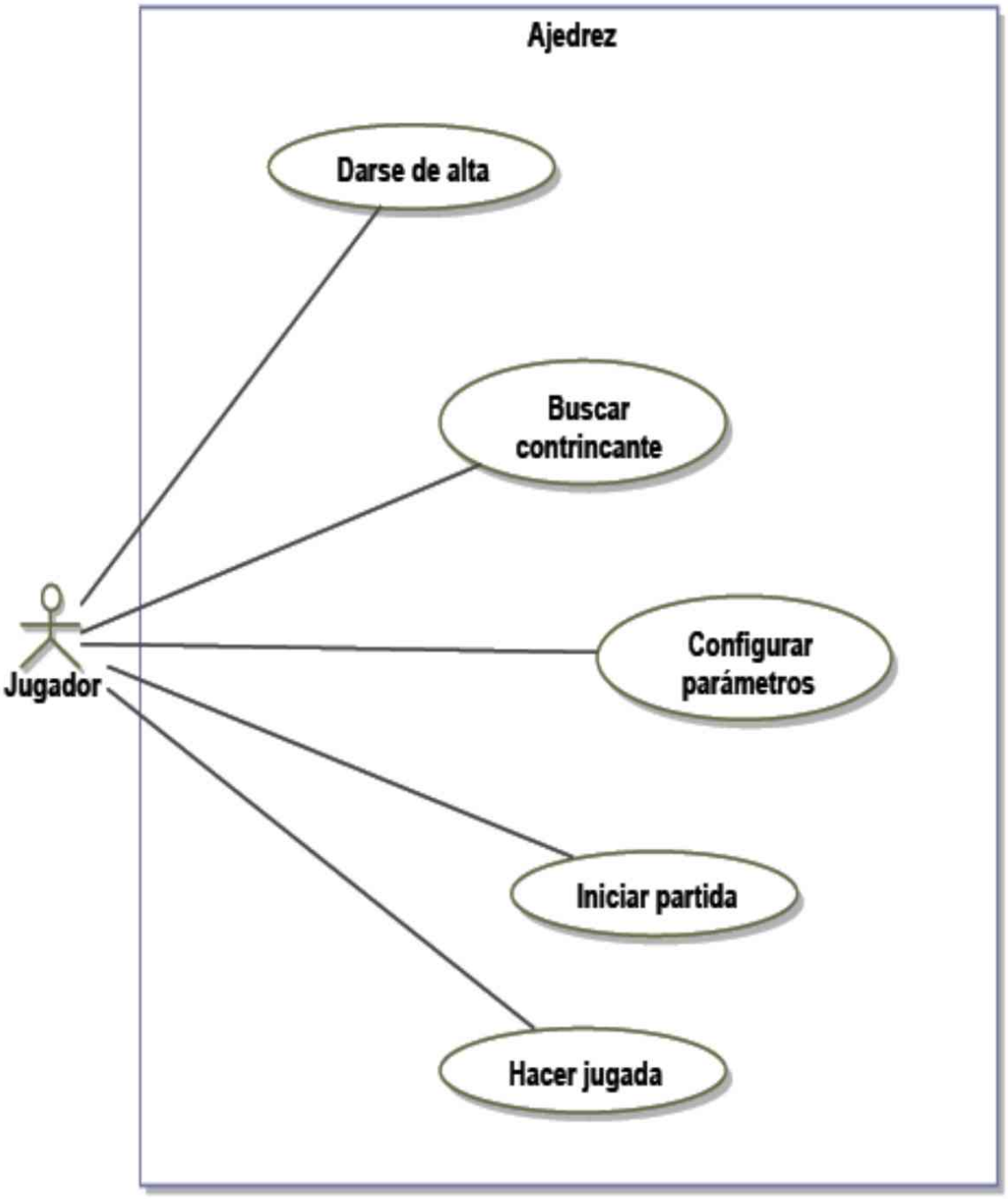


Figura 2.3. Ejemplo de diagrama de casos de uso para el juego de ajedrez

En el diagrama anterior se muestra el actor principal para un juego de ajedrez. El actor es "*Jugador*" que es el que inicia la partida y realiza los movimientos. El jugador representa al usuario humano y externo que interactúa con las funcionalidades del sistema. El rectángulo a modo de marco que rodea los casos de uso demuestra los límites del sistema con los actores

externos.

Especificación de los casos de uso

Dado el ejemplo de la figura 2.3, los cinco casos de uso representados se pueden especificar mediante una plantilla, basada en textos de referencia [Arlow07], en la que se debe recoger la siguiente información:

Caso de uso: Nombre
Identificador
Breve descripción
Actores primarios
Actores secundarios
Precondiciones
Flujo principal
Postcondiciones
Flujos alternativos

Tabla 2.1. Plantilla de especificación de un caso de uso

Donde el *identificador* identifica unívocamente el caso de uso mediante un número; la *descripción* resume de forma elemental el cometido del caso de uso en el contexto del sistema, mientras que la sección de *actores primarios* y *actores secundarios* especifican los actores involucrados en el sistema y que actúan directa o indirectamente con los casos de uso presentes. Las *precondiciones* indican las condiciones que se tienen que cumplir primeramente para que el caso de uso pueda comenzar, es decir, qué situación "a priori" es necesaria para ejecutarse la acción.

Una vez cumplidas las precondiciones se ejecuta la sección de *flujo principal* donde se ejecutan las acciones básicas asociadas al caso de uso. Ésta es considerada una parte troncal de la especificación y donde se definen las acciones fundamentales del caso de uso. Una vez finalizadas las sentencias básicas del caso de uso nos encontramos con las *postcondiciones* que indican

los requisitos necesarios del estado del sistema cuando el caso de uso ha finalizado, es decir, la situación "a posteriori" necesaria para cumplir la terminación del caso de uso. La última sección es para los *flujos alternativos* al flujo principal, que son una ampliación de las posibles situaciones anómalas o variaciones con respeto al tronco principal de ejecución del caso de uso.

Flujo principal

Para indicar el cauce de ejecución del caso de uso lo especificaremos con un número que representa un paso o macroinstrucción concreta. En el caso de que el paso se bifurque o se ramifique en pasos más pequeños debemos indicar la subdivisión mediante el uso de un punto y seguido.

Sentencia condicional “si”

Utilizaremos esta sentencia para indicar un salto condicional en la secuencia de ejecución. Para ello utilizaremos la palabra clave “si”[8] y posteriormente subdividiremos la numeración. En el ejemplo del caso de uso para mover pieza podemos apreciar la aplicación de “*si*”:

Caso de uso: Hacer jugada
Id: 4
Breve descripción: El jugador mueve una pieza.
Actores primarios: Jugador.
Precondiciones: 1. El turno debe ser del jugador actual. 2. Deben quedar piezas en el tablero. 3. No ha terminado el juego por tablas o victoria del contrincante.
Flujo principal: 1. El jugador selecciona una pieza con el cursor. 2. **Si** el jugador selecciona una casilla de destino. 2.1Mueve la pieza determinada a esa casilla. 2.2Dibuja la pieza. 2.3Comprueba si hay mate. 2.4Comprueba si está amenazada. 3. **Si** el jugador mueve el rey dos casillas, mover también la torre correspondiente (enroque).
Postcondiciones:

1. La casilla de destino está vacía u ocupada por una ficha enemiga.
Flujos alternativos: **PosiciónInválida.**

Tabla 2.2. Especificación para Hacer jugada

Flujo alternativo: Hacer jugada: Posición Inválida
Id: 4.1
Breve descripción: El sistema informa que el usuario ha movido la pieza a una casilla ocupada por otra de su mismo color.
Actores primarios: Jugador.
Precondiciones: 1. El jugador ha posicionado la pieza en un lugar incorrecto.
Flujo alternativo: 1. El flujo alternativo comienza en el punto 2.1 del flujo principal. 2. El sistema informa de que el jugador debe reposicionar la pieza.
Postcondiciones: Ninguna.

Tabla 2.3. Flujo alternativo para el caso de uso Hacer jugada

Durante el flujo principal en la tabla 2.2 se produce una ejecución de una sentencia condicional "*si*" y la numeración se parte para volver a comenzar en una nueva rama. Otro aspecto a tener en cuenta en la especificación de un caso de uso son las condiciones de bifurcación en caso de excepción, fallo o interrupciones dentro del flujo principal. Estas situaciones se deben especificar en la sección de *flujos alternativos* e indicar detalladamente su

descomposición en otra tabla especificativa (véase tabla 2.3). Durante la especificación del flujo alternativo debemos señalar el punto desde donde se produjo el fallo o la excepción, y posteriormente continuar la ejecución para el tratamiento del error siguiendo una determinada secuencia de pasos.

Sentencia de control “para”

La sentencia de control “*para*” permite definir una repetición de operaciones del caso de uso. Adicionalmente se añade una expresión condicional que se debe evaluar en cada iteración para finalizar la sentencia. A continuación se muestra un ejemplo con el caso de uso para configurar parámetros.

Caso de uso: Configurar parámetros
Id: 2
Breve descripción: Configura opciones del sistema.
Actores primarios: Jugador.
Precondiciones: 1. No debe haber comenzado la partida.
Flujo principal: 1. Preguntar tipo de control. 2. Preguntar nivel de dificultad. 3. Elegir color de las piezas. 4. Elegir diseño piezas. 4.1**Para** cada pieza del tablero. 4.1.1Elegir modelo. 4.1.2Elegir material.
Postcondiciones: Ninguna
Flujos alternativos: Ninguno.

Tabla 2.4. Ejemplo de la sentencia “para”

Sentencia de control “mientras”

Esta sentencia realiza un bucle de iteraciones con las acciones del caso de uso hasta que la expresión *booleana* sea evaluada a true. A modo de ejemplo se muestra a continuación un caso de uso donde se implementa esta sentencia:

Caso de uso: Validar usuario
Id: 1
Breve descripción: Comprueba que el usuario está registrado en el sistema.
Actores primarios: Jugador.
Precondiciones: 1. Existe conexión.
Flujo principal: 1. **Mientras** el usuario y/o la contraseña no sean correctos. 1.1 Pedir nombre de usuario. 1.2 Pedir contraseña. 1.3 Consultar en la base de datos.
Postcondiciones: Ninguna.
Flujos alternativos: Ninguno.

Tabla 2.5. Sentencia “mientras”

uso de <<include>> y <<extend>>

El uso de <<*include*>> permite, en el contexto de los casos de uso, una llamada a subrutina, reutilizando un caso de uso sencillo en otro más complejo. El modo de uso implica la asociación entre el caso de uso base y el incluido. El caso de uso base debe indicar el punto exacto donde debe realizarse la llamada al caso de uso repetitivo. Cuando el caso de uso invocado finaliza su flujo principal devuelve el control al caso de uso invocador. En la figura 2.4 se muestran dos casos de utilización de <<*include*>> para el juego del ajedrez.

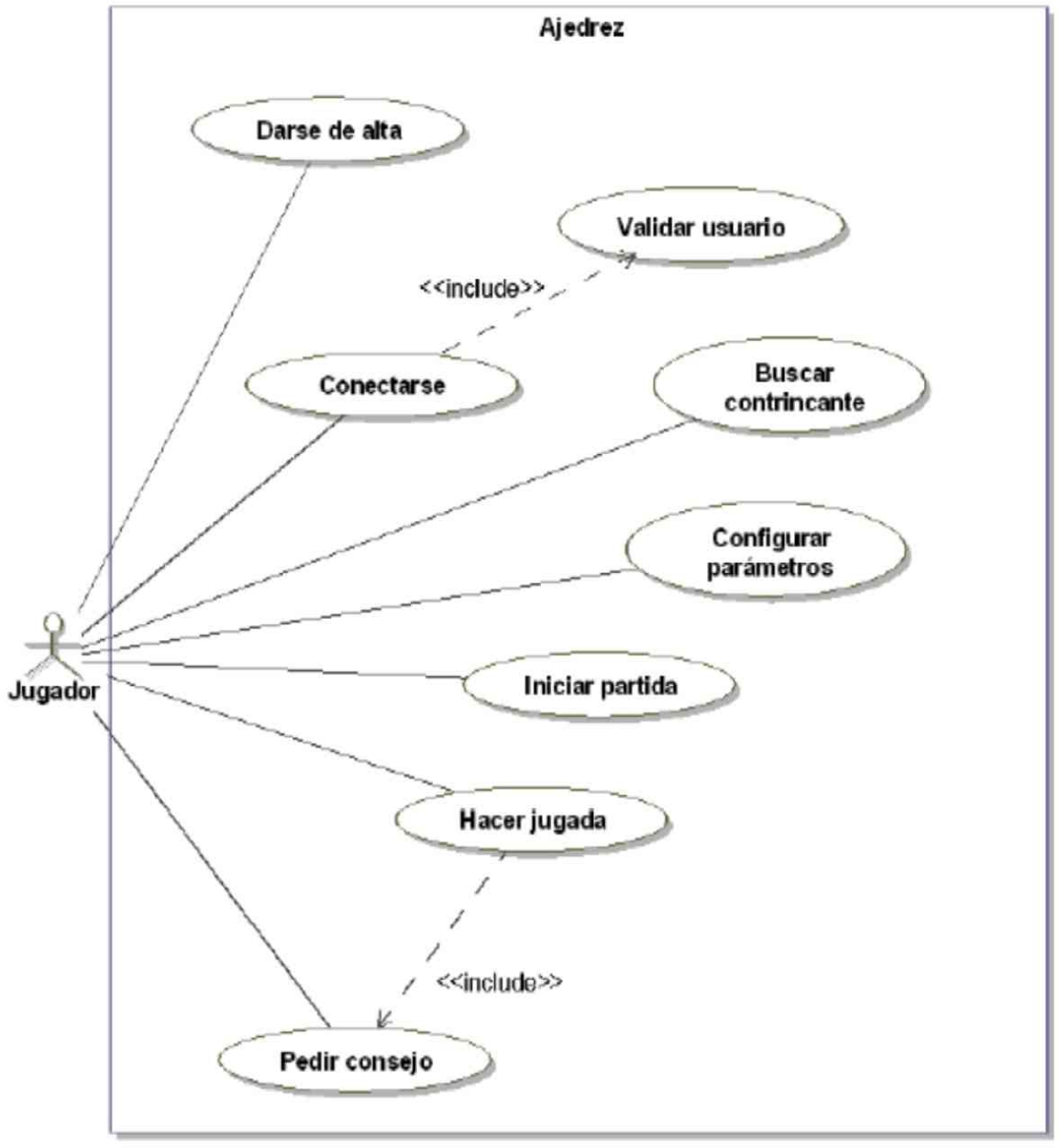


Figura 2.4. Diagrama ampliado para el juego del ajedrez

Caso de uso: Hacer jugada
Id: 4
Breve descripción: El jugador mueve una pieza.
Actores primarios: Jugador.
Precondiciones: 1. El turno debe ser del jugador actual. 2. Deben quedar piezas en el tablero. 3. No ha terminado el juego por tablas o victoria del contrincante.
Flujo principal: 1. El jugador selecciona una pieza con el cursor. **2. Si** el jugador selecciona una casilla de destino. 2.1 Mueve la pieza determinada a esa casilla. 2.2 Dibuja la pieza. 2.3 Comprueba si hay mate. 2.4 Comprueba si está amenazada. **3. Si** el jugador mueve el rey dos casillas, mover también la torre correspondiente (enroque). **4. Si** el usuario pide consejo 4.1 **Include** (Pedir consejo)
Postcondiciones: Ninguna.
Flujos alternativos: PosiciónInválida.

Tabla 2.6. Utilización de «include» en el caso de uso Hacer jugada.

Caso de uso: Pedir consejo
Id: 5
Breve descripción: Calcula una posible jugada.
Actores primarios: Jugador.
Precondiciones: 1. El turno debe ser del jugador actual que pide consejo.
Flujo principal: 1. **Mientras** no encontrada solución. 1.1. Buscar alternativa óptima.
Postcondiciones: 1. 1. El movimiento resultante debe ser óptimo.
Flujos alternativos: Ninguno.

Tabla 2.7. Caso de uso para Pedir consejo pieza.

En el caso de <<*extend*>> la situación difiere en cuanto a la relación entre el caso de uso que invoca el caso de uso invocado. Un caso de uso "extiende" a otro cuando tiene un comportamiento muy similar pero con una pequeña diferencia (es parecida a la herencia entre clases). Es decir, el caso de uso que recibe el <<*extend*>> añade la nueva funcionalidad del otro caso de uso en una determinada situación excepcional. El caso de uso que es extendido indica en un "punto de extensión" (en inglés "extension point") el lugar del

caso de uso en el que puede haber diferencias. Puesto que el caso de uso base no conoce nada sobre el caso que extiende, debe indicarlo mediante la cláusula: *extension point*, seguida del nombre del caso de uso extendido.

Con la finalidad de comprender mejor la utilización de estas características extendidas de los diagramas de casos de uso, mostramos a continuación el modelo completo para el diagrama del juego de ajedrez:

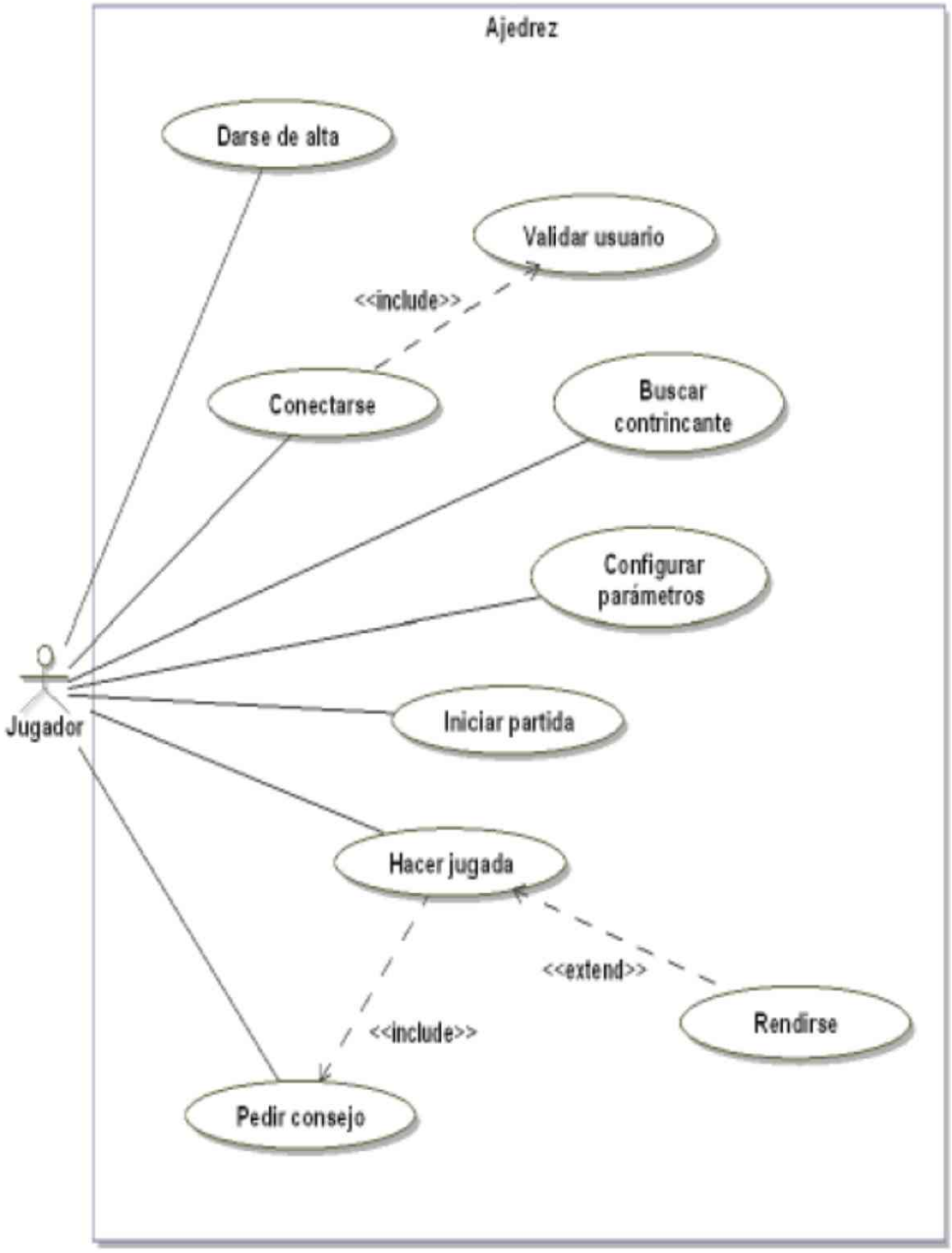


Figura 2.5. Diagrama completo con el uso de «extend»

Caso de uso: Hacer jugada
Id: 4

Breve descripción: El jugador (usuario) mueve una pieza en el tablero.
Actores primarios: Jugador.
Precondiciones: 1. El turno debe ser del jugador actual. 2. Deben quedar piezas en el tablero.
Flujo principal: 1. El jugador selecciona una pieza con el cursor. 2. **Si** el jugador selecciona una casilla de destino. 2.1 Mueve la pieza determinada a esa casilla. 2.2 Dibuja la pieza. 2.3 Comprueba si hay mate. 2.4 Comprueba si está amenazada. 3. **Si** el jugador mueve el rey dos casillas, mover también la torre correspondiente (enroque). 4. **Si** el usuario pide consejo 4.1 Include (Pedir consejo) 5. **Extension point: Rendirse**
Postcondiciones: 1. La casilla de destino debe estar vacía.
Flujos alternativos: PosiciónInválida.

Tabla 2.8. Caso de uso invocador.

Caso de uso: Rendirse

Id: 6
Breve descripción: El usuario decide rendirse.
Actores primarios: Jugador.
Segmento 1 Precondiciones: Ninguna.
Segmento 1 Flujo: 1. El usuario pulsa el botón de rendición. 2. El sistema pregunta si realmente se da por vencido. 3. **Si** el usuario confirma rendición, finalizar partida.
Segmento 1 Postcondiciones: Ninguna.

Tabla 2.9. Caso de uso a insertar

La posibilidad de utilizar *varios segmentos (uno o más)* es también válida para la opción de «*extend*». Ésta es útil cuando se requiere especificar claramente el caso de uso extendido y adicionalmente se pretende regresar al flujo principal del caso de uso base. La solución viene por ampliar el número de segmentos.

CASO de ESTUDIO: mercurial

Un programa de control de versiones, CVS (Concurrent Versions System), es un repositorio donde se almacenan las diferentes versiones de los archivos de un proyecto en desarrollo. Dicho repositorio es creado y mantenido por el administrador del proyecto, mientras que los desarrolladores se encargan de actualizar el repositorio añadiendo, leyendo o borrando ficheros de código fuente. El administrador es el único que tiene privilegios para crear, clonar y borrar repositorios, además de ser la autoridad encargada de conceder los permisos al resto de usuarios del sistema de control de versiones. Los usuarios, es decir, los programadores, tendrán la posibilidad de actualizar el repositorio, en tiempo real y concurrentemente, con sus últimas versiones del programa, creando un árbol jerarquizado de números de versión de software. Además, podrán comparar dos ramas de dicho árbol. El administrador, se encargará de fusionar varias ramas para crear un mismo tronco común de desarrollo. Finalmente, y puesto que es un programa distribuido en red, es necesaria la autenticación de los usuarios, al iniciarse el sistema, para poder registrarse en el proyecto actual y poder realizar operaciones dentro del repositorio.

El programa *Mercurial* es un buen ejemplo de estudio y especialmente adecuado para aplicar los conocimientos adquiridos en este capítulo. Este sistema de control de versiones de software permite el seguimiento por varios desarrolladores de las versiones que han sido utilizadas durante la construcción de un proyecto. La aplicación está disponible en Internet con licencia GPL y se encuentra implementada en varias plataformas como Linux, Windows y Mac OS X.

Finalmente, y aunque a lo largo de los siguientes capítulos continuaremos expandiendo los modelos hasta su implementación, no se pretende emular aquí el programa real de red, sino que más bien se crearán partes concretas

que se encuentren estrechamente ligadas con el contenido teórico del libro.

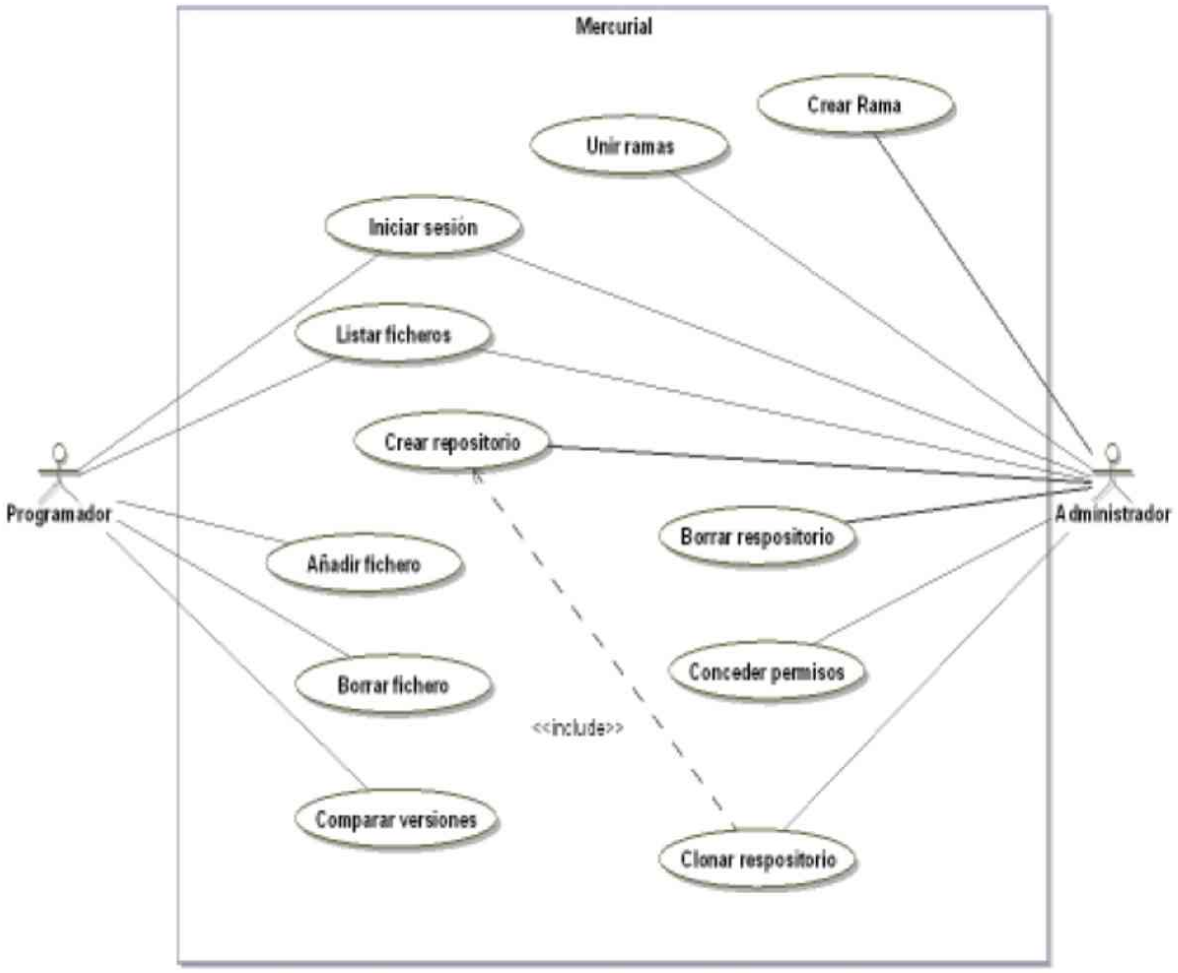


Figura 2.6. Diagrama de casos de uso del programa Mercurial

En el diagrama se pueden apreciar los dos actores principales que interactúan con el sistema. Son los dos perfiles de usuario, es decir, el "*Programador*" y el "*Administrador*". El administrador realiza la acción "*Clonar repositorio*" que invocará implícitamente al caso de uso "*Crear repositorio*", ya que clonar implica crear otro directorio con idéntica estructura que el objeto clonado.

Finalmente, el programador será el responsable de realizar las acciones que se relacionan con añadir, borrar, listar y comparar versiones.

Para ampliar la semántica del diagrama de la figura 2.6 continuaremos el estudio realizando las especificaciones de tres de ellos que por sus características tienen cierta relevancia en el contexto de la aplicación. En primer lugar se especificará el caso de uso "*Clonar repositorio*" que incluye una

llamada implícita a "*Crear repositorio*" y finalmente "*Listar ficheros*" cuyo caso de uso es compartido por los dos actores principales que intervienen en el sistema.

Caso de uso: Clonar repositorio
Id: 1
Breve descripción: El administrador realiza una copia exacta y duplicada del directorio de ficheros.
Actores primarios: Administrador.
Precondiciones: 1. Ningún fichero debe estar bloqueado.
Flujo principal: 1. El administrador selecciona el directorio origen. 2. **Include** (Crear repositorio) 3. **Para** cada archivo en directorio origen. 3.1 Copia archivo de directorio origen a directorio destino.
Postcondiciones: Ninguna.
Flujos alternativos: Ninguno.

Tabla 2.10. Caso de uso Clonar repositorio

Caso de uso: Crear repositorio
Id: 2
Breve descripción: Crea un directorio vacío.

Actores primarios: Administrador.
Precondiciones: 1. Debe quedar espacio de almacenamiento.
Flujo principal: 1. Crea directorio.
Postcondiciones: 1. No debe generarse ningún error del sistema.
Flujos alternativos: Ninguno.

Tabla 2.11. Caso de uso Crear repositorio

Caso de uso: Listar ficheros
Id: 3
Breve descripción: El usuario solicita una lista de archivos del repositorio.
Actores primarios: Administrador, Programador.
Actores secundarios: Ninguno.
Precondiciones: Ninguna.

Flujo principal: 1. El usuario selecciona el repositorio. 2. **Para** cada archivo en directorio seleccionado. 2.1 Mostrar archivo en pantalla del cliente.
Postcondiciones: 1. Deben haberse listado todos los archivos sin errores.
Flujos alternativos: Ninguno.

Tabla 2.12. Caso de uso Listar ficheros

En las anteriores tablas se recogen de forma concreta las especificaciones de los casos de usos anteriormente citados (si bien son los que mejor describen el funcionamiento de Mercurial y su estructura conceptual).

En todos los diagramas se incluyen las notaciones vistas durante el capítulo. Así, por ejemplo, en la tabla 2.10 se muestra la utilización de la palabra reservada *include* para especificar el caso de inclusión de la *creación de repositorio* al clonar un repositorio por el administrador.

Cada una de las especificaciones ya vistas introduce las secciones correspondientes de precondiciones y postcondiciones de la misma forma que se indicó en la plantilla de la tabla 2.1. Dichas guardas nos servirán de advertencias a considerar en el diseño e implementación de los modelos UML.

A modo de ejemplo veamos continuación un caso real de creación de proyecto y uso del *commit* en Mercurial:

(1) $ hg init (project-directory)

(2) $ cd (project-directory)

(3) $ (add some files)

(4) $ hg add

(5) $ hg commit -m ‘Initial commit’

(1) Inicia sesión creando un nuevo repositorio.

(2) Accede al repositorio.

(3) y (4) Añade ficheros al repositorio.

(5) Aplica los cambios mediante “commit”.



Figura 2.7. Logotipo de Mercurial (GPLv2)

CASO de ESTUDIO: servicio de cifrado remoto

El ejemplo propuesto de servicio cifrado remoto consiste en una aplicación distrubuida entre varios clientes y varios servidores. Los clientes, en los que trabajan los usuarios noveles, permiten únicamente cifrar mensajes hacia los servidores (ver flecha unidireccional en figura 2.8); mientras que los usuarios avanzados descifran los mensajes provenientes de los clientes en los servidores. Los servidores pueden cifrar y descifrar mensajes entre usuarios avanzados como muestran las flechas bidireccionales de la figura 2.8. Los métodos de encriptación utilizados en el sistema estarán basados en cifrado simétrico con el algoritmo AES (*Advanced Encryption Standard*) y cifrado híbrido mediante una combinación de cifrado simétrico AES y cifrado asimétrico RSA (*Rivest, Shamir y Adleman*). La aplicación, tanto en el servidor como en el cliente, dispondrá de una interfaz textual con un menú de opciones para elegir el tipo de algoritmo de cifrado y poder también salir al terminal del sistema operativo.

Un sistema de cifrado simétrico utiliza una misma clave privada para cifrar y descifrar entre los dos extremos. De forma alternativa, un sistema de cifrado asimétrico utiliza un par de claves pública y privada para cifrar y descifrar respectivamente. Así, por ejemplo, si el cliente quiere enviar un mensaje cifrado al servidor, éste utilizará la clave pública del servidor para cifrar dicho mensaje. Cuando el servidor reciba el mensaje del cliente, lo descifrará mediante su clave privada correspondiente. En un caso de cifrado híbrido se empleará conjuntamente tanto un sistema simétrico como asimétrico. Dicho sistema híbrido utilizará una clave pública para cifrar una clave de sesión simétrica previamente generada, después, el mensaje a enviar al servidor se cifrará utilizando el anterior algoritmo simétrico. Por último, el servidor utilizará su clave privada para desencriptar la clave simétrica de sesión y descifrar con ella el mensaje completo.

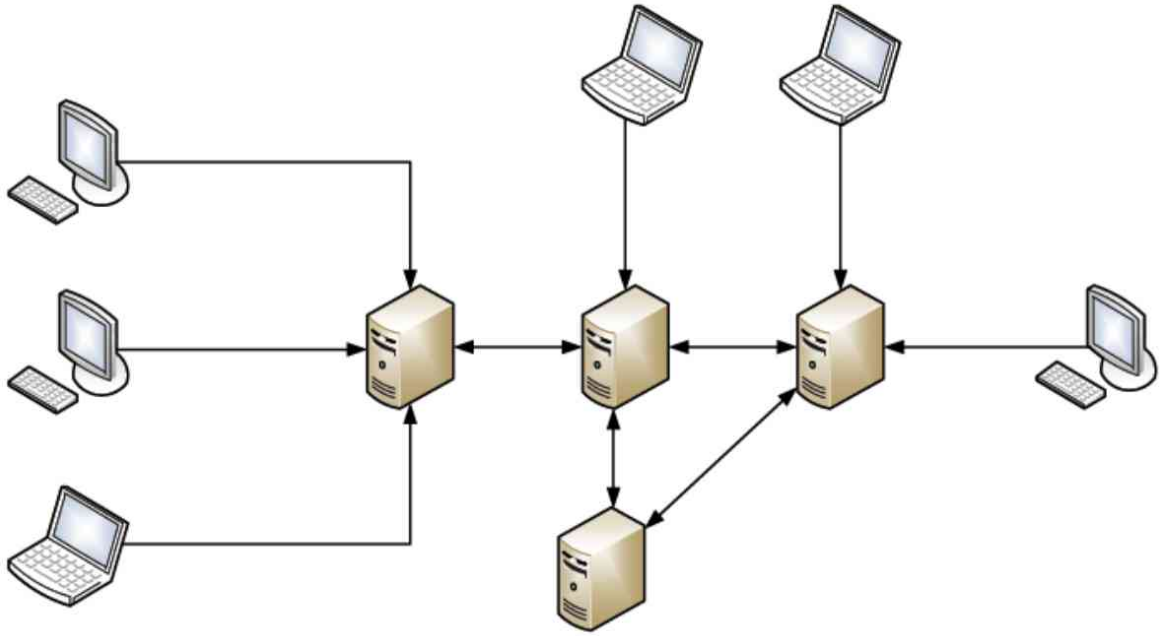


Figura 2.8. Servicio de cifrado remoto

La figura 2.9 muestra el diagrama de casos de uso de la aplicación cliente:

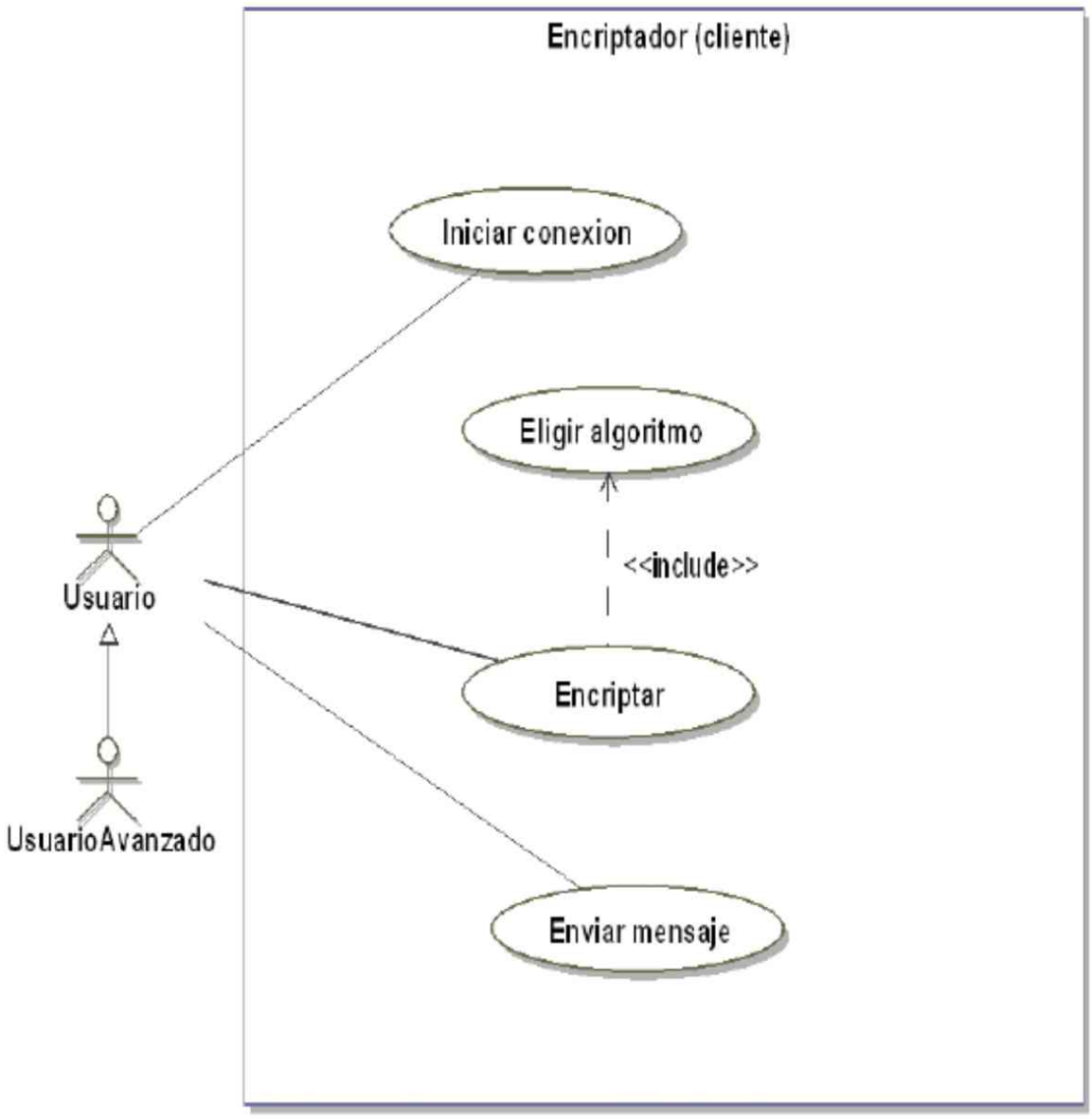


Figura 2.9. Diagrama de casos de uso para el cliente

En el diagrama se pueden apreciar los diferentes casos de uso para el usuario novel. Estos son:

1. *"Iniciar conexión":* Realiza la conexión con el servidor mediante el protocolo TCP/IP.
2. *"Elegir algoritmo":* El usuario novel puede optar por cifrar de forma simétrica (AES) o de forma híbrida con una combinación RSA-AES.
3. *"Encriptar":* Se realiza el cifrado del mensaje de texto y se genera la

clave simétrica y/o asimétrica dependiendo del caso.

4. *"Enviar mensaje"*: Una vez cifrado el mensaje se procede al envío del mismo por la red.

Veamos ahora la especificación del caso de uso "Encriptar":

Caso de uso: Encriptar
Id: 1
Breve descripción: El usuario novel encripta el mensaje de texto mediante uno solo de los algoritmos elegidos.
Actores primarios: Usuario.
Precondiciones: 1. Deben generarse las claves simétricas y/o asimétricas según el caso.
Flujo principal: 1. **Include** (Elegir algoritmo). 2. El usuario novel escribe el mensaje de texto. 3. **Si** cifrado es simétrico 3.1 Genera clave privada y cifra con AES. 4. Si cifrado es híbrido 4.1 Genera clave de sesión simétrica con AES, privada y pública con RSA. 4.2 Cifra clave sesión simétrica con clave pública del servidor. 4.3 Cifra mensaje con clave simétrica.
Postcondiciones: El mensaje de texto no puede estar en blanco.
Flujos alternativos:

Ninguno.

Tabla 2.13. Caso de uso Clonar repositorio

En la tabla 2.13 se realiza la especificación del caso de uso "Encriptar" que va a requerir de una llamada a la interfaz de usuario para seleccionar el algoritmo determinado, introducir el mensaje de texto en el terminal y generar las claves según se haya optado por un algoritmo simétrico o híbrido. Finalmente, se requiere la postcondición de que el mensaje de texto no se encuentre vacío.

Caso de uso: Elegir algoritmo
Id: 2
Breve descripción: El usuario novel elige entre algoritmo simétrico o híbrido.
Actores primarios: Usuario.
Precondiciones: 1. Ninguna.
Flujo principal: 1. Selecciona un algoritmo determinado.
Postcondiciones: 1. La opción debe ser la correcta.
Flujos alternativos: Ninguno.

Tabla 2.14. Caso de uso Crear repositorio

Ahora procedemos a diseñar el diagrama de casos de uso del servidor, tal

como muestra la figura 2.10.

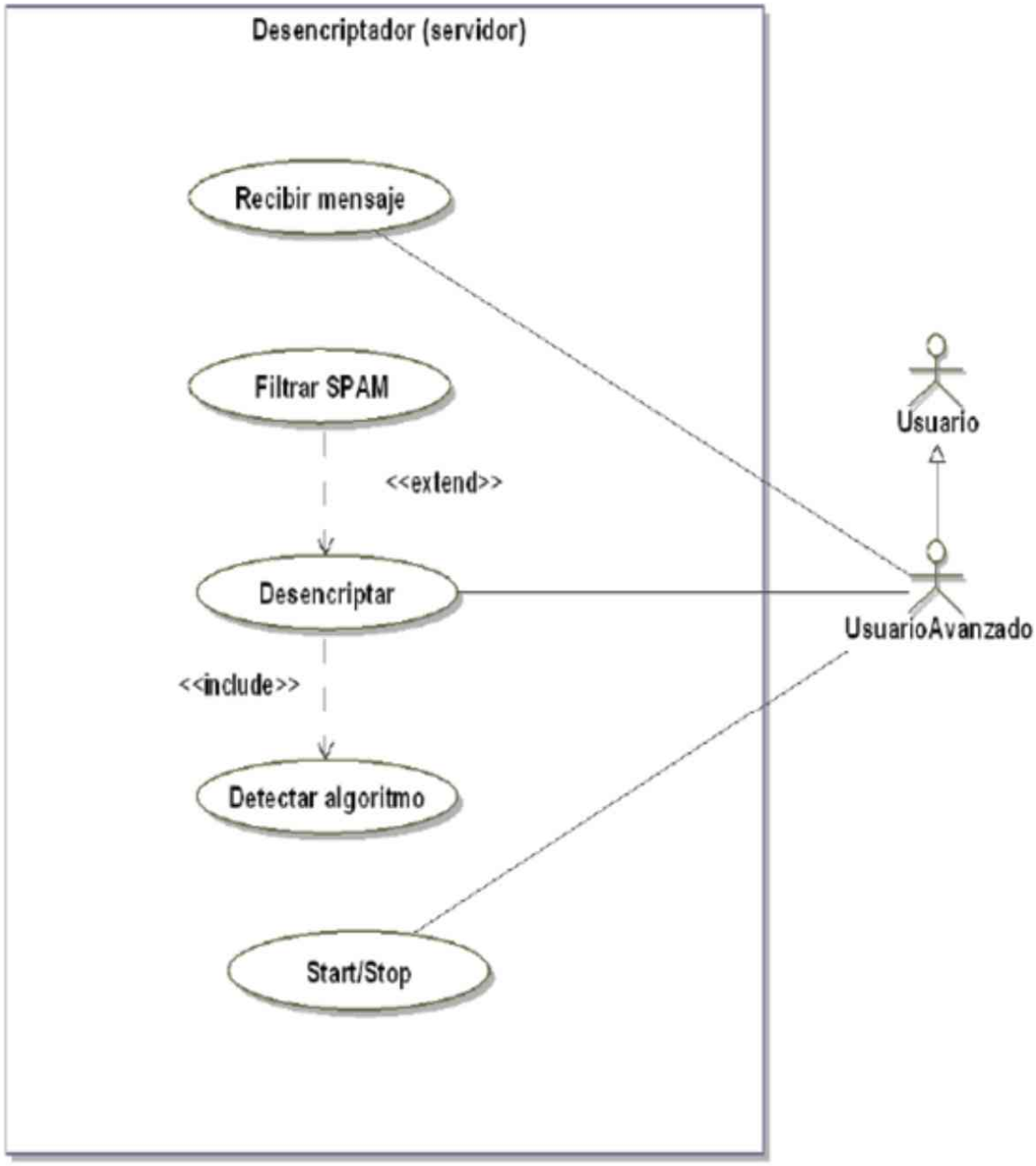


Figura 2.10. Diagrama de casos de uso para el servidor

que incluye los siguientes casos de uso:

- *"Recibir mensaje":* Recibe el mensaje cifrado enviado por un cliente (usuario novel) o un servidor (usuario avanzado).
- *"Detectar algoritmo":* Realiza una detección del método concreto de

cifrado del mensaje. Prueba primero con el algoritmo simétrico y luego con el híbrido.

- *"Desencriptar":* Realiza la desencriptación con el algoritmo correcto.
- *"Filtrar SPAM":* Filtrar mensajes no deseados sobre el mensaje descifrado.

Ahora procedemos a generar la especificación del caso de uso "*Desencriptar*":

Caso de uso: Desencriptar
Id: 3
Breve descripción: El usuario avanzado desencripta el mensaje cifrado del cliente o de otro servidor.
Actores primarios: Usuario avanzado.
Precondiciones: 1. El mensaje no debe estar vacío. 2. Debe contar con la clave de sesión para el descifrado híbrido.
Flujo principal: 1. **Include** (Detectar algoritmo). 2. **Si** el algoritmo es simétrico. 2.1 Descifra con clave privada simétrica. 3. **Si** el algoritmo es híbrido. 3.1 Descifra clave sesión simétrica con clave privada del servidor. 3.2 Descifra mensaje con clave sesión simétrica descifrada. 4. **Extension po**int: Filtrar spam

Postcondiciones: 1. No debe generar error del sistema al descifrar.
Flujos alternativos: Ninguno.

Tabla 2.15. Caso de uso desencriptar

Caso de uso: Detectar algoritmo
Id: 4
Breve descripción: El sistema detecta el tipo de algoritmo de cifrado del mensaje.
Actores primarios: Usuario avanzado.
Precondiciones: 1. El mensaje no debe estar vacío.
Flujo principal: 1. Prueba los dos tipos de métodos de cifrado utilizados.
Postcondiciones: 1. No debe generar excepción del sistema.
Flujos alternativos: Ninguno.

Tabla 2.16. Caso de uso invocado

Caso de uso: Filtrar spam
Id: 5

Breve descripción: Realiza el filtrado de mensajes no deseados.
Actores primarios: Usuario avanzado.
Segmento 1 Precondiciones: El mensaje debe estar descifrado y no vacío.
Segmento 1 Flujo: 1. **Mientras** no mensaje limpio 1.1 Recorre palabra por palabra 1.2 Si palabra detectada como dañina en sistema experto 1.2.1Elimina palabra.
Segmento 1 Postcondiciones: Mensaje limpio de basura.

Tabla 2.17. Caso de uso a insertar

La tabla 2.15 dirigirá el flujo principal al realizar primero una detección del algoritmo concreto de cifrado (tabla 2.16) y proceder, posteriormente, al descifrado del mensaje dependiendo del tipo del algoritmo utilizado. Por último, una vez que se ha desencriptado el mensaje del cliente (usuario o usuario avanzado), mediante uno de los dos algoritmos anteriormente vistos, el sistema ofrece la posibilidad de filtrar mensajes nocivos procedentes de la red (tabla 2.17), con la utilización de un algoritmo experto que no se detallará en este libro.

8 Con el fin de no mezclar el español y el inglés usaremos la palabra "si" en la descripción de los pasos condicionales.

diagramas de robustez

«—El análisis es el alma del pensamiento y el fantasma del ingenio».

(Raheel Farooq).

Entramos en la fase del modelo de análisis, justo después de haber realizado la captura de requisitos mediante los diagramas de casos de uso. En esta etapa se comenzarán a extraer las primeras clases preliminares pertenecientes al modelo de dominio y las correspondientes a la interfaz de usuario. Desde esta perspectiva, iremos detallando cada caso de uso especificado en el capítulo anterior. En este capítulo se crearán un conjunto de diagramas que contendrán entidades cuyo origen está en los casos de uso vistos en el capítulo precedente.

conceptos básicos

Con el fin de relacionar cada caso de uso con un conjunto de objetos que participen en la descripción de los requisitos funcionales del mismo, es necesaria la ampliación de nuestro léxico dentro de la terminología UML con las siguientes nuevas notaciones:

1. **Objetos de frontera:** Estos objetos representan entidades del sistema con las cuales el actor se relaciona directamente. Vienen a ser interfaces de usuario, ventanas, formularios, incluso botones. Se representan mediante la siguiente notación:

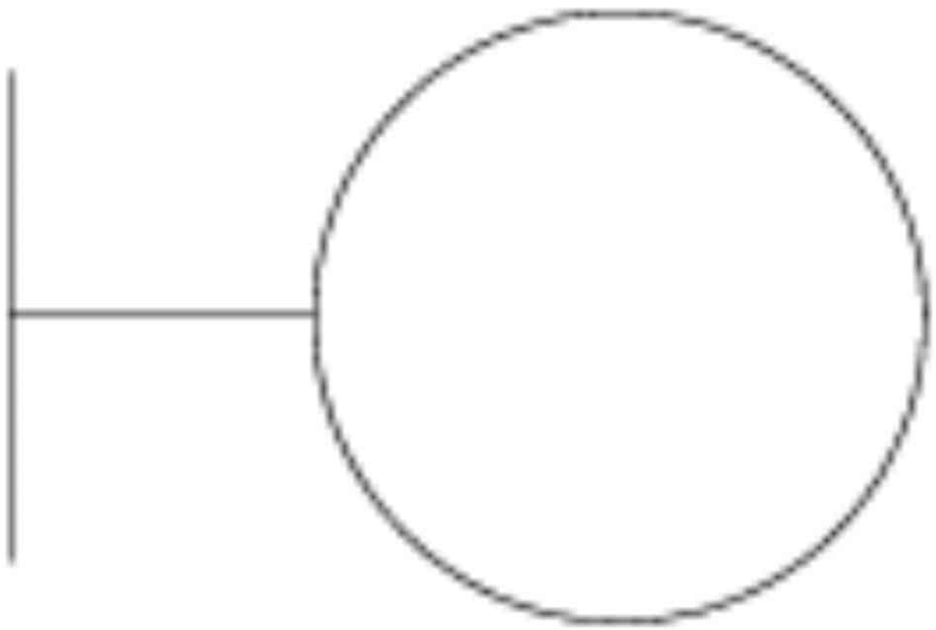


Figura 3.1. Objeto de frontera (Boundary)

2. **Objetos entidad**: Representan objetos extraídos del modelo del dominio como por ejemplo: bases de datos, ficheros, tablas de bases de datos o clases que almacenen algún tipo de información. La notación que utilizaremos para representarlos será la siguiente:

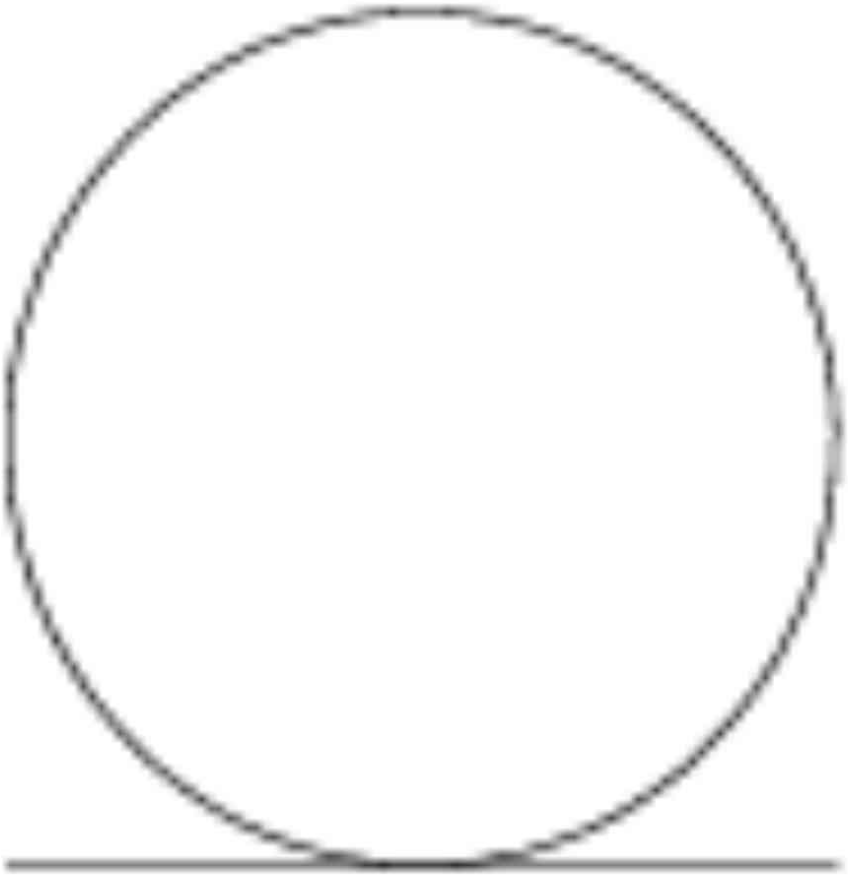


Figura 3.2. Objeto Entidad (Entity)

3. **Objetos de control:** Las clases de control son, en definitiva, las que manejan a las anteriores, soportando lo que se ha venido en llamar*lógica de negocio*. Permiten la interconexión entre los objetos entidad y los de frontera.

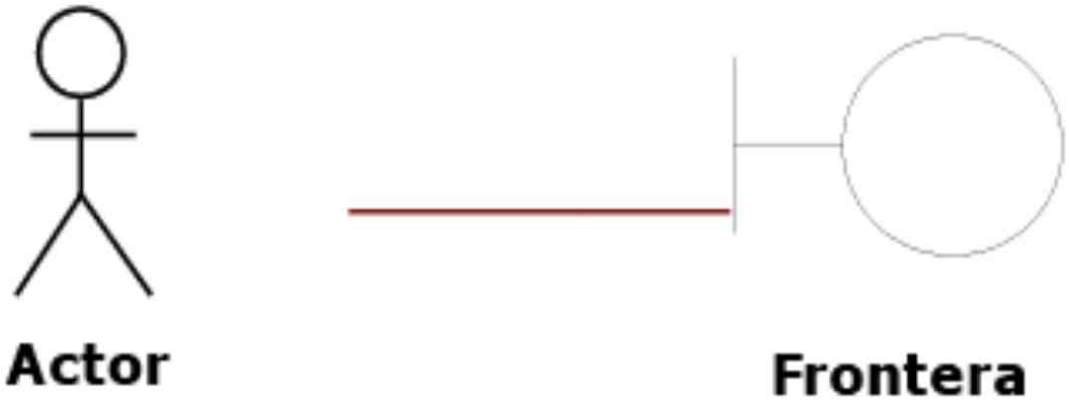
Las situaciones que se ajustan a los objetos de control son, por ejemplo: las funciones de validación, algoritmos u operaciones que trabajan con datos. Se representa como:



Figura 3.3. Objeto de control

Desde la perspectiva de la interconexión entre elementos de este modelo, es importante conocer las siguientes restricciones:

- Los actores solo pueden relacionarse con objetos de frontera:



- Los objetos de frontera pueden comunicarse únicamente con controladores en ambos sentidos:



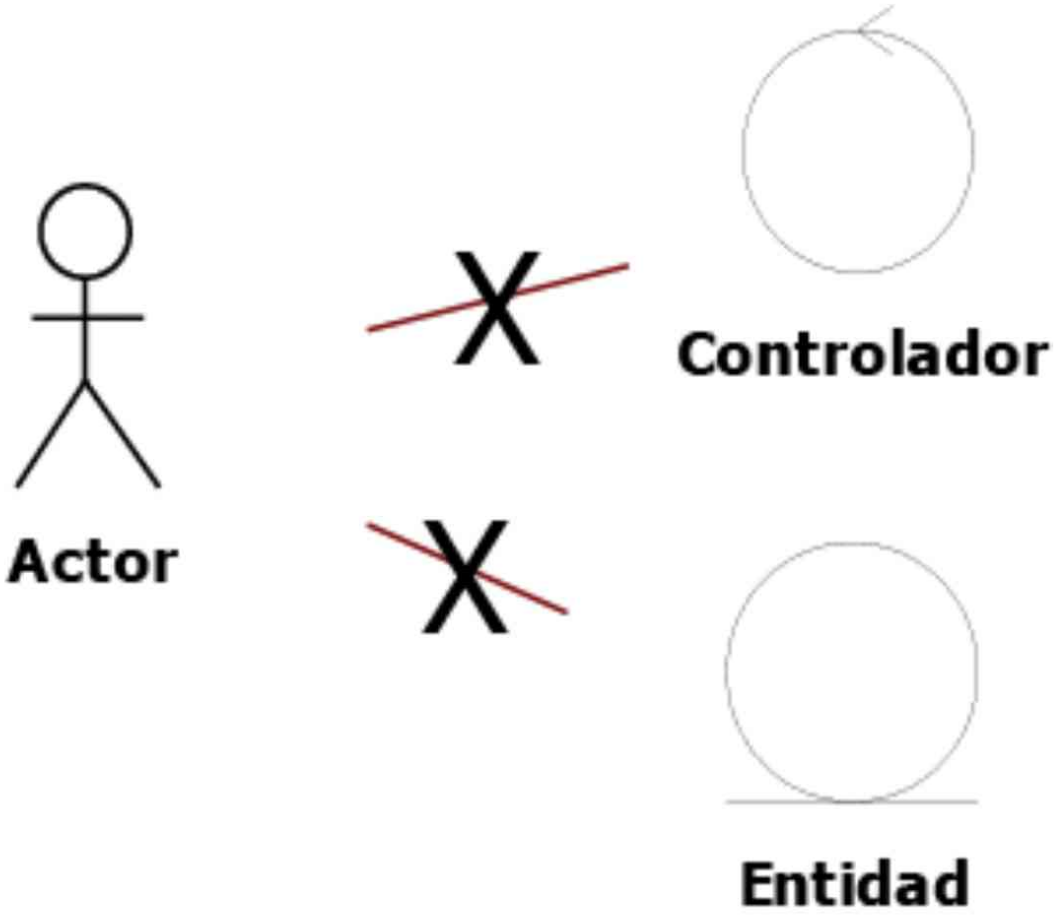
- Los objetos de control pueden comunicarse entre ellos sin ninguna limitación:



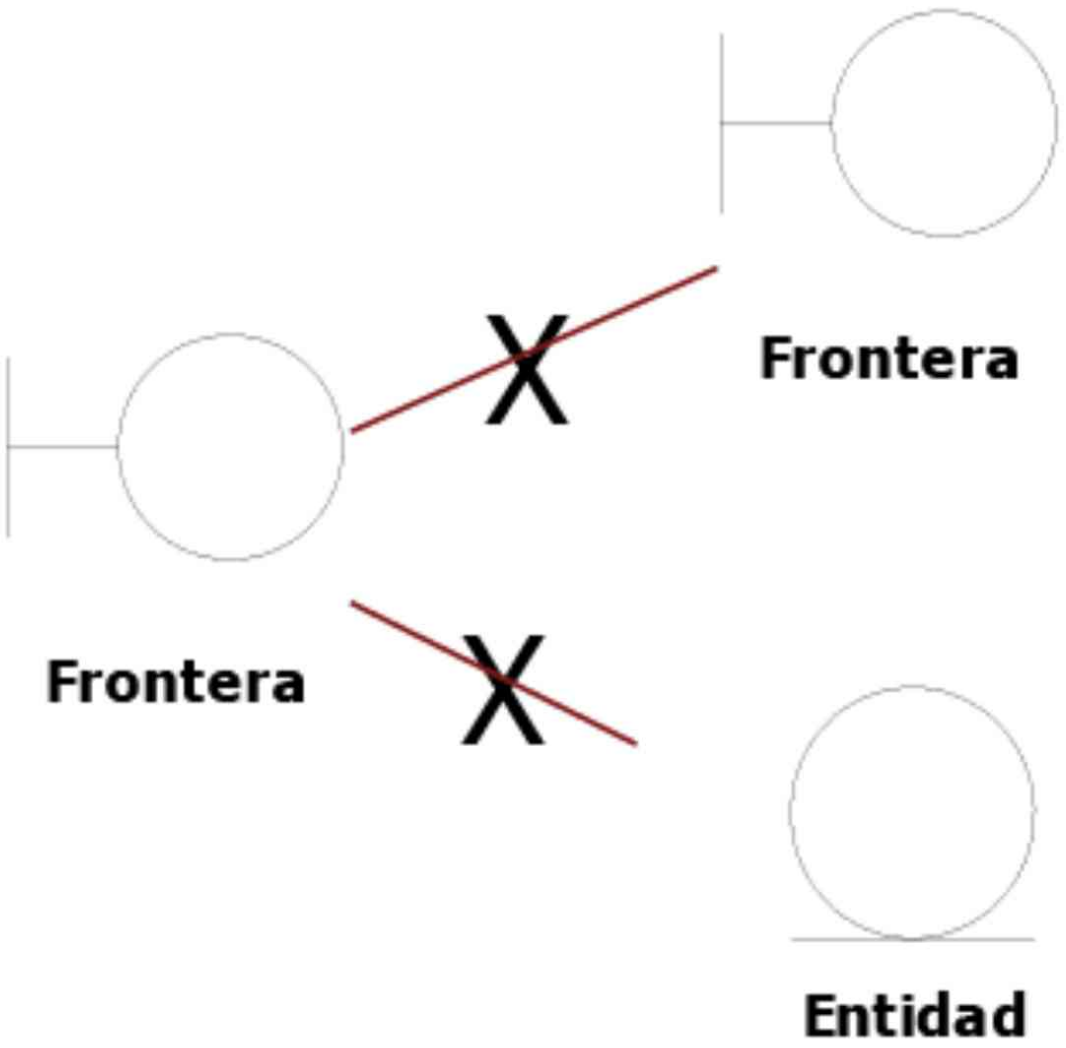
- Los objetos de control se pueden relacionar con objetos de entidad en los dos sentidos:



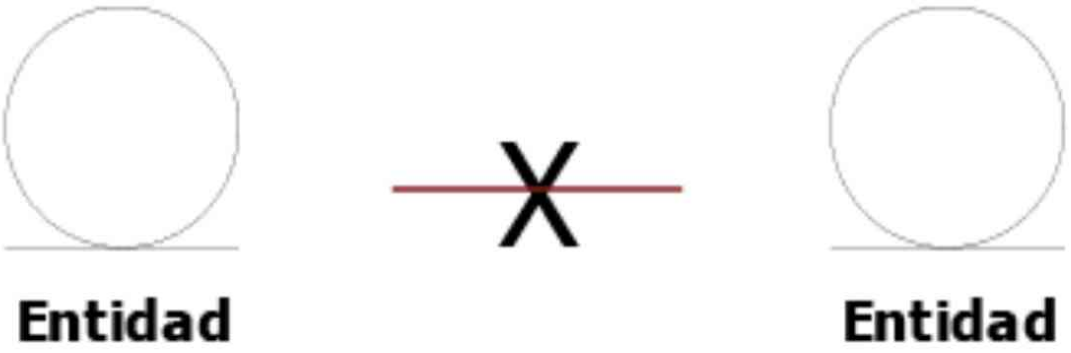
- Los objetos de control y de entidad no pueden relacionarse con los actores:



- Un objeto de frontera no puede relacionarse con otro de frontera, ni con uno de entidad:



- Dos objetos de entidad no pueden interrelacionarse entre ellos:



importancia de los diagramas de robustez

Los diagramas de robustez se encuentran ubicados entre las fases de especificación y la de diseño. Son una forma de representar los casos de uso de un modo homogéneo y permiten comprobar su corrección desde el punto de vista de la semántica del análisis. Entre las claves para la incorporación de los diagramas de robustez a nuestro proyecto destacan la ayuda para descubrir si hemos olvidado algún objeto del sistema que no hemos tenido en cuenta, y comprobar si el comportamiento del sistema es razonable y coherente dentro del modelo. Al realizar estos pasos con el diagrama siempre es posible encontrar entidades mal identificadas (clases o componentes del dominio) o que realizan indebidamente una determinada operación dentro del conjunto del sistema.

En general, las ventajas de estos diagramas es que son más sencillos que otros más detallados de UML como los de secuencias (interacción). Los símbolos propios de este diagrama están claramente basados en el patrón *Modelo-Vista-Con-trolador (MVC)* como veremos en el capítulo dedicado a los patrones de diseño. Entre otros beneficios, facilita una trazabilidad entre lo que el sistema hace (casos de uso) y el modo de funcionamiento (diagramas de secuencia). Obliga a escribir los casos de uso usando un estilo consistente y correcto.

Como se verá en este capítulo, existe una gran diferencia de facilidad de lectura de los diagramas de robustez frente a los diagramas de secuencias. Estos últimos requieren de una sintaxis más compleja, unida a un conjunto de símbolos más amplio y una semántica más extensa. Además de estas ventajas anteriormente enumeradas, se incluyen: la capacidad para separar y ordenar adecuadamente las capas de la lógica de negocio y la interfaz de usuario para aplicaciones cliente/servidor, y como conexión entre el hueco semántico existente entre el análisis de requerimientos (casos de uso) y el diseño (diagramas

de secuencias).

Después de describir los diagramas de robustez a partir de las especificaciones de los diagramas de caso de uso, es conveniente realizar un modelo estático del domino, que será el precedente del diagrama de clases. Con este fin es importante no emplear mucho tiempo en detallar este diagrama, pues esta función se realizará en el diseño.

caso de estudio: ajedrez

Caso de uso "Hacer jugada"

Comenzamos la explicación de cómo diseñar un diagrama de robustez partiendo del caso de uso del juego de ajedrez. Para ello partimos del diagrama de casos de uso, tal como se ilustró en la figura 2.5 y las especificaciones de las tablas 2.7, 2.8 y 2.9 del capítulo 2.

En la figura 3.5, puede verse el diagrama completo para el caso de uso *"Hacer jugada"* y sus casos relacionados a través de las cláusulas <<include>> y <<extend>>: *"Pedir consejo"* y *"Rendirse"*, que implican dos situaciones diferentes presentadas en el transcurso funcional del caso de uso base.

Para proceder al diseño de robustez, es necesario en primera instancia examinar detenidamente las especificaciones de los casos de uso relacionados. En este caso son *"Hacer jugada"*, *"Pedir consejo"* y *"Rendirse"*. Debemos comenzar con la *identificación* de objetos que cumplan los criterios para los símbolos de este tipo de diagrama, tal como se mencionó en el apartado 3.1. Así para el caso de uso *"Hacer jugada"*, en el que se incluyen los casos relacionados a través de <<include>> y <<extend>>, tenemos en primer lugar al actor *"Jugador"* que se relaciona con los cuatro objetos de frontera: *"Mover ficha"*, *"Enrocarse"*, *"Pedir consejo"* y *"Rendirse"*. *"Mover ficha"* permite que el jugador (actor) interactúe con el sistema por medio de la interfaz de usuario gráfica representada por un tablero de ajedrez, y donde el usuario puede mover la correspondiente ficha a otra posición dentro del tablero. Éste es, sin embargo, el punto de interacción entre el actor humano y el sistema o aplicación. Así mismo, como especialización o caso particular del objeto de frontera (véase *herencia* en capítulo cuatro) *"Mover ficha"* se encuentra el objeto *"Enrocarse"*, el cual es una caso concreto que es utilizado cuando el rey debe cruzarse con la torre.

Estos objetos de frontera comentados se deben ahora relacionar con algún objeto de control para poder transitar desde la capa de interfaz de usuario a la

capa de la lógica de negocio. De esta forma, los objetos *"Mover ficha"*, *"Enrocarse"* y "*Pedir consejo*" se comunican con el objeto de control *"Control juego"*. Éste será el encargado de redirigir la petición y manejar una cierta parte de lógica relacionada con el control del juego. Por este motivo, *"Control juego"* se intercomunica con los objetos de entidad *"Estado piezas humano/IA"* (para verificar/consultar la posición de la piezas y el movimiento correcto dentro de las reglas del ajedrez). También se relaciona con "*Control juego*" "*AlgoritmoIA"* (que es el objeto de control que implementa el algoritmo de Inteligencia Artificial "minimax" o la "poda alfa-beta) y se encargará de las heurísticas y la estrategia de la computadora.

Como en cualquier aplicación profesional, el jugador también tiene la posibilidad de pedir un asesoramiento al motor de IA del sistema, mediante el objeto de frontera "*Pedir consejo*". Este objeto será un botón con el cual se indica la necesidad de ayuda. A esta llamada responderá el objeto *"Control juego"*, que consultará al objeto *"AlgoritmoIA"* y éste a *"Parámetros juego"* y *"Estado piezas humano/IA"* para el asesoramiento de la jugada más adecuada dependiendo del nivel de dificultad elegido. El objeto de entidad *"Estado piezas humano/IA"* se comunicará con el objeto de control *"Dibujar piezas*" con el fin de dibujar los *bitmaps* de las piezas del ajedrez una vez se haya decidido la jugada tanto por el jugador humano como por la IA.

Por último, se añade al diagrama el objeto de frontera *"Rendirse"*, que en los casos de uso se representaba mediante la relación <<extend>> e indica la posibilidad del jugador de abandonar la partida. Éste se comunicará con el objeto de control *"Finalizar juego"* que será el responsable de terminar la partida.

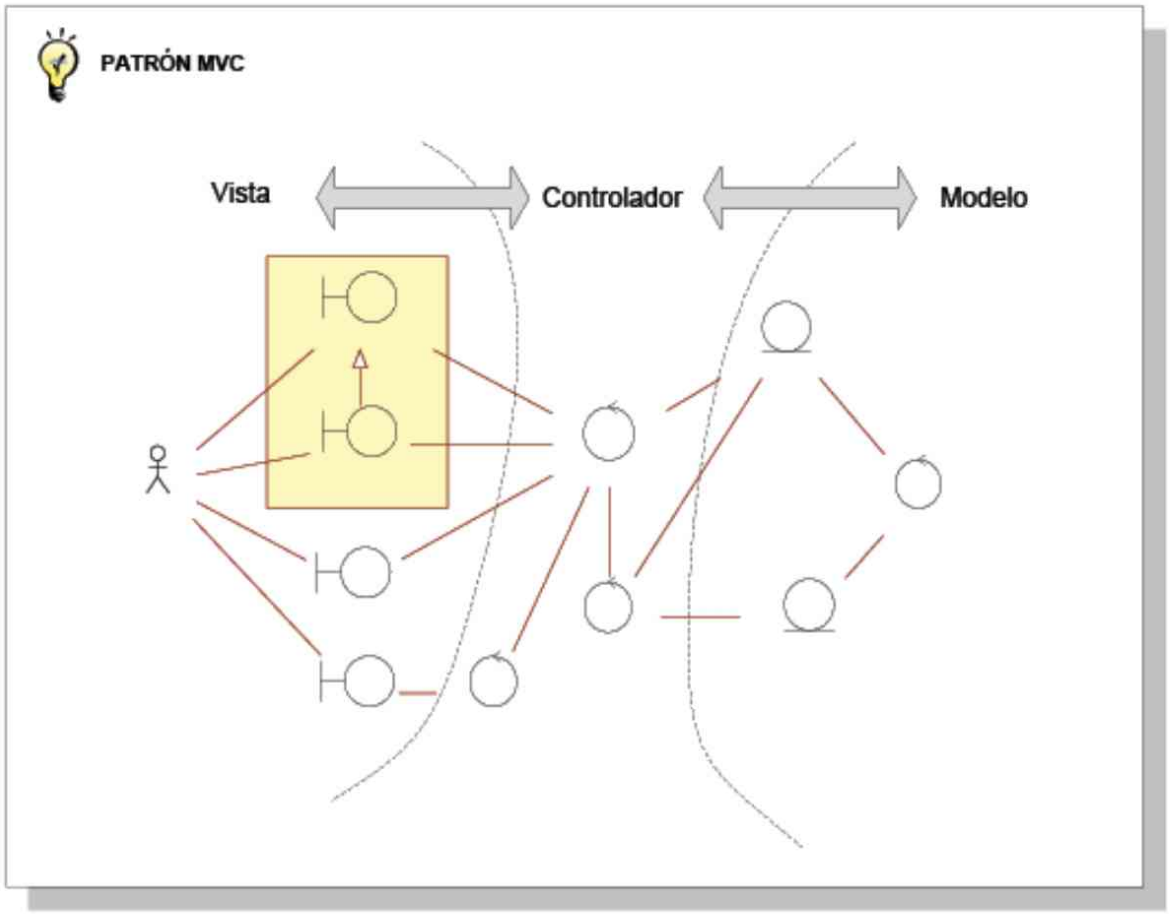


Figura 3.4. Los casos de uso de la aplicación ajedrez se adecuan al patrón MVC

La aplicación del juego de ajedrez se amolda perfectamente al patrón MVC (*Modelo-Vista-Controlador*) como es el caso de la distribución de los objetos del diagrama de robustez presentado en la figura 3.5 que refleja fielmente dicho patrón arquitectónico.

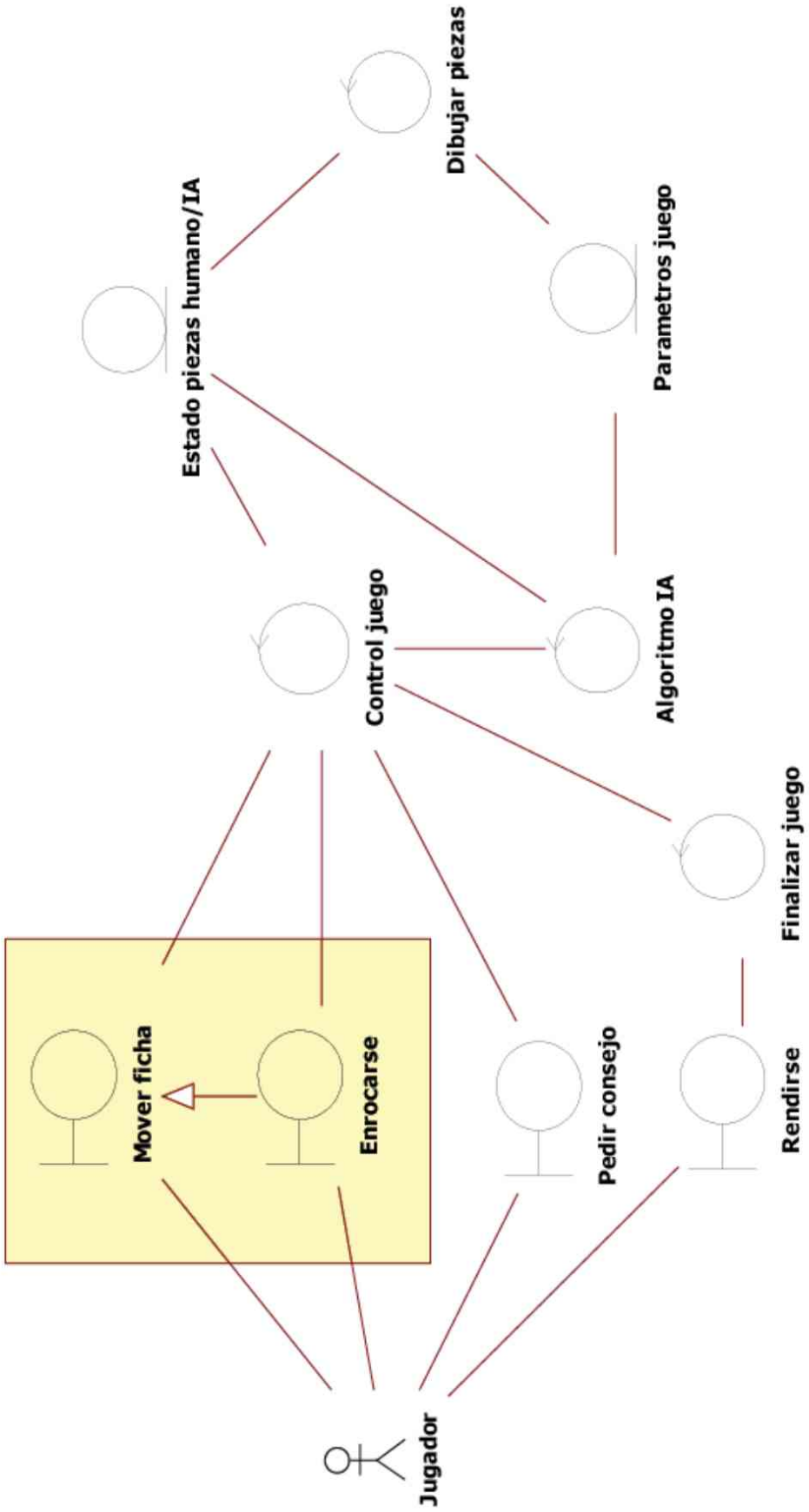


Figura 3.5. Diagrama de robustez del caso de uso “Hacer jugada”

En relación a los enlaces entre símbolos, algunos autores prefieren indicar las asociaciones mediante flechas dirigidas. Esto es una opción que depende de la semántica que el diseñador pretenda dar a su modelo. En nuestro caso, como puede apreciarse en la figura 3.5, hemos optado por asociaciones no dirigidas. Gracias a esto no se añade complejidad innecesaria al modelo de análisis, postergando ese tipo de decisiones al diseño detallado.

Caso de uso "Configurar parámetros"

Volviendo a la figura 2.5 y revisando la especificación del caso de uso *"Configurar parámetros"* indicada en la tabla 2.4, obtenemos el diagrama de robustez de la figura 3.6. Para obtener este diagrama se analiza en primer lugar los actores implicados en el sistema. Como bien indica la especificación, el actor principal del caso de uso es *"Jugador"* el cual es el responsable de iniciar la acción y comunicar al sistema su deseo de cambiar las opciones de la partida. En segundo lugar hemos de identificar los objetos de borde. En este caso existe únicamente el objeto *"Opciones"* que representa el menú de selección de preferencias del usuario. Puesto que las opciones seleccionadas por pantalla deben ser redirigidas a algún objeto de la capa de negocio que se encargue de gestionar la configuración de la partida, surge la necesidad de relacionar ambas partes mediante un objeto de control. En este caso el objeto *"Configurador"* será el responsable de enlazar la llamada del objeto de frontera con el objeto de entidad *"Parámetros juego"*.

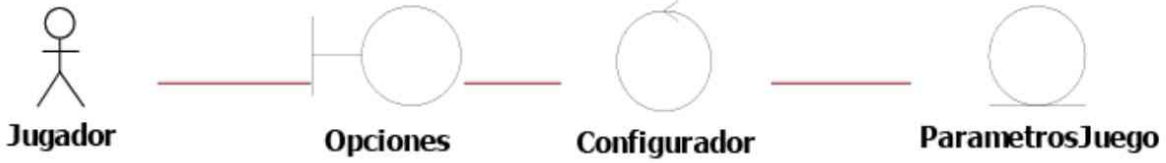


Figura 3.6. Diagrama de robustez del caso de uso "Configurar parámetros"

Como puede observar, este caso de uso es mucho más simple que el anterior, pero afortunadamente nos sirve como modelo para entender la filosofía de los tres símbolos básicos implicados en los diagramas de robustez. Sin ir más lejos, este ejemplo ilustra lo que podría ser una implementación UML del patrón "*Modelo-Vista-Controlador*" para una aplicación de tres capas.

Es importante que identifique correctamente cada uno de los objetos componentes del diagrama, y que siga el curso dentro del ciclo de vida elegido para su aplicación. Si antes ha realizado los diagramas de casos de uso, éstos le facilitarán los pasos adecuados para identificar con claridad cada clase de

objeto en este diagrama.

Caso de uso "Validar usuario"

Para el caso de uso *"Validar usuario"* se procede de la misma forma que para el ejemplo anterior. Si bien en este diagrama se aplica a una transacción cliente/servidor en Internet.



Figura 3.7. Ejemplo para el caso de uso "Validar usuario"

En este ejemplo existe de nuevo el actor jugador que solicitará la petición para registrarse en el sistema de juego por medio de la interfaz de acceso. Después se remitirá la petición al servidor con el control *"Validar usuario"*, para ser finalmente recibido en la base de datos de usuarios (representada en la figura como un objeto de entidad).

caso de estudio: mercurial

Caso de uso "Listar ficheros"

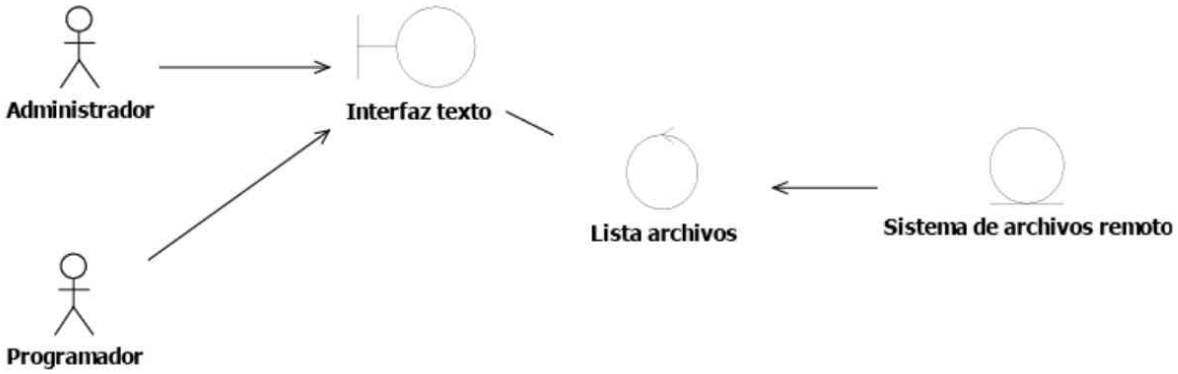


Figura 3.8. Diagrama de robustez para el caso de uso "Listar ficheros" de Mercurial

De igual forma que en el caso que hemos visto sobre el juego de Ajedrez, el *"Administrador"* y el *"Programador"* son los actores principales que inician la interacción con el sistema invocando a los objetos de frontera en primer lugar. El objeto de frontera *"Interfaz texto"* será el responsable de iniciar el listado de ficheros invocando al objeto de control *"Lista archivos"* que extraerá los datos del objeto de entidad "*Sistema de archivos remoto*". El flujo de información con la lista de archivos realizará el camino de vuelta hacia el objeto de frontera "*Interfaz texto*" con la finalidad de mostrar los datos a los usuarios.

Caso de uso "Clonar repositorio"

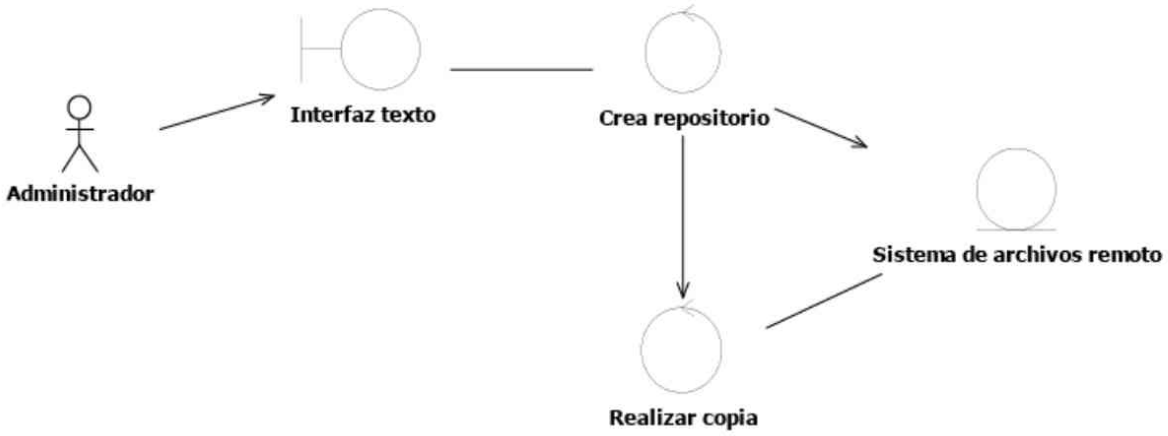


Figura 3.9. Diseño de robustez del caso de uso "Clonar repositorio" de Mercurial

En este caso la lógica de interacción es la misma. El actor principal *"Administrador"* que es el único responsable de replicar la estructura de directorios del sistema de archivos remoto, realizará una invocación al objeto de frontera *"Interfaz texto"* y éste a su vez pedirá al objeto de control *"Crea repositorio"* que cree un nuevo directorio o realice una réplica del actual repositorio de archivos remotos, mediante la llamada a *"Realizar copia"*. La invocación al objeto de entidad es evidente para ambos casos, por lo que no es necesario explicar su semántica.

caso de estudio: servicio de cifrado remoto

Caso de uso “Encriptar”



Figura 3.10. Diagrama de robustez para el caso de uso “Encriptar” del cliente

En la figura 3.10 se muestra el diagrama de robustez para el caso de uso “*Encriptar*” del lado cliente del servicio de cifrado remoto. Es posible comprobar cómo el flujo de información se configura también como un patrón MVC, desde la selección del algoritmo por el actor “*Usuario*” en el objeto frontera *“Menu texto”*, pasando por los objetos de control *“Elegir algoritmo”* y “*Encriptar”*, hasta la obtención del objeto entidad *“Mensaje cifrado”*.

Caso de uso “Desencriptar”



Figura 3.11. Diagrama de robustez para el caso de uso “Desencriptar” del servidor

Tal como vemos en la figura 3.11, el diseño del caso de uso para desencriptar en el servidor también es un caso de patrón MVC. Observemos cómo se transita desde la activación del servicio en el objeto frontera *“Menu texto”* por el usuario avanzado, pasando por los objetos de control para detectar, desencriptar y filtrar el spam, hasta la obtención del objeto entidad con el mensaje descifrado.

modelo del dominio

«—Señor, una golondrina sola no hace verano».

(Miguel de Cervantes: El ingenioso hidalgo don Quijote de la Mancha, Parte 1, Capítulo 13).

Con la fase del modelo del dominio comenzamos a visualizar los primeros conceptos, asociaciones y atributos de los que consta nuestra aplicación. Para ello debemos realizar un esfuerzo de abstracción con el fin de reconocer las principales entidades en el terreno del dominio del problema. Con este modelado nos proveeremos de una potente herramienta para extraer conceptos que serán de gran utilidad en las posteriores fases del ciclo de vida, más concretamente en la fase de diseño detallado.

El trabajo fundamental en esta parte consistirá en centrarse únicamente en el dominio de la aplicación, dejando a un lado otros conceptos de más bajo nivel relacionados con los aspectos técnicos o del lenguaje de programación. Por este motivo, el resultado del *Modelado del Dominio* consistirá siempre en una visión de alto nivel, abstracta y general que recoja únicamente los aspectos relevantes de la literatura del programa.

extracción de conceptos

Puesto que el modelo del dominio contiene principalmente las entidades que conforman el problema, comenzaremos por tanto a distinguir cada uno de estos componentes mediante su identificación en el propio contexto de los requisitos.

Un concepto es una entidad o un elemento que puede ser tanto concreto como abstracto. Por ejemplo: *Venta, Cliente, Pedido*, etc. Los conceptos suelen estar dentro de las categorías de cosas, ideas, eventos, agentes, organizaciones y generalmente sustantivos significativos dentro del texto del enunciado del problema.

Para la identificación de los mismos se puede recurrir al propio texto de los requisitos; aunque lo común es encontrar que el lenguaje natural suele ser ambiguo y por tanto una frecuente fuente de errores. Generalmente y según [Larman02] la estrategia común para identificar conceptos consiste en la utilización de una lista de sustantivos o la identificación de frases nominales en las especificaciones de los casos de uso.

En la fase de identificación suele ser muy común omitir algún concepto y confundir los conceptos con los atributos. Por este motivo la mejor técnica para identificar las entidades principales del modelo del dominio consiste en aplicar una "visión global o de conjunto". Para ello es necesario realizar un esfuerzo de abstracción en el dominio del problema centrándonos en los conceptos que son propiamente del mismo y omitiendo los que puedan sugerirnos mayor nivel de detalle o complejidad.

Los conceptos los representaremos habitualmente por medio de un rectángulo dividido en dos partes: una superior para el nombre y la parte inferior para los atributos.

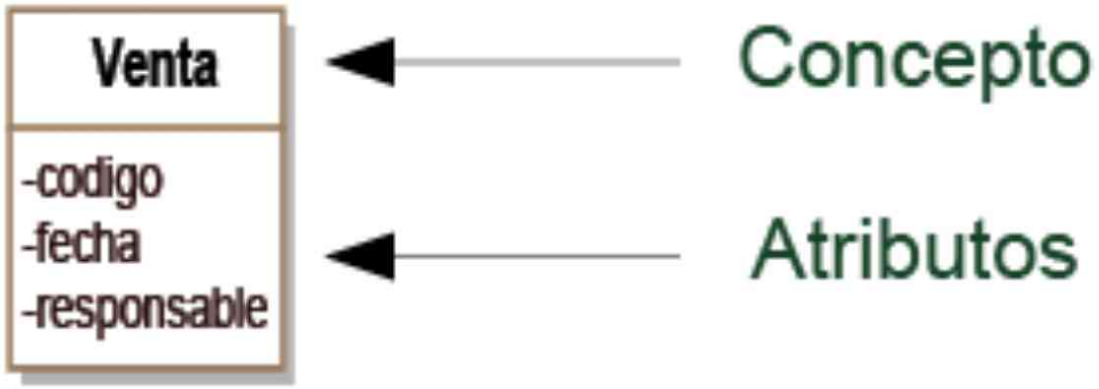


Figura 4.1. Concepto

identificar asociaciones

Previo paso a la extracción de atributos, nos encontramos con la necesidad de relacionar cada uno de los conceptos mediante enlaces, con la idea de unir acciones con entidades y/o partes continentes con sus respectivas partes de contenido. Típicamente las asociaciones indican relaciones semánticas entre conceptos.

Las asociaciones generalmente se representan mediante una línea continua que indica las dos partes relacionadas:

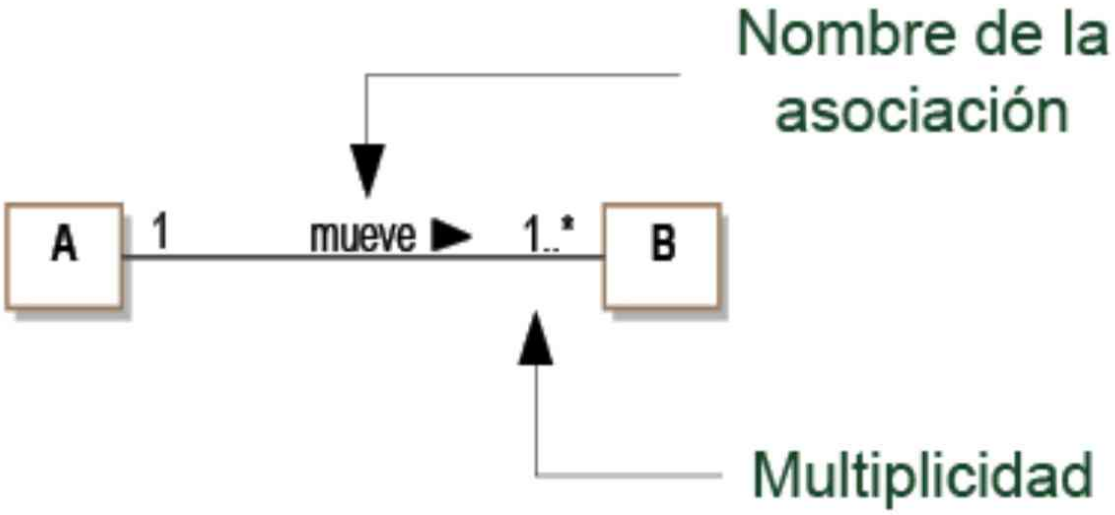


Figura 4.2. Asociación

En el ejemplo de la figura 4.2 se muestra la asociación de A con B mediante una línea que une ambos conceptos. Un triángulo relleno nos indica la dirección de lectura de la asociación, mientras que la línea continua nos indica la navegabilidad que es inherentemente bidireccional en este caso.

Otro aspecto importante en las asociaciones es la cardinalidad o multiplicidad, que indica las veces que una determinada instancia está relacionada o contenida con o en otras. Para representar multiplicidades recurriremos a la identificación en el texto de artículos indefinidos (un, una, unos, unas, muchos, muchas) y numerales (dos, cuatro, etc.). Así, en el ejemplo de la figura 4.2 se indica que la parte A está relacionada al menos una o más

veces con la parte B, es decir, la entidad A mueve una o muchas entidades B. Generalmente, las asociaciones suelen ser del tipo A contiene a B, A es una parte de B, A actúa con B ó A se comunica con B, etc.

En la siguiente tabla se muestra las diferentes configuraciones que pueden surgir en los distintos tipos de asociaciones:

Símbolo	Cardinalidad
*	Cero o más
1..*	Al menos uno o más
0..1	Cero o uno
1	Exactamente uno
4	Exactamente cuatro
1, 3, 5	Exactamente uno, tres o cinco

Tabla 4.1. Multiplicidades

Las notaciones habituales en diseño de software suelen recaer en las cuatro primeras entradas, sin embargo, UML permite un abanico mucho más amplio. Aunque las cardinalidades 0..* y * son ambas equivalentes y válidas, en diseño UML utilizaremos siempre la segunda opción.

establecer los atributos

Otra parte en el diseño del modelo del dominio es añadir los atributos o propiedades a las entidades anteriormente citadas. Por tanto, definiremos un *atributo* como propiedad descriptiva de un concepto o entidad.

Puesto que la idea principal es añadir atributos de forma que el modelo mantenga su consistencia y refleje fielmente la visión completa del dominio, debemos identificar aquellos atributos que cumplan con los requerimientos de la información proporcionada por el enunciado del problema. Es frecuente confundir en esta fase los atributos con los conceptos y por consiguiente asumir como atributo una entidad que se ajustaría mejor como un concepto y viceversa. Para evitar dicha confusión debemos recordar que los atributos siempre se ajustan mejor a tipos primitivos (*int*, *float*, *string*) o tipos de datos simples como *Vector3D*, *Coordenadas*, *Rectángulo*, *Dirección*, *Color*, *Códigos*, *etc.*

En la figura 4.3 se muestra un caso habitual de confusión entre atributo y concepto:

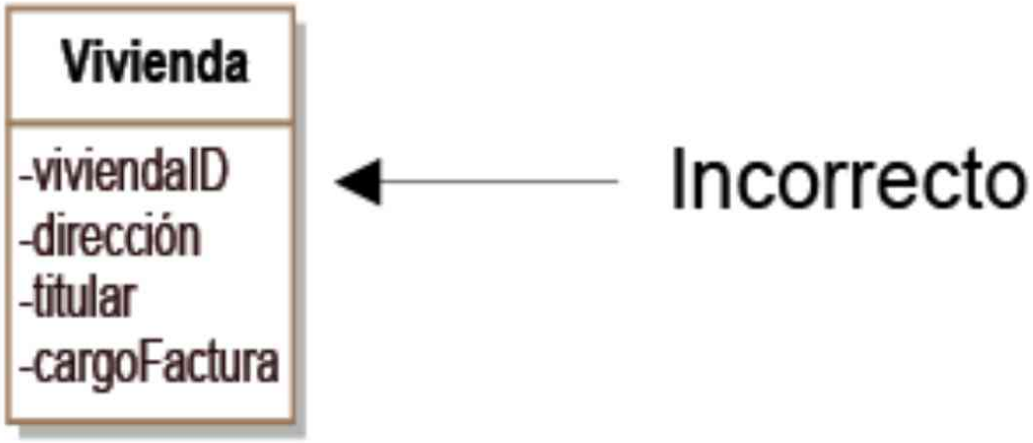




Figura 4.3. Un caso común de confusión

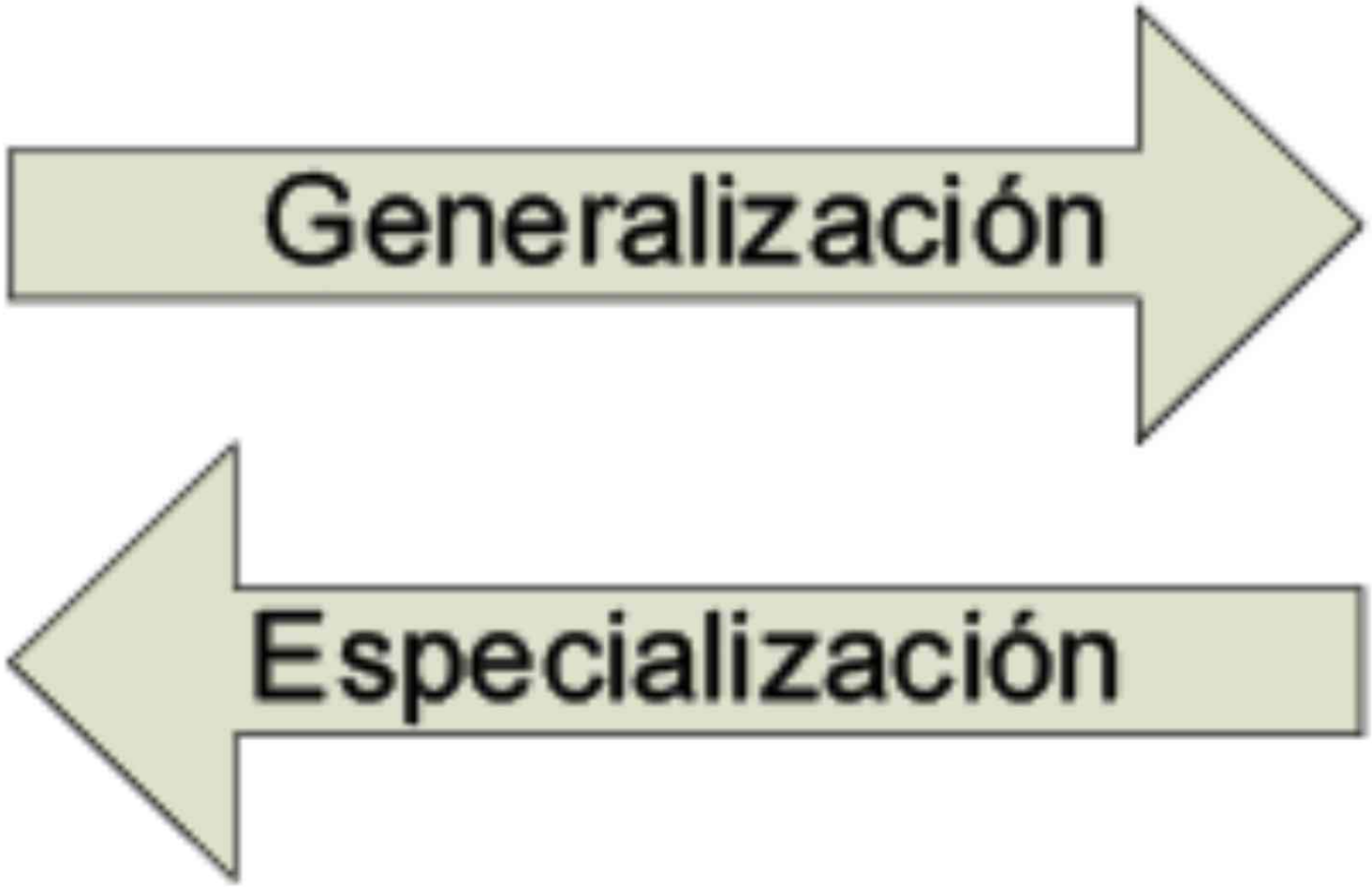
Cada *vivienda* tiene un *registro de luz* que se especificaría con un concepto nuevo en vez de un atributo, puesto que el registro de luz se adapta mejor a un concepto que a un atributo. El ejemplo anterior muestra que debemos siempre elegir el esquema de la parte inferior de la figura 4.3 cuando se nos presente esta situación; aunque en caso de duda, debemos representarlo siempre como un concepto antes que como un atributo.

Herencia

La generalización y especialización de conceptos en UML implica una relación entre entidades padre e hijo, donde la entidad hijo hereda[9] los atributos y métodos de su entidad padre. Las relaciones de herencia pueden ser múltiples, por lo que una entidad hijo puede heredar de varias entidades padre.

En UML la herencia se representa mediante un triángulo equilátero de color blanco:


Figura
Curva


Generalización
Especialización

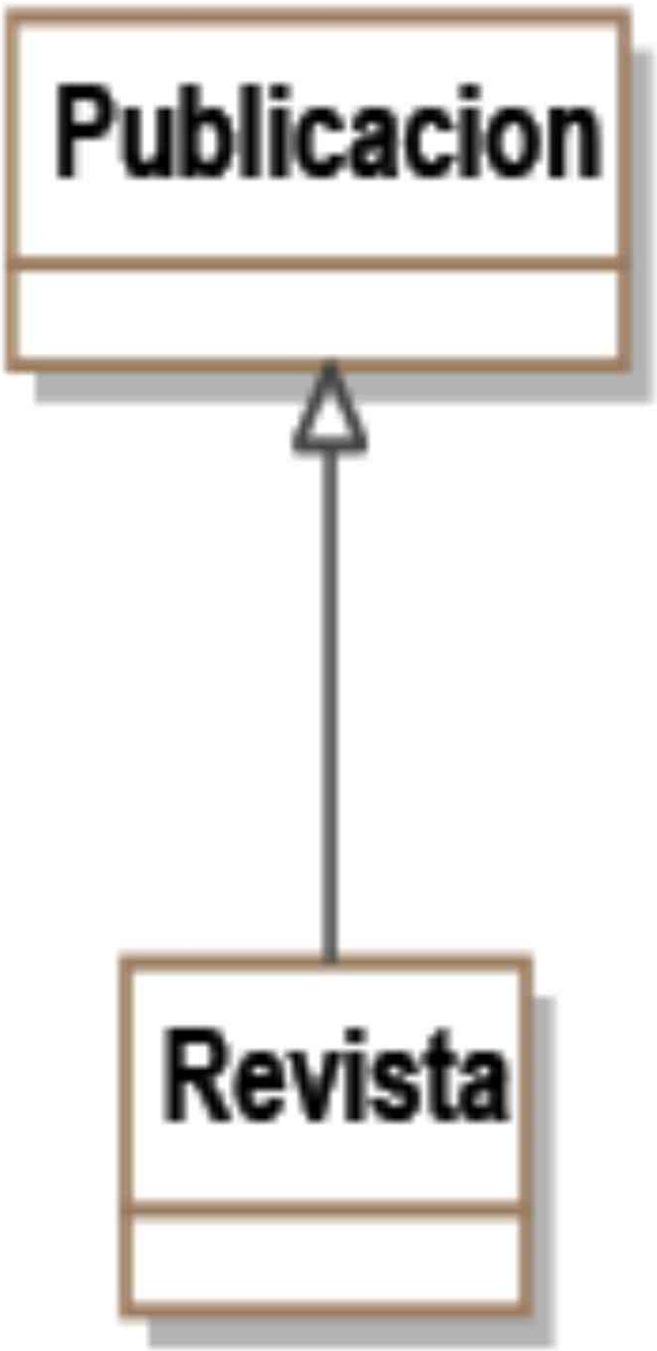


Figura 4.4. Ejemplo de herencia simple

En el ejemplo anterior una *Curva* es un tipo de *Figura* geométrica y hereda las propiedades de ésta. De igual forma ocurre con el concepto *Revista*.
Según se interprete el sentido de la herencia se hablará de *especialización* o *generalización*. Una relación de generalización se produce cuando se lee en sentido ascendente, mientras que una relación de especialización se lee en sentido descendente.

agregación y composición

Los últimos dos términos UML que necesitaremos para poder definir con más nitidez la relación entre los conceptos del domino son la agregación y la composición. Ambas notaciones indican igualmente que una entidad está formada por una o varias entidades que la determinan. El uso de una u otra dependerá de la semántica que se desee añadir al propio diagrama.

Para el caso de la *agregación,* el símbolo será un rombo o un diamante blanco tal como se muestra en la figura inferior:



Figura 4.5. Ejemplo de agregación

En este caso la agregación representa la parte que forma al todo pero sin la condición estricta de la inclusión de una de las partes. Por ejemplo, en el caso de de la figura 4.5 una casa puede tener o no objetos, pero el hecho de que falte un objeto no pone en evidencia el concepto de casa. Los objetos podrían llevarse o compartirse con otra vivienda y no poner en cuestión el concepto de casa.

De manera diferente ocurre con el símbolo de *composición* que se representa mediante un rombo o diamante negro:

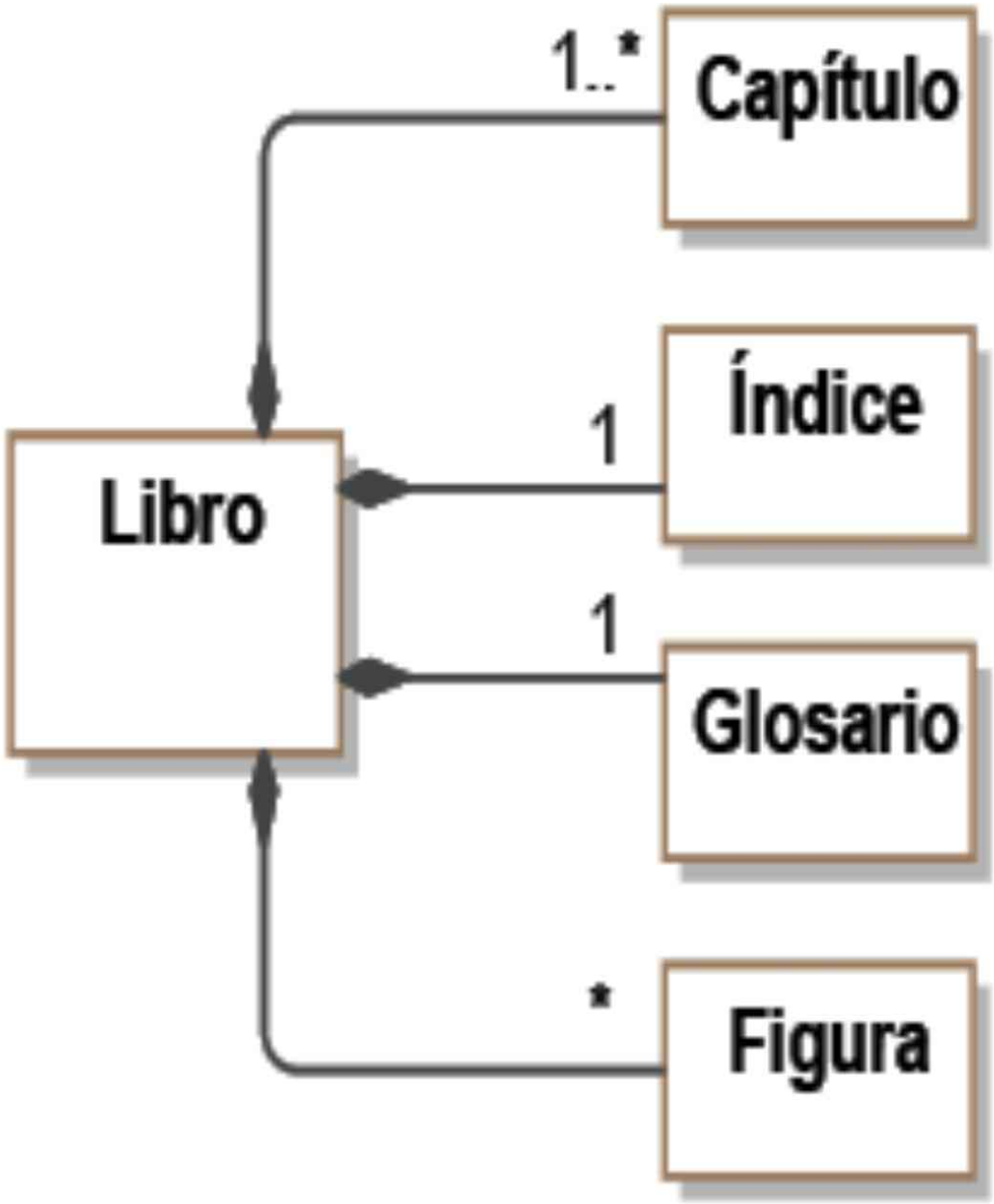


Figura 4.6. Ejemplo de composición

La diferencia fundamental de la *composición* con respecto a la agregación se basa en que todas las partes deben pertenecer estrictamente al todo para que éste tenga suficiente sentido dentro del contexto del dominio de la aplicación. En el ejemplo anterior, un libro debe estar formado por la posibilidad estricta de capítulos, un índice, un glosario y cero o más imágenes. De no ser así el

concepto libro quedaría incompleto dentro de la semántica que se le intenta aplicar. En general, en la composición, cuando la parte contenedora desaparece también desaparecen las partes contenidas, pues no tienen sentido sin ésta.

ejemplo de modelado: “spindizzy”

Proponemos ahora un caso sencillo de modelado del dominio donde se representen las principales notaciones explicadas con anterioridad y las funciones que desempeñan dentro del modelo.

Suponga el siguiente ejemplo extraído del dominio de los videojuegos:

«Un clásico juego en perspectiva isométrica de las plataformas de 8-bits consiste en mover una peonza por un laberinto formado por muchas habitaciones que contienen un conjunto de objetos similares. Cada habitación del laberinto conecta con al menos una habitación y a lo sumo con cuatro. Al comienzo del juego se selecciona una habitación al azar donde comenzará la aventura. Las habitaciones están formadas por unos bloques de ladrillos de varios tipos: de teletransporte a otra habitación del laberinto, movibles y desintegrables. En las habitaciones se encuentran varios objetos para recoger (diamantes y estrellas) para así completar la misión del juego. Además de los objetos recogibles, en la habitación pueden existir otros seres, como Torretas que se mantienen estáticas disparando al jugador; Robots y Octaedros que persiguen a la peonza a una determinada velocidad y alcance. Solo la Torreta puede disparar fuego o rayos, mientras que los Robots y los Octaedros persiguen a la peonza a una determinada velocidad y alcance. Todos los enemigos tienen un escudo de vulnerabilidad y también la peonza; pero esta última únicamente reduce en una unidad su escudo cuando recibe un disparo o golpe».



Figura 4.7. Vista del remake de Spindizzy GPLv3 (Sourceforge)

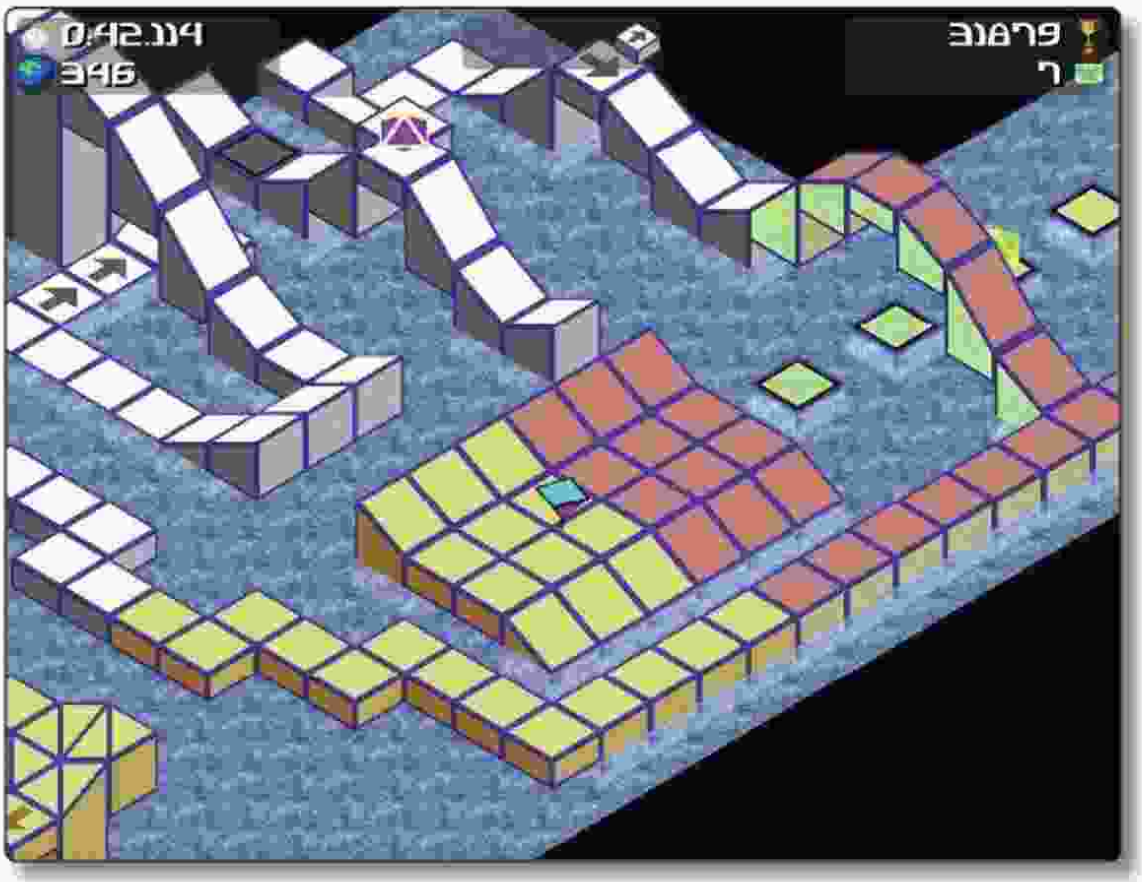


Figura 4.8. Vista del remake de Spindizzy GPLv3 (Sourceforge)

Con el enunciado y las imágenes proporcionadas anteriormente tenemos suficiente información para realizar el modelado del dominio (figura 4.9):

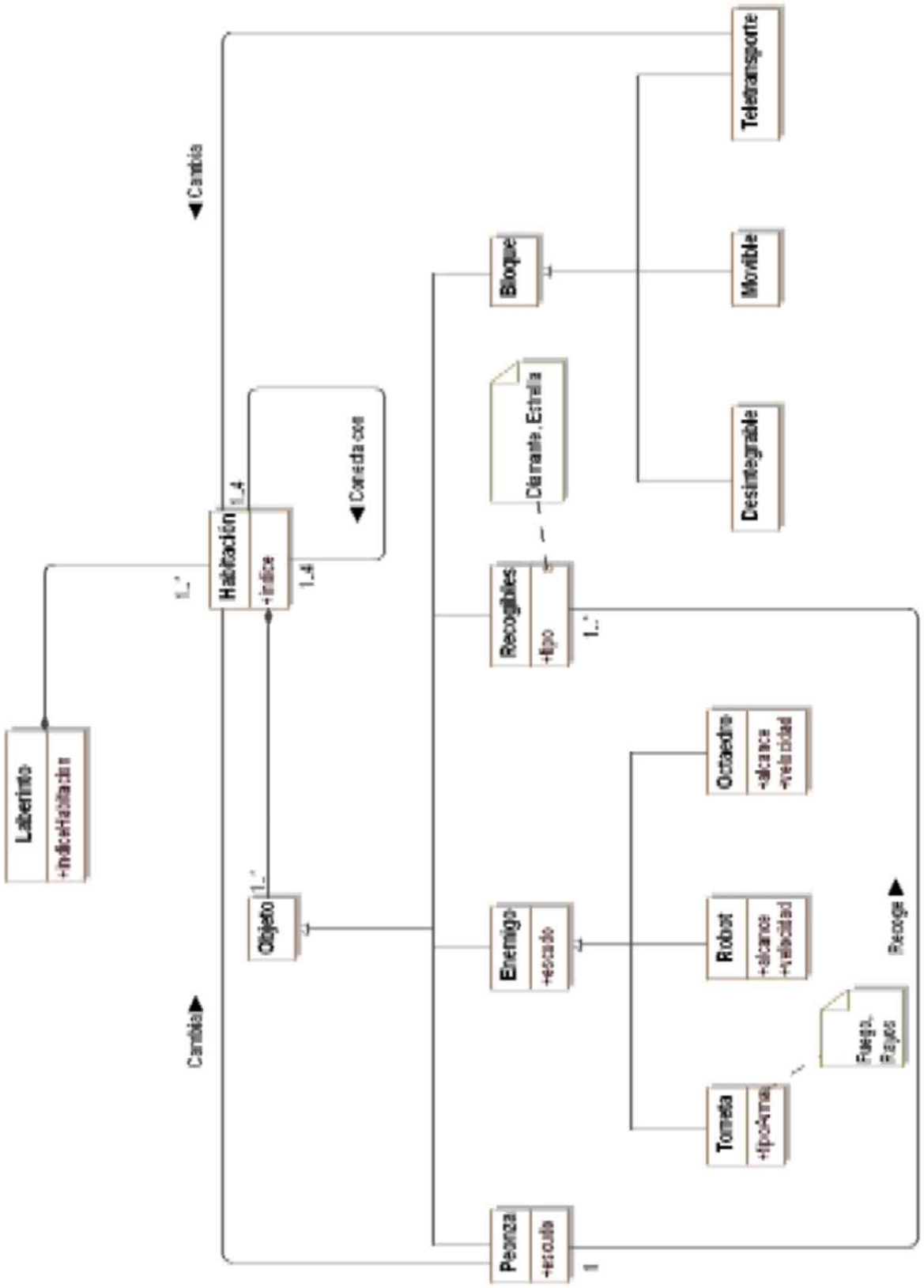


Figura 4.9. Modelo del dominio de "Spindizzy"

En el diagrama de la página anterior se representa el modelo del dominio

completo para el caso del juego "*Spindizzy*". En él se muestran las principales entidades significativas del terreno del problema propuesto, pero sin llegar a ser demasiado exhaustivos. Por este motivo, del texto del enunciado del problema se han seleccionado los sustantivos candidatos para la representación de las principales entidades y atributos conceptuales desde el punto de vista de la literatura de los requisitos.

Después de una segunda lectura más detenida se llegó a la conclusión de elegir los siguientes sinónimos:

Entidades seleccionadas	Atributos seleccionados
Laberinto Habitación Objeto Peonza Enemigo: Torreta, Robot y Octaedro Recogibles Bloque: Desintegrable, Movible y Teletransporte	Laberinto: *índiceHabitación* Habitación: *índice* Peonza: *escudo* Enemigo: *escudo* Torreta: *tipoArma* Robot: *alcance y velocidad* *Octaedro:* alcance y velocidad Recogibles: *tipo*

Tabla 4.2. Sinónimos seleccionados como entidades o atributos

Por ejemplo, para el caso del nombre *Peonza* se ajusta mejor a un concepto, mientras que los sustantivos *alcance* y *velocidad* se adaptan mejor a un tipo de dato primitivo, debido a su simplicidad y a la lógica de conformación de las entidades *Robot* y *Octaedro*.

Por otro lado, con vistas a la identificación de *asociaciones* será de utilidad las estructuras verbales, tales como: tiene, posee, conecta, recoge, etc. En una lectura pausada del texto es fácil detectar dichas relaciones semánticas a través de proposiciones como:

- *"En las habitaciones se encuentran varios objetos para recoger (diamantes*

y estrellas) para así completar la misión del juego...."

- *"Cada habitación del laberinto conecta con al menos una habitación y a lo sumo con cuatro..."*

Para completar la semántica de la notación asociativa faltaría añadir las *multiplicidades*. Para tal propósito recurriremos de nuevo al texto del enunciado para identificar artículos indefinidos y numerales. De esta forma encontramos, por ejemplo, que:

- *"Cada habitación del laberinto conecta con al menos **una** habitación y **a lo sumo cuatro**..."*
- *"En las habitaciones se encuentran **varios** objetos para recoger..."*

En el primer caso se especifica que una habitación conecta al menos con "una" habitación y un máximo de "cuatro", mientras que en el segundo ejemplo indica que se pueden recoger "varios" objetos, lógicamente por el protagonista (*Peonza*).

Dentro de la *composición* entre entidades observamos las siguientes proposiciones:

- *"consiste en mover una peonza por **un laberinto formado por muchas habitaciones**...*
- ***...que contienen un conjunto de objetos** similares..."*

Donde se sobrentiende que dicha inclusión es prioritaria para la semántica conceptual del *Laberinto* y de las *Habitaciones*. De igual forma, se extrae de la sintaxis de la oración la cardinalidad de dichas asociaciones (*muchas* y *un conjunto*) para ambas entidades respectivamente.

Finalmente se identifican las *herencias*, donde se observa claramente el

siguiente caso:

- *"Las habitaciones están formadas por unos* ***bloques*** *de ladrillos* ***devarios tipos****: de teletransporte a otra habitación del laberinto, movibles y desintegrables..."*

Aunque la sintaxis global del texto es clara para identificar sustantivos, verbos y artículos, también es posible extrapolar o intuir el significado general de las oraciones para identificar jerarquías y otros elementos del modelo, aún cuando no existan estructuras sintácticas evidentes en el lenguaje para su identificación. Así, en el caso de la jerarquía de *Objetos* del juego se puede extraer de la semántica del texto su existencia, como es el caso de la derivación para *Peonza*, *Enemigo*, *Recogibles* y *Bloque*. La misma regla intuitiva se puede aplicar para la jerarquía de *Enemigos*.

caso de estudio: ajedrez

Veamos a continuación la construcción del modelo del dominio para el juego de ajedrez. En primer lugar procederemos a la identificación de las entidades conceptuales que representarán las principales partes del modelo. Para este propósito las especificaciones de los casos de uso vistos en el capítulo dos serán de una magnífica utilidad para la identificación de conceptos.

Entidades seleccionadas	Atributos seleccionados
Jugador	Jugador: *color*
IA	IA: *nivel*
JugadorHumano	JugadorHumano: *nombre y password*
Jugada	Jugada: *fila y columna*
Pieza	Pieza: *posición* y *tipo*
Tablero	
ControlJuego	
Algoritmo	
Modelo	
Material	

Tabla 4.3. Entidades y atributos para el juego de Ajedrez

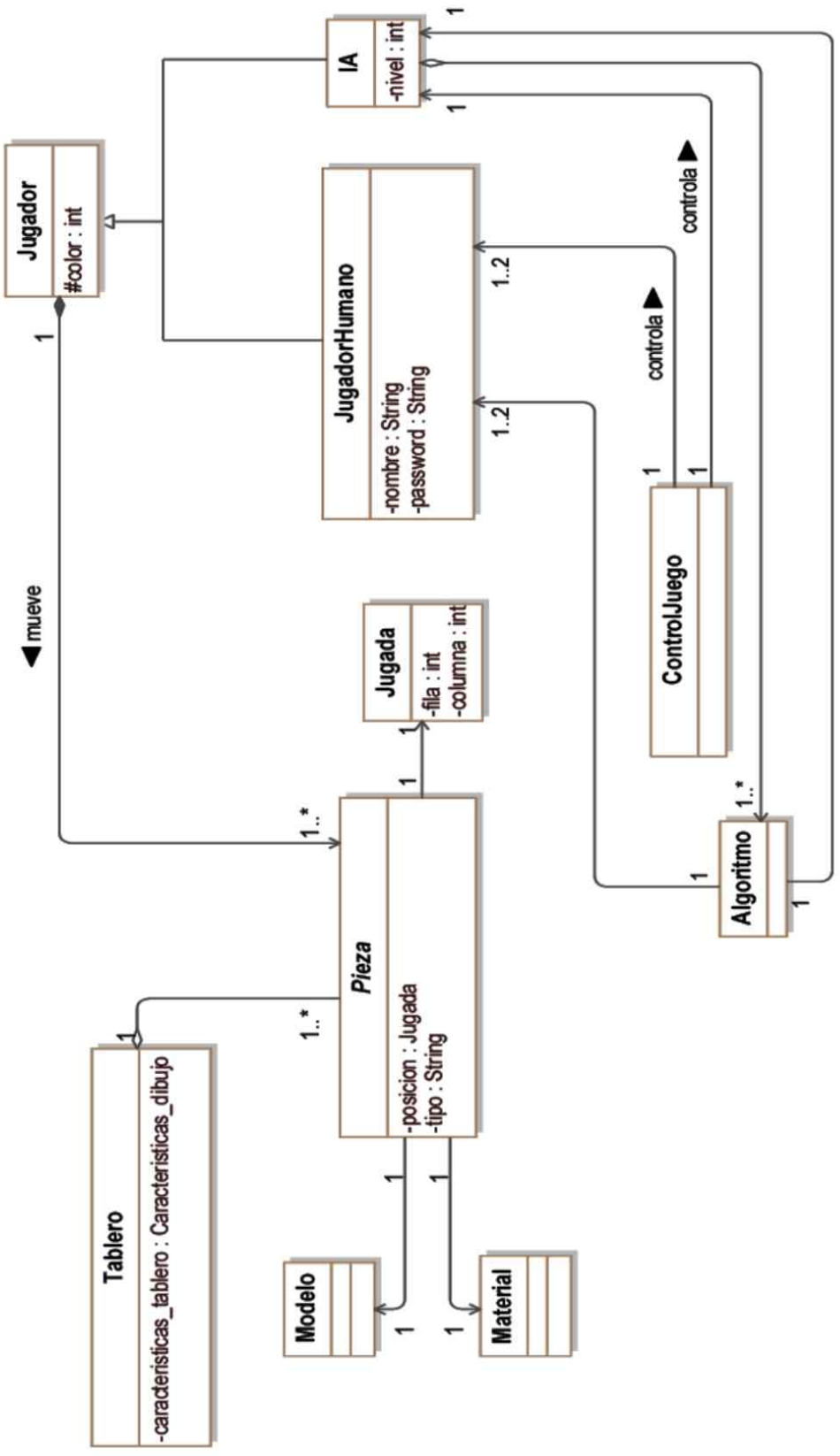


Figura 4.10. Modelo del dominio para Ajedrez

Como se aprecia en el modelo de la figura 4.10, las principales entidades

extraídas de las especificaciones de los casos de uso se representan mediante la notación UML de clase junto con el grupo de atributos identificados para cada concepto. En este modelo ya se anticipan los núcleos que conformarán los principales subsistemas de la aplicación. Uno de los principales núcleos es la entidad *Tablero* que se asocia con un conjunto de piezas de un jugador (IA o humano). La entidad *Jugador* contiene una composición de uno-a-muchos con la entidad *Pieza* que mantiene el conjunto de las piezas del ajedrez. La entidad *Pieza* tiene una referencia a la entidad *Jugada* que alberga la estructura de datos con la posición (fila, columna) de una pieza de ajedrez. Bajo la entidad generalizada *Jugador* se especializan *JugadorHumano* e *IA*, que son respectivamente el actor o actores humanos (a lo sumo dos jugadores) y la computadora. La IA se relaciona con los algoritmos que permiten aplicar estrategias de jugadas (*minimax*, *alfa-beta*) y que se representan mediante el concepto *Algoritmo*. Finalmente, la entidad *ControlJuego* será el corazón del sistema al encargarse de centralizar y gestionar la lógica del juego del ajedrez.

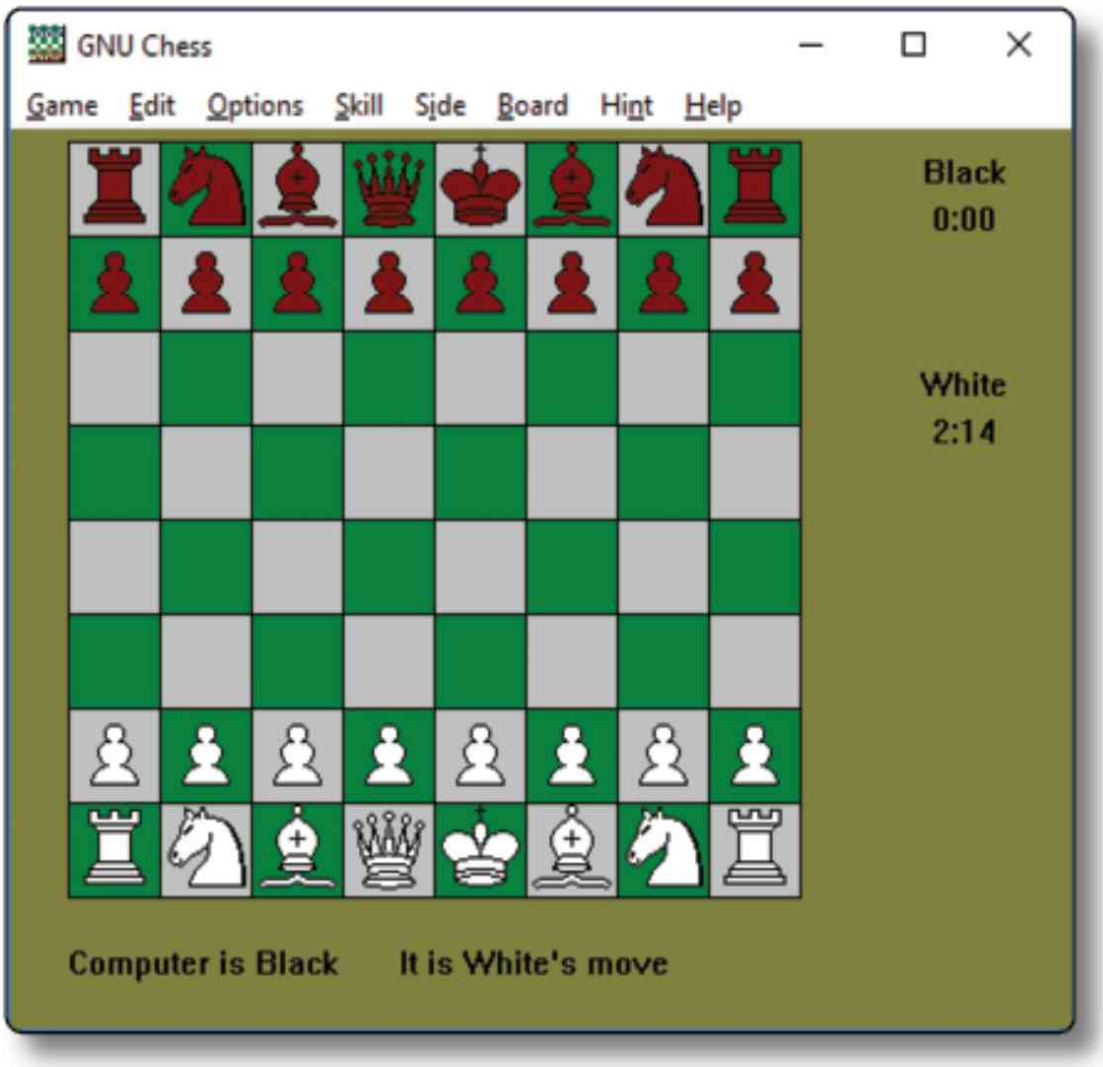


Figura 4.11. GNU Chess (versión equivalente a la implementada en este libro)

caso de estudio: mercurial

Para el caso *Mercurial* nos encontramos con una situación parecida. Las especificaciones de casos de uso son una buena fuente para identificar las entidades y atributos a modelar.

Para ello elaboraremos la siguiente tabla con conceptos propios del terreno y sin bajar a nivel de software:

Entidades seleccionadas	Atributos seleccionados
Usuario ClienteProgramador Administrador Directorio Fichero	Usuario: *password y login* Directorio: *nombre* Fichero: *nombre*

Tabla 4.4. Entidades y atributos para la aplicación "Mercurial"

En la tabla 4.4 hemos identificado las principales entidades que conformarán la aplicación CVS Mercurial. Principalmente encontramos la jerarquía de usuarios: *ClienteProgramador* y *Administrador*, que representan los dos actores principales vistos en la especificación del caso de uso "*Listar ficheros*". De la lectura de la especificación identificamos como atributos clave el *login* y el *password* para registro en el servidor y que formarán parte de la entidad base *Usuario*. De igual forma se representa mediante asociación la relación uno-a-muchos existente entre un programador y su jerarquía de ficheros, representada aquí mediante una asociación reflexiva. Esta asociación reflexiva invita a la definición de una estructura de datos recursiva para albergar la jerarquía de directorios y ficheros. Como veremos en el capítulo nueve, esta estructura será identificada posteriormente como un patrón de diseño.

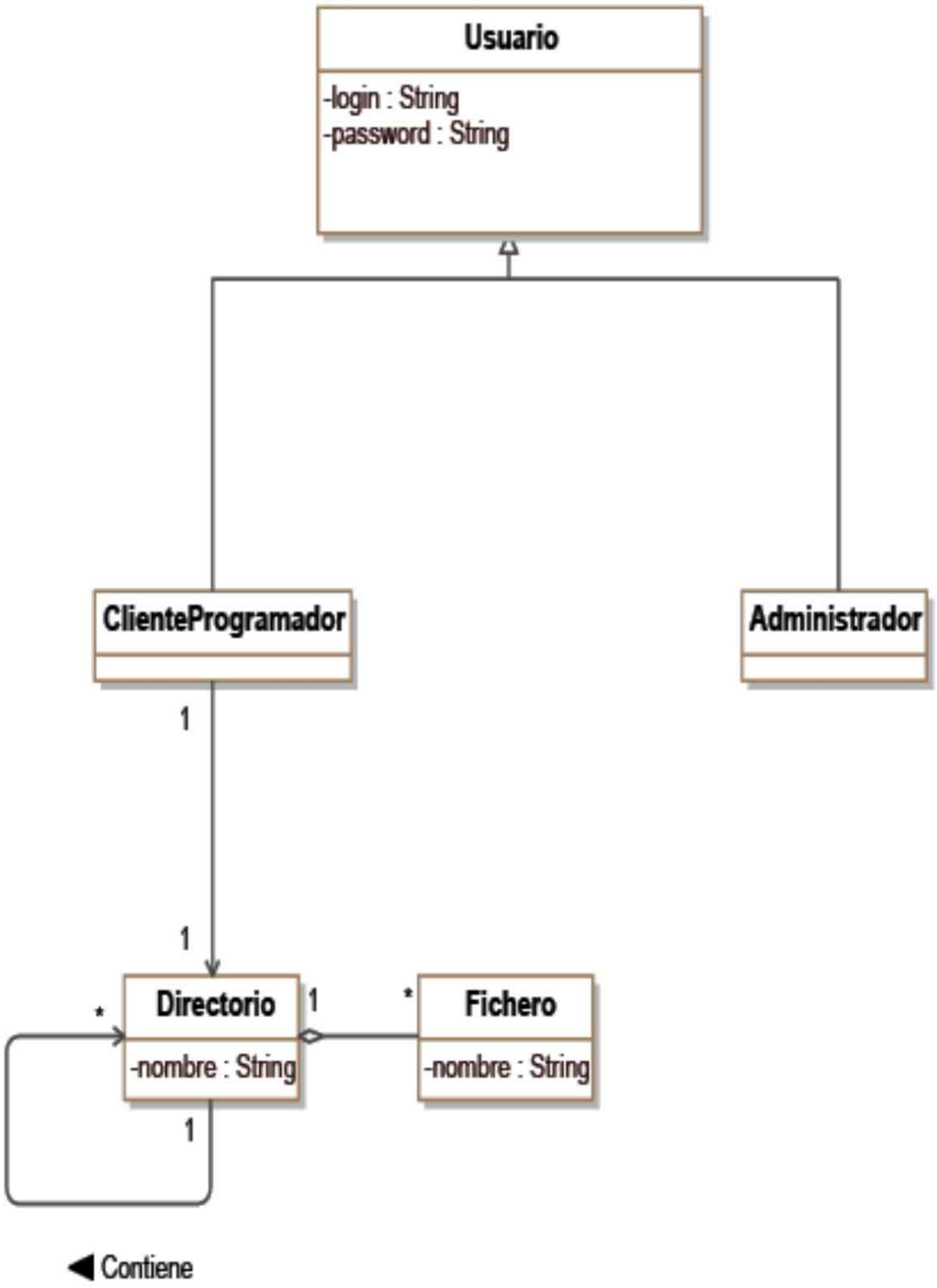


Figura 4.12. Modelo del dominio para "Mercurial"

Hasta aquí nos hemos limitado a modelar una versión preliminar de los diagramas conceptuales del dominio ciñéndonos a la visión inicial que tenemos

del problema. Sin un análisis más detenido no podemos obtener aún un modelo más detallado, puesto que a partir de los datos proporcionados durante el análisis de requisitos no es posible llegar aún al modelo de clases. Para un diseño más detallado debemos realizar una previsión de la arquitectura así como de las principales funcionalidades y concreciones del sistema.

caso de estudio: servicio de cifrado remoto

Mediante el análisis de los casos de uso y los requerimientos explicados en la sección 2.8, somos capaces ya de obtener una síntesis abstracta de las principales entidades y atributos del dominio separadas para el caso del cliente y del servidor.

Cliente

Entidades seleccionadas	Atributos seleccionados
Usuario Cifrador CifradorSimetrico CifradorAsimetrico	Usuario: *numero de accesos, nombre, login y password.*

Tabla 4.5. Entidades y atributos para el servicio de cifrado remoto (lado cliente)

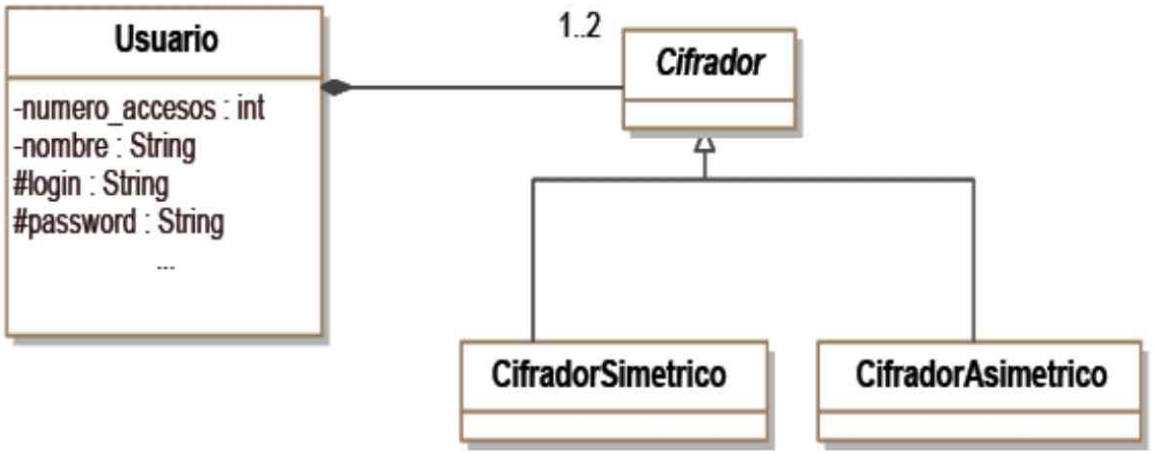


Figura 4.13. Modelo del dominio para el cliente

En el diagrama de la figura 4.13 no se han detallado las características técnicas de otros componentes como la interfaz de usuario textual, los métodos algorítmicos específicos o el componente de comunicaciones, permaneciendo a un nivel de abstracción elevado y de visión global.

Servidor

Entidades seleccionadas	Atributos seleccionados
Usuario UsuarioAvanzado Cifrador CifradorSimetrico Cifrador Asimetrico Descifrador DescifradorSimetrico DescifradoAsimetrico	Usuario: *número de accesos, nombre, login* y *password*. UsuarioAvanzado: *rango* y *nombre simbólico*.

Tabla 4.6. Entidades y atributos para el servicio de cifrado remoto (lado del servidor)

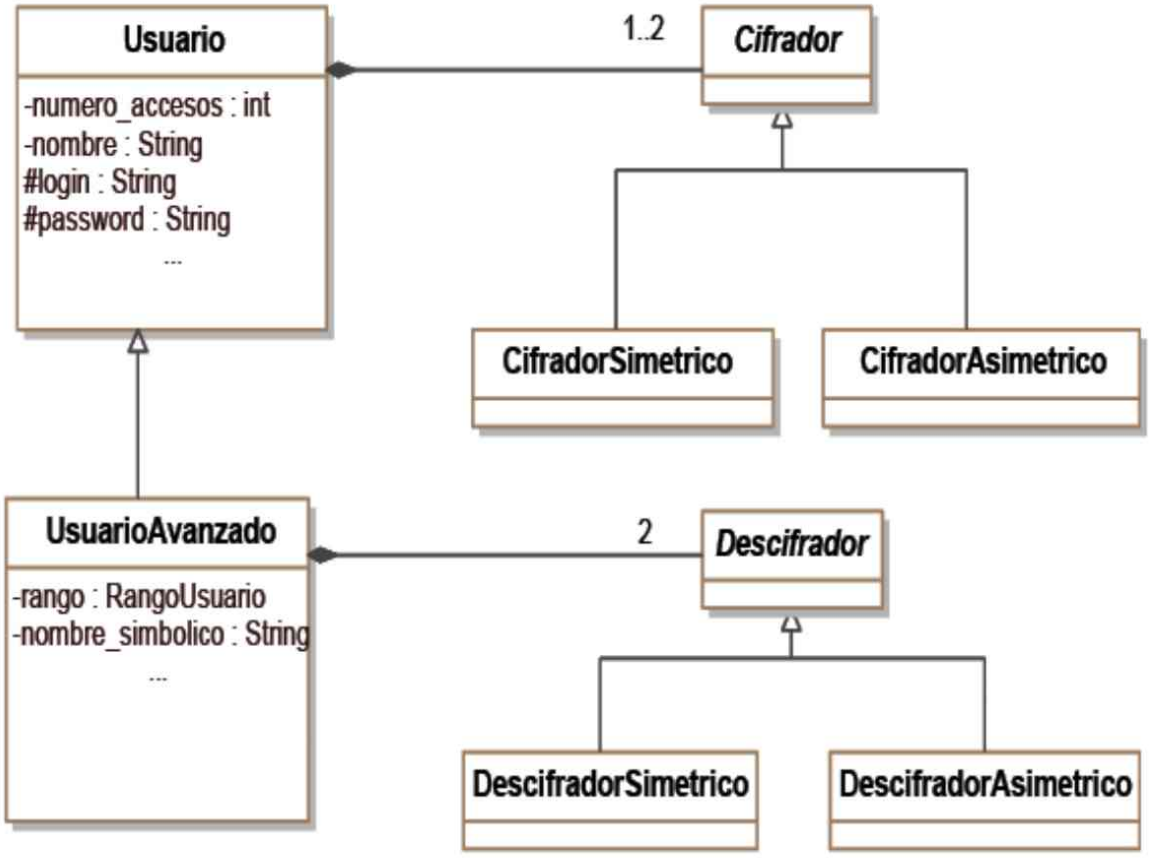


Figura 4.14. Modelo del dominio para el servidor

De igual modo procedemos a analizar los casos de uso del servidor vistos en

la sección 2.8 para extraer de ellos la literatura del terreno sin entrar en detalles meramente técnicos. Como podemos apreciar, en la tabla 4.6 se han enumerado, a la izquierda, el conjunto de entidades que comprende el lado servidor y se ha ampliado el nivel ontológico mediante la inclusión por herencia de la entidad *UsuarioAvanzado*. A la derecha observamos los atributos seleccionados preliminarmente a este nivel de abstracción para los dos tipos de usuario. Finalmente, en la figura 4.14 se modela el dominio del problema que describe, a grandes rasgos, la aplicación que implementará el servidor. Puesto que en la composición entre *Usuario* y *Cifrador* se puede optar por un tipo de algoritmo u otro en el momento de enviar el mensaje, se ha decidido fijar la cardinalidad de la asociación en 1..2 dependiendo de si se utiliza un cifrado simétrico o uno híbrido, ya que este último requiere de ambas clases. Sin embargo, respecto a la composición entre *UsuarioAvanzado* y *Descifrador* se crearán dos objetos del tipo *DescifradorSimetrico* y *DescifradorAsimetrico* respectivamente, con el fin de proceder al descifrado del mensaje probando ambos algoritmos.

9. En lenguajes como C++ pueden restringirse la herencia de atributos y métodos.

diagramas orientados a la arquitectura

«Sé firme como una torre, cuya cúspide no se doblega jamás al embate de los tiempos».

(Dante: La Divina Comedia: Purgatorio, Canto V).

Después de la fase de requisitos y de análisis nos encontramos en un hito singular del ciclo de vida del proyecto. Esta nueva fase la denominaremos la fase de *diseño arquitectónico*. La palabra arquitectura puede traernos a la memoria el proceso de diseño de un edificio. Sin ir mucho más lejos, la arquitectura de software tiene muchos aspectos en común con la construcción de un edificio, pero con la diferencia principal de que el software no es un objeto físico. De igual forma que se diseña un edificio con unas determinadas características habitables, estéticas y coherentes con el resto de los edificios del entorno, la arquitectura del software debe de responder a los mismos principios de diseño. En general cuando se diseña software a alto nivel deben cuidarse aspectos de coherencia con la plataforma, el lenguaje, el rendimiento, la seguridad, la eficiencia e incluso aspectos meramente estéticos. Según *Bass* y sus colegas «La arquitectura de software de un sistema de programa o computación es la estructura de las estructuras del sistema, la cual comprende los componentes del software, la propiedades de esos componentes visibles externamente, y las relaciones entre ellos».[10]

taxonomía de estilos arquitectónicos

A medida que la complejidad del software aumenta, los algoritmos y las estructuras de datos no dan respuesta a los nuevos problemas que emergen del diseño. ¿Qué hacer entonces cuando nuestro programa se compone de una gran cantidad de módulos, estructuras de datos y algoritmos? Obviamente surge la necesidad urgente de dar respuesta a esta pregunta.

Como explicábamos en la introducción de este capítulo, la arquitectura del software tiene puntos de similitud con la arquitectura de un edificio. De esta forma, y a lo largo de la historia de la arquitectura e ingeniería, la respuesta ha sido siempre la misma: "no volver a inventar la rueda".

Puesto que la arquitectura es esencial para el éxito del diseño de un sistema y representa una visión de alto nivel de los componentes principales que integran el conjunto de la aplicación y sus relaciones, propondremos a continuación una lista de las principales arquitecturas software existentes hasta la fecha. Dichas arquitecturas representan la sabiduría acumulada en el campo de la IS desde los comienzos de la programación y su evolución hacia los lenguajes de alto nivel y los tipos abstractos de datos. Por ende, la arquitectura se podría definir como la respuesta de nivel global sobre el conjunto de componentes que interactúan en un sistema y su organización lógica a modo de patrón prediseñado.

A continuación se presentarán resumidamente las arquitecturas más relevantes en el diseño de software que comprenden la mayor parte de aplicaciones actuales. La organización y distribución de sus partes componentes son la pieza clave para su clasificación, así como la orientación a cada tipo de desarrollo específico. Los modelos aquí expuestos son completamente diferentes entre sí y le permitirán deducir la elección del tipo de arquitectura que mejor se adecua a su proyecto.

Tuberías y filtros

Los componentes de las arquitecturas formadas por tuberías y filtros tienen una serie de entradas y salidas. De esta forma a los componentes se les llama "filtros", mientras que los conectores se les denomina "tuberías". La información fluye por las tuberías y es procesada dentro de los filtros que generan la salida por la correspondiente tubería. Ejemplos de esta arquitectura son los programas escritos con las tuberías del *shell* de UNIX y los compiladores.

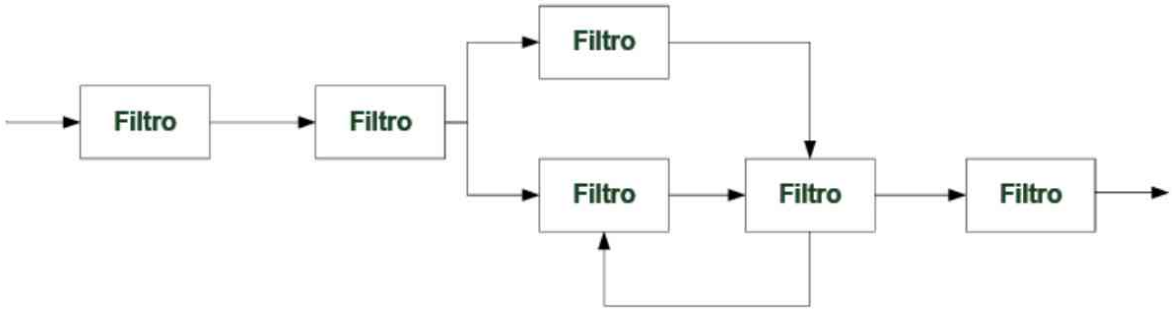


Figura 5.1. Arquitectura de tuberías y filtros

Por capas

Estas arquitecturas son frecuentemente usadas por variedad de aplicaciones empresariales. Se componen de un conjunto de capas superpuestas en la que la capa inferior proporciona un servicio a la capa superior. La capa superior se comporta como cliente de las funciones provistas por la capa inmediatamente inferior. Para la comunicación entre las dos capas, la capa inferior proporciona una interfaz donde se especifican los servicios que ofrece.

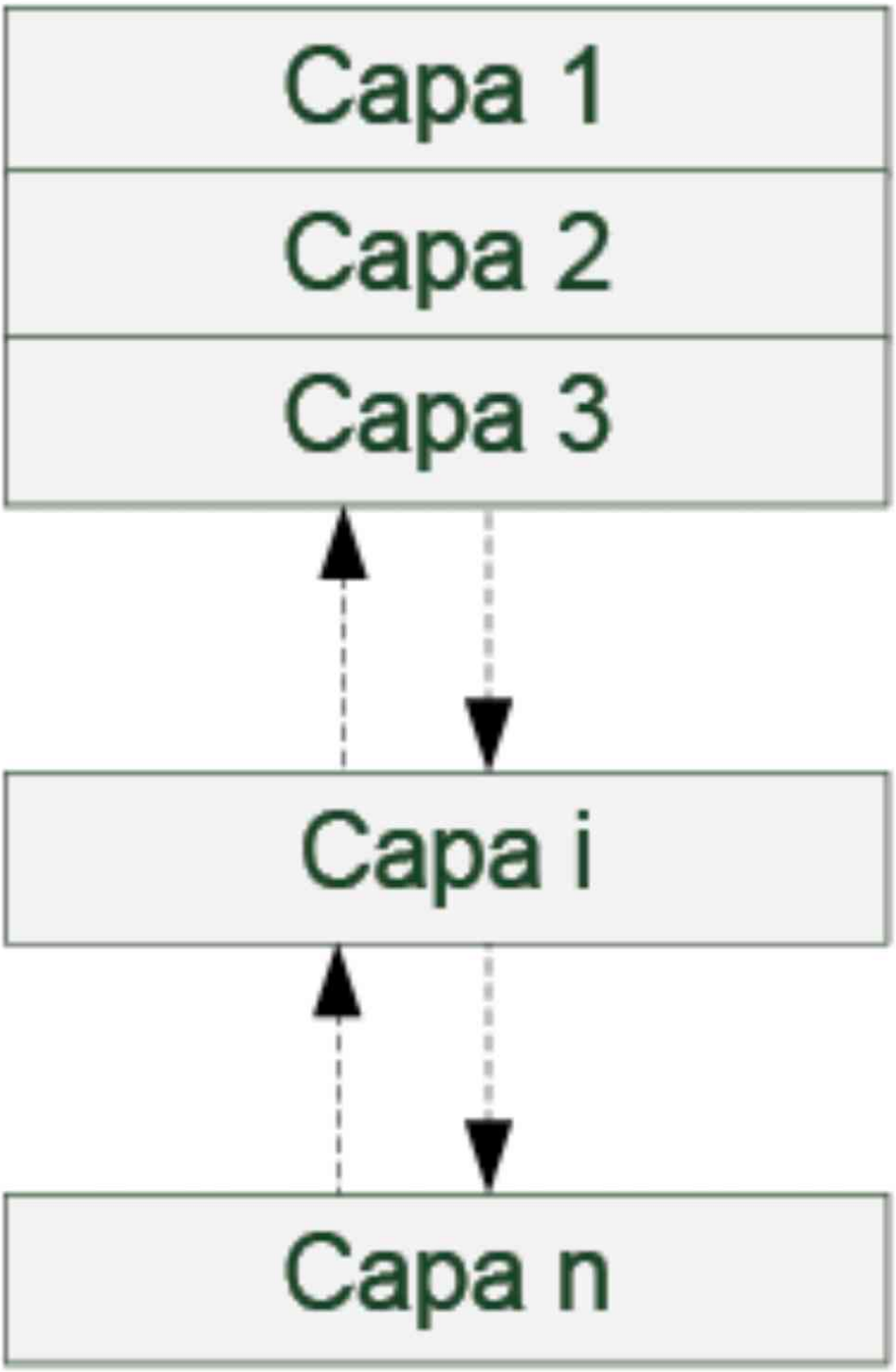


Figura 5.2. Arquitectura basada en capas

Ejemplos de esta arquitectura son los sistemas operativos, las aplicaciones empresariales multicapa (JEE, SAP ERP, etc.), las bases de datos y la pila de protocolos TCP/IP o ATM.

Repositorios

La arquitectura de repositorios consiste en dos componentes principales: uno central llamado "pizarra" que mantiene el estado general y los datos compartidos, y las "fuentes de conocimiento" que interactúan con los datos centralizados de la pizarra.

Las fuentes de conocimiento nunca interactúan entre ellas, comunicándose únicamente a través de la estructura de pizarra centralizada. Las fuentes de conocimiento actúan cuando se produce un cambio en la estructura de datos centralizada de la pizarra.

Un ejemplo de aplicación de esta arquitectura es el procesado de señal y el reconocimiento de patrones.

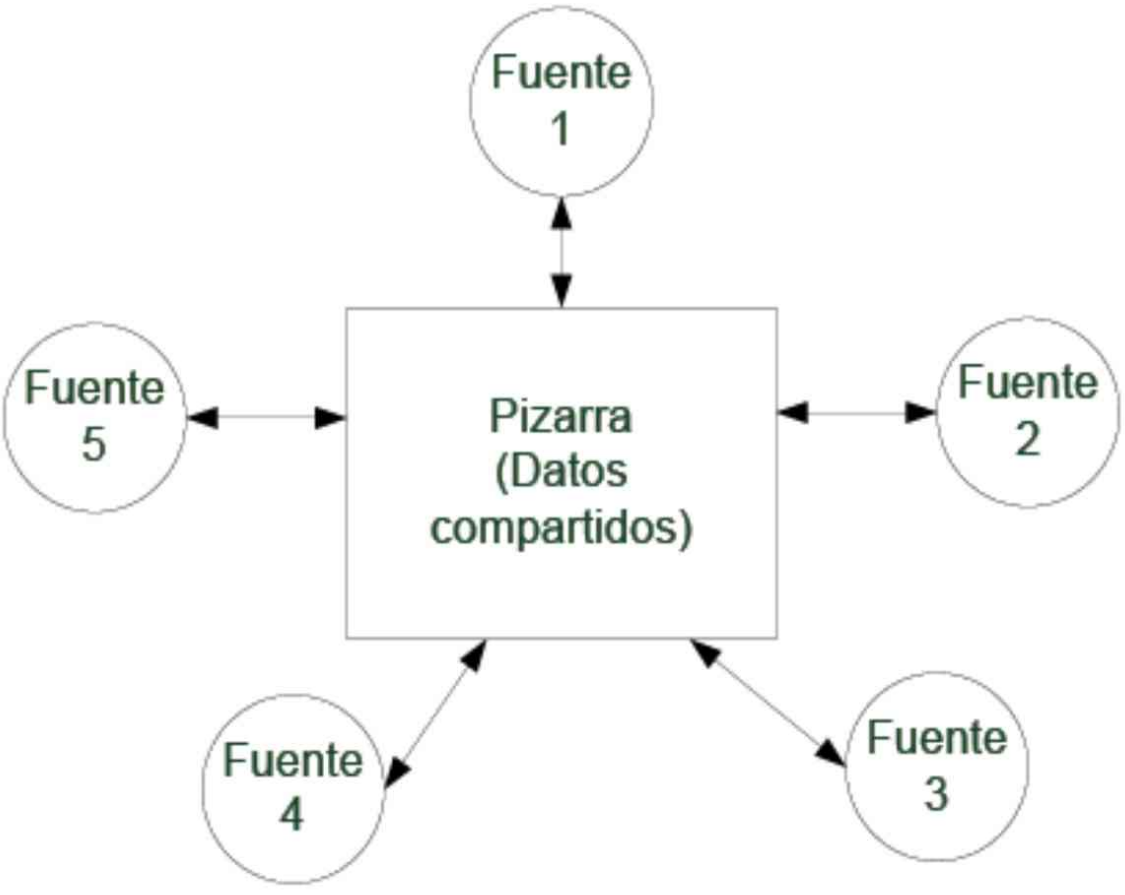


Figura 5.3. Arquitectura de repositorio

Intérprete

La arquitectura de intérprete se utiliza para implementar una máquina virtual en software. Las partes componentes de estar arquitectura son:

- La memoria con el código del programa a ser interpretado.
- El motor de interpretación o intérprete.
- Un registro del estado del motor de interpretación.
- Un registro del estado actual de ejecución del programa que está siendo simulado.

El motor de interpretación es el encargado de mantener el estado de ejecución del programa en memoria, así como su registro de activación.
Ejemplos de estar arquitecturas son VMware o la máquina virtual de Java (JVM).

Basadas en eventos

Tradicionalmente las interfaces de los componentes proporcionan una serie de procedimientos y funciones que son invocados explícitamente; no obstante, resulta también útil la invocación implícita en muchos otros tipos de sistemas. La invocación implícita no consiste en llamar directamente a un procedimiento o función como lo haría la invocación explícita, sino que los componentes registran su interés en un evento asociándolo con un procedimiento. Así, por ejemplo, si una ventana de una interfaz de usuario necesita actuar de acuerdo con la pulsación de uno de sus botones o el movimiento del ratón sobre su superficie, puede registrar su interés en dichos eventos mediante un método específico. Cuando dicho evento de produce, se realizará una invocación implícita a los procedimientos o métodos en el módulo donde se gestionan los eventos de la ventana. Igualmente, la invocación implícita puede complementarse con la invocación explícita en un sistema basado en eventos. Una de las ventajas del mecanismo de la invocación implícita es su gran reutilización. Otra ventaja es la facilidad para la evolución del sistema, ya que los componentes pueden ser reemplazados por otros sin afectar a los interfaces de otros componentes en el sistema. Sin embargo, la principal desventaja tiene lugar cuando un componente anuncia un evento, pero desconoce qué otros componentes responderán a este, ni sabe su identidad ni su orden de llamada. También desconocerá cuándo finalizarán los eventos. Otro problema es el intercambio de datos, ya que algunas veces los datos son pasados con el evento, pero en otras ocasiones, suelen utilizarse repositorios compartidos. Esto puede acarrear un problema de rendimiento global y de gestión de recursos.

Las arquitecturas basadas en eventos son muy utilizadas en sistemas distribuidos, en interfaces gráficas de usuario para separar la presentación del modelo de negocio, en sistemas gestores de bases de datos (SGBD), etc.

[Shaw93].

Distribuidas

Las arquitecturas distribuidas suelen ejecutarse en procesos ubicados en redes de ordenadores, por lo que las topologías básicas son en estrella, en bus, en anillo, etc. Unos de los subtipos de sistema distribuido más comúnmente utilizado en la industria es la arquitectura *cliente-servidor*, en cuyos extremos de la red se encuentra un proceso servidor que ofrece una serie de servicios a otro proceso cliente que los solicita desde otra ubicación.



Figura 5.4. Arquitectura cliente-servidor

Otras variantes de arquitecturas distribuidas más sofisticadas dentro de esta clase son las arquitecturas P2P (*Peer-to-Peer*), de servidores Proxy, de múltiples servidores, etc.

Programa principal / subprograma

En esta categoría residen las arquitecturas basadas en la estructura clásica de programación, donde un conjunto de componentes (procedimientos y funciones) interactúan entre ellos para realizar una tarea, y un programa –principal– invoca a un número determinado de los componentes citados anteriormente.

diagramas de componentes

«Un *componente* se define como una parte modular, reemplazable y significativa del sistema que empaqueta una implementación y expone una interfaz».[11] Los *diagramas de componentes* permiten tener una visión estática y arquitectónica de los componentes software utilizados en la aplicación. Un componente es una entidad software que abstrae una funcionalidad bajo los principios de la encapsulación, ocultación, modularidad y la reutilización propios de la programación orientada a objetos. Dicho componente expone su funcionalidad mediante las interfaces, que son los puntos de interconexión con otros componentes del sistema.

Por componente software entendemos los componentes *lógicos* tales como componentes de negocio, componentes de proceso, etc., y componentes *físicos* como las tecnologías EJB, CORBA, COM+, .NET y WSDL.

Para comprender mejor la idea de la organización arquitectónica del diagrama de componentes es necesario tener una leve noción del concepto de *interfaz*.

Interfaces

Una interfaz define un conjunto de métodos sin cuerpo que debe implementar una clase. La utilización de interfaces permite el aumento de la abstracción con respecto a la implementación posterior de la clase hija. Los lenguajes de cuarta generación como Java proporcionan facilidades para la implementación de interfaces. En el caso de C++, las interfaces se definen mediante clases abstractas. Obviamente, las interfaces no pueden ser instanciadas como ocurre con las clases normales. Esta característica de las interfaces facilita la posibilidad de mejorar o cambiar, en futuras versiones, la implementación interna de la clase o subsistema sin la necesidad de cambiar la especificación externa.

Componentes

En el diagrama de componentes UML el símbolo de componente se representa como se ilustra a continuación:

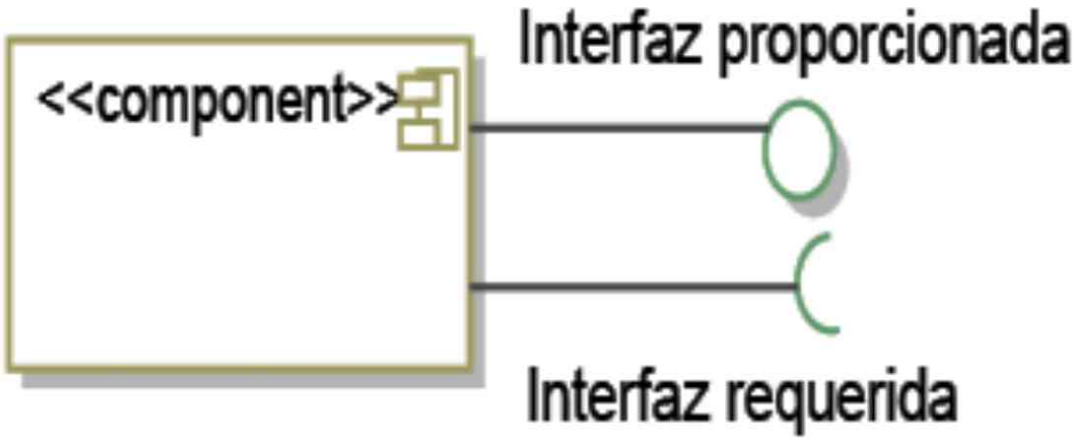


Figura 5.5. Componente UML

Donde el componente se comporta como una caja negra, ocultando su información e implementación, y exterioriza sus funcionalidades mediante las interfaces. Por ello, la interfaz con forma de círculo[12] representa la especificación que ofrece el componente, mientras que la interfaz inferior (marcada con un semicírculo[13]) representa la conexión con una interfaz que proporciona una entrada de datos.

Para definir un componente y otros elementos extra en UML se recurre a los estereotipos. Estos se distinguen por el uso de la notación <<nombre-estereotipo>>. En general, los estereotipos en UML se utilizan con frecuencia para ampliar el conjunto de clasificadores que se utilizan en un diagrama, o lo que es lo mismo, el conjunto de elementos de modelado. Por ejemplo, en la figura 5.5 recurrimos al estereotipo <<component>> para indicar el elemento extra de componente. Como veremos a lo largo de los siguientes capítulos, otros de los estereotipos ampliamente utilizados en UML son <<use>>, <<interface>>, etc.

En UML, los diagramas de componentes pueden participar en asociaciones, generalizaciones y pueden tener atributos y operaciones. Además, el propio componente puede estar formado por otros componentes internos, como se muestra en la figura 5.6. En este caso las interfaces interiores del componente deben "delegar" sus responsabilidades a las interfaces exteriores.

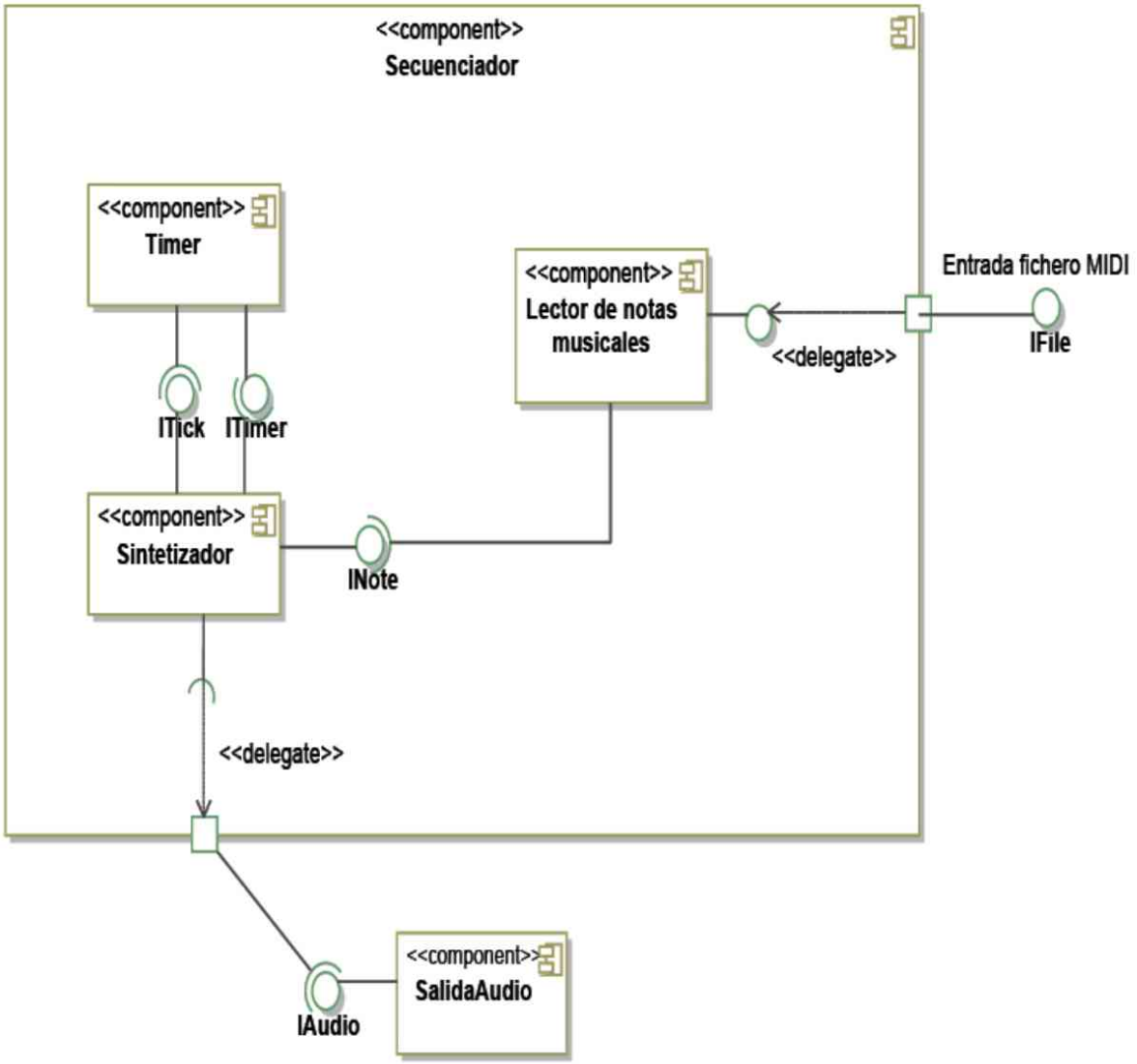


Figura 5.6. Modelo de componente con subcomponentes

En el diagrama de la figura anterior, el componente de nivel superior encapsula al resto de las interfaces internas y delega sus interfaces a los componentes de nivel inferior que se relacionan entre ellos para llevar a cabo la funcionalidad del componente exterior.

Un componente tiene una estructura interna que puede visualizarse por medio del propio UML, tal como se muestra en la figura 5.7. Esta visualización contiene todos los atributos, operaciones e interfaces requeridas y provistas por el componente. En la sección de <<*artifacts*>> indicaremos el artefacto físico que proporciona la implementación del mismo.

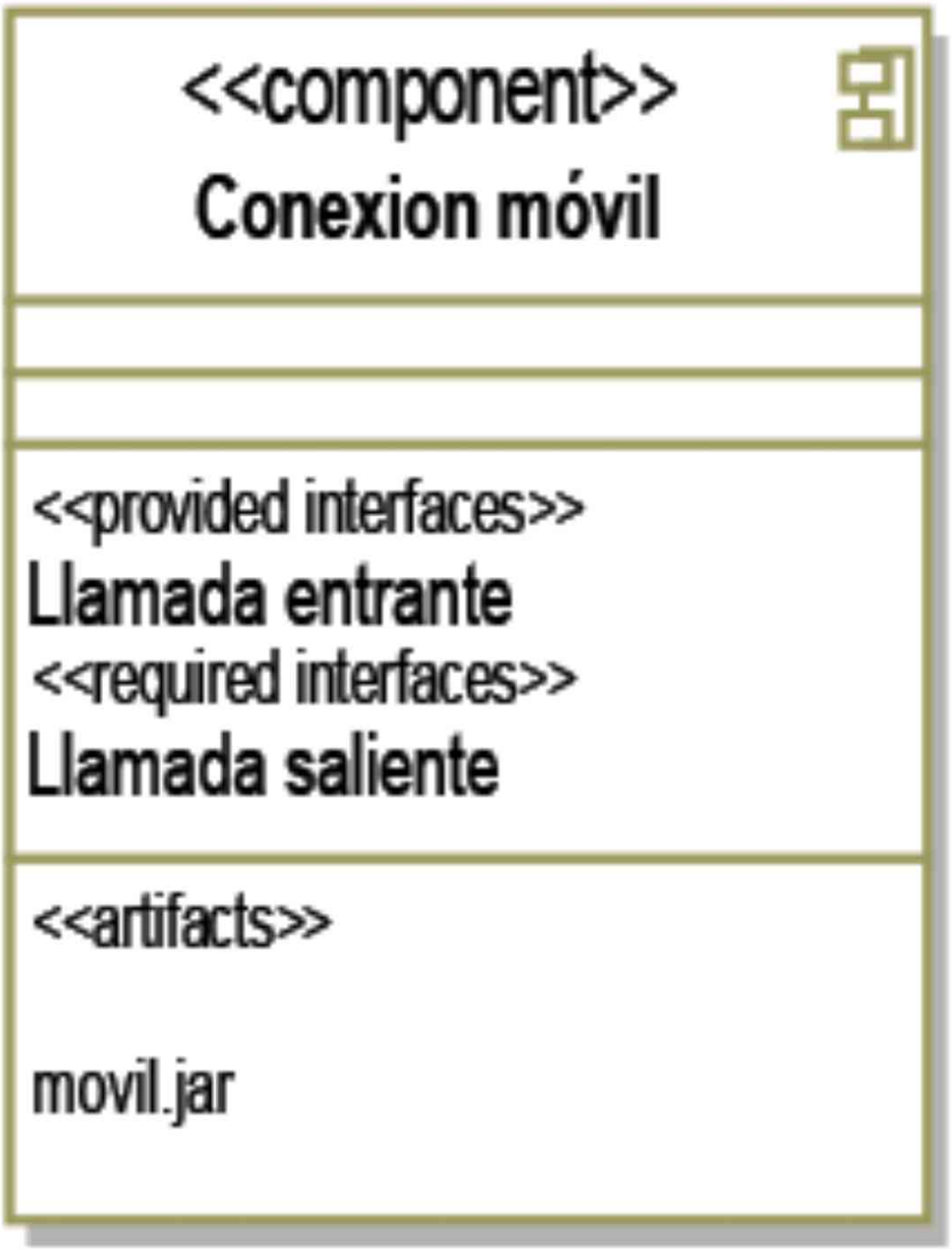


Figura 5.7. Visión interna del componente

Puertos

Como se muestra en la figura 5.8, los puertos tiene la utilidad de agrupar semánticamente un conjunto de interfaces. Funciona a modo de punto de conexión del componente con su entorno. La finalidad es clarificar, agrupar y simplificar la representación de las interfaces requeridas y proporcionadas.

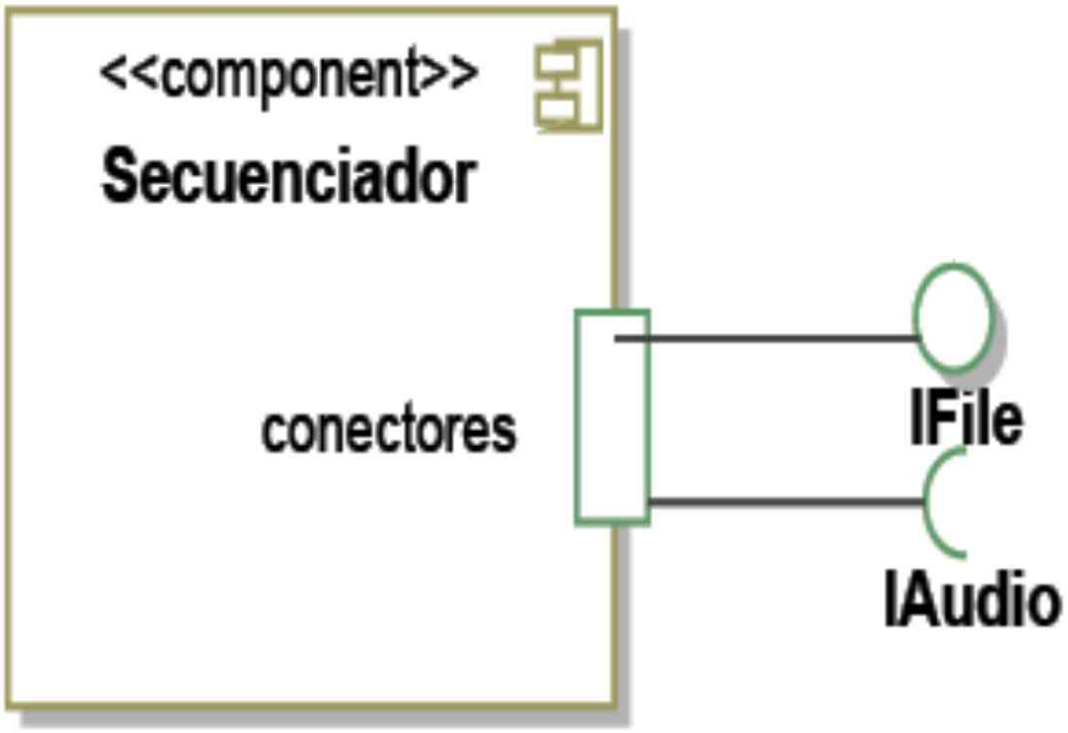


Figura 5.8. Ejemplo de puerto

Un ejemplo de simplificación de diagrama y agrupación de interfaces es la que se puede apreciar en la figura 5.9:

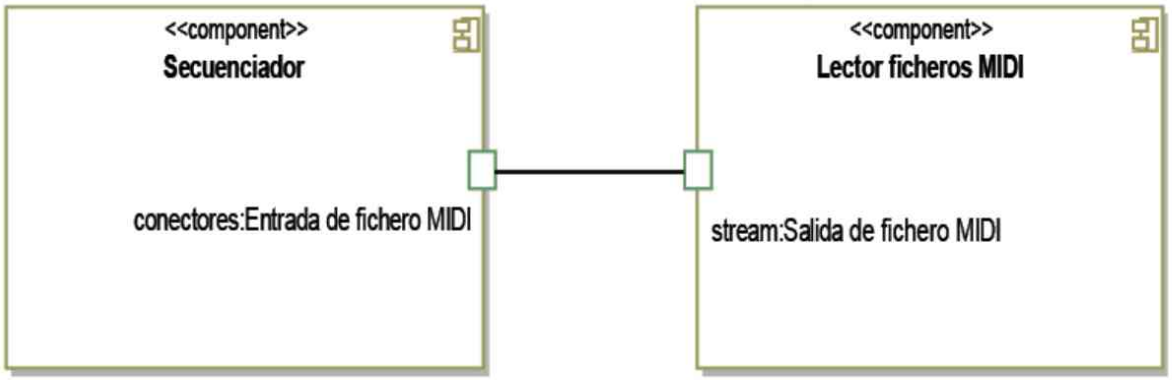


Figura 5.9. Conexión de dos componentes a través de puertos

Obviamente deben corresponderse implícitamente las conexiones entre las interfaces provistas y las requeridas.

Finalmente, los puertos pueden tener multiplicidad y visibilidad, por lo que si situamos el puerto dentro de los bordes del componente, la visibilidad será protegida, mientras que si lo situamos en el exterior tendrá visibilidad pública. La multiplicidad del puerto indica el número de sus instancias, por lo que por ejemplo se puede representar conectores:IAudio[3]si hubiera tres instancias de la interfaz requerida IAudio.

Dependencias

Las dependencias en UML en general indican que un elemento proveedor del modelo tiene un efecto sobre otro elemento que denominaremos cliente. En los diagramas de componentes en particular especifican que el cliente está conectado o invoca a las operaciones proporcionadas por las interfaces del proveedor o también que depende de él para su implementación. Las dependencias se representan mediante una línea discontinua acabada en punta de flecha abierta en la parte del proveedor.



Figura 5.10. Ejemplo de dependencia de componentes

Caso de estudio: Ajedrez

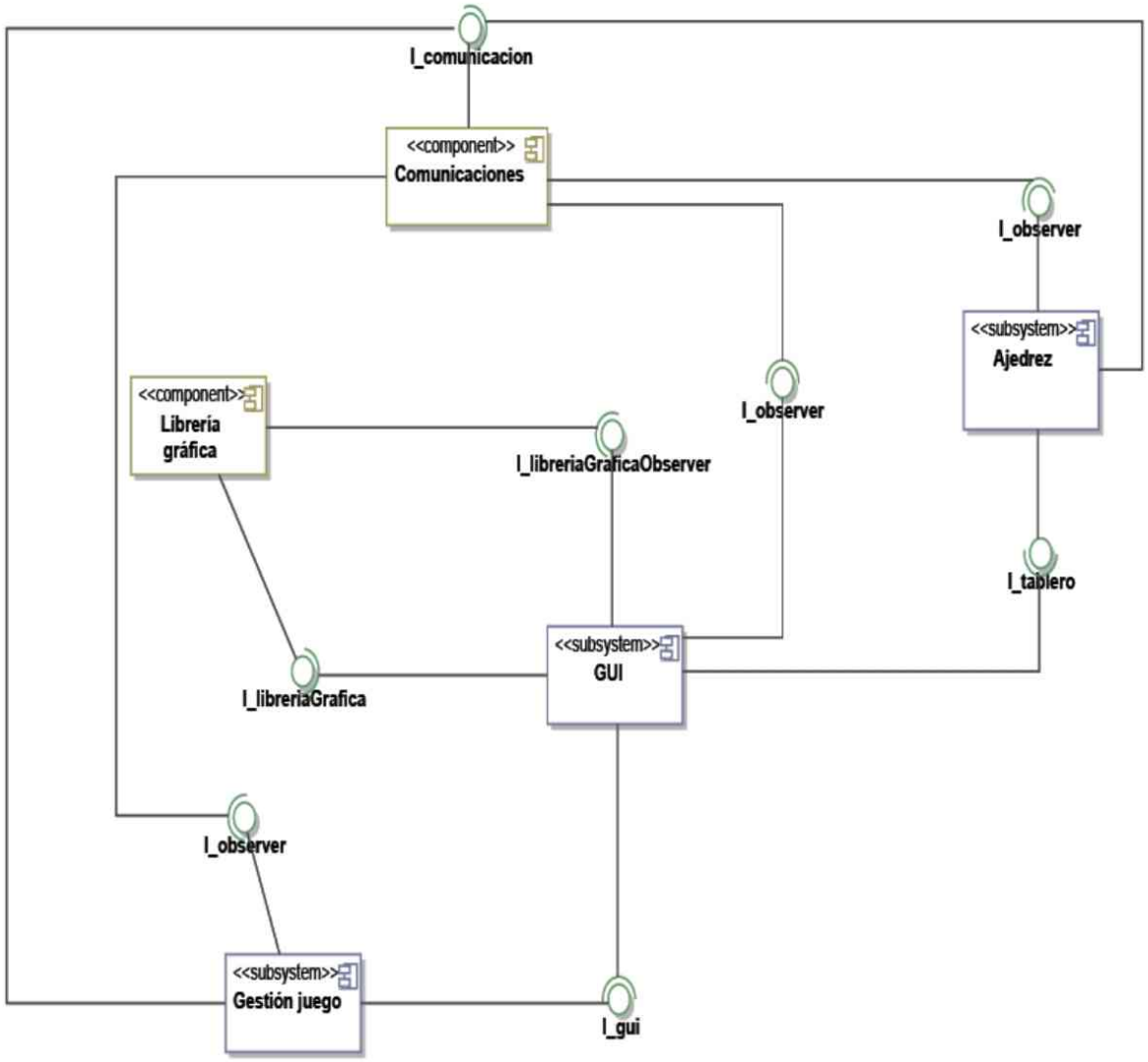


Figura 5.11. Diagrama de componentes para el juego de ajedrez

En la figura 5.11 se representan los principales componentes de la arquitectura de la aplicación del ajedrez. Para ello nos hemos servido del modelo del dominio y el análisis de los requisitos. Cada componente agrupa un subsistema clave del modelo del dominio del juego de ajedrez y proporciona un conjunto de interfaces proporcionadas e interfaces requeridas. Ambas se interconectan con otras interfaces de otros componentes o subsistemas para completar el modelo. A modo de ejemplo, se observa que el componente *Comunicaciones* implementa la interfaz *I_comunicacion* y llama a las operaciones definidas en la interfaz *I_observer* implementada en los subsistemas *Ajedrez*,

GUI y *GestiónJuego*.

Caso de estudio: Mercurial

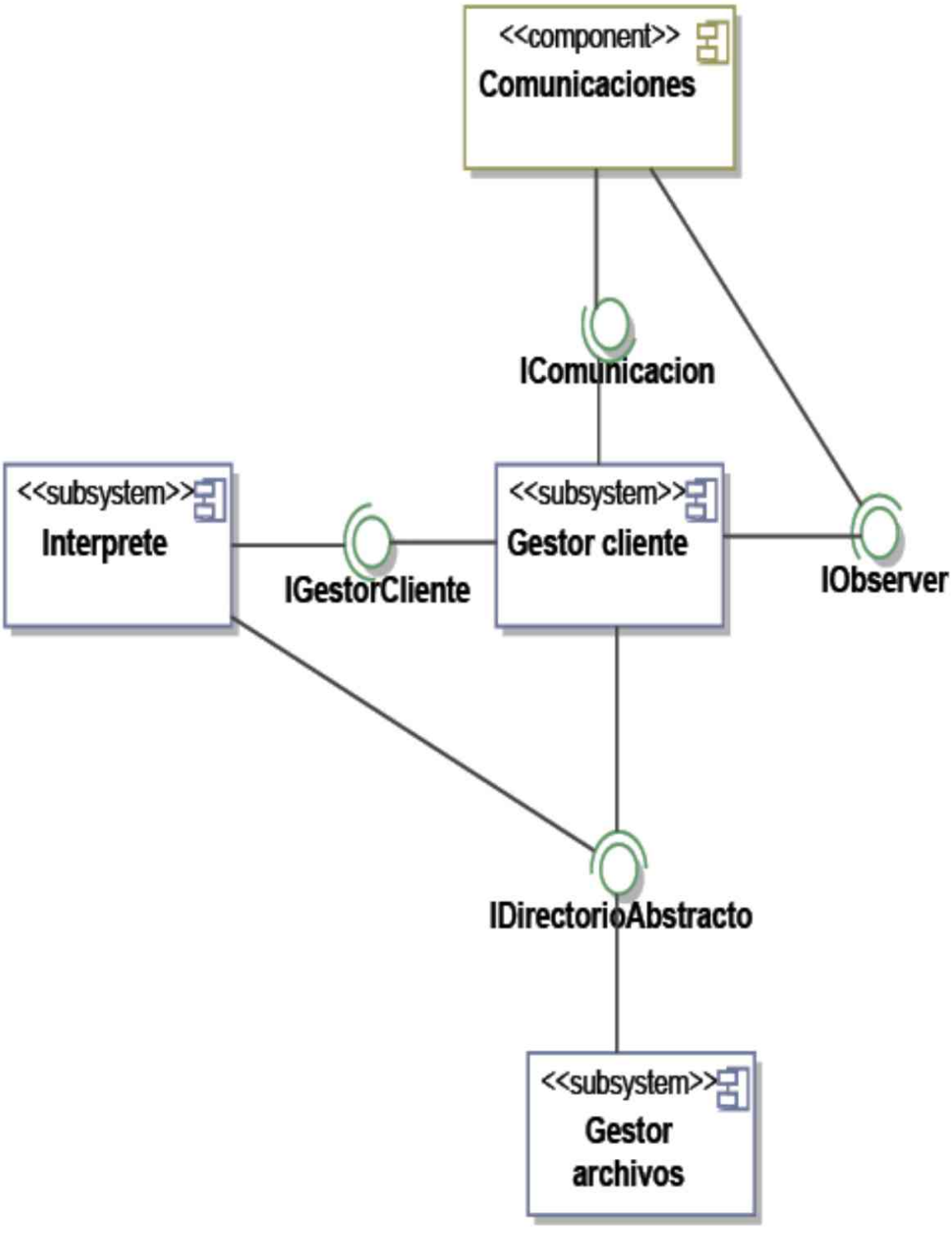


Figura 5.12. Diagrama de componentes para la aplicación Mercurial

Al igual que en el caso anterior, en la figura 5.12 se representan los componentes principales de la aplicación *Mercurial*. Los cuatro componentes

representados implementan interfaces que proporcionan a otros componentes o subsistemas y además llaman a interfaces de otros componentes o subsistemas.

Por ejemplo, el subsistema *Gestor archivos* implementa la interfaz *IDirectorioAbstracto* que es llamada por los subsistemas *Interprete* y *Gestor cliente*.

Como se verá más detenidamente en el capítulo seis, cada componente o subsistema agrupa un conjunto de clases que representan una parte vital de la aplicación. La división del sistema en componentes es una estrategia arquitectónica que asegura los principios de la programación orientada a objetos y la distribución eficaz de las funcionalidades en compartimentos interconectados por interfaces, aumentando de esta forma la cohesión y reduciendo el acoplamiento entre los componentes.

Caso de estudio: Servicio de cifrado remoto

Lado cliente

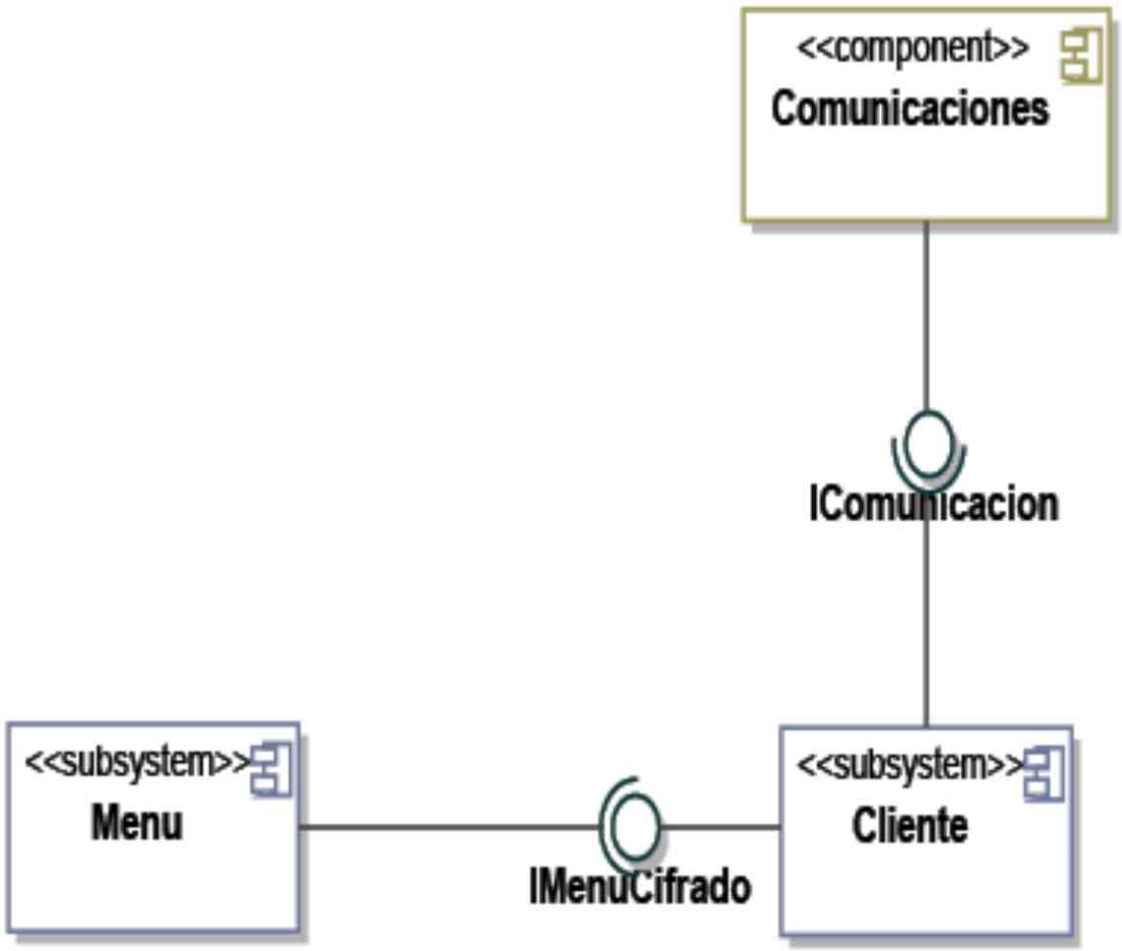


Figura 5.13. Diagrama de componentes para el lado cliente

La división principal en módulos de la aplicación de servicio de cifrado remoto puede apreciarse en la figura 5.13. El subsistema *Menu* será muy parecido tanto en el cliente como en el servidor al igual que el componente *Comunicaciones*; sin embargo, el subsistema *Cliente* será una versión más reducida que su homólogo en el servidor.

Respecto a la implementación de las interfaces, será el componente *Comunicaciones* quien exhiba la interfaz por la que interactuar con sus funciones de red, mientras el subsistema *Cliente* expondrá la interfaz *IMenuCifrado* para operar con las funcionalidades de encriptado.

Lado servidor

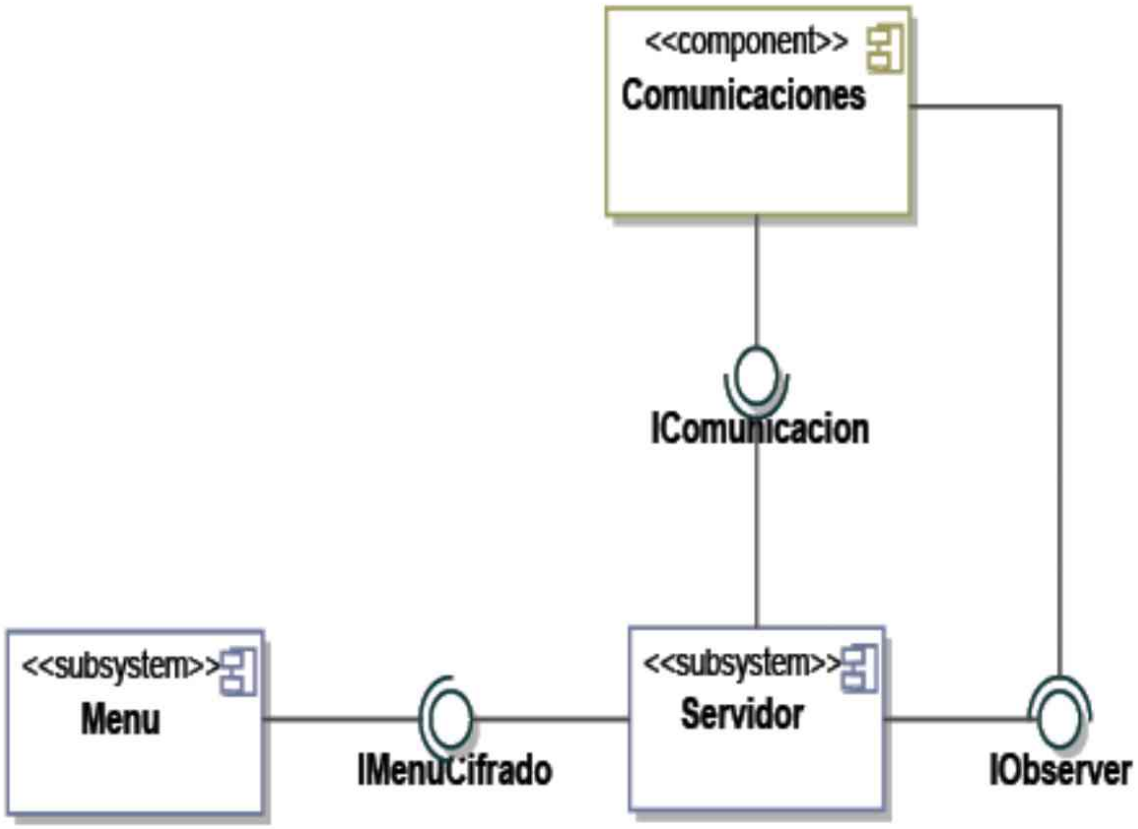


Figura 5.14. Diagrama de componentes para el lado servidor

El lado servidor mantiene un modelo muy parecido al del lado cliente, a excepción de la interfaz *IObserver* en el subsistema *Servidor* de la figura 5.14, procedente del patrón *Observer* (ver capítulo nueve) que se responsabilizará de recibir las notificaciones de mensajes y claves de sesión procedentes de la red.

Respecto al subsistema *Menu,* no varía mucho en consideración al lado cliente, mientras que es el subsistema *Servidor* el que ahora se encargará de las funciones de encriptación y desencriptación.

diagramas de despliegue

Dentro del diseño arquitectónico nos encontramos con un nuevo tipo de diagrama que nos permite modelar la organización del hardware. Este modelo arquitectónico que representa el despliegue de la aplicación sobre diferentes ubicaciones físicas puede servirnos para visualizar la infraestructura de ejecución de los artefactos software. La importancia de este diagrama estriba en la necesidad de abordar a tiempo y dentro del ciclo de vida del proyecto la toma de decisiones oportunas con respecto a los *requerimientos no funcionales*. El *diagrama de despliegue* trabaja con las instancias principales del hardware y del software, por lo que se compone de dos elementos fundamentales: *nodos* y *artefactos*.

Nodos

Los nodos son la entidad fundamental del *diagrama de despliegue* y representan a elementos hardware y software complejos. Se especifican mediante un nombre de nodo (las instancias concretas se subrayarán) y un estereotipo que identifica el tipo de nodo. Concretamente, en UML 2.x existen dos tipos de nodos:

- <<device>>: Modela lo que correspondería a un dispositivo físico tal como una impresora o un *host*.
- <<execution environment>>: Representan componentes específicos que son desplegados como artefactos ejecutables. Por ejemplo: navegadores, servidores Web, aplicaciones cliente/servidor, etc.

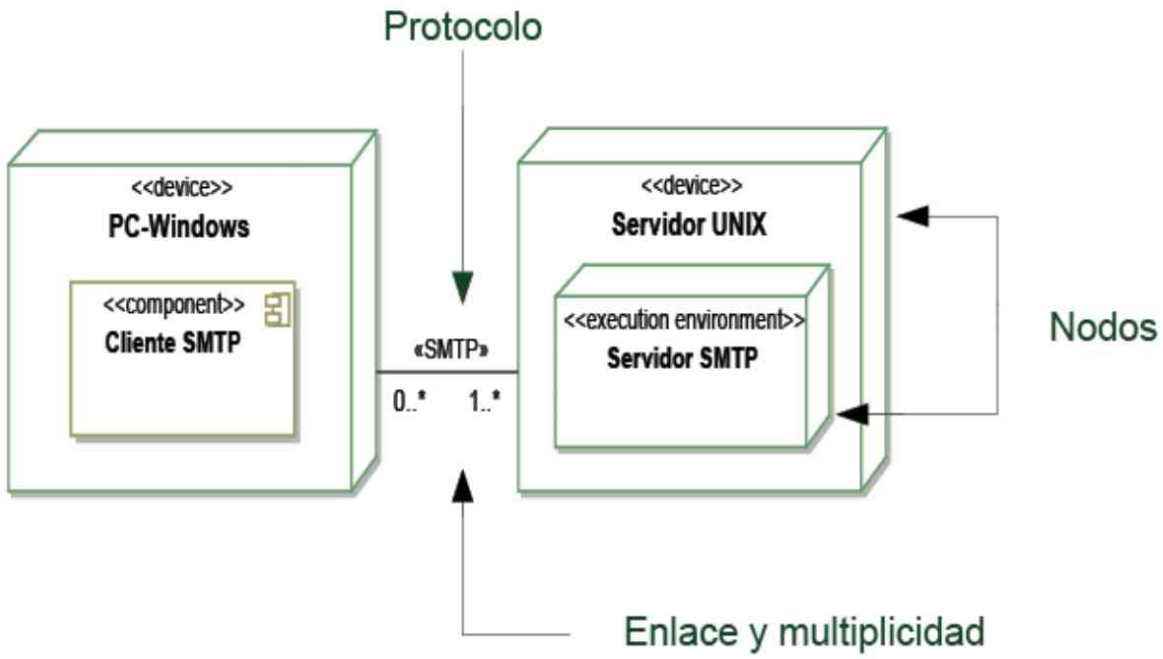


Figura 5.15. Diagrama de despliegue

Los nodos pueden tener asociaciones y multiplicidad (cardinalidad), por lo pueden existir relaciones entre elementos físicos. Por ejemplo, en la figura 5.15 se representa la interconexión de una arquitectura cliente/servidor a

través de una red IP con el protocolo SMTP. En el nodo cliente del tipo <<device>> se halla un componente que implementa un agente de usuario de correo electrónico. Al otro lado de la conexión se encuentra un host del tipo estación de trabajo UNIX con su respectiva aplicación servidor identificada mediante el estereotipo <<execution environment>>.

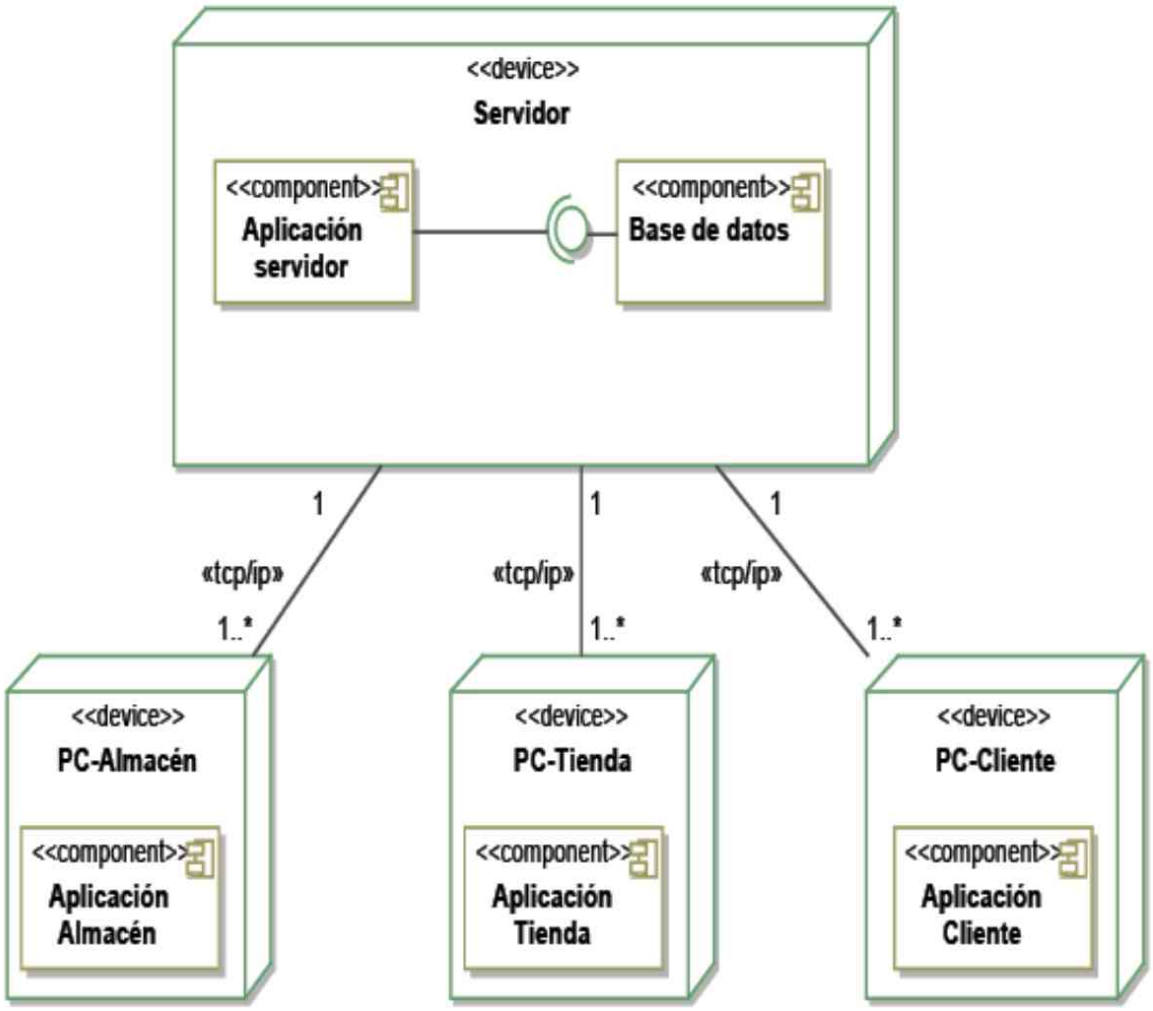


Figura 5.16. Ejemplo de diagrama de despliegue con varios nodos y componentes

En el ejemplo anterior (figura 5.16) puede observarse la distribución de los nodos <<device>> para los clientes que acceden simultáneamente a un nodo <<device>> servidor. El nodo servidor contiene un pequeño diagrama de componentes para la conexión entre la aplicación servidor (por ejemplo JEE) y la base de datos. Los enlaces entre los nodos cliente y el nodo servidor indican

la topología de la red de comunicaciones, así como el número de clientes que pueden existir en cada lugar simultáneamente.

Artefactos

Definiremos a los *artefactos* como los elementos software que se despliegan dentro de los nodos. Estos artefactos pueden ser: código fuente, librerías, ficheros ejecutables, ficheros de script, bases de datos, ficheros XML, ficheros de configuración y otro tipo de documentos del desarrollo del proyecto.

Caso de estudio: Ajedrez

Veamos ahora un ejemplo práctico de diagrama de despliegue para el caso del juego de ajedrez en su versión C++ para Windows y Linux. En la figura 5.17 se observan los artefactos desplegados tanto en el cliente como en el servidor. Los artefactos para el lado cliente son: las librerías de comunicaciones, de gráficos y el propio juego ejecutable en el lado cliente. Para el lado del servidor tenemos una situación similar, con la versión Linux de la librería de comunicaciones y el programa servidor del juego.

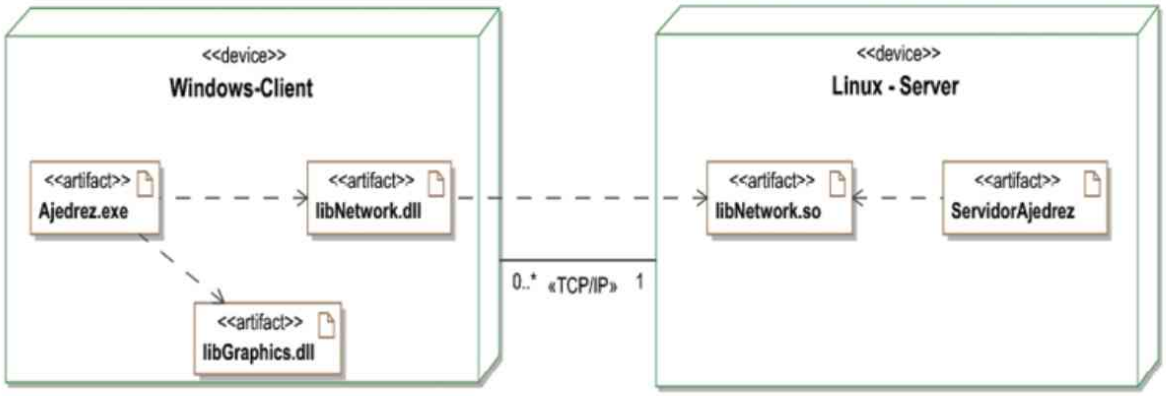


Figura 5.17. Diagrama de despliegue para el juego de ajedrez en C++

Caso de estudio: Mercurial

El diagrama de despliegue con el modelo arquitectónico y de implementación para el programa *Mercurial* en versión Java puede apreciarse en la figura 5.18. En este modelo las flechas discontinuas representan las dependencias entre artefactos.

Es importante destacar previamente que en caso de existir una asociación entre los nodos (como en el ejemplo con http y tcp/ip) es necesario interrelacionar los respectivos artefactos que llevan cabo dicha comunicación (*java.net* y *libNetwork.dll*). En general, cuando exista una dependencia entre dos artefactos en diferentes nodos, ésta debe ser tenida en cuenta y representada en el diagrama UML.

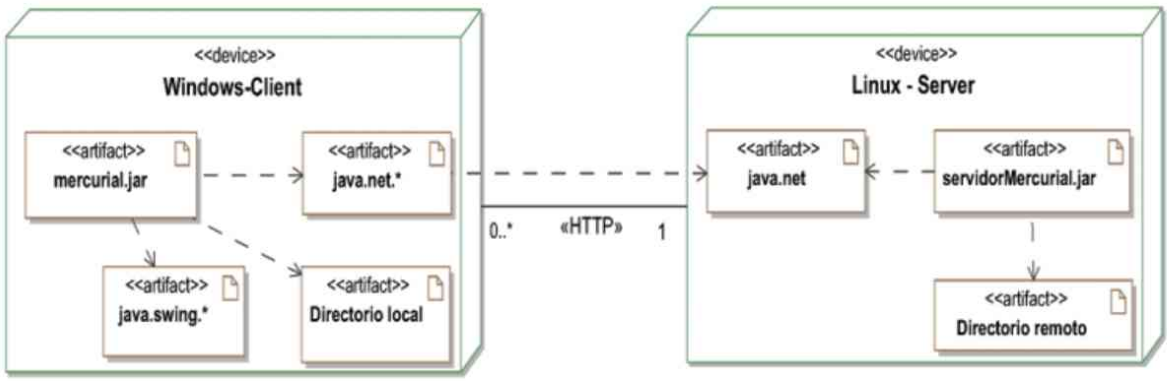


Figura 5.18. Diagrama de despliegue para Mercurial en versión Java

Caso de estudio: Servicio de cifrado remoto

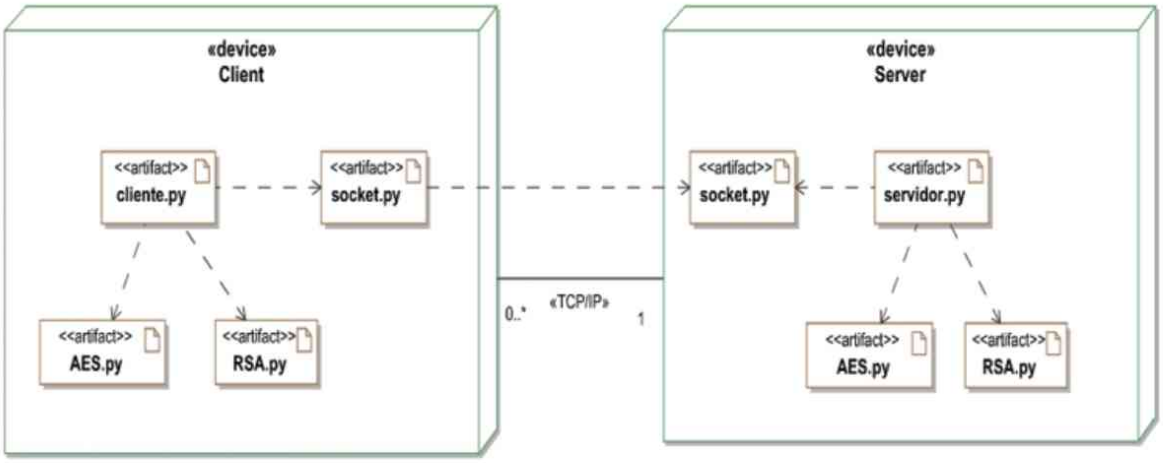


Figura 5.19. Diagrama de despliegue para la aplicación de cifrado remoto

Como vemos en la figura 5.19, tanto el lado cliente como el lado servidor se componen de cuatro artefactos: el subsistema cliente, la librería de red y las librerías de algoritmos de cifrado/descifrado. Dichos artefactos se encuentran relacionados según la dependencia entre ellos. Finalmente, apreciamos el enlace de red (línea continua sin flechas) que corresponde al protocolo de transporte TCP/IP de Internet.

diagramas de paquetes

Entramos en la última sección dedicada al diseño arquitectónico. Previo al paso del diseño detallado, será necesario organizar la estructura lógica de los principales artefactos utilizados para implementar la aplicación. Por este motivo surgen los *diagramas de paquetes*, los cuales nos ayudarán a ordenar y agrupar todos los elementos tales como diagramas, librerías, relaciones, ejecutables, etc. y en general cualquier elemento del modelo UML.

Paquetes

Los paquetes son los contenedores que facilitan la agrupación de los elementos o artefactos UML en unidades lógicas. Además, cuando estos elementos son asignados a un paquete, se les asigna *un espacio de nombres*[14] con el cual son identificados unívocamente. Consiguientemente esta idea fomenta el agrupamiento semántico de artefactos relacionados entre sí, permitiendo de esta forma dividir la estructura de ficheros del programa en bloques independientes y jerárquicamente interrelacionados.

En lo que respecta a la organización del *espacio de nombres,* en C++ disponemos de la palabra reservada *namespace* y en Java la cláusula *package*; por lo tanto, definiendo correctamente estas sentencias es posible la agrupación de clases y estructuras del lenguaje en una unidad identificable y separada.

En UML representaremos el paquete mediante varias formas de notación:

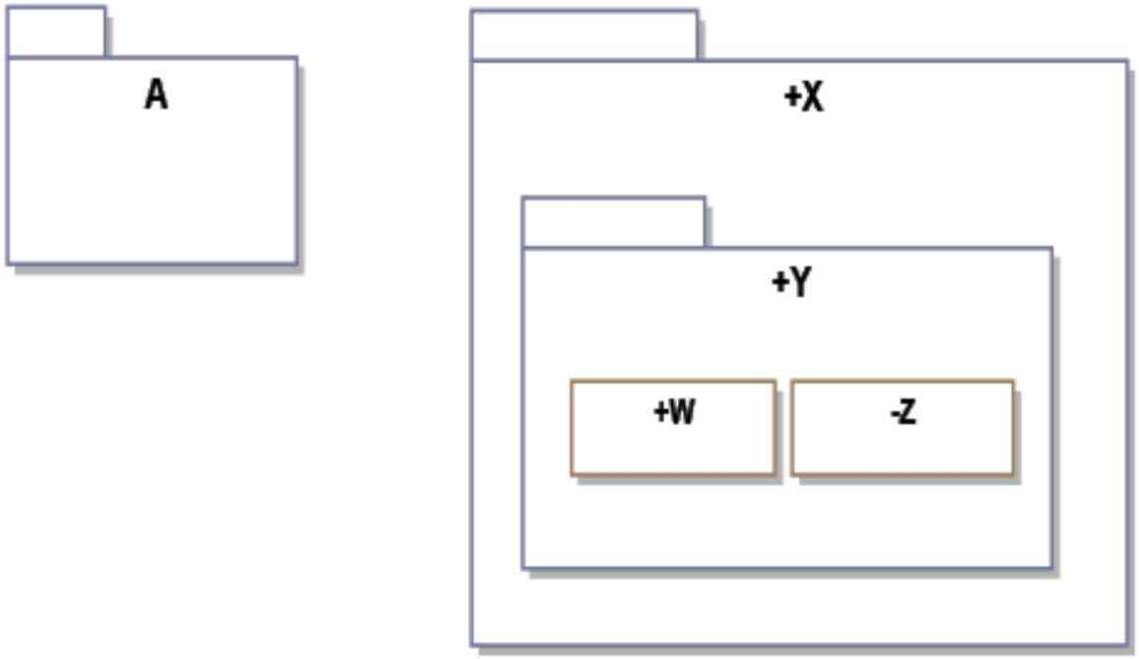


Figura 5.20. Ejemplos de diferentes tipos de paquetes

En la figura 5.20 se muestra el paquete "A" a la izquierda, mientras que a la derecha vemos el paquete "X" con el paquete interno "Y" que contiene las

clases “W” y “Z”.

Con la finalidad de acceder a los elementos de un paquete será necesario especificarlos mediante el acceso a través de los identificadores de los *espacios de nombres*. Así, para el caso del ejemplo de la clase “W” tendremos que especificar su acceso mediante la siguiente sintaxis: X::Y::W.

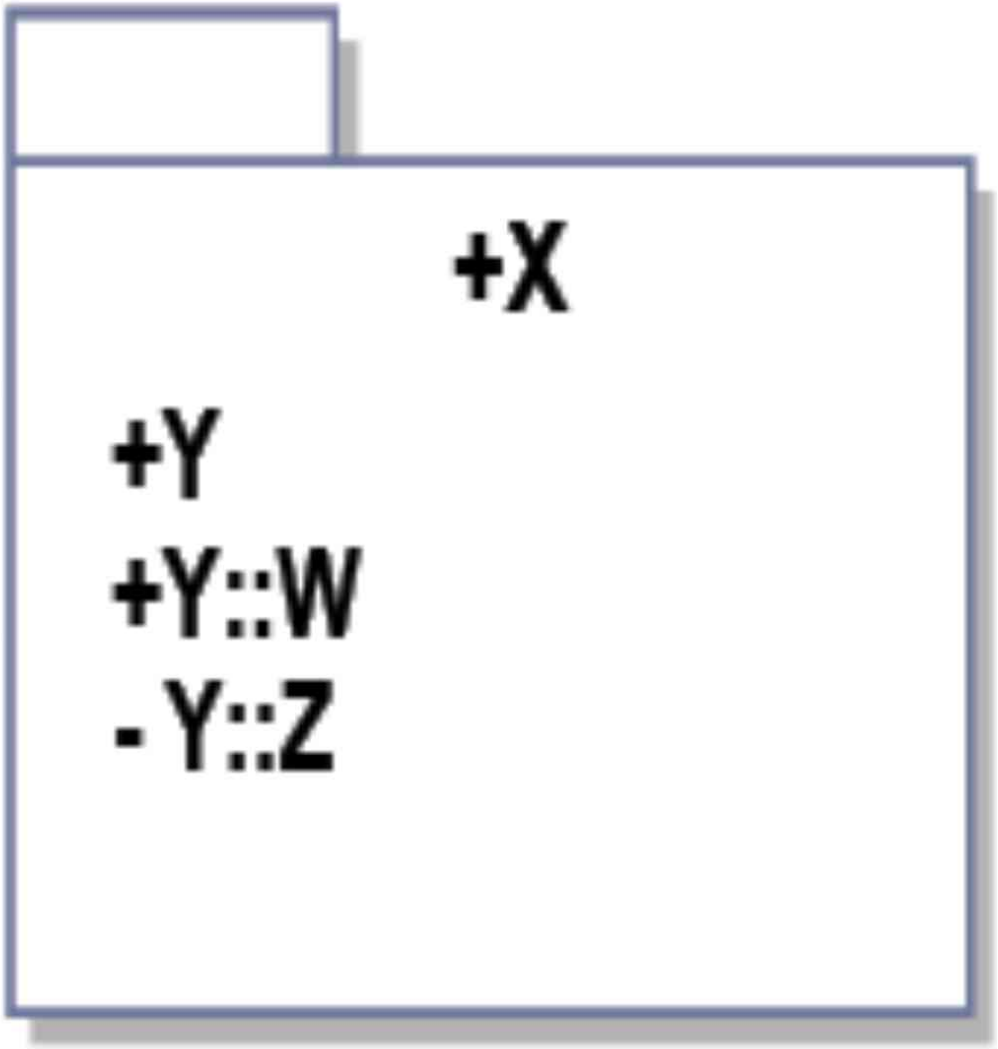


Figura 5.21. Representación alternativa del paquete X

Existen especificadores para los elementos contenidos dentro de un paquete. En la figura 5.21 el paquete “Y” aparece con visibilidad *pública* por lo que será visible por otros elementos fuera del paquete, mientras que la clase “Z” tiene

visibilidad *privada* por lo que no podrá ser accedida por los elementos exteriores, sin embargo, la clase "W" es exportada públicamente desde el paquete "Y" y es visible fuera del mismo.

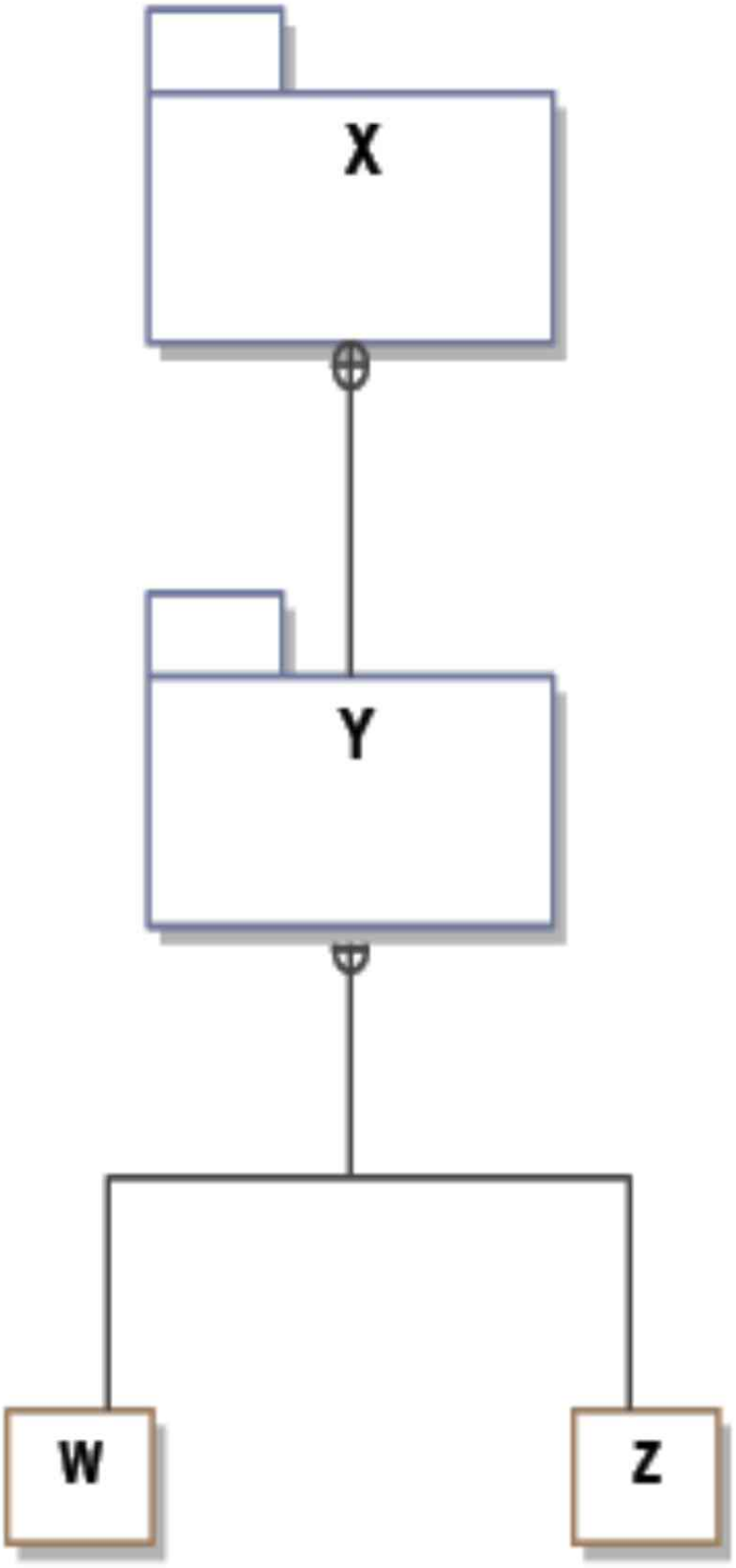


Figura 5.22. Otra representación alternativa del paquete X

Esta representación es útil cuando existen una gran cantidad de paquetes anidados y es imposible representarlos dentro un solo paquete.

Generalización

En el diagrama de paquetes también es posible la herencia de elementos de los paquetes padres a los paquetes hijos. Cuando esto ocurre, lo paquetes hijos heredan los elementos públicos de los padres como ocurre con el siguiente ejemplo de la figura 5.23: Las clases *EntradaDatos* y *SalidaDatos* son heredadas.

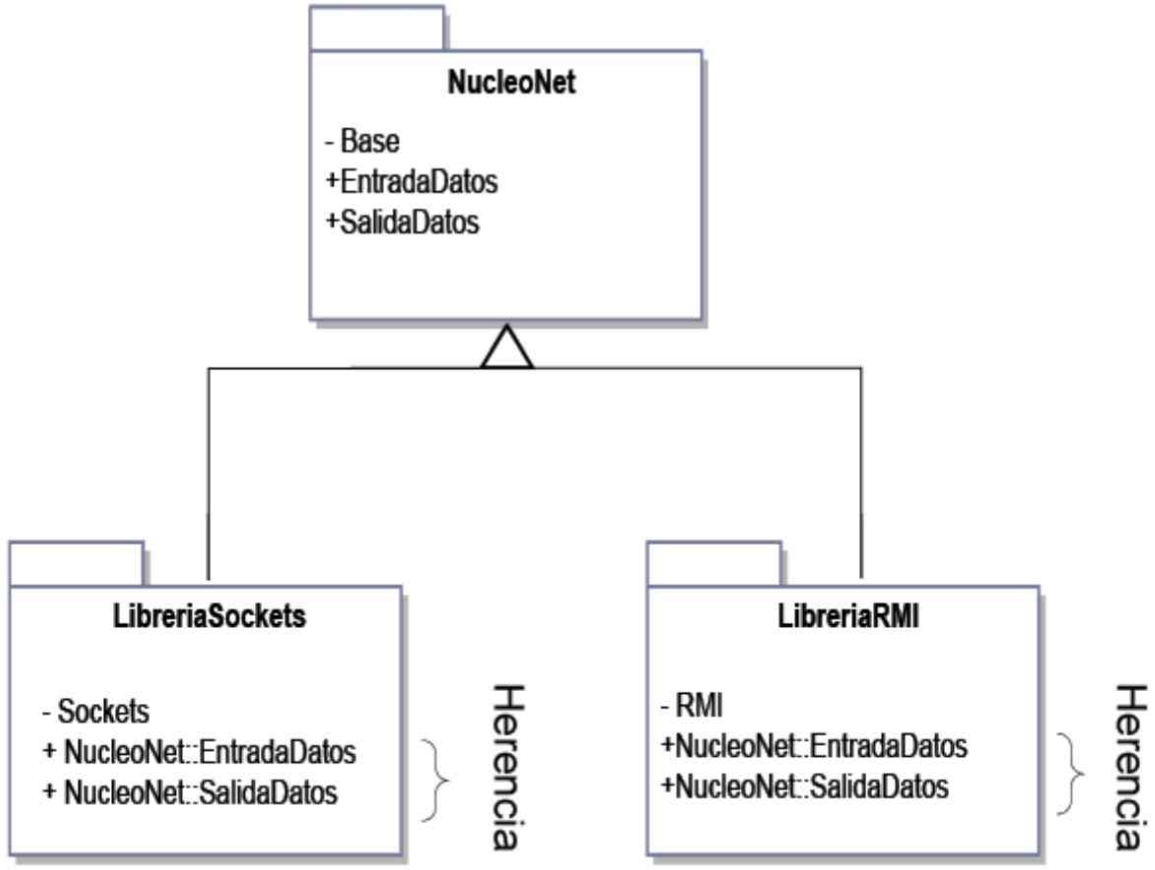


Figura 5.23. Ejemplo de generalización

Relaciones de dependencia

Como en el resto de los diagramas de UML, existen dependencias establecidas entre paquetes relacionados. Cuando surge una relación de dependencia entre dos paquetes suelen compartirse o restringirse la inclusión de elementos de uno a otro.

Es importante evitar relaciones de dependencia cíclicas entre paquetes (en cualquier lenguaje), es decir, relaciones del tipo A ↔ B. Cuando ocurre un situación de este tipo podemos optar por unir los paquetes A y B en un sólo paquete A, o separarlos en varios paquetes interrelacionados: A→ C, B → C y A→ B.

En la figura 5.24 se muestran los diferentes modos de dependencia existentes entre paquetes:

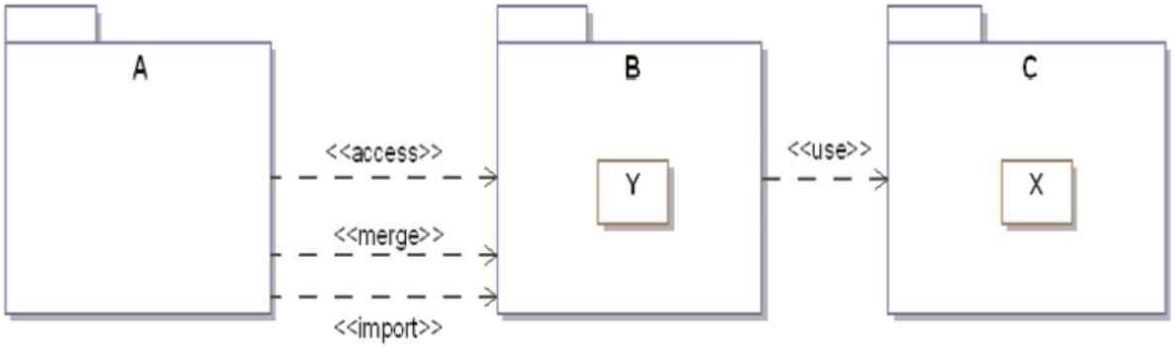


Figura 5.24. Diversos modos de dependencia entre paquetes

Modo	Descripción
<<use>>	Indica que el paquete "B" *usa* los elementos públicos del paquete "C". Si no se especifica un estereotipo se asume éste por defecto.
<<access>>	Los elementos públicos de "B" son *añadidos* como elementos privados de "A".
<<merge>>	Los elementos públicos de "B" se *unen* a los elementos de "A". No suele usarse en

	modelado de paquetes.
<<import>>	Los elementos públicos de "B" se *añaden* como elementos públicos de "A". No son necesarios los especificadores.

Tabla 5.1. Modos de acceso

Caso de estudio: Ajedrez

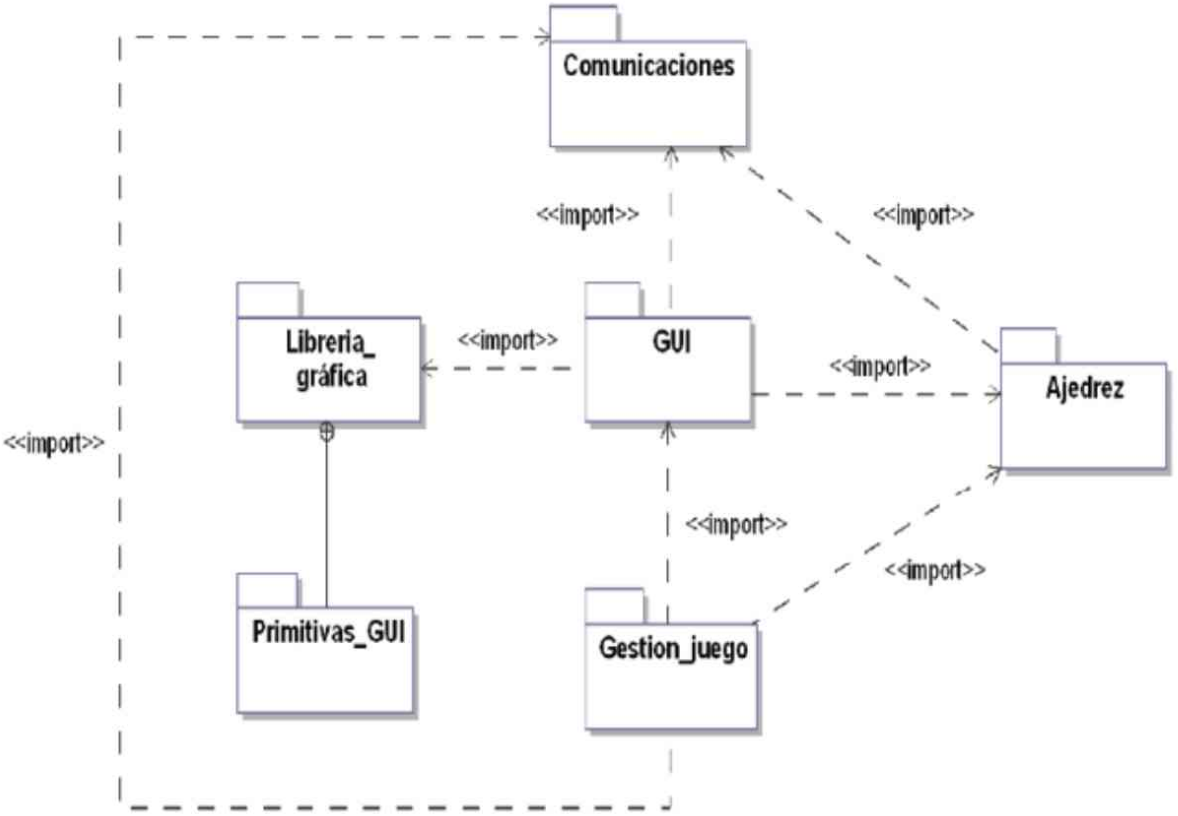


Figura 5.25. Diagrama de paquetes para el juego de ajedrez en versión C++

Como se puede apreciar cada paquete se corresponde generalmente con un componente o subsistema identificado en el análisis arquitectónico de componentes. Si bien se han ampliado nuevos paquetes incluidos en otros paquetes para jerarquizar con más detalle el conjunto de clases especiales para el tratamiento de ciertos requisitos funcionales. El paquete *Ajedrez* ha sido resultado de la unión de los paquetes relacionados con los algoritmos de Inteligencia Artificial y el subsistema de control de la IA del juego. Como casi todos los subsistemas requieren de la oferta de las interfaces de comunicación, es necesario la inclusión del paquete de *Comunicaciones* en dichos subsistemas basados en componentes.

No será objeto de este estudio ampliar a la implementación de detalles de los paquetes relacionados con las librerías gráficas, por lo que se dejará como simple indicación dentro del modelo. Tampoco se ha tratado el modelo de

paquetes relacionado con la parte del servidor, por lo que se propone al lector su posible resolución.

Caso de estudio: Mercurial

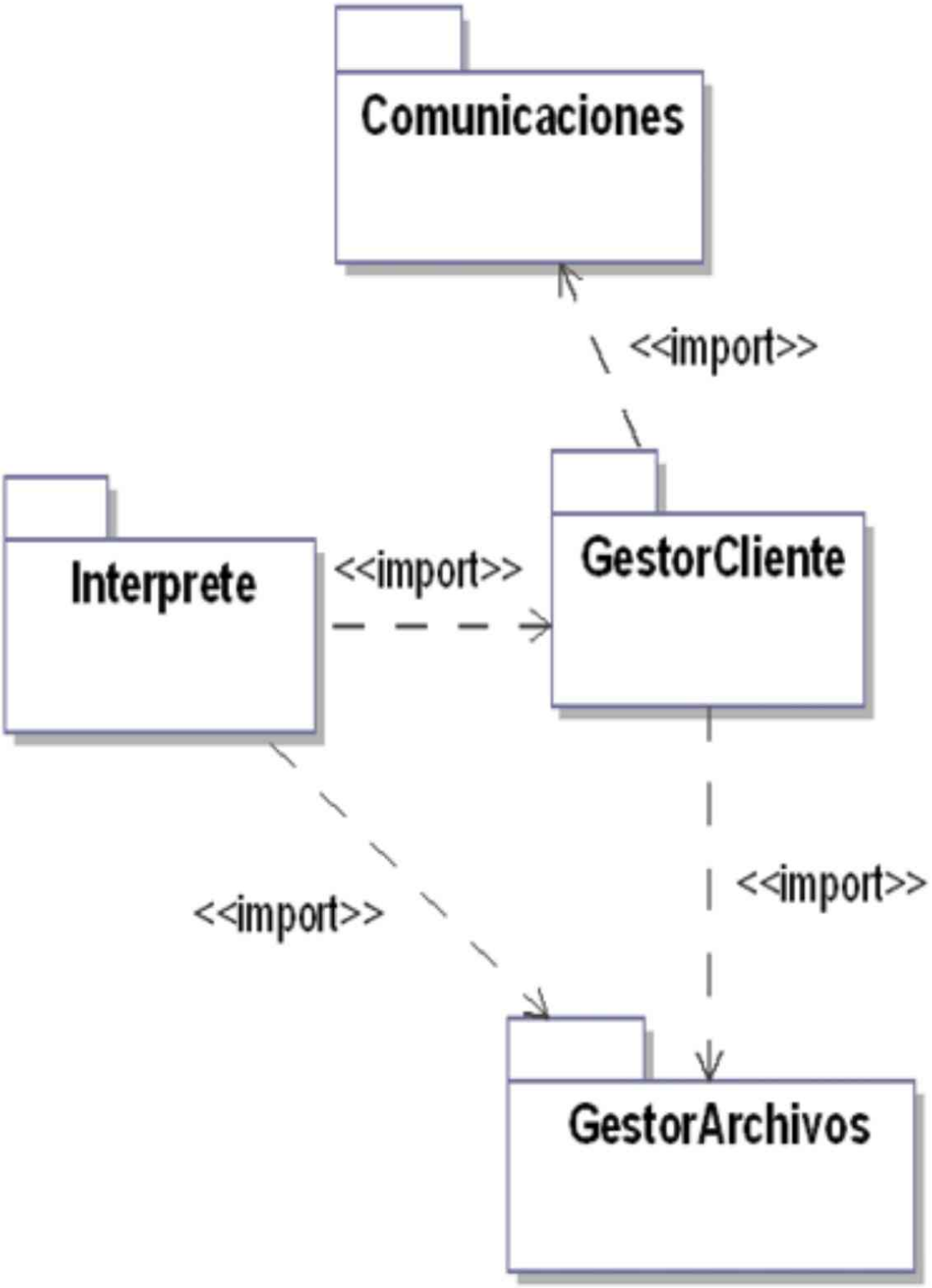


Figura 5.26. Diagrama de paquetes para la aplicación Mercurial en versión Java

En la figura 5.26 se recoge el modelo de paquetes para la aplicación *Mercurial*. Como en el caso del ajedrez, cada paquete se relaciona con un subsistema del diagrama de componentes. Las dependencias existentes entre los paquetes reflejan las relaciones o invocaciones entre dichos subsistemas, puesto que implican la inclusión de clases e interfaces. Cada paquete contendrá un conjunto de clases que se detallarán en el próximo capítulo y que están basadas en el diseño preliminar realizado en el modelo del dominio.

Esta aplicación es mucho más sencilla que su homóloga basada en el juego del ajedrez, como veremos en el próximo capítulo. La simplificación de la arquitectura de paquetes responde al enfoque orientado únicamente a la parte cliente que se pretende en esta obra. No obstante, se deja como ejercicio propuesto la resolución del enfoque del lado del servidor.

Caso de estudio: Servicio de cifrado remoto

Lado cliente

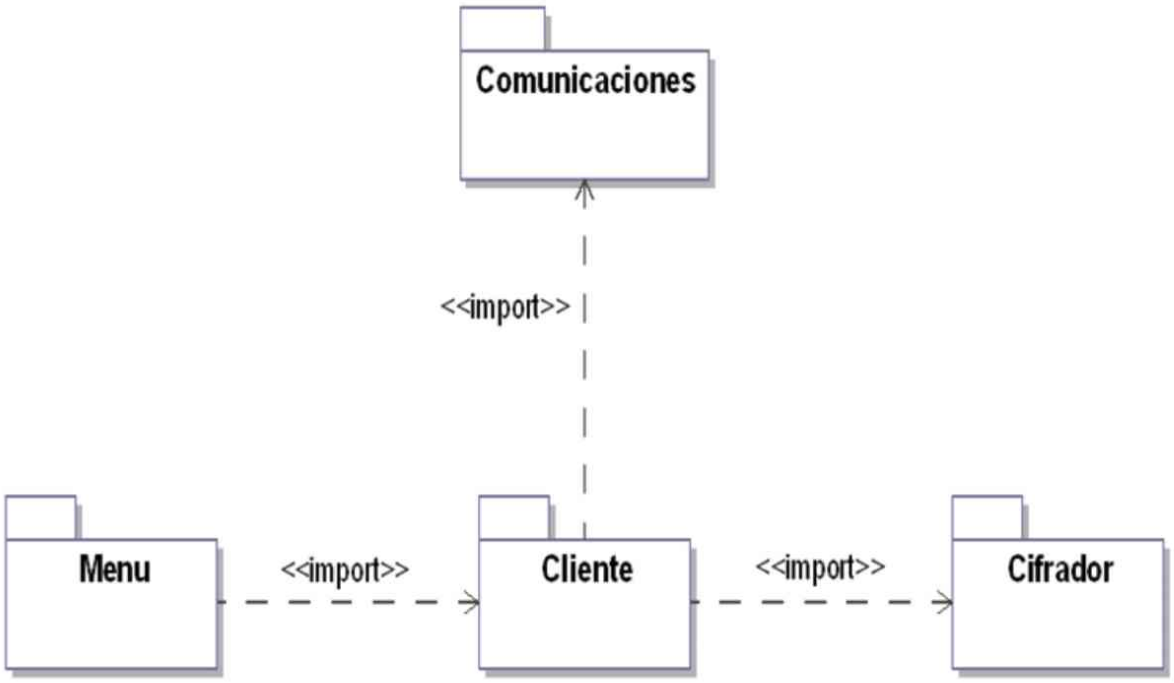


Figura 5.27. Diagrama de paquetes para el lado cliente

En el diagrama de paquetes de la figura 5.27 se observa la distribución de los componentes y subsistemas anteriormente descritos. Las líneas discontinuas modelan la dependencia entre ellos mediante la dirección de la punta de flecha. En el paquete *Menu* se incluirán las entidades relacionadas con la fachada de presentación en modo de texto, mientras que el paquete *Comunicaciones* desplegará el componente relacionado con la interconexión de red de la aplicación distribuida. Por último, el paquete *Cliente* contendrá las entidades relacionadas con el usuario novel visto en el capítulo anterior y que deberá importar las librerías implementadas en el paquete *Cifrador*.

Lado servidor

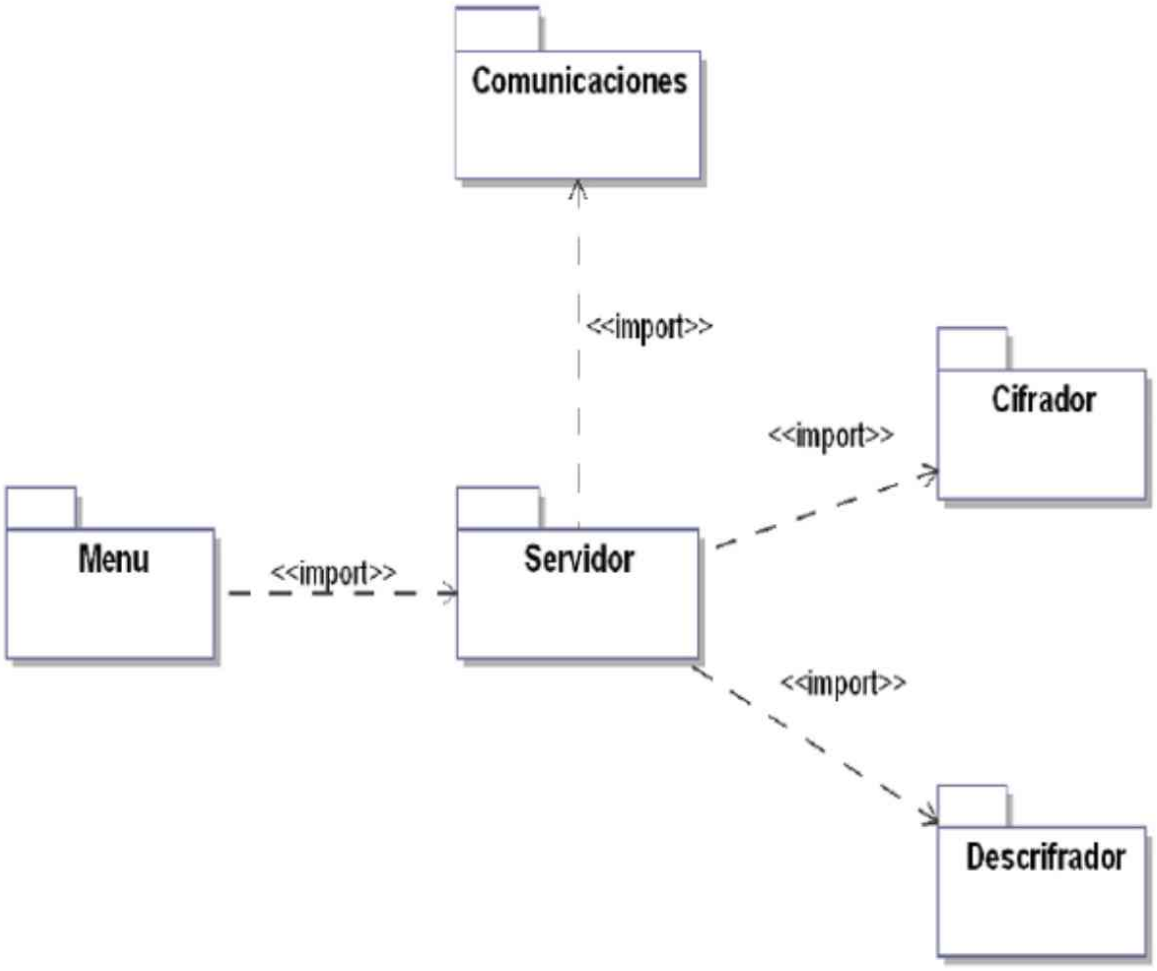


Figura 5.28. Diagrama de paquetes para el lado servidor

De forma análoga a la figura 5.27 se modela el diagrama de paquetes del lado del servidor (figura 5.28). Volvemos a encontrarnos con el paquete *Menu* que contendrá la fachada de interfaz en modo texto; el paquete *Comunicaciones* que que contendrá el componente de conexión a red mediante *sockets*; el *Servidor* que implementará el subsistema con la capa de negocio y será el encargado de invocar la importación de los paquetes *Cifrador* y *Descifrador* que implementan los algoritmos de encriptación y desencriptación de mensajes respectivamente.

---

10 Bass, L., P. Clements y R. Kazman, Software Architecture in Practice, Addison-Wesley, 1998.

11 [OMG1].

12 También se le conoce con el nombre anglosajón de “lollipop”.

13 En la jerga de UML se le reconoce con el nombre de “socket”.

14 Un espacio de nombres (namespace en inglés) permite crear un contenedor abstracto que incluya un conjunto de identificadores definidos independientemente de otros espacios de nombres y así evitar colisiones.

diagramas de clases

«Un error mínimo al principio puede ser máximo al final».
(Aristóteles).

En el capítulo cuatro vimos cómo crear el modelo del dominio a partir de elementos conceptuales aparecidos en la literatura de los de requisitos o en la especificación de los casos de uso. Ahora comenzamos con la fase de diseño detallado, en la cual se intentará modelar los diagramas que desembocarán en la fase de implementación del sistema. Por este motivo, se utilizarán los diagramas de la fase de análisis para detallarlos con más precisión y crear un modelo definitivo de diseño coherente con la estructura del software. Por lo tanto, el trabajo del ingeniero en esta fase es delimitar los subsistemas que conforman la aplicación, definir las clases del modelo del dominio de forma completa y determinar el comportamiento del software de forma dinámica y cómo los objetos interactúan entre ellos para llevar a cabo las funcionalidades del sistema.

En este capítulo se tratará el *modelo estructural estático* con el *diagrama de clases* el cual muestra una estructura conceptual de entidades completas relacionadas, con sus atributos y operaciones que permiten abstraer el dominio de la aplicación. De la correcta organización y composición de las entidades y su distribución en subsistemas en este diagrama dependerán las siguientes fases y la correspondiente implementación, pero ante todo, dependerá del trabajo correctamente hecho en las anteriores fases.

clases

El *diagrama de clases* es el diagrama más importante de la especificación UML. Describe un modelo estático del sistema en términos de las entidades, interfaces, asociaciones, herencias y dependencias. La entidad fundamental de este diagrama es la *clase*, la cual describe en forma de plantilla abstracta a un conjunto de objetos que comparten los mismos atributos, operaciones, relaciones y semántica. Las clases permiten abstraer el dominio del problema mediante unidades que se instancian como objetos. En la siguiente figura se muestra un ejemplo de clase:

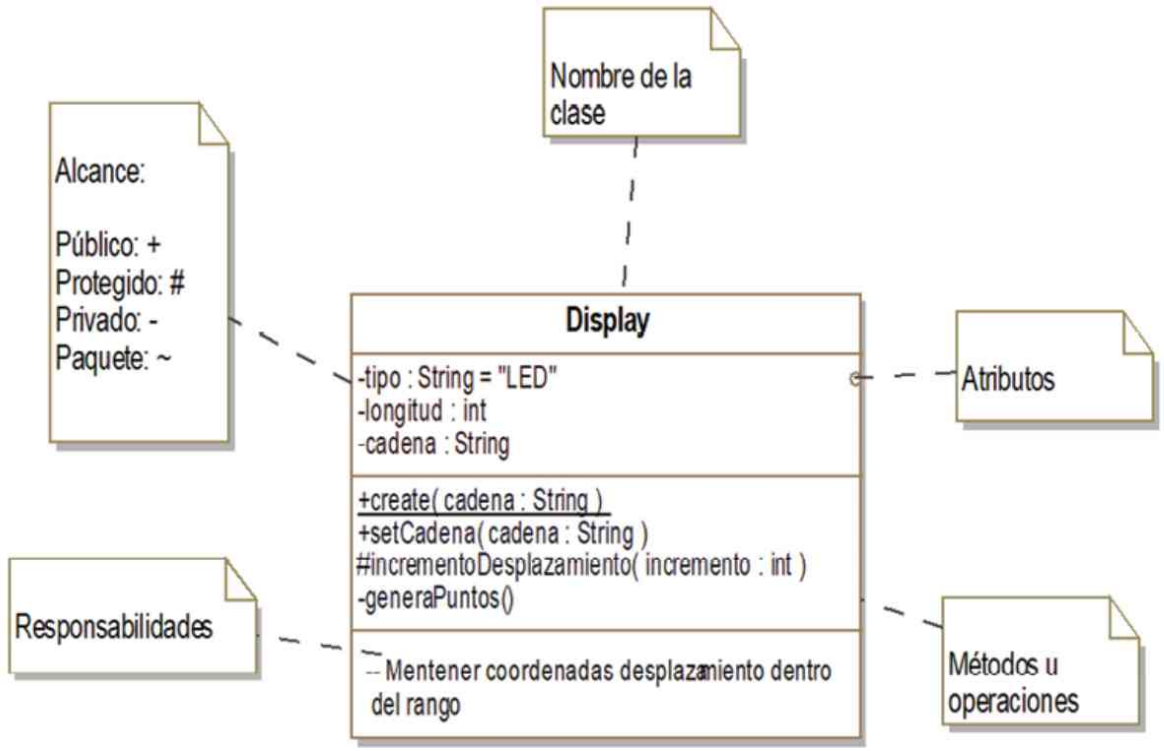


Figura 6.1. Plantilla para una clase en UML

Veamos a continuación cada una de estas partes:

- **Nombre:** Identifica unívocamente a la clase dentro del diagrama de la aplicación. El nombre de la clase debe ser una cadena alfanumérica conteniendo cualquier carácter excepto los espacios en blanco y el operador de resolución de ámbito (::), puesto que este operador

permite resolver un identificador (clase, método o atributo) o un espacio de nombres antes de su utilización. El nombre de clase puede aparecer opcionalmente precedido del nombre del paquete al que pertenece; su sintaxis se define pues, en notación de expresión regular[15]: [paquete::]NombreClase.

Cuando se trate de *clases abstractas* el nombre se especificará en letra cursiva, indicando que dicha clase no se podrá instanciar como objeto.

- **Atributos:** Los atributos describen las propiedades que representan a una clase y se definen mediante una cadena alfanumérica precedida por el tipo de dato. La especificación de los atributos es la primera sección dentro del diseño de la clase. Por lo tanto se definirá un atributo mediante la siguiente notación:

  [alcance] Atributo[:Tipo[multiplicidad]][=Valor inicial]

  Ejemplo:-saldo:float = 200.0

  Donde el alcance o visibilidad indica qué otras clases puede acceder a él. Por tanto, un atributo público (+) es visible para cualquier clase desde fuera relacionada con ella, mientras que un atributo protegido (#) es visible para cualquier descendiente en la jerarquía de herencia y un atributo privado (-) es sólo visible para la propia clase y no puede ser heredado ni compartido con otras clases. El Java, por ejemplo, si no se indica el tipo de visibilidad "*private, public* o *protected*" se adoptará un visibilidad por defecto de paquete (*package*), es decir, pública dentro del paquete. En general, la visibilidad por defecto debería ser privada para los atributos y pública para los métodos.

  Finalmente el ámbito indica el nivel de acceso que tiene el método o el atributo en el contexto general del diagrama. Éste puede ser de dos tipos:

- *De instancia*: cuando cada propiedad del objeto posee un valor diferente al resto de objetos para el misma tipo de clase, por ejemplo, los de la figura 6.1.
- *De clase*: cuando todas las instancias de una clase comparten el mismo valor. Nos referimos en este caso a las variables estáticas de Java o C++[16] y los constructores. Por ejemplo, en la figura 6.2.

- **Operaciones:** Las operaciones son las acciones o funcionalidades básicas que ofrece el objeto de la clase hacia el resto de los objetos o para la propia gestión del estado interno. El resultado de la ejecución de una operación (frecuentemente conocida como *método* o *función miembro* de una clase) producirá el cambio de estado del objeto al recibir el mensaje generado por otro objeto, lógicamente al modificar éste el valor de sus atributos. Las operaciones se definen mediante expresiones regulares como:

  [visibilidad]nombre([parámetros])[:tipoRetorno]

  y donde los parámetros se especifican de acuerdo a la siguiente forma:

  [dirección]nombre:tipo[multiplicidad][=valorPorDefecto]

  Ejemplo:

  +setEnergia(vIni:int = 10, vFin:int = 0):boolean
  +setTiempo(tiempo:Time = Time(2,54,27))

- *Dirección*:

- *in:* Parámetro de entrada (por defecto). Paso por valor.
- *out:* Parámetro de salida. Paso por referencia.
- *inout:* Parámetro de entrada/salida. Paso por referencia.

- *Tipo:* Existen varios tipos primitivos predefinidos en UML 2.x como: *Integer*, *Boolean*, *String* que debemos utilizar a menudo para conseguir independencia de la plataforma; no obstante si trabajamos con un lenguaje de programación específico como Java, C++, etc. podemos utilizar naturalmente sus tipos primitivos, siempre y cuando se vean reflejados en la implementación posterior.

La *multiplicidad* en el caso de los atributos y las operaciones hace referencia a tipos de datos con múltiples elementos, es decir, colecciones de datos en forma de vectores. Ejemplo:

direcciones: String [10]	*Vector de 10 Strings*
alturas: Integer [1..*]	*Vector de 1 elemento a infinito*
reservados: Boolean [*]	*Vector indefinido*
usuarios: String [0..10]	*Vector de 10 elementos que incluye la posibilidad de NULL (0)*

Tabla 6.1. Multiplicidad de de tipos de datos en atributos y operaciones

Los valores posibles para la visibilidad o alcance de la operación son los mismos que se han visto anteriormente para el caso de los atributos.

- **Responsabilidades:** Una responsabilidad es un contrato u obligación que una clase tiene con sus clientes. En esta sección suele indicarse la funcionalidad de la clase en el conjunto del diagrama. Generalmente

no se suele representar por los diseñadores de UML.

De igual forma que vimos que las clases podían definirse como abstractas, también las operaciones pueden definirse de igual modo. Cuando ocurre esta situación definiremos el método en letra cursiva. En programación, las clases que contienen este tipo de operaciones abstractas requieren que las subclases proporcionen los mismos métodos con igual nombre y parámetros que en la clase abstracta pero con una implementación o redefinidas también como abstractas. A este método sobrescrito se le denominará concreto. No es necesario definir la operación concreta en el diseño UML de la subclase, puesto que ésta queda sobreentendida por el contexto.

En UML, los atributos u operaciones estáticos se resaltan dentro de la clase subrayándolos para indicar que todas las instancias comparten la misma dirección de memoria (ver figura 6.2)

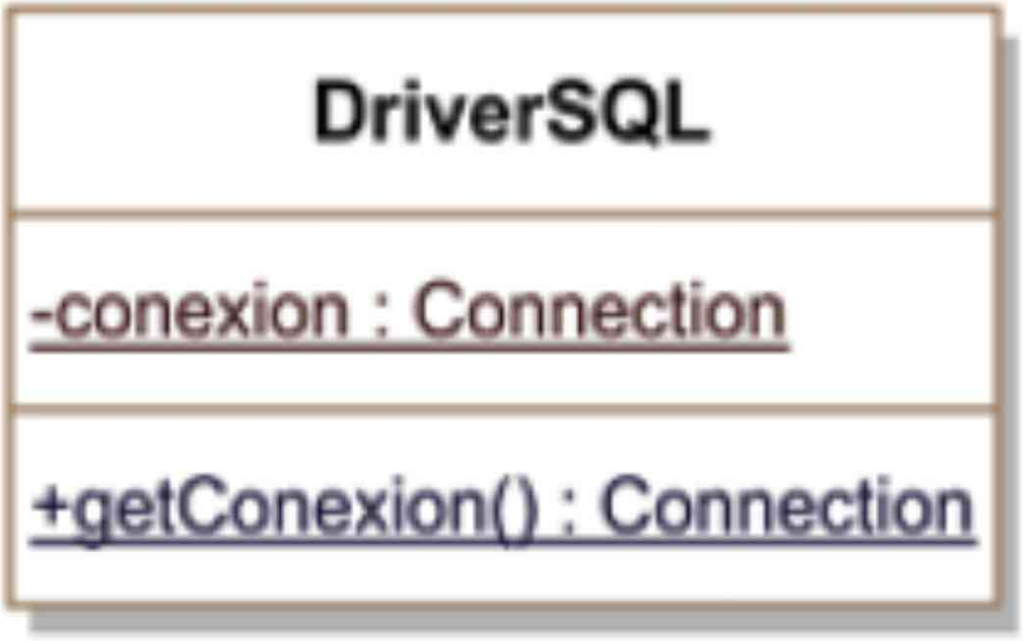


Figura 6.2. Clase con atributo y método estáticos

Los atributos estáticos solo tienen una copia de su valor y siempre están

accesibles aunque no exista ninguna instancia de la clase en el ámbito del programa. De igual forma, los métodos estáticos se pueden invocar sin que se requiera instancias de la clase y únicamente pueden utilizar los atributos estáticos declarados previamente. En general, la utilización de métodos y atributos estáticos no se considera una práctica puramente Orientada a Objetos sino más bien una reminiscencia de la programación estructurada.

La información que se ha incluido en la figura 6.1 es mucha más de la que se suele mostrar: una clase puede representarse con el símbolo de un rectángulo que contiene su nombre como se muestra en la figura 6.3. También es común omitir parte de la información de un apartado (sobre todo con los métodos de acceso *set* y *get*). Cuando se intenta abreviar un método o una parte de una sección recurriremos a utilizar los puntos suspensivos (...) para indicar que la representación de una clase o un método es abreviada. También se utilizarán los puntos suspensivos para indicar que faltan atributos y operaciones que no se ha considerado necesario detallar para el propósito del diagrama en cuestión.



Figura 6.3. La representación de clase más sencilla posible

Finalmente, las instancias de clase se denominan *objetos,* los cuales se diferencian por el valor único de cada uno de sus atributos (estado) en un momento dado en el transcurso de la ejecución del proceso asociado a la aplicación.

asociaciones

La asociación en UML es una relación semántica entre dos clasificadores que permite relacionar a estos dependiendo del papel que jueguen en el modelo. Especifica qué instancias de una clase están conectadas a otra, la forma y el número de veces. No se trata de una relación fuerte, es decir, el tiempo de vida de los objetos relacionados es independiente de la misma.

En la figura 6.4 podemos observar la representación gráfica de una asociación, en la que básicamente se dibuja una línea que conecta dos clases en ambos extremos. Los *roles* indican el papel que juegan los objetos dentro de la asociación. Por ejemplo, para una determinada clase, su rol suele indicarse en el otro extremo de la asociación, justo al lado de la otra clase a la que asocia. Además se pueden añadir opcionalmente diversas indicaciones que representan el contexto del dominio de la aplicación, como el *nombre* de la asociación, que en el caso de utilizarlo debemos prescindir de los roles. De igual forma, un *triángulo relleno* nos indicará cómo es la *dirección de lectura de la asociación* (en el caso de que sea simétrica). La *navegabilidad* nos indicará la posibilidad de acceder a nivel de programación desde el objeto origen al objeto destino, es decir, implica la visibilidad de los atributos y métodos públicos del objeto destino por parte del objeto origen. La navegabilidad, como veremos en los capítulos de implementación, consiste en un atributo privado de una clase que referencia una o varias instancias de la otra clase de la asociación, ya sea de forma unidimensional o multidimensional. Por defecto la navegabilidad es bidireccional. La *multiplicidad*, también conocida como cardinalidad, nos ayudará a especificar el número de objetos que se relacionan con uno o varios objetos de la clase asociada.

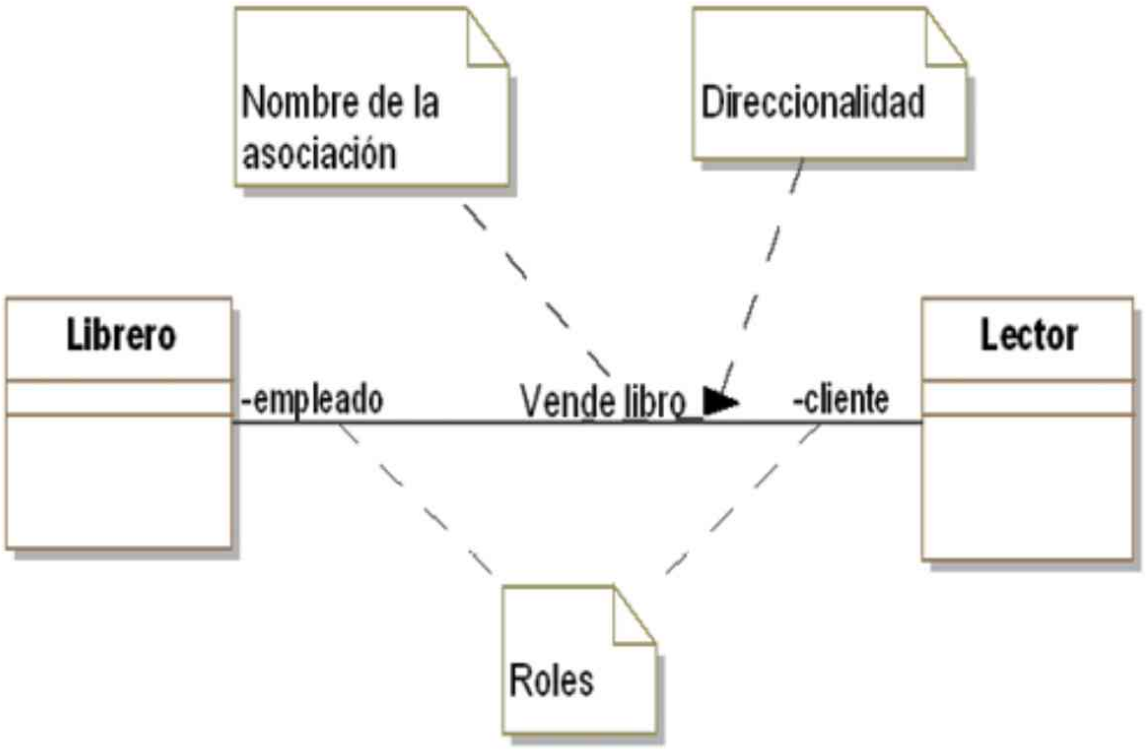


Figura 6.4. Ejemplo de una asociación con navegabilidad simétrica

Finalmente, la *calificación* es un atributo de la asociación que permite localizar un objeto de la asociación cuando éste es múltiple en el otro extremo. (véase figura 6.6). Cuando se representa la calificación se reducen las multiplicidades a uno con el propósito de indicar la referencia al único objeto buscado. En el ejemplo se localiza un empleado en concreto mediante el atributo DNI de la clase *Empleado*.

Una asociación *reflexiva* relaciona entre sí a objetos de una misma clase de forma que existe un enlace a modo de árbol o grafo de instancias interrelacionadas. En la figura 6.9 puede verse un tipo de asociación reflexiva.

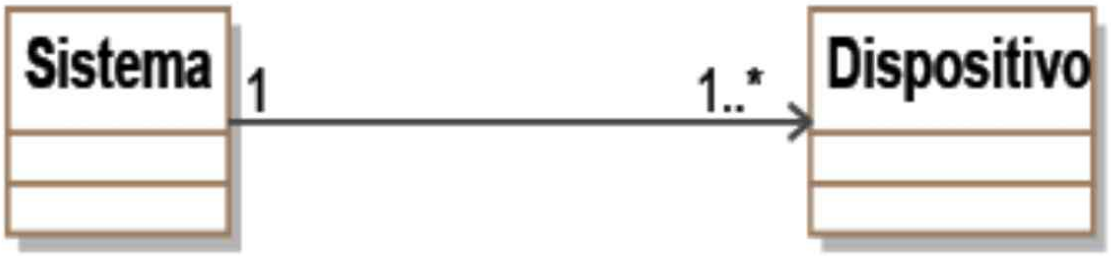


Figura 6.5. Navegabilidad desde Sistema a Dispositivo





Figura 6.6. Calificación

Cuando las asociaciones pueden involucrar a más de dos clases se les denomina ternarias, cuaternarias, etc.

En algunos casos es necesario crear una clase para una asociación del tipo ternaria, sobre todo en relaciones muchos-a-muchos. Las clases de asociación se modelan mediante la utilización de una clase intermedia entre las dos clases participantes, de la cual parte una línea discontinua que une las dos entidades relacionadas.

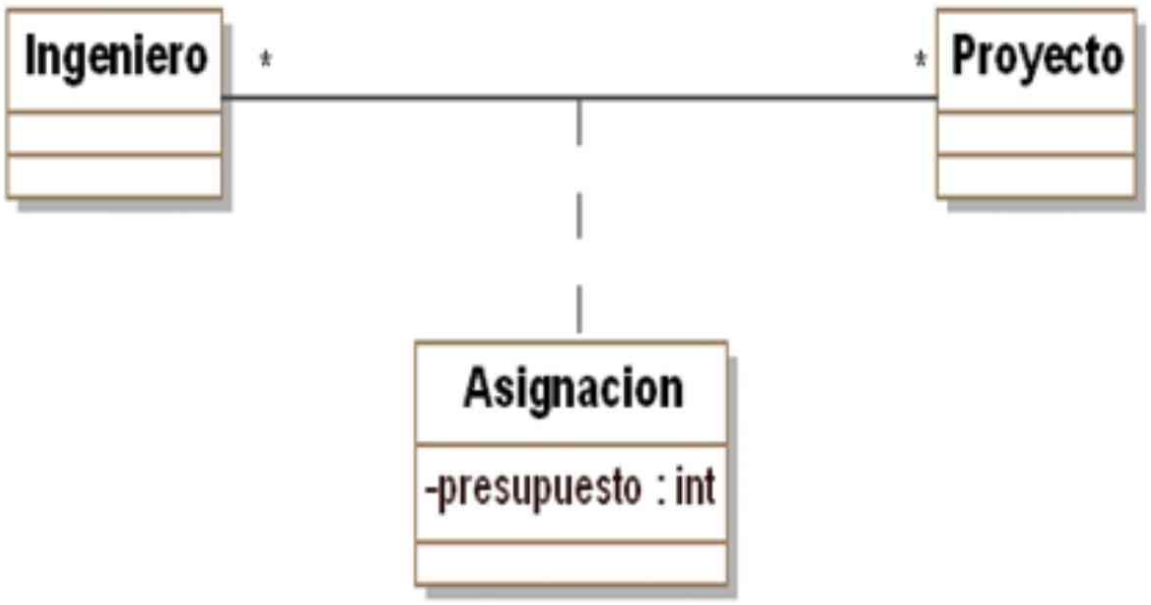


Figura 6.7. Clase asociación

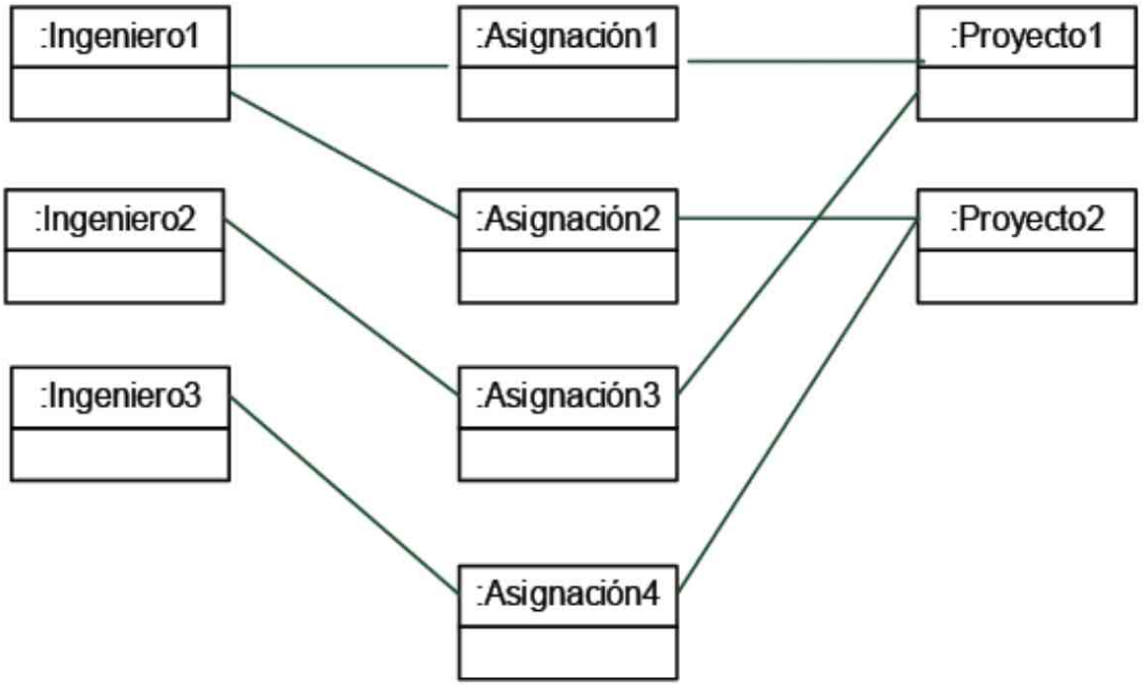


Figura 6.8. Asignación entre pares de objetos (diagrama de objetos)

Por cada par de objetos *Ingeniero-Proyecto* existe un objeto *Asignación* que contiene información sobre la relación de esos dos objetos. En la figura 6.8 el Ingeniero1 y el Ingeniero2 trabajan ambos en el Proyecto1, mientras que el Ingeniero3 trabaja únicamente en el Proyecto2. Las instancias de la clase

Asignación permitirán listar o acceder a la información de las parejas relacionadas.

Multiplicidad

La multiplicidad o cardinalidad indica el número de veces que un objeto está relacionado con otro. Como se comentó en el capítulo cuatro en la tabla 4.1, la multiplicidad puede utilizarse de diferentes formas que pasamos a recordarlas de nuevo:

- *Números concretos: E*specifican una sola posibilidad. Por ejemplo 4, que indica que el objeto se relaciona únicamente con cuatro instancias de otro objeto.
- *Intervalos:* Indican un rango de posibilidades. Por ejemplo 2..6 indica que el objeto se puede relacionar con el intervalo de instancias de mínimo *dos* a máximo *seis*.
- *Asterisco:* El símbolo de (*) significa ninguno o muchos. Quiere decir que el objeto se relaciona con una cantidad indeterminada de objetos que pueden abarcar desde el cero al infinito.
- *Combinación de elementos anteriores:* Por ejemplo: 1..3, 6, 8..* (que significa entre 1 y 3, ó 6 únicamente, ó 8 ó más de 8, es decir, los valores 0, 5 ,7 no estarían permitidos).

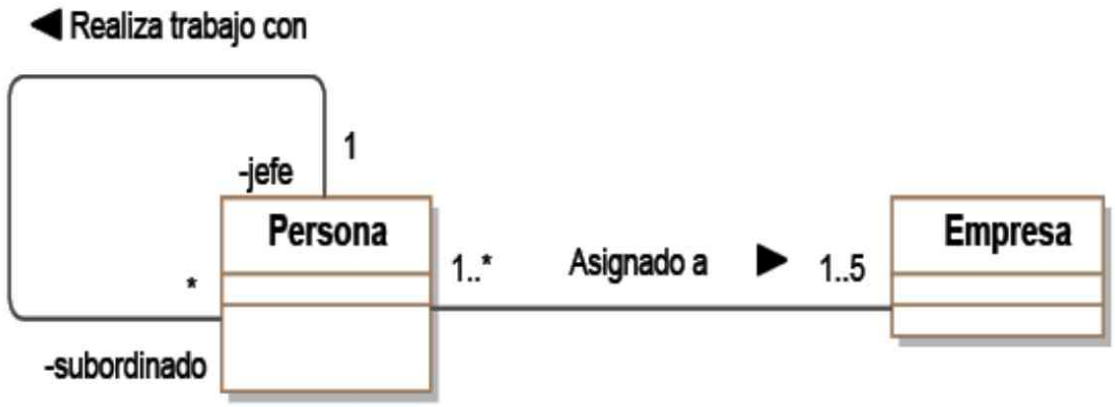


Figura 6.9. Relación reflexiva de la clase Persona

Para el ejemplo representado en la figura 6.9 se hacen las siguientes suposiciones:

- Una persona debe estar asignada como mínimo a una empresa y como máximo a 5, mientras que a una empresa se le debe asignar una o muchas personas.
- Una persona puede realizar el trabajo junto a muchas o ninguna persona. Esta relación se muestra como reflexiva.
- La relación reflexiva impone una jerarquía de roles entre los jefes y los subordinados.

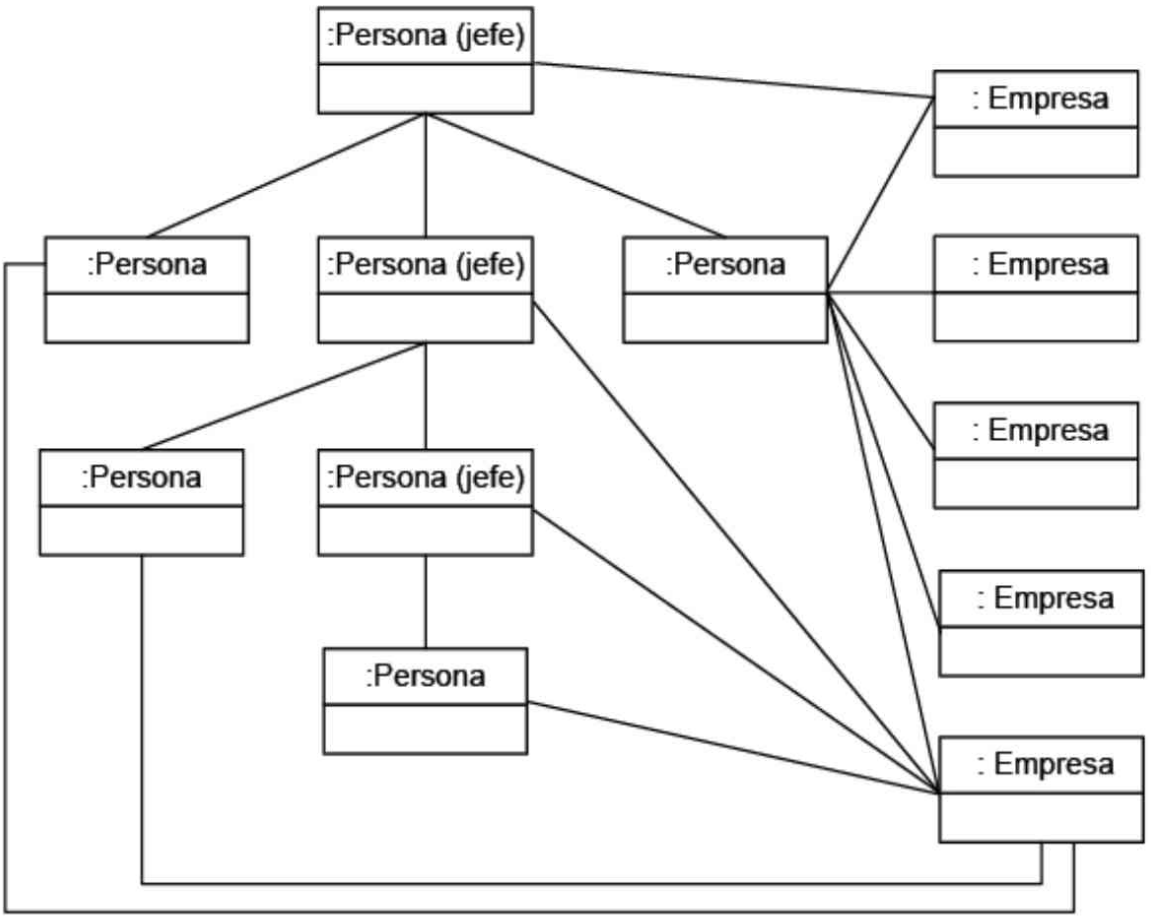


Figura 6.10. Diagrama de objetos de la figura 6.9

Agregación y composición

Un caso típico de la notación UML que a menudo lleva a confusión a los diseñadores de software es la diferencia entre *agregación* y *composición*. Estas notaciones sirven para modelar elementos que se relacionan utilizando expresiones como: "*es parte de*", "*tiene un*", "*consta de*", etc.

La agregación y la composición son dos clases especiales de asociación en la que se indica la forma de integrarse las diferentes clases contenedoras y contenidas. De esta manera, utilizaremos un símbolo de *agregación* que indicará una asociación del tipo todo/parte cuando las partes no son totalmente necesarias para formar el todo o no ponen en cuestión su existencia. Se trata de una relación débil y se representa con un símbolo de diamante blanco tal como se muestra en la figura 6.11.

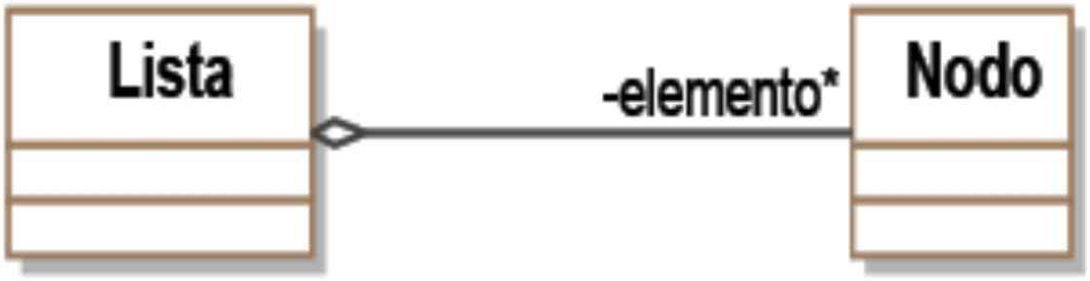


Figura 6.11. Agregación con rol

Sin embargo, en la composición se indica que la/s parte/s son necesarias para la concepción del todo, por tanto la parte contenedora es responsable de la eliminación de las partes contenidas. En una composición, cada componente pertenece a un todo y sólo a uno (en la figura 6.11 no se cumple esto, puesto que un nodo podría pertenecer a varias listas). La composición es un tipo de relación fuerte y se representa mediante un diamante de color negro. En la figura 6.12 se aprecia un ejemplo de composición:

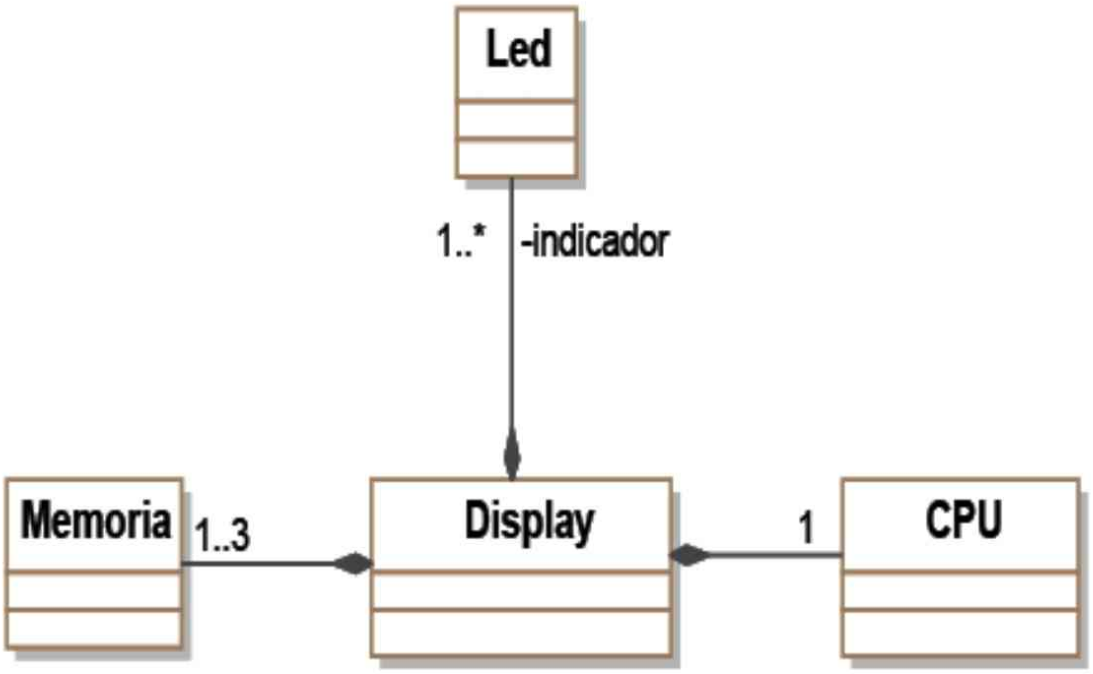


Figura 6.12. Ejemplo de composición

En este caso son fácilmente identificables las partes componentes de un "display" para mostrar texto deslizante (*scroller*). Los componentes o partes son imprescindibles para la conformación y semántica del todo. En este caso, un "display" requiere de forma imprescindible tres módulos de memoria, leds y una CPU para funcionar.

Es importante, tanto en la composición como en la agregación, evitar situaciones donde se generen ciclos sobre varias instancias de clase participantes.

herencia

La herencia indica que la subclase comparte todo o parte de las operaciones y atributos definidos en la superclase. Este tipo de conexión se lee como "*es un tipo de*". En la figura 6.13 se muestra un ejemplo de representación:

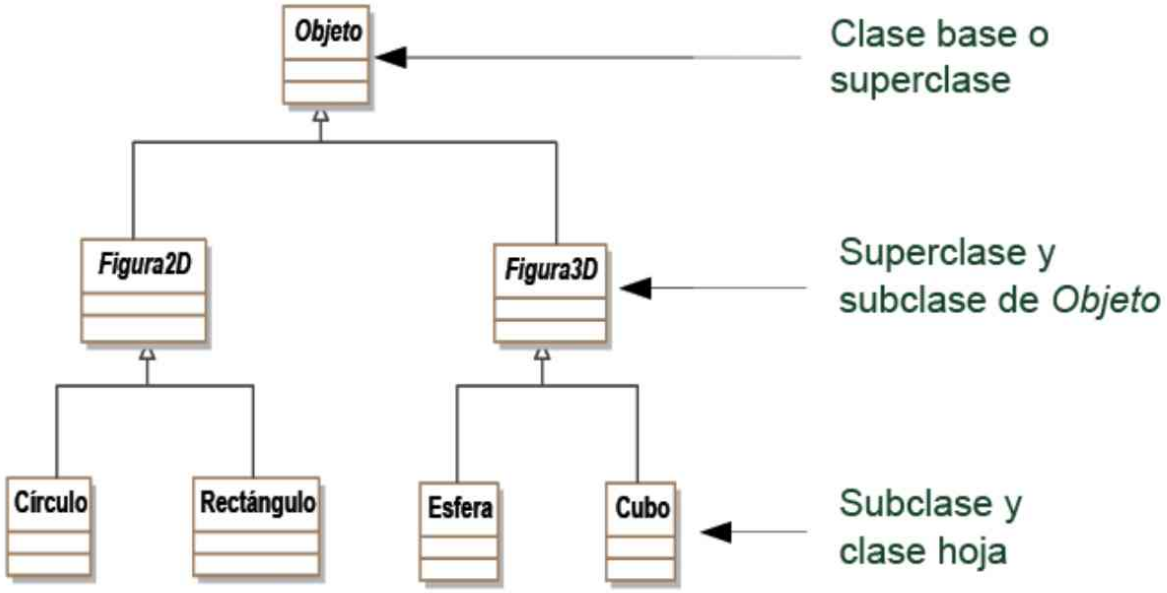


Figura 6.13. Herencia (Los nombres de las clases abstractas se escriben en cursiva)

Las clases pueden optar por no heredar de ninguna otra clase, heredar de una sola clase (herencia simple) y heredar de muchas clases (herencia múltiple). La red jerárquica de clases facilita la organización y taxonomía de los conceptos utilizados en el domino de la aplicación[17]. Según [Meyer00], la herencia es un potente mecanismo para tratar con la evolución natural de un sistema y con modificación incremental.

La generalización también facilita el *polimorfismo*. Dicha propiedad de los lenguajes orientados a objetos es la posibilidad de aplicar un comportamiento común a un conjunto de clases heredadas especificando las funciones de la clase base como virtuales. De esta forma, al crear una variable de un tipo base que referencia (mediante un puntero en C++) a un objeto inferior en la jerarquía, las funciones de las clases base virtuales serán redirigidas a las

funciones concretas en el objeto inferior mediante ligadura dinámica.

Uno de los beneficios de la herencia es el del ahorro de código, pues la construcción de objetos es más sencilla en los que heredan porque delegan parte de su estructura. El problema de la generalización en software orientado a objetos es el acoplamiento, es decir, si una clase base cambia será necesario recompilar todo el sistema. Por lo tanto es aconsejable huir de la excesiva generalización en la medida de lo posible y sustituir la herencia por la composición. En general la herencia debe evitarse en la mayor medida de lo posible, pues es motivo de pérdida de legibilidad del código y de la disminución del rendimiento neto de la aplicación. Además es un buen principio de abstracción del diseño aumentar la cohesión y reducir el acoplamiento entre módulos.

Interfaces

Las interfaces son una especificación de responsabilidades y permiten definir parte de las operaciones de entidades del modelo. Las interfaces únicamente definen el prototipo de la función[18] como si se tratara de una clase esquemática, a expensas de ser implementadas en otras clases. Son importantes cuando varias clases implementan la misma interfaz y por tanto permite, a partir de un puntero a ella, controlar un conjunto de objetos que la implementan. Obviamente las interfaces no se pueden instanciar como objetos.

Las clases que requieren implementar una determinada interfaz en UML lo hacen por medio de la realización, que es similar a la herencia pero que se representa mediante una línea discontinua acabada en triángulo.

En la figura 6.14 puede observarse las variantes de representación en el diagrama de clases de una interfaz. A la izquierda se muestra la estructura más utilizada comúnmente en UML, mientras que a la derecha se representa la versión simplificada con una línea acabada en una circunferencia.

Todos los elementos de la interfaz son públicos y no pueden tener atributos. Los iconos que representan la interfaz deben ser identificados con el estereotipo <<interface>>. Al igual que en la generalización, las interfaces permiten crear jerarquías de interfaces, mediante las relaciones de herencia entre ellas, tal como se muestra en la figura 6.15. Esta práctica suele ser típica en diseño de contenedores de listas, árboles y grafos de Java o la STL de C++.

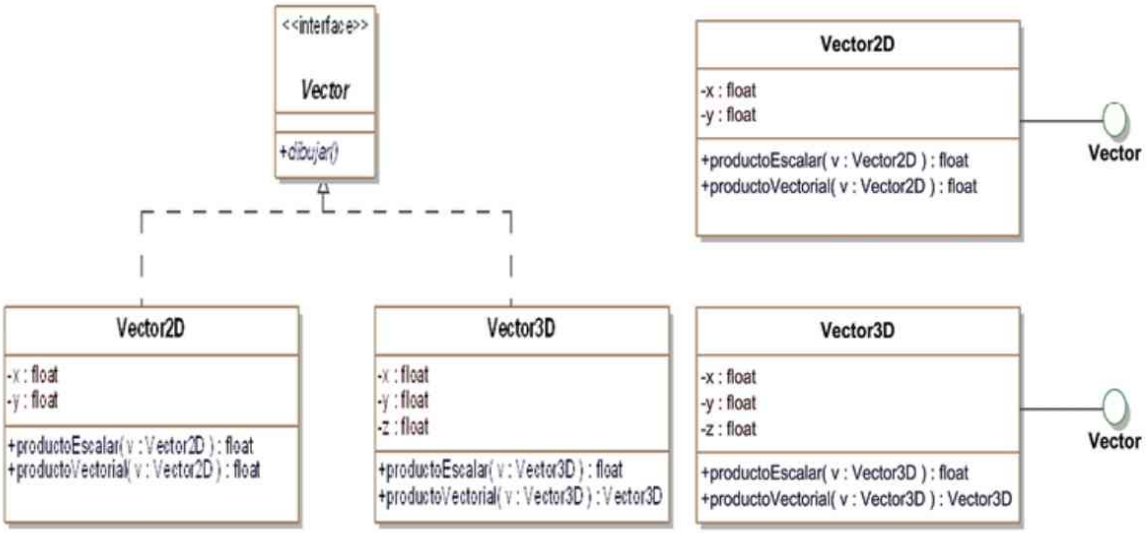


Figura 6.14. Dos representaciones de la interface Vector

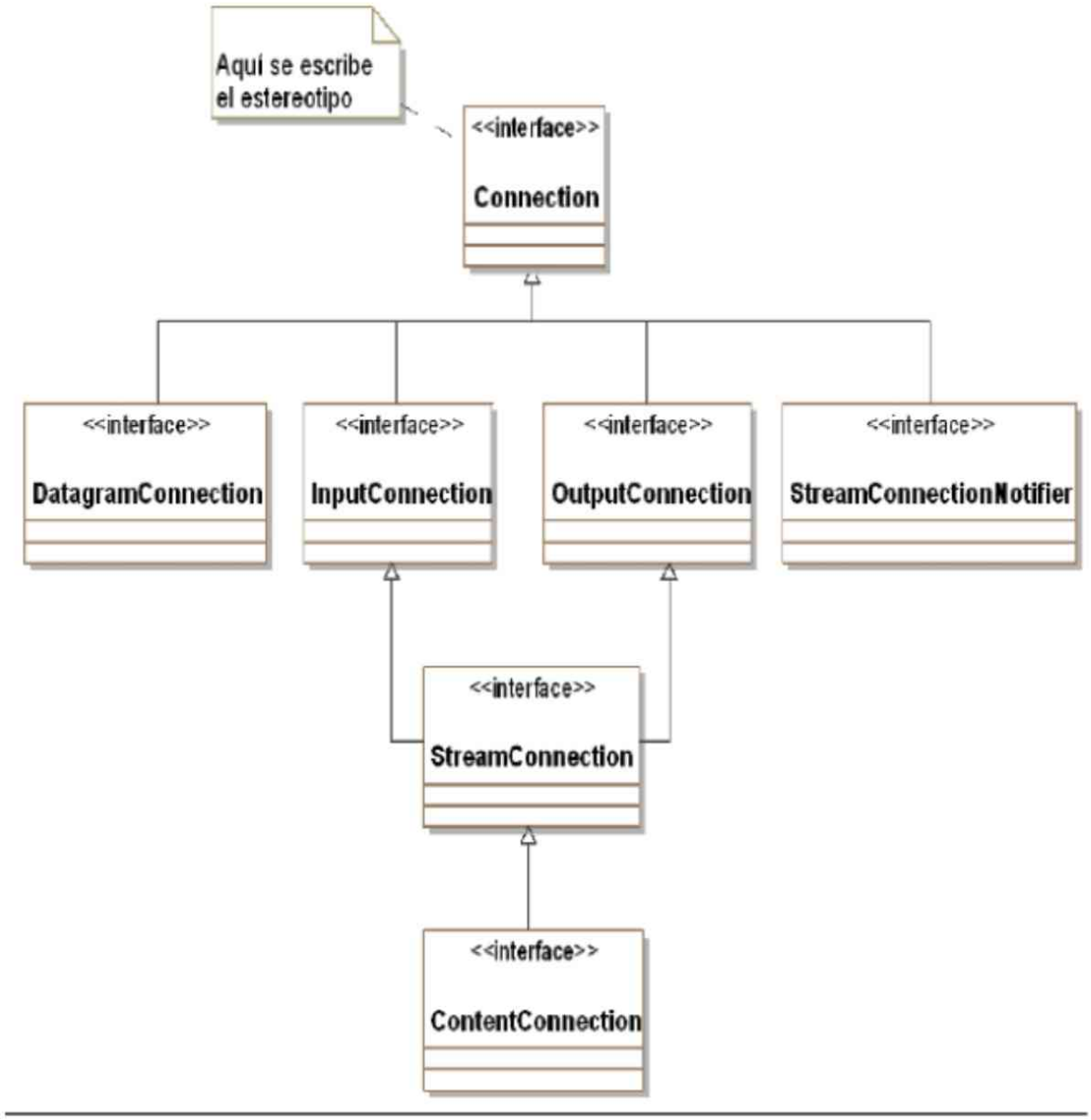


Figura 6.15. Jerarquía de interfaces en una de las librerías de Java

dependencias

Las relaciones de dependencia son un tipo de relación entre entidades de UML. Este tipo de relación suele ser unidireccional y se interpreta muchas veces como una relación "de uso", entendiéndose que una clase "usa" a otra cuando ésta implica el paso de argumentos o el retorno de valores entre operaciones. Se suele denotar como una línea discontinua como se muestra en la figura 6.16. En este ejemplo el método *listarProductos()* de la clase *Tienda* requiere obtener los precios de todos los productos para calcular el total de ganancias. Mediante el retorno del objeto *Producto* por medio del método *getProducto()* de *Almacén* es posible aplicar la dependencia.

El término <<use>> suele omitirse en la relación de dependencia. En general las dependencias no se suelen representar a menos que el diseñador lo considere oportuno.

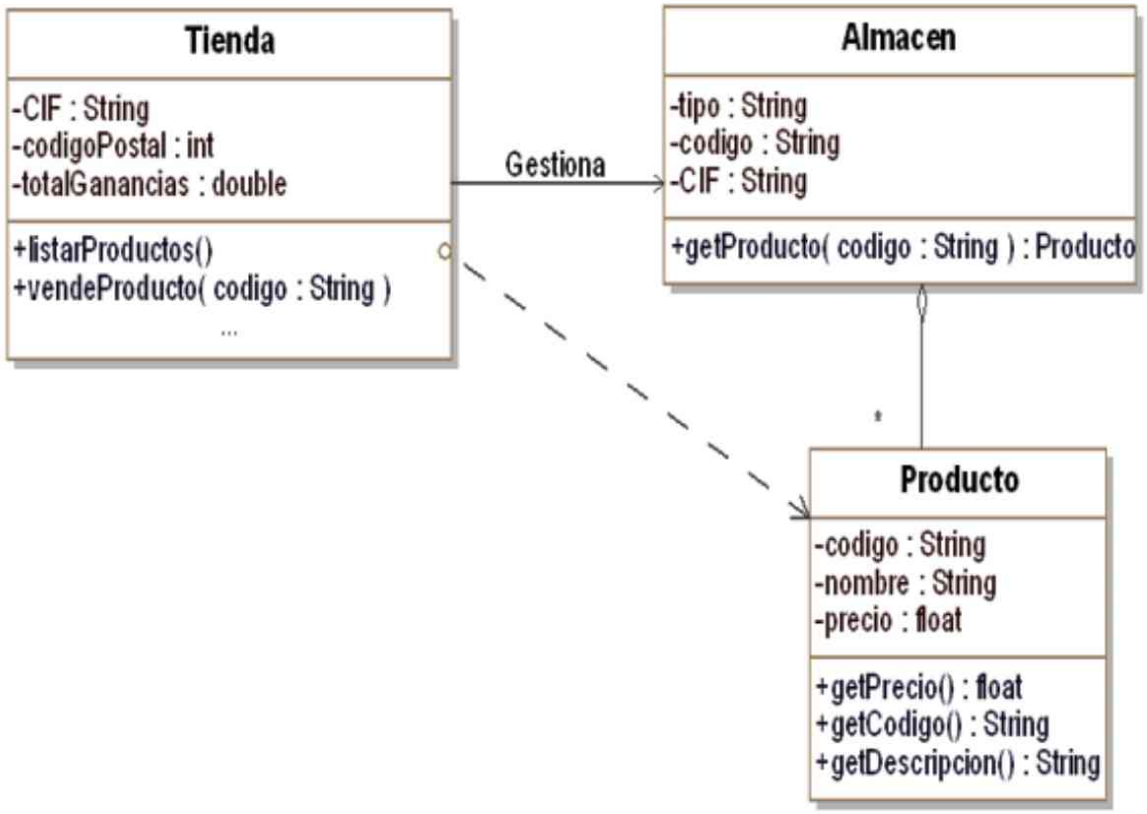


Figura 6.16. Ejemplo de dependencia entre la clase "Tienda" y la clase "Producto"

excepciones

En los diagramas de clases en UML es posible especificar el lanzamiento y la recepción de excepciones. Para dotar a nuestro diagrama de las características ofrecidas por el sistema de excepciones recurriremos al estereotipo <<throws>>.

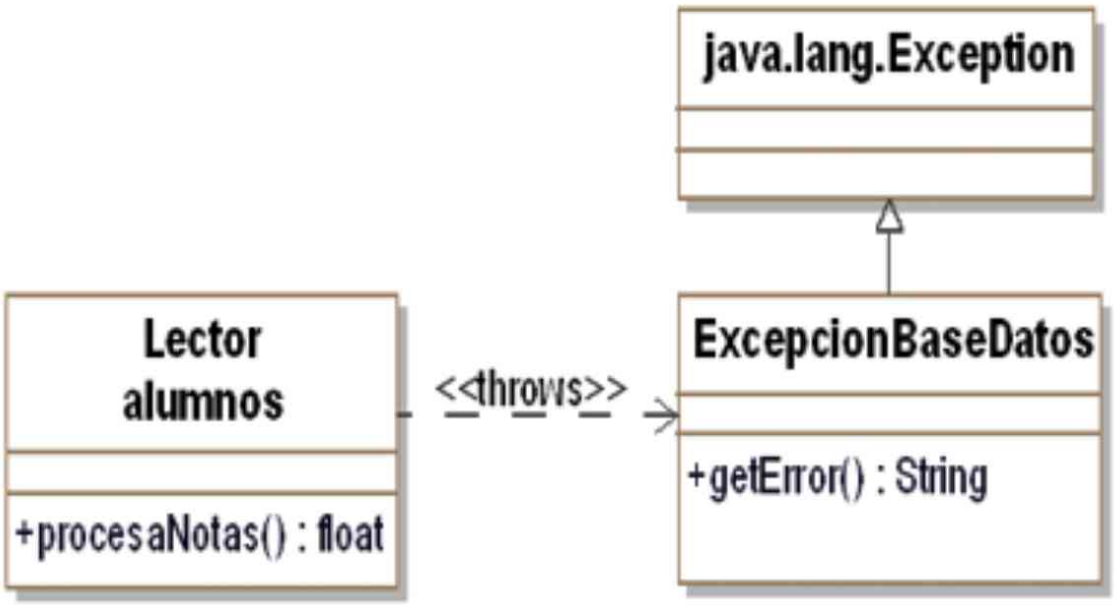


Figura 6.17. Ejemplo de uso de excepciones

clases parametrizables

C++ y Java permiten la utilización de tipos genéricos en clases y funciones de plantilla (templates). La programación genérica permite la independencia del tipo de datos, con lo que se consigue mayor portabilidad y concentración en la secciones de código. En contrapartida, el tamaño del código ejecutable crece considerablemente, entre otros aspectos.

En UML es posible la utilización de clases parametrizables o plantillas dentro del diagrama de clases. Para conseguir esta funcionalidad disponemos de la notación de clase parametrizable como se ilustra en la figura 6.18:

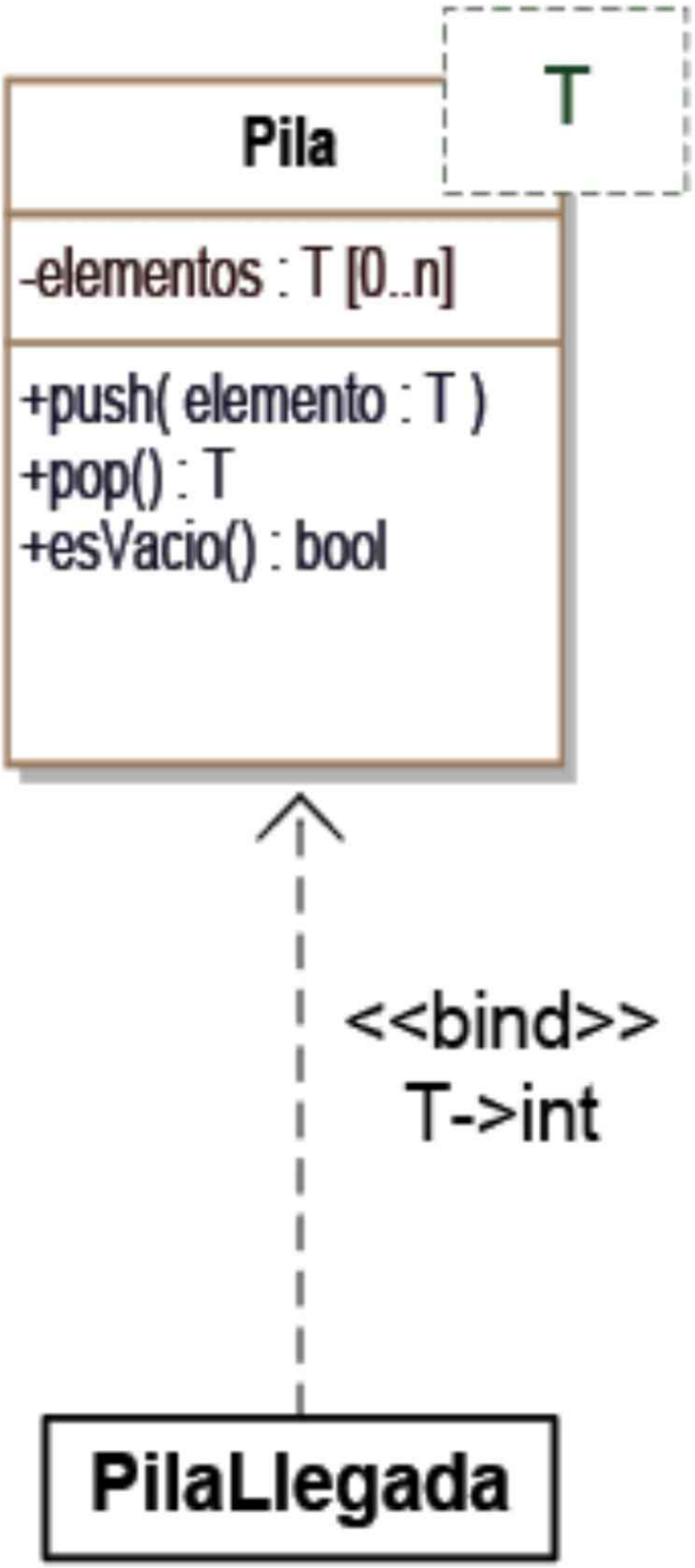


Figura 6.18. Clase plantilla y enlace al tipo concreto mediante <<bind>>



Figura 6.19. Otra forma de enlazar un tipo genérico con un tipo concreto

En la figura 6.19 se ejemplifica una clase *Pila* con un parámetro genérico enmarcado en un cuadro de línea discontinua en la parte superior derecha de la clase. La forma de especificar que otro elemento está utilizando la clase parametrizable es por medio de un enlace con estereotipo <<bind>> y el tipo de dato concreto al cual se adapta.

ejemplo de modelado: "spindizzy"

Veamos ahora cómo sería el modelo de clases del juego "Spindizzy" del que ya se hizo el modelado del dominio en el capítulo cuatro. Para facilitar el diseño recurriremos de nuevo a la descripción del juego en modo textual:

«Un clásico juego en perspectiva isométrica de las plataformas de 8-bits consiste en mover una peonza por un laberinto formado por muchas habitaciones que contienen un conjunto de objetos similares. Cada habitación del laberinto conecta con al menos una habitación y a lo sumo con cuatro. Al comienzo del juego se selecciona una habitación al azar donde comenzará la aventura. Las habitaciones están formadas por unos bloques de ladrillos de varios tipos: de teletransporte a otra habitación del laberinto, movibles y desintegrables. En las habitaciones se encuentran varios objetos para recoger (diamantes y estrellas) para así completar la misión del juego. Además de los objetos recogibles, en la habitación pueden existir otros seres, como Torretas que se mantienen estáticas disparando al jugador; Robots y Octaedros que persiguen a la peonza a una determinada velocidad y alcance. Solo la Torreta puede disparar fuego o rayo, mientras que los Robots y los Octaedros persiguen a la peonza a una determinada velocidad y alcance. Todos los enemigos tienen un escudo de vulnerabilidad y también la peonza; pero esta última únicamente reduce en una unidad su escudo cuando recibe un disparo o golpe».

Entidades seleccionadas	Atributos seleccionados
Laberinto Habitación ObjetoIsométrica: Peonza, Enemigo, Recogibles y Bloque Fotograma Sonido Enemigo: Fijo y Móvil Bloque: Desintegrable, Movible y	Laberinto: *labTimer e índiceHabitac.* Habitación: *contenido e índice* ObjetoIsométrica: *x, y, isoX, isoY y fotograma* Fotograma: *tile* Sonido: *muestra* Peonza: *vidas y escudo* Enemigo: *tipo y escudo*

Teletransporte	Fijo: *tipoArma* Móvil: *alcance y velocidad* Recogibles: *tipo y total* Movible: *dirección*

Tabla 6.2. Sustantivos seleccionados como entidades y atributos del juego

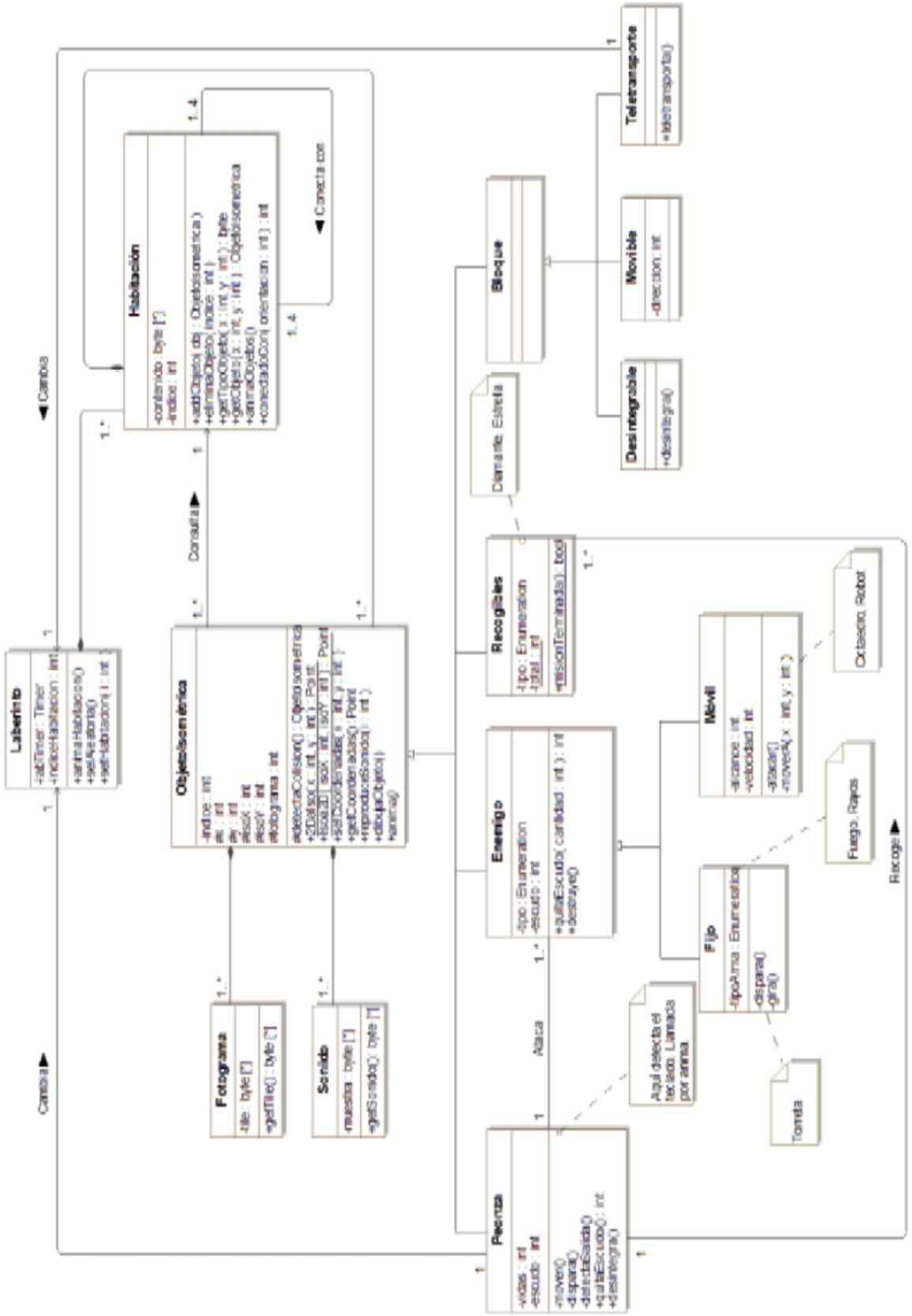


Figura 6.20. Diagrama de clases del juego "Spindizzy"

Como se puede observar en el diagrama de la figura 6.20, se han añadido

nuevas clases y se han simplificado otras que compartían atributos comunes. Puesto que se trata de un juego en perspectiva isométrica[19] es necesario especificar dos atributos para la localización en el array de objetos que se encuentra incluido en la clase *Habitación* en el atributo *contenido*. Dichos objetos podrán ser transformados a perspectiva isométrica con una función especial (*Isoa2D*) y almacenados en los atributos *isoX* e *isoY*. Obviamente los objetos del juego como la Peonza, Enemigos, Bloques, etc. deben tener un conjunto de fotogramas almacenados como *tiles* y apuntados por el índice del atributo *fotograma* de la clase *ObjetoIsométrica*. La mayor diferencia con el modelo del dominio estriba en la clase *Enemigo* que se ha simplificado al crear una jerarquía clasificada como enemigo móvil o enemigo fijo. Esto es así debido a que los dos enemigos móviles: *Robot* y *Octaedro* comparten las mismas propiedades y características de comportamiento y por lo tanto se identificarán ahora por medio de una enumeración. Como podrá observar, se han añadido asociaciones con navegabilidad y direccionalidad entre *Peonza—Laberinto* y *Teletransporte—Laberinto* debido a la posibilidad que tiene la peonza de cambiar de habitación del laberinto de forma secuencial y el bloque de teletransporte de saltar a una habitación de forma indexada. Es por tanto necesario que estas dos clases llamen al método *setHabitación(i: int)* de la clase *Laberinto* por medio de un atributo.

Es de destacar la asociación existente entre *ObjetoIsométrica* y *Habitación* pues como decíamos anteriormente es necesario que cada personaje del juego consulte de forma periódica el contenido del array de objetos situados en una habitación. Es el caso, por ejemplo, del método *detectaColisión()* que debe comprobar qué otros objetos existen en el entorno de un determinado personaje y para ello se sirve de los métodos públicos de *Habitación*: *getTipoObjeto()* y *getObjeto()*. Respecto a la dinámica de la animación, es la clase *Laberinto* la que gestiona el bucle de la animación, llamando a la secuencia de

métodos concretos y virtuales que indicamos a continuación por cada iteración:

Iteración bucle → *Retardo* → Laberinto::animaHabitación() → Habitación::animaObjetos() → *ObjetoIsométrica::anima()*...

Será al implementar las funciones virtuales en las clases derivadas de *ObjetoIsométrica* las que den vida a los personajes y gestionen la dinámica del juego.

caso de estudio: ajedrez

Para el caso de estudio del juego del ajedrez nos encontramos con una situación bien diferente. Necesitaremos crear interfaces para manejar la abstracción de los algoritmos de inteligencia artificial, las heurísticas y la implementación de los recibidores de mensajes provenientes de los objetos de comunicación de frontera.

La figura 6.21 de la siguiente página ilustra el conjunto de clases con atributos, operaciones y asociaciones de forma detallada. El diagrama de clases obtenido es el resultado de la concreción de las abstracciones realizadas en capítulos anteriores con el modelo del dominio, el diagrama de componentes y paquetes. Reelaborando los diagramas de los capítulos anteriores y mediante un análisis minucioso de los requisitos funcionales de la lógica del juego de ajedrez se llega a la conclusión aquí presentada.

Las interfaces que se representaron en el diagrama de componentes de la figura 5.11 aparecen aquí diseñadas como entidades con el estereotipo <<interface>>, con nombre y métodos en cursiva. Las cinco interfaces definidas en el modelo son implementadas en clases concretas que redefinen y amplían su semántica para la gestión íntegra de la aplicación. La mayoría de los componentes y subsistemas aparecen también aquí definidos.

Quizá se haya preguntado acerca del origen de las nuevas clases que no se tuvieron en cuenta en el modelo del dominio, como son el caso de los algoritmos secuencial y GPU[20], la jerarquía de funciones heurísticas y las clases de gestión de red. La respuesta es que estas clases son el resultado de un mayor análisis de los requisitos y responden a la lógica de negocio necesaria para implementarlos.

Se ha creado la clase *Control_juego* que es la encargada de implementar la dinámica de la partida. Ya que es posible jugar en red, la clase *Control_juego* se asocia con multiplicidad 1..2 a la clase *Jugador_humano* para agrupar tanto

al jugador local como al jugador rival. También se modela la clase *IA* para la gestión algorítmica y el asesoramiento estratégico. La superclase *Jugador* mantiene un array con las dieciséis piezas del bando, sus respectivas posiciones, métodos de posicionamiento y dibujo. Finalmente los subsistemas *GUI* y *Comunicaciones* son también detallados con sus métodos gráficos y de comunicaciones sobre Internet respectivamente.

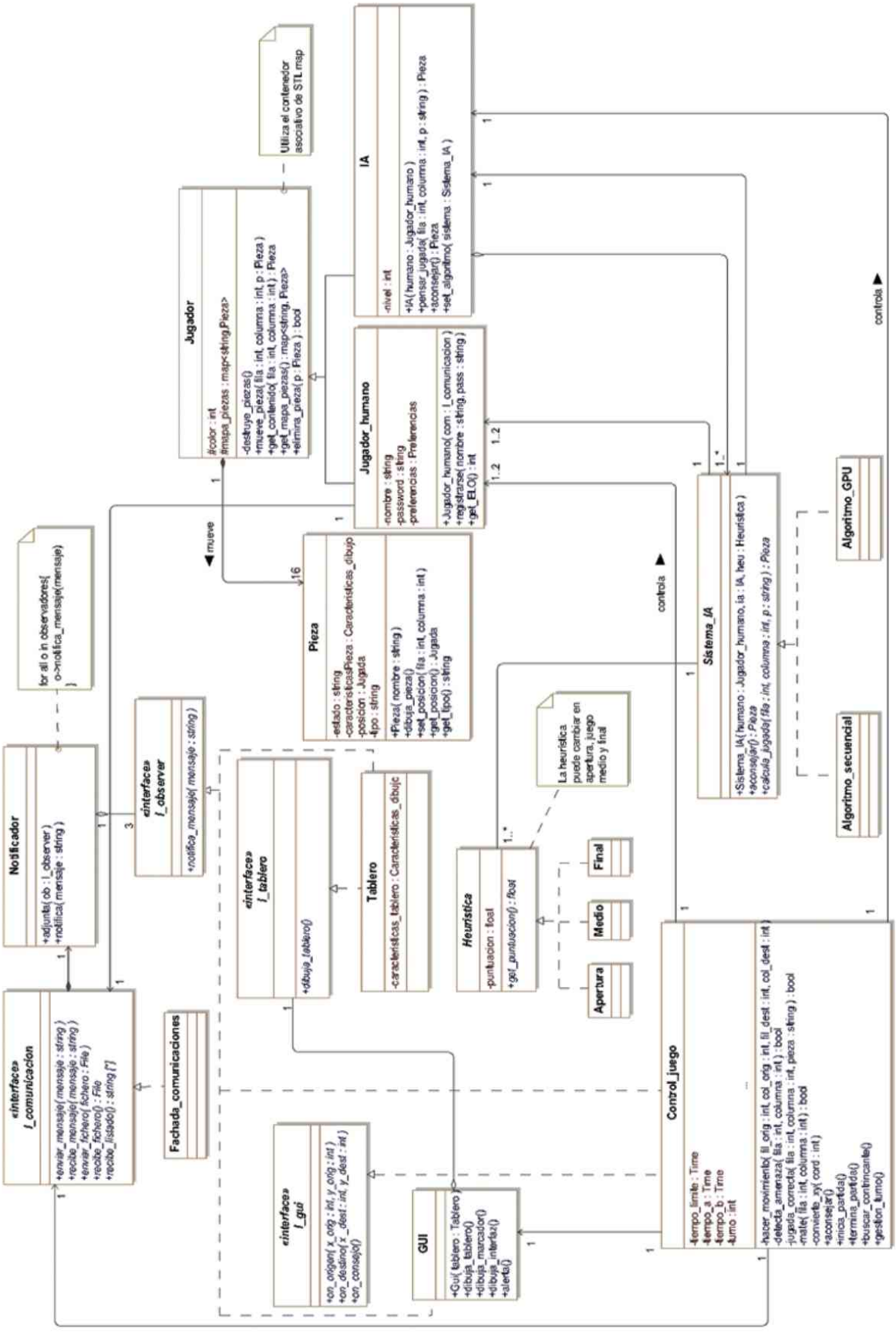


Figura 6.21. Diagrama de clases de Ajedrez

caso de estudio: mercurial

Al igual que en el caso del diagrama anterior del ajedrez, el modelo de *Mercurial* requiere de una análisis detenido y profundo de los aspectos clave de la aplicación.

En el diagrama de la figura 6.22 de la siguiente página se representan al detalle las cuatro interfaces provenientes del modelo arquitectónico de componentes. Igualmente los cuatro subsistemas que se definieron en el modelo de componentes se encuentran aquí completamente representados. Al igual que en el anterior diagrama, se ha recurrido a la notación de los puntos suspensivos para indicar que las sección de atributos y métodos no se han extendido a un nivel de aplicación real.

En el diseño del modelo de clases de Mercurial se recoge la representación de la clase *FachadaComunicaciones* que es la responsable de recibir mensajes de protocolo por el *socket* y notificarlo a la clase *Notificador*. La clase *Notificador* se encarga a su vez de avisar a todas las clases que implementan dicha interfaz por medio del método virtual *notificaMensaje(mensaje:String)*. De momento no hablaremos más de este subsistema ya que entraremos más en detalle en este tema en el capítulo dedicado a patrones.

La clase *FachadaInterprete* implementa un mini-intérprete de comandos en modo de texto para ser transferidos a la jerarquía de clases de *Usuario*. Esta asociación se realizará a través de la interfaz *IGestorCliente* que es la encargada de definir el paso de la clase *Comando*. Definiremos la clase *Comando* como el resultado del procesamiento de una línea de texto de comandos. Por tanto, dicha estructura contiene una serie de atributos y métodos de acceso que proporcionan el tipo de operación y sus argumentos.

La jerarquía de *Usuario* muestra por un lado la clase *ClienteProgramador,* que se asocia con un sistema de ficheros local[21], y la clase *Administrador* que implementa los roles de administrador de los repositorios. Finalmente, el

subsistema *Gestor archivos* define la interfaz *IDirectorioAbstracto* que será implementada en las clases *Fichero* y *Directorio* con el doble objetivo de representar una estructura de datos recursiva y reutilizar un patrón de diseño.

Los diagramas de clases que se han modelado anteriormente no son la única solución al problema, usted mismo podría crear un modelo diferente. ¡El diseño UML es muy subjetivo y depende mucho del punto de vista del diseñador!

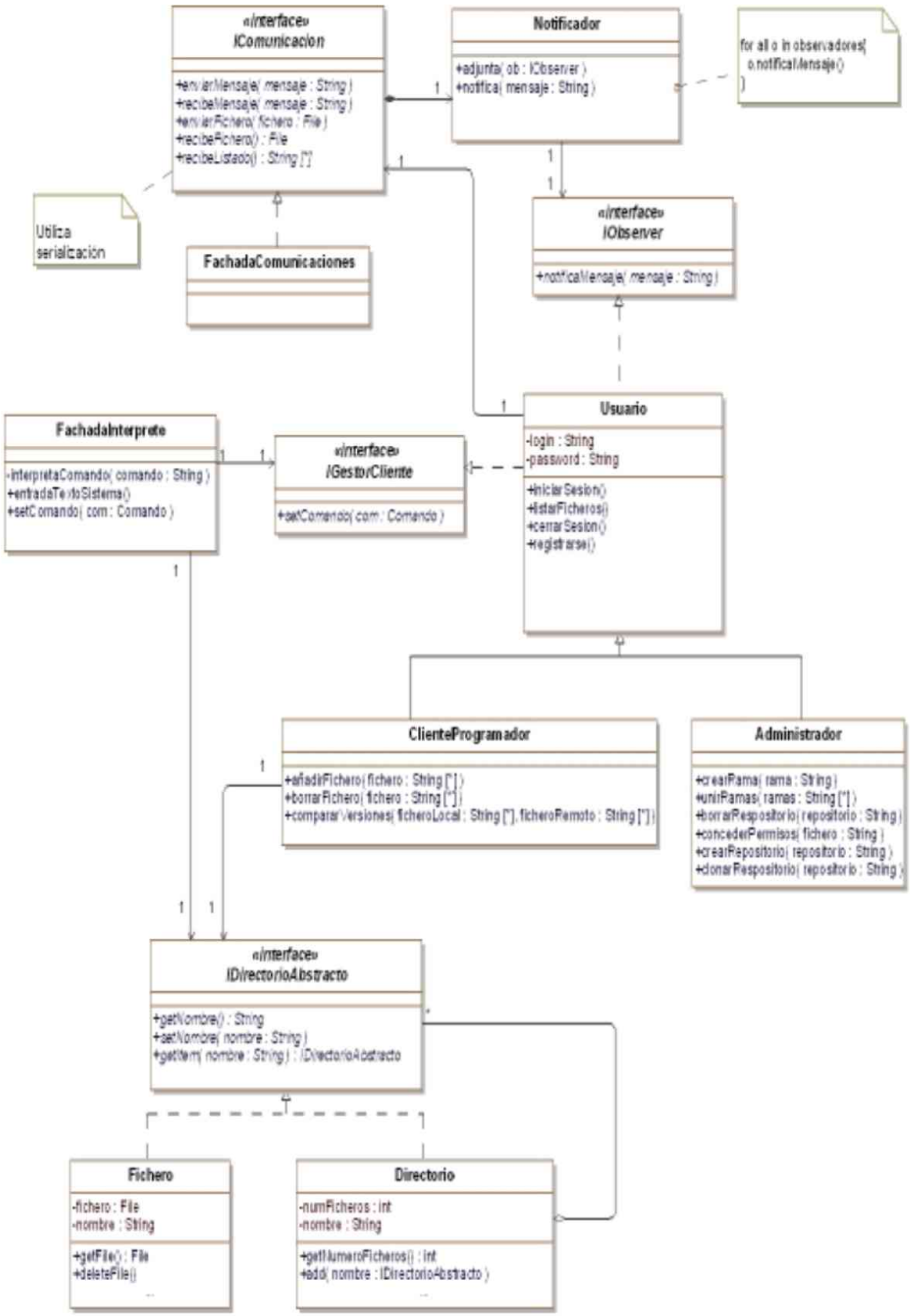


Figura 6.22. Diagrama de clases de Mercurial

caso de estudio: servicio de cifrado remoto

Lado cliente

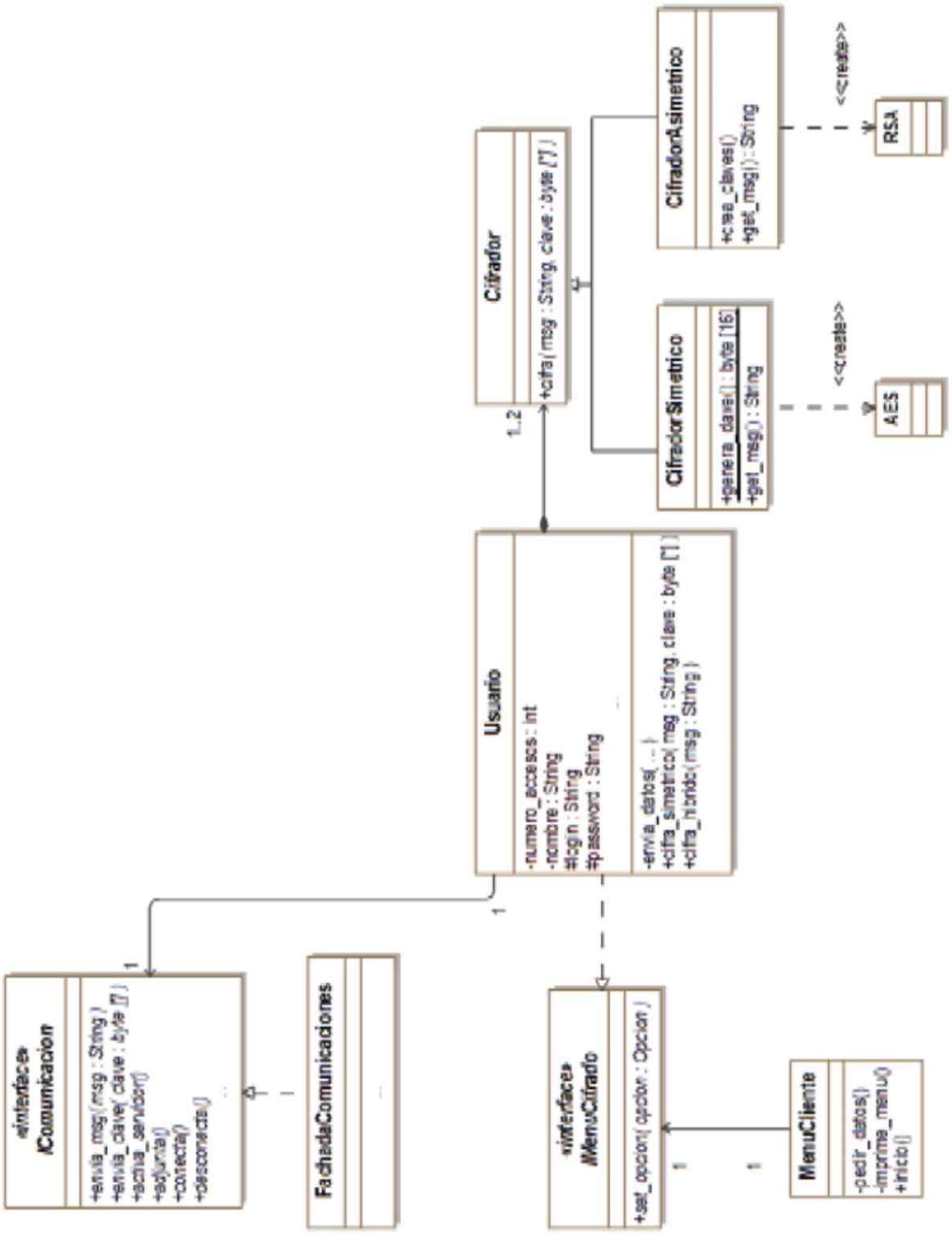


Figura 6.23. Diagrama de clases del lado cliente

Como en los dos casos anteriores, los diagramas elaborados durante las fases de análisis de requisitos y el diseño arquitectónico nos servirán ahora para

realizar el análisis detallado del diagrama de clases tal como muestra en el modelo de la figura 6.23.

En primera instancia observamos las interfaces *IComunicacion* e *IMenu-Cifrado,* vistas en el diagrama de componentes de la sección 5.2.7 del capítulo anterior, y que implementan el nexo con el que se invocarán a las clases *FachadaComunicaciones* y *MenuCliente* respectivamente. La clase *Fachada-Comunicaciones* realizará la implementación de la interfaz IComunicacion y será la responsable de proporcionar las funciones de red mediante *sockets.* Más abajo, la clase *MenuCliente* invoca a la interfaz *IMenuCifrado* e implementa la clase que crea el modo de texto para el cliente.

La clase *Usuario* es la análoga a la dispuesta en el modelo del dominio y viene a representar al usuario iniciado que solo puede cifrar mensajes. Esta clase, además, realiza la implementación de la interfaz *IMenuCifrado* con el fin de aceptar las llamadas proporcionadas por el subsistema menú y gestionar las opciones correspondientes. Así mismo, mantiene una referencia a la interfaz *IComunicación* con el fin de realizar las llamadas a las funciones de red.

Finalmente, se proporciona, desde la clase *Usuario,* una composición hacia la herencia de la interfaz *Cifrador* con la idea de instanciar una clase *CifradorSimetrico,* cuando el usuario requiera cifrado simétrico, y una clase *CifradorSimetrico* y *CifradorAsimetrico,* cuando el usuario requiera un algoritmo de cifrado tipo híbrido. De esta forma se entiende la razón de establecer la cardinalidad a 1..2.

Tanto la clase *CifradorSimetrico* como *CifradorAsimetrico* serán las responsables de crear las clases de la librería criptográfica para los algoritmos AES y RSA respectivamente según corresponda.

Lado servidor

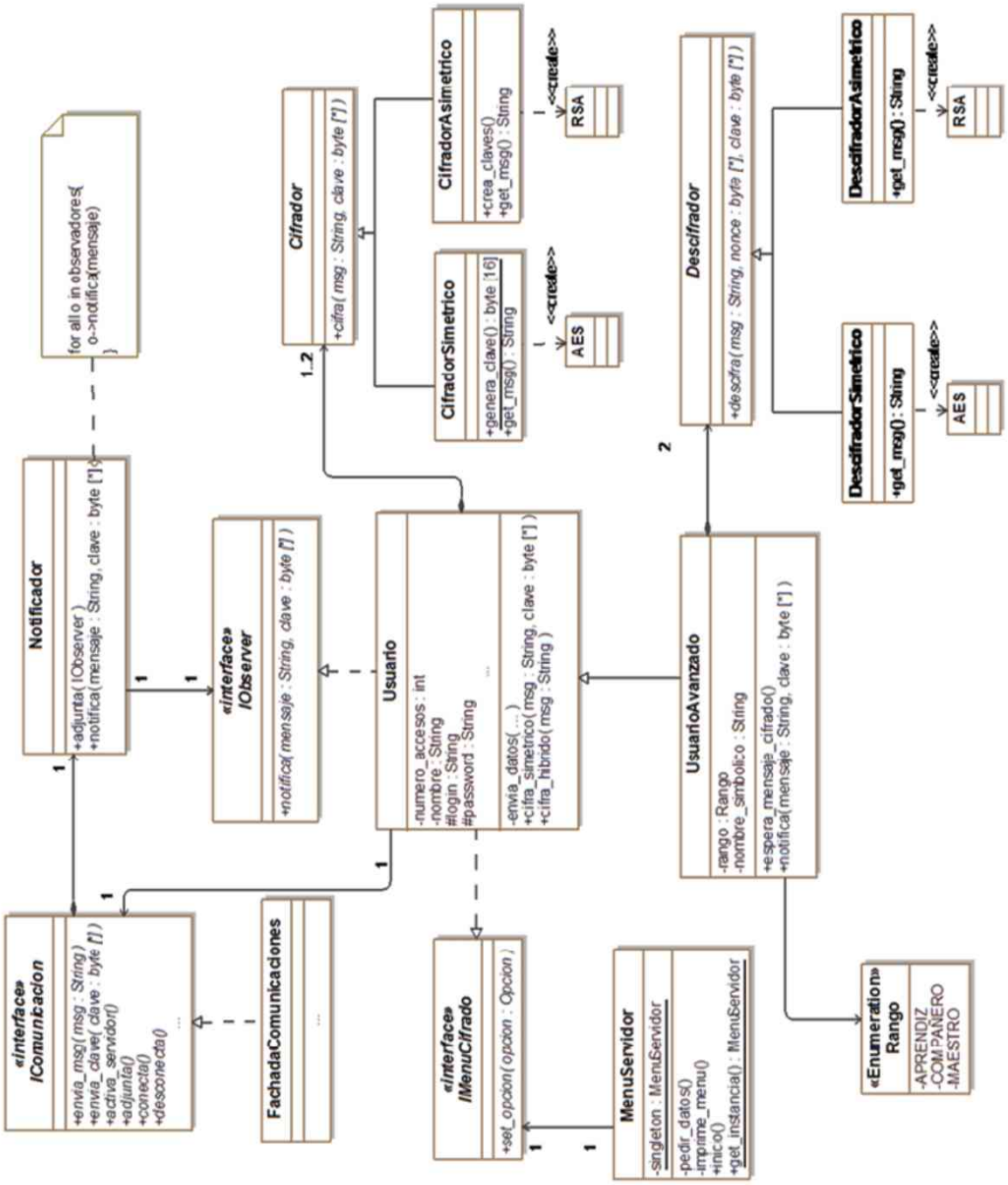


Figura 6.24. Diagrama de clases del lado servidor

Respecto al diagrama de clases del lado del servidor, apreciamos una ligera diferencia en la clase *MenuServidor* para implementar el patrón *Singleton,* de manera que solo exista una instancia de la aplicación servidor en el sistema. Dicho patrón se analizará con más detenimiento en el capítulo nueve

dedicado a patrones de diseño.

La interfaz *IComunicación* se reserva la responsabilidad de ser la puerta de entrada a las llamadas a la clase *FachadaComunicaciones,* teniendo ahora la función adicional de adjuntar los observadores representados por la interfaz *IObserver.* Dichos observadores son gestionados por la clase *Notificador* y serán invocados tan pronto como un mensaje cifrado o una clave de sesión cifrada llegue a *FachadaComunicaciones,* mediante una llamada al método *notifica* de la clase *Notificador*. Este conjunto de clases anteriormente mencionadas conforma el patrón *Observer* que se explicará con detenimiento en el capítulo nueve.

La clase *Usuario* será la que implementará la interfaz *IObserver,* pero ésta no tendrá validez hasta que se herede en la clase *UsuarioAvanzado* y se sobreescriba el método *notifica*. Pues, una vez que la clase *UsuarioAvanzado* recibe un mensaje cifrado desde el citado método, se procede a someter dicho mensaje a los dos tipos de algoritmos que utiliza el sistema mediante la invocación del método abstracto *descifra* de la interfaz *Descifrador.*

Como se aprecia en el modelo, todo el resto de las clases permanece igual que en el lado cliente, como *Usuario,* que será la superclase que luego especialice la clase *UsuarioAvanzado*. Vemos aquí, por tanto, un caso de herencia que extiende las posibilidades de la superclase en la subclase.

El último detalle a añadir al modelo es el tipo enumerado *Rango,* que se referencia desde la clase *UsuarioAvanzado,* y permite tener una clasificación del tipo de usuario avanzado en el sistema.

---

15 En una expresión regular los corchetes "[...] " indican optatividad.

16 Una variable o método estático se define en C++ y en Java con el modificador static.

17 En Java no existe la herencia múltiple a nivel de clase, pero es posible a nivel de interface.

18 En C++ se representan mediante funciones virtuales puras.

19 Para más información sobre la perspectiva isométrica consultar la URL:
http://gamedevelopment.tutsplus.com/tutorials/creating-isometric-worlds-a-primer-for-game-d
gamedev-6511

20 El algoritmo GPU utiliza una versión multiprocesador para la aceleración en paralelo de los algoritmos de cálculos estratégicos.

21 La versión del servidor tiene una estructura de archivos similar.

diagramas de secuencias

«Los hombres construimos demasiados muros y no suficientes puentes».

(Isaac Newton).

El modelo de comportamiento permite describir el funcionamiento dinámico de la aplicación, representar las transiciones entre los diferentes estados y las principales interacciones de los objetos del dominio en tiempo de ejecución. Los diagramas de comportamiento que se tratarán en este libro se clasifican *en diagramas de interacción* que a su vez se componen de *diagramas de secuencias* y *de comunicación*. Dentro de los diagramas de comportamiento también encontramos los *diagramas de actividad* y los de *estado*. Dichos diagramas intentan modelar la evolución y la dinámica de los objetos durante su tiempo de vida y los eventos generados en ellos. Es importante destacar que los diagramas de interacción pueden estar ubicados tanto en la fase de análisis como en la de diseño, pues su sintaxis describe con fidelidad ambos contextos.

En este capítulo profundizaremos acerca de cómo los diagramas de secuencias permiten describir los objetos que interactúan entre sí a largo de una línea de tiempo en un determinado contexto de la aplicación. Veremos la correspondiente notación UML, los conceptos básicos para modelar diferentes tipos de situaciones en un lenguaje estructurado e ilustraremos ejemplos prácticos que le orientarán en la tarea de modelado de secuencias basados en casos de uso.

conceptos preliminares

Previo a explicar la creación de diagramas de secuencia con todas sus peculiaridades y situaciones reales, haremos un breve hincapié en los términos y conceptos clave relacionados con los diagramas de interacción de este capítulo y los siguientes.

El elemento básico de un diagrama de interacción es el *mensaje*. Éste permite intercambiar información entre instancias a modo de estímulo.

En consecuencia, en un diagrama de interacción figuran:

- **Instancias**[22]: Son las principales entidades derivadas de los diagramas de clases, de componentes, de casos de uso y de las que se envían o reciben un conjunto de operaciones. Estas instancias llevan consigo un estado asociado (cambios en sus atributos) correspondiente a las acciones que se ha realizados sobre ellas. Son los protagonistas de los diagramas de interacción.
- **Acciones:** Las acciones se dividen tanto en predefinidas (<<create>> o <<destroy>>) como otras definidas por el desarrollador.
- **Operaciones:** Hacen referencia a los diferentes servicios que pueden solicitarse de las instancias. Estos servicios se corresponden con los métodos que se han definido para una determinada clase.
- **Estímulos:** Las invocaciones de operaciones pueden ser de dos tipos

    - **Síncronas:** las instancias en comunicación tienen que estar sincronizadas, esto es, la instancia que invoca se queda bloqueada hasta recibir una respuesta. En general, las llamadas síncronas son las más usuales en los diagramas de secuencias.
    - **Asíncronas:** Es la contraria a la anterior, es decir, la instancia que invoca la operación continúa su ejecución sin detenerse a esperar respuesta.

estructura básica

En el diagrama de secuencias se muestran los objetos ya instanciados y los mensajes que se intercambian a lo largo del tiempo. Dentro del diagrama de secuencias pueden incluirse elementos de otros diagramas, como por ejemplo, los actores de los casos de uso y elementos de los diagramas de robustez. En UML los objetos se pueden representar como un rectángulo con el nombre y el tipo subrayados, aunque en las últimas versiones de UML ya no es necesario este subrayado.

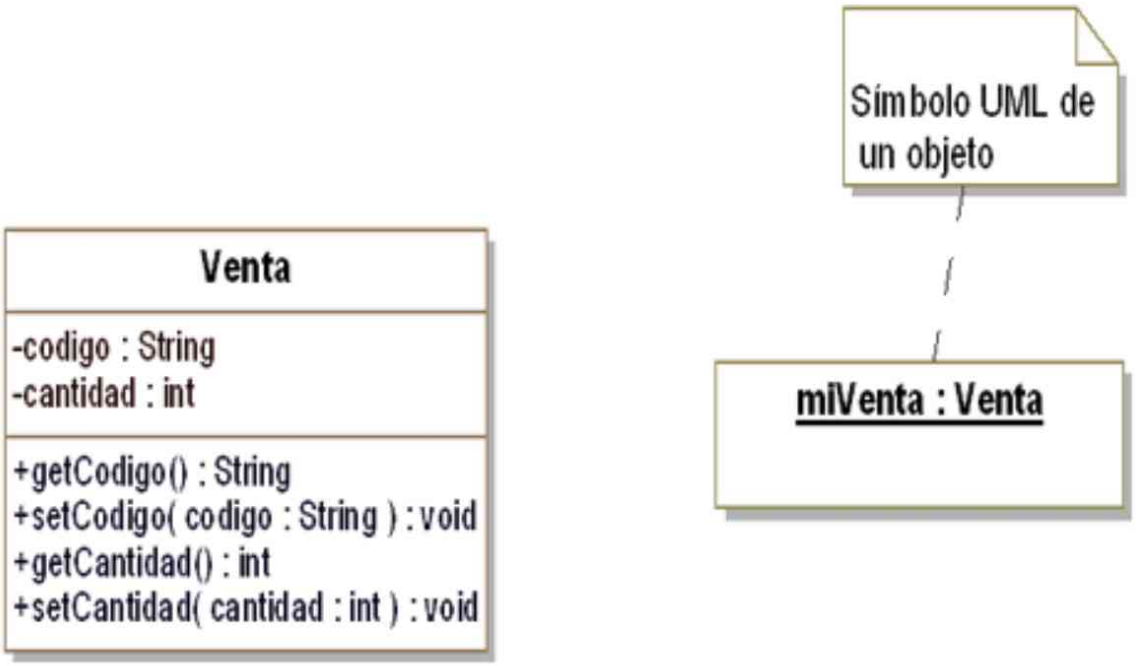


Figura 7.1. Ejemplo de clase (izquierda) e instancia de la misma (derecha)

Explicaremos a continuación cada uno de los elementos que conforman un diagrama de secuencias:

Línea de vida

Es una definición visual de la evolución del tiempo dentro del escenario donde se desarrolla las interacciones entre objetos. Se representa por una línea discontinua vertical y con la instancia del objeto en la parte superior de la misma.

En el ejemplo de la figura 7.2 se muestran dos objetos con sus respectivas líneas de vida. El *Objeto1* envía un mensaje síncrono al *Objeto2* que será el receptor de dicha operación o acción. Después de la ejecución del mensaje la evolución de la secuencia continúa.

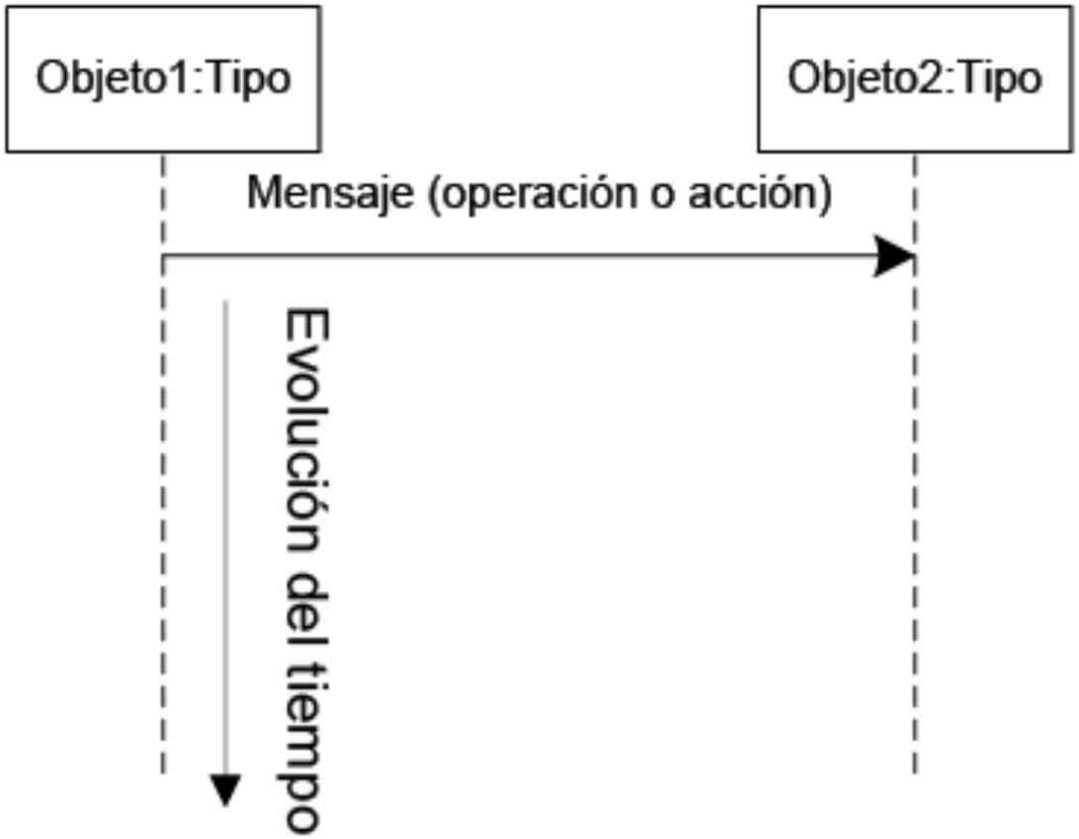


Figura 7.2. Línea de vida y mensaje

Activación

La activación se produce cuando un objeto recibe un estímulo y procede a llamar a su respectivo método de ejecución. Un rectángulo blanco vertical sobre la línea de vida indicará el tiempo de procesamiento del mensaje recibido. Un ejemplo se muestra en la figura 7.3, en la que el actor "vendedor" realiza una venta.

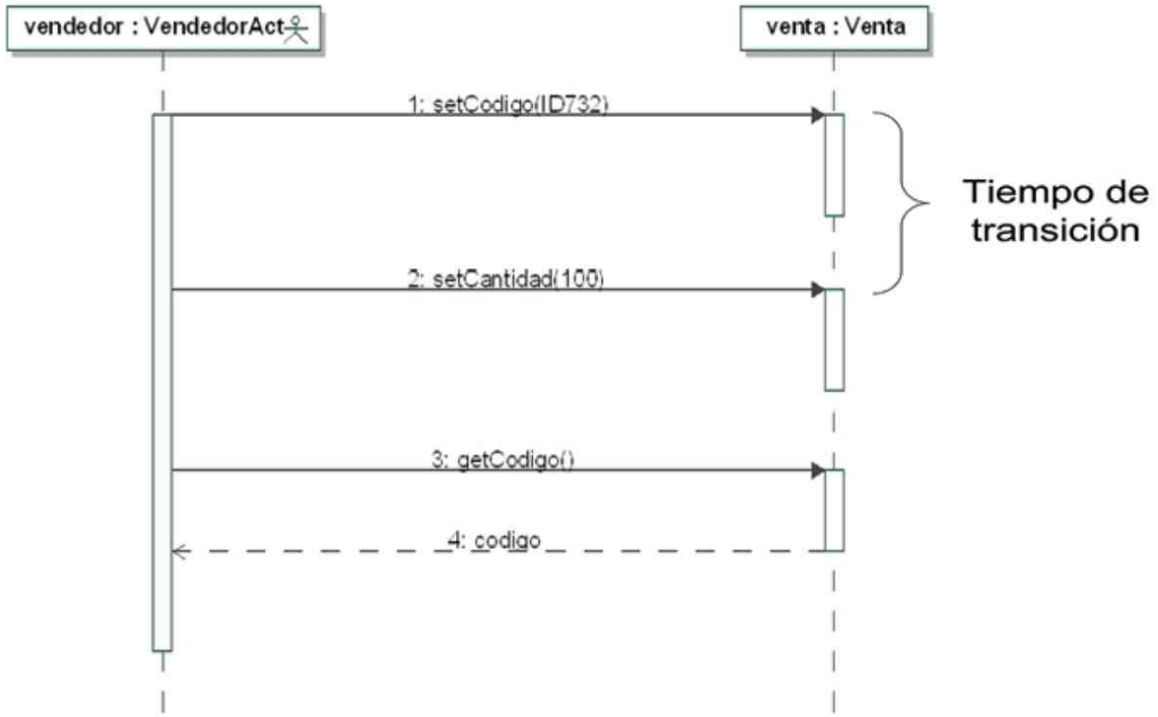


Figura 7.3. Diagrama de secuencias de una venta

En la figura 7.3 se muestra el envío de tres mensajes por el actor *Vendedor*: *setCodigo()*, *setCantidad()* y *getCodigo()*. Dichos métodos pertenecen al objeto *Venta* que los procesará en el orden indicado en la flecha de mensaje. El último mensaje retorna el código en cuestión y del mismo tipo que el establecido en el diagrama de clases. Los mensajes de retorno se describen mediante una línea discontinua devolviendo el valor requerido.

Mensajes síncronos y asíncronos

Como se comentó en la sección 7.1, los tipos de estímulos pueden ser de dos tipos:

- *Síncronos*: en los que la entidad que envía permanece a la espera hasta la ejecución del mensaje y su retorno. Es preferible usar esta opción para indicar la secuencia estricta de las llamadas a operaciones. Para simbolizar el mensaje síncrono se utiliza una flecha con punta cerrada como se ilustra en la figura 7.4.



Figura 7.4. Mensajes síncronos

En el ejemplo de la figura 7.4 el objeto *Gestor* envía un mensaje al objeto *BaseDatos* que realizará el procesamiento de consulta sobre el fichero indexado. El método *getVivienda()* quedará bloqueado hasta que el objeto *BaseDatos* procese y retorne el resultado ("vivienda").

- *Asíncronos*: la entidad remitente envía el mensaje y continúa su

ejecución sin esperar el retorno del receptor, es decir, sin bloquearse. Los mensajes asíncronos son utilizados por aplicaciones *multithread*, lo que implica la posibilidad de concurrencia. Los mensajes asíncronos se representan mediante una flecha de punta abierta (véase el ejemplo de la figura 7.5).

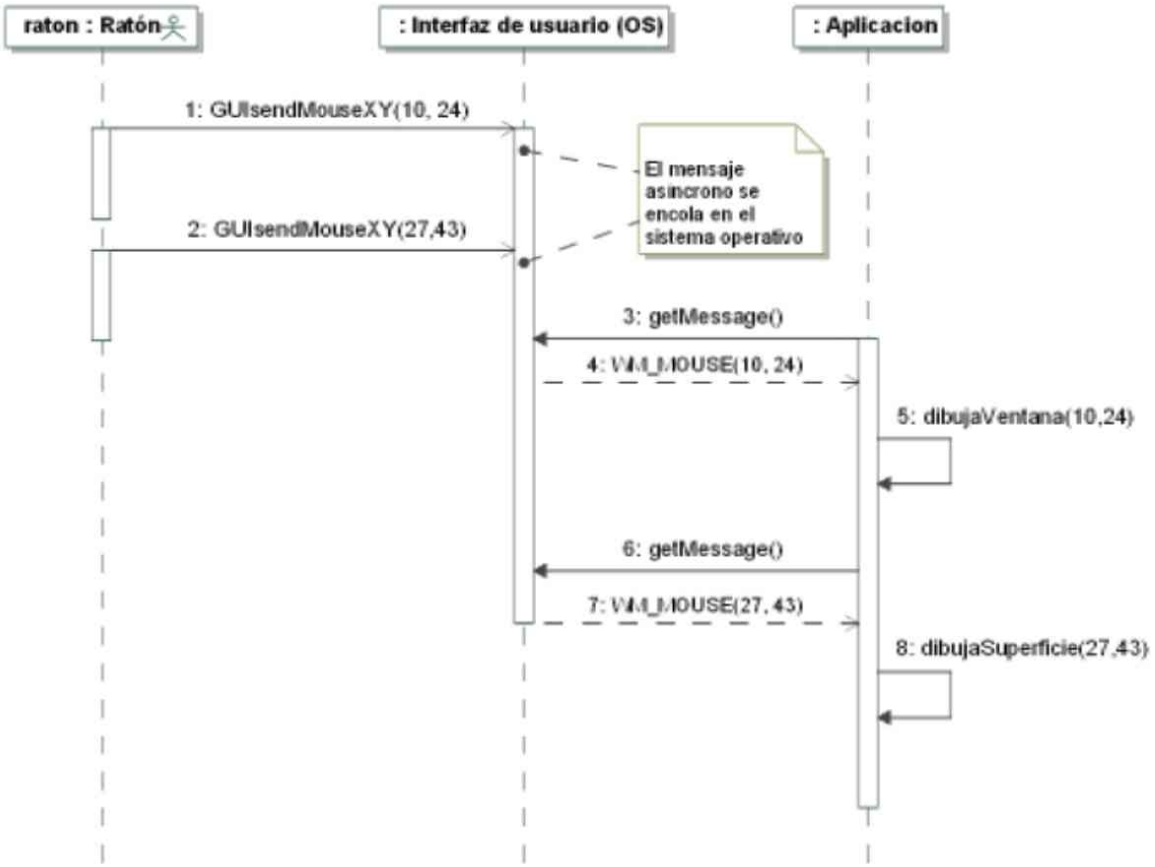


Figura 7.5. Ejemplo de mensajes asíncronos

En el anterior ejemplo, el objeto *Interfaz de usuario* del sistema operativo consulta el puerto del ratón en *multithread* y procede a encolar los datos en la cola del sistema operativo con el ID de la aplicación a la que van destinados. Consecutivamente, el *thread* principal del objeto *Aplicación* procede a consultar la cola de mensajes del sistema operativo con *getMessage*() que es bloqueante. Cuando la aplicación recibe un mensaje, ejecuta una serie de operaciones síncronas sobre su espacio de memoria de vídeo para dibujar una ventana y

una superficie en pantalla.

Respecto a la sintaxis de las operaciones que se explicitan en los mensajes entre objetos en los diagramas de secuencia es generalmente de la siguiente forma:

$$\text{mensaje}(\text{argumento}_1, \text{argumento}_2, \ldots, \text{argumento}_n)$$

Creación, destrucción e invocación recursiva

Los mensajes de creación son asociados a los operadores *new* de creación de objetos en C++ y Java. Se suelen representar mediante un mensaje de creación que termina en la cabecera de una línea de vida (la de la nueva instancia creada), mientras que la destrucción se representa mediante un aspa que figura al final de la línea de vida del objeto y corresponde con el operador *delete* de C++ y *finalize* de Java (véase figura 7.6).

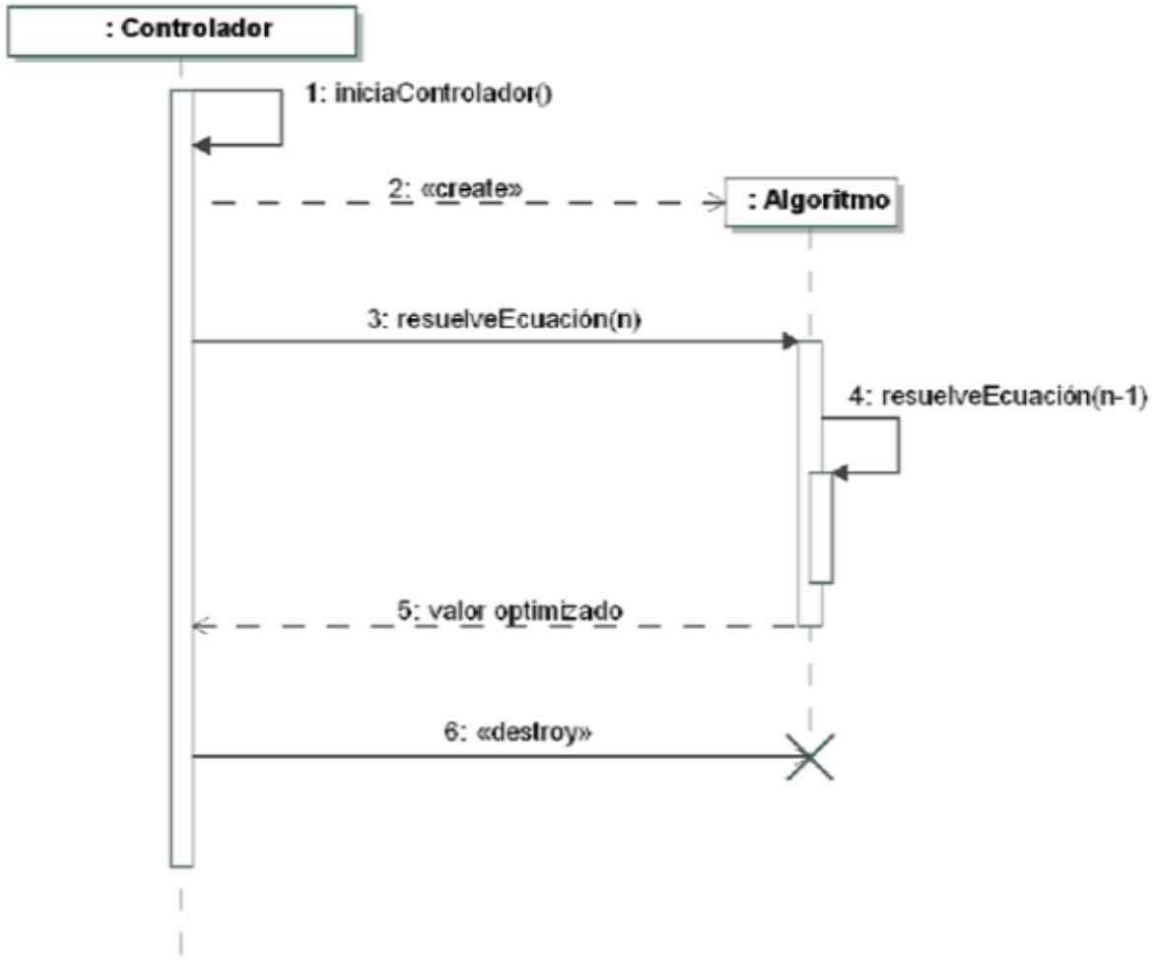


Figura 7.6. Creación de objeto, invocación recursiva y destrucción de objeto

En la interacción 1 se muestra un mensaje reflexivo de un objeto sobre su propio método (en este caso una llamada de inicialización) que puede corresponderse a una operación sobre un método *public* o *private*. En cuanto a la invocación recursiva, se simboliza superponiendo un rectángulo al que representa al método desde el que se activó (mensaje 4).

ejemplo de modelado: “servidor ftp”

Un solo cliente

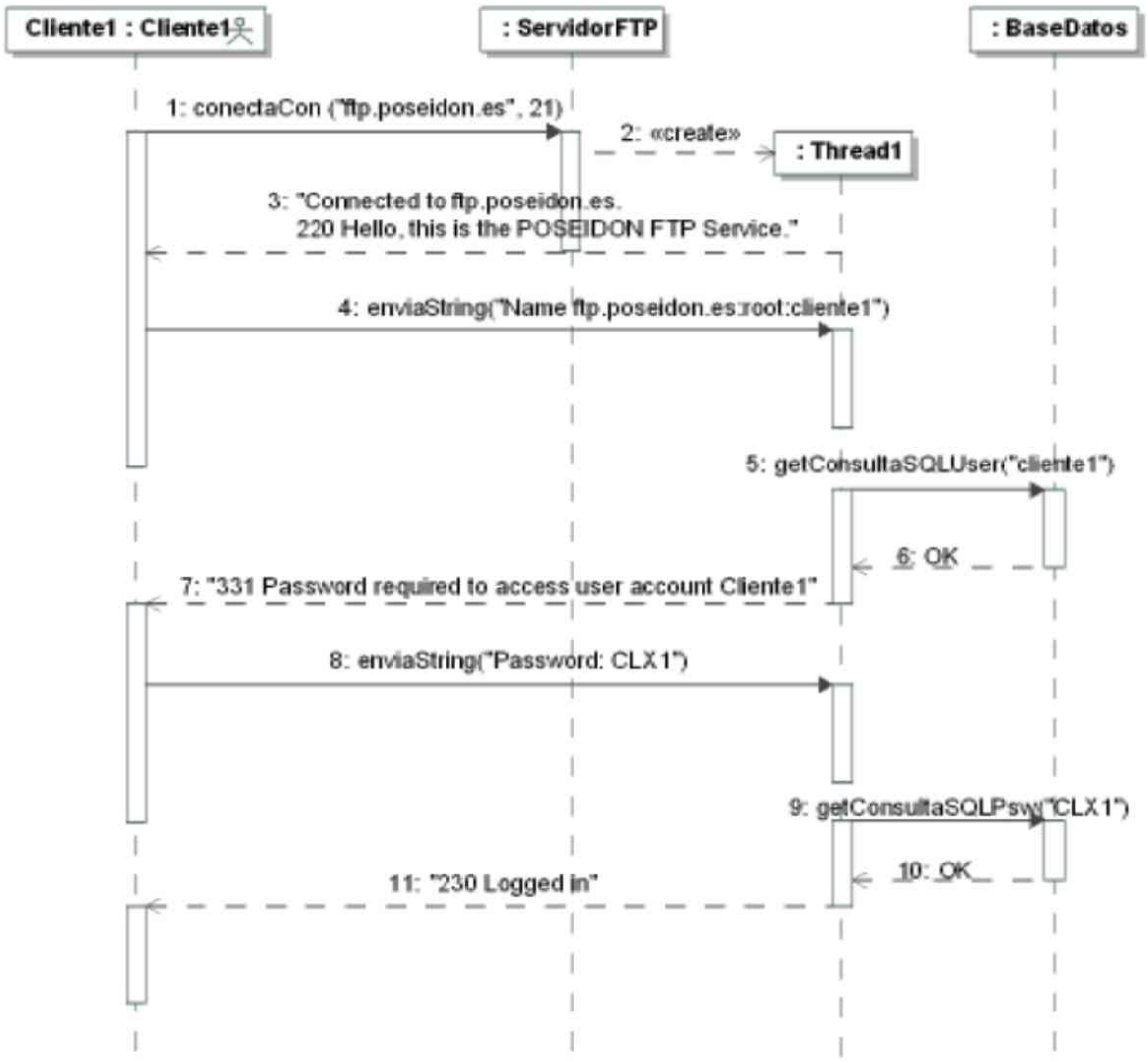


Figura 7.7. Conexión con servidor FTP usando un cliente

Dos clientes

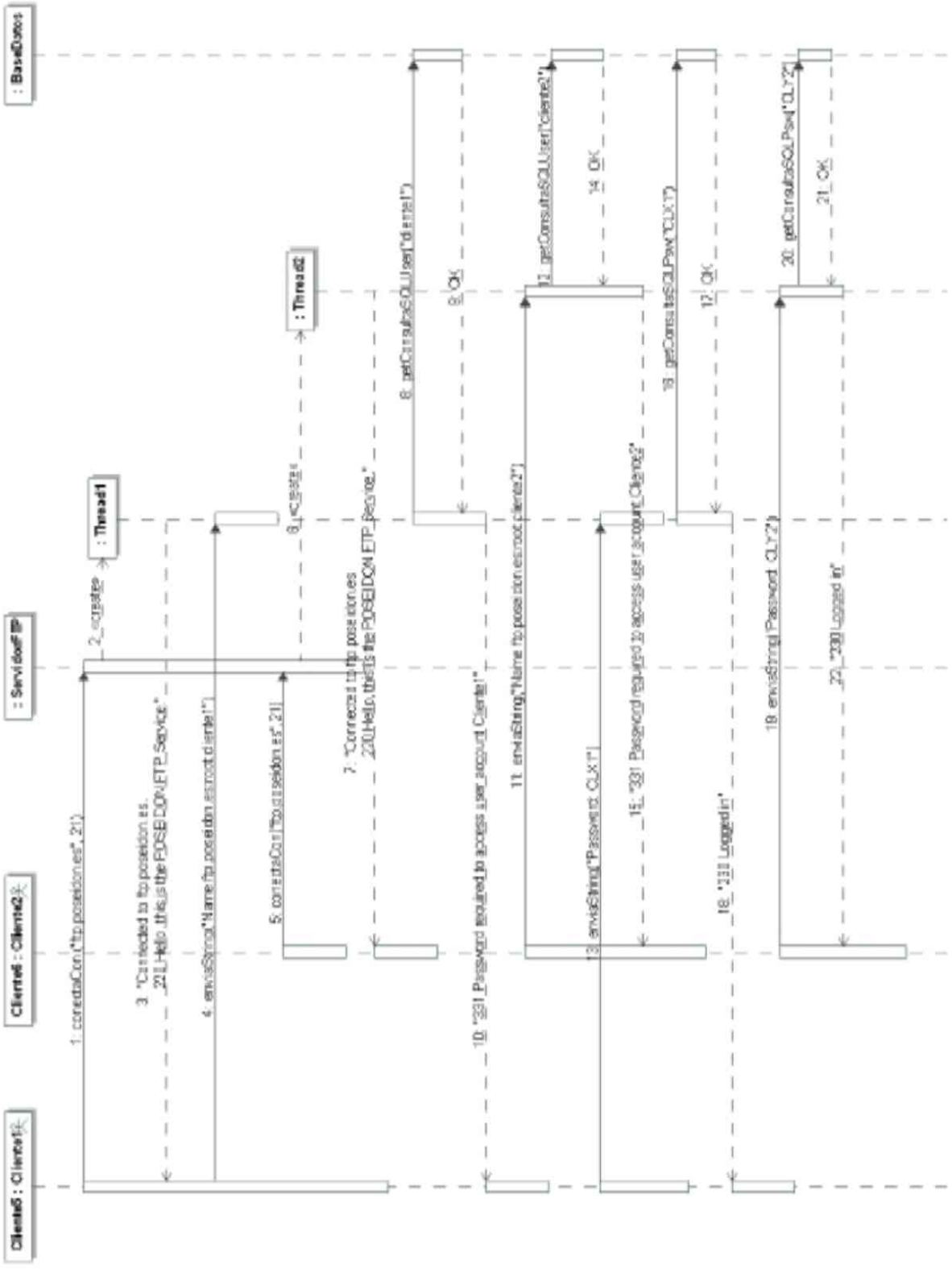


Figura 7.8. Conexión con servidor FTP usando dos clientes

En la figura 7.7 se representa un ejemplo de inicio de sesión de un solo

cliente en un servidor FTP. En el correspondiente diagrama de secuencias, el cliente comienza la sesión conectando con el servidor *"ftp.poseidon.es"*. Cuando el servidor reciba la conexión creará un nuevo *thread* (*thread1*) para responder al cliente y se liberará. Posteriormente, el *thread1* será el encargado de consultar a la base de datos el nombre de usuario y la contraseña asociados a la cuenta FTP. Puesto que el *cliente1* debe bloquearse a la espera de los mensajes de respuesta procedentes del *thread1*, dichos mensajes serán definidos como *síncronos*.

En cuanto a la figura 7.8, se nos plantea el mismo problema pero duplicando el número de usuarios. En este caso los mensajes de *cliente1* y *cliente2* serán correspondidos por el *thread1* y *thread2* que intercalarán las consultas a la base de datos, por lo que en la implementación debería considerarse la posibilidad de concurrencia.

fragmentos combinados

La sintaxis UML de los diagramas de secuencia permite definir un conjunto de fragmentos para representar situaciones y contextos típicos de interacción a menudo presentes en los lenguajes de programación estructurados. Un fragmento combinado se define mediante un operador de interacción y sus correspondientes operandos.

Saltos condicionales

Los saltos condicionales se pueden expresar mediante los operadores *opt* y *alt*. El operador *opt* define un sentencia condicional simple de la forma:

opt: *si* (condición X) *luego* realiza acción X *finsi*

Así, por ejemplo, en la figura 7.9 se representa un objeto *timer* que envía un mensaje al objeto *dispositivo* si y sólo si el contador "tick" ha superado el límite de 1000 milisegundos. La guarda de la condición debe representarse entre corchetes y el mensaje afectado por la condición dentro una región etiquetada con el operador *opt*.

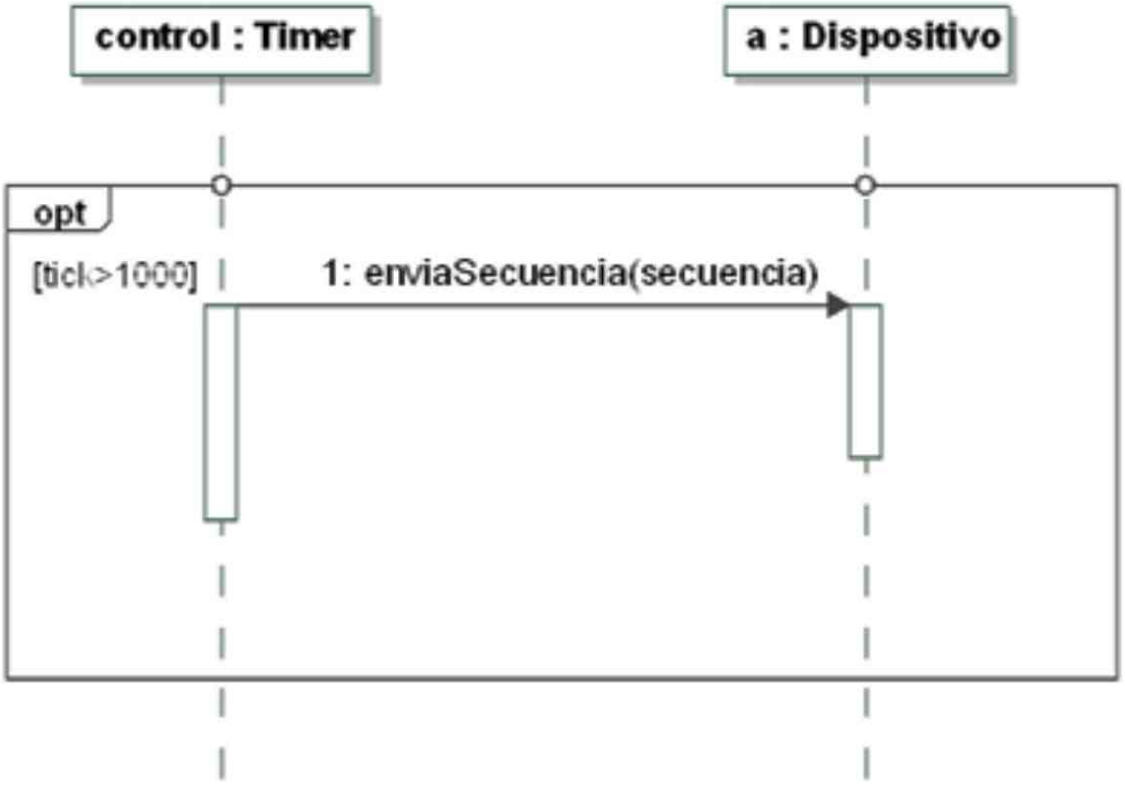


Figura 7.9. Operador opt

Así mismo el operador *alt* representa una sentencia condicional compuesta con la misma semántica que la utilizada en un lenguaje de programación estructurado:

alt:

si (condición A) luego

realiza acción A

else si (condición B) luego

realiza acción B

.

.

.

else

acción OTRA

finsi

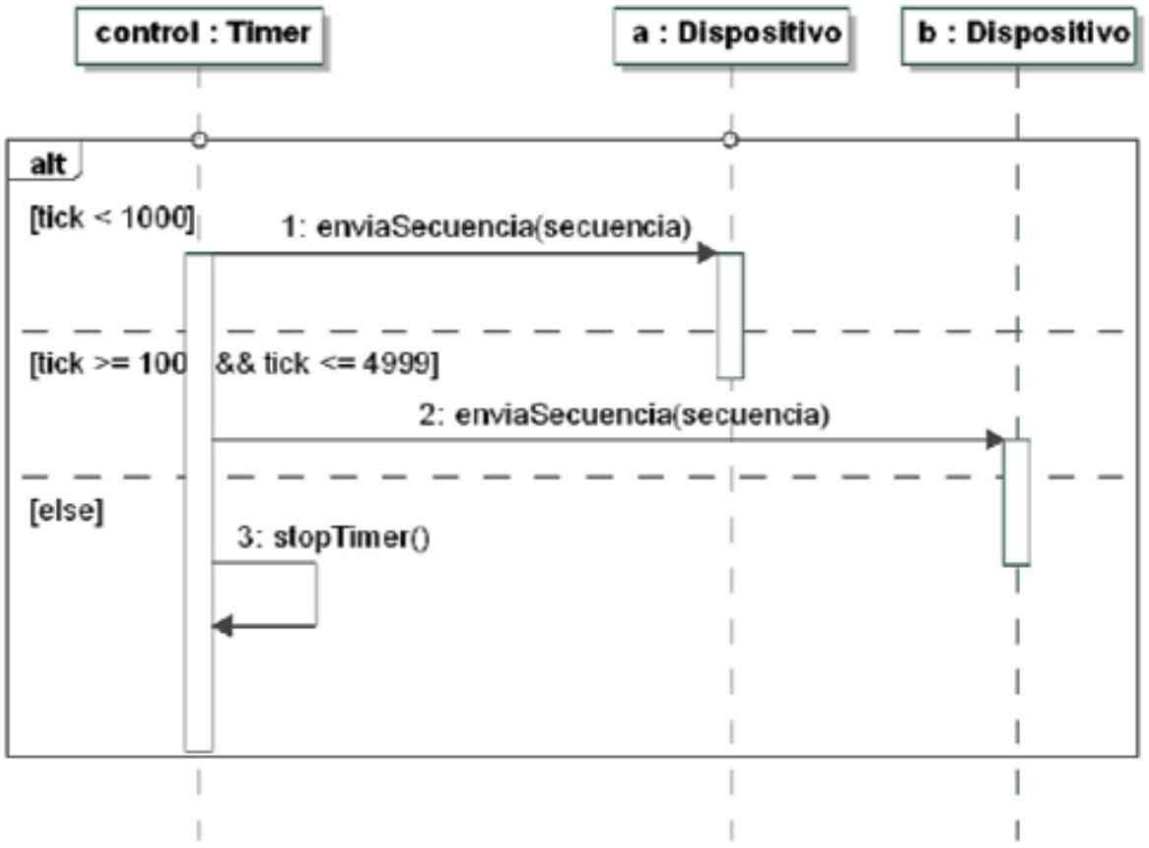


Figura 7.10. Operador alt

En la figura 7.10 se representa un escenario donde se utiliza el operador *alt*

con tres condiciones. Al comenzar se inicia a cero el contador de *ticks* del *timer*. Cuando el contador de pulsos de reloj es menor de 1000 milisegundos se invocará al dispositivo "A". De igual forma cuando el contador esté comprendido entre los 1000 y 4999 milisegundos se invocará al dispositivo "B". Finalmente cuando se halle fuera del rango [0, 4999] se ejecutará la condición *else* invocando la llamada a un automensaje (*this*) para finalizar el *timer*. Como en el caso del operador *opt*, las sentencias afectadas por la estructura condicional del fragmento combinado deben definirse dentro del marco o región etiquetada con el operador *alt*. Las diferentes ramas de la sentencia condicional deben estar separadas por líneas horizontales discontinuas, y obviamente, la guarda de la condición entre corchetes.

Iteraciones

Una forma de representar las sentencias iterativas en UML es mediante el operador *loop*. Este operador nos permitirá definir la mayoría de los tipos de bucle asociados a los lenguajes de programación estructurados. Según [Arlow07] y la especificación UML 2.x [OMG1] las sentencias iterativas se pueden expresar mediante modificaciones de los parámetros del operador:

Tipo de loop	Semántica	Expresión

while (*true*)
{sentencias}
endwhile
	Bucle infinito.	loop ó loop *

for i = i to j {sentencias} endfor	Repite (j – i) veces.	loop i, j

while (exp. booleana)
{sentencias}
endwhile
	Repite mientras la expresión booleana es cierta.	loop [exp. booleana]

repeat {sentencias} while {exp. booleana}	Ejecuta al menos una vez el *repeat* mientras la expresión booleana es cierta.	loop 1, * [exp. booleana]
forEach object in collection { sentencias} endfor	Ejecuta las sentencias del bucle una vez para cada objeto en la colección.	loop [for each object in collection]

Tabla 7.1. Modificaciones del operador loop

A modo de ejemplo proponemos el caso de un objeto de *rendering* que debe realizar una función de rellenado de un área de tamaño 640 x 480 píxeles. Por lo tanto, será necesario definir un doble bucle anidado como se muestra

en la figura 7.11:

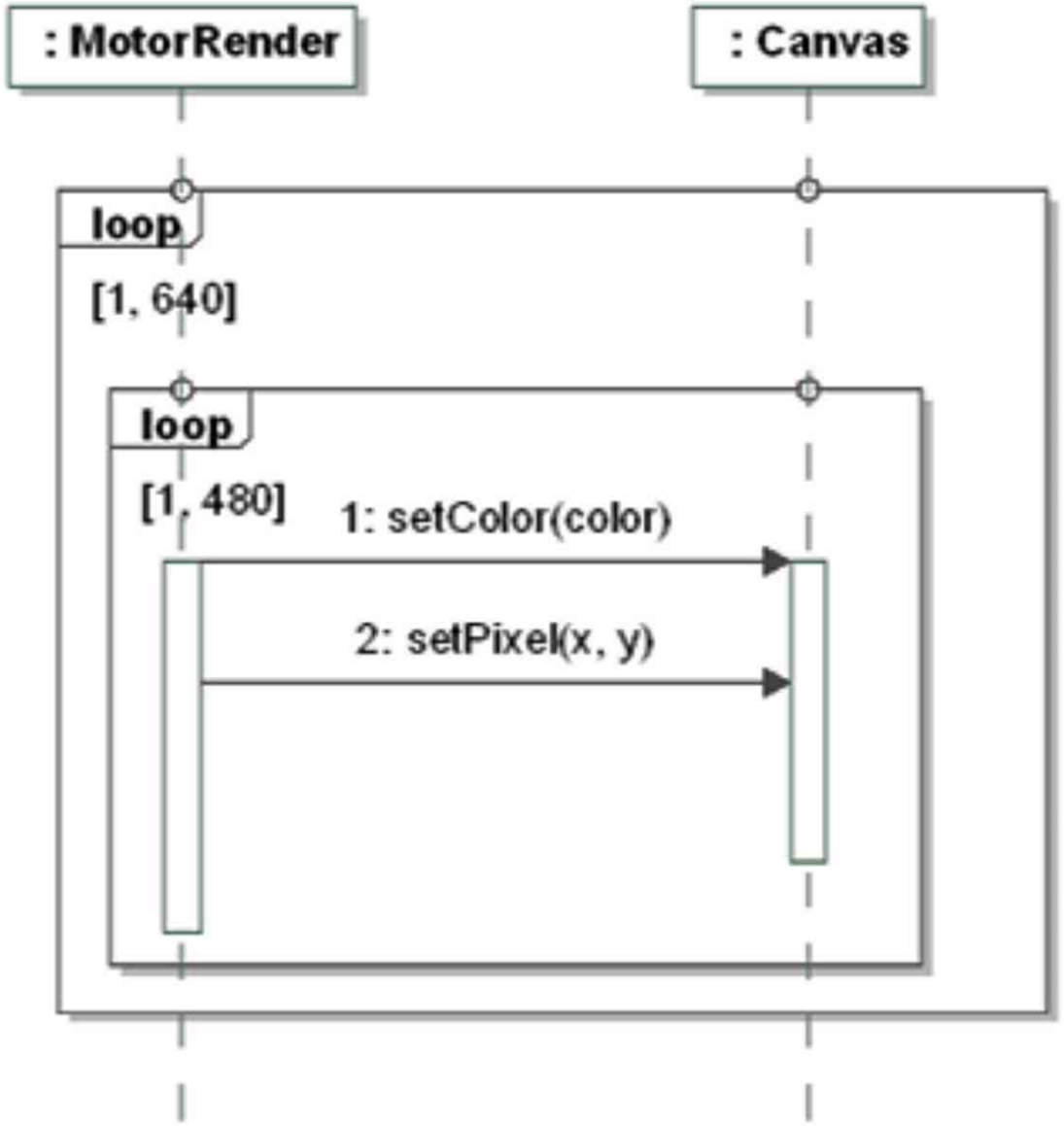


Figura 7.11. Ejemplo de bucle anidado

El operador *break* se encuentra estrechamente relacionado con el operador *loop*, puesto que en la programación estructurada se utiliza para finalizar la iteración en curso. Tan pronto como la guarda de la condición sea evaluada a cierto se ejecutará el operador *break* y terminará el bucle *loop*.

Para ilustrar esta situación supongamos un ejemplo en donde es necesario iterar sobre los libros de una biblioteca hasta encontrar un ejemplar solicitado:

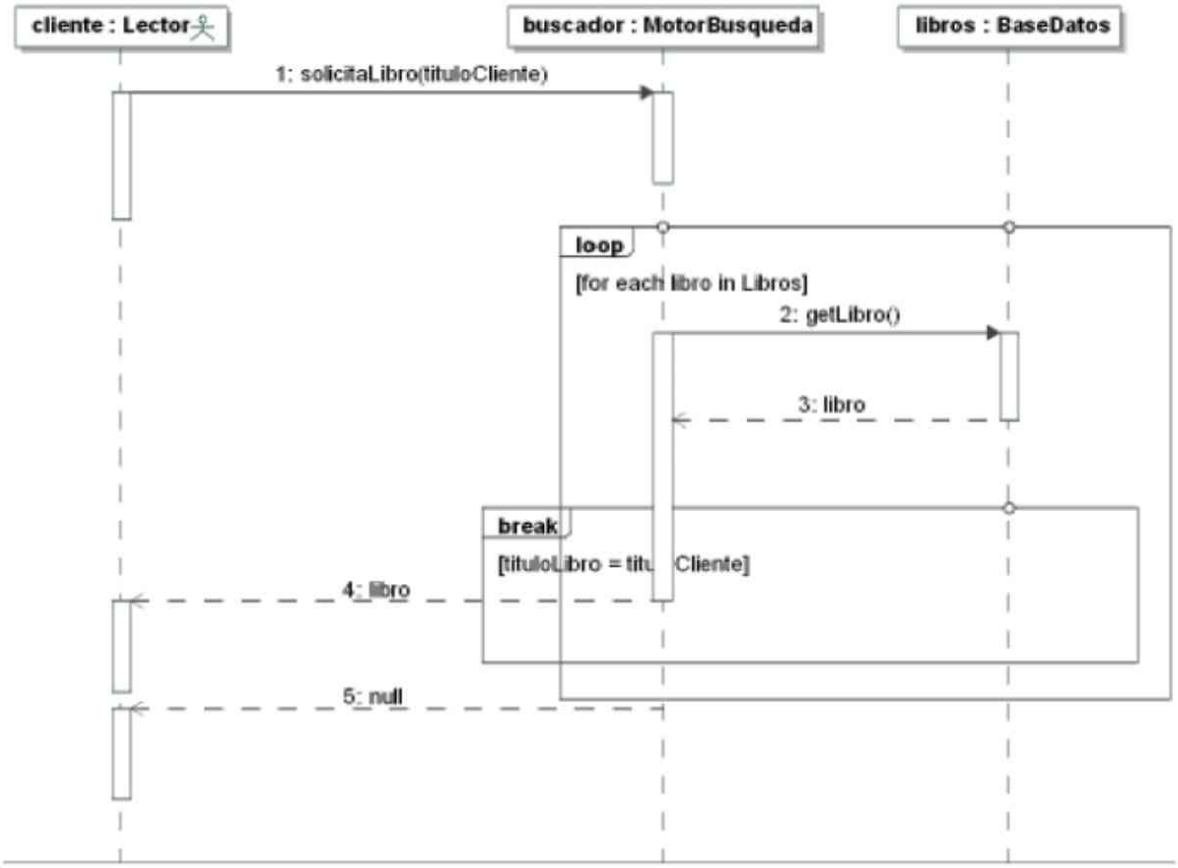


Figura 7.12. Terminación de bucle con "break"

El motor de búsqueda realiza la petición solicitada por el actor cliente. Al ejecutar el bucle se listan todos los ejemplares de la colección que cumplen con los criterios especificados. Si se verifica la condición de coincidencia de patrón se ejecutará el operador *break,* finalizará la iteración y se devolverá el correspondiente valor. Por último, es importante definir el fragmento *break* fuera del ámbito del bucle y solapando todas las líneas de vida de los objetos a los que afecta la iteración finalizada.

Paralelismo

La versión 2.x de UML permite especificar situaciones en donde los procesos se ejecutan de forma paralela, ya sea en un cluster o sistema multiprocesador. El operador que lleva a cabo la definición de un escenario paralelo se le denomina *par*.

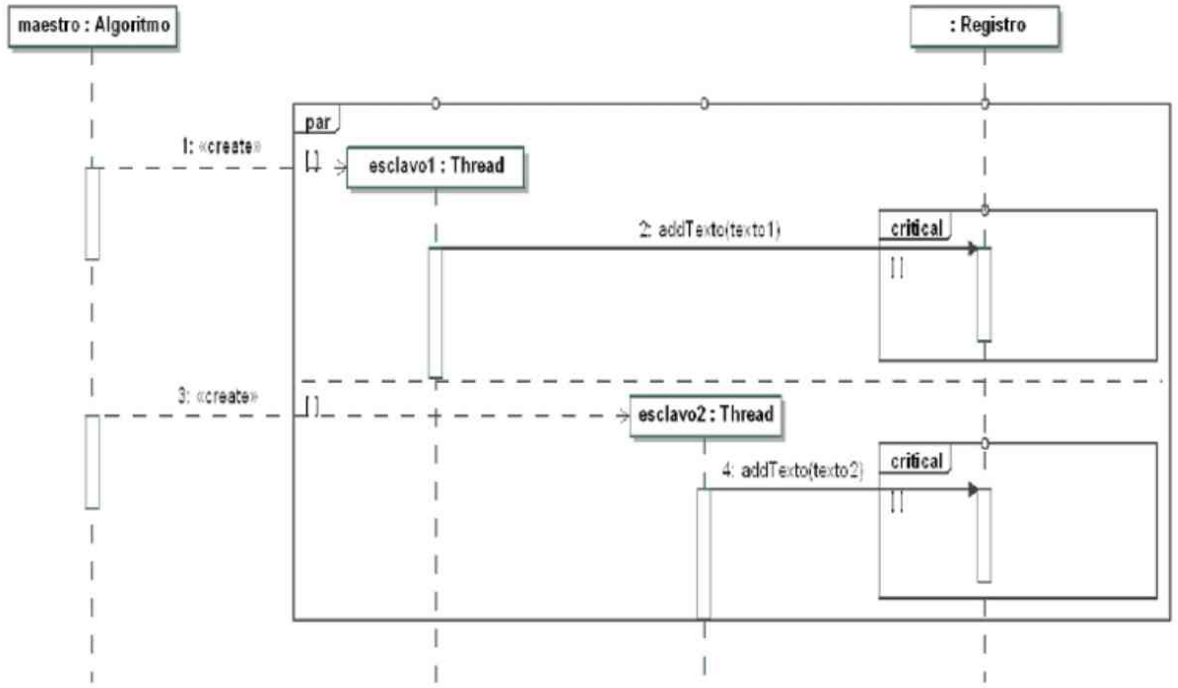


Figura 7.13. Ejemplo de paralelismo con sección crítica

En el ejemplo de la figura 7.13 el proceso *maestro* crea dos procesos esclavos en diferentes unidades de procesamiento que acceden simultáneamente a un registro de una tabla de una base de datos. Puesto que existe una situación de paralelismo entre estos dos *threads*, la sección de mensajes deber ser solapada con un fragmento del tipo *par*. Así mismo, puesto que los dos *threads* acceden a la vez al mismo recurso compartido, es necesario especificar la protección del recurso mediante una sección crítica por medio del operador de UML *critical*.

Parametrización

En UML 2.x las interacciones pueden ser parametrizadas y de esta forma simplificar el diagrama de secuencias cuando éste es excesivamente complejo.

Imagínese una situación donde es necesario realizar frecuentemente una llamada a un objeto para devolver un valor. Con la parametrización es posible encapsular un diagrama de secuencias para ser invocado desde otro diagrama a modo de subrutina. El operador que hace esto posible se denomina *ref*. Un ejemplo de parametrización puede verse en la figura 7.14:

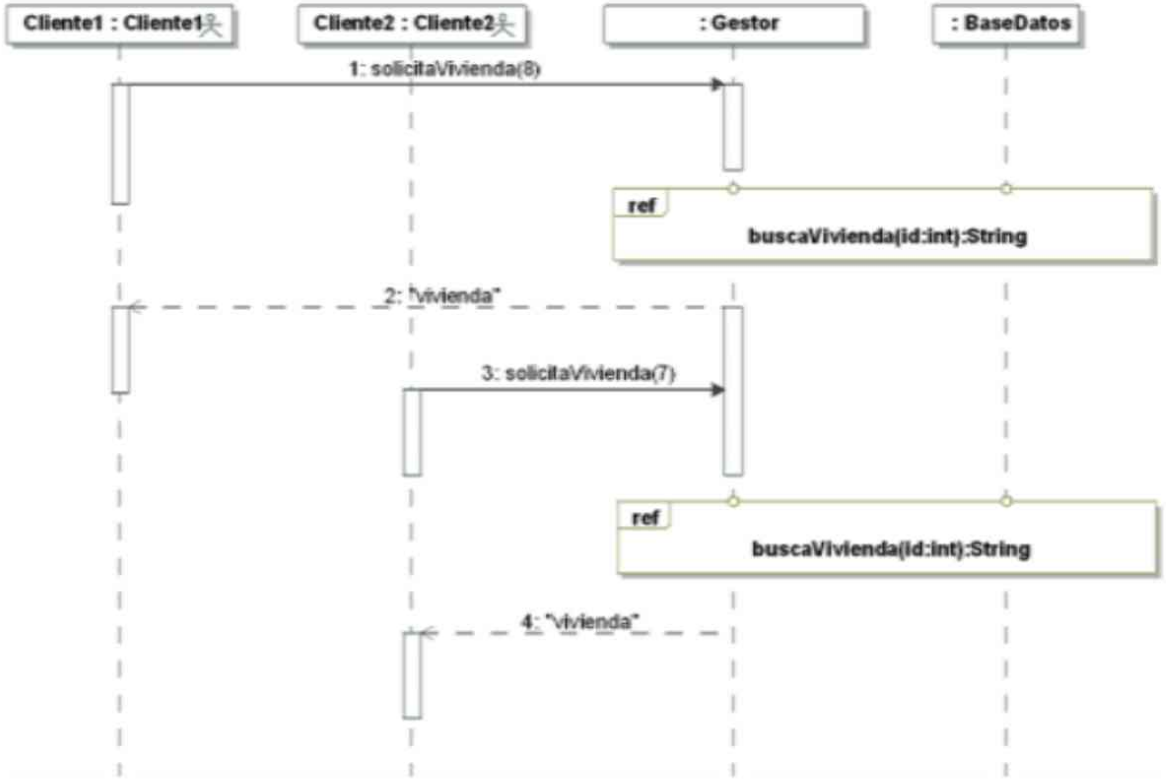


Figura 7.14. Llamada a otro diagrama de secuencias

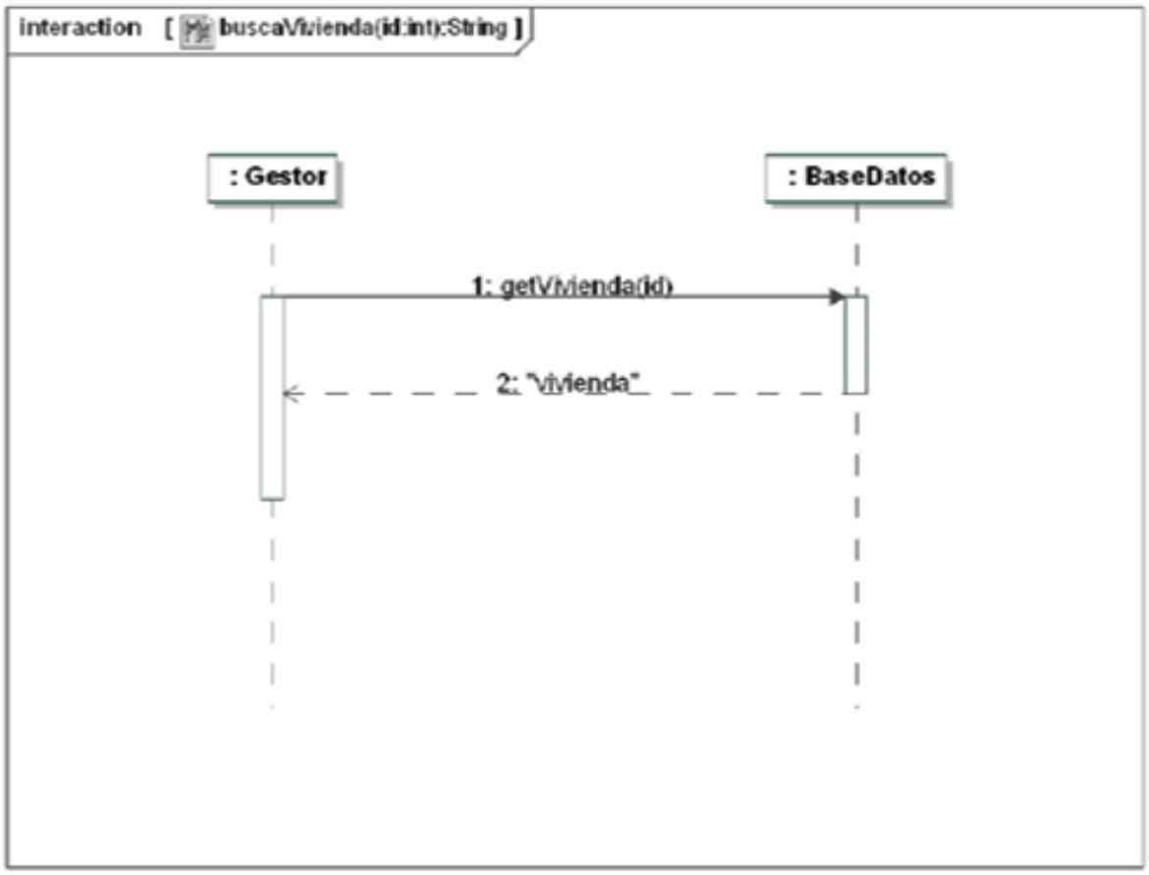


Figura 7.15. Diagrama de secuencias invocado

En el ejemplo anterior el diagrama de secuencias *buscaVivienda* encapsula la funcionalidad requerida entre el objeto *Gestor* y la Base de Datos. Cuando el diagrama de la figura 7.14 invoca la búsqueda de una vivienda ésta será redirigida hacia la función representada en la figura 7.15 mediante *ref* a modo de subrutina.

caso de estudio: ajedrez

Se ha dividido el escenario de una jugada del ajedrez en dos diagramas, uno para representar la interacción y ataque de un jugador humano sobre el tablero virtual y otro para el cálculo de la estrategia y contraataque de la máquina.

Turno del jugador humano

En el diagrama de la figura 7.16 se ilustra la secuencia de operaciones entre instancias de clases para realizar el movimiento de un usuario. A continuación describimos en detalle la secuencia de la jugada:

- 1 y 2: El usuario desplaza el ratón desde la posición de origen hasta la posición de destino. Se convierten las coordenadas de pantalla a filas y columnas del tablero.
- 3: Se informa al motor de juego que el usuario ha movido una pieza en unas determinadas coordenadas en filas y columnas.
- 4 y 5: Se averigua qué pieza está en las coordenadas de origen (*fil1*,*col1*) y se devuelve el objeto *Pieza*.
- 6: Se comprueba que la jugada a las posiciones de destino: (fil2, *col2*) están permitidas en el reglamento del ajedrez.
- 7, 8 y 9: Se informa al objeto *Pieza* para que se reposicione en las nuevas coordenadas y se procede a dibujar la pieza en el *backbuffer*.
- 10: Se redibuja el tablero completo sobre el buffer frontal.
- 11 y 12: Se comprueba si realiza algún mate y si la pieza está amenazada en las posiciones de destino.
- 13: Se ha detectado una amenaza. Se informa al usuario.
- 14: El usuario pide consejo a la IA.
- 15 a 20: Se procede a redirigir la petición de consejo al algoritmo paralelo y devuelve un objeto *Pieza* con la pieza que debe mover.

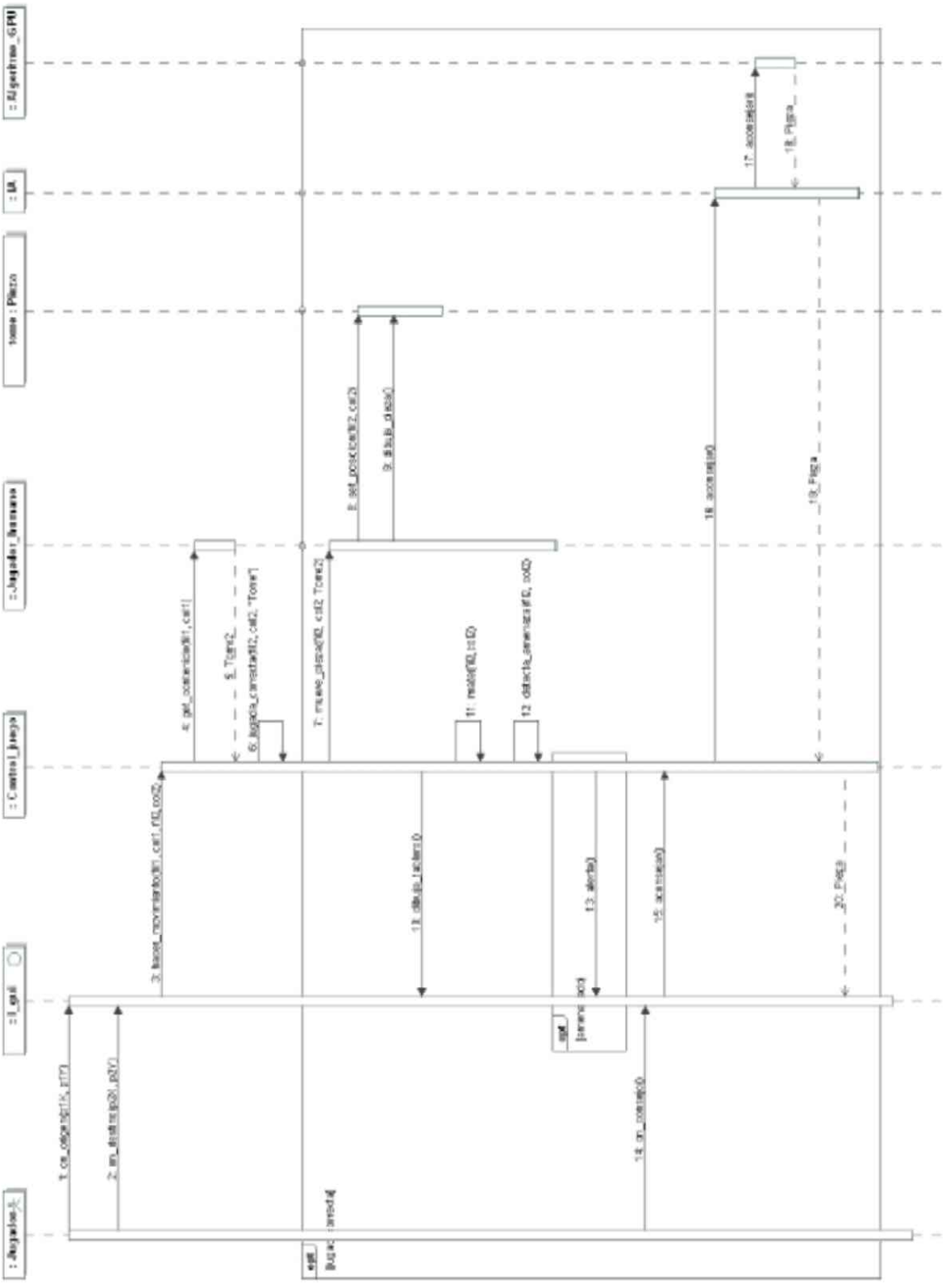


Figura 7.16. Diagrama de secuencias (mueve humano). Sigue.

Turno de la IA

En el diagrama de la figura 7.17 se recogen las interacciones entre objetos del dominio del juego de ajedrez que describen la decisión algorítmica de contraataque de la máquina y el jaque mate final al rey del jugador humano. La secuencia que lleva a cabo la estrategia IA es la siguiente:

- 1: El motor de juego informa a la IA de que el jugador humano ha movido una "Torre" a las posiciones de destino *fil2* y *col2*.
- 2: La IA pide al objeto*Algoritmo_GPU* que procese una estrategia con la versión paralela del algoritmo minimax.
- 3 y 4: El objeto*Algoritmo_GPU* obtiene un mapa asociativo del jugador humano con identificadores de piezas y posiciones.
- 5 y 6: El algoritmo de GPU consulta el objeto de heurística final con el propósito de obtener un valor óptimo de la función evaluadora.
- 7, 8, 9 y 10: El algoritmo GPU procesa el resultado a partir de la información proporcionada por las anteriores interacciones y devuelve la jugada óptima a la IA.
- 11: El motor de juego comprueba que el movimiento de destino devuelto por la IA cumple las normas del ajedrez.
- 12, 13 y 14: Se informa a la IA para que realice el movimiento a la casilla de destino y dibuje la pieza en el *backbuffer*.
- 15 y 16: Se consulta qué pieza se encuentra en la posición de ataque.
- 17: Comprueba si la posición de destino se encuentra amenazada.
- 18: Comprueba si existe mate en la posición de destino.
- 19 y 20: Puesto que se ha dado jaque al rey, el motor de juego elimina la pieza del tablero.
- 21: El motor de juego determina que ha finalizado la partida.



Figura 7.17. Diagrama de secuencias (contraataque de la IA). Fin partida

caso de estudio: mercurial

Pasamos ahora a explicar el escenario de borrado de un fichero local por un usuario de la aplicación CVS de *Mercurial*. En este caso la conexión y las transferencias en la red son prácticamente inexistentes. El motivo de no realizar la petición al servidor remoto justifica un diagrama de secuencias más completo en el que se describa las iteraciones sobre la estructura del patrón *Composite* (véase capítulo nueve).

A continuación se detallan la secuencia de interacción de este caso:

- 1: El actor *Programador* introduce una cadena de texto en el terminal del sistema.
- 2: La fachada del intérprete analiza la cadena alfanumérica.
- 3: Una vez creado el objeto*Comando* se pasa al objeto *ClienteProgramador*.
- 5, 6 y 7: Se repite la misma acción que 1-3 pero especificando una ruta para el comando de borrado.
- 8: Se obtiene primero un objeto que implemente la interfaz*IDirectorioAbstacto* (típicamente un objeto Directorio) pasando como parámetro la secuencia jerárquica de la ruta.
- 9: Se entra en un bucle *loop* invocando iterativamente el método*getItem()* del objeto de tipo *IDirectorioAbstracto* hasta que se encuentra el fichero especificado.
- 10: Si se halla el fichero se procede a borrarlo del sistema de archivos.
- 11 y 12: Finalmente se borra el fichero y se destruye el objeto.
- 13: En caso de que la llamada al método*getItem()* devuelva NULL se lanzará un excepción informando del error.

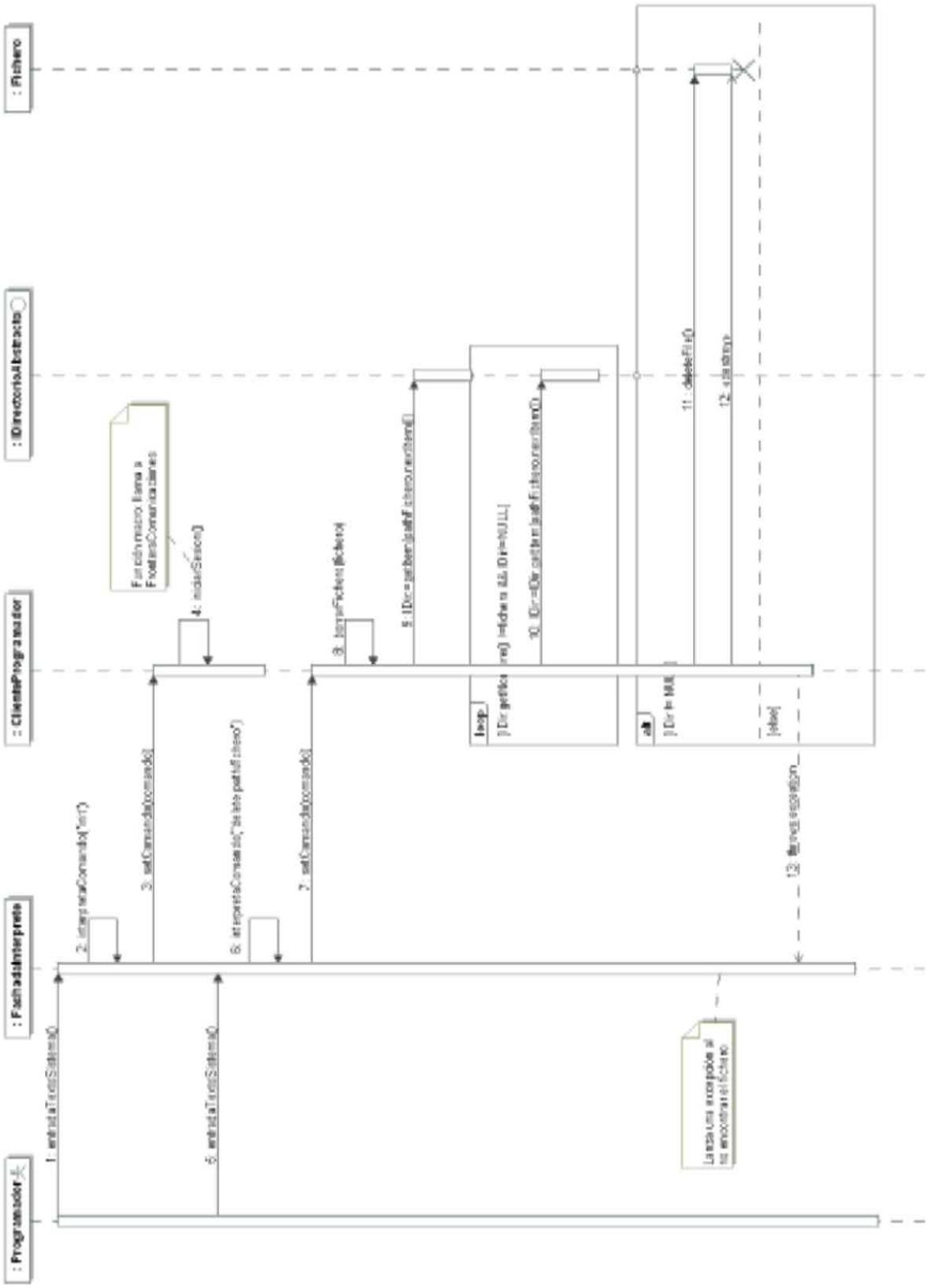


Figura 7.18. Diagrama de secuencias para el borrado de un fichero local en Mercurial

caso de estudiO: servicio de cifrado remoto

En la figura 7.19 se modela el caso de uso de cifrado en el lado cliente para el algoritmo del tipo híbrido. Para ello se realiza la siguiente secuencia de llamadas que se ilustran en el diagrama.

- 1: Crea el objeto *FachadaComunicaciones* para proceder al envío del mensaje cifrado.
- 2 y 3: Después de la petición recibida de la fachada del menú para el cifrado híbrido, el *Usuario* crea las instancias de las clases *CifradorSimetrico* y *CifradorAsimetrico*.
- 4: Crea el par del claves pública y privada necesarias para el cifrado RSA mediante llamada a el objeto *CifradorAsimetrico*.
- 5: Genera la clave de sesión de 16 bytes en el objeto *CifradorSimetrico*.
- 6: Retorna la clave de sesión al objeto cliente.
- 7: Llama al objeto del tipo *CifradorAsimetrico* para cifrar la clave de sesión mediante la clave pública del usuario avanzado.
- 8: Llama al método *get_msg()* para devolver la clave de sesión cifrada.
- 9: Retorna la clave de sesión cifrada con la clave pública RSA del servidor.
- 10, 11 y 12: Llama al objeto *CifradorSimetrico* para cifrar el mensaje mediante el algoritmo AES y la clave de sesión. Llama a *get_msg()* que retornará el mensaje ya cifrado.
- 13: Automensaje que prepara a *Usuario* para enviar los datos.
- 14 y 17: Conecta y desconecta con el servidor respectivamente.
- 15 y 16: Envía el mensaje cifrado y la clave de sesión cifrada al servidor.

Puesto que la estructura de mensajes es similar en el servidor, se propone al lector el modelado del diagrama de secuencias de descifrado como ejercicio.

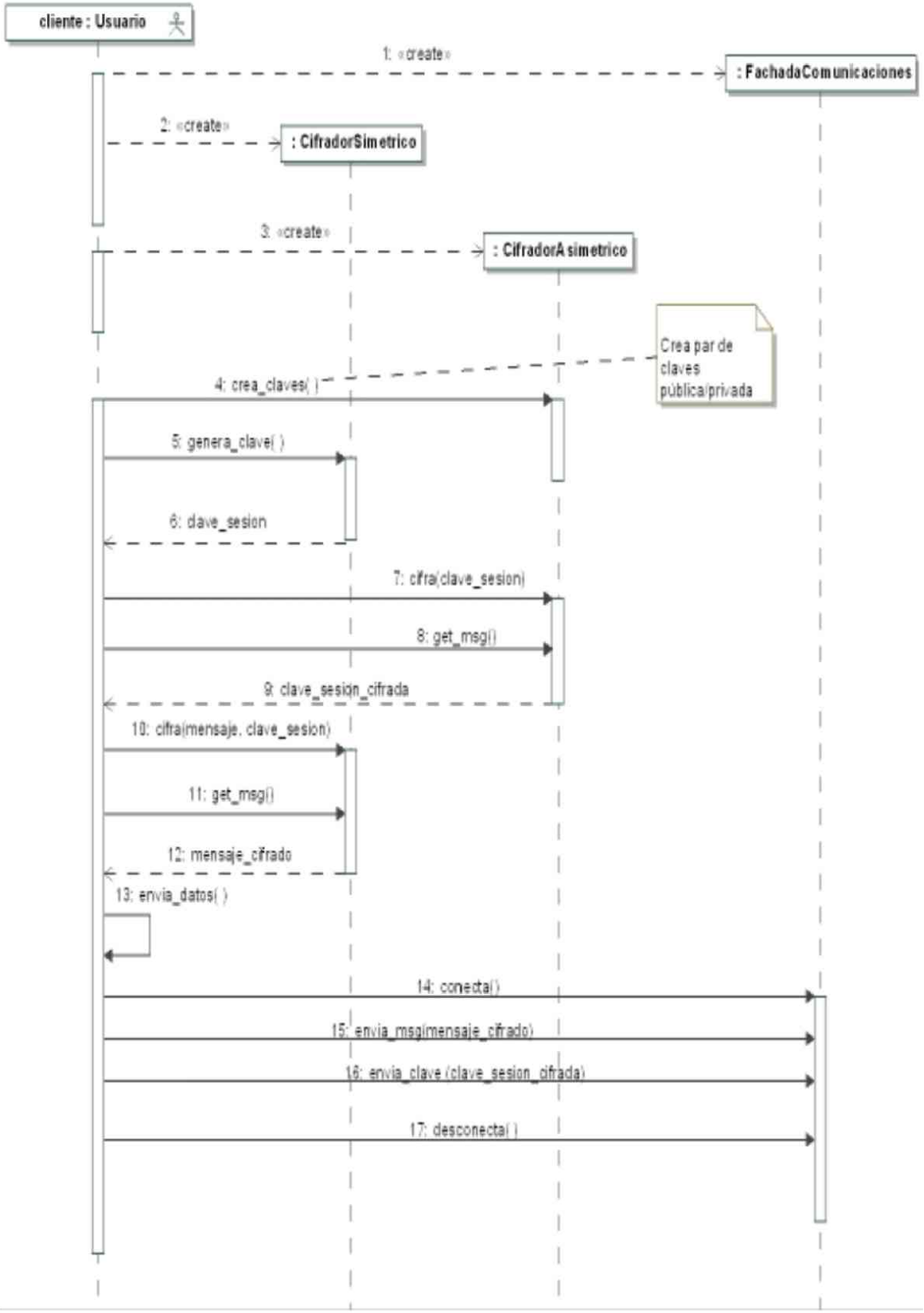


Figura 7.19. Diagrama de secuencias para cifrado híbrido en el lado cliente

22 Las instancias de clase se las denomina objetos.

diagramas de comunicación

«Conducir con orden mis pensamientos, empezando por los objetos más simples y más fáciles de conocer, para ascender poco a poco, gradualmente, hasta el conocimiento de los más complejos, y suponiendo incluso un orden entre ellos que no se parecen naturalmente unos a otros [...]».

(René Descartes: Discurso del método, Parte II).

Seguimos en la fase de diseño detallado, más concretamente con los diagramas relacionados con los modelos de interacción. El *diagrama de comunicación* permite visualizar las interacciones entre instancias añadiendo información con una semántica diferente al de diagrama de secuencias. A pesar de la similitud con estos, los diagramas de comunicación muestran de una forma alternativa la secuencia de mensajes, el orden, el control y las bifurcaciones en el flujo de llamadas.

El diagrama de comunicación es un diagrama más compacto y menos complejo que el de secuencias, si bien se puede obtener automáticamente de este último y viceversa en algunas herramientas CASE. En general se podrían ver los diagramas de comunicación como una instantánea en el tiempo de un escenario representado por un diagrama de secuencias. Por tanto, este tipo de diagrama que veremos ahora muestra menos eficazmente la evolución temporal.

No obstante, se utilizará para modelar situaciones donde es necesario capturar las interacciones y el flujo de ejecución de los mensajes entre objetos que implementan una acción[23] en un momento dado.

estructura básica

Las entidades principales del diagrama de comunicación son los objetos. Estos se representarán tal como se vio en los diagramas de secuencias (ver figura 7.1), aunque generalmente los diseñadores de UML suelen omitir el nombre de la clase y especificar únicamente el tipo.

Otro elemento fundamental del diagrama son los enlaces que indican la existencia de una relación semántica entre los objetos que une. Esto no implica que tengan obligatoriamente una asociación en el diagrama de clases para poder enviarse mensajes.

Por último están los mensajes que fluyen a través de los enlaces, los cuales poseen un determinado número de identificación que especifica el orden o la secuencia de actuación. Cuando es necesario detallar o jerarquizar una serie de mensajes se recurrirá a expresarlos mediante un punto lexicográfico. Los mensajes también poseen una flecha que indica la direccionalidad de los mismos y que viene a representar cuál es el objeto invocador y cuál el invocado. A modo de ejemplo, la figura 8.1 muestra lo que sería este tipo de diagrama. En él se aprecia un escenario de serialización de dos objetos: *formulario1* y *formulario2* desde un diálogo de captura de datos de empresas.

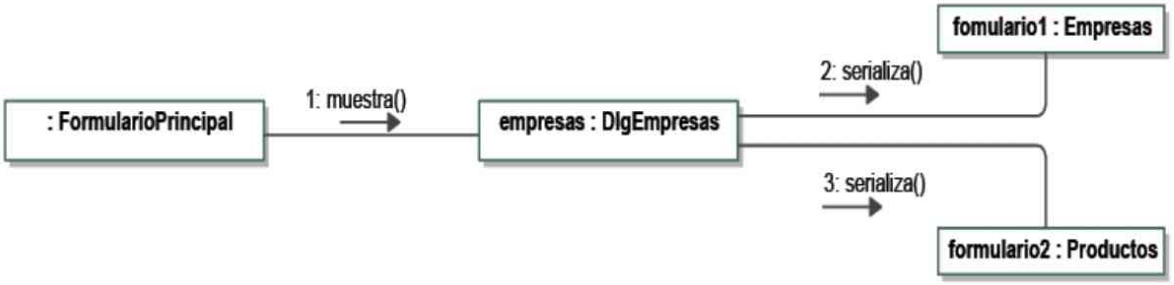


Figura 8.1. Ejemplo de diagrama de comunicación

La sintaxis del mensaje es idéntica a la explicada para los diagramas de secuencias. Respecto a la numeración, se debe seguir la siguiente regla sintáctica:

$secuencia_1.secuencia_2.(...).secuencia_i$:mensaje

Y donde cada secuencia tiene la siguiente sintaxis[24]:

[número | letra] [recurrencia]

El *número* indica el orden de la secuencia. Como se ilustra en la figura 8.1, el primer mensaje es el enviado desde el formulario principal al diálogo de empresas, el segundo el enviado desde el diálogo de empresas al formulario 1, y el último, el enviado desde el diálogo de empresas al formulario 2. Sin embargo, en el ejemplo de la figura 8.2 el orden de anidamiento indica el orden de la subsecuencia dentro de la secuencia, como en los casos: (1a.1) y (1b.1)[25].

En los casos donde se utilice una *letra* implicará una situación de envío de mensajes concurrentes (véase figura 8.2).

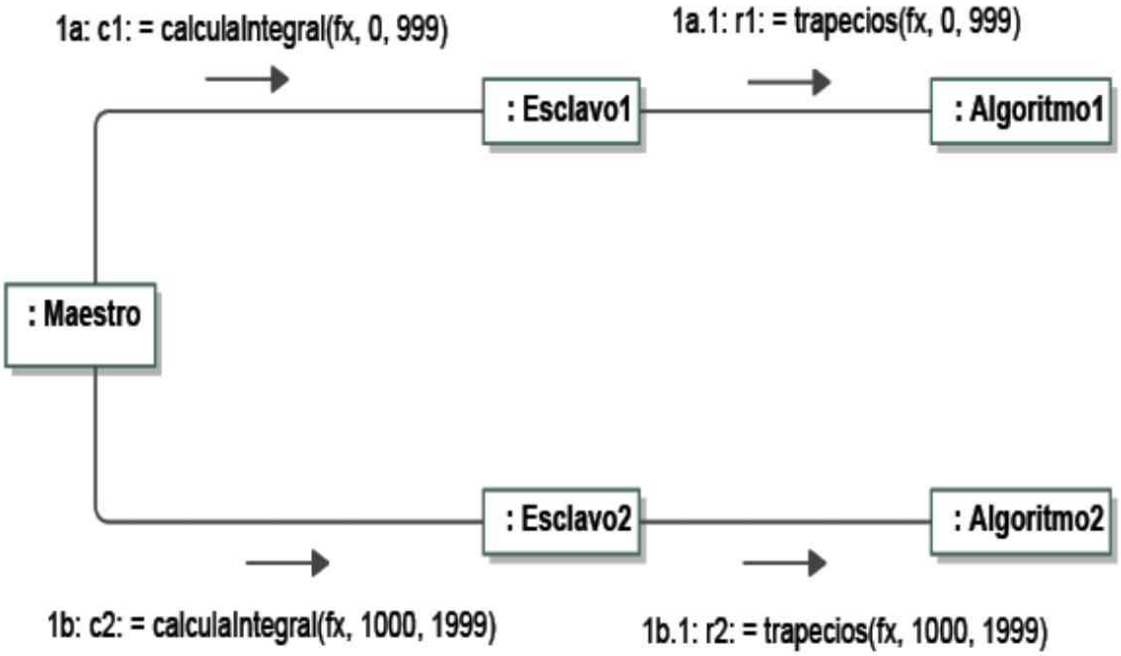


Figura 8.2. Diagrama con situación de concurrencia

En el diagrama de la figura 8.2 existe un escenario concurrente especificado con los mensajes (1a) y (1b) lanzados síncronamente desde el *Maestro* hacia

los esclavos. Estos, a su vez, enviarán mensajes síncronos (1a.1 y 1b.1) hacia sus respectivos algoritmos de cálculo que devolverán los correspondientes resultados hacia el *Maestro*.

La *recurrencia*, como veremos más adelante expresa sentencias condicionales y de control.

Otra situación comúnmente producida en los diagramas de comunicación suele ser los mensajes de un objeto a él mismo, conocidos en algunos lenguajes como mensajes "*self*" o "*this*" (figura 8.3).

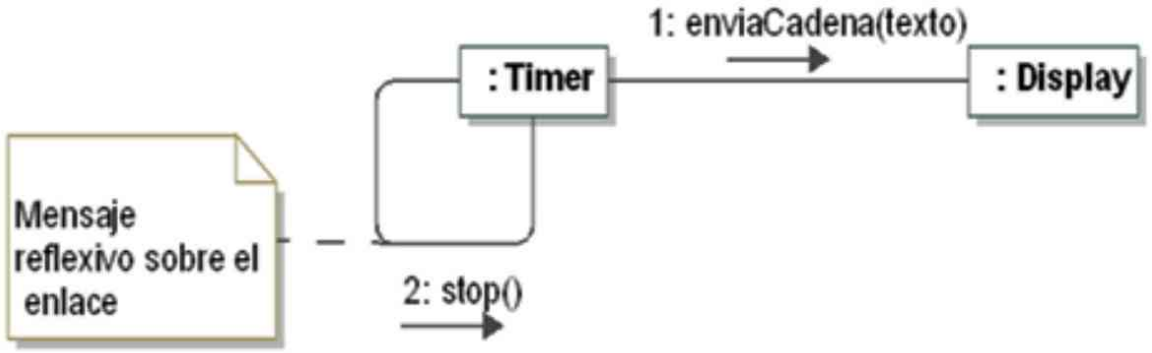
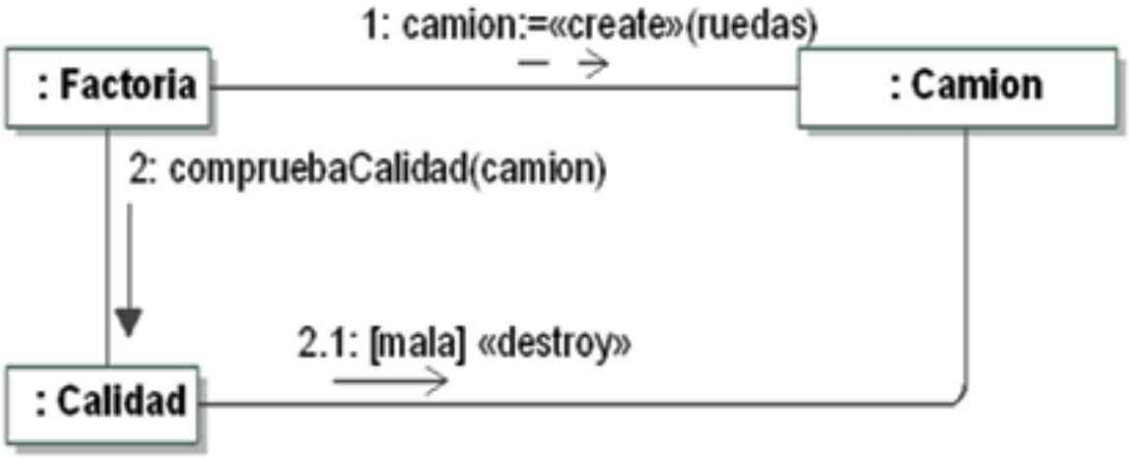


Figura 8.3. Ejemplo de mensaje reflexivo

En el anterior ejemplo el objeto *Timer* envía una cadena de texto a un controlador de un "display" cuando recibe el evento de tiempo y finalmente se detiene enviando un mensaje de *stop()* sobre el mismo temporizador.

La creación y destrucción de objetos se realiza de manera análoga a la explicada en el diagrama de secuencias y sigue la misma lógica que un lenguaje orientado a objetos:


1: camion:=«create»(ruedas)
: Factoria
: Camion
2: compruebaCalidad(camion)
2.1: [mala] «destroy»
: Calidad

saltos condicionales

Durante el proceso de interacción e intercambio de mensajes entre objetos que se representan en el diagrama de comunicación, también es posible especificar situaciones donde se produce una bifurcación en el control del envío de mensajes. De esta forma es posible indicar que un mensaje se enviará siempre y cuando se especifique la expresión condicional entre corchetes. Cuando dicha expresión o guarda es evaluada como cierta entonces el mensaje será enviado.

En la figura 8.4 se modela una situación de este estilo. El cliente envía un mensaje de pulsación de ratón en un área de la pantalla; acto seguido el *gestor de eventos* discernirá a cuál de los dos botones van dirigidas las coordenadas. Según se pulse uno u otro, se enviará un mensaje de imprimir un listado de nóminas o facturas a los respetivos objetos que lo gestionan. Finalmente, un mensaje asíncrono es enviado desde los objetos *Nóminas* y *Facturas* hacia el *gestor de la impresora*, siempre y cuando ésta esté liberada.

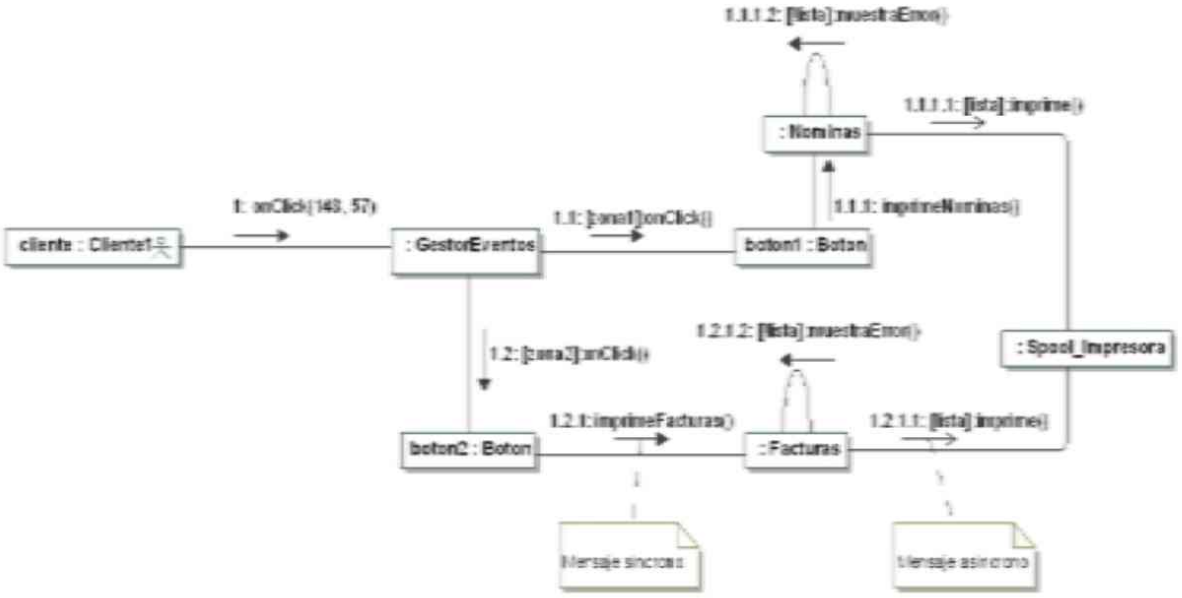


Figura 8.4. Ejemplo de mensajes condicionales

iteraciones

Las iteraciones en los diagramas de comunicación indican la repetición de un mensaje enviado entre dos objetos. La notación de las iteraciones se especifica mediante el uso del asterisco seguido de la expresión del bucle.

Generalmente la sintaxis indicada por UML para los bucles condicionales es la siguiente:

* [bucle [guarda]]:mensaje

sin que exista una nomenclatura especificada por UML para el *bucle* y la *guarda*[26].

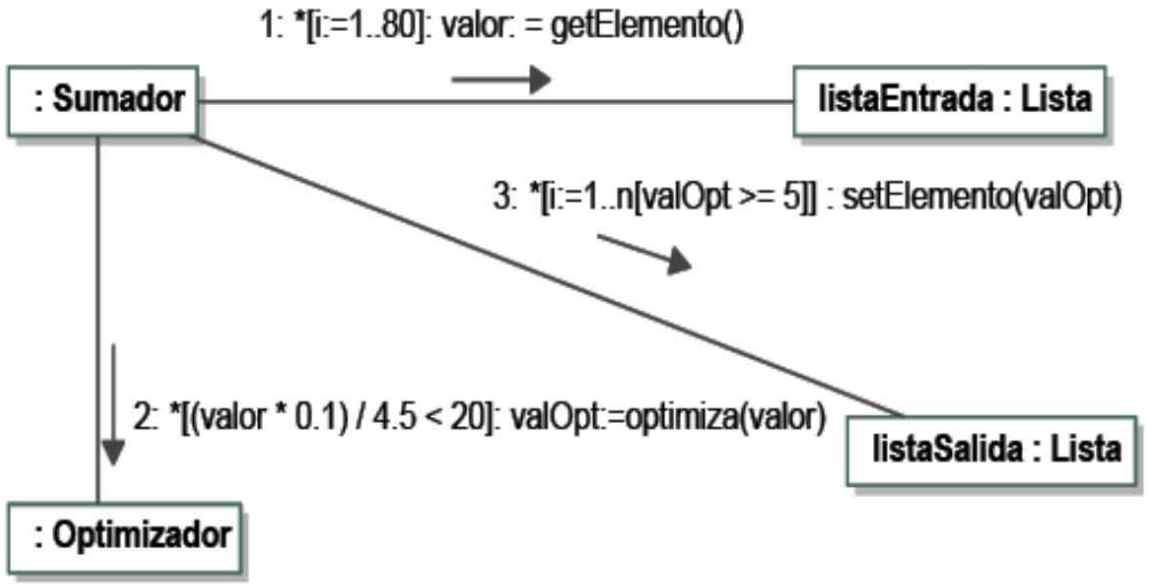


Figura 8.5. Ejemplo de iteraciones y condicionales

La figura 8.5 es un claro ejemplo de un diagrama de comunicación con una iteración normal y otra con condición. Como se puede apreciar, el objeto *Sumador* itera 80 veces sobre *una lista de entrada* para extraer sus valores. Cada uno de dichos valores será enviado al *Optimizador* siempre que cumplan la guarda del mensaje (2). Finalmente, cada valor optimizado se enviará a la lista de salida (3) para almacenar los elementos optimizados cuando se cumple la condición de su guarda.

caso de estudio: ajedrez

El diagrama de la figura 8.6 ilustra un escenario del juego del ajedrez donde se recibe, a través de la fachada de comunicaciones, un mensaje que informa del movimiento de un peón por el jugador remoto. Obviamente, el host local debe actualizar gráficamente dicho movimiento. La secuencia de llamadas que mejor muestra esta situación es la siguiente:

- 1: El objeto*Notificador* realiza tres iteraciones informando a las interfaces*I_observer* que se ha recibido un mensaje por el socket.
- 1.1 El objeto*Control_juego* que implementa la interfaz*I_observer* recoge el mensaje y obtiene el mapa de piezas del rival.
- 1.2 El objeto Control_juego informa al objeto rival local que tiene que mover el peón 4.
- 1.2.1 y 1.2.2: El objeto*Jugador_humano* que representa al *rival* remoto actualiza la posición y dibuja la pieza en el *backbuffer*.
- 1.3 Finalmente se redibuja el tablero completo sobre la memoria de vídeo.

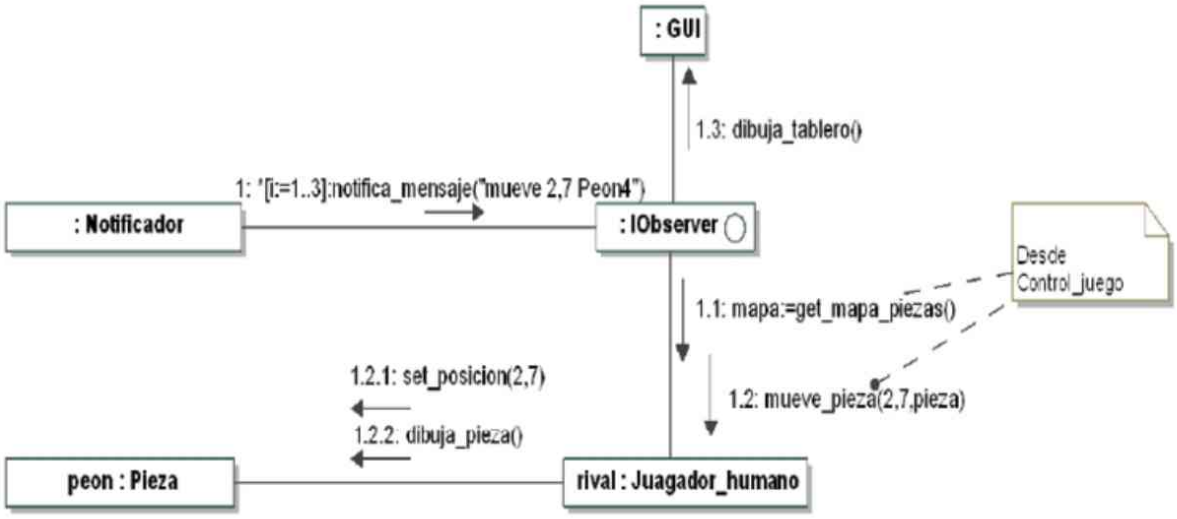


Figura 8.6. Diagrama de comunicación para un movimiento de pieza del rival

caso de estudio: mercurial

Completamos la sección de diagramas de comunicación explicando el caso de estudio de *Mercurial*. En el escenario propuesto en la figura 8.7 se describe la secuencia de interacción del diagrama de secuencias homólogo visto en el capítulo siete. La secuencia de pasos es muy similar, aunque en este diagrama se ejemplifica el borrado del fichero: "a/b/p2.c".

- 1: El actor *Programador* introduce un texto con el comando de borrado en el terminal del sistema.
- 1.1: La fachada del intérprete realiza el análisis de la cadena de texto.
- 1.2 Se pasa el objeto*Comando* con toda la información desglosada al objeto *ClienteProgramador*.
- 1.2.1 El objeto*ClienteProgramador* llama a su método*borrarFichero,* que internamente invoca al objeto que representa al directorio raíz pasándole el primer elemento de la jerarquía.
- 1.2.2 Se itera con el siguiente elemento de la jerarquía. Esta vez el objeto del directorio "a" nos devuelve uno de sus hijos que coincide con la secuencia.
- 1.2.3 Se realiza el mismo proceso que en 1.2.2 pero sobre el directorio "b" para obtener su hijo "p2.c".
- 1.2.4 Finalmente, una vez encontrada la coincidencia, se procede a eliminar el fichero.

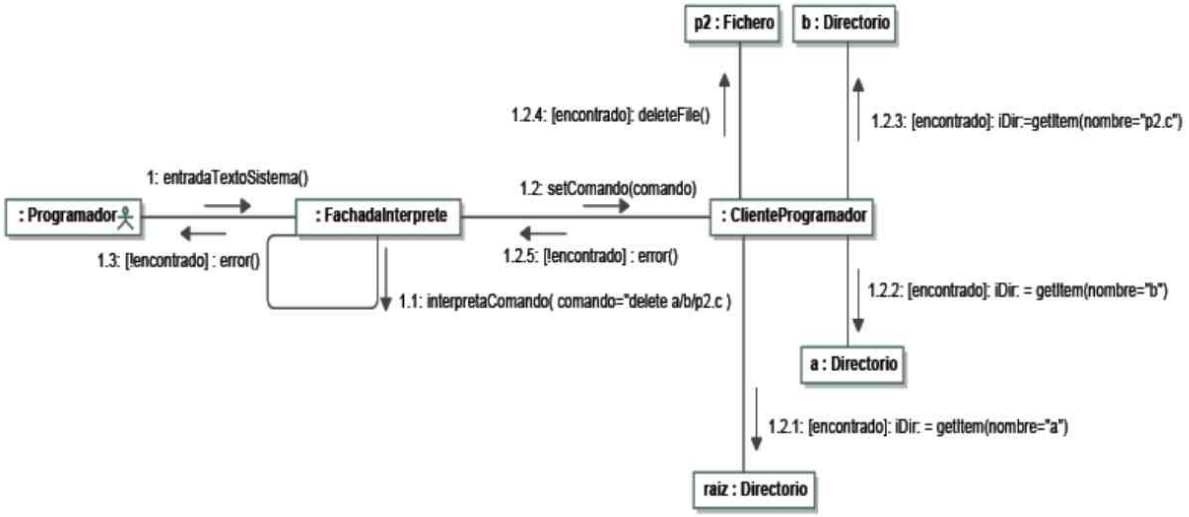


Figura 8.7. Diagrama de comunicación para el borrado de un fichero local

caso de estudio: servicio de cifrado remoto

El diagrama de la figura 8.8 puede ilustrar perfectamente el caso de uso para la realización de un descifrado de un mensaje tanto simétrico como híbrido.

- 1: La fachada de comunicaciones informa al objeto *Notificador* de la llegada de un mensaje cifrado junto con la clave de sesión también cifrada.
- 1.1: El objeto *Notificador* informa a sus observadores (*UsuarioAvanzado*) de la llegada de un mensaje cifrado.
- 1.1.1: Procede a crear el objeto *DescifradorSimetrico*.
- 1.1.2: Llama al método abstracto, *descifra,* para descifrar el mensaje sobre el objeto *DescifradorSimetrico*.
- 1.1.3: Si se produce un error de excepción de descifrado, crea el objeto *DescifradorAsimetrico*.
- 1.1.4: Vuelve a llamar al método abstracto, *descifra*,para descifrar la clave de sesión en *DescifradorAsimetrico* con la clave privada del servidor.
- 1.1.4.1: Retorna la clave de sesión simétrica descifrada con clave privada RSA.
- 1.1.5: Descifra el mensaje mediante el algoritmo representado por el objeto *DescifradorSimetrico* mediante la clave de sesión AES previamente descifrada.

En el caso del lado cliente la secuencia de mensajes de cifrado es similar, por lo que se deja la resolución del diagrama de comunicaciones para esta parte como ejercicio propuesto.

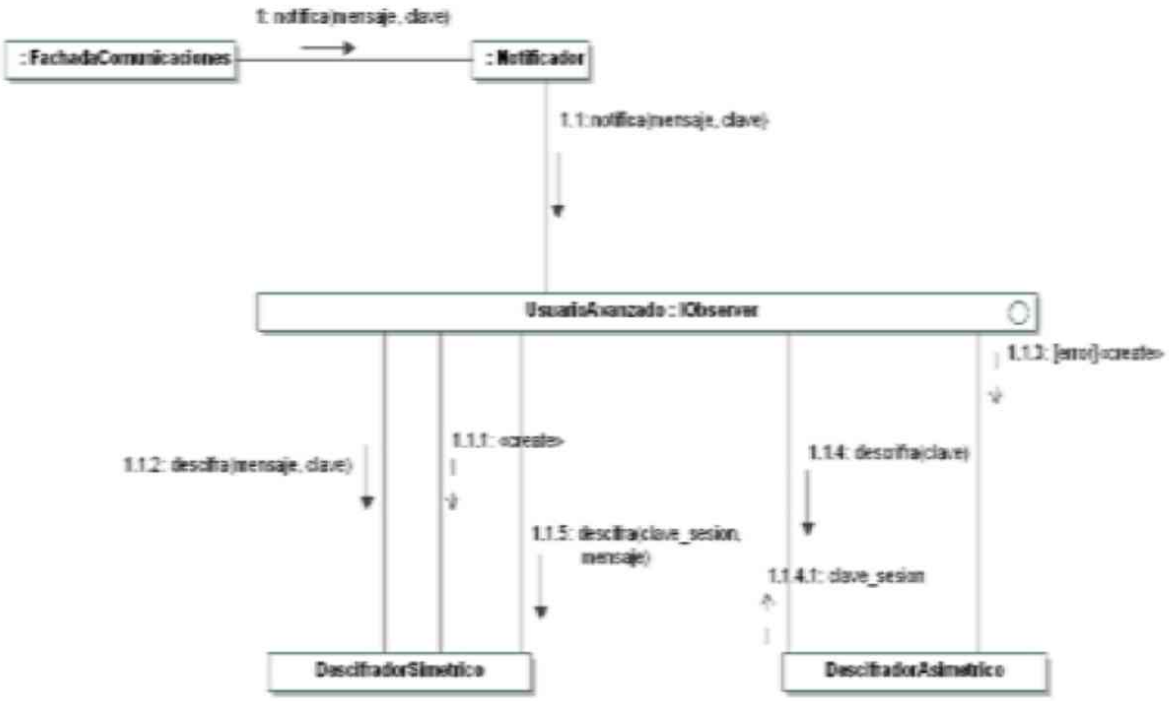


Figura 8.8. Diagrama de comunicación para el caso de uso de descifrado

23 Dicha acción podría ser un caso de uso.

24 El operador " | " significa disyunción.

25 El orden de transmisión de control sería prácticamente como el de la lectura del índice de un libro, es decir, el control se transmite mientras exista una secuencia hija para una determinada secuencia padre.

26 Se supone que se puede expresar en pseudocódigo o en un lenguaje de programación actual.

patrones de diseño

«La imaginación, cuando se ve forzada a trabajar dentro de un marco muy estricto, debe realizar el mayor de los esfuerzos, lo que le llevará a producir sus mejores ideas».

(T.S. Eliot).

Dentro del diseño orientado a objetos el ingeniero de software debe decidir cómo va a orientar su solución al programa. En la fase de diseño existen unos cuantos principios elementales que se deben considerar antes de proceder a diseñar el modelo del sistema.

Durante años, los programadores han intentado dar forma a los problemas de implementación que se les presentaban reiteradamente en diferentes contextos. Después de años de experiencia han podido identificar, aislar y documentar estos problemas que se repetían y darles un nombre.

Por tanto, un buen diseñador debe ser capaz de reconocer estructuras que ya le han surgido en otros contextos y, en consecuencia, reutilizar las soluciones. Es entonces cuando surgen los *patrones de diseño* con la idea de "no volver a reinventar la rueda" para cada situación similar.

En este capítulo se detallarán algunos de los patrones más utilizados en la fase diseño de software, puesto que su aplicación es una actividad cada vez más frecuente en el desarrollo de aplicaciones orientadas a objetos y otras disciplinas técnicas.

Finalmente, veremos cómo aplicar los patrones de diseño a los casos de estudio tratados a lo largo del libro, con el fin de mostrar la reutilización de estructuras predefinidas con cualidades invariantes en casos reales.

¿por qué patrones?

La idea partió en un principio del arquitecto *Christopher Alexander*[27] en los años 70. Este autor se dio cuenta de que en todas las áreas de la ingeniería y la arquitectura están presentes los principios de la reutilización por medio de patrones y que por tanto existen problemas recurrentes de arquitectura que tienen una solución común. Por ejemplo, en la construcción de edificios existen patrones arquitectónicos en relación al diseño de la fachada o de los tejados y normativas en relación al medio ambiente o a la acústica. Los neumáticos de los automóviles siguen una normativa en cuanto a diámetro, material y seguridad.

En el mundo de la Ingeniería Informática, en concreto, existe cierto paralelismo con la arquitectura antigua de edificios, así un *patrón software* es una estructura formal que ya ha sido descubierta durante años de experiencia en programación y diseño de software. En consecuencia, la reutilización e identificación de estas estructuras invariantes por los ingenieros de software lleva a una solución correcta para situaciones concretas del desarrollo del proyecto.

En los patrones software deben estar presentes los principios de modularidad, reutilización, las especificaciones funcionales, el diseño de la interfaz de usuario y la interconexión o acoplamiento entre subsistemas.

En el lado opuesto surge el concepto de *anti-patrón* el cual se concibe como una solución errónea para el diseño. Cuando se documenta un anti-patrón se informa al diseñador de cómo debe evitar escoger soluciones que a priori parecen fáciles pero a posteriori conducen a resultados nefastos. Existen una gran variedad de anti-patrones, todos ellos clasificados por categorías y que el ingeniero debe evitar en la medida de lo posible.

El *framework* (armazón en inglés) suele ser un sistema de clases reutilizable orientado a facilitar la construcción de diferentes aplicaciones en un dominio. Por ejemplo, *frameworks* de construcción de interfaces de usuario o de

desarrollo de videojuegos.

A menudo un patrón suele confundirse con un *framework*. Normalmente un framework contiene varios patrones y no tiene tanto grado de abstracción como estos. En general, el framework se diferencia del patrón principalmente en el tamaño, puesto que los patrones son elementos arquitectónicos más pequeños y están menos especializados que los frameworks.

tipos de patrones

A lo largo de la historia los patrones se han clasificado de diferentes formas. En esta sección proponemos una manera sencilla de organizarlos:

- *Patrones arquitectónicos*: los patrones arquitectónicos proporcionan soluciones reutilizables a problemas basados en la organización estructural a alto nivel de un sistema software. Consiste en la organización modular de los principales componentes o subsistemas a modo de grandes bloques funcionales.
- *Patrones de diseño*: los patrones de diseño definen estructuras para dar respuesta a un problema en la fase de diseño detallado, para ello es necesario que el patrón resuelva eficaz y eficientemente dicho problema y pueda reutilizarse en diferentes contextos.
- *Patrones de codificación*: estos patrones están referidos a las diferentes estrategias de implementación de un programa en un determinado lenguaje.
- *Patrones de análisis*: los patrones de análisis se utilizan fundamentalmente para la creación de modelos en esta fase.

patrones arquitectónicos

La arquitectura organiza la aplicación en grandes bloques bien definidos. Este paso es el primero en la fase de diseño de alto nivel y en la que se basará para la división del trabajo en equipos. La clasificación que proponemos a continuación, junto al apartado 5.1, es un esfuerzo por recopilar las diferentes visiones de la arquitectura del software desde el punto de vista de los patrones.

Sistemas genéricos

capas

El patrón *Capas* ("layers") organiza la aplicación en una serie de bloques superpuestos en las que las capas superiores solicitan servicios que las capas inferiores proveen. Un ejemplo de esta arquitectura se representó en la figura 5.2. Un caso típico son las aplicaciones de "tres capas" para la implementación de software empresarial (véase figura 9.1).

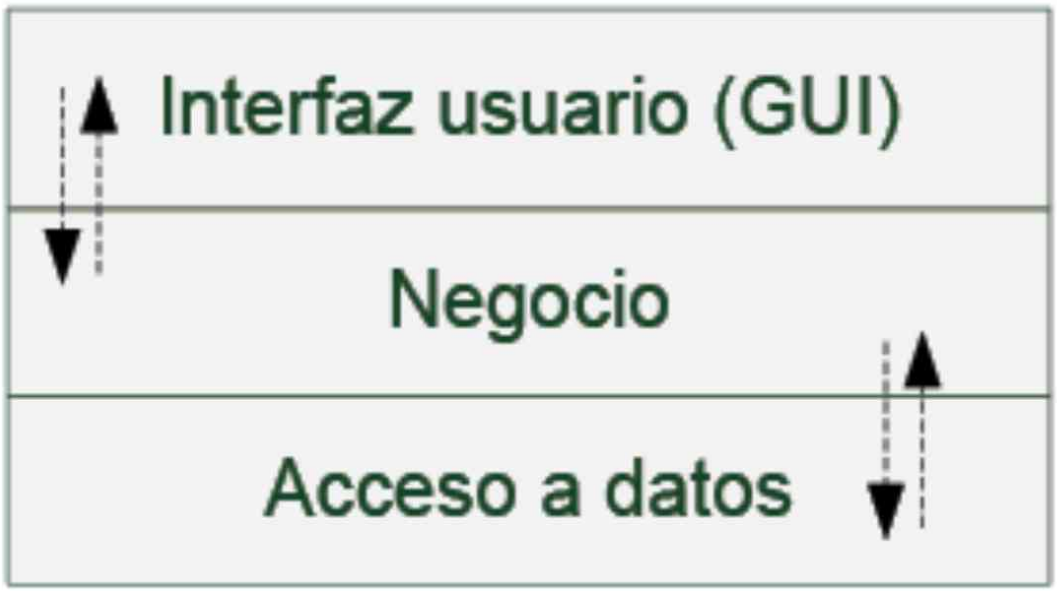


Figura 9.1. Modelo de "tres capas"

Como se puede apreciar en la figura 9.1, la *capa de acceso a datos* se encarga de encapsular las clases de acceso al sistema de gestión de base de datos específico. La *capa de negocio* recoge la información proporcionada por la capa inferior para tratarla según la reglas de funcionamiento de la organización o dominio. Finalmente, la *capa de interfaz de usuario* muestra los datos al cliente para su visualización. El ejemplo de la figura 9.1 muestra un caso de lectura de datos, aunque también es posible la situación inversa.

tuberías y filtros

Este tipo de arquitectura que ya vimos en la figura 5.1 permite el flujo de datos entre procesos a través de conectores llamados tuberías. En orientación a objetos se definiría una clase por cada elemento relacionado con las tuberías y los filtros. Un ejemplo de esta arquitectura se puede observar en las fases de compilación y los sistemas operativos como UNIX.

pizarra

Este patrón consta de un elemento central llamado "Pizarra" y una serie de fuentes de conocimiento que actualizan información sobre la estructura central ("Pizarra") (véase figura 5.3). La pizarra realiza una función de coordinación sobre las fuentes que son las encargadas de leer la información, procesarla autónomamente y devolverla a la pizarra. La respuesta se alcanza cuando el resultado en la pizarra satisface los objetivos propuestos al inicio. Esta arquitectura se utiliza comúnmente en sistemas expertos y sistemas multiagente.

plug-in

La arquitectura de "plug-in" permite ampliar las funcionalidades de un sistema como una extensión que se integra en la arquitectura general de la aplicación. Suele acoplarse por medio de ficheros que pueden ser desplegados como librerías de enlace dinámico (*dll* en Windows o *so* en Linux) o también mediante extensiones *ad-hoc* propietarias de los mismos sistemas software. Dichas extensiones o "plug-ins" van a permitir que terceros desarrolladores amplíen o incorporen nuevas características a la aplicación. Es habitual encontrar sistemas de "plug-in" en las aplicaciones de escritorio utilizadas diariamente (correo electrónico, navegadores, herramientas ofimáticas, etc.), pero de especial manera en aplicaciones de ingeniería de sonido, videojuegos, reproductores de audio, herramientas de animación/diseño gráfico y otro software profesional nativo.

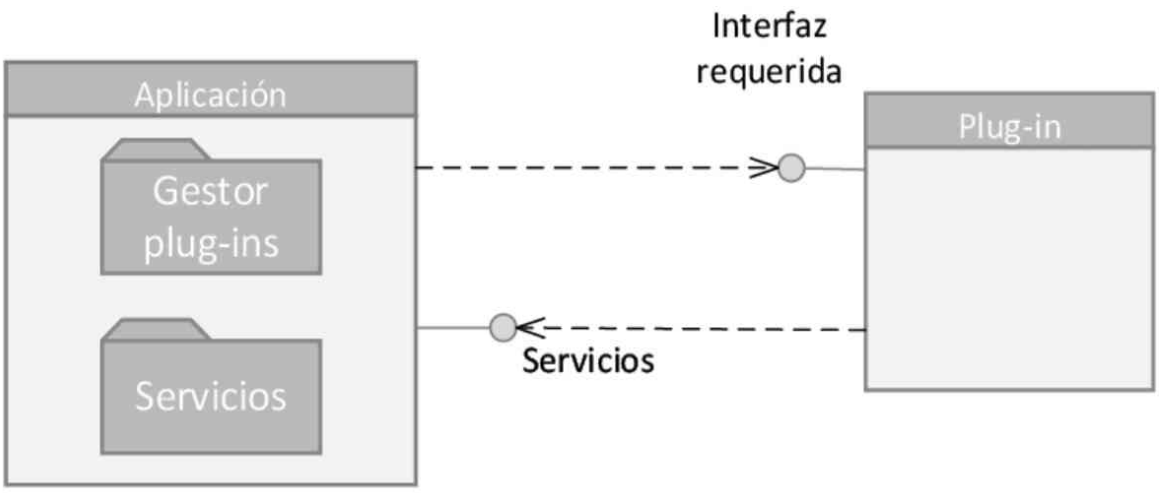


Figura 9.2. Arquitectura de "plug-in". [Lac19].

Sistemas distribuidos

Broker

La ventaja adquirida de su uso es que la aplicación puede acceder a los servicios remotos comunicándose directamente con el objeto requerido, evitando así el acoplamiento entre cliente/servidor y facilitando la escalabilidad, transparencia y heterogeneidad típica de los sistemas distribuidos. En consecuencia se evita la necesidad añadida de acceder a intercomunicación a más bajo nivel. El *broker* o intermediario coordina la comunicación entre el cliente y el servidor notificando cualquier situación de error en la transmisión o invocación. Este patrón es útil en sistemas distribuidos donde se tienen varios *hosts* remotos o CPUs diferentes que interactúan entre sí. Se ha utilizado concretamente en *middelwares* ampliamente difundidos como CORBA o DCOM.

Peer-to-peer

Es un tipo de arquitectura descentralizada donde no se considera la existencia de ningún servidor fijo, conectándose entre ellos mediante la selección de un nodo o nodos como coordinadores. Cada "igual" en una red P2P envía y recibe información hacia los nodos centrales de coordinación o alguno de sus vecinos en una red de interconexión virtual, lógica y abstracta llamada *red de superposición*.

orientadas a servicios (SOA)

Son las más novedosas y permiten la invocación de un servicio remoto mediante un protocolo conocido de comunicaciones. Los más comunes llamados *Servicios Web* permiten la comunicación mediante estándares abiertos, tal como se muestra en la figura 9.3:

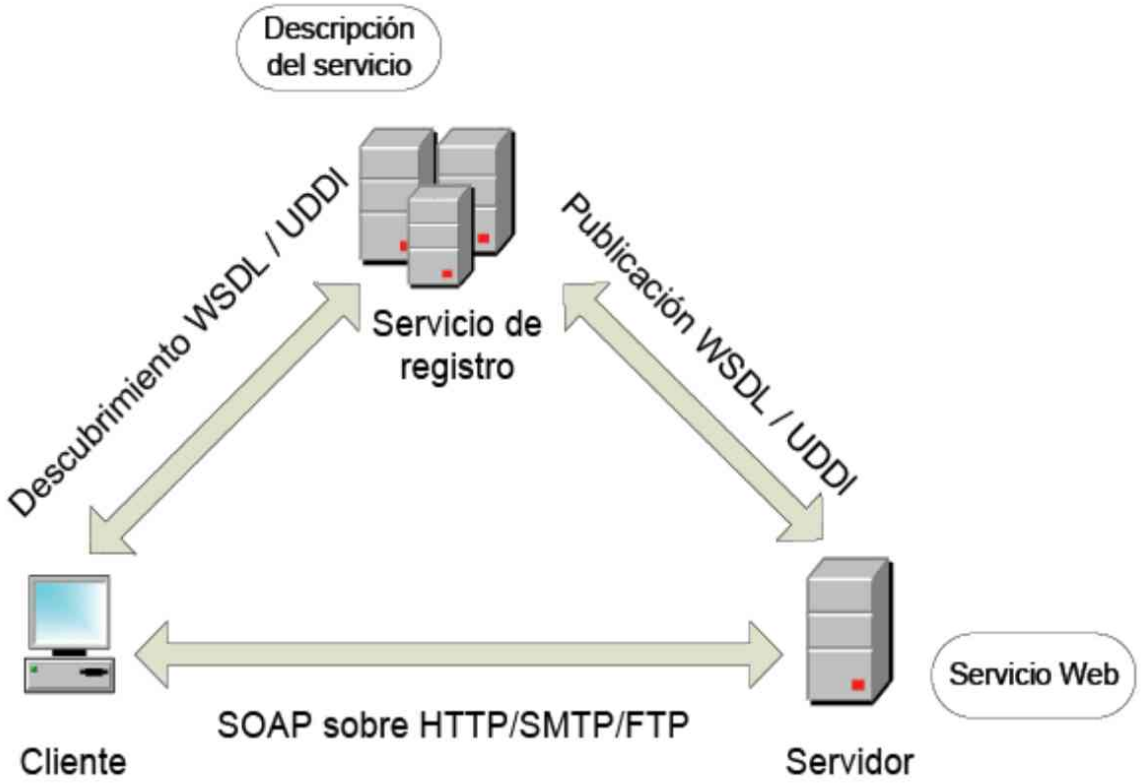


Figura 9.3. Arquitectura de un servicio Web

SOAP (*Simple Object Access Protocol*) es un protocolo que funciona sobre HTTP, FTP o SMTP en XML y permite la intercomunicación y el ordenamiento de mensajes entre el cliente y el servidor. Previo a la comunicación entre cliente y servidor es imprescindible que el servidor anuncie la disponibilidad del servicio mediante el protocolo UDDI (*Universal Description, Discovery and Integration*) con contenido generado con WSDL (*Web Services Description Language*) en codificación XML. Igualmente el cliente debe descubrir el documento WSDL también mediante UDDI.

Sistemas interactivos

Modelo-vista-controlador (MVC)

El patrón MVC (*Modelo-Vista-Controlador*) es otro patrón para el desarrollo de aplicaciones, generalmente empresariales, cuya estructura queda divida en tres componentes como se muestra en la siguiente imagen:

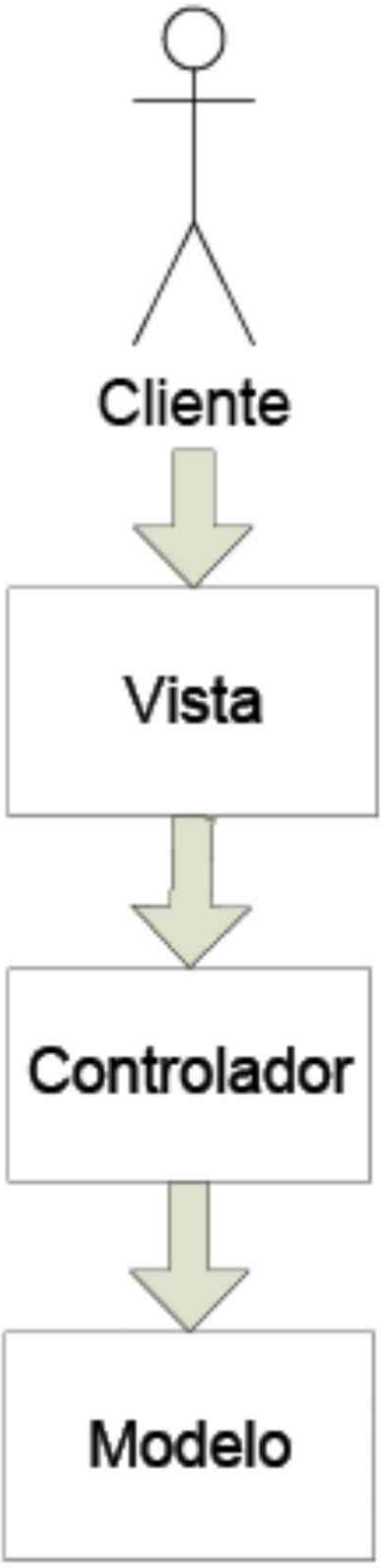


Figura 9.4. Patrón Modelo-Vista-Controlador

La *vista* recibe las peticiones del cliente y las delega al *controlador*. El *controlador*, a su vez, redirige la petición a la función adecuada del *modelo*. El *modelo* implementa la lógica de negocio de la aplicación y la *vista* finalmente muestra los resultados que previamente han sido procesados en el modelo o recibe los datos de entrada provenientes de la GUI.

patrones de diseño GoF

El diseño detallado es la etapa más compleja en la fase del ciclo de vida del software. Es por tanto una etapa en la que la experiencia de diseño del equipo de ingenieros cumple una función crucial. Por este motivo, los patrones vienen en auxilio de aquellas situaciones donde se requiere de una solución prediseñada en el contexto del problema.

Los patrones que veremos a continuación están basados en la clasificación del libro de [Gamma95] cuyos autores son *Erich Gamma, Richard Helm, Ralph Johnson* y *John Vlissides* conocidos en la jerga técnica como como patrones GoF (Gang of Four), ya que recogen una amplia variedad de patrones utilizados en diferentes aplicaciones. Esta clasificación está ordenada según la orientación del patrón, dividiéndose en tres bloques principales dependiendo de si el enfoque es para la creación de objetos, las estructuras de datos o el comportamiento de los objetos en tiempo de ejecución.

Descripción de un patrón de diseño

Según [Gamma95] los patrones se especifican en varias partes:

1. *Nombredel patrón*. Breve frase que identifica la esencia del mismo.
2. *Intencionalidad*. Justificación del motivo para el cual el patrón es utilizado.
3. *Otros nombres*. Otros nombres alternativos por el cual es conocido.
4. *Motivación*. Muestra un escenario de un determinado problema y explica cómo es resuelto por el patrón.
5. *Aplicaciones*. Ilustra situaciones o escenarios donde es posible aplicar el patrón.
6. *Estructura*. Representación del esquema del patrón en UML 2.x.
7. *Participantes*. Muestra las diferentes clases u objetos que intervienen en el esquema y sus responsabilidades.
8. *Colaboraciones*. Explica la manera en que los participantes colaboran para llevar a cabo sus responsabilidades.
9. *Consecuencias*. Como la misma palabra indica, refleja los resultados de los objetivos inicialmente propuestos.
10. *Implementación*. Aglutina un conjunto de consejos y técnicas a tener en cuenta a la hora de la codificación en un lenguaje de programación.
11. *Código de ejemplo*. Ejemplifica algún caso de implementación en un lenguaje de programación conocido.
12. *Usos conocidos:* Casos de éxito y aplicaciones del patrón en el mundo real.
13. *Patrones relacionados:* Lista de patrones que tienen alguna similitud o son próximos en la idea de diseño que transmiten.

En el presente libro únicamente describiremos al patrón mediante el

nombre, los participantes, la estructura en UML y un ejemplo práctico.

Patrones de creación

Los patrones de creación describen situaciones comunes en donde es necesario tomar decisiones para la creación de objetos de una forma lo más independiente y desacoplada posible y donde se deleguen a las clases apropiadas esta responsabilidad.

abstract factory

El patrón *Abstract Factory* (factoría abstracta) permite crear familias de objetos relacionados (productos) sin necesidad de especificar las clases concretas.
Los principales participantes de este patrón son:

- **AbstractFactory:** Es una clase abstracta que define las funciones virtuales que implementarán las clases concretas de creación.
- **ConcreteFactory:** Hereda e implementa las funciones virtuales de creación de los productos concretos de cada tipo.
- **AbstractProduct:** Clase abstracta para los productos que se van a crear.
- **Product:** Implementa las funciones virtuales definidas en la clase base y que definen la lógica de negocio de cada producto concreto.
- **Client:** Es el responsable de manejar las clases *AbstractFactory* y *AbstractProduct*.

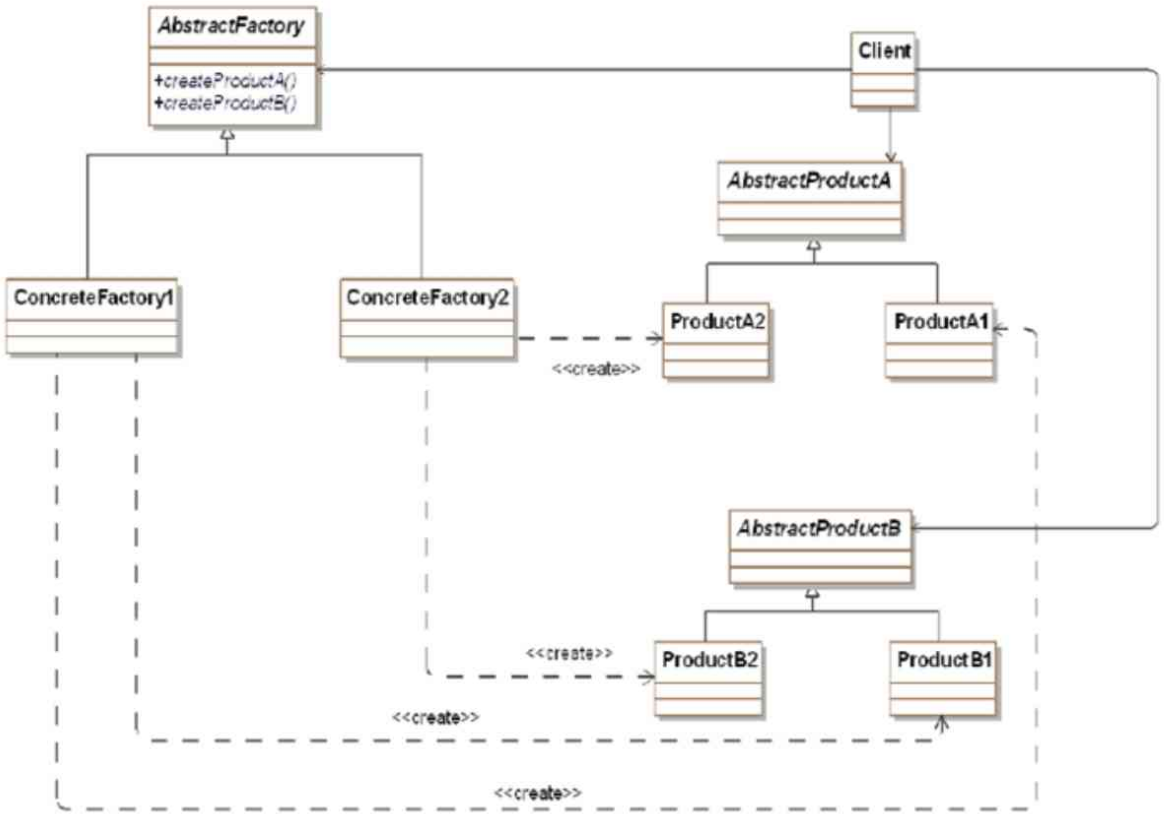


Figura 9.5. Estructura del patrón Abstract Factory

Este patrón es útil cuando se intentan crear varias familias de objetos diferentes, pero puede no ser rentable cuando se añaden nuevos objetos o son muy cambiantes. Por medio de los métodos factoría se pueden crear una variada tipología de objetos para el cliente. El cliente se independiza de las funciones explícitas de creación, limitándose a mantener una referencia a las interfaces de los productos.

El patrón *Abstract Factory* es ampliamente utilizado en la industria del software, especialmente en el desarrollo de librerías y frameworks de interfaces de gráficas.

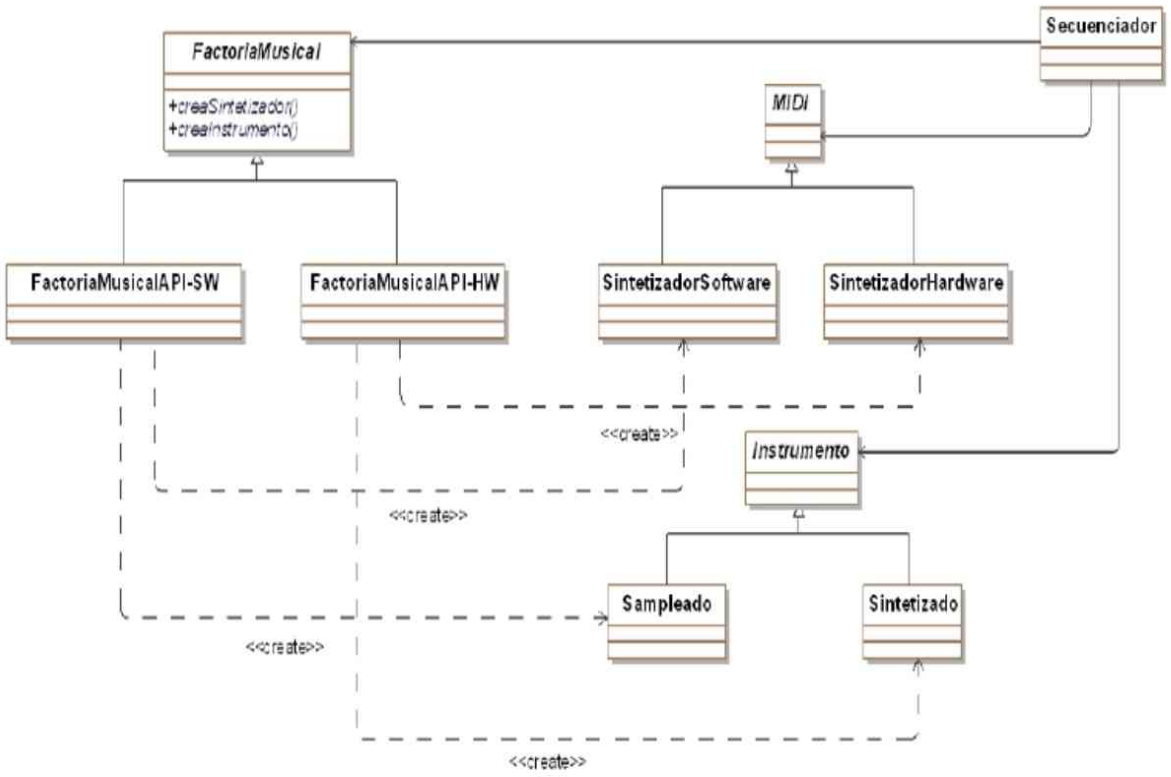


Figura 9.6. Ejemplo del patrón Abstract Factory

El diagrama de la figura 9.6 es un ejemplo de aplicación del patrón Abstract Factory. En él se puede observar al objeto *Secuenciador* (Cliente) que mantiene una referencia a la *FactoriaMusical* (AbstractFactory) que sirve de base para las dos clases de creación de sinterizadotes musicales: emulación por software y acceso directo al hardware. La clase *Secuenciador* también mantiene una referencia a los objetos *Instrumento* que son creados por las factorías musicales software (SW) y hardware (HW). Un instrumento sampleado consiste en una secuencia de bytes con el código PCM (*Pulse code modulation*) de la forma de onda, mientras que uno sintetizado contiene los parámetros de generación para los osciladores y filtros.

El objeto *Secuenciador* se abstrae de los detalles de creación de los productos, delegando esta tarea a las factorías. Una vez creados los sintetizadores y los intrumentos, el cliente procede a interconectarlos para generar sonido.

Con este patrón se consigue desacoplar el sistema principal de secuenciación de sus objetos creados y de cómo estos se crean.

builder

El patrón *Builder* (constructor virtual) permite crear objetos complejos paso a paso y separar la forma de construirlos de una representación concreta.

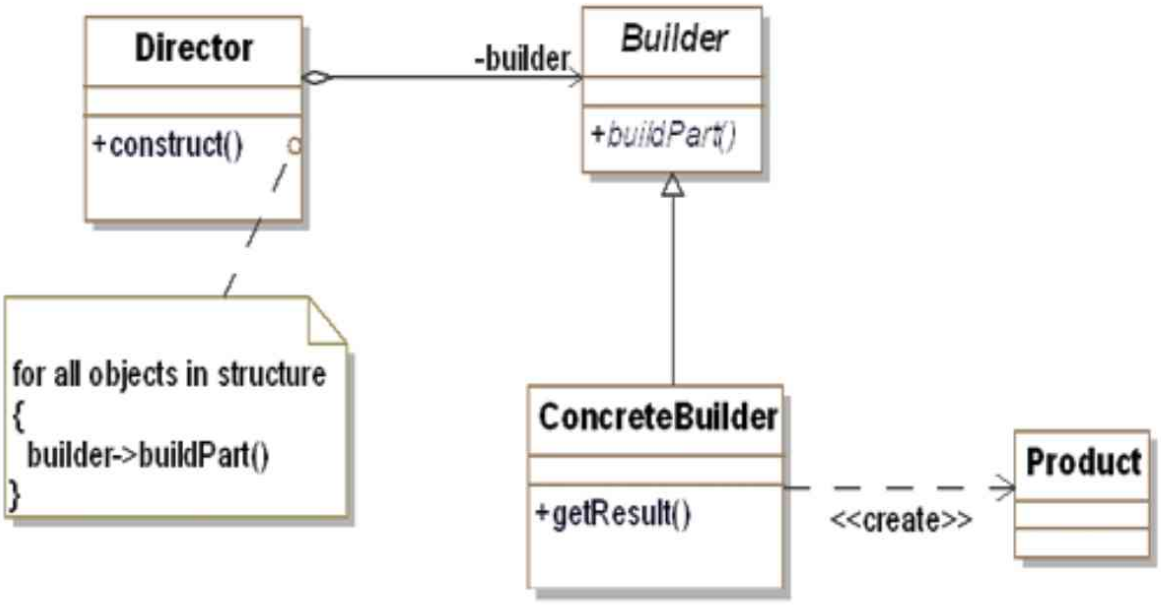


Figura 9.7. Estructura del patrón Builder

Los participantes de este patrón son:

- **Builder**: Es una clase abstracta de tipo interfaz donde se define la función virtual creadora.
- **ConcreteBuilder**: Implementa la clase antecesora con sus funciones para la creación del producto concreto.
- **Director**: Delega la creación de objetos a la clase*ConcreteBuilder*.
- **Product**: El objeto a crear en cuestión.

Una de las ventajas principales del patrón *Builder* es poder reducir el acoplamiento de las clases concretas con la interfaz Builder.

En el siguiente ejemplo se muestra un caso de aplicación de conversión de código HTML a formatos de *LaTeX, Microsoft Word* y *Open Office*. Para que esto funcione se necesita una clase directora que atienda la petición del

cliente con el fin de procesar el código HTML. Cuando se recibe una petición de conversión se procesa el código HTML etiqueta a etiqueta y se redirige la solicitud de construcción del formato de documento al conversor adecuado (ConcreteBuilder) y según se haya instanciado el objeto que representa el tipo de documento a crear.

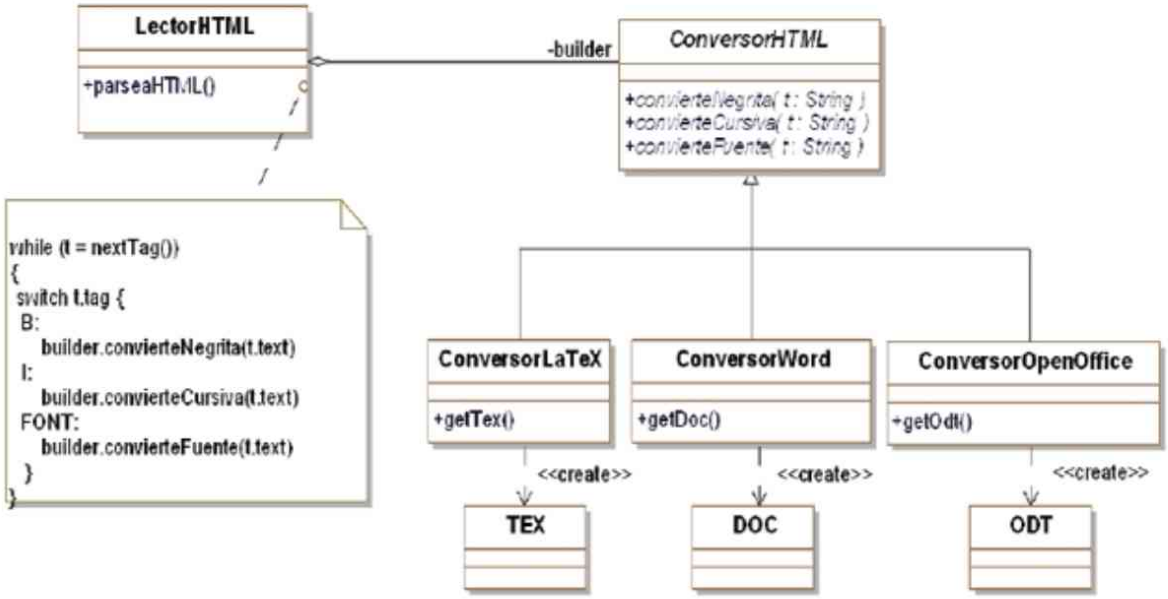


Figura 9.8. Ejemplo del patrón Builder

Más fácilmente se puede entender este patrón en el diagrama 9.9, en donde la aplicación solicita la conversión de código HTML a formato DOC de *Microsoft Word*. Para ello será necesario crear previamente la clase directora (LectorHTML) y la clase *ConversorWord* (ConcreteBuilder). Cuando la aplicación pide la conversión, la clase *LectorHTML* (Director) redirigirá automáticamente cada solicitud de conversión de "tag" al constructor concreto que se necesita para el documento de Word (*ConversorWord*). Finalmente, el documento generado y debidamente construido por esta clase se devolverá a la aplicación solicitante mediante el método *getDoc()* de la clase *ConversorWord*.

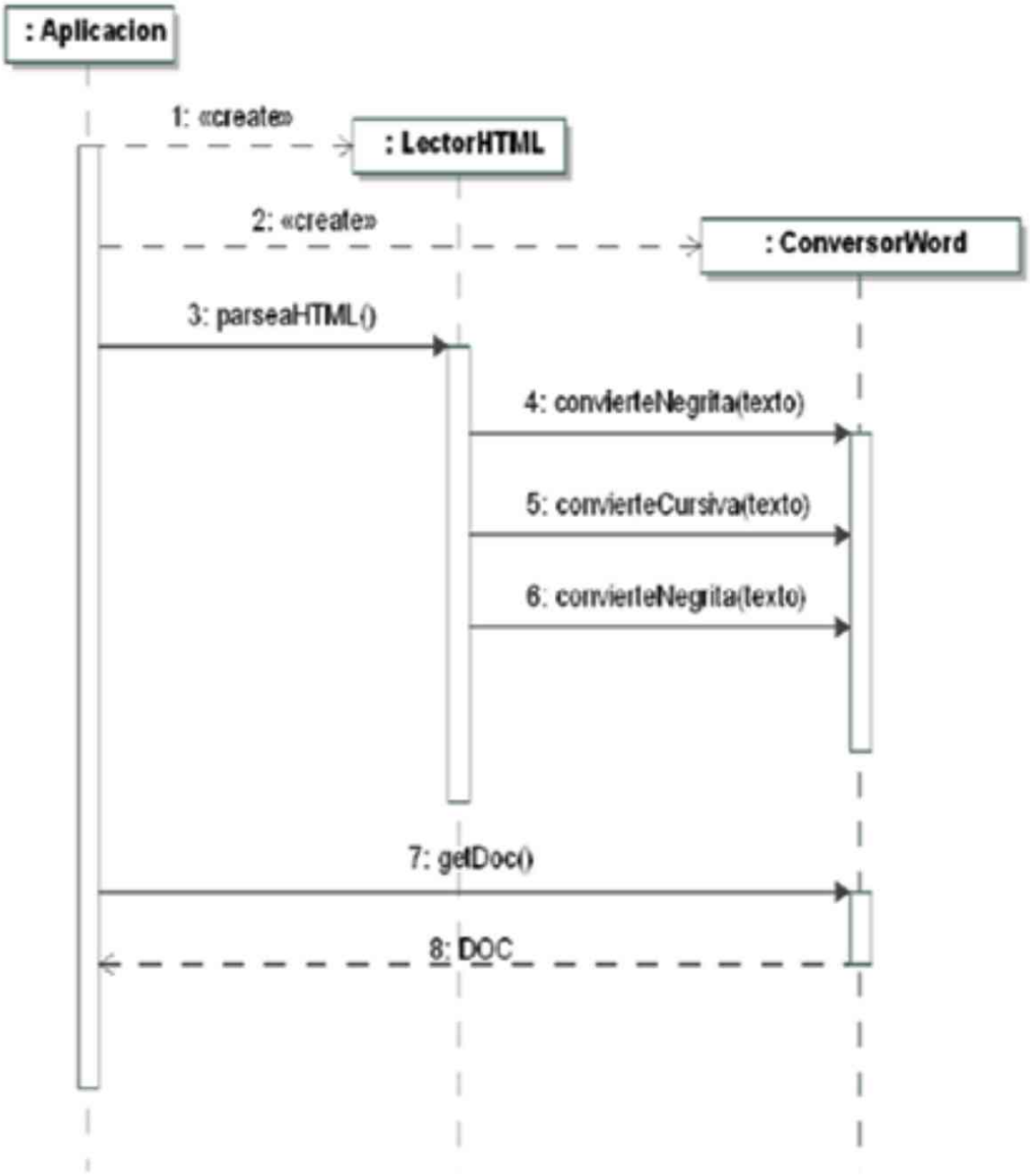


Figura 9.9. Diagrama de secuencias del ejemplo anterior

factory method

El patrón *Factory Method* (método factoría) define una interfaz común para crear objetos permitiendo a las subclases crear los productos concretos.

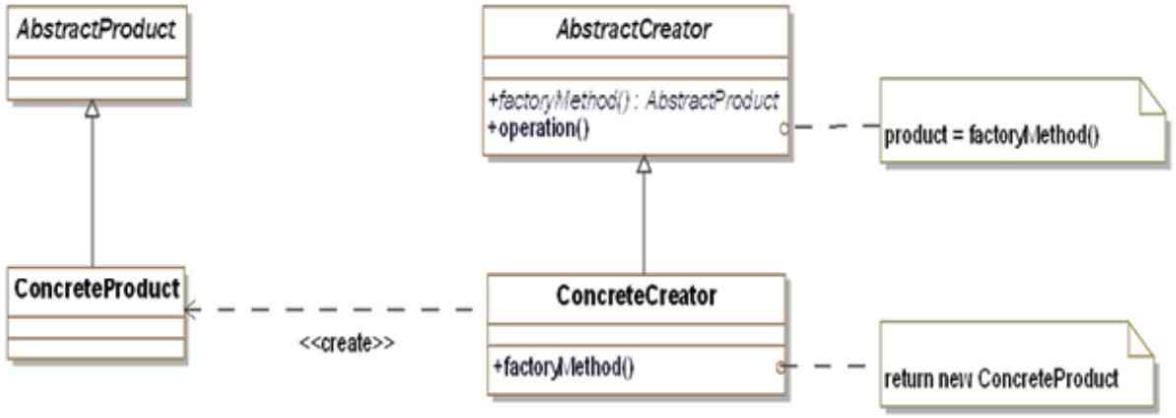


Figura 9.10. Estructura del patrón Factory Method

El patrón consta de:

- **AbstractProduct**: Clase abstracta que define la interfaz del producto a crear.
- **ConcreteProduct**: Implementación de la clase*AbstractProduct* que representa el producto concreto a crear.
- **AbstractCreator**: Clase abstracta que define la función virtual del método factoría y que devuelve un objeto del tipo *AbstractProduct*.
- **ConcreteCreator**: Implementa la clase abstracta antecesora y la función creadora *factoryMethod()* que devolverá una instancia de la clase*ConcreteProduct*.

La ventaja de este patrón es la posibilidad de delegar la responsabilidad a las clases derivadas (*ConcreteProduct* y *ConcreteCreator*) consiguiendo independizar a la aplicación de clases específicas. Esta característica nos permitirá acceder a las funcionalidades de la lógica de negocio de las clases de una forma más precisa y abstracta, diminuyendo de esta manera el acoplamiento y aumentando la cohesión. Una desventaja del uso de este patrón es la necesidad de implementar la clase *AbstractCreator* en la aplicación únicamente para crear un objeto *ConcreteProduct* tal como veremos en el ejemplo de la figura

9.11.

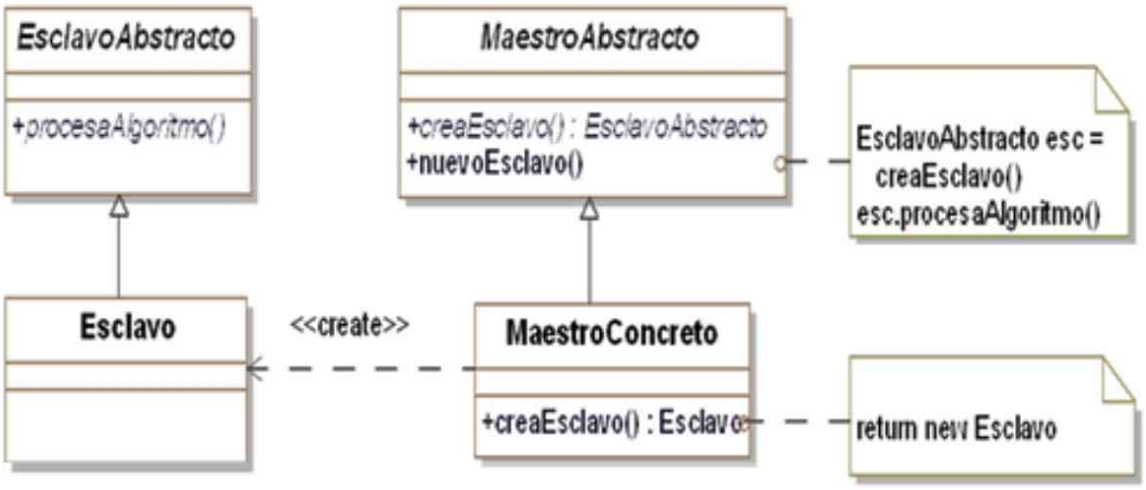


Figura 9.11. Ejemplo del patrón Factory Method

En la figura 9.11 se muestra un caso concreto de aplicación del patrón *Factory Method* en aplicaciones que requieren crear procesos esclavos para repartir la carga del procesamiento. Este ejemplo muestra la separación de roles del patrón, con la clase *MaestroAbstracto* (AbstractCreator) y la clase heredada *MaestroConcreto* (ConcreteCreator). Mediante la delegación de la creación a los métodos abstractos (arriba) se permite definir una interfaz independiente para la creación de esclavos (abajo). Por lo tanto, son las subclases las que decidirán qué objeto concreto desean construir.

singleton

El patrón *Singleton* permite que solo exista una única instancia de la clase a la que se pueda acceder. Imagínese una aplicación que optimiza una unidad de estado sólido; si ejecutamos más de dos instancias de la aplicación, posiblemente no conseguiremos que funcione correctamente.

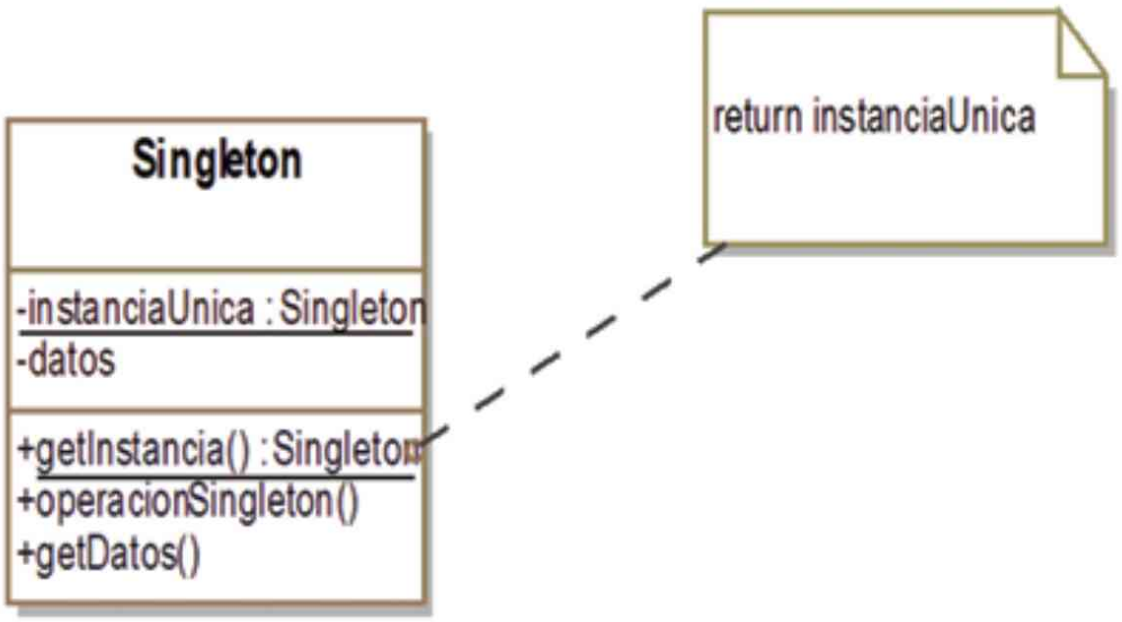


Figura 9.12. Patrón Singleton

El único participante del patrón es:

- **Singleton:** Es el responsable de crear una única instancia de la clase y definir una operación que permita a los clientes acceder a la única instancia.

Una de las ventajas del patrón Singleton es la posibilidad de controlar de forma estricta cómo y cuándo acceden los clientes a dicha instancia. Otra ventaja es la posibilidad de ajustar el número de instancias a una cifra determinada que no sea únicamente una. También evita que se sobrecarguen los espacios de nombres con variables globales, consiguiendo una mejora en la implementación.

En la sección 9.7.2.1 del caso de estudio del servicio de cifrado remoto se explicará la utilización de este patrón en la implementación de la parte servidor.

Patrones estructurales

Los patrones estructurales permiten representar grandes estructuras de datos, facilitando la organización jerárquica y el comportamiento en tiempo real para el manejo de la composición de objetos.

composite

El patrón C*omposite* (objeto compuesto) se utilizará con el fin de representar objetos que a su vez se componen de objetos, o lo que es lo mismo, estructuras de datos recursivas.

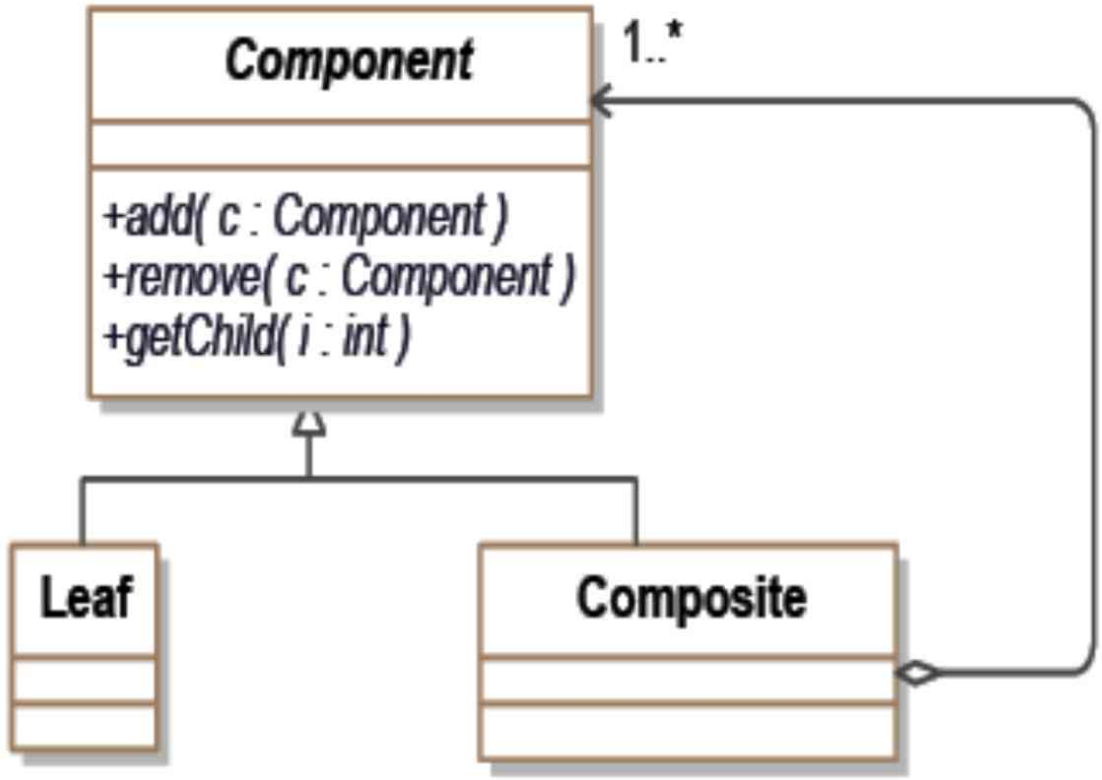


Figura 9.13. Estructura del patrón Composite

Los participantes en este patrón son:

- **Component**: Clase base que define la interfaz de la composición y gestiona a todos sus componentes hijo.
- **Leaf**: Representa a objetos terminales en la estructura recursiva e

implementa las operaciones de negocio de la superclase y las específicas del nodo hoja.

- **Composite**: Implementa las operaciones definidas en la superclase y da forma a la estructura recursiva mediante una agregación (uno-a-muchos) hacia la clase *Component*.

Este patrón será útil cuando necesitemos implementar estructuras de datos recursivas como árboles, grafos y redes acíclicos, etc. De la misma forma, también será práctico para definir relaciones de jerarquía en una aplicación donde los objetos sean tratados por igual.

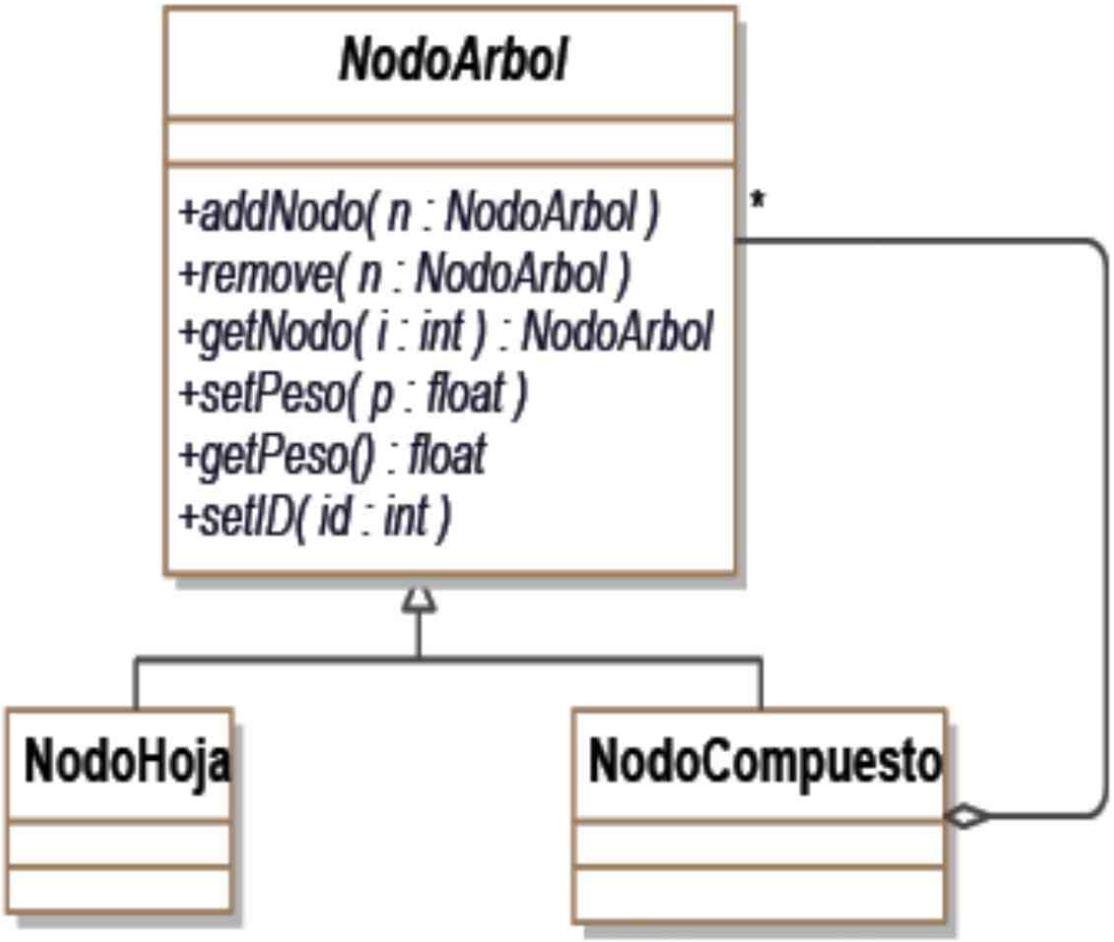


Figura 9.14. Ejemplo de patrón Composite

En la figura 9.14 se modela la estructura recursiva de un árbol mediante la

utilización del patrón *Composite*. La clase base se puede definir como una interfaz o clase abstracta, aunque la utilización de esta última permite abrir el abanico de posibilidades. En la clase base *NodoArbol* se definen los métodos de inserción y eliminación que serán las responsables a todos los efectos de insertar o eliminar los ítems en el array adecuado. Otras funciones adicionales pueden definirse tanto en la clase base como en las clases heredadas de *NodoArbol* para cualquier otro propósito.

Como puede observarse, la clase heredada *NodoCompuesto* contiene una agregación a objetos del tipo de la clase padre *NodoArbol*. La implementación aquí de las funciones virtuales debe asegurar la consistencia de los nodos en la estructura de datos. Al realizar este enlace sobre la propia herencia se consigue el efecto deseado de recursividad que proporciona el patrón. Sin embargo, la clase hija *NodoHoja* no posee agregación, con lo que de esta forma se consiguen objetos terminales en la estructura de árbol.

En la sección 9.6.2 se comenta otra utilización del patrón Composite para el caso de estudio de Mercurial.

facade

El patrón *Facade* (fachada) es una clase que hace de puerto de comunicación o punto de referencia hacia un subsistema. La clase Facade realiza el papel de clase de frontera. En el ejemplo de la figura 9.15 la clase *Facade* interconecta (a modo de frontera de abstracción) el cliente de la parte derecha del diagrama con el subsistema proveedor de la parte izquierda.

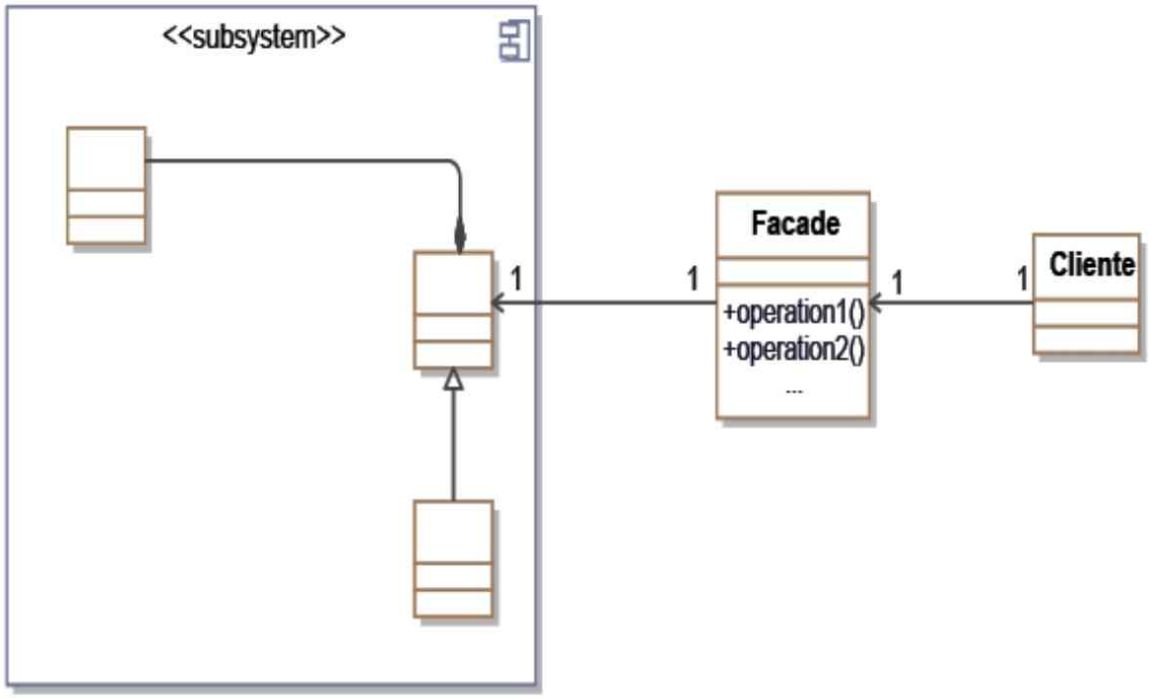


Figura 9.15. Estructura del patrón Facade

En el patrón Facade participan los siguientes elementos:

- **Facade**: Es el punto de interrelación con el subsistema de frontera donde redirigir las operaciones.
- **Subsistema**: Es el conjunto de clases que lleva a cabo una funcionalidad del sistema e interactúan con las operaciones enviadas desde la clase*Facade*.

El patrón *Facade* será ampliamente explicado en las secciones 9.5.1 y 9.6.1.

Patrones de comportamiento

Los patrones de comportamiento describen las interacciones entre objetos y su comportamiento algorítmico en tiempo de ejecución. La separación de la parte cliente de la complejidad algorítmica permite centrarse en los aspectos más relacionados con la interconexión de objetos del dominio y menos en la parte del flujo de control.

command

El patrón *Command* (comando, orden) permite parametrizar las operaciones con la finalidad de controlarlas con más precisión, delegando la invocación de las funcionalidades a las clases concretas responsables de ejecutar el comando. Facilita también las operaciones de deshacer (undo).

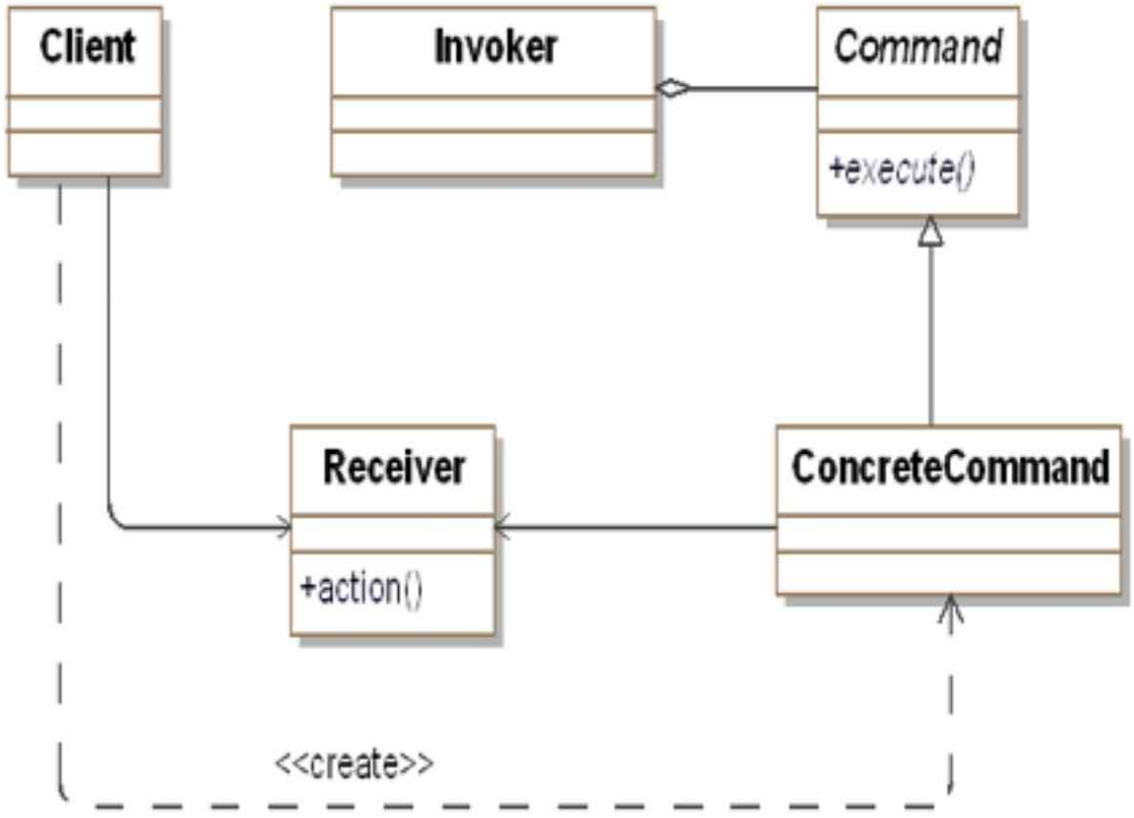


Figura 9.16. Estructura del patrón Command

El patrón Command tiene los siguientes participantes:

- **Command**: Es la interfaz común para las clases *ConcreteCommand*. Define el comando a utilizar.
- **ConcreteCommand:** Implementa la interfaz*Command* y la operación *execute()* que será llamada por el invocador.
- **Invoker:** Este objeto será el encargado de realizar la invocación de la acción mediante la llamada a *execute()*.
- **Receiver:** Realiza la acción definida para una determinada orden del comando.

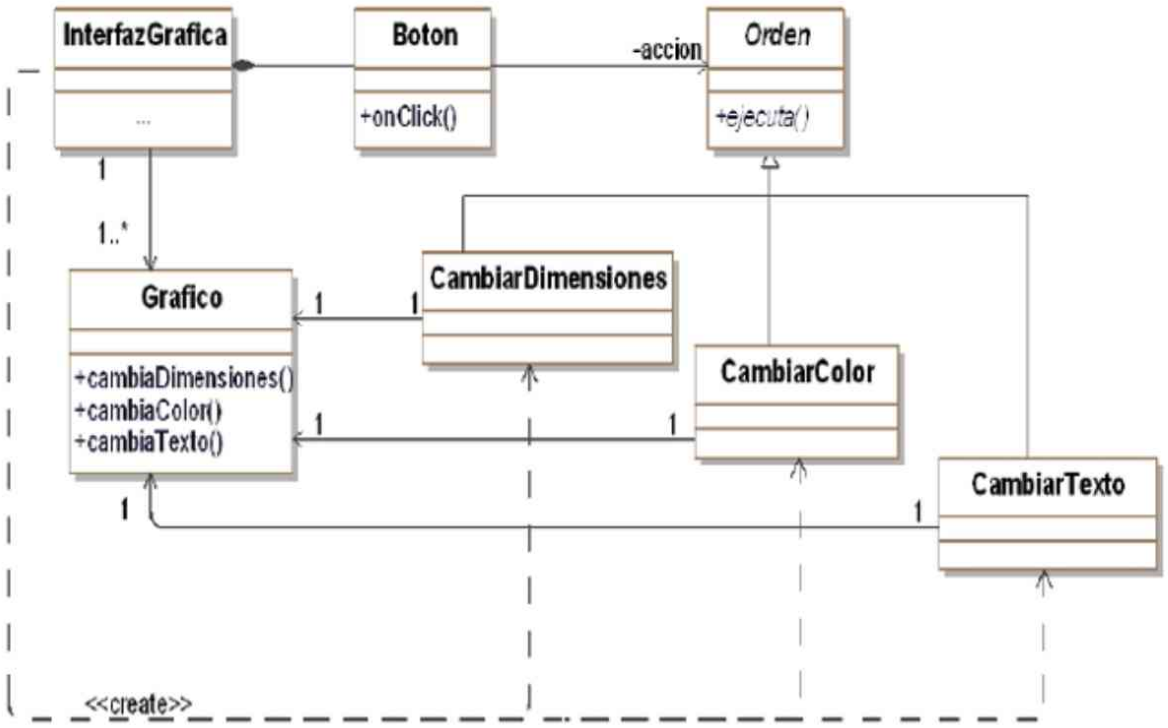


Figura 9.17. Ejemplo del patrón Command

El ejemplo 9.17 implementa fielmente la estructura del patrón *Command*. En este caso se ha aplicado el modelo a la parte de edición de un programa gráfico o editor de textos. Como todo buen editor que se precie, debe proporcionar las opciones de *cambiar color, dimensiones, etc.* Dichas acciones se establecen como clases heredadas de la clase abstracta *Orden*. En estas clases hijas se implementa el método *ejecuta()* que es el responsable de redirigir las

acciones respectivas hacia el objeto receptor del tipo *Grafico*. Una vez recibidas las órdenes (comandos) en el objeto gráfico, se procede a su ejecución dentro del contexto del cliente. Este ejemplo puede entenderse más fácilmente en el diagrama de secuencias de la figura 9.18 de la siguiente página.

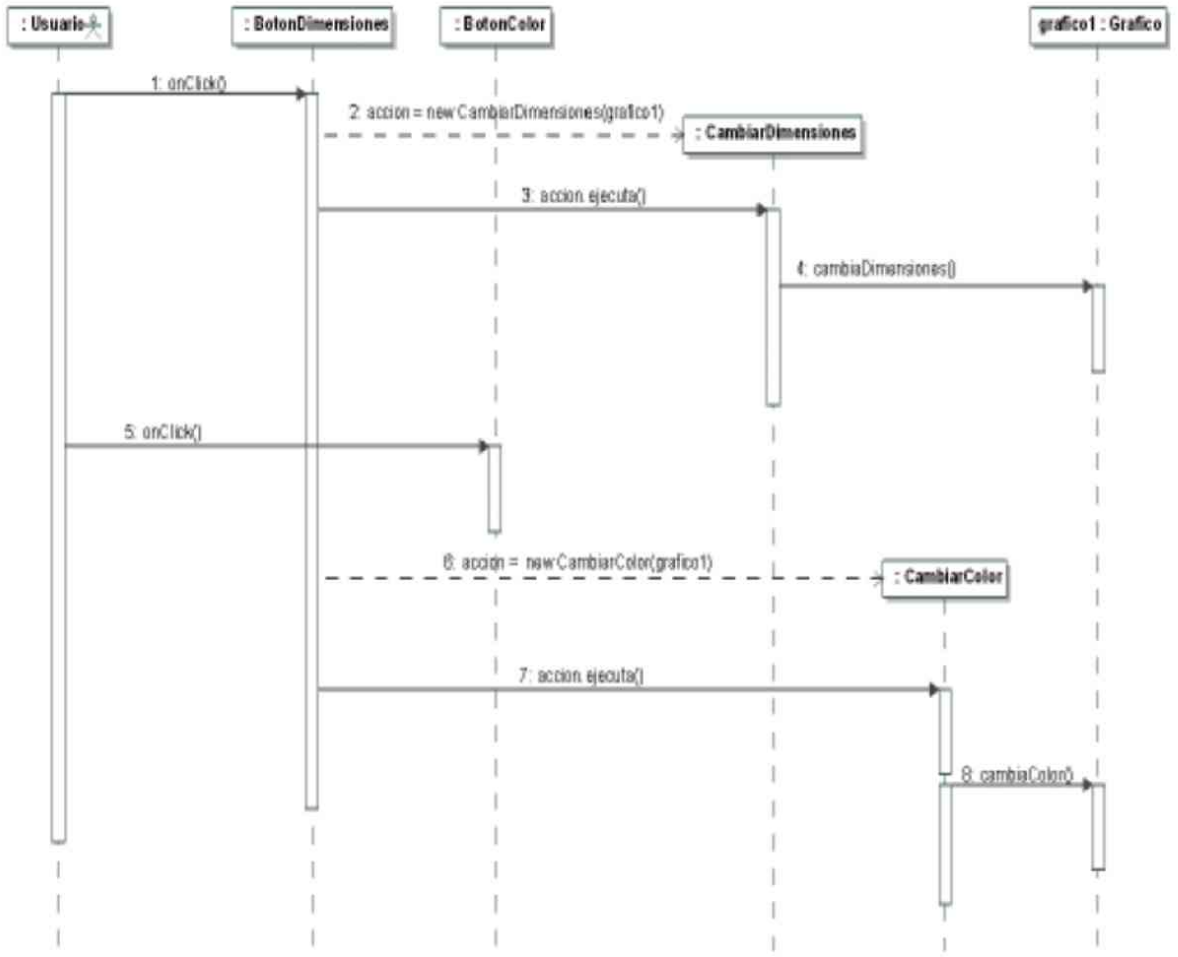


Figura 9.18. Diagrama de secuencias del ejemplo anterior

En el diagrama de secuencias de la figura 9.18 se implementa el escenario de interacción del ejemplo de la página anterior. Como se puede apreciar en los mensajes 2 y 6, antes de proceder a ejecutar el comando concreto debe instanciarse la clase *ConcreteCommand* pasándole como parámetro el objeto destino sobre el que actuar. Una vez instanciadas las clases, el objeto *Command* queda a la espera de un evento para realizar la acción concreta y llamar al método *ejecuta*(). Finalmente será el objeto recibidor el que implementará las acciones pertinentes cuando se produzca un evento del usuario (botón).

observer

Con el patrón *Observer* (observador) es posible delegar la función de notificación de eventos a objetos, separando y desacoplando dentro del sistema a estos objetos de los observadores o receptores de los eventos.

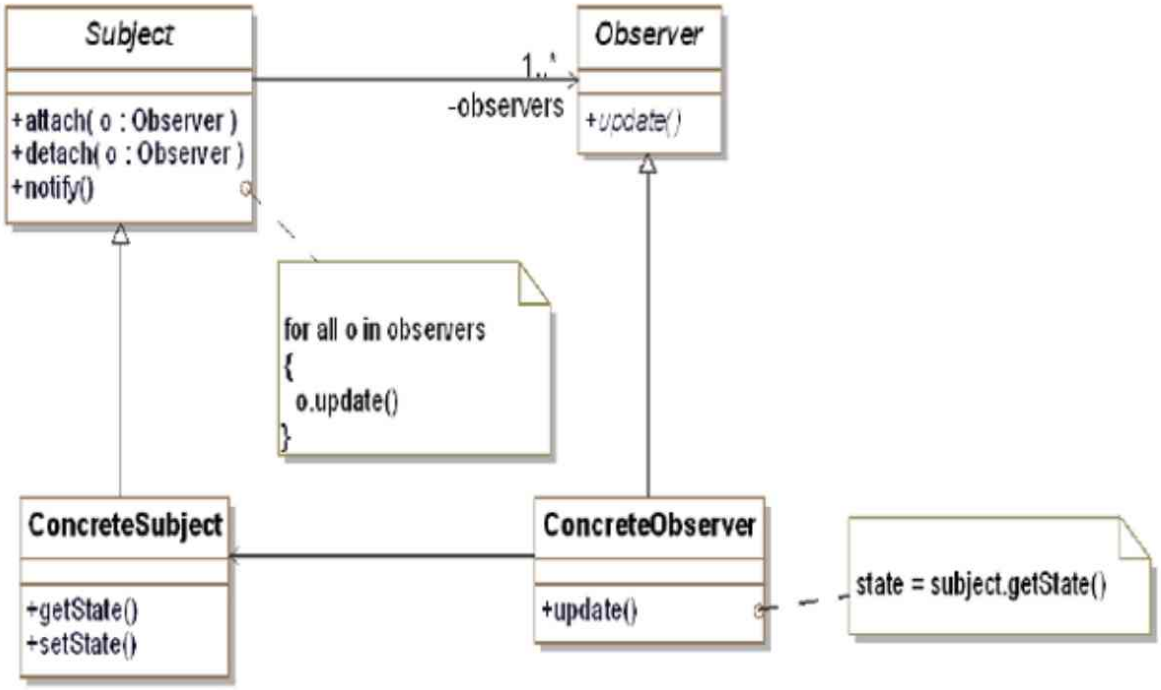


Figura 9.19. Estructura del patrón Observer

Los participantes en el patrón Observer son:

- **Subject:** Define la interfaz con las operaciones abstractas para *ConcreteSubject* y donde se especifican los métodos abstractos de inserción de *Observers*.
- **ConcreteSubject:** Implementa la interfaz anterior y la lógica de negocio que activará las notificaciones.
- **Observer:** Interfaz con la función de recepción de notificación.
- **ConcreteObserver:** Implementa los objetos observadores que recibirán las notificaciones por parte del *ConcreteSubject*.

El patrón Observer es útil cuando es necesario implementar objetos que

estén alerta para la recepción de mensajes de otros objetos. Además, permite delegar y desacoplar las funcionalidades entre el emisor y el receptor. Este patrón suele ser implementado en *frameworks* de interfaces de usuario y librerías de manejo de dispositivos.

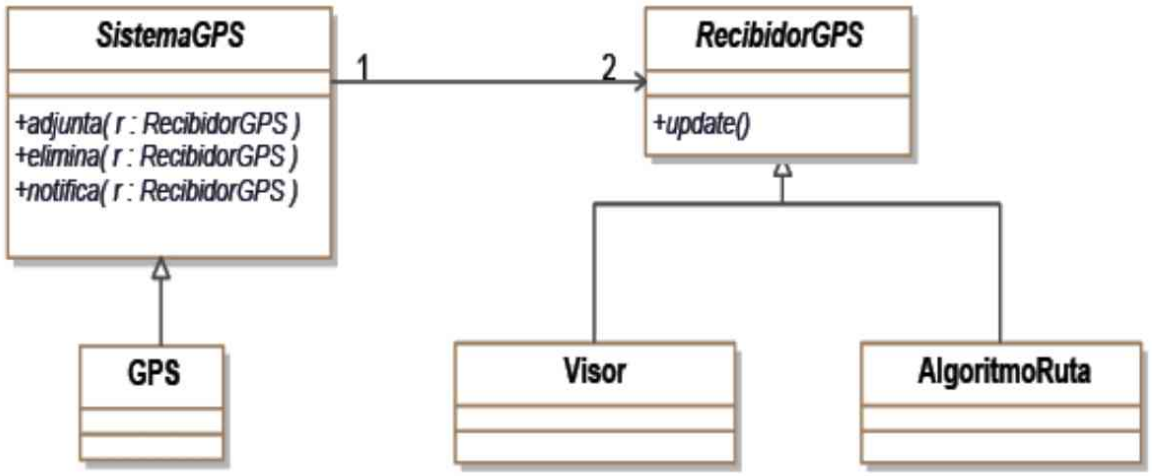


Figura 9.20. Ejemplo de patrón Observer

El diagrama de clases de la figura 9.20 ilustra la aplicación del patrón *Observer* en el software de navegación de un automóvil. El ejemplo divide el modelo entre la clase *GPS* (ConcreteSubject) y los observadores (*Visor* y *AlgoritmoRuta*). Cuando un objeto *Observer* requiere de la notificación de eventos por parte de un *Subject*, llama al método *adjunta()* para subscribirse en la cola de espera de notificaciones. Una vez ocurrido un evento de recepción de coordenadas en el objeto GPS, el método *notifica()* recorrerá la lista de objetos observadores con el fin de hacerles llegar la infomación a través de la función virtual *update()*.

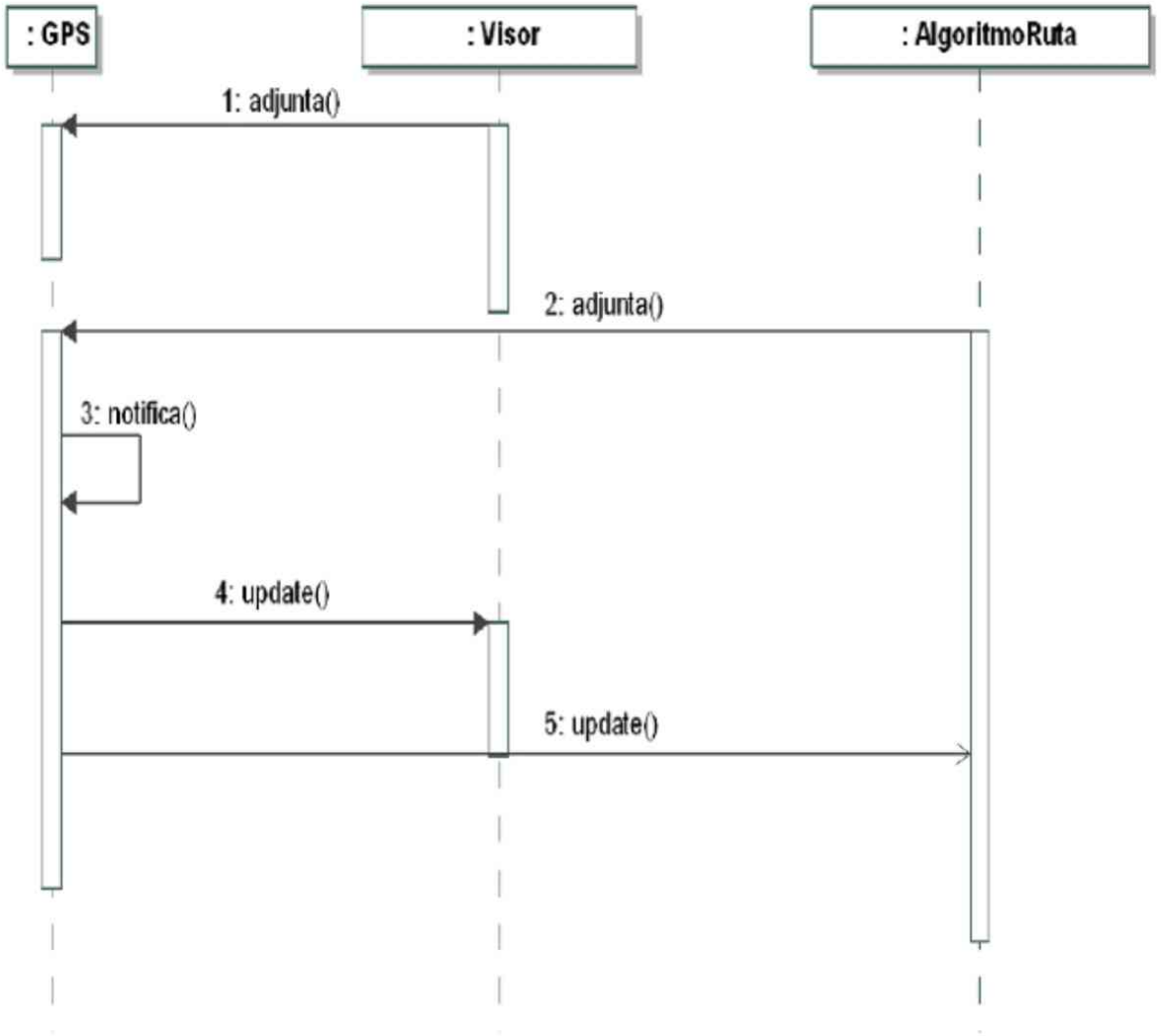


Figura 9.21. Diagrama de secuencias del ejemplo anterior

La figura 9.21 corresponde a las interacciones de los objetos participantes en el patrón *Observer*. Por un lado los objetos *Visor* y *AlgoritmoRuta* se subscriben al sistema GPS mediante la invocación del mensaje *adjunta()*. Cuando la señal del satélite es recibida en el GPS se procede a notificar, mediante la llamada a *update()*, a cada uno de los objetos subscritos, en este caso *Visor* y *AlgoritmoRuta* que heredan de *RecibidorGPS*. El patrón *Observer* se explicará con detenimiento en la sección 9.5.2 y la sección 9.7.2.3.

strategy

El patrón *Strategy* (estrategia) permite seleccionar el algoritmo que se quiere emplear en una situación dada. La forma de aplicarlo es tener una clase base

abstracta y posteriormente definir en las subclases cada algoritmo con una estrategia diferente. La selección del algoritmo se realiza en función del objeto y puede cambiar dinámicamente en tiempo de ejecución mediante funciones virtuales definidas previamente en la clase base.

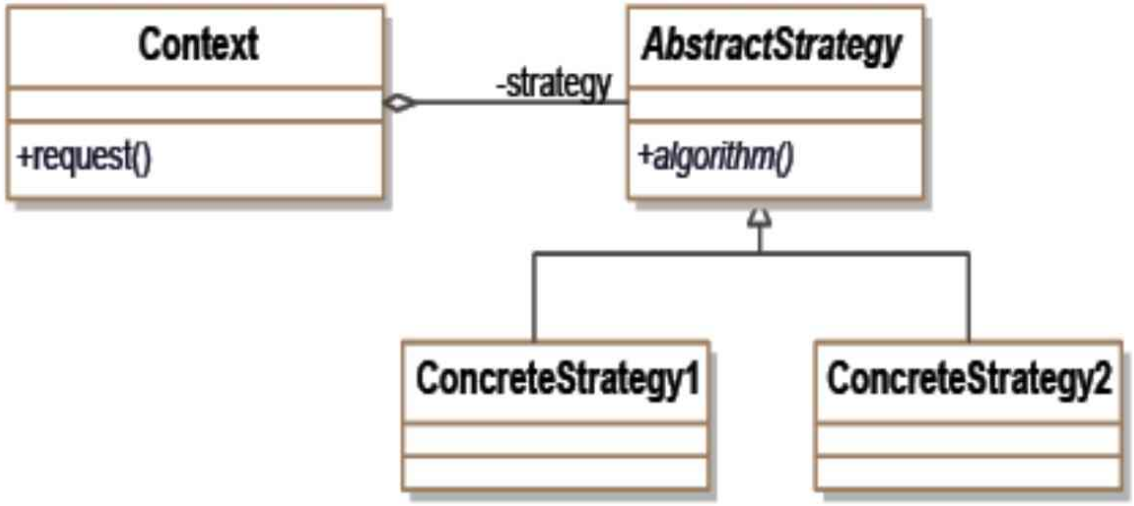


Figura 9.22. Estructura del patrón Strategy

En el patrón Strategy participan:

- **AbstractStrategy**: Define la interfaz base para la clasificación y delegación de las diferentes estrategias.
- **ConcreteStrategy**: Implementa un determinado algoritmo concreto a partir de la interfaz proporcionada en *AbstractStrategy*.
- **Context**: Pertenece al dominio de la aplicación y mantiene una referencia a los objetos estrategia.

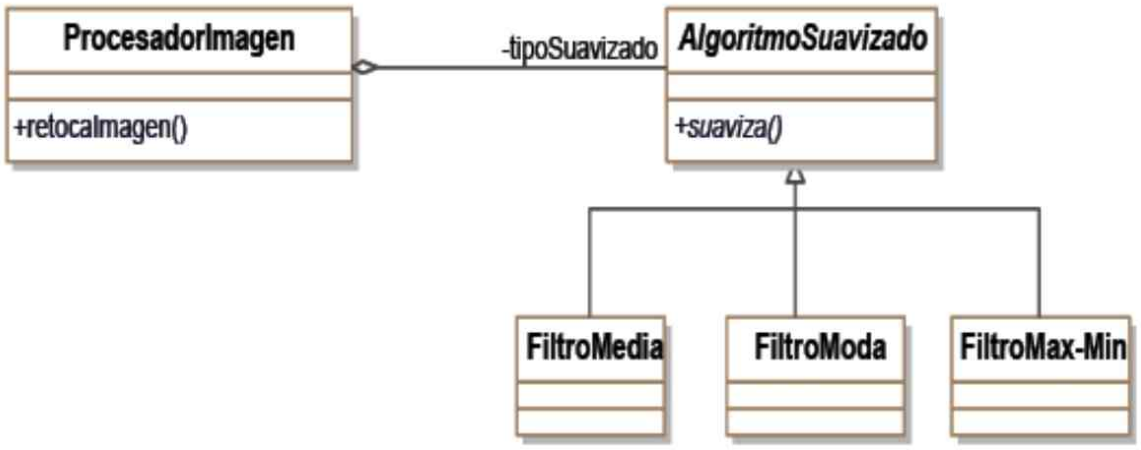


Figura 9.23. Ejemplo del patrón Strategy

La ventaja del patrón Strategy es que permite aplicar diferentes estrategias o tipos de comportamiento delegando según el contexto de la aplicación.

En el ejemplo de la figura 9.23 una aplicación de procesado de imagen invoca al método virtual *suaviza()* con el fin realizar esta operación en el filtro correspondiente. El filtro de la media realiza la operación de eliminación de ruido calculando la media aritmética en un entorno de vecindad de la imagen, mientras que el de la moda calcula el valor de la moda estadística en dicho entorno. El filtro max-min procede de igual forma que los anteriores pero calculando los valores máximos y mínimos de la ventana. El patrón *Strategy* será aplicado al juego de ajedrez en la sección 9.5.3 y en el servicio de cifrado remoto en la secciones 9.7.1.2 y 9.7.2.2.

state

Con el patrón *State* (estado) podemos gestionar los estados por los que transita un objeto delegando el control de los mismos a una jerarquía de clases separada.

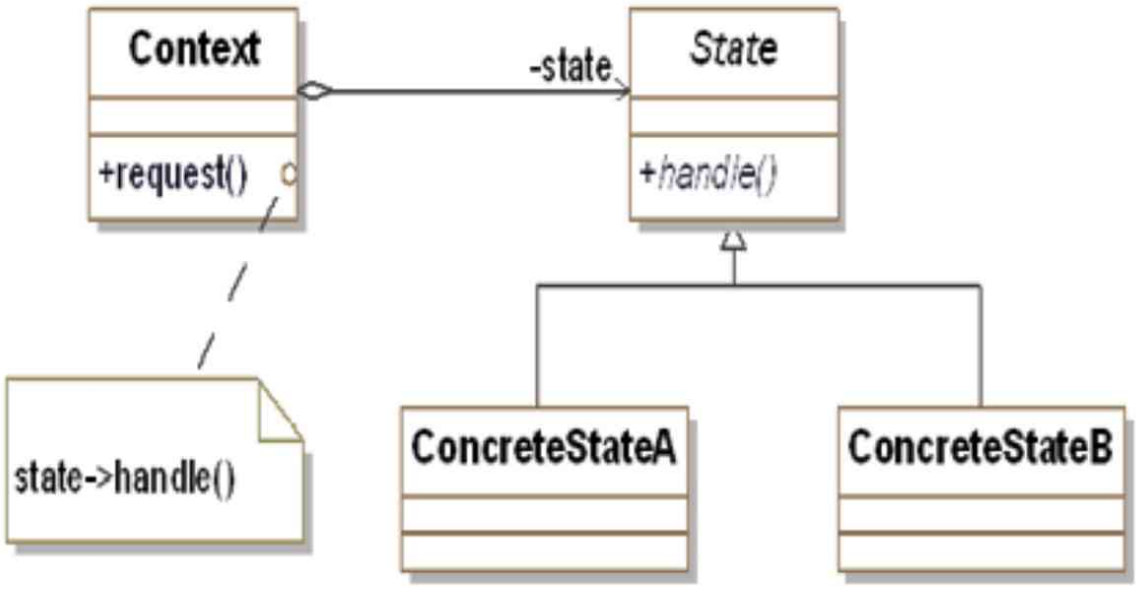


Figura 9.24. Estructura del patrón State

Los participantes en este patrón son:

- **Context**: Representa el objeto que debe mantener las instancias de estados concretos y proporciona una interfaz a los clientes.
- **State**: Interfaz común que redirige a cada uno de los estados concretos.
- **ConcreteState**: Implementan el comportamiento de cada uno de los estados del contexto (*Context*) de la aplicación y ejecuta el cambio de estado sobre *Context*.

La ventaja del patrón *State* es la facilidad de abstracción de una máquina de estados en una jerarquía de clases. Al realizarlo de forma estructurada dicha implementación carecería de legibilidad; sin embargo la utilización de este patrón orientado a objetos facilita que cada estado quede abstraído dentro un objeto separado del resto. Por ejemplo, en la figura 9.25 los posibles estados en los que puede encontrarse un ascensor quedan encapsulados dentro de clases independientes derivadas de la clase abstracta *Estado*. Como se verá

más detenidamente en los capítulos quince, dieciséis y diecisiete, la invocación de un método en un estado concreto provocará la actualización de la variable miembro *estado* en *Context* a otro nuevo estado.

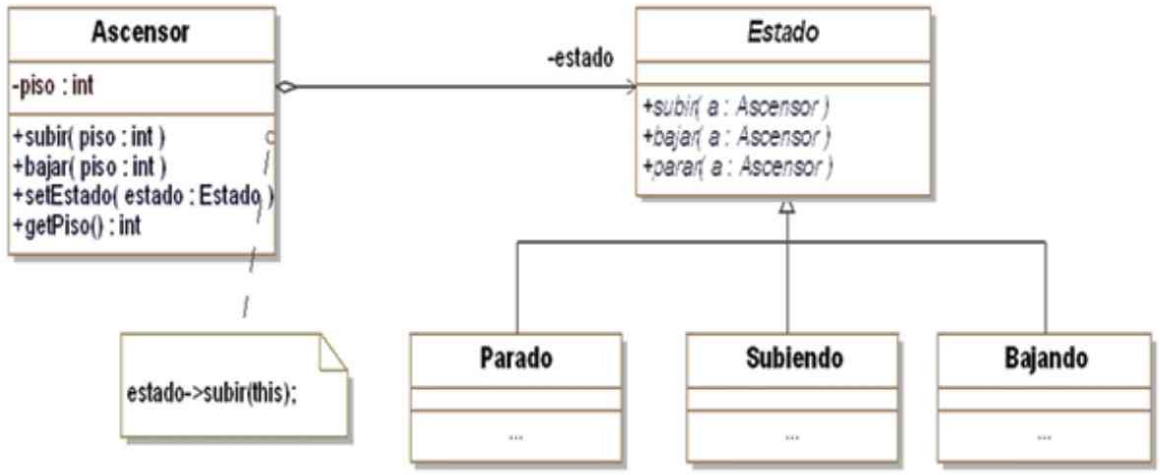


Figura 9.25. Ejemplo del patrón State

En el ejemplo de la figura 9.25 la clase *Ascensor* gestiona todos sus estados mediante objetos de la superclase *Estado*. Al ejecutar uno de los métodos *subir*, *bajar* o *parar* concretos se cambia la referencia del atributo *estado* de *Ascensor* a un nuevo objeto de la clase *Estado* concreto: *Parado*, *Subiendo* o *Bajando*. Es decir, la llamada a las funciones virtuales de la clase base *Estado* de esos objetos hará que se transite a otro objeto de estado continuamente. Para ello es necesario pasar una referencia a *Ascensor* cuando se llama a un método concreto de *subir*, *bajar* o *parar* en las clases que derivan de *Estado*. Estos crearán o tomarán una instancia al objeto derivado de estado al que transita y lo notificará a *Ascensor*.

interpreter

El patrón *Interpreter* (intérprete) permite la abstracción de una gramática libre de contexto con la finalidad de interpretar sentencias de un determinado lenguaje. Por ejemplo si tenemos la siguiente fórmula de lógica proposicional:

$$(p \rightarrow q) \square s$$

Se definirá con la siguiente gramática BNF[28]:

Expresión ::= ExpresionOR | ExpresiónImplica | '(' Expresión ')' | Literal
ExpresiónOR ::= Expresión □ Expresión
ExpresiónImplica ::= Expresión → Expresión
Literal ::= 'p' | 'q' | 's'

En consecuencia el patrón que permite implementar dicha gramática se modelará de la siguiente forma:

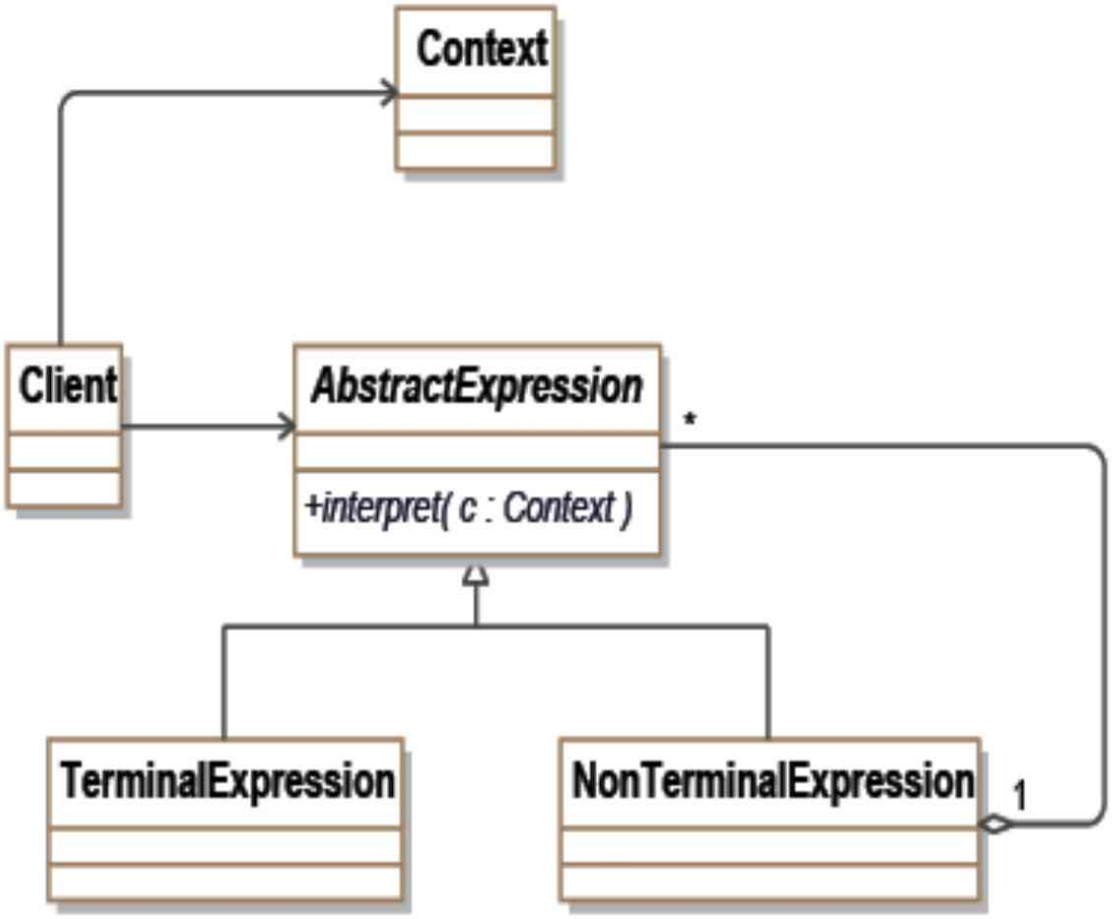


Figura 9.26. Estructura del patrón Interpreter

y donde los participantes del patrón son:

- **AbstractExpression**: Interfaz abstracta que define la operación común a todas las instancias en el árbol de análisis gramatical.
- **TerminalExpression**: Es la clase que representa a los símbolos terminales.
- **NonTerminalExpression**: Clase que implementa los símbolos no terminales de la gramática BNF.
- **Context y Client**: Representan las clases de la lógica de negocio que requieren de la interpretación del lenguaje específico.

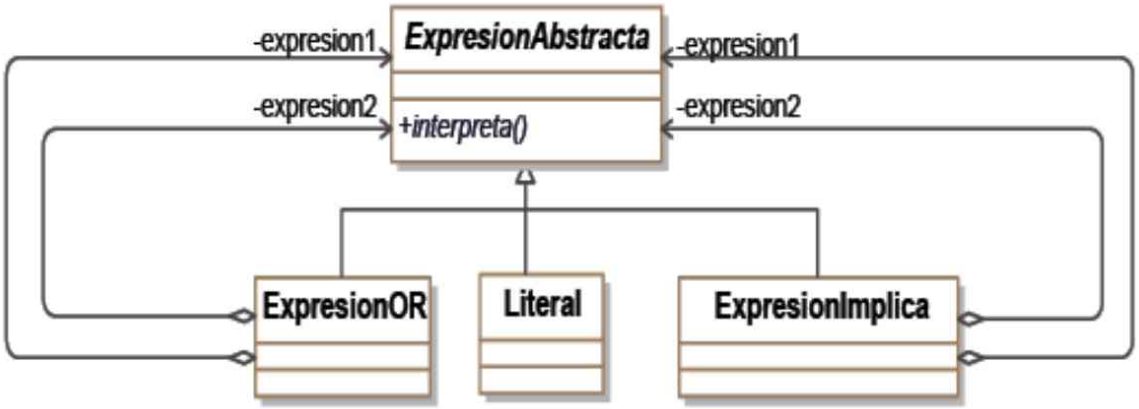


Figura 9.27. Ejemplo del patrón Interpreter

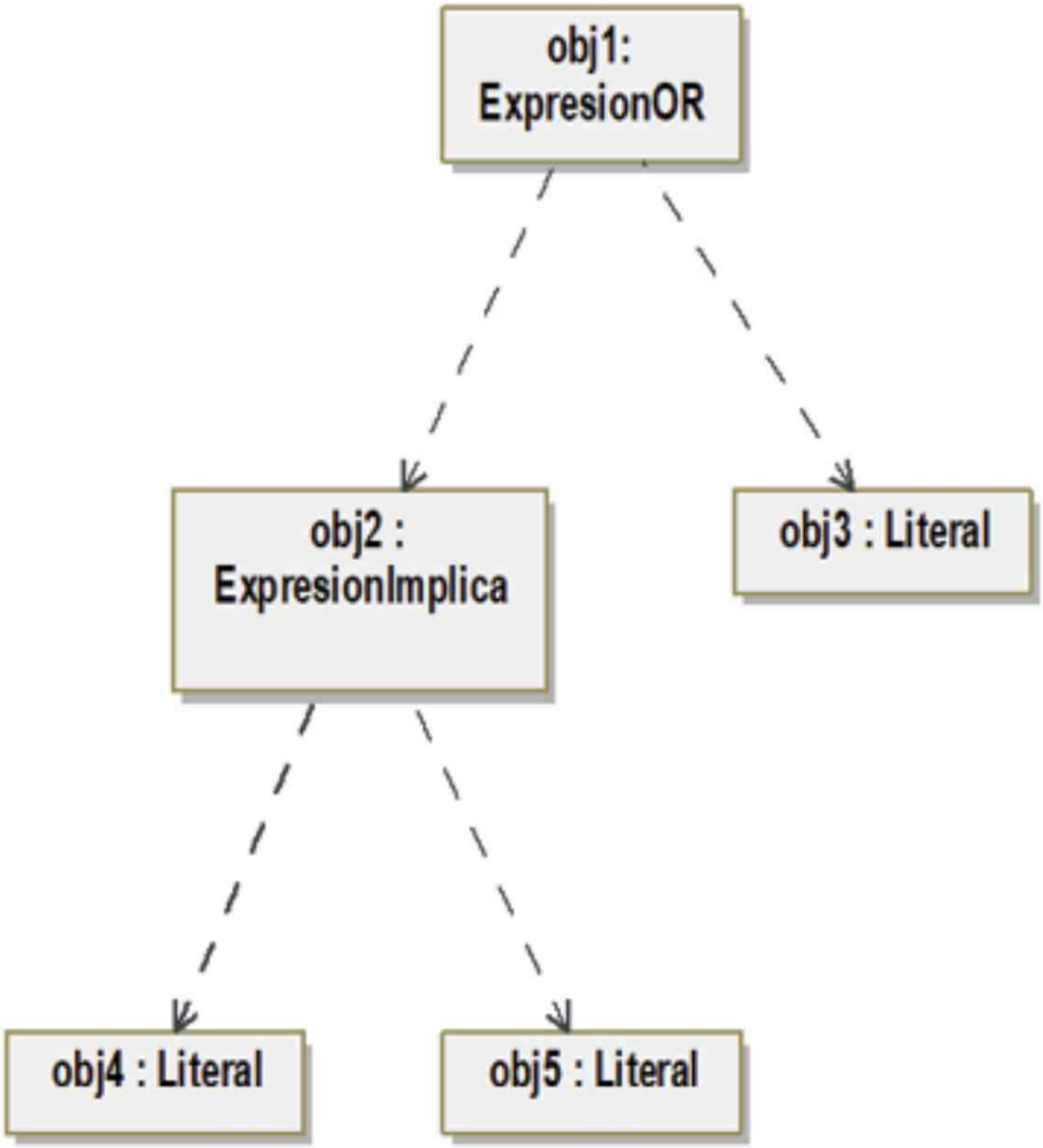


Figura 9.28. Distribución de los objetos en memoria

Se establece una relación jerárquica entre los objetos que representan las expresiones. El recorrido recursivo desde la raíz de esta estructura permite la evaluación e interpretación de la expresión anteriormente expuesta. Es decir, como todo analizador sintáctico que se precie necesita evaluar primeramente los nodos terminales para aplicar las operaciones a las expresiones de menor precedencia. El patrón *Interpreter* se explicará detenidamente en la sección 9.6.1.

caso de estudio: ajedrez

Examinaremos ahora el diagrama de clases del juego de ajedrez para identificar los principales patrones que se han aplicado en la solución del modelo.

Patrón Facade

El patrón *Facade* ha sido utilizado principalmente para realizar clases de frontera que comunican con otros subsistemas. En concreto, este patrón proporciona una gran utilidad para referenciar al subsistema que implementa las clases de *sockets* del tipo TCP asíncrono.

La clase *Fachada_comunicaciones* es la puerta de entrada y salida de los datos que van dirigidos a la red mediante el subsistema *Sockets*. Como se puede apreciar, la clase *Fachada_comunicaciones* implementa la interface *I_comunicacion*, por lo tanto implementa los métodos de envío y recibo de cadenas de texto. La clase *TCPSocket* debe implementar el método virtual *on_receive*() proporcionado por la clase abstracta *AsyncSocket*. Dicho método en *TCPSocket* se encargará posteriormente de indicar al notificador que avise a los objetos "observer" con la información recibida.

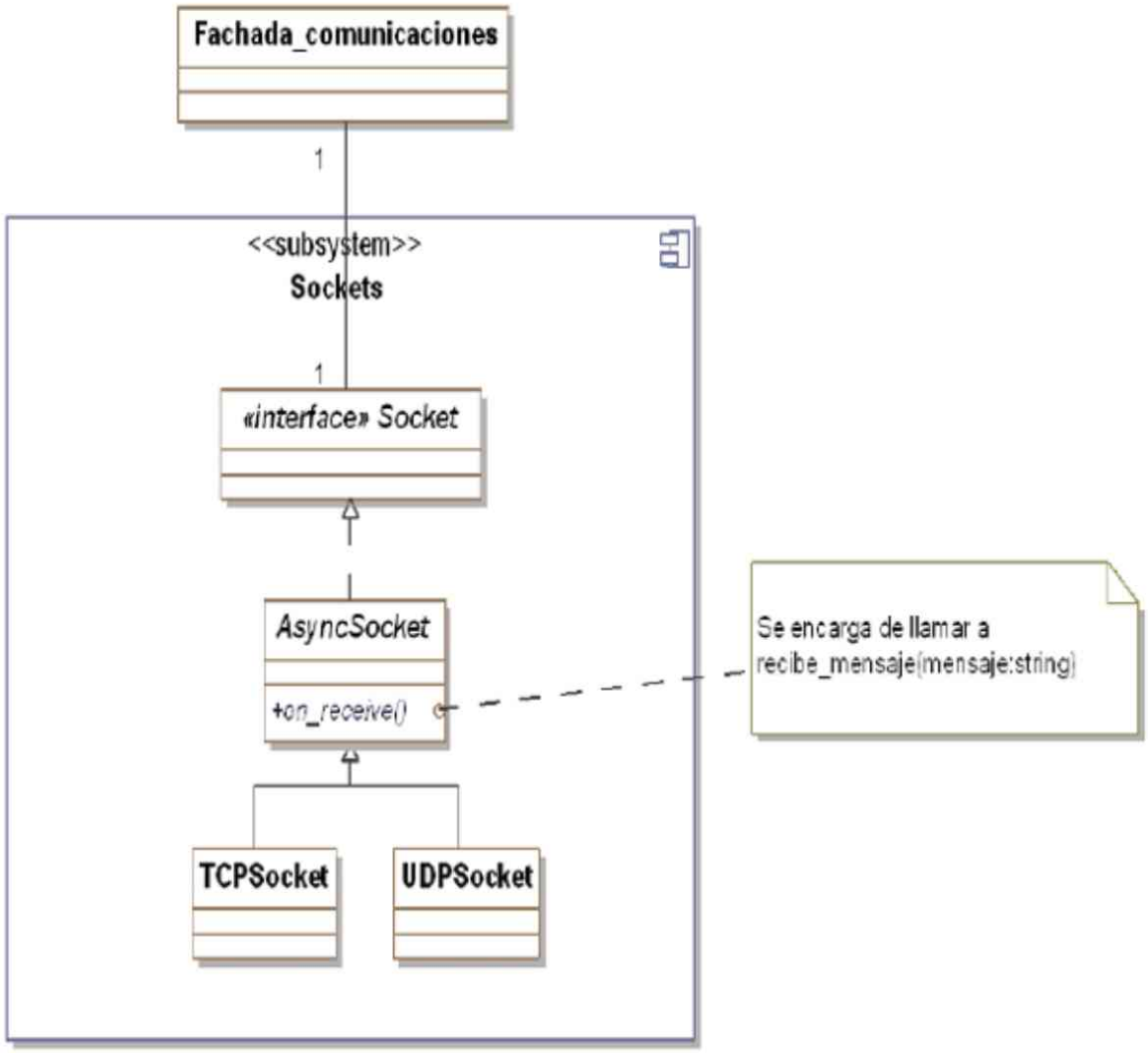


Figura 9.29. Aplicación del patrón Facade

Patrón Observer

También identificamos en el juego de ajedrez al patrón *Observer* que permite conectar el patrón de *Fachada_comunicaciones* con el notificador de eventos del patrón *Observer* con el fin de notificar los mensajes que llegan de la red a los objetos subscritos.

Con el *Observer* es necesario un objeto *Subject* al cual se adjuntan los diferentes observadores (*Observers*). Dichos observadores recibirán las notificaciones de llegada de un evento por la red.

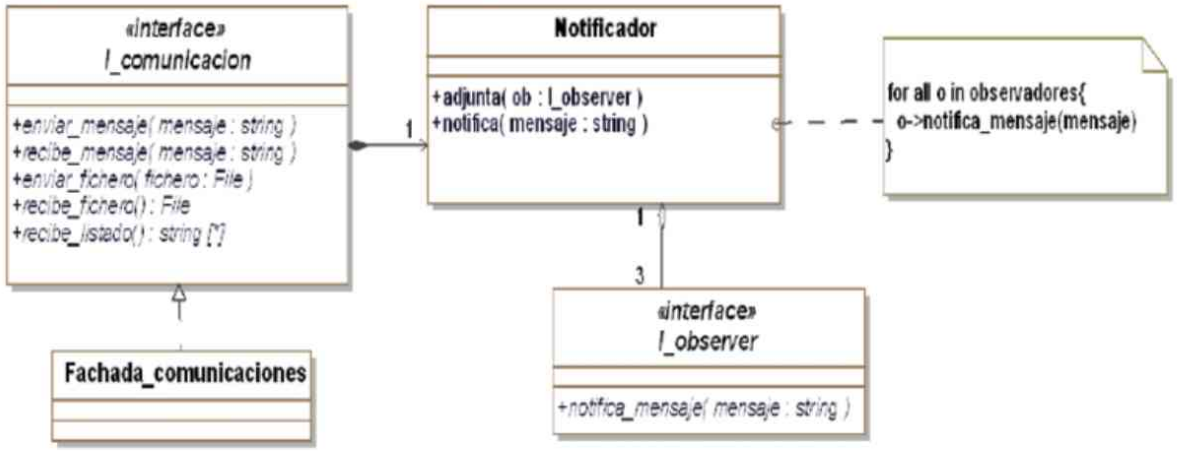


Figura 9.30. Aplicación del patrón Observer

En la figura 9.30 podemos ver la correspondencia entre las clases abstractas y concretas de la estructura del patrón a clases correspondientes al dominio del juego de ajedrez. La correlación es la siguiente:

Clases del patrón	Clases de ajedrez
Subject	Notificador
ConcreteSubject	Notificador
Observer	I_observer
ConcreteObserver	GUI, Control_juego, Tablero

Tabla 9.1. Correspondencia entre clases

El funcionamiento es el siguiente: Cuando un nuevo mensaje llega al objeto *Fachada_comunicaciones (Cliente)* se notifica al objeto *Notificador (Subject)*

mediante la llamada a *notifica(mensaje:String)*. Éste a su vez enviará dicho mensaje a todos los objetos que implementan *I_observer (Observer)* mediante la función virtual pura *notifica_mensaje(mensaje:string)*.

Patrón Strategy

El último patrón que se ha aplicado en el diseño de clases de ajedrez es el *Strategy*. La utilización del patrón de estrategia viene justificada por la necesidad de implementar diferentes algoritmos de cálculo de jugadas en las partidas de ajedrez. Su estructura permite definir una interfaz común que comparten los diversos algoritmos concretos a los que se ha recurrido. El objeto que gestiona la Inteligencia Artificial del juego realizará el papel de contexto cliente del patrón *Strategy*. De esta forma la adaptación del patrón al modelo de clases ha sido el siguiente:

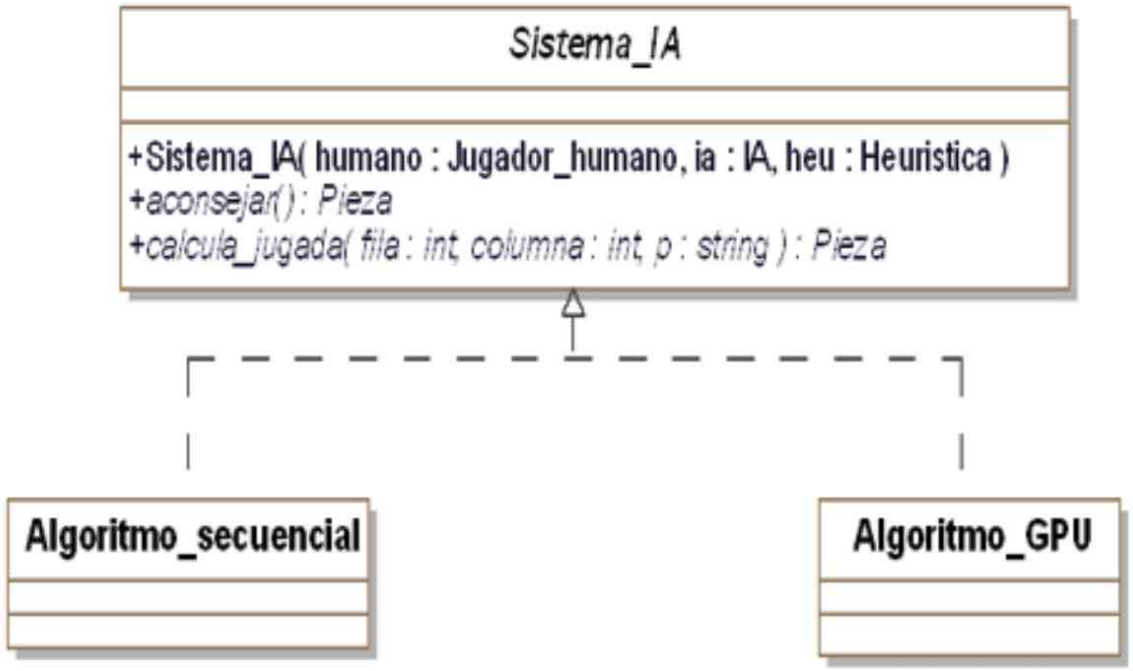


Figura 9.31. Aplicación del patrón Strategy

En la clase abstracta *Sistema_IA* de la figura 9.31 se representa la operación virtual *calcula_jugada()* que será redirigida al objeto concreto dependiendo de la estrategia seleccionada por el usuario. De este modo si el usuario posee un dispositivo gráfico de alto rendimiento, la operación de cálculo será delegada al objeto *Algoritmo_GPU*, en caso contrario el objeto *Algoritmo_secuencial* se encargará de procesar las operaciones en la CPU del sistema.

caso de estudio: mercurial

Al igual que en el caso del ajedrez y en otras muchas aplicaciones, el sistema CVS de *Mercurial* también es susceptible de aplicación de patrones de diseño. En general, las aplicaciones cliente-servidor se prestan con frecuencia al uso de patrones GoF[29].

Patrón Facade e Interpreter

El patrón *Facade* nos es útil en este contexto para establecer un punto de conexión con el sistema de interpretación de comandos textuales. De esta forma se aplican dos patrones simultáneamente: *Facade* como frontera del subsistema e *Interpreter* para el análisis sintáctico de cadenas alfanuméricas de los comandos CVS.

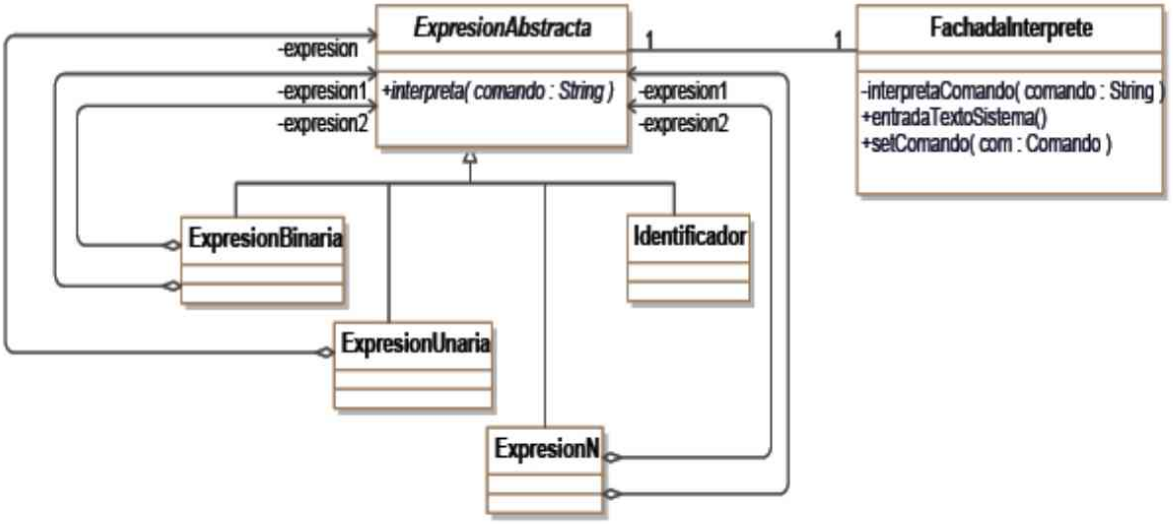


Figura 9.32. Aplicación conjunta del patrón Facade e Interpreter

En la parte izquierda de la figura 9.32 se implementa el patrón *Interpreter* con el fin de analizar sintácticamente órdenes de texto pasadas a través del método *interpreta(comando:String)*. Posteriormente se analizará la cadena de texto para determinar el tipo de orden (objeto *Comando*). Una vez que se ha determinado el comando se procede a analizar recursivamente la secuencia de identificadores según el tipo de operador de expresión. Finalmente, la expresión base devuelve el objeto *Comando* a la fachada del intérprete mediante *setComando()* que se encargará de transmitir el resultado al resto de la aplicación.

Patrón Composite

El patrón *Composite* es junto con *Observer* uno de los patrones más fácilmente inidentificables en esta aplicación. En el caso de *Mercurial* ha sido inmediata la reutilización de este patrón para la jerarquía del sistema de archivos.

Puesto que un sistema de archivos se compone de una jerarquía de directorios con ficheros que contienen a su vez más directorios y ficheros, es consecuentemente una estructura idónea para la aplicación del patrón Composite.

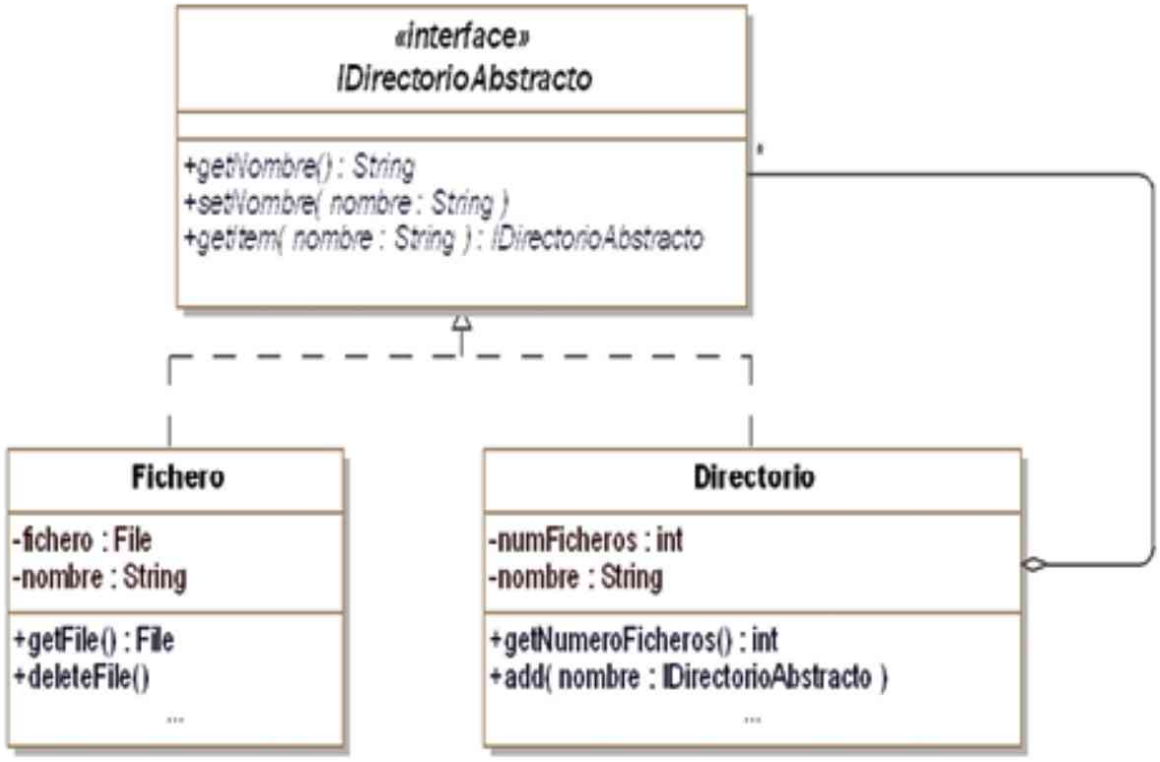


Figura 9.33. Aplicación del patrón Composite

El funcionamiento del patrón Composite en este ejemplo se explica gracias a la operación *getItem(nombre:String):IDirectorioAbstracto*. Este método devuelve un objeto *Directorio* o *Fichero* a partir del identificador proporcionado. De esta forma si se busca un ítem determinado dentro de un objeto *Directorio*, dicho objeto *Directorio* buscará en su *array* o lista de objetos del tipo *IDirectorioAbstracto* una correspondencia con el ítem en cuestión. Una vez obtenido el ítem se procederá a actuar de la misma forma con el objeto

retornado mediante recursividad.

caso de estudio: servicio de cifrado remoto

Procedemos a analizar ahora el caso de la aplicación de patrones GoF a la aplicación distribuida cliente/servidor del servicio de cifrado remoto. Como ha ocurrido en el caso del juego de ajedrez y en la aplicación Mercurial, aquí también es adecuada la utilización de patrones de diseño en la mayoría de las situaciones.

Lado cliente

patrón facade

En la aplicación de cifrado remoto utilizaremos el patrón *Facade* tanto en el cliente como en el servidor con el fin de suplir una frontera entre el subsistema de implementación de los *sockets* y la lógica de negocio del subsistema de cifrado/descifrado.

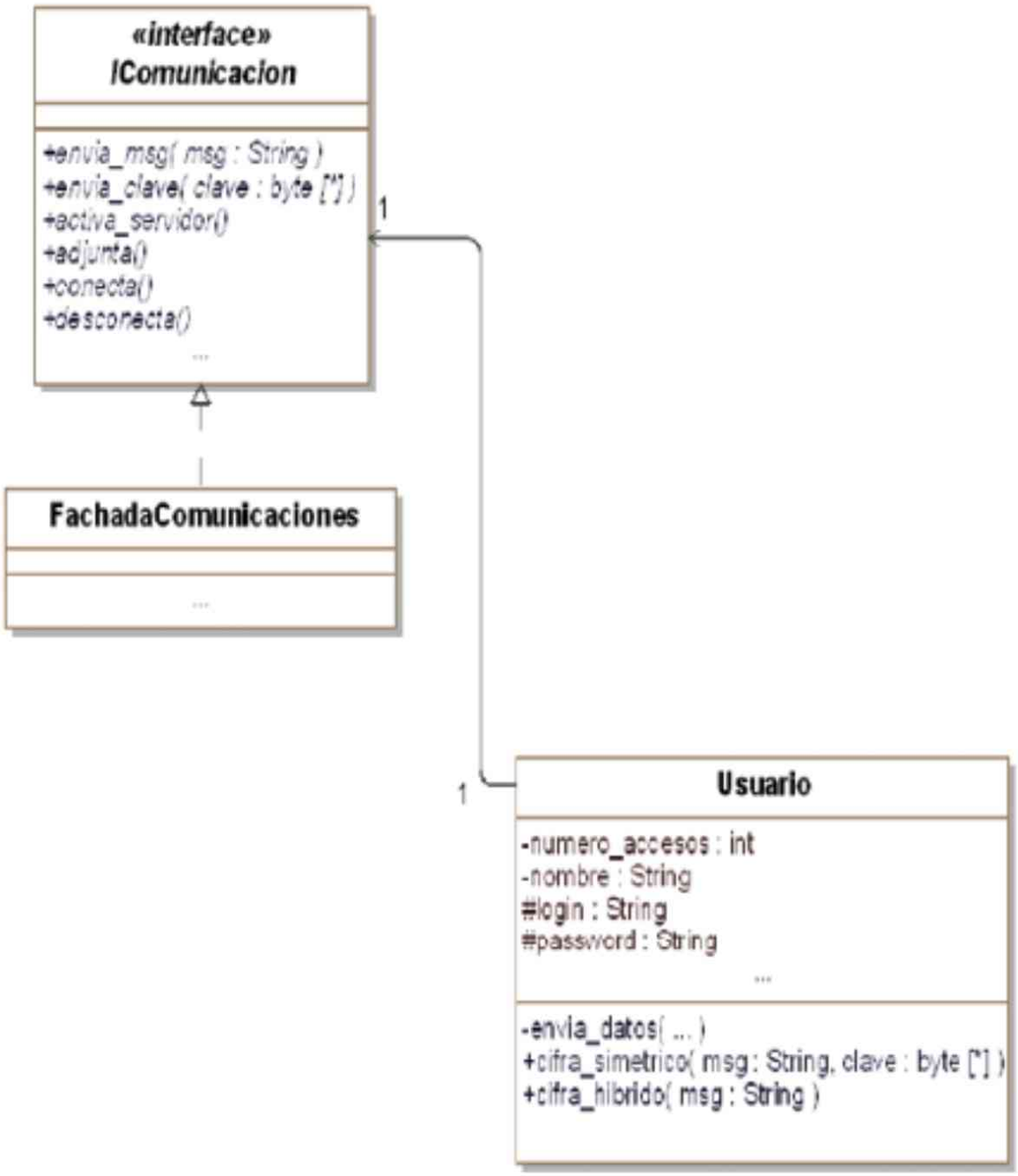


Figura 9.34. Aplicación del patrón Facade

La separación de funcionalidades entre la fachada y el subsistema de cifrado/descifrado permite mejorar la cohesión y disminuir el acoplamiento general del sistema.

patrón strategy

La reutilización de este patrón en el contexto del servicio de cifrado remoto se ve demostrada por la necesidad de invocar diferentes tipos de algoritmos de cifrado según las preferencias del usuario.

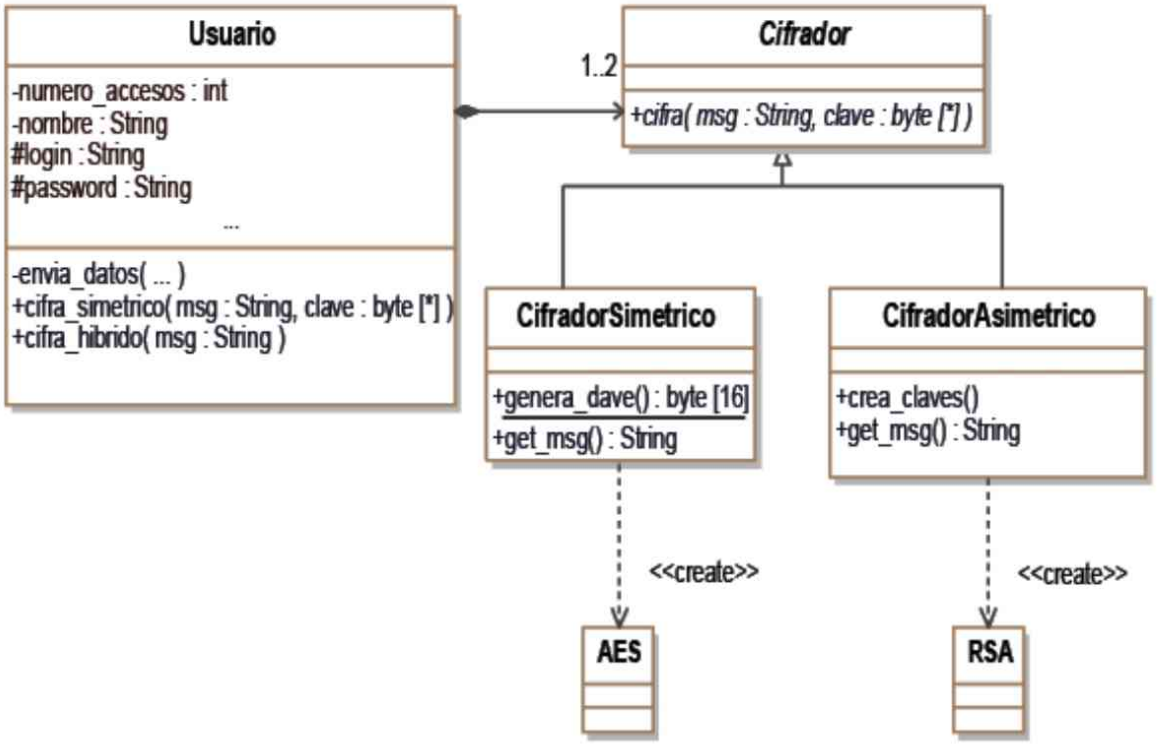


Figura 9.35. Aplicación del patrón Strategy en el cliente

Lado servidor

patrón singleton

El requerimiento de utilización del patrón *Singleton* viene justificado por la arquitectura cliente/servidor, ya que en este caso debe existir una única instancia de la aplicación servidor en el sistema para evitar interferencias en el socket oyente. Mientras que las aplicaciones cliente pueden instanciarse de forma múltiple en un sistema terminal, los servidores deben permanecer instanciados una sola vez en la computadora de servicio, tal como se ejemplificó en la sección 2.8.

Gracias al campo estático *singleton* del tipo de clase *MenuServidor* es posible tener un contador de instancias (figura 9.36).

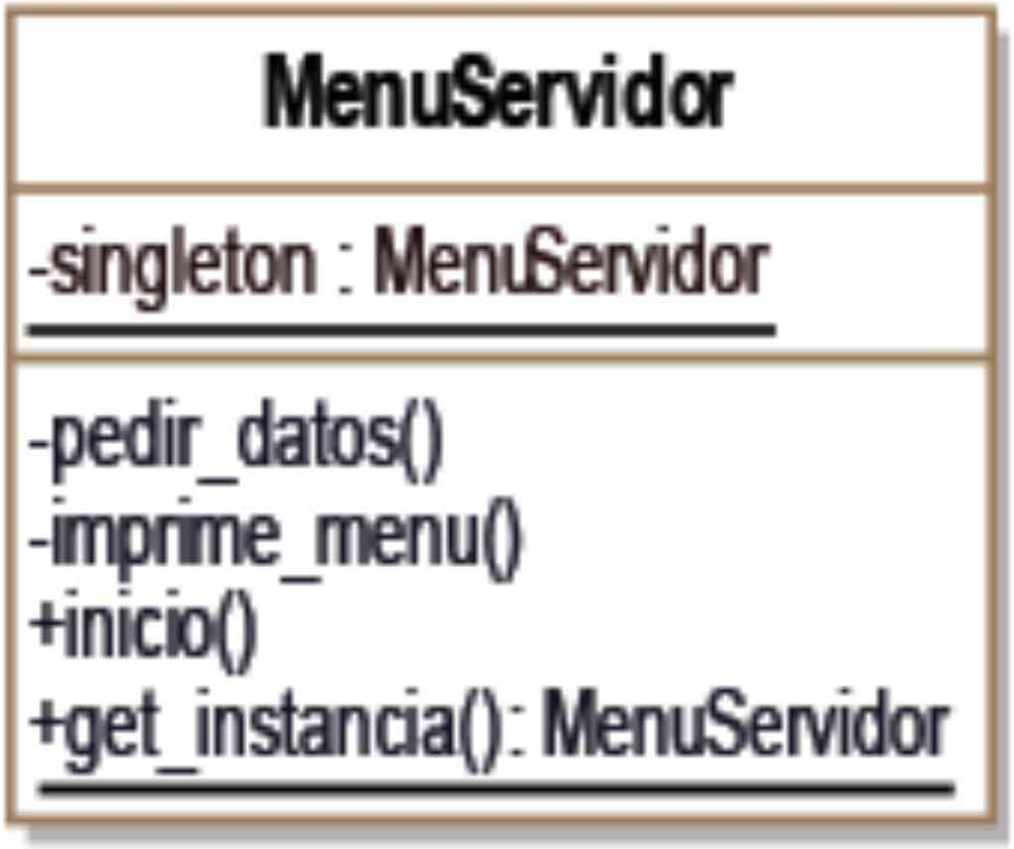


Figura 9.36. Una única instancia del servidor con el patrón Singleton

patrón strategy

Al igual que sucedió en el juego de ajedrez para la selección de algoritmo de IA y en el lado cliente del servicio de cifrado remoto, aquí también debemos recurrir al patrón de estrategia con el propósito de someter a prueba el mensaje cifrado con los dos algoritmos de desencriptación establecidos.

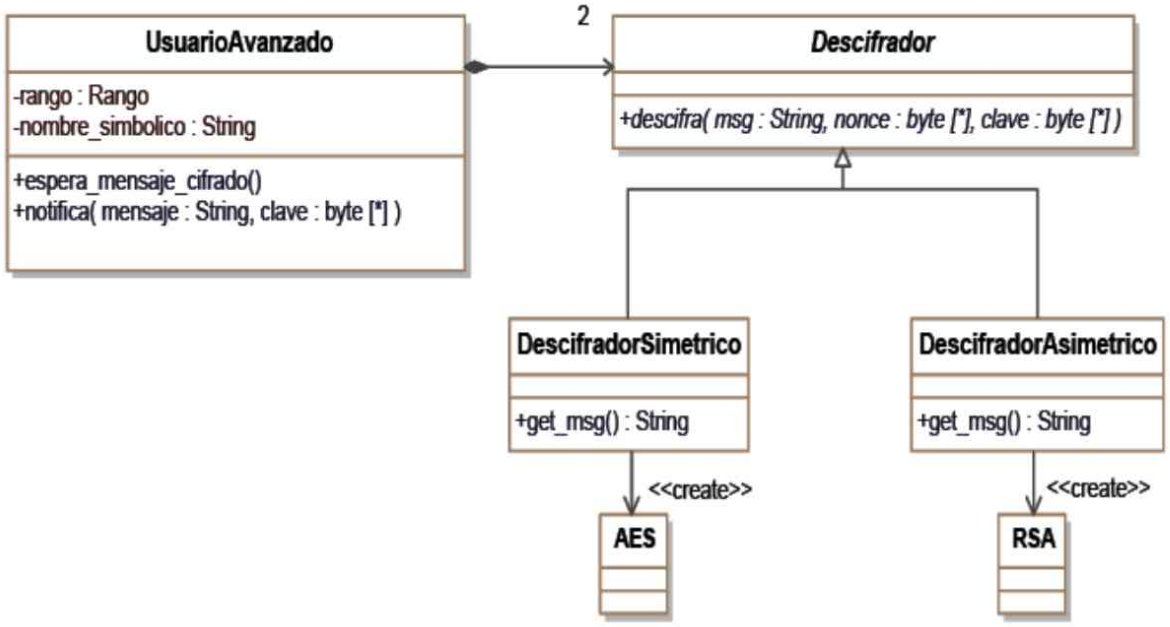


Figura 9.37. Aplicación del patrón Strategy en el servidor

patrón observer

El patrón *Observer* se reutiliza de la misma forma que en los contextos del juego de ajedrez y la aplicación CVS Mercurial:

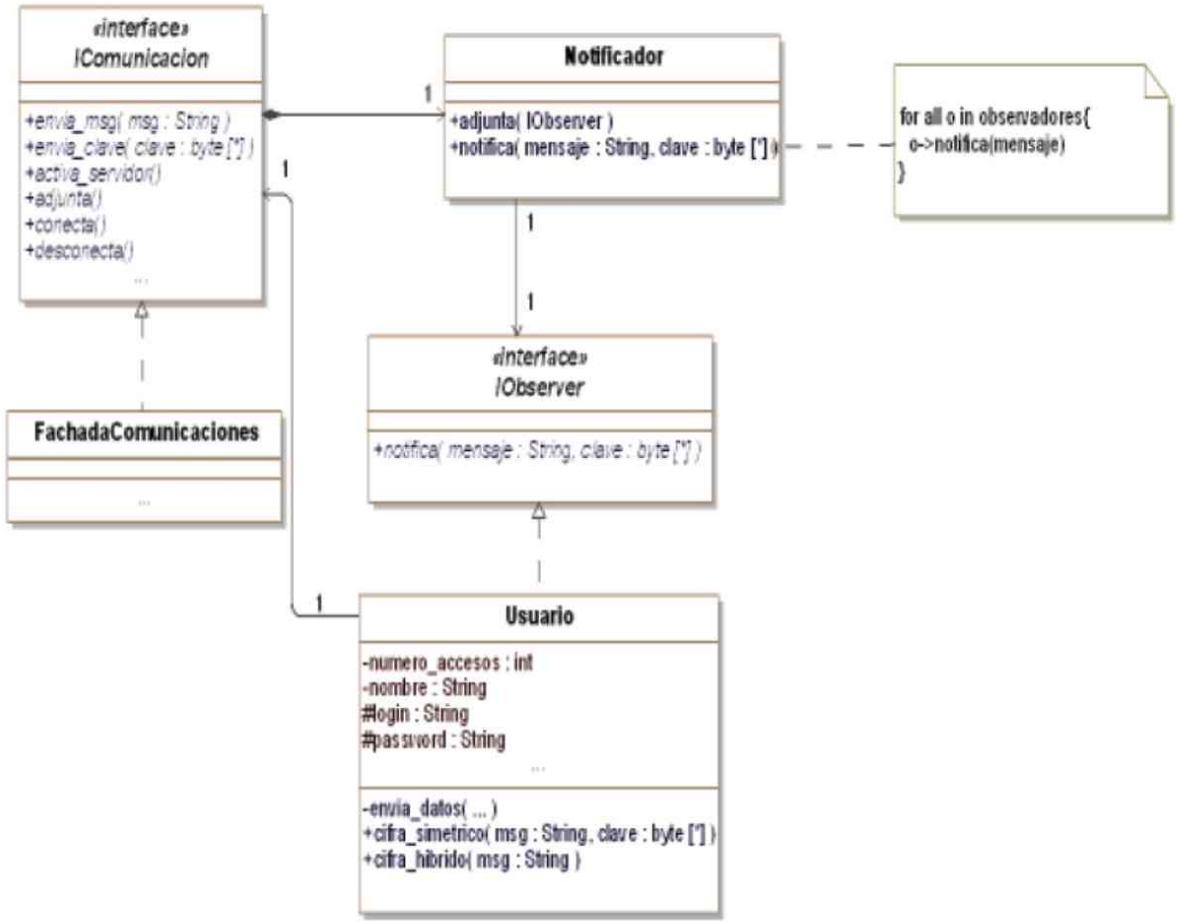


Figura 9.38. Aplicación del patrón Observer en el servidor

Clases del patrón	Clases del servidor
Subject	Notificador
ConcreteSubject	Notificador
Observer	IObserver
ConcreteObserver	UsuarioAvanzado

Tabla 9.2. Correspondencia entre clases

Observamos en la tabla 9.2 la correlación entre las entidades del patrón y las clases del servidor de cifrado remoto. Si bien es el *Usuario* el que implementa la interfaz *Observer*, es en realidad la clase *UsuarioAvanzado* la que recibe las notificaciones procedentes de la fachada de comunicaciones a través de herencia.

27 Describió en los libros « The Timeless Way of Building, (1979) (El modo intemporal de construir)» y «A Pattern Language, (1977)» como unas referencias formales y teóricas de buenas prácticas para el diseño en cualquier área de conocimiento.

28 La notación Backus–Naur Form permite definir gramáticas libres de contexto.

29 Gang of Four (GoF) es el nombre con el que se conoce en el argot de la orientación a objetos a los autores del libro Design Pattern [Gamma95].

buenas prácticas de programación

«La senda de la virtud es muy estrecha y el camino del vicio ancho y espacioso».

(Miguel de Cervantes:El ingenioso hidalgo don Quijote de la Mancha, Parte 2, Capítulo 6).

Como buenas prácticas entenderemos las acciones realizadas durante las fases del ciclo vida del desarrollo de software para mejorar la calidad en todos sus aspectos formales con el fin de entregar un producto bien realizado. Los patrones de diseño vistos en el capítulo anterior son también modelos de buenas prácticas en la construcción de software basados en la experiencia de los desarrolladores. En este capítulo profundizaremos más aún si cabe en dichas prácticas mediante la aplicación de patrones GRASP en proyectos de nuestro dominio.

patrones de diseño grasp

Los patrones GRASP (*General Responsibility Assignment Software Patterns*) conocidos como patrones generales de software para asignación de responsabilidades, son una serie de recomendaciones para el diseño de objetos de una forma precisa y sin ambigüedades. Las explicaciones aquí proporcionadas sobre los patrones GRASP están basadas en el texto de *Craig Larman* "UML y Patrones" [Larman02], pero con diferentes ejemplos.

¿Qué es una responsabilidad?

Para entender correctamente cómo se aplican los patrones GRASP es imprescindible conocer el concepto de responsabilidad. Podemos asociar la responsabilidad con una operación que debe realizar un clasificador en relación a su comportamiento, como pueden ser crear un objeto, enviar un mensaje a otro/s objeto/s para conocer su información o iniciar una acción, etc. Dichas responsabilidades se asignan a las clases durante el diseño de objetos.

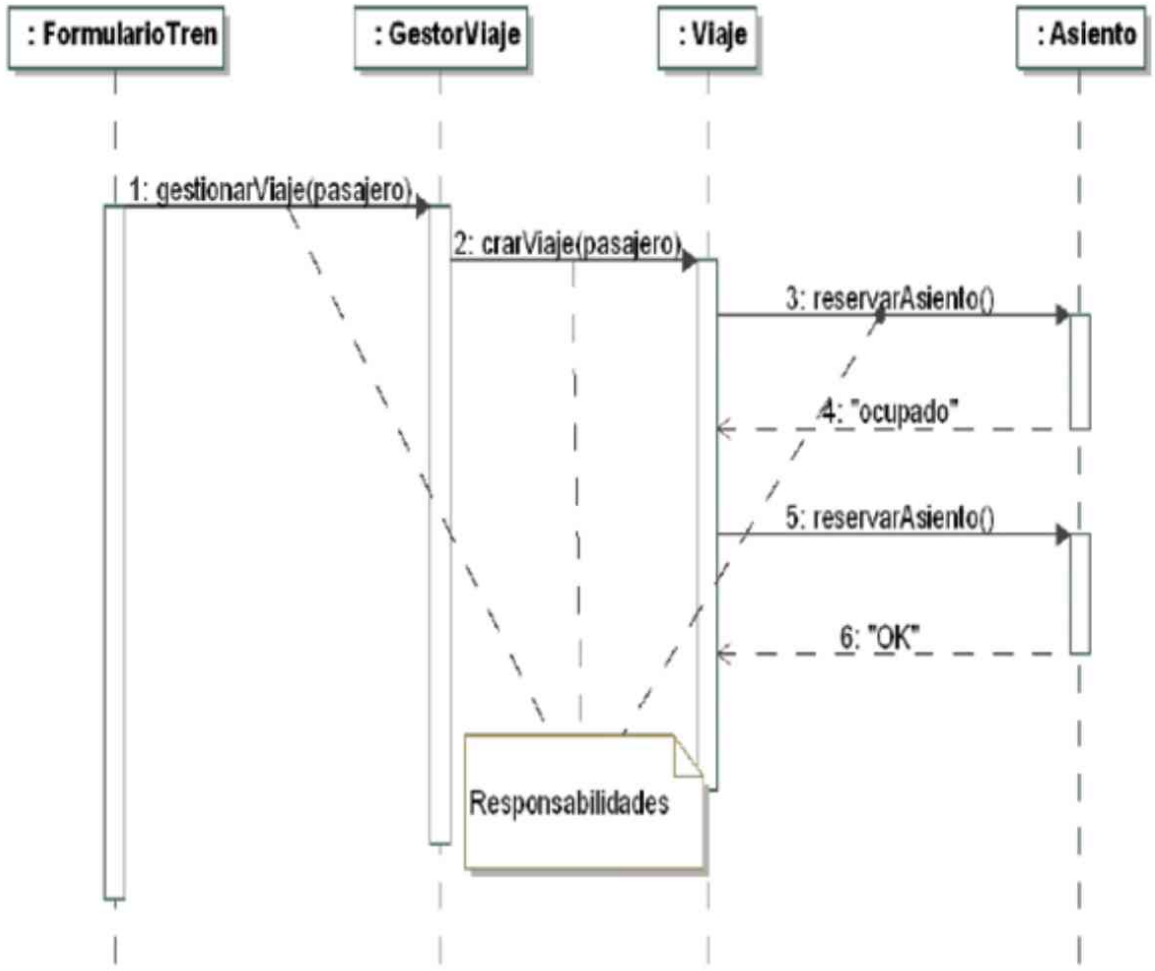


Figura 10.1. Ejemplo de responsabilidades

En la figura 10.1 podemos apreciar las dos responsabilidades que pertenecen a los objetos *Viaje y Asiento* con las operaciones *crearViaje()* y *reservarAsiento()* respectivamente.

Basándonos en el concepto de responsabilidad, procederemos al estudio de los nueve patrones GRASP más conocidos.

Experto en información

La función principal de este patrón es asignar responsabilidades a la clase que posee una información requerida para cumplimentar la responsabilidad. En el ejemplo de la figura 10.1, ¿quién debería ser el responsable de conocer el estado de un asiento? Si miramos en su diagrama de clases del modelo de diseño, veremos la información interna que pueda proporcionarnos la respuesta.

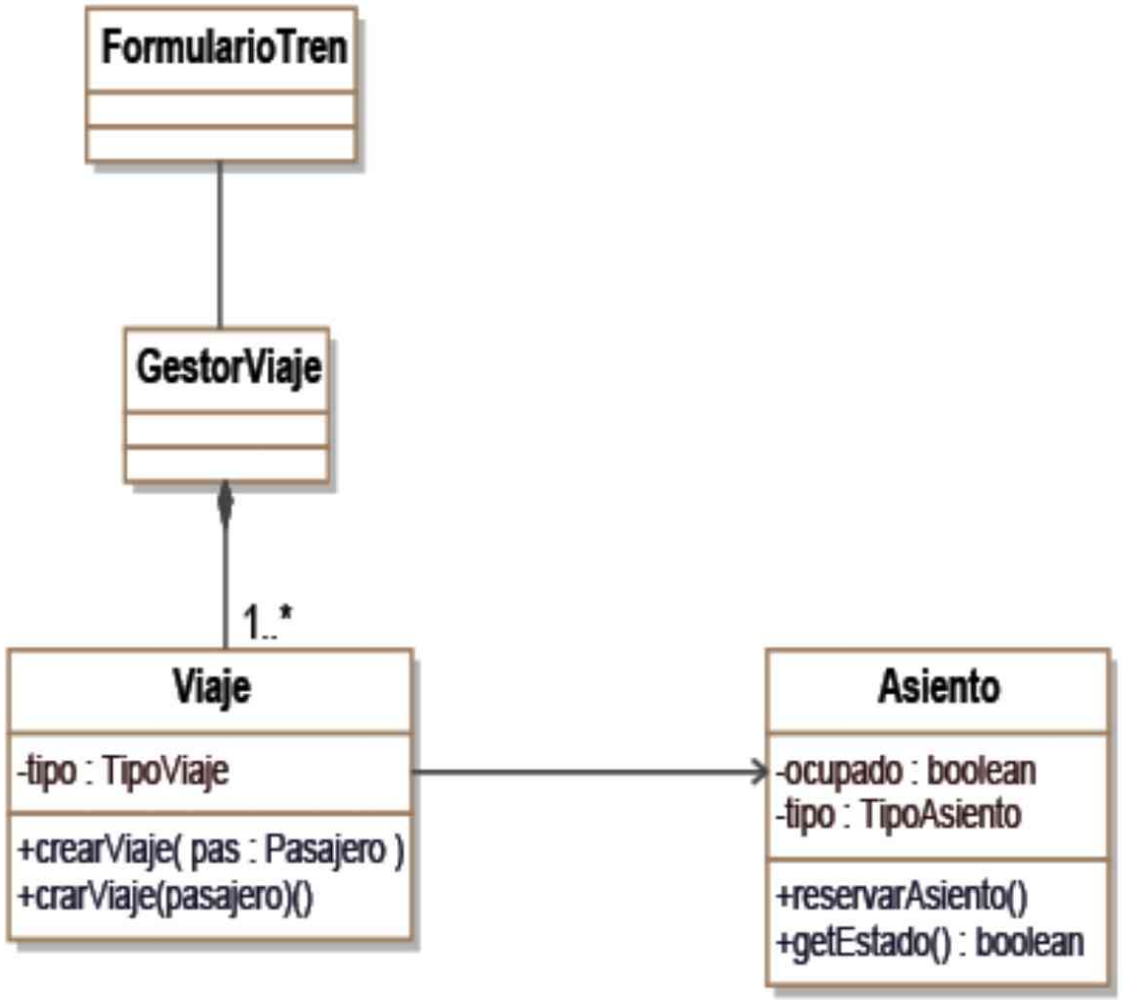


Figura 10.2. Diagrama de clases para la reserva de viaje en tren

Puesto que es la clase *Asiento* la que alberga la información relacionada al estado libre/ocupado del mismo, es necesario asignarle la responsabilidad *reservarAsiento()* a dicha clase.

Entre las ventajas asociadas al patrón *Experto en información* cabe destacar que se mejora el encapsulamiento de la información y por lo tanto mejora la cohesión del sistema disminuyendo el acoplamiento. Veremos estos conceptos más detalladamente en los apartados 10.1.4 y 10.1.5.

Creador

La característica que define a este patrón es la de asignar la responsabilidad a la clase B de crear una instancia de una clase A cuando ocurre alguno de los siguientes escenarios:

- B *agrega* objetos de A.
- B *contiene* objetos de A.
- B *registra* instancias de objetos de A.
- B *utiliza más estrechamente* objetos de A.
- B *tiene los datos de inicialización* que se pasarán a un objeto de A cuando sea creado (por tanto, B es un experto con respecto a la creación de A).

En el ejemplo de la figura 10.2, la clase *GestorViaje* es la clase que contiene la lista de viajes. Por tanto, asignaremos la responsabilidad de crear objetos *Viaje* a dicha clase, tal como vimos en el diagrama de secuencias de la figura 10.1.

Una fórmula para encontrar las clases creadoras es buscando las clases que contienen los datos de creación. Sin embargo, cuando la creación alcanza una gran complejidad es preferible recurrir a los patrones vistos en la sección 9.4.2. La ventaja de utilizar este patrón es la disminución del acoplamiento en el sistema.

Bajo acoplamiento

El concepto **acoplamiento** es de vital importancia en el diseño orientado a objetos y está relacionado con el grado de conexión o dependencia con otros elementos del sistema. Un elemento con acoplamiento débil tiene a conocer o a conectarse con pocos elementos; mientras que un elemento con acoplamiento alto tiende a relacionarse o depender de muchas otras clases, siendo no deseable para el diseño.

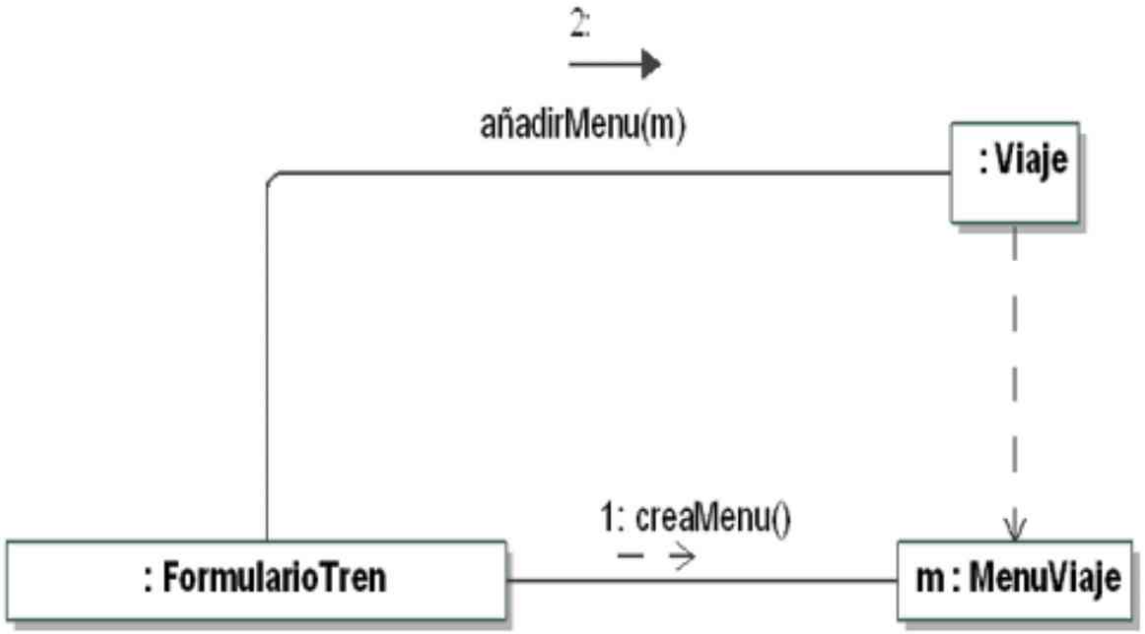


Figura 10.3. Diseño con alto acoplamiento (incorrecto)

Supongamos el caso de seleccionar una serie de menús gastronómicos para el viaje que ofrece la compañía. El diseño del diagrama de comunicación de la figura 10.3 es un diseño erróneo que presenta un alto grado de dependencia entre los objetos *Viaje* y los objetos *MenuViaje*. Además, no realiza una separación entra la vista y el modelo. Para realizarlo de forma correcta debemos recurrir al modelo del diagrama 10.4, en el que se desacopla la selección de menú del formulario de la compañía y se integra en el del viaje.

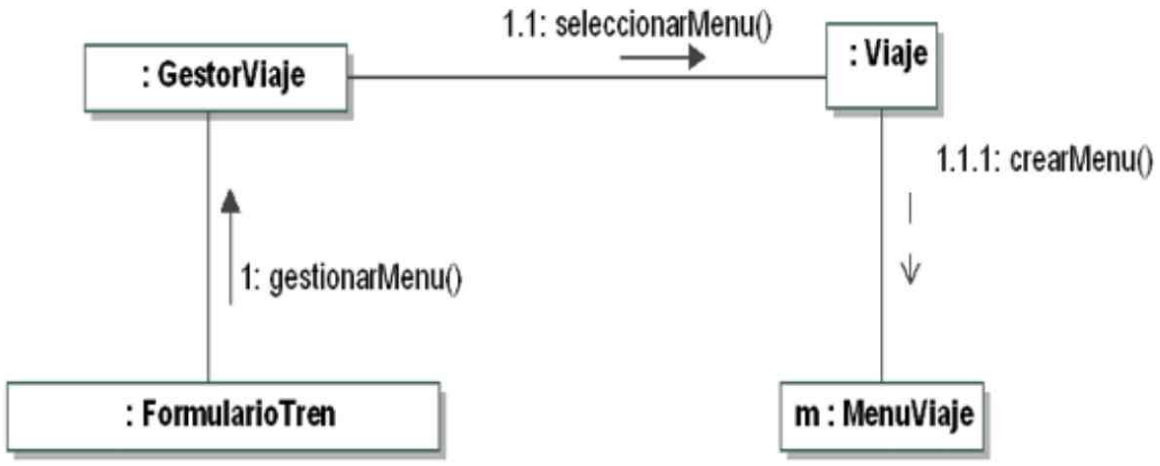


Figura 10.4. Diseño con bajo acoplamiento (correcto)

Utilizar el patrón de bajo acoplamiento será siempre útil en las decisiones de diseño, aunque este criterio no se cumple cuando el acoplamiento es nulo, ya que no es deseable en una arquitectura orientada a objetos donde los elementos suelen comunicarse mediante el paso de mensajes.

Entre las ventajas de este patrón caben destacar que los cambios que se realizan al componente no afectarán a otros componentes y que fomenta la reutilización.

Alta cohesión

Un diseño altamente cohesionado de una clase, sistema o subsistema es el que tiende a realizar la menor cantidad de trabajo y sus responsabilidades están altamente relacionadas, es decir, es la que desempeña una función bien definida dentro del dominio del problema. Una clase con poca cohesión tiende a realizar muchas funciones que no guardan relación entre ellas, mientras que una con alta cohesión suele tener un número relativamente pequeño de métodos. Es también conocida la estrecha relación que guardan la cohesión y el acoplamiento, ya que cuando aumenta una, disminuye la otra, y viceversa.

El ejemplo de la figura 10.3 puede servirnos de ejemplo, ya que reducimos el trabajo de *FormularioTren* a las tareas que le corresponden como formulario de registro, mientras que delegamos la responsabilidad de seleccionar menú gastronómico a cada viaje que realiza el pasajero (figura 10.4).

Las ventajas de este patrón comprenden la facilidad de mantenimiento, la claridad del código y la reutilización del mismo.

Controlador

Este patrón asigna la responsabilidad a una clase encargada de redirigir un mansaje proveniente de un evento generado por un actor externo del sistema cuando ésta representa un sistema, subsistema o dispositivo que hace de *controlador de fachada*. También cuando la clase representa un caso de uso relacionado con un evento del sistema, denominado *controlador de sesión o de caso de uso*. Este patrón está estrechamente relacionado con el patrón *Fachada* visto en la sección 9.4.3.2 y el patrón *Command* 9.4.4.1.

No modela un objeto de la capa de interfaz de usuario, sino que funciona como coordinador que delega los eventos provenientes de la interfaz de usuario a la capa del dominio. Dicha clase controladora guarda una relación con los *objetos controlador* vistos en el capítulo tres sobre los Diagramas de Robustez.

A modo de ejemplo se propone un escenario de realización de una encuesta por un usuario y recogida en un formulario Web (figura 10.5). Una vez que el usuario acepta la información introducida se produce un evento sobre el controlador que hará que redirija la información proporcionada al objeto que gestiona el análisis estadístico correspondiente.

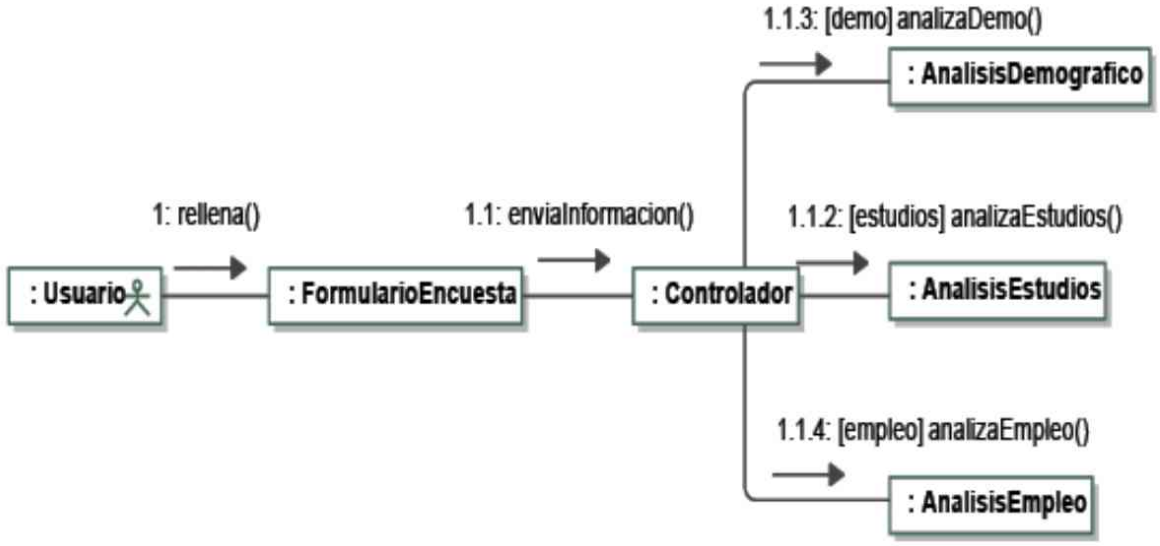


Figura 10.5. Ejemplo de clase controladora

Un aspecto importante en el diseño del controlador es tener en cuenta que debe delegar el trabajo a otras clases y no realizarlo todo ella.
El aumento de la reutilización y la capacidad de asegurar que la lógica de negocio no se realiza en la capa de la interfaz de usuario son sus principales ventajas. En resumen, una clase controladora bien diseñada se caracteriza por ser altamente cohesiva.

Polimorfismo

Utilizaremos este patrón cuando existan varios comportamientos relacionados que dependan del tipo, por lo tanto, debemos evitar hacer comprobaciones acerca del tipo mediante sentencias condicionales (*if, switch,* etc.) y delegar dicha responsabilidad al mecanismo del polimorfismo proporcionado en los lenguajes de programación orientados a objetos.

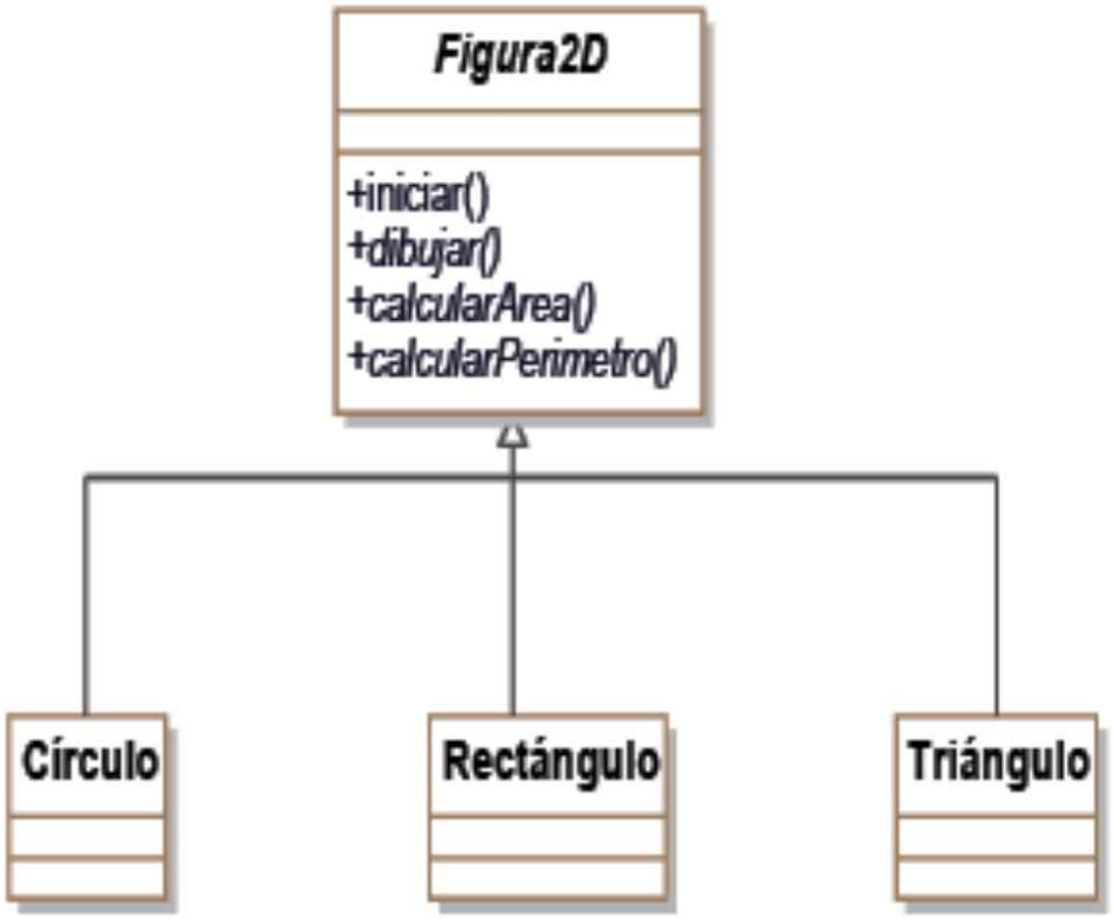


Figura 10.6. Ejemplo de polimorfismo

En la figura 10.6 se puede apreciar un caso de utilización del polimorfismo desde la clase *Figura2D* hacia las tres clases hijas que heredan de ella. Las operaciones *dibujar(), calcularArea() y calcularPerimetro()* son todas ellas virtuales puras y se redirigirán a su operación concreta en la subclase de *Figura2D* que le corresponda.

Los beneficios de este patrón, como veremos más adelante en el patrón Variaciones Protegidas, es la posibilidad de añadir nuevas extensiones para futuros requerimientos, aumentando así la flexibilidad del diseño sin necesidad de afectar a los clientes.

Fabricación Pura

En este caso se trata de asignar un conjunto de responsabilidades homogéneas a una clase altamente cohesiva y que no esté relacionada con el dominio del problema. Un ejemplo podría ser una clase que gestiona la conexión con el *driver* específico de una base de datos, o un dispositivo concreto.



Figura 10.7. Utilización del patrón Fabricación Pura para un formulario de envío SMTP

Obviamente, las ventajas de este patrón son la alta cohesión de un conjunto

de funciones estrechamente relacionadas y el aumento de la reutilización en otras aplicaciones y dominios.

Indirección

El patrón *Indirección* funciona a modo de intermediario, fachada o puente entre componentes o servicios de forma que no exista un acoplamiento directo entre ellos.

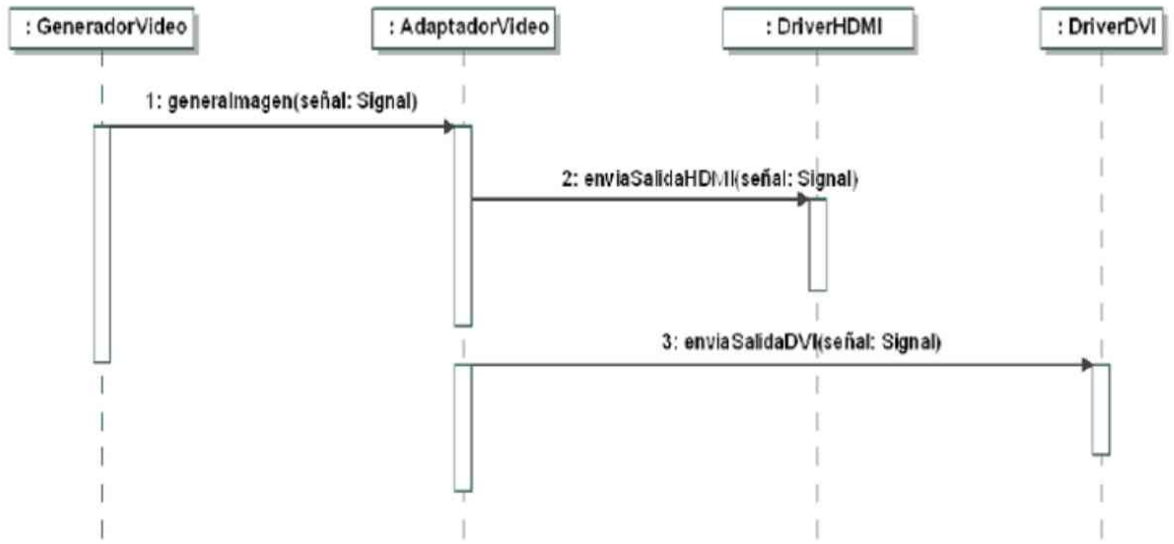


Figura 10.8. Ejemplo de patrón Indirección

En el ejemplo de la imagen 10.8, la clase *AdaptadorVideo* realiza el desacoplamiento entre la lógica de negocio (*GeneradorVideo*) y el manejador del dispositivo externo (*DriverHDMI y DriverDVI*).

La ventaja principal de este patrón es la capacidad para reducir el acoplamiento entre componentes.

Variaciones Protegidas (VP)

El objetivo de este patrón es detectar puntos críticos de inestabilidad y asignar responsabilidades mediante una interfaz estable en torno a ellos. La idea principal consiste en crear sistemas y subsistemas de forma que cualquier variación en estos no afecte de forma negativa a otros componentes del software.

Por ejemplo, en la figura 10.9 se presenta un caso de ampliación de un gestor de métodos de pago con un nuevo tipo de método, como podría ser PayPal o las criptomonedas. En este tipo de situaciones conviene ser prudente y tener en cuenta este patrón.

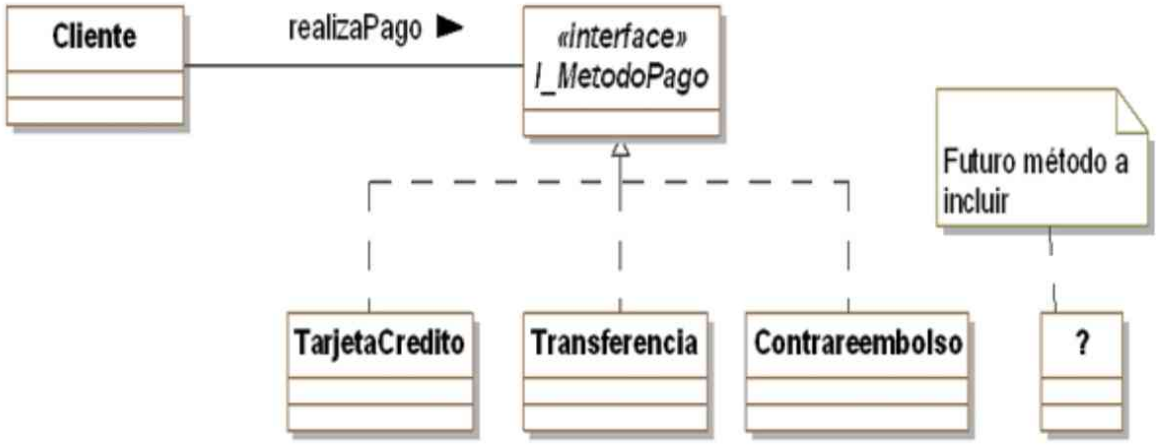


Figura 10.9. Un caso de utilización del patrón de Variaciones protegidas

Este patrón fue propuesto por primera vez en 1996 por *A. Cockburn*, aunque lleva utilizándose durante décadas con otros nombres.

Los mecanismos motivados por VP son:

- Encapsulación, interfaces, polimorfismo, indirección, etc.
- Diseños dirigidos por datos: ficheros de configuración, metadatos, hojas de estilo, ficheros de propiedades, etc.
- Búsqueda de servicios: JNDI, Jini de Java, TraderService, UDDI (véase sección 9.3.2.3), etc.
- Principio de sustitución de *Liskov* (PSL): En dicho principio el código

(métodos, clases, ...) que hace referencia a un tipo T (interfaz o superclase abstracta) debería trabajar con cualquier implementación o subclase de T que lo sustituya, tal como se implementarían los ejemplos 10.6 y 10.9.

- *Ley de Demetero No Hable con Extraños*: Básicamente consiste en evitar diseñar sistemas o subsistemas que impliquen largos caminos de llamadas o navegación por atributos y métodos para realizar una determinada operación. Por ejemplo, no debe realizar lo siguiente:

```
// Sentencia frágil y punto de inestabilidad
Vivienda vivienda =
ciudad.getBarrio(x).getManzana(y).getVivienda(z);
```

La Ley de Demeter restringe los objetos a los que se deben enviar mensajes dentro de un método. Estos son:

- El objeto *this* o *self*.
- Un parámetro del método.
- Un atributo de *this*.
- Un elemento de una colección que es un atributo de *this*.
- Un objeto creado en el método.

Una de las principales desventajas del patrón VP es el coste de diseñar la variación protegida que es superior a un diseño simple. Por ello, los ingenieros de software expertos prefieren "un diseño simple y frágil en el que existe un equilibrio entre el coste del cambio y su probabilidad" [Larman02].
Entre las ventajas del patrón VP caben citar la posibilidad de añadir nuevas extensiones, añadir nuevas implementaciones sin afectar a los clientes,

reducir el acoplamiento y disminuir el impacto o coste de los cambios.

Para finalizar, y a modo de resumen, la mayoría de los patrones GoF y principios de diseño como el polimorfismo, las interfaces, encapsulación de datos, indirección, etc. son mecanismos para variaciones protegidas.

caso de estudio: ajedrez

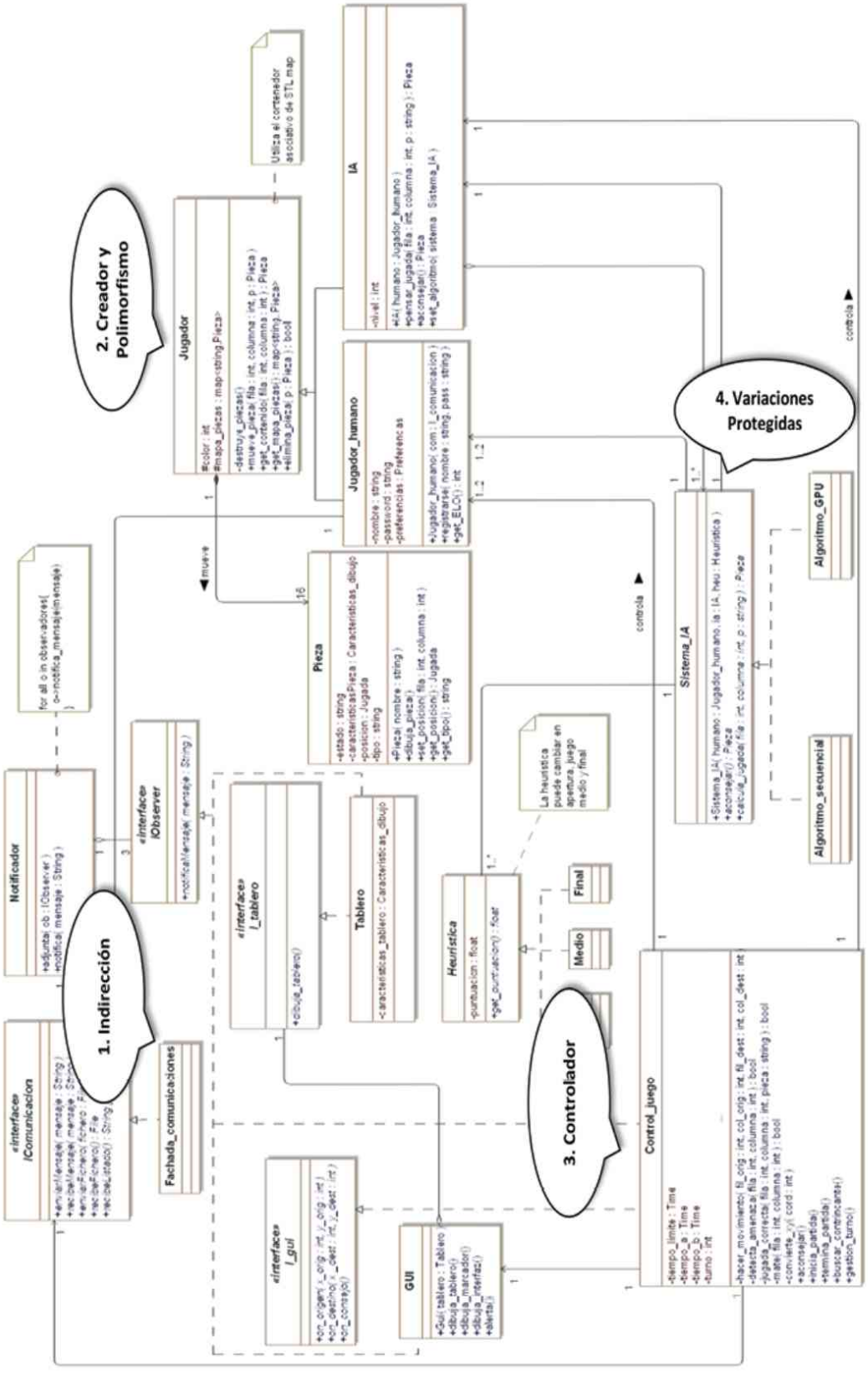


Figura 10.10. Distribución de los patrones GRASP en el juego de ajedrez

1. La clase *Fachada_comunicaciones* es un caso evidente de patrón indirección, pues realiza la actividad de intermediario entre el juego de ajedrez y la clase que envuelve los sockets con el propósito de facilitar esta función al sistema.
2. La clase *Jugador,Jugador_humano e IA* realizan las funciones de clases creadoras de las piezas con las que van a jugar la partida. Finalmente, también existe un caso de patrón de polimorfismo al clasificar los dos tipos correspondientes de jugadores.
3. La clase *Control_juego* es un controlador que se responsabiliza de la recibir los eventos del sistema GUI para gestionar de la partida, controlando información vital como los tiempos de cada jugador, los turnos, el control de la IA, etc.
4. Las clases que heredan de la clase abstracta*Sistema_IA* (*Algoritmo_secuencial* y *Algoritmo_GPU*) podrían considerarse como un caso de patrón de variaciones protegidas al tener en cuenta la posibilidad futura de añadir otro tipo de algoritmo y/o tecnología.

caso de estudio: mercurial

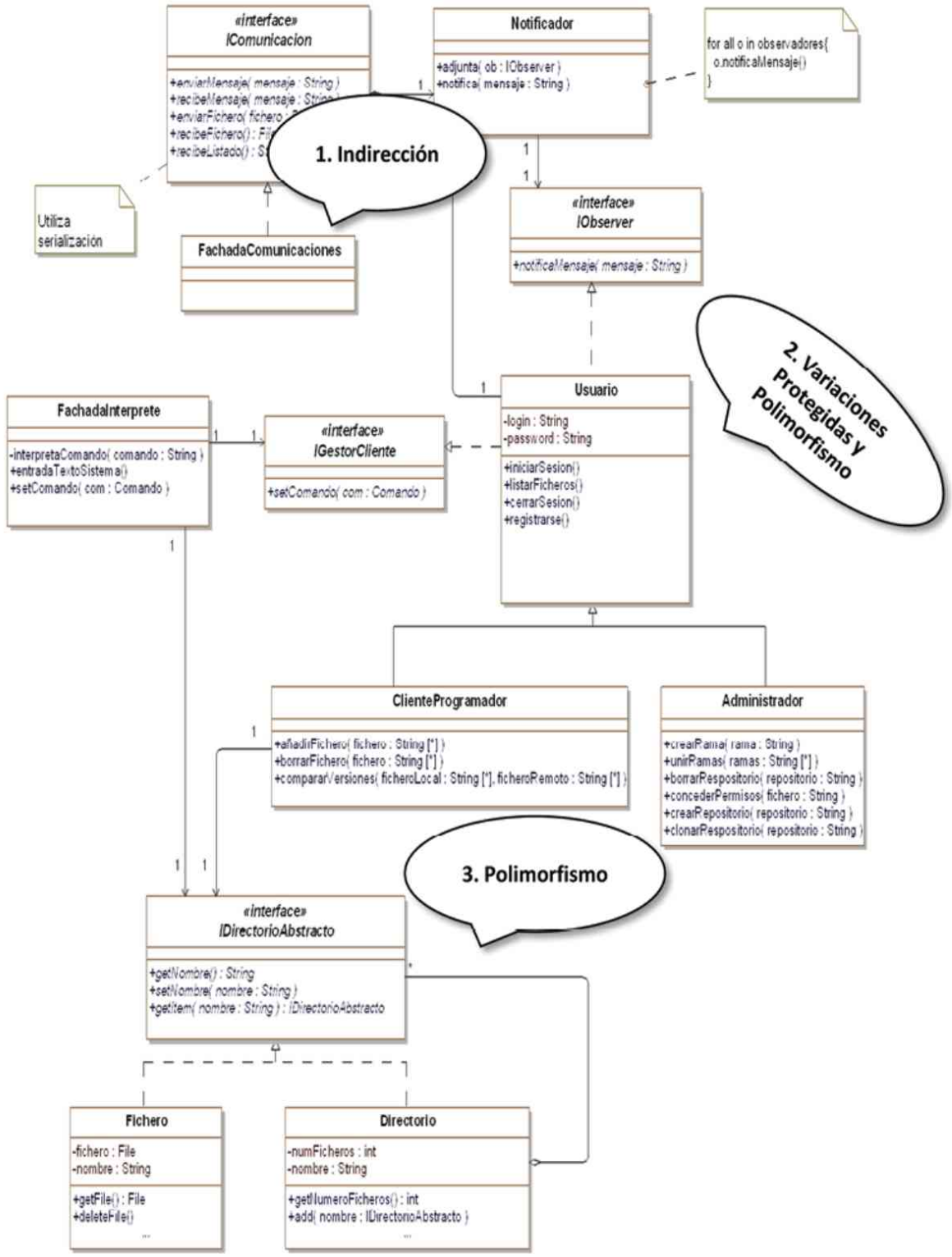


Figura 10.11. Distribución de los patrones GRASP en la aplicación Mercurial

1. Como en el caso del juego de ajedrez, la clase *FachadaComunicaciones* se hará responsable de intermediar entre el subsistema que envuelve los sockets y el sistema de la aplicación de CVS.
2. La clase *Usuario* implementa un patrón de variaciones protegidas, al tener en cuenta otro tipo de usuario en requerimientos futuros. Finalmente, la relación de herencia entre *Usuario*, *ClienteProgramador* y *Administrador* evidencia un caso de patrón polimorfismo para diferenciar los diversos tipos de usuario.
3. Existe también un caso de polimorfismo en la realización de la interfaz *IDirectorioAbstracto* en las clases *Fichero* y *Directorio*.

caso de estudio: servicio de cifrado remoto

Lado cliente

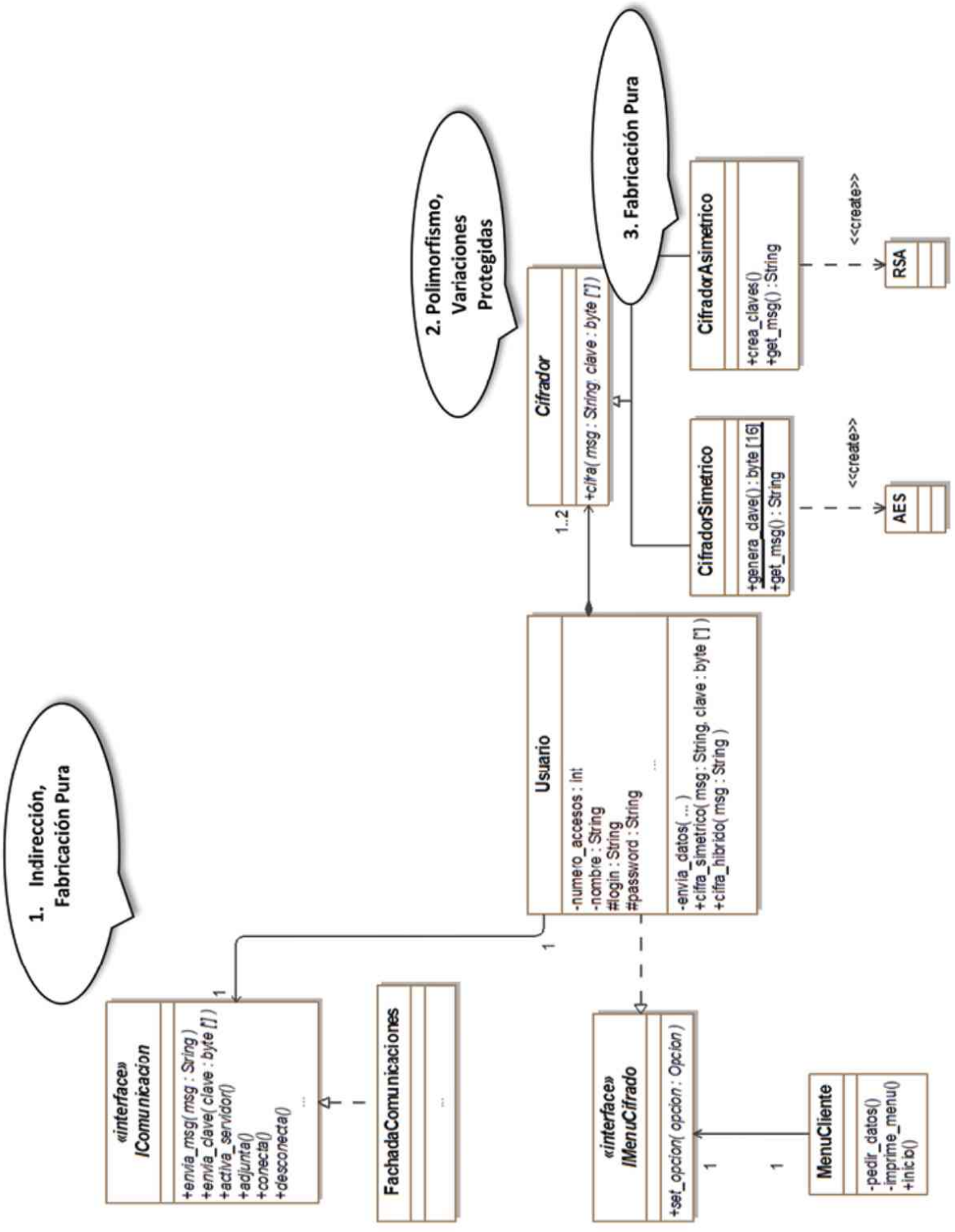


Figura 10.12. Distribución de los patrones GRASP en el lado cliente

Si procedemos a aplicar y reconocer patrones GRASP mediante los diagramas del modelo del dominio en primer lugar y los diagramas de clases después,

observamos cómo se repiten los modelos explicados en el capítulo 9, capítulo 10 y los ejemplos anteriores. La lección magistral que hay que extraer de esto es la facilidad de los patrones para reutilizar otras ideas y modelos existentes que han sido probadas con éxito durante la historia de la programación. Procedamos a reconocerlos en las figuras 10.12 y 10.13:

1. La clase *FachadaComunicaciones* realiza la función de patrón indirección al tener la responsabilidad de intermediar con el subsistema de *sockets* para el envío de mensajes cifrados. Así mismo, también realiza la función de patrón fabricación pura al ser una clase altamente cohesiva responsable de las comunicaciones y reutilizable en otros contextos.
2. La clase abstracta*Cifrador* modela un caso de patrón polimorfismo al responsabilizarse de ser la clase padre de la que heredan los diferentes tipos de cifrado. De forma similar, cumple la responsabilidad de patrón variaciones protegidas al abrirse a futuras ampliaciones de tipos de cifrado.
3. Finalmente, las clases que implementan el cifrado son candidatas a patrón de fabricación pura, al encargarse de realizar toda la lógica de negocio de los algoritmos de cifrado y, por tanto, reutilizables en otros dominios de aplicación.

Lado servidor

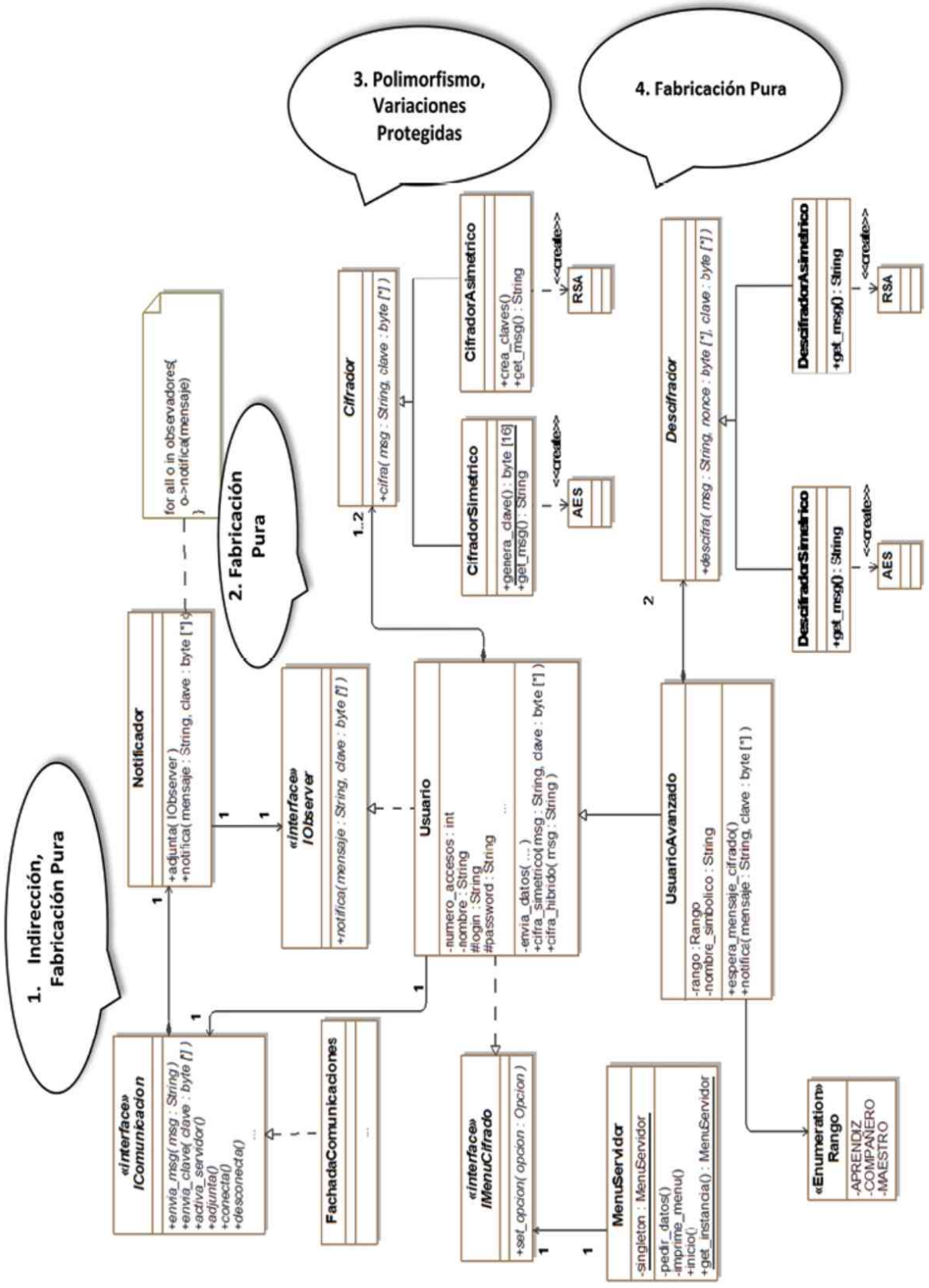


Figura 10.13. Distribución de los patrones GRASP en el lado servidor

1. La clase *FachadaComunicaciones* realiza la responsabilidad del patrón indirección al delegar a la clase *Socket* las funciones de comunicación por red. También es un patrón de fabricación pura al realizar de forma altamente cohesiva las tareas relacionadas con las funciones de sockets, siendo reutilizada también en el lado servidor.
2. Las clases *DescifradorSimetrico, DescifradorAsimetricoCifradorSimetrico* y *CifradorAsimetrico* son cuatro casos de patrón GRASP de fabricación pura, puesto que realizan una tarea altamente cohesiva relacionada con el cifrado del mensaje, pudiendo ser reutilizables en otras aplicaciones si fuera necesario.
3. La clase abstracta*Cifrador* y la clase abstracta *Descifrador* son un claro ejemplo de patrón de estrategia de GoF, pero además cumplen con las responsabilidades de ser las interfaces base para el polimorfismo de las clases de cifrado y descifrado, previendo la futura necesidad de ampliar los algoritmos mediante el patrón de varaciones protegidas.
4. El conjunto de clases e interfaz que implementan el patrón observer también es un caso de frabriación pura, pues en general, todos los patrones de diseño GoF son también patrones fabriación pura.

diagramas de estado

«Cuando eres joven quieres ser mayor, cuando eres mayor quieres ser joven».

(Anónimo).

Después del estudio de los patrones de diseño más importantes y su aplicación práctica en algunos ejemplos, retomamos de nuevo los diagramas asociados al modelado del comportamiento.

En este capítulo nos centraremos en los diagramas de estados como notación formal de UML para modelar autómatas. El principal objetivo de estos diagramas es representar gráficamente el comportamiento de los clasificadores más importantes de UML. Estos clasificadores pueden ser actores, clases, interfaces, nodos, señales, casos de uso, componentes, subsistemas, etc. Lo más frecuente es utilizar los diagramas de estado para representar el comportamiento de los diferentes valores que van adquiriendo los atributos de un objeto cuando se realiza una operación sobre ellos.

Los diagramas de estados suelen ser autómatas finitos, es decir, que alcanzan un estado final de aceptación en el cual terminan su progresión. Son representados como grafos dirigidos con nodos y aristas a los que se le añaden etiquetas informativas. Como veremos, las máquinas de estados son muy utilizadas en electrónica, reconocimiento del lenguaje y en un amplio rango de disciplinas informáticas y científicas. Los diagramas de estados en UML están basados en el formalismo de los diagramas del ingeniero *David Harel*, inventor de la notación original inspirada en grafos.

conceptos básicos

Antes de pasar a detallar los diferentes tipos de diagramas de estados proporcionados por la especificación UML explicaremos los elementos primordiales que estructuran estos diagramas.

- **Estado:** Es el momento presente donde se encuentra el autómata mientras transita entre diferentes situaciones de ejecución.
- **Evento:** Es el suceso o causa que propicia el cambio de estado.
- **Transición:** Es el paso que se realiza entre un estado y otro al producirse un evento.

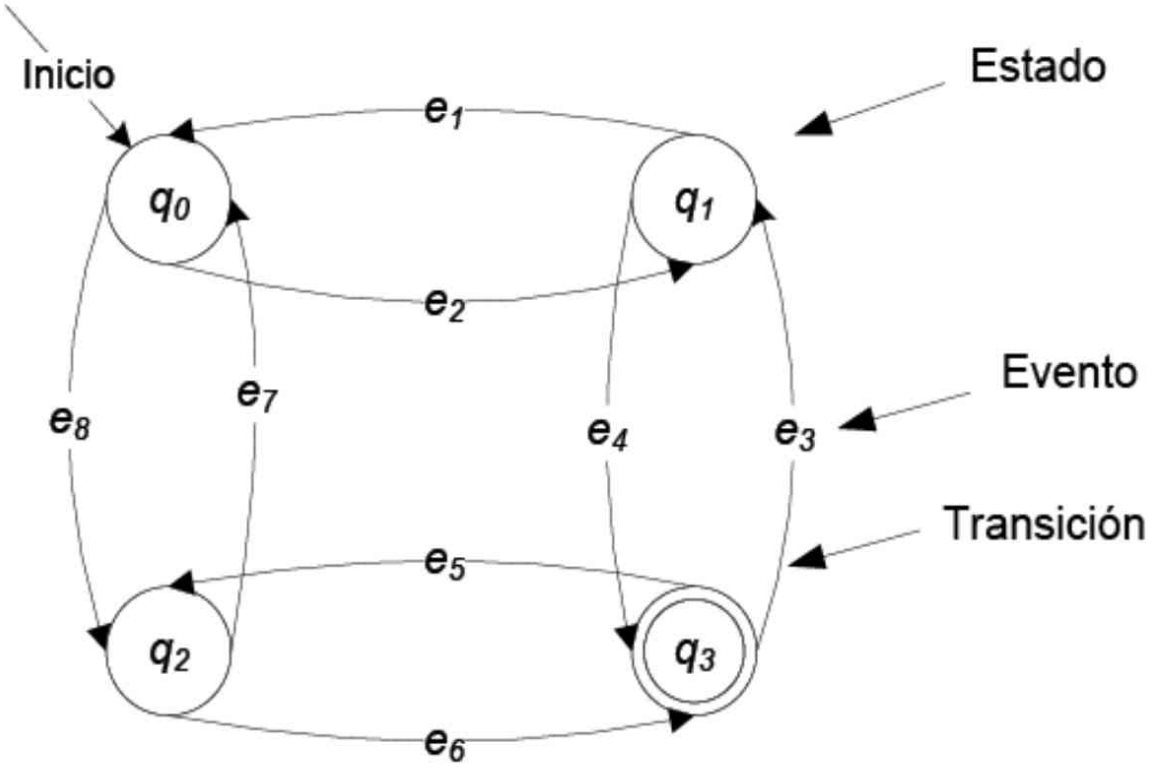


Figura 11.1. Diagrama de estados básico

La figura 11.1 muestra un diagrama de estados tradicional con cuatro estados: $\{q_0, q_1, q_2, q_3\}$, ocho eventos: $\{e_1, e_2, e_3, e_4, e_5, e_6, e_7, e_8\}$ y un estado final de aceptación: $\{q_3\}$. El movimiento o transición de un estado q_i à q_j implica la

llegada de un evento e_i .

La figura bien podría modelar el comportamiento dinámico de la instancia o instancias asociadas a una clase según se realizan las llamadas a sus operaciones. Los estados equivaldrían a los valores que van adoptando los atributos y los eventos a los sucesos que hacen que se ejecuten en ellos las operaciones. Según la OMG los diagramas de estados en UML se dividen en *máquinas de estado de comportamiento* y *máquinas de estado de protocolo. Las máquinas de estado de comportamiento* modelan el comportamiento de clasificadores de UML. *Las máquinas de estado de protocolo* modelan protocolos, por ejemplo, el ciclo de vida de los objetos y clasificadores como interfaces y puertos ya que estos no poseen ningún tipo de comportamiento.

estructura de un estado

La estructura de estado definida en UML 2.x es:

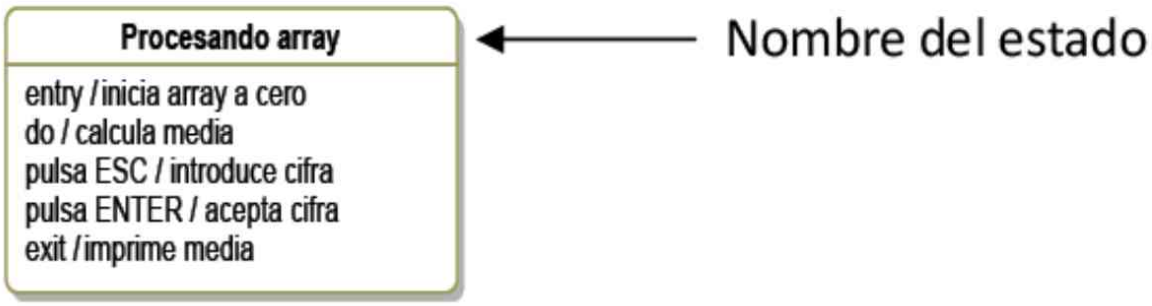


Figura 11.2. Ejemplo de estado

Los componentes del estado se dividen en:

Acciones y actividades	Significado
Entry	Acción que se realiza cuando se entra en el estado.
Exit	Acción que se realiza cuando se sale del estado.
Eventos internos	Es un evento que ocurre dentro del estado y produce un efecto o acción. Es el caso de ESC y ENTER.
Do	Realiza una actividad interna.

Tabla 11.1. Acciones y actividades de un estado

Donde las *acciones* se asocian con transiciones internas del estado en respuesta a un evento, implican una ejecución instantánea y no pueden ser interrumpidas, mientras que las *actividades* toman un tiempo finito y pueden ser interrumpidas.

Es importante destacar que los diagramas de protocolo no contienen los elementos de la tabla 11.1.

estructura de las transiciones

Las transiciones representan el cambio de un estado a otro a causa de un evento o eventos. Según se trate de un diagrama de estado de comportamiento o de protocolo la estructura de la transición tomará un formato u otro. De este modo para los *diagramas de comportamiento* las transiciones tienen la siguiente notación[30]:

[evento[,evento]*[guarda][/acción]*]



Figura 11.3. Transiciones en diagramas de estado de comportamiento

Dados dos estados separados por un evento, si se cumple la *guarda*, que es una condición *booleana* en el evento, se lleva a cabo la acción y se pasa de un estado a otro.

En los *diagramas de estado de protocolo* la estructura de la transición es la siguiente:

[[precondición][evento[,evento]*[/postcondición]]



Figura 11.4. Transiciones en diagramas de estado de protocolo

En general, en la expresión [evento[,evento]*] se considera que se ejecutará una transición siempre y cuando se cumpla que:

(evento$_1$ OR evento$_2$ OR ... evento$_n$) AND (guarda)à Transita

Normalmente, en la mayoría de los diagramas UML no se llega a detallar tanta información dentro de los estados y en las transiciones. Con frecuencia recurriremos a indicar generalmente el nombre del estado, las guardas y eventos en las transiciones, sin llegar a ser excesivamente detallistas en la especificación de estos.

tipos de nodos

Nodos inicial y final

El nodo inicial especifica el punto de partida donde comienza la ejecución del autómata y da paso al primer estado que tiene lugar dentro del diagrama. Se representa mediante un círculo de color negro. El estado final es el punto de llegada y terminación del autómata e implica la conclusión de las transiciones entre estados. Se representa mediante un círculo blanco con otro círculo concéntrico interior de color negro.

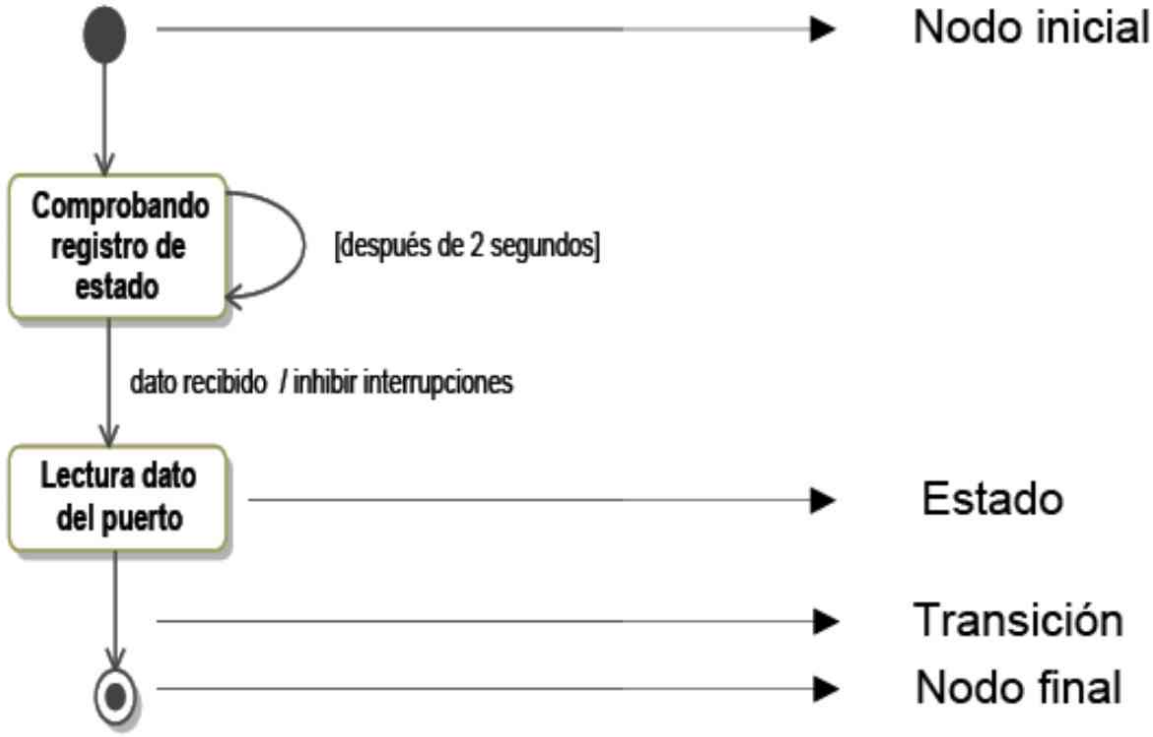


Figura 11.5. Diagrama de estados elemental

El diagrama de la figura 11.5 modela el algoritmo de una clase que implementa la lectura de un puerto de E/S en un sistema operativo. El autómata arranca en el nodo inicial para realizar una transición al estado de comprobación del registro de estado de la controladora. Mientras no se recibe el evento de llegada de un dato al puerto, el estado se mantiene sobre él mismo. Cuando se recibe, la transición ejecuta la acción de inhibir las interrupciones de E/S antes de la lectura del puerto. El diagrama termina definitivamente con la llegada al nodo final que implica el fin de la ejecución del proceso.

Nodos de interconexión

Puede suceder que varias líneas de transiciones deban concurrir sobre un mismo estado o que varias transiciones partan de una condición hacia varios estados. Cuando ocurre esta situación utilizaremos un nodo de interconexión con el fin de fusionar o bifurcar transiciones. Dicho nodo se representará como un círculo de color negro de tamaño inferior al nodo inicial.

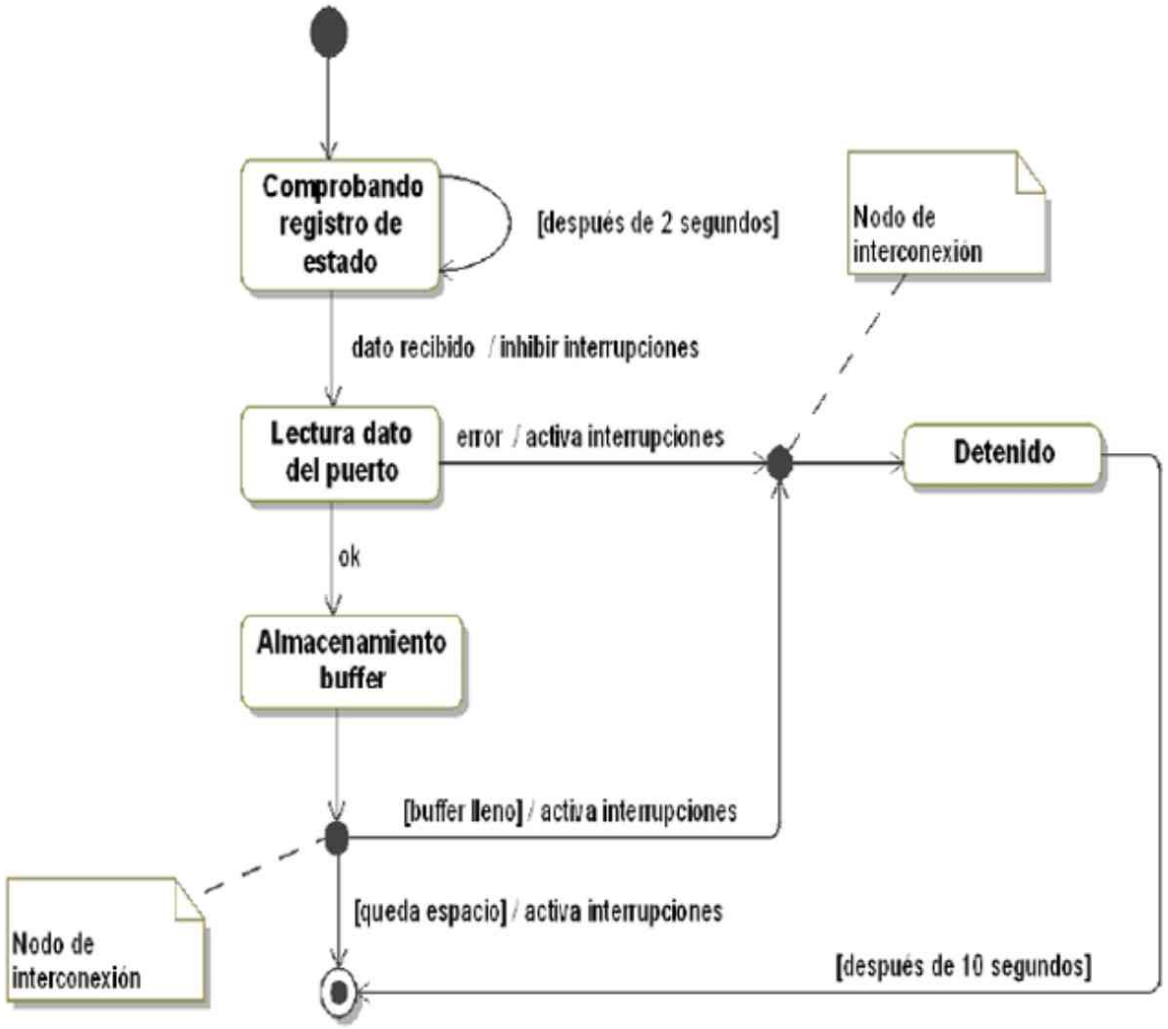


Figura 11.6. Ejemplo de diagrama con nodos de interconexión

Como se muestra en la figura 11.6 el estado de almacenamiento en el buffer puede producir una situación condicional dependiendo de si se encuentra lleno o no. De igual forma, tanto si se produce un error de E/S como si el buffer está lleno las transiciones salientes confluirán en un nodo de interconexión antes llegar al estado de detención.

Nodos condicionales

De forma alternativa y cuando se trate de una situación de bifurcación condicional podemos utilizar el símbolo del rombo para indicar la salida de las diferentes transiciones.

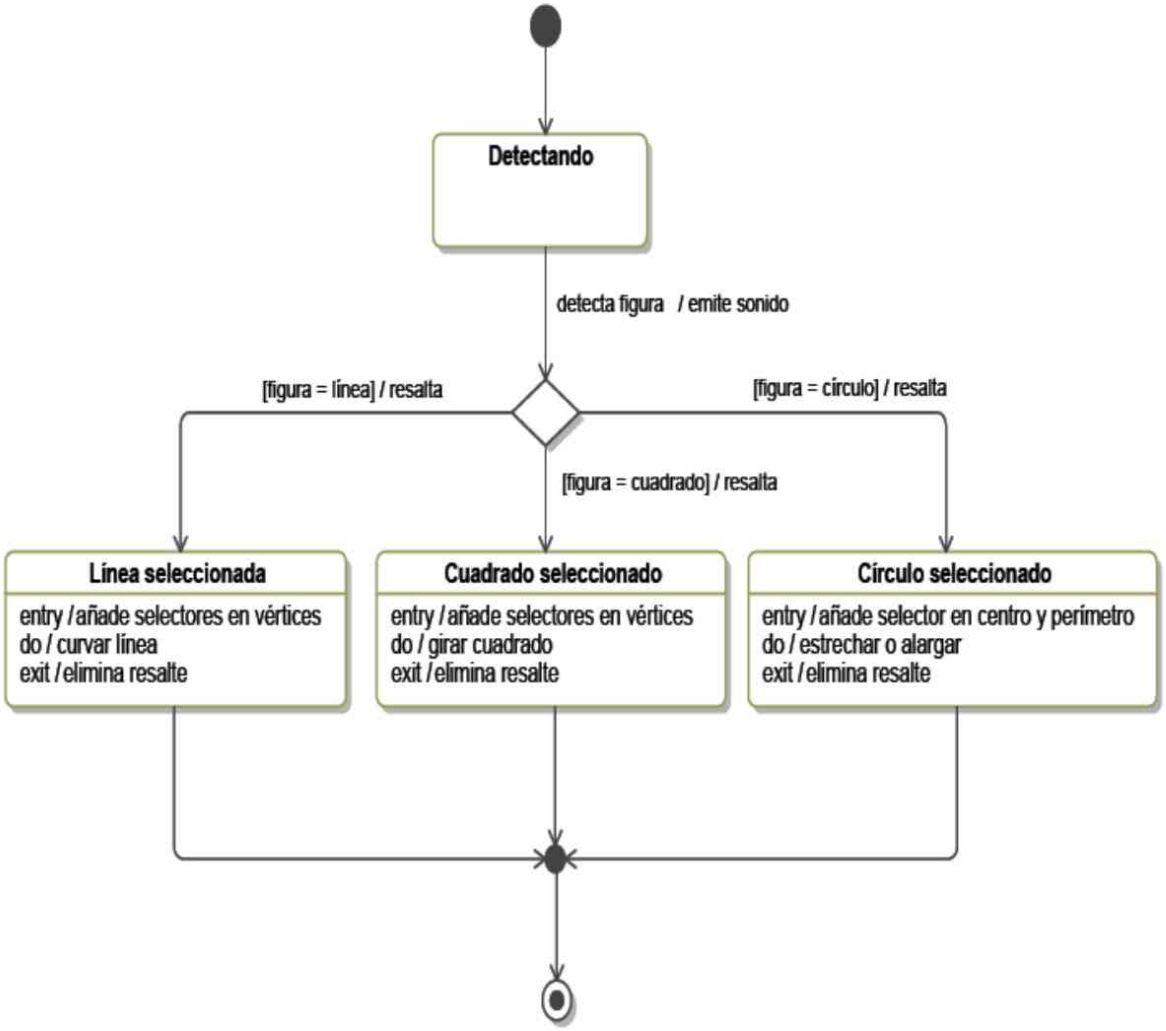


Figura 11.7. Utilización del nodo condicional

En el ejemplo podemos observar la utilización del nodo de decisión como alternativa más legible al nodo de interconexión. Las opciones del nodo condicional deben ser siempre excluyentes, no pudiendo reflejar ningún tipo de ambigüedad. El ejemplo 11.7 modela los estados de una clase *figura* que es seleccionada por el usuario de una aplicación gráfica. Dependiendo del tipo de figura que seleccione, el autómata se redirigirá a diferentes estados donde se le aplicarán las respectivas transformaciones geométricas.

eventos

Como se mencionó al comienzo del capítulo, uno de los elementos principales del diagrama de estados son los eventos. Cuando tiene lugar un evento (al ocurrir una circunstancia externa programada en la clase), se produce inmediatamente la transición de un estado a otro. Dependiendo del uso que se le dé al evento se clasificarán en: eventos de llamada, eventos de señalización, eventos de cambio y eventos de tiempo. A continuación se describen las características esenciales de los dos más importantes:

Eventos de llamada

El caso más común ocurre cuando se solicita la realización de una operación de la instancia de una clase como evento. Para ilustrar este caso considérese la siguiente clase que modela una *Tienda*:

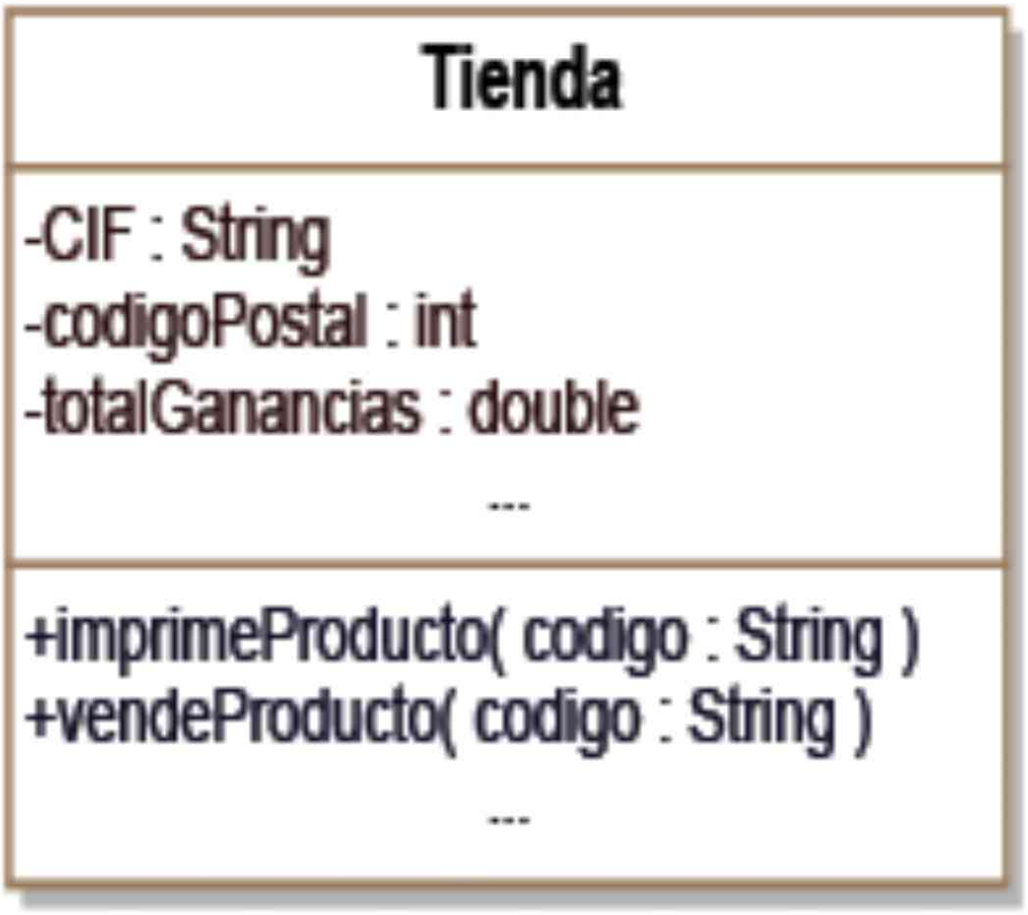


Figura 11.8. Clase Tienda

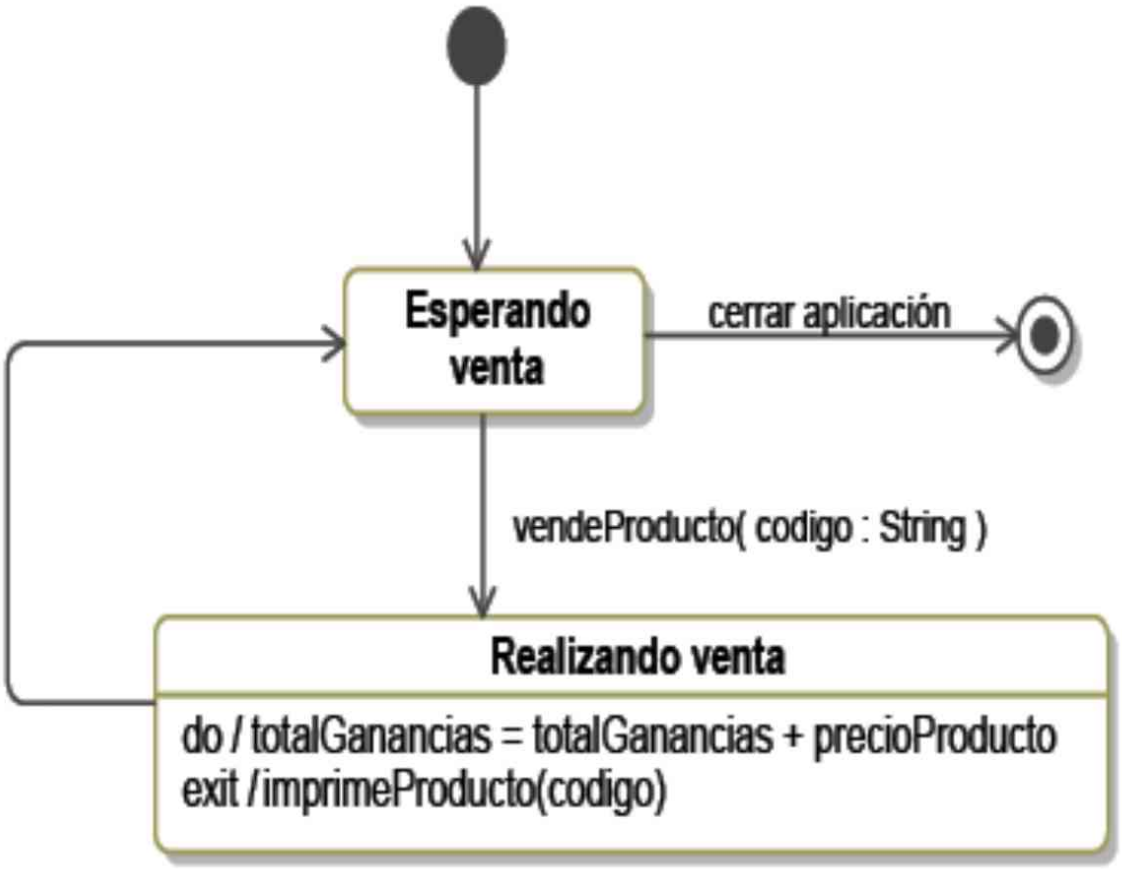


Figura 11.9. Diagrama de estados con eventos de llamada

El hecho de realizar una llamada a la operación *vendeProducto* sobre la instancia de la clase *Tienda* provoca un cambio de estado inducido por la generación del evento.

En el ejemplo de la figura 11.9 se han utilizado dos operaciones de la clase Tienda: *vendeProducto e imprimeProducto* de forma externa e interna al estado respectivamente.

Eventos de tiempo

Otro caso muy común es el evento de tiempo. Este tipo de eventos se generan como respuesta a un suceso de transcurso del tiempo, por ejemplo: milisegundos, segundos, horas, días, meses, etc. La figura 11.10 ejemplifica una situación de una máquina de estados controlada por un evento de tiempo:

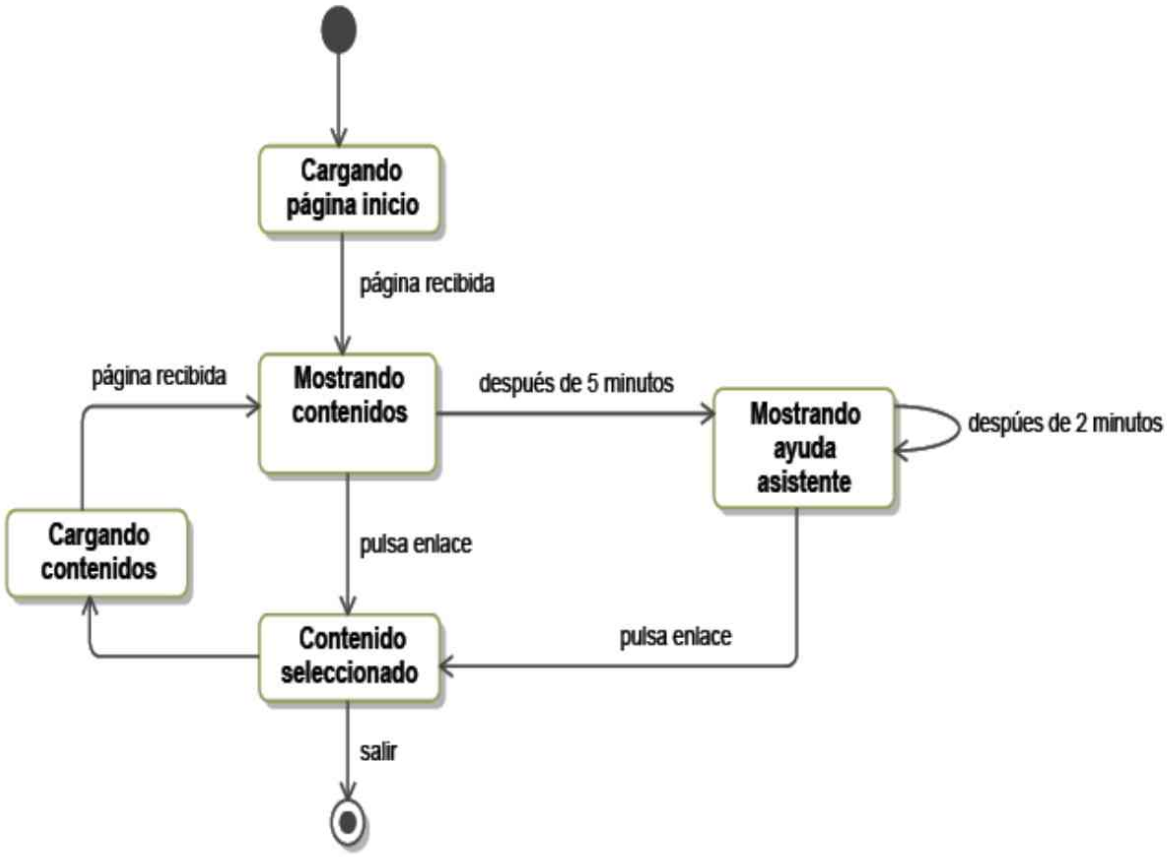


Figura 11.10. Ejemplo de diagrama de estados con dos eventos de tiempo

En el ejemplo de la figura 11.10 se supone un caso de aplicación que muestra una página de Web de información con diferentes opciones. Si el usuario no sabe qué opción elegir transcurridos cinco minutos después de la carga y la visualización, se mostrará un asistente para guiarle en el uso de la página. Del mismo modo, si transcurren dos minutos más sin que el usuario seleccione una opción, se volverá a mostrar un mensaje de ayuda del asistente.

estados compuestos

Los estados compuestos son aquellos formados por uno o más máquinas de estados anidadas o que se ejecutan de forma concurrente o paralela. En esta sección veremos los dos casos de composición de estados: la *composición simple* y la *ortogonal*.

Estados compuestos simples

En los estados compuestos simples únicamente existe una región o cauce de ejecución para el conjunto de estados anidados y donde no se produce concurrencia o paralelismo. Suele representarse como un superestado que agrupa varios conjuntos de estados a modo de bloque. Este bloque se comunica con el resto de los bloques por medio de pseudo-estados, es decir, estados no convencionales como el estado de inicio, terminación o bifurcación. En este caso los pseudo-estados son los estados de entrada y salida de los superestados.

La figura 11.11 muestra un caso de ejemplo muy habitual en los navegadores Web. Se trata de dos superestados para resolver el *hostname* (DNS) y para la carga del contenido HTML respectivamente. El superestado de la parte superior de la imagen se comunica con el superestado inferior por medio de pseudo-esta-dos. Encontramos aquí dos pseudo-estados de entrada y cinco de salida (dichos estados siempre se ubican en los límites del superestado). En caso de que sean de entrada se mostrarán como un círculo en blanco, mientras que si son de salida se mostrarán mediante un círculo con un aspa en su interior. Dichos pseudo-estados equivaldrían a puertas que comunican un superestado con el resto del diagrama.

Otro tercer tipo de pseudo-estado es el de terminación que es el que se encuentra en el ejemplo asociado al superestado para mostrar el contenido HTML y en los mensajes de error. Los pseudo-estados de terminación implican que se termina la ejecución de esta máquina de estado por medio de su objeto de contexto. En el caso de la figura 11.11, al encontrar código malicioso (malware) se finaliza el objeto que interpreta el código HTML. De igual forma, en el caso de fallo de transmisión si no se resuelve el *hostname* o si se produce un error HTTP, el objeto relacionado se destruye en espera de una nueva petición de URL.

El ejemplo analizado en esta sección es un modelo simple para el caso de carga de páginas HTML desde un navegador; no obstante en la realidad dichos diagramas se pueden complicar en exceso. También se evidencia que en este tipo de estado compuesto no sucede ninguna ejecución concurrente, siendo más bien el proceso de la máquina de estados y subestados una secuencia lineal del cauce de ejecución.

En la siguiente sección veremos un caso donde sí ocurre paralelismo.

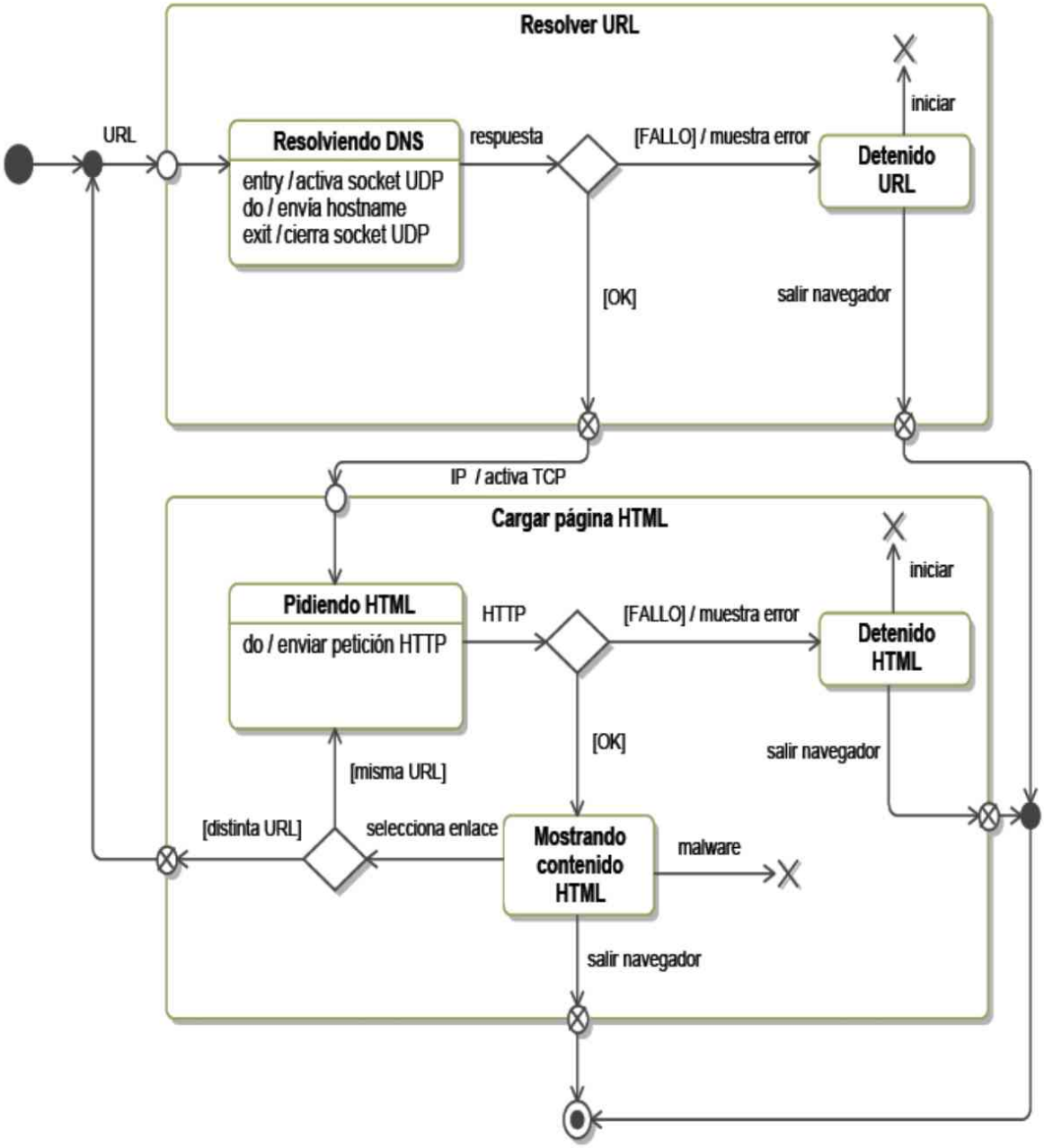


Figura 11.11. Ejemplo de estado compuesto simple

Estados compuestos ortogonales

En este tipo de estados compuestos nos encontramos con dos o más regiones o cauces de ejecución paralelos o concurrentes. La notación de UML para especificar la existencia de varias submáquinas ejecutándose en paralelo es mediante la siguiente estructura:

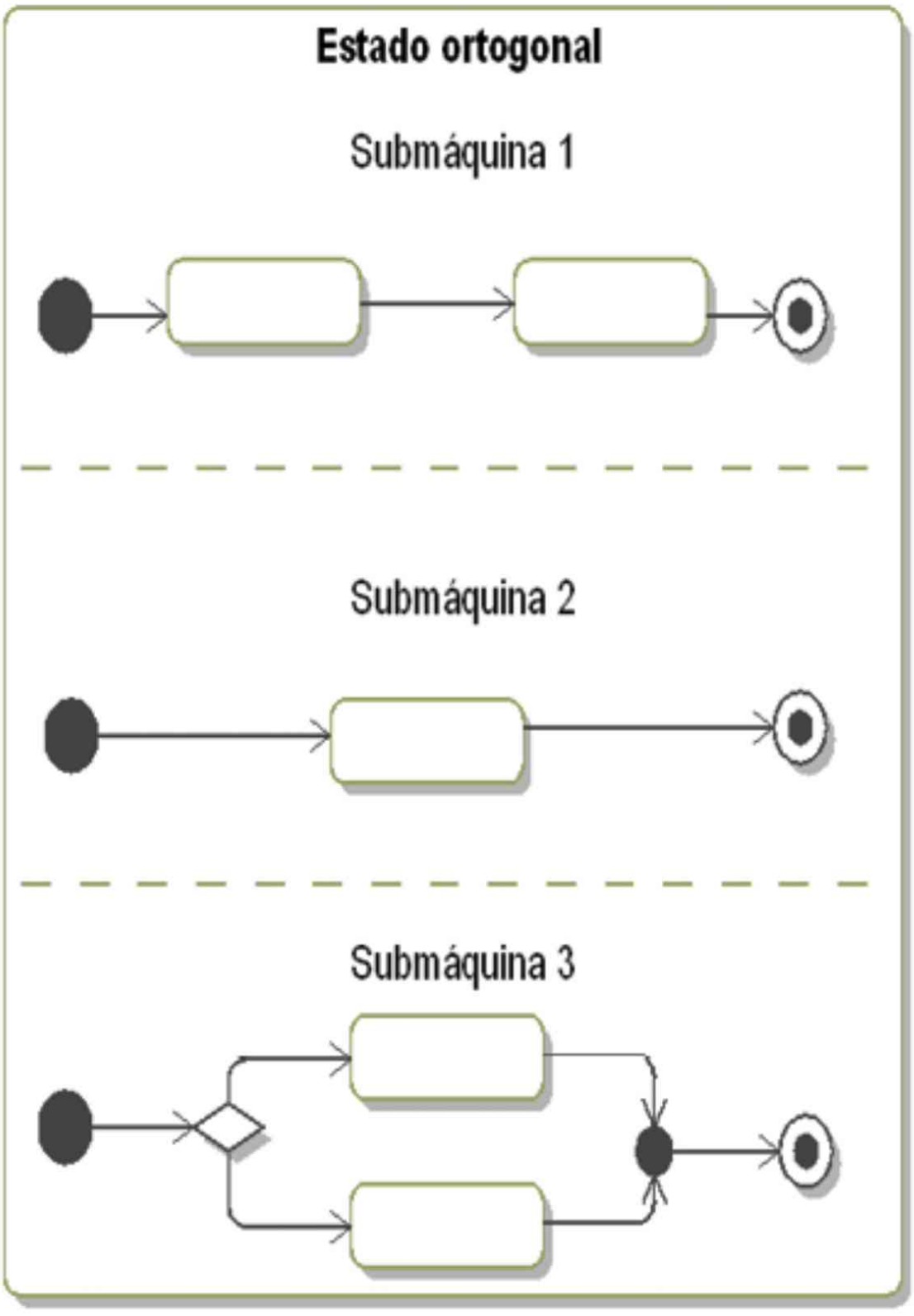


Figura 11.12. Estado compuesto ortogonal (3 submáquinas)

En el ejemplo de la figura 11.12 encontramos tres máquinas de estados

ejecutándose concurrentemente o en paralelo. Dichos autómatas contendrán regiones separadas mediante líneas discontinuas en donde se realizarán los estados y las transiciones de manera independiente.

En algunas situaciones es necesario descomponer la transición de entrada en varias transiciones de entrada para cada submáquina. De igual forma las transiciones generadas dentro de cada máquina deberían ser recogidas en una barrera de sincronización. Para dichas situaciones disponemos de las notaciones *fork* (bifurcar) y *join* (unir) que se especifican mediante una barra vertical u horizontal. En el ejemplo de la figura 11.13 podemos observar un caso de uso de fork y join.

Cada nodo final termina su cauce paralelo (submáquina) independiente del resto. Si utilizáramos un pseudo-estado de terminación (aspa), finalizarían todos los cauces juntos, es decir, toda la máquina de estados completa inmediatamente.

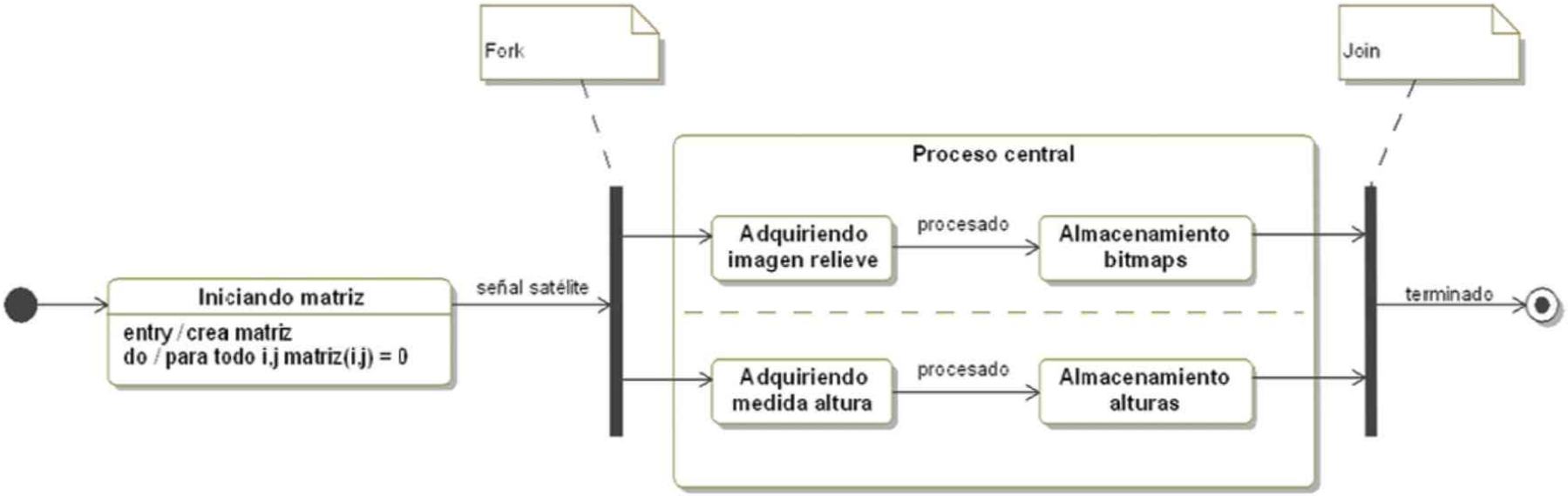

Fork
Join
Proceso central
Iniciando matriz
entry / crea matriz
do / para todo i,j matriz(i,j) = 0
señal satélite
Adquiriendo imagen relieve
procesado
Almacenamiento bitmaps
Adquiriendo medida altura
procesado
Almacenamiento alturas
terminado

Figura 11.13. Estado ortogonal con fork y join

sincronización de submáquinas

¿Cómo es posible sincronizar los estados de diferentes submáquinas ejecutándose en distintas regiones? La respuesta a esta pregunta la encontramos en la definición de transición, más concretamente en la sección de guarda. La utilización de dicha sección como *flag* permite establecer un orden de ejecución en las transiciones de un estado a otro. Simplemente recurriremos a los atributos de los estados para prefijar los valores de dichos flags que se utilizarán a modo de semáforos de sincronización.

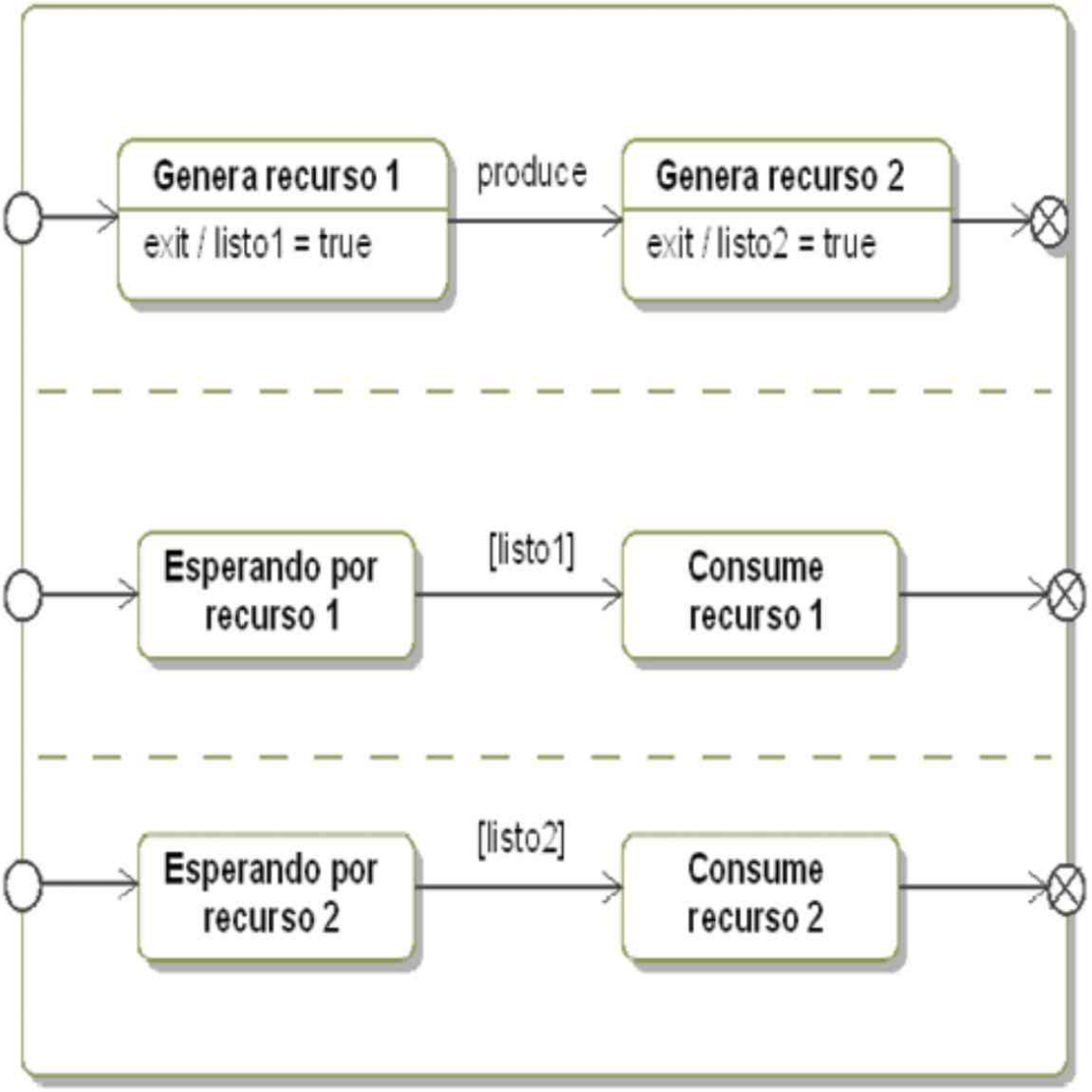


Figura 11.14. Tres submáquinas concurrentes con sincronización

La utilización de sincronismo en el ejemplo de la figura 11.14 se consigue mediante la utilización del flag *listo*, que es establecido por las acciones de los estados productores y controlado por las guardas de los estados consumidores. De esta forma se consigue bloquear la transición de un estado mientras exista concurrencia.

simplificación del diagrama de estados

Es frecuente la complejidad que suelen alcanzar los diagramas de estados, normalmente provocados por la gran cantidad de posibles valores que se pueden asignar a los atributos de las cada vez más complejas clases.

Con la finalidad de simplificar la tarea al modelador y a su vez crear diagramas de estados más legibles y ordenados, surgen entidades en UML como el *icono de estado compuesto* para facilitar en gran medida esta labor.

El *icono de estado compuesto* es una simplificación de lo que podría equivaler a un diagrama compuesto simple u ortogonal, a modo de caja negra que encapsula y abstrae otro/s diagrama/s de estados. Se representa mediante un macroestado con entradas/salidas y en el cual se encapsulan sus correspondientes transiciones.

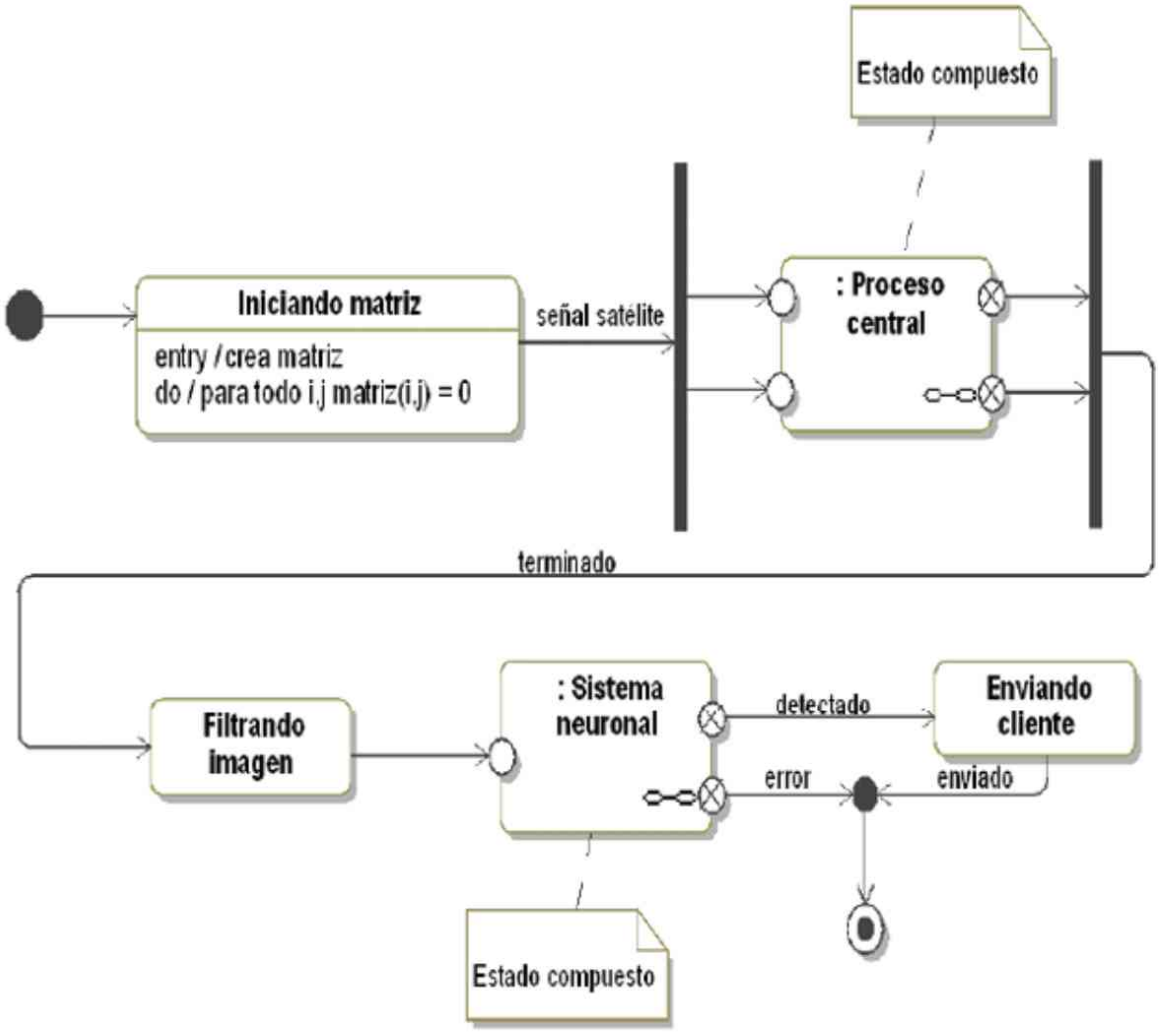


Figura 11.15. Simplificación de la figura 11.13 utilizando iconos de estado compuesto

El ejemplo de la figura 11.15 utiliza las dos submáquinas representadas en el ejemplo de la figura 11.13. Para simplificar el diagrama, la máquina que abstrae el *procesamiento central* se representa como una entidad encapsulada que no muestra sus contenidos en el modelo. También se muestra otra máquina compuesta denominada *Sistema neuronal* que no utiliza concurrencia y que ilustra igualmente otro caso de simplificación de un estado compuesto simple.

El ejemplo basado en las figuras 11.13 y 11.15 modela un sistema de adquisición de imágenes vía satélite para el análisis topográfico del terreno. En la

figura 11.13 el proceso comienza inicializando una matriz a cero para almacenar los datos recibidos. Cuando se recibe la señal del satélite se lanzan dos submáquinas paralelas que reciben octetos con información gráfica del terreno junto a una medición de las alturas. Una vez terminados, ambos procesos se sincronizan en una barrera "join" y se procede al filtrado de la imagen (figura 11.15). El resultado de esta imagen filtrada se analizará en un sistema neuronal que detectará una determinada forma y la enviará al cliente en caso de coincidencia.

historial

Puede ser útil en algunas ocasiones mantener información del estado donde nos encontrábamos cuando se abandona un conjunto de estados. Esta situación puede presentarse a consecuencia de una transición que nos "saca" de un estado interno del superestado hacia otro estado externo. Si posteriormente otra transición nos lleva de nuevo al mismo superestado no será posible recordar en qué subestado nos encontrábamos.

Estas situaciones se resuelven por medio de los *pseudo-estados historia* que permiten solucionar el problema anteriormente descrito y facilitan la depuración de errores.

Historia superficial

Con la *historia superficial* conseguimos mantener una memoria del último estado donde estuvimos al mismo nivel donde se encuentra el nodo de historia superficial. Por ejemplo, el hecho de que se sitúe el pseudo-estado H en el superestado del nivel 2 de la figura 11.16 implica, que en caso de salida de emergencia de un subestado en ese nivel, al regresar únicamente recordará en qué subestado nos encontrábamos dentro de ese nivel.

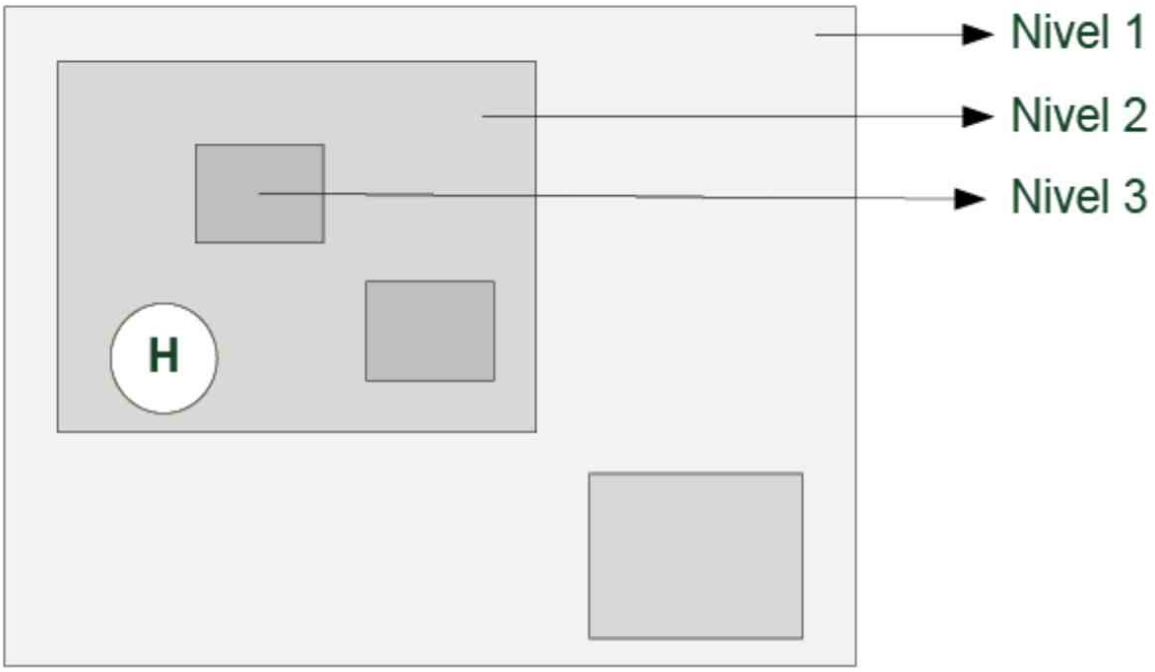


Figura 11.16. El nodo H recuerda en qué estado estábamos dentro del Nivel 2 únicamente

Historia profunda

Sin embargo, la *historia profunda* realiza las mismas acciones de memoria que la historia superficial pero en este caso no solo recuerda el nivel actual sino que mantiene un historial de los estados que se encuentran por debajo de él.

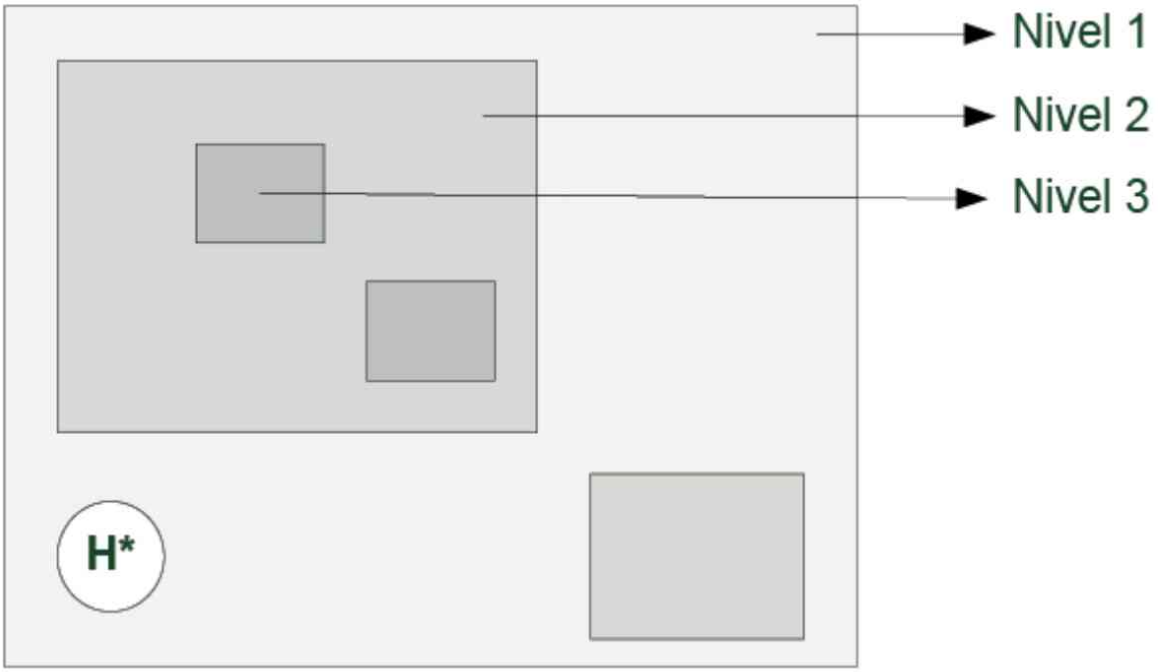


Figura 11.17. El nodo H* recuerda en qué estado estábamos desde el Nivel 1 hasta el Nivel 3

En el ejemplo de la figura 11.18 se representan los dos casos de utilización de los nodos de historial. El diagrama modela una línea de producción en cadena donde se requieren dos componentes para formar la pieza *Alfa*. En este contexto pueden suceder dos tipos de eventos: el primero consistiría en un agotamiento de la reserva de componentes de ensamblaje, y el segundo, una caída del suministro eléctrico que afectaría a toda la cadena. En el primer caso el nodo H permite continuar en el estado de ensamblaje donde se encontraba, mientras que en el segundo caso el nodo H* recuerda cualquier subestado dentro del superestado de montaje de pieza Alfa.

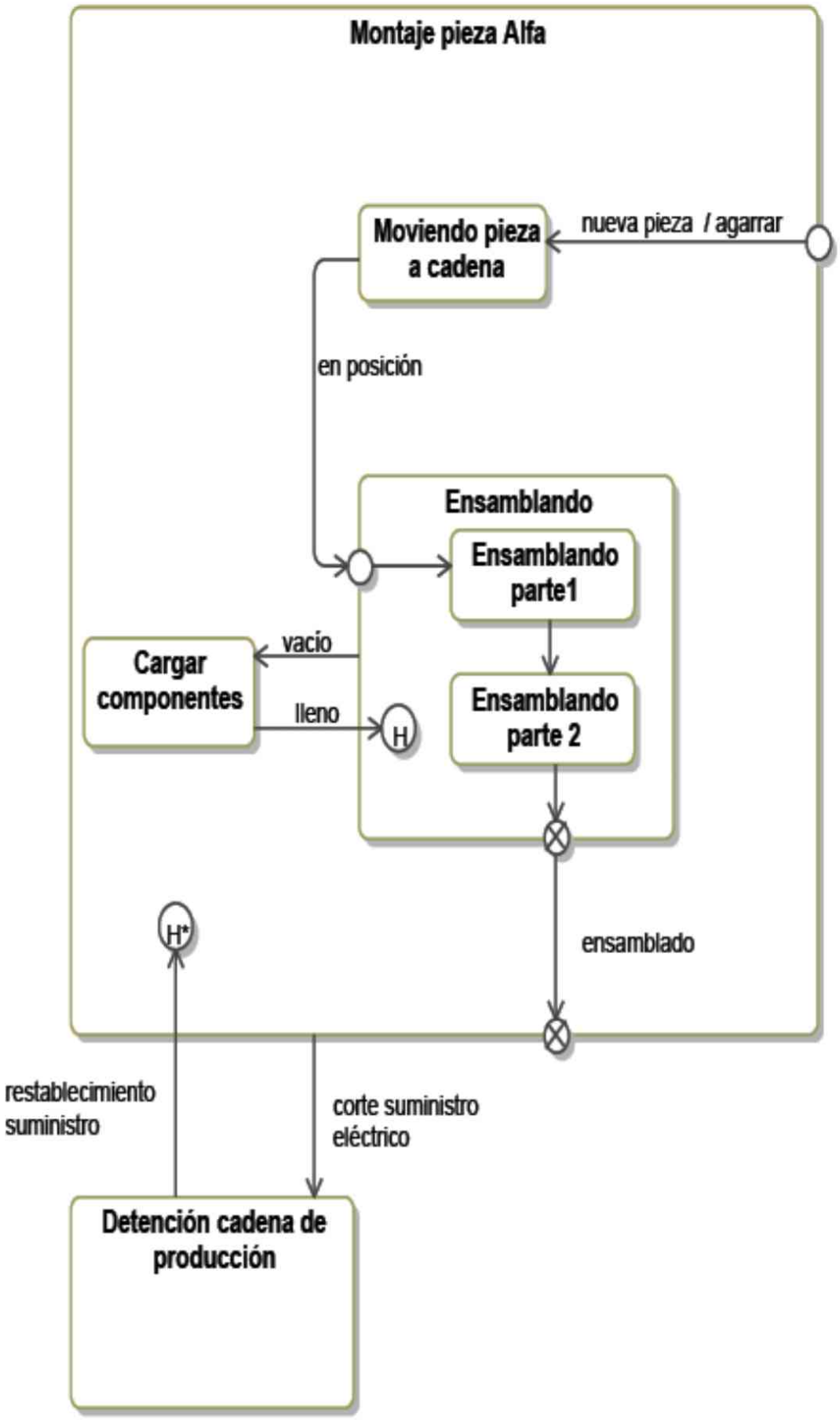

Montaje pieza Alfa
Moviendo pieza a cadena
nueva pieza / agarrar
en posición
Ensamblando
Ensamblando parte1
Ensamblando parte 2
vacío
Cargar componentes
lleno
H
H*
ensamblado
restablecimiento suministro
corte suministro eléctrico
Detención cadena de producción

Figura 11.18. Ejemplo de uso de los dos tipos de nodos de historial (H y H*)

caso de estudio: ajedrez

En la figura 11.19 de la página siguiente se representa el diagrama de estados por los que transita el juego del ajedrez durante una partida.

El diagrama refleja la vida de la aplicación desde que se inicia hasta que termina. Al principio de la secuencia de estados se pregunta al usuario si desea jugar en red o con la máquina. En caso de que desee jugar con la IA se procede a la espera de selección automática del nivel ELO del jugador. En caso contrario se transita a un estado compuesto simple que procede a la conexión al servidor. Terminado el protocolo de conexión con el servidor se procede a la búsqueda del contrincante, para ello se entra en un estado de comparación de los niveles de destreza (ELO). Nótese que dentro de este estado compuesto existe la posibilidad de pérdida de la conexión. Para dar respuesta a este problema se ha recurrido al uso de la notación H (historia superficial) con la finalidad de recordar el estado donde nos encontrábamos después de la caída de la conexión. El nodo H viene del estado ":*Conectando*" que no es más que un estado reutilizado del que realiza las acciones de conexión al servidor.

Una vez encontrado el jugador rival comienza el juego en sí; aunque en este caso y con fin de no complicar en exceso el diagrama se ha abstraído su complejidad mediante un estado compuesto. Antes de jugar la partida es necesario ejecutar el evento de espera de cinco segundos, permitiendo así la adecuada preparación mental del jugador humano.

Finalmente, una vez terminada la partida, se llega a un nodo de condición que se bifurca dependiendo de si se ha jugado en red o mediante la IA. En caso de haber elegido jugar la partida en red se entra en el estado compuesto ":*CerrandoSesión*" que realiza el protocolo de desconexión con el servidor central de usuarios.

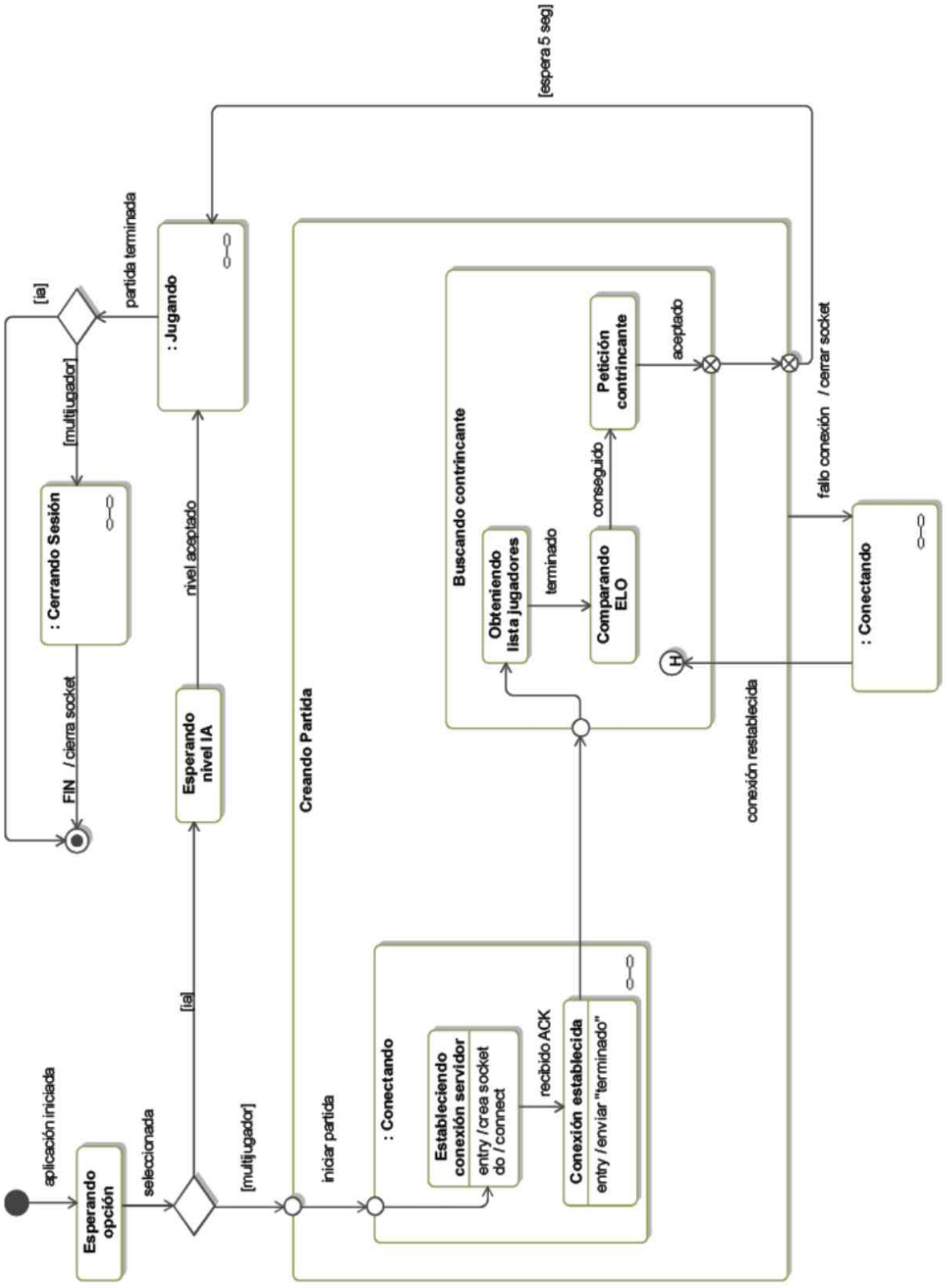


Figura 11.19. Diagrama de estados de ajedrez

caso de estudio: mercurial

En la figura 11.20 se muestra el contexto de ejecución de un diagrama de estados para el proceso de análisis de una cadena de comandos en *Mercurial*. La máquina de estados comienza cuando el proceso está en espera de la entrada de una orden. A partir de entonces se inicia un ciclo de lectura de caracteres que finaliza con la entrada del carácter de retorno de carro. Una vez obtenida la cadena de texto, comienza el estado de análisis sintáctico, el cual solo volverá al inicio en caso de mala construcción del comando. Finalmente, si la cadena de texto pasa el proceso de *parsing* se analizará también su semántica y se creará el objeto *Comando*.


Esperando orden
orden recibida
Recibiendo caracteres
caracter
ENTER
Verificando sintaxis
finalizado
[error] / imprime error
[correcto] / resalta sintaxis
Analizando semántica
finalizado
[error] / imprime error
[correcto] / crea comando

Figura 11.20. Diagrama de estados para procesar un comando en Mercurial

caso de estudio: servicio de cifrado remoto

Como se muestra en la figura 11.21, el diagrama de estados modela la recepción de un mensaje por el *socket* oyente y la determinación del algoritmo concreto de cifrado en el servidor. En el primer nodo condicional se procede a detectar si el mensaje está cifrado con un algoritmo simétrico tipo AES, en caso contrario, se prueba a detectar si ha sido cifrado por un algorimto híbrido del tipo RSA-AES.

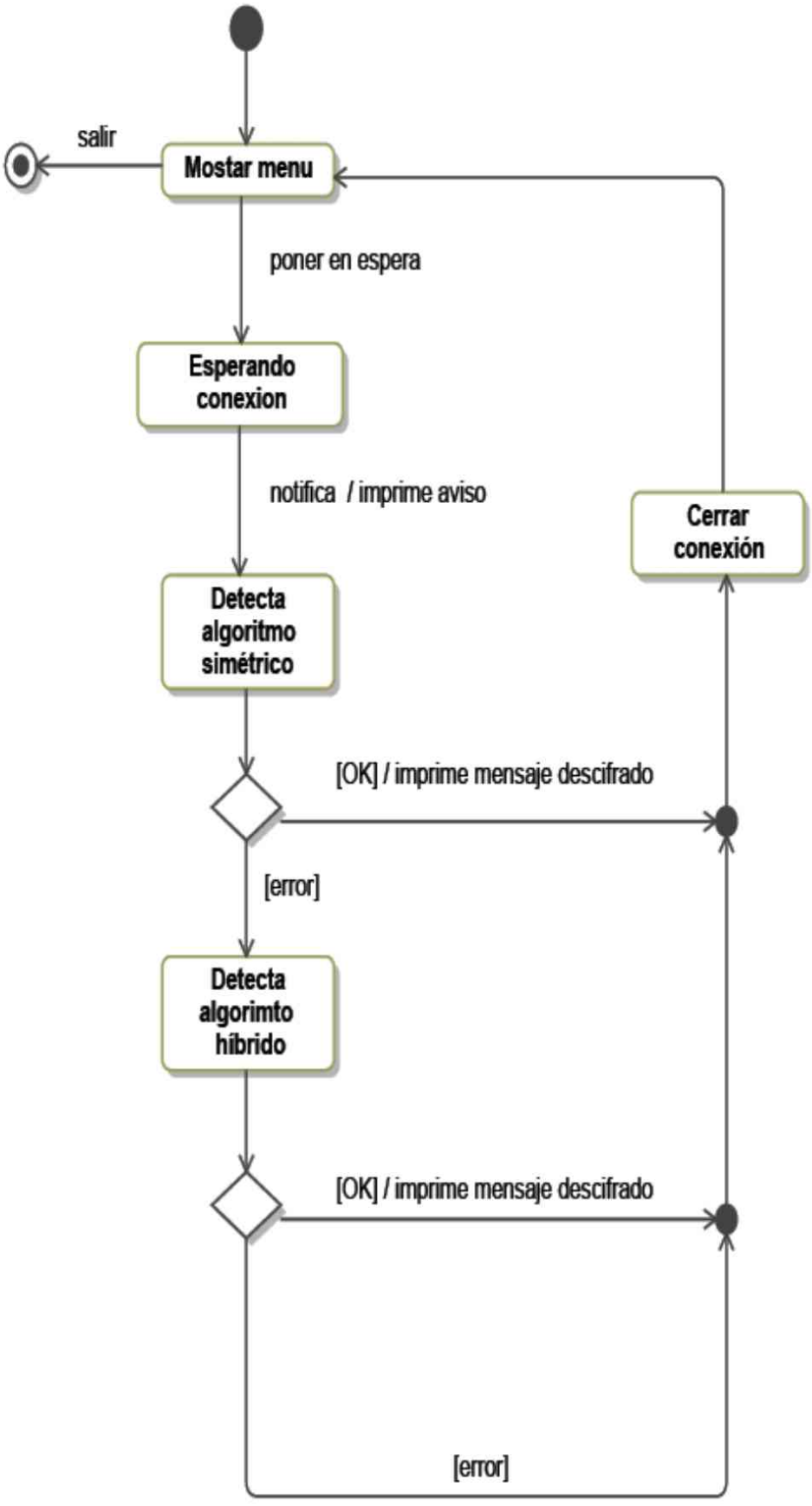


Figura 11.21. Diagrama de estados para detectar algoritmo en el lado servidor

30 El signo " * " representa la repetición del elemento ninguna o muchas veces.

diagramas de actividad

«En vano me esfuerzo por creer que algo tan obvio como dibujar figuras que se complementan mutuamente no se le hubiera ocurrido a alguien antes».

(M. C. Escher).

Los *diagramas de actividades* representen un tipo de diagrama de estados cuyas transiciones no están producidas por eventos externos. El diagrama de actividad es otro ejemplo de modelado del comportamiento en el que los nodos representan acciones que se suceden secuencial o concurrentemente desde un estado inicial a un estado final. Este tipo de diagrama está inspirado en las *redes de Petri* y añade una semántica muy potente para expresar procesos de tiempo real en las aplicaciones diseñadas en UML. En UML 1.x los diagramas de actividad se reducían a una especie de diagramas de estado ampliados, fue con UML 2.0 cuando se creó una sintaxis mejorada y más completa.

estructura básica

La estructura del diagrama de actividades está compuesta básicamente por nodos. Los nodos se clasifican en tres tipos diferentes: *los nodos de acción*, los *de control* y los *de objeto*. Los *nodos de acción* constan de unidades indivisibles de una tarea, es decir, las acciones que se llevan a cabo dentro de un determinado estado. *Los nodos de control* permiten alterar el flujo de ejecución de las actividades, bifurcándolas o redirigiéndolas a otros procedimientos. Finalmente, *los nodos de objeto* representan objetos usados en el escenario de la actividad, entendiendo a estos como instancias concretas de clasificadores.

- **Nodo inicial:** Representa el comienzo de la secuencia del flujo de acciones. Se simboliza mediante un círculo relleno de color negro.
- **Nodos de acción:** Englobanunidades de trabajo atómicas dentro del diagrama.
- **Nodo final:** Representa el estado final donde termina el flujo de una secuencia de acciones.
- **Transiciones:** Muestran el paso de un nodo de acción a otro. En los diagramas de actividad se representan mediante una flecha con punta abierta.

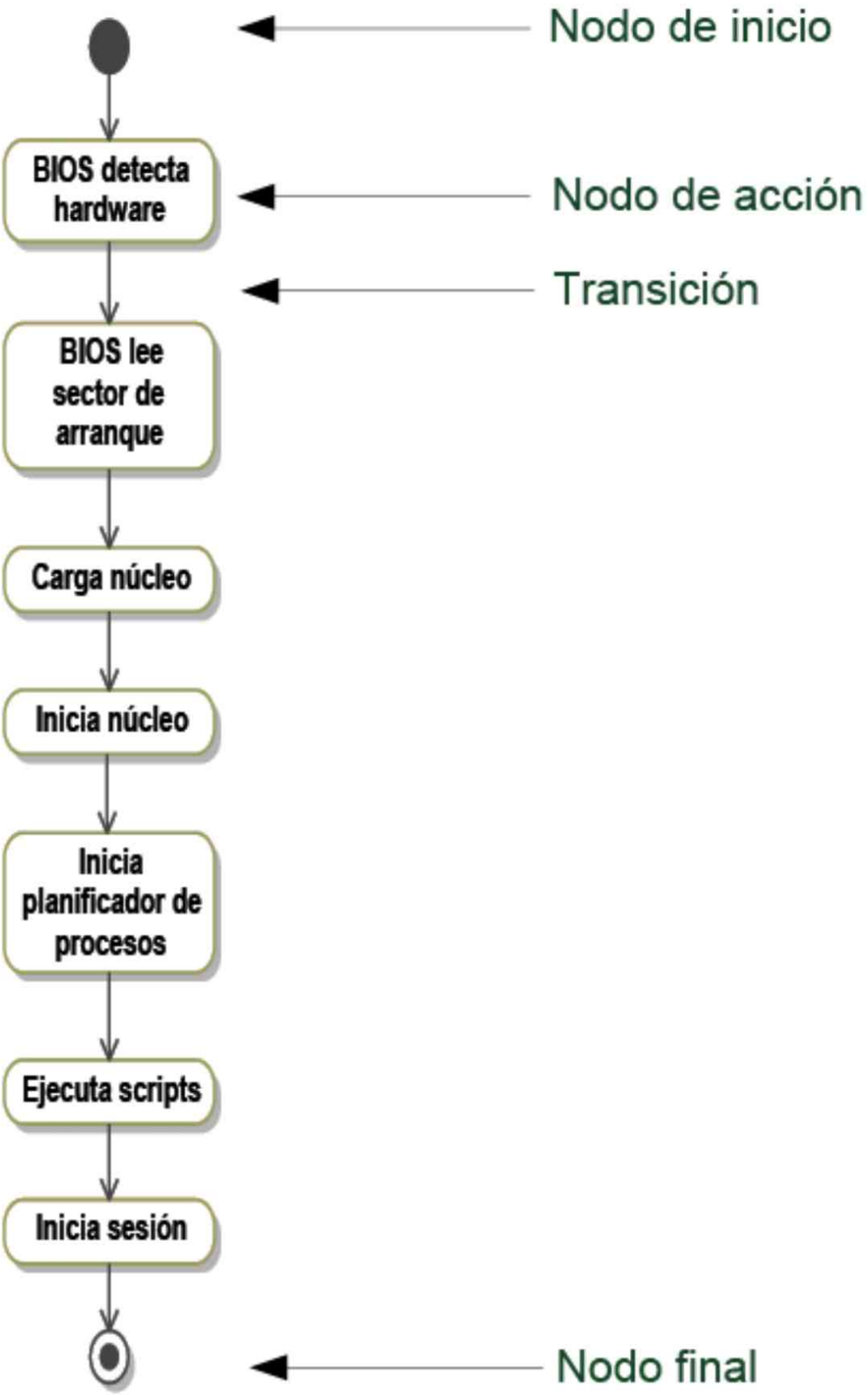


Figura 12.1. Ejemplo básico de un diagrama de actividad

La figura 12.1 es un ejemplo de la estructura del diagrama de actividad. En él se modela el proceso de arranque del sistema operativo UNIX. El proceso se inicia en el punto del nodo inicial marcado como un círculo negro, transita por los diferentes estados de la actividad hasta llegar al nodo final donde se termina dicho proceso.

Además de estas características enunciadas anteriormente es posible asignar opcionalmente restricciones para la ejecución de una acción. Tales restricciones se dividen en *precondiciones* y *postcondiciones* y se ubican en la entrada y en la salida del nodo de acción respectivamente. Las precondiciones hacen referencia a las premisas iniciales que deben satisfacerse para la correcta ejecución de la acción, mientras que las postcondiciones indican los requisitos precisos que deben cumplirse al salir de la acción.

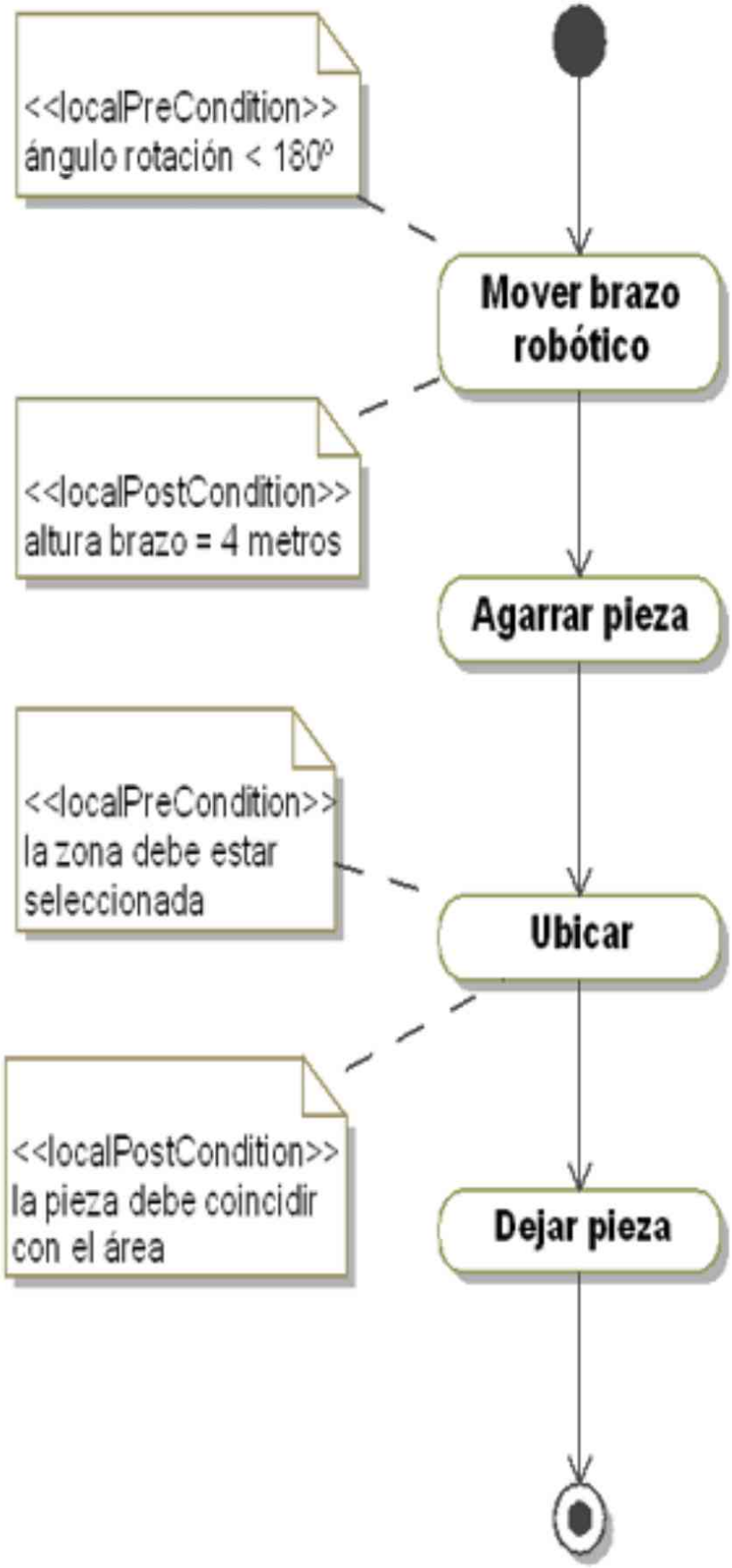

<<localPreCondition>>
ángulo rotación < 180º
Mover brazo robótico
<<localPostCondition>>
altura brazo = 4 metros
Agarrar pieza
<<localPreCondition>>
la zona debe estar seleccionada
Ubicar
<<localPostCondition>>
la pieza debe coincidir con el área
Dejar pieza

Figura 12.2. Ejemplo de uso de restricciones

estructuras de control

Puesto que los diagramas de actividad representan la puesta en escena de un caso de uso y de las entidades que lo conforman, es lógica la existencia de nodos de control para redirigir el flujo de un conjunto de acciones. Se entiende entonces la necesidad de utilización de sentencias de bifurcación y de concurrencia para detallar con precisión la semántica del problema.

Nodos de decisión

Los nodos de decisión permiten bifurcar el flujo de ejecución hacia un conjunto de nodos de acción u otro. Es importante que las guardas de las condiciones sean mutuamente excluyentes, es decir, la semántica de las guardas debe ser lo más clara posible de forma que no lleven a ambigüedades.

Los nodos de decisión en los diagramas de actividades de UML se representan mediante un rombo con tantas flechas de salida como bifurcaciones haya. Los nodos de fusión, de lo contrario, sirven para recoger las diferentes ramas separadas por el nodo de decisión.

Así, en el ejemplo de la figura 12.3, se representa una situación que bien podría estar sacada del contexto de un videojuego clásico. Como se ilustra en el ejemplo, después de partir del nodo de inicio se entra en un primer estado que detecta la pulsación de una tecla. Posteriormente nos encontramos con el primer nodo de decisión que indaga sobre el estado de la pulsación. En caso de que la tecla esté pulsada se intentará empujar la puerta del laberinto, en caso contrario volverá a consultar el estado de la pulsación. Después de pulsar la tecla, la aplicación comprobará si la puerta está cerrada. De ser así, el personaje deberá usar la llave, abrir la puerta y luchar contra el dragón. Si el protagonista vence, el programa presentará la siguiente pantalla del laberinto, de lo contrario el jugador tendrá que combatir en el averno. Finalmente los dos caminos posibles se unirán en el nodo de fusión antes de finalizar la actividad: "*Jugador llega a puerta*".

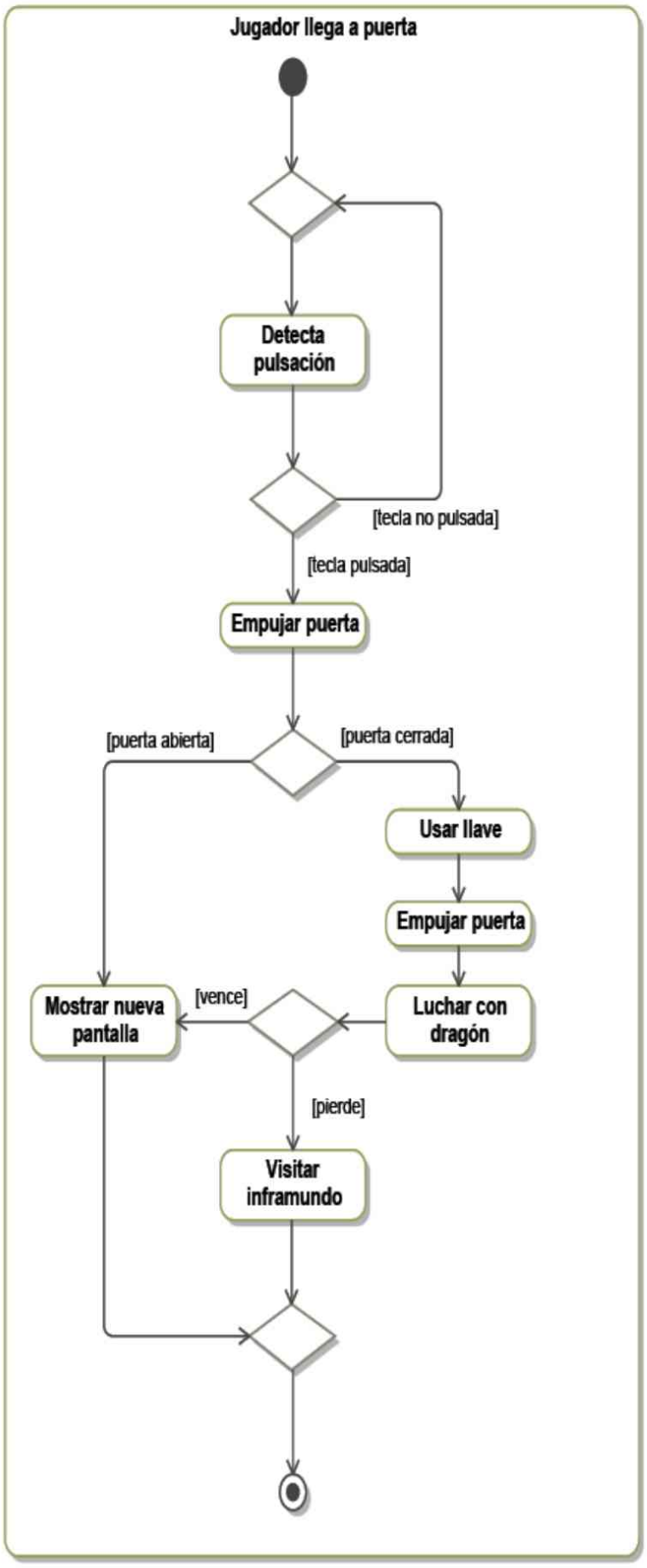

Jugador llega a puerta
Detecta pulsación
[tecla no pulsada]
[tecla pulsada]
Empujar puerta
[puerta abierta]
[puerta cerrada]
Usar llave
Empujar puerta
Mostrar nueva pantalla
[vence]
Luchar con dragón
[pierde]
Visitar inframundo

Figura 12.3. Ejemplo de tres nodos de decisión y uno de fusión

Nodos de concurrencia y paralelismo

También pueden existir situaciones donde el flujo de acciones deba expandirse en varios hilos concurrentes o paralelos, creando un escenario de varios conjuntos de acciones ejecutándose al mismo tiempo. Para indicar que una actividad se descompone en varios conjuntos de acciones concurrentes o paralelas, utilizaremos una barra negra desde donde parten flechas (nodo *fork*) a cada uno de los conjuntos de estados de actividad. De igual forma, los diferentes hilos se unifican en un nodo de fusión llamado barrera de sincronización (nodo *join*).

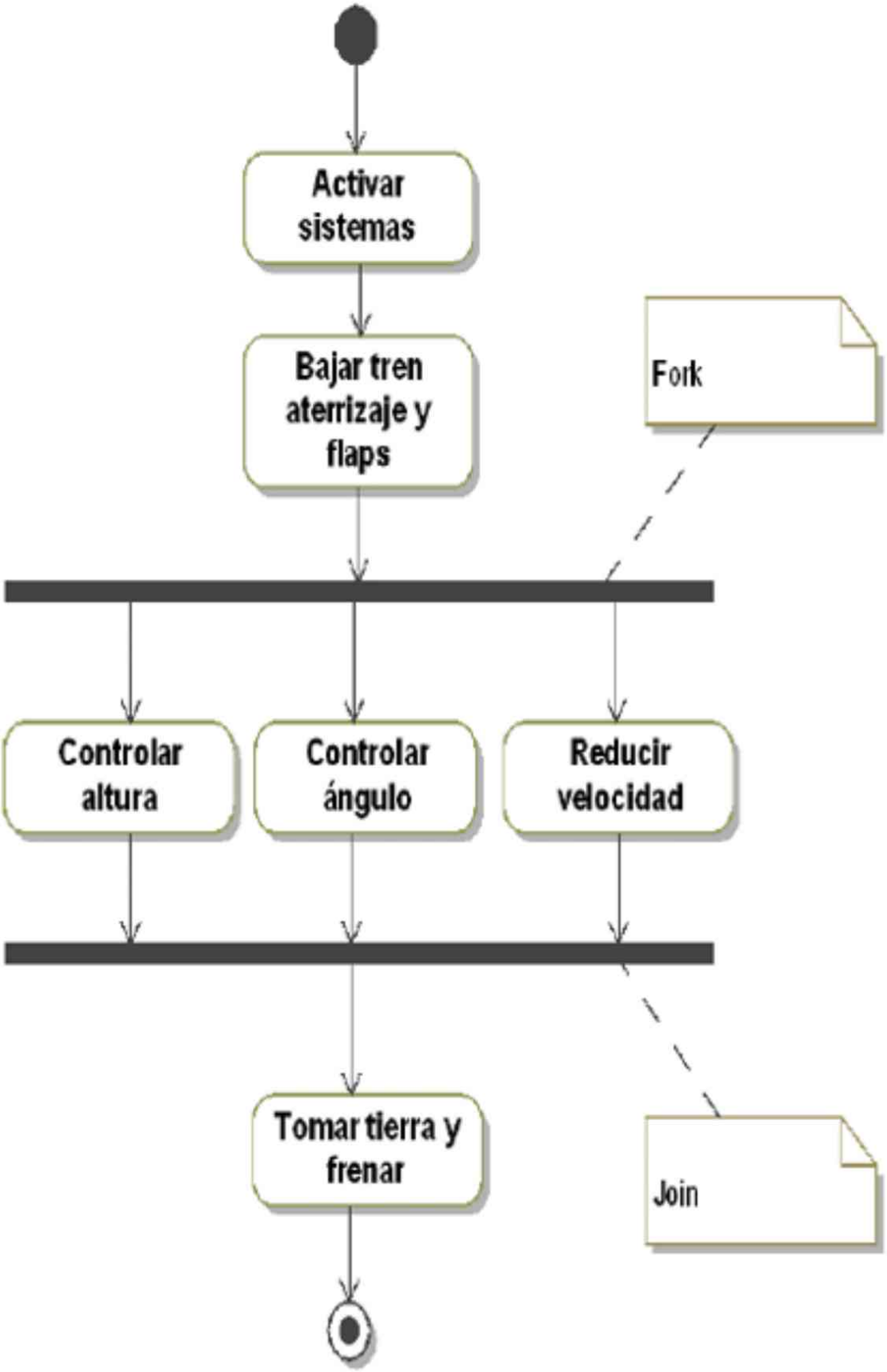


Figura 12.4. Paralelismo en el aterrizaje de un avión

El ejemplo de la figura 12.4 muestra el escenario de ejecución del proceso de aterrizaje de un avión. Después de bajar el tren de aterrizaje se activan tres acciones paralelas durante el descenso de la aeronave que finalizan en el momento de toma de tierra. Para ilustrar esta situación se dibujan las barras de inicio y finalización de paralelismo (nodos *fork* y *join*).

eventos de tiempo

Los nodos de tiempo simulan las situaciones donde es necesario esperar una cantidad de tiempo para continuar el flujo de acciones. La expresión asociada con el icono del evento de tiempo puede especificar la demora en forma de cifra o textualmente.

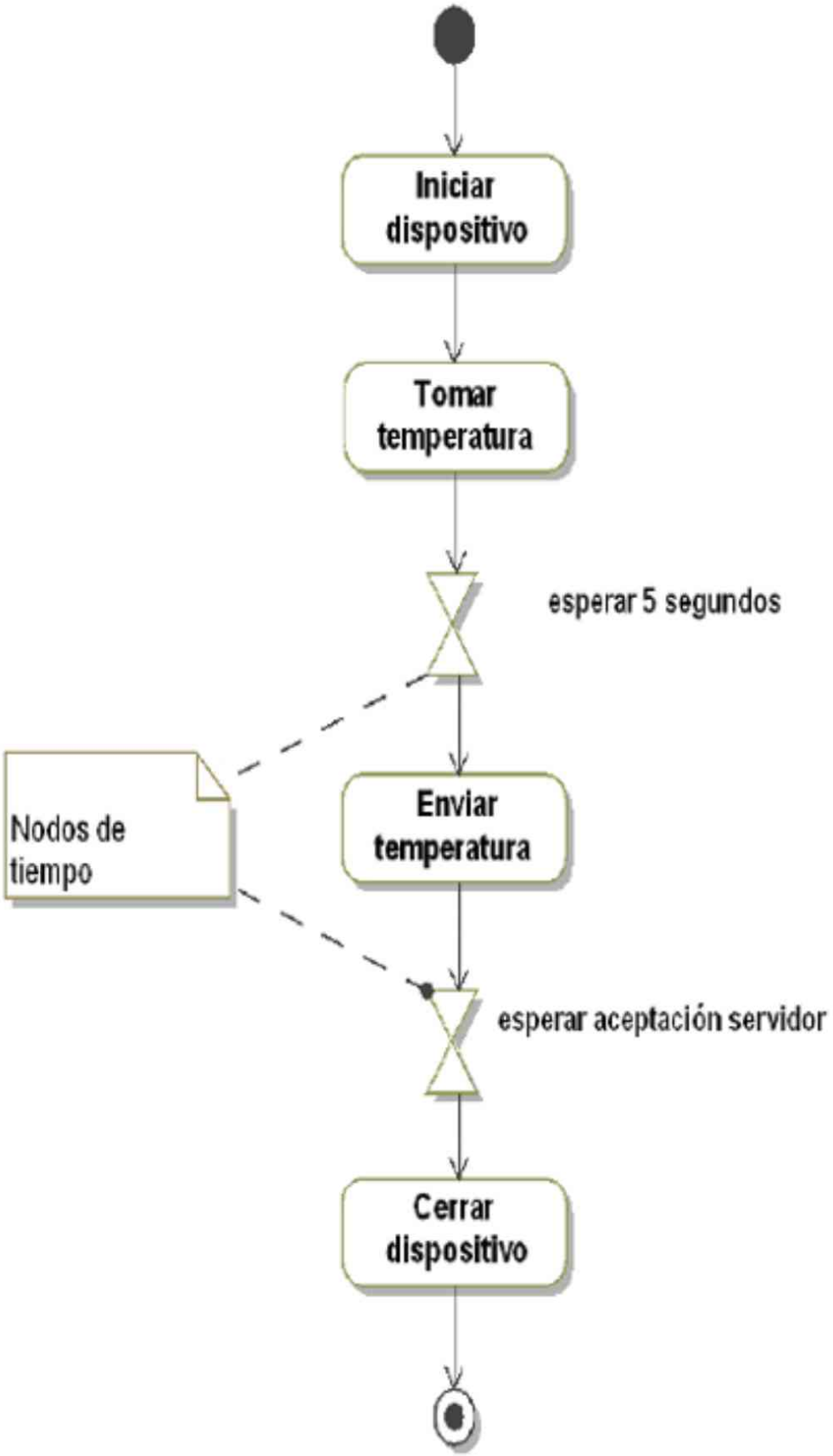


Figura 12.5. Diagrama con dos eventos de tiempo

La figura 12.5 ilustra un ejemplo de utilización de dos eventos de tiempo UML: uno cuantitativo y otro cualitativo. El diagrama modela un sistema de telemetría que espera 5 segundos antes de enviar la temperatura a un sistema centralizado. Después de enviar la información al sistema remoto, espera la confirmación de éste.

nodos objeto

Como veremos más adelante, los nodos objetos se utilizan en los diagramas de actividad para indicar que las instancias de clasificadores provenientes del modelo de negocio son referenciados en el diagrama de actividad para utilizarlos como entrada o salida de un determinado estado. Dichos objetos suelen ser generalmente instancias de clases. Los objetos se representan como rectángulos con el nombre del clasificador. Opcionalmente pueden llevar asociado un estado aclarativo en el que se encuentra dicho objeto.

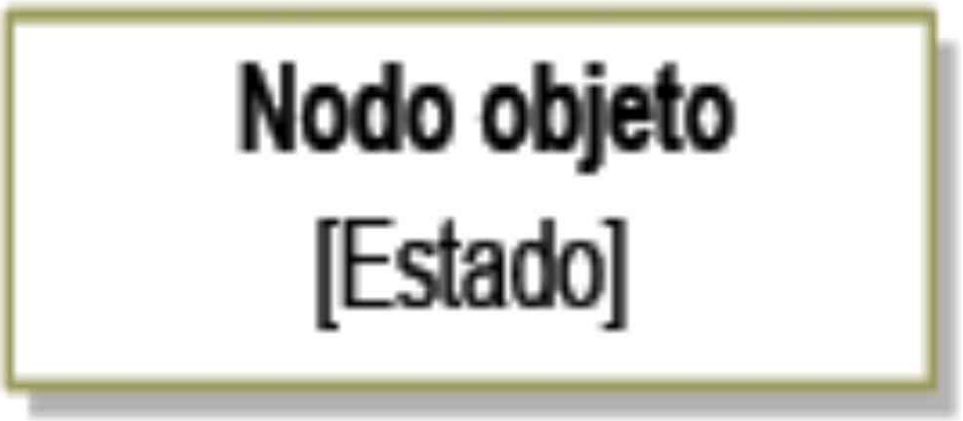


Figura 12.6. Ejemplo de nodo objeto con estado

nodos de datos

Los nodos de datos son un tipo especial de nodo que permite especificar una base de datos o un soporte lógico de almacenamiento. En general es una manera de especificar un nodo de almacenamiento de información persistente en contraposición a los otros nodos donde la información es temporal. Los nodos de datos se representan de igual forma que los nodos objeto pero con el estereotipo <<datastore>>.

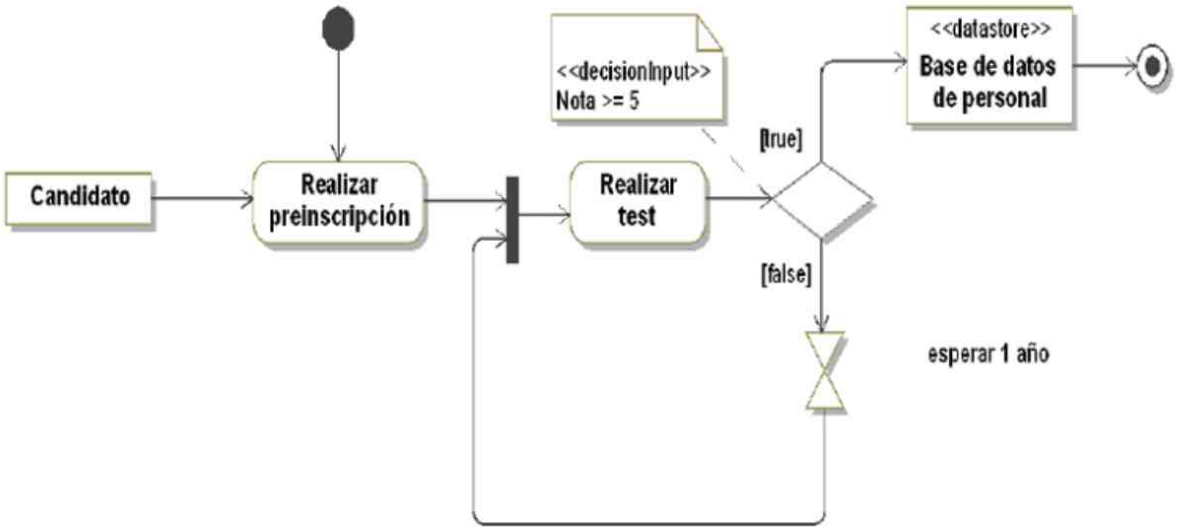


Figura 12.7. Ejemplo de uso de un nodo de datos

La figura 12.7 muestra la aplicación de un nodo de datos. Cuando los flujos concurrentes de recepción de la preinscripción y el transcurso de un año confluyen se realiza un test para evaluar al candidato. En caso de ser aprobado se almacena en la base de datos y la actividad finaliza. En caso contrario vuelve a esperar un año a otro candidato. En este diagrama se introduce la noción de nodo objeto con "Candidato".

particiones

Las actividades son realizadas por personas, procesos o clases. Una forma de indicar la responsabilidad de estos actores es la utilización de carriles ("swimlanes"), como se puede apreciar en la figura 12.8:

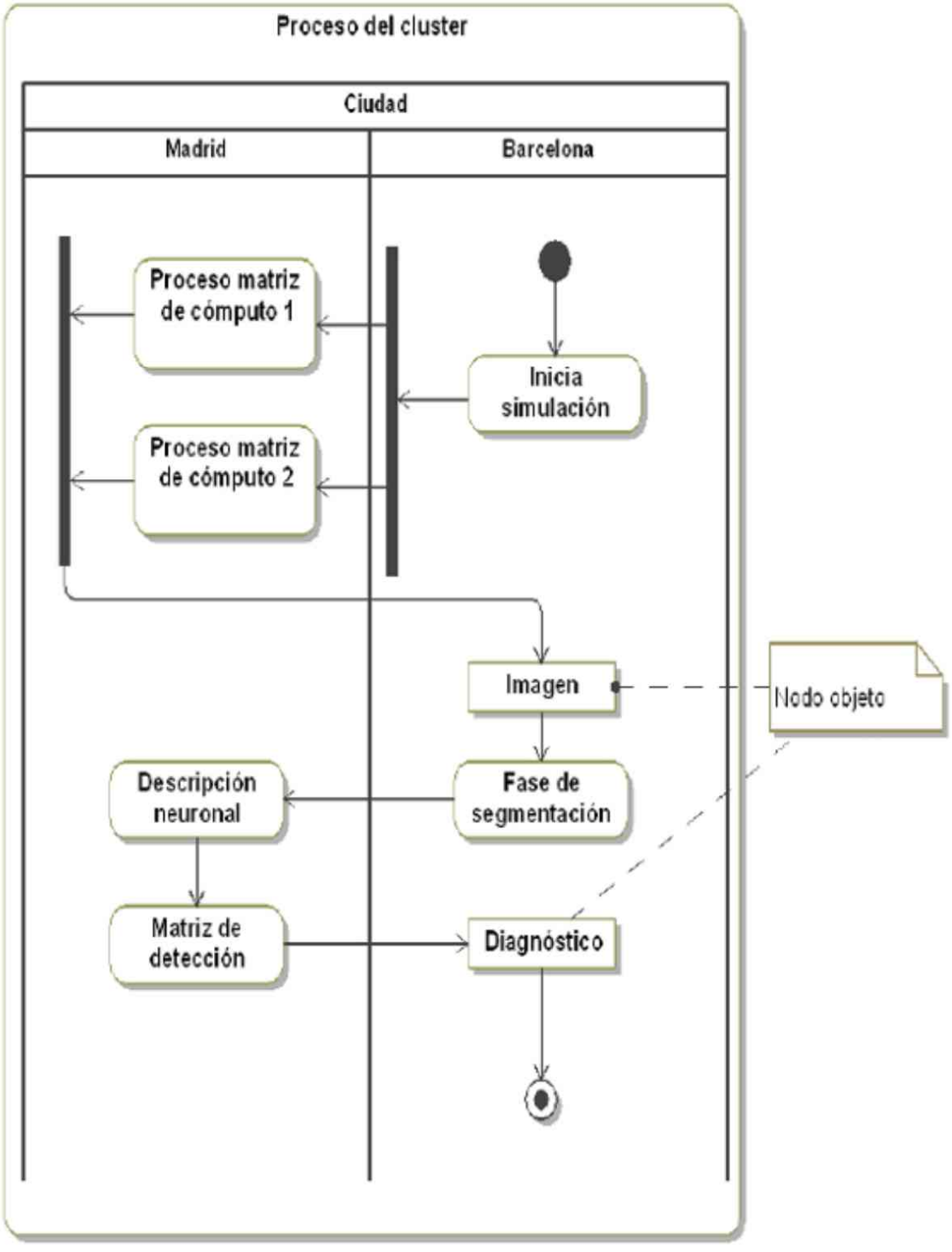


Figura 12.8. Ejemplo de actividad partida

En el ejemplo de la figura 12.8, un proceso de diagnóstico basado en imagen requiere que un ordenador ubicado en Barcelona inicie una simulación de un proceso orgánico humano (por ejemplo, una simulación del cerebro) en dos matrices de cómputo en un *mainframe* situado en Madrid. El resultado de la simulación se devolverá en un objeto *Imagen* que se procesará en Barcelona para obtener diferentes regiones y enviarlo de nuevo a Madrid para un procesamiento neuronal en una matriz de multiprocesadores. Finalmente el resultado del diagnóstico se devuelve en un objeto antes de la finalización.

parametrización

En los diagramas de actividades es posible utilizar nodos objeto como entradas y salidas de los procesos. De esta forma se consigue una estructura modular de actividades que pueden ser utilizadas en diagramas más complejos. Para especificar que una actividad tiene un conjunto de objetos de entrada y/o de salida se dibujan los objetos en el borde del superconjunto que representa la actividad, tal como se muestra en el siguiente ejemplo:

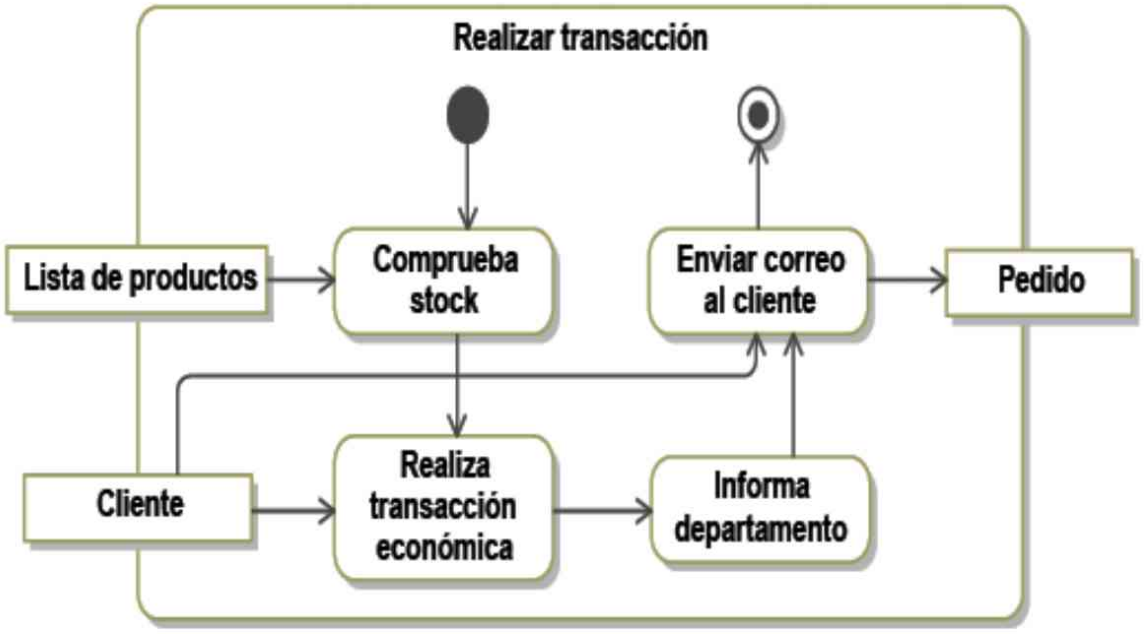


Figura 12.9. Actividad con dos entradas y una salida

Ahora bien, cuando el número de objetos entre actividades se incrementa en exceso es necesario recurrir a una notación que permita simplificar la parametrización. Con este fin el lenguaje UML proporciona un nuevo símbolo denominado *pin*. Los pines facilitan la parametrización de objetos de entrada salida mediante un cuadrado adyacente al conjunto de la actividad (véase figura 12.10) que indicará, según la dirección de las transiciones, si es de entrada o de salida.

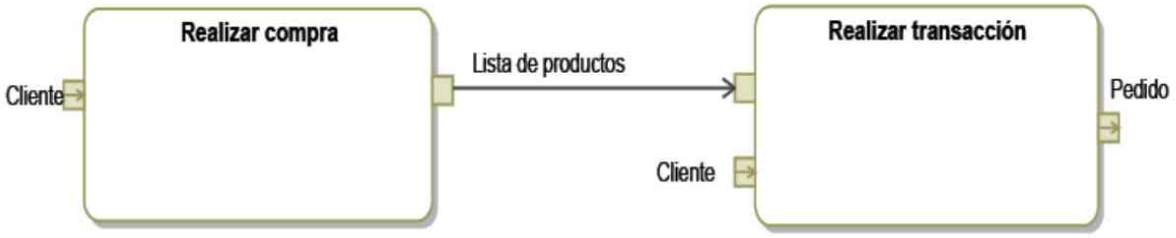


Figura 12.10. Ejemplo de dos actividades interconectadas por pines

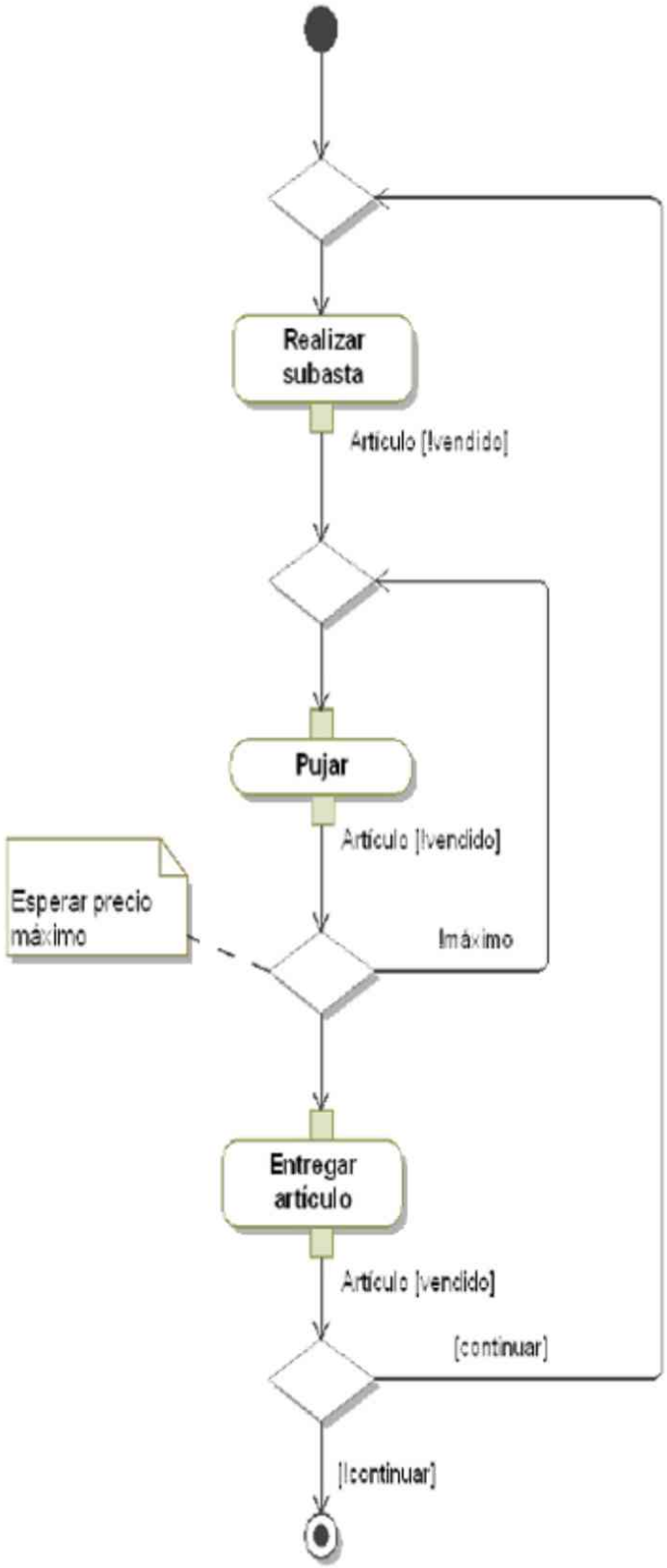


Figura 12.11. Ejemplo de pines con estado

Al utilizar pines es posible que el diseñador requiera especificar el estado en

el que se encuentran en las entradas y salidas de las acciones (ver figura 12.11).

regiones

Regiones de expansión

A veces es útil procesar conjuntos de objetos (colecciones) en vez de un objeto individualmente. UML 2.x tiene una gran variedad de posibilidades para especificar esta situación. Cuando utilizamos una *región de expansión* estamos considerando que dicha acción contiene una colección de objetos de entrada o de salida llamada *nodos de expansión*. De hecho los nodos objeto vistos en la sección 12.4 pueden considerarse opcionalmente como un buffer o array que contiene elementos u objetos de un determinado tipo. Así mismo, los nodos de expansión de entrada y salida pueden diferir en cuanto a tamaño y tipo de datos. Finalmente una región de expansión se ejecutará tantas veces como elementos haya en el nodo de expansión de entrada.

Existen diferentes alternativas para los modos en que puede ejecutarse un nodo de expansión. En UML 2.x existen actualmente tres:

Modo (estereotipo)	Significado
<<iterative>>	Ejecuta cada elemento de entrada de la colección secuencialmente.
<<parallel>>	Ejecuta los elementos de la colección de entrada en paralelo.
<<streaming>>	Procesa los elementos de entrada tan pronto como sean recibidos.

Tabla 12.1. Modos de procesamiento de nodos de expansión

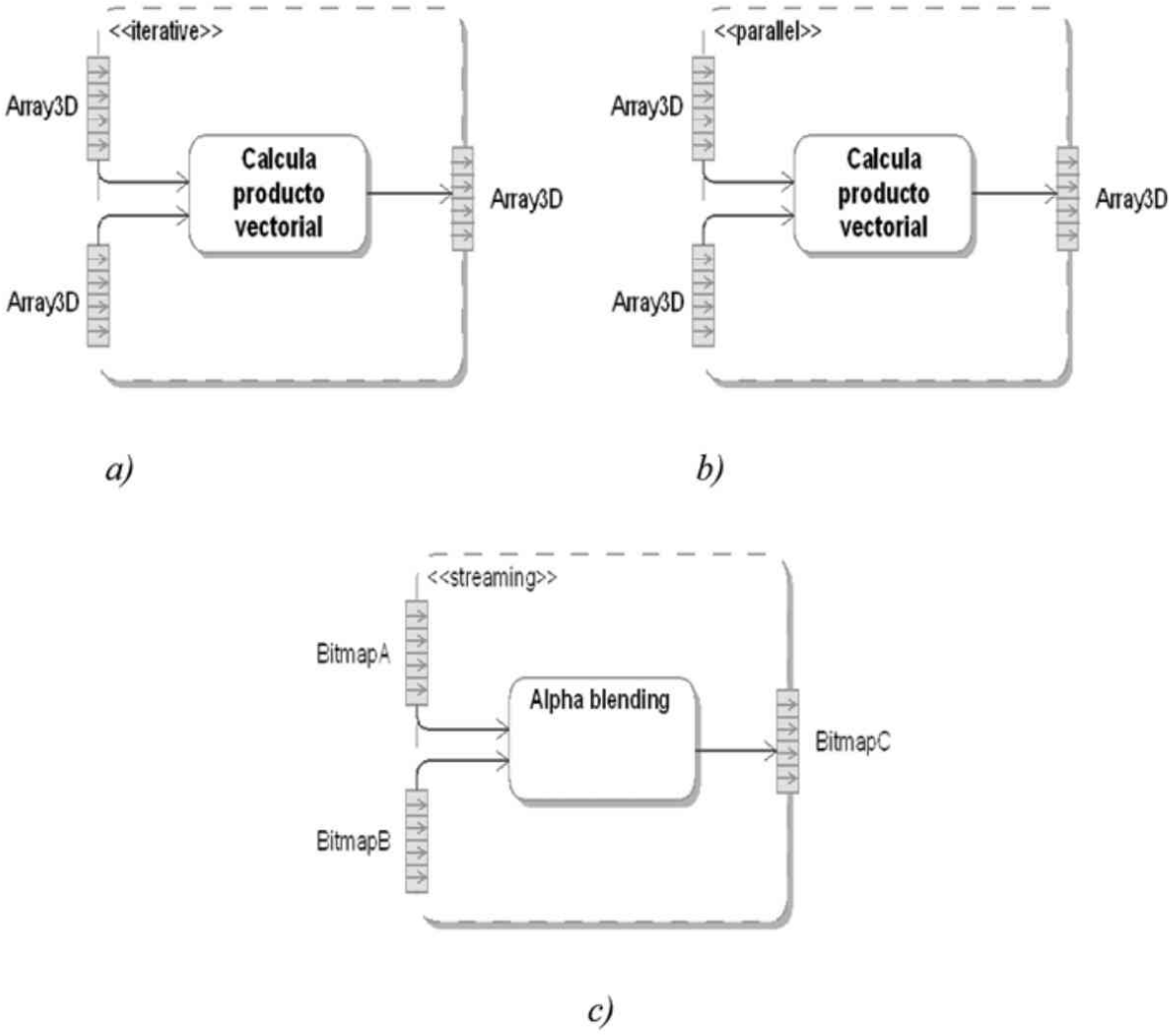


Figura 12.12. Ejemplos de los tres modos de las regiones de expansión

En los ejemplos a) y b) de la figura 12.12 se reciben dos arrays de *n* elementos con vectores 3D para realizar el producto vectorial y devolver otro array de *n* elementos con el resultado. En el caso a) se realiza el determinante por cada elemento de forma secuencial, es decir, uno a uno; mientras que en el b) se lleva a cabo de forma paralela. Finalmente, en el caso c) un algoritmo de mezclado por canal alfa (*alpha blending*) procesa en *streaming* dos arrays de datos RGB (*bitmap*) hacia otro bitmap de salida con la imagen ya mezclada. *Streaming* significa que procesa la información en un flujo continuo de bytes de acuerdo a un paradigma *productor/consumidor* en tiempo real.

Regiones interrumpibles

En los diagramas de actividad es posible especificar situaciones que son interrumpidas por eventos generados por otras acciones. Cuando ocurre un evento de interrupción desde dentro de una *región interrumpible* se finaliza el proceso que especifica la actividad conjunta. Esta característica permite modelar excepciones y escenarios asíncronos en el flujo de ejecución de las acciones.

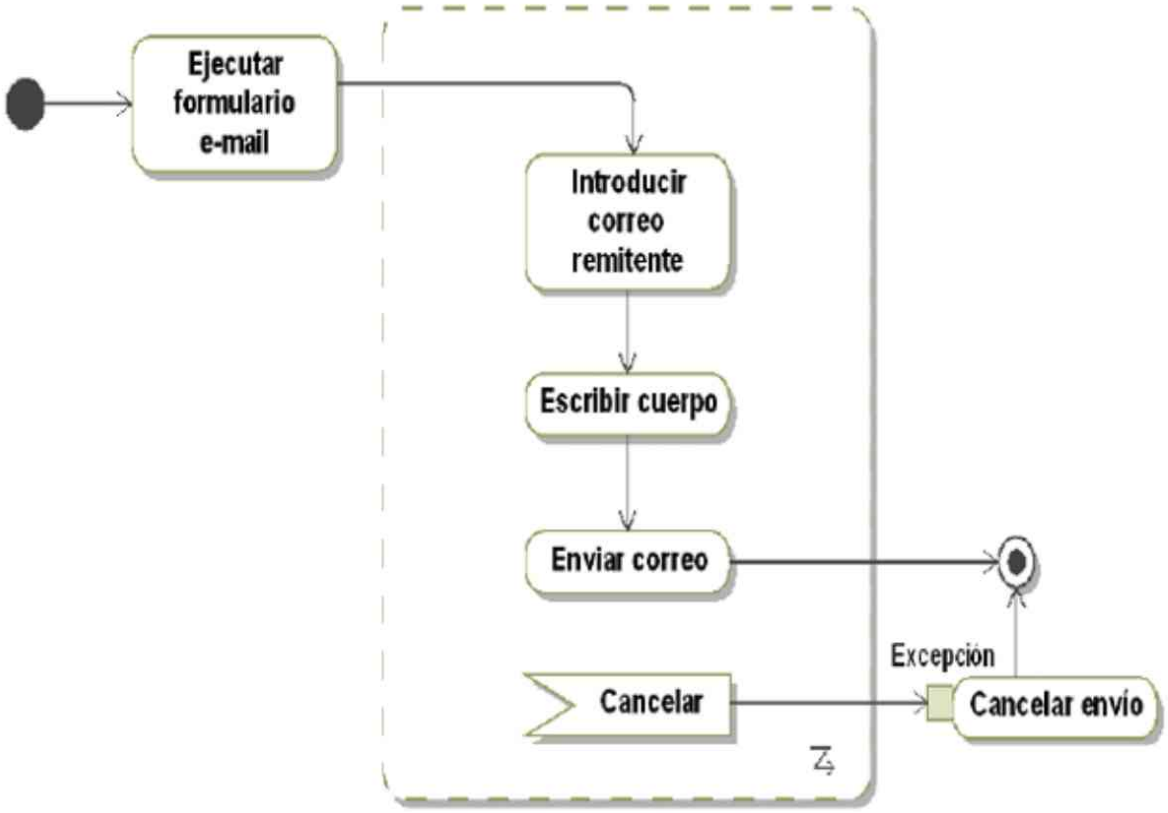


Figura 12.13. Ejemplo de región interrumpible en un formulario Web

En el ejemplo de la figura 12.13 la actividad comienza cuando el usuario invoca una página HTML con un formulario para enviar un correo electrónico corporativo. Cuando entra dentro de la región interrumpible, el usuario puede indicar su dirección de correo y escribir el cuerpo del mensaje. A menos que se dispare el evento "cancelar", el proceso continuará hasta enviar el correo por la red. Para especificar que existe una posibilidad de interrupción es necesario indicarla dentro del área de la región interrumpible. La

acción que se encarga de agrupar la interrupción y generación de la excepción se denotan mediante un rectángulo acabado en dos puntas. De ahí parte la transición a la acción de manejo de la excepción.

Regiones “if”

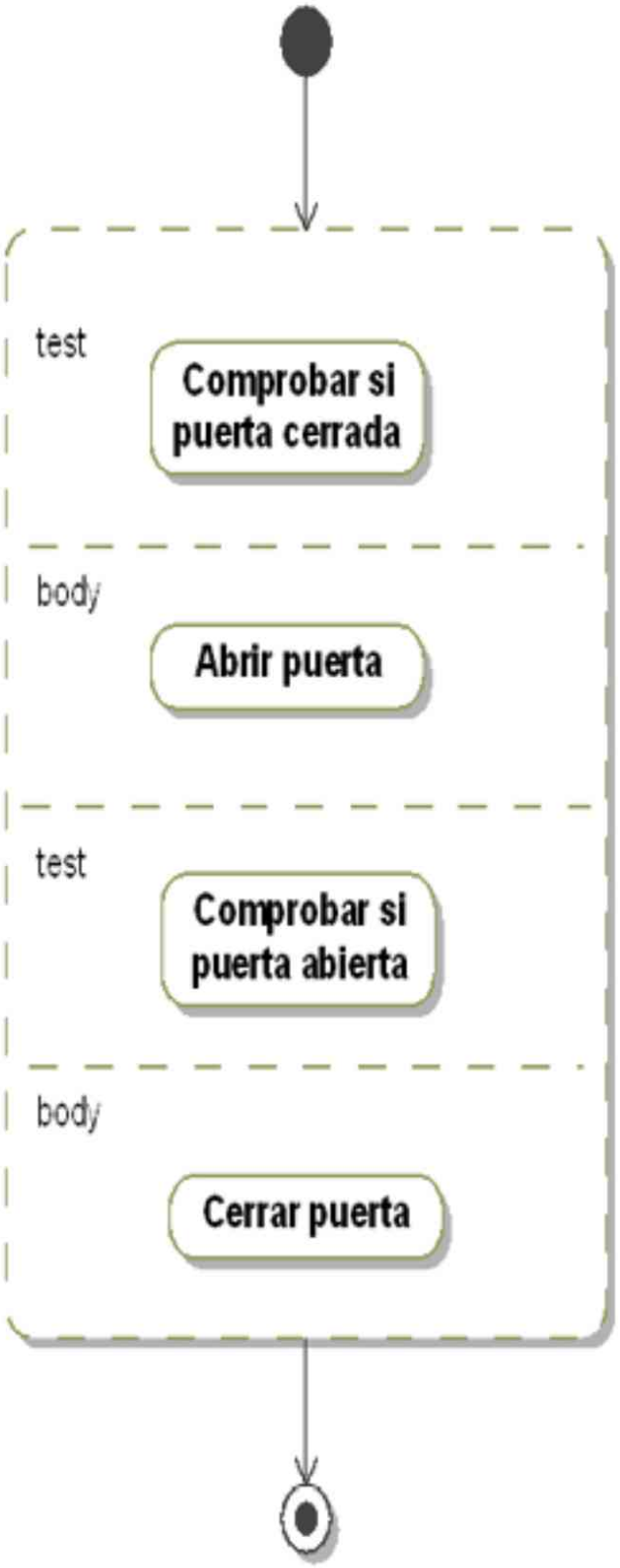


Figura 12.14. Ejemplo de región “if”

Las regiones “if” simplifican la complejidad que supone utilizar una gran cantidad de nodos condicionales. Mediante esta región es posible especificar una larga cadena de acciones sujetas a varias condiciones. Las partes principales de la región son:

- **Test**: Guarda especificada por una o varias acciones interrelacionadas.
- **Body**: Sección principal de acciones a llevar a cabo cuando se cumple la condición.

Regiones “loop”

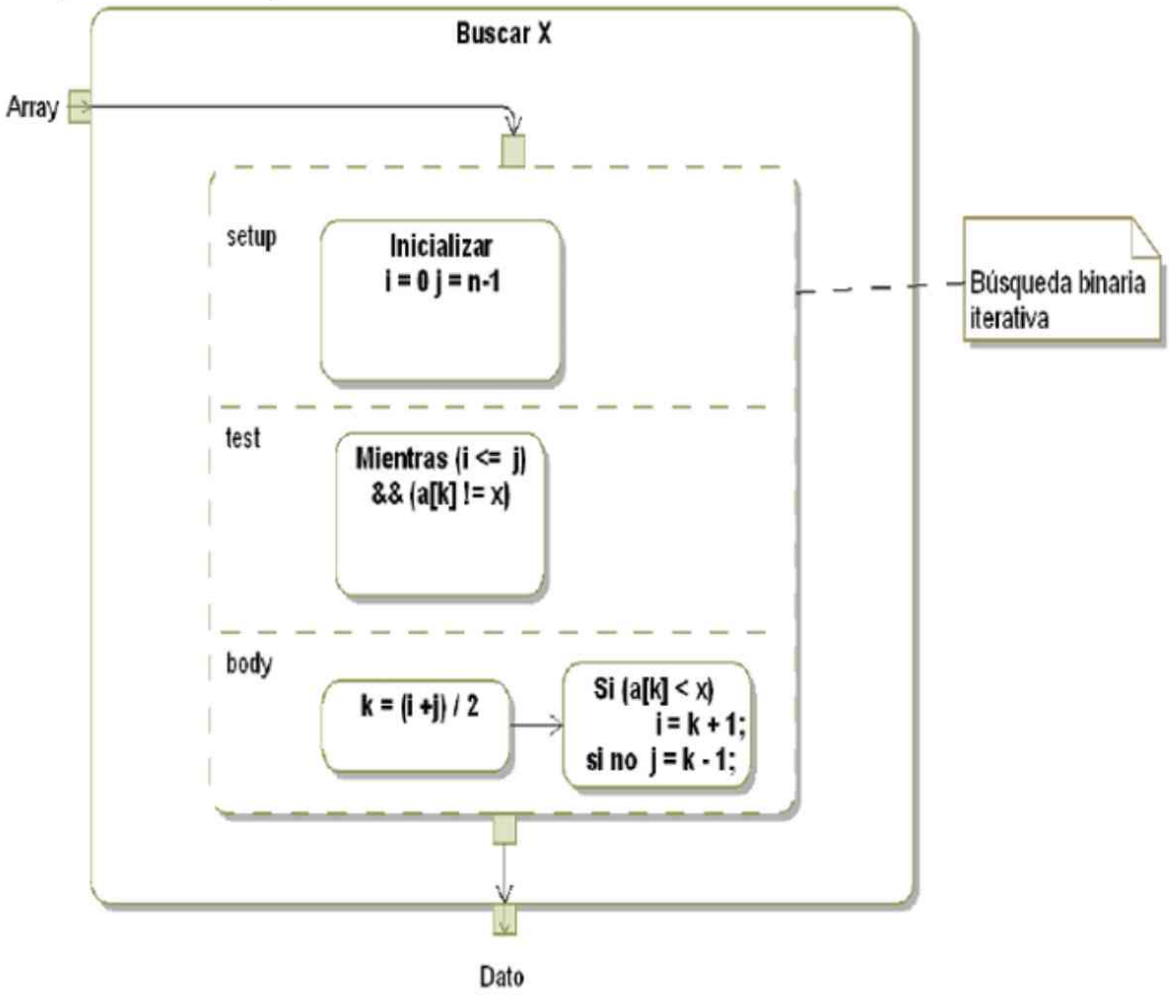


Figura 12.15. Región” loop”

Como se aprecia en la figura 12.15, las regiones *loop* permiten utilizar estructuras de control anidadas para especificar un procesamiento iterativo de las acciones sobre sus objetos de entrada. Las principales secciones de la región son:

- **Setup**: Configura las variables o el estado de inicio.
- **Test**: Guarda de comprobación del bucle.
- **Body**: Grupo de acciones a realizar durante las iteraciones.

Obviamente, en cada sección pueden ubicarse diferentes acciones

relacionadas.

En general, las regiones “loop” e “if” deben ser evitadas en la medida de lo posible, pues los diagramas de actividades no deben modelar a tan bajo nivel.

Manejo de excepciones

Para modelar situaciones de generación de excepciones no es necesario recurrir a una región interrumpible. Para especificar que una actividad realiza una gestión de excepción tan solo es necesario utilizar la transición de generación de interrupción y un *pin* de entrada en la acción receptora.

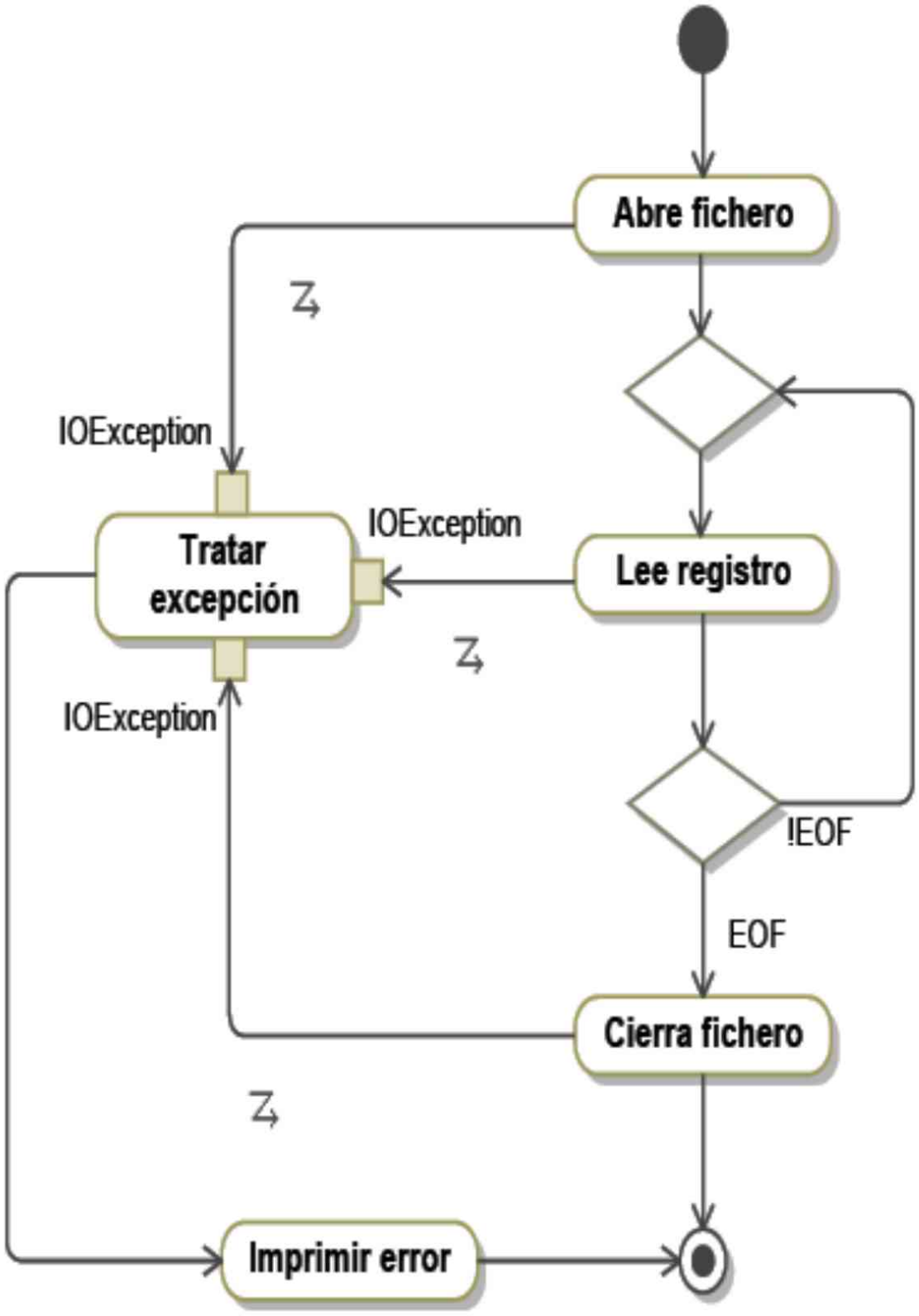


Figura 12.16. Lectura de fichero y contexto de manejo de excepciones

El ejemplo de la figura 12.16 es un típico caso de manejo de excepciones en

Java, aunque es idéntico en C++ y en Python. La lectura de ficheros implica la posibilidad de situaciones anómalas por el estado del archivo: protección contra escritura, error de procesamiento de registro, etc. En consecuencia, esta notación simplifica y aumenta la legibilidad del diagrama considerablemente al reducir elementos excesivamente detallistas.

conectores

Al igual que los *pines*, los conectores pueden ser una medida de reducción de la complejidad de los diagramas de actividades. Utilizar conectores implica especificar que un flujo de acciones continúa en otra parte del diagrama, es decir, permite descomponer el diagrama por medio de etiquetas de llamada. Aunque esta notación puede parecer inicialmente útil para evitar confusión cuando las líneas de transición se prolongan demasiado y enredan el diagrama, en general es mejor evitarla en la medida de lo posible.

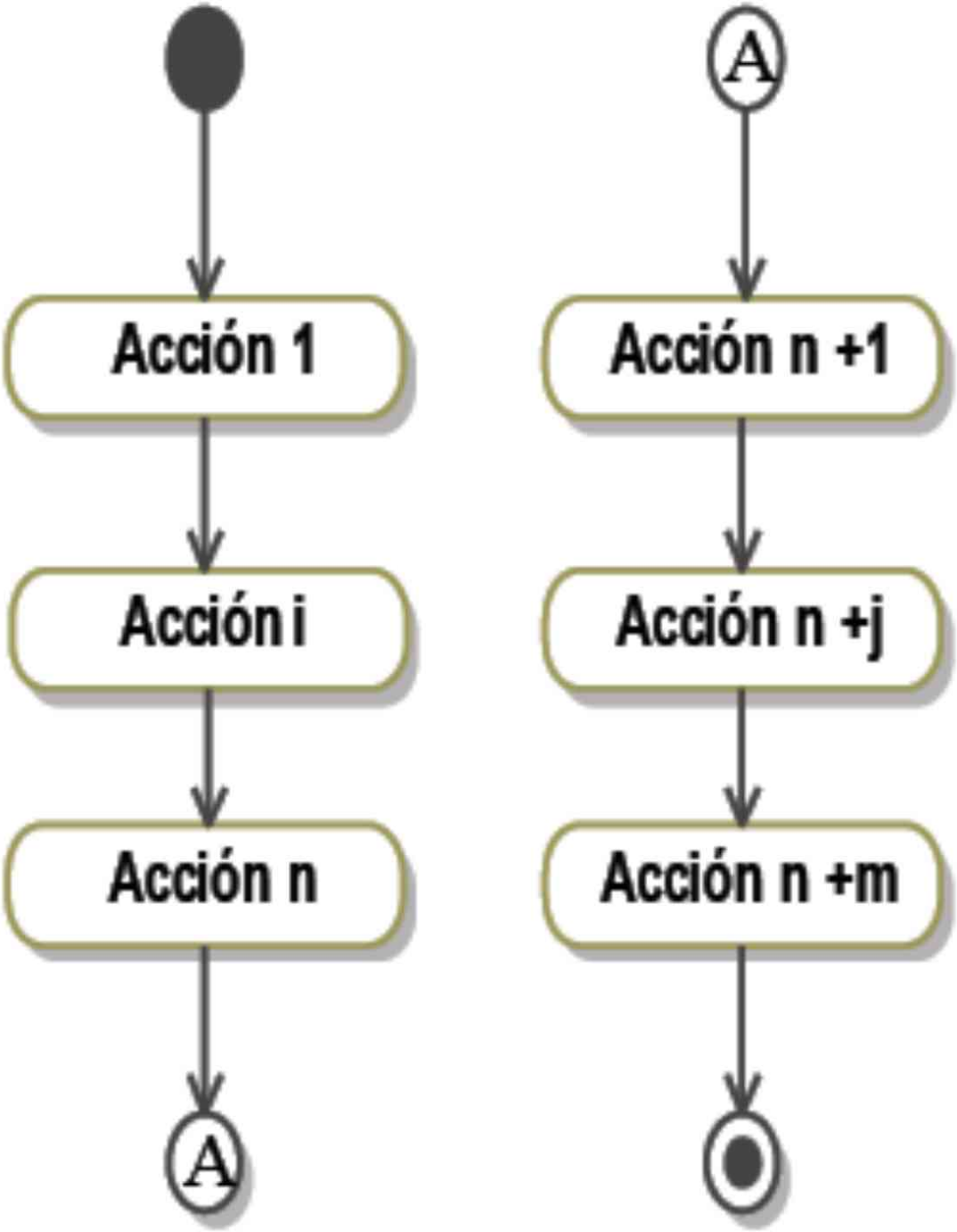


Figura 12.17. Las actividades 1 a n continúan en la indicación de la etiqueta

señales y eventos

Una de las ventajas de los diagramas de actividades es la posibilidad de representar llamadas entre acciones de forma asíncrona. Cuando esto ocurre, un nodo de envío de señal invoca a un nodo de aceptación de evento para continuar el flujo asíncronamente. El envío de una señal implica la posible creación de varios flujos concurrentes que se ejecutan de manera independiente.

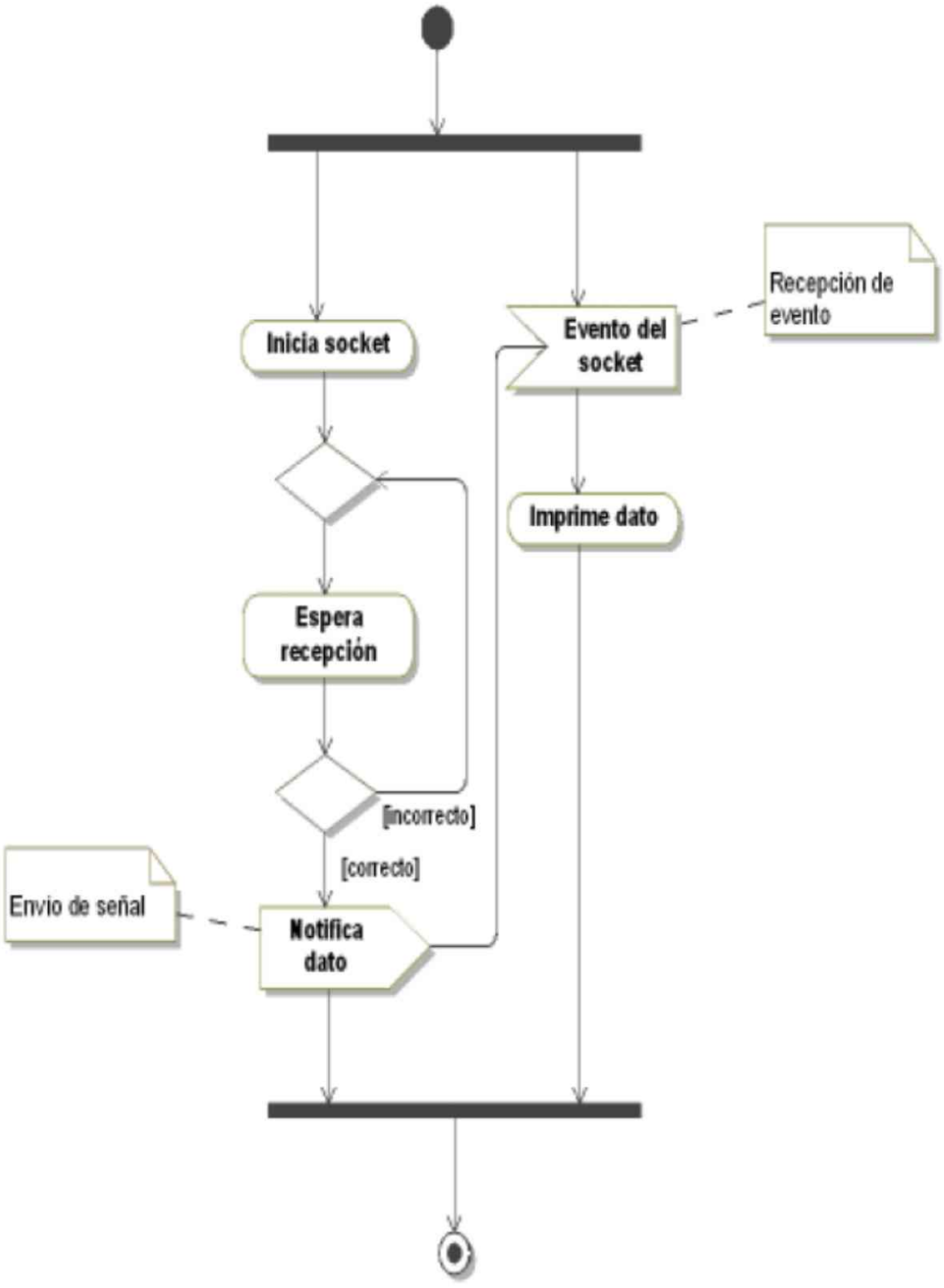


Figura 12.18. Ejemplo de señal y evento

En el ejemplo de la figura 12.18 se realizan las siguientes acciones:

- Se crean dos hilos concurrentes: uno para crear el *socket* de recepción y otro para recibir el evento del *socket*. Los dos comienzan a ejecutarse simultáneamente.
- El primer hilo (el de la izquierda) crea el *socket* y ejecuta la primitiva bloqueante de recepción. Cuando recibe el dato correcto, activa el nodo de señalización, en caso contrario vuelve a esperar otro dato.
- El segundo hilo (el de la derecha) queda en espera hasta la recepción del evento de llegada de dato correcto por el *socket*.

múltiples flujos

Es posible darse situaciones de paralelismo, concurrencia o señalización donde varios flujos divergen del flujo principal, creando de esta forma varios caminos alternativos. Así mismo, los nodos condicionales son factibles también de crear varios flujos. Cuando se derivan varios flujos de acciones es necesario finalizarlos independientemente mediante un *nodo de final de flujo*. Dicho nodo final se representa con un aspa inscrita en un círculo (véase figura 12.19). Evidentemente, la activación de un nodo final de flujo no afecta a los demas flujos de la actividad.

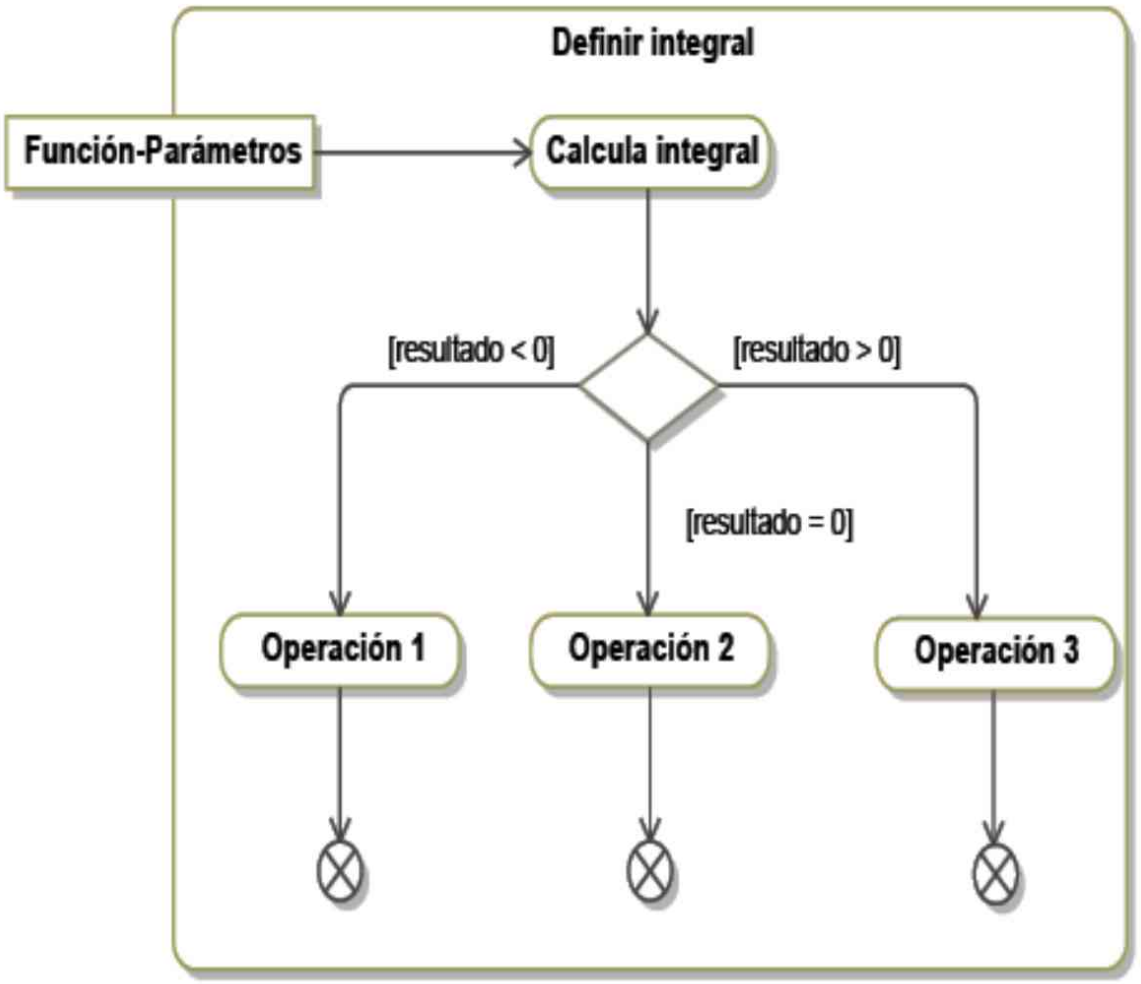


Figura 12.19. Varios flujos a partir de un nodo condicional(al finalizar un flujo los otros no se alteran)

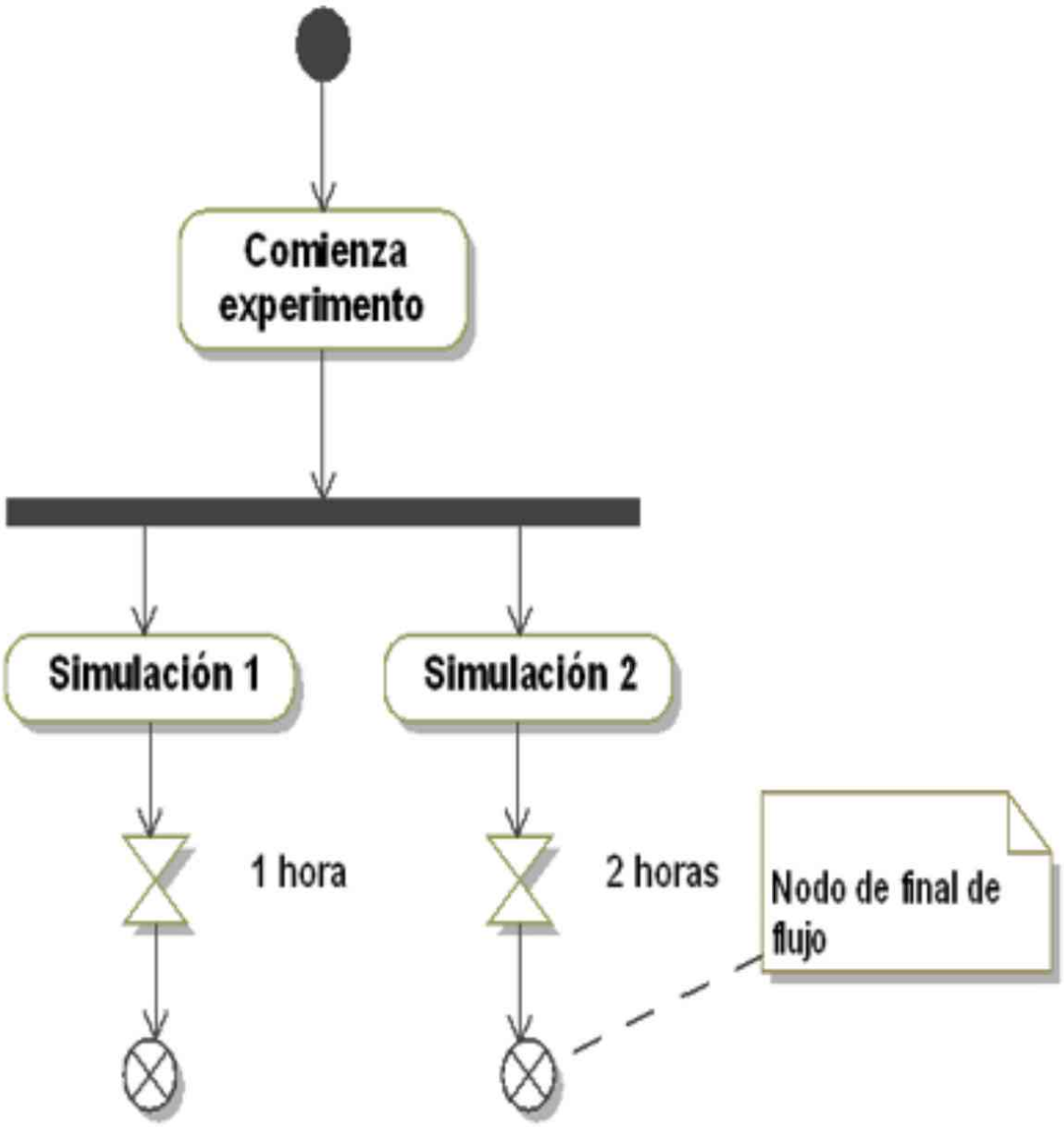


Figura 12.20. Dos flujos concurrentes finalizados independientemente

Es importante saber que si ejecutamos un *nodo de final de actividad* ◉ se finalizarán automáticamente todos los otros flujos que se encuentren activos.

streaming

Normalmente pueden existir situaciones donde es necesaria la comunicación de información entre dos acciones de forma continuada. Esta circunstancia es típica de *software en tiempo real* como aplicaciones multimedia y de procesamiento de señal. UML 2.x proporciona una gran variedad léxica para especificar estas situaciones. En este libro en concreto utilizaremos la palabra clave {stream} para indicar un flujo continuo de información. En el ejemplo de la figura 12.21 podemos observar un fragmento de procesamiento de información por *streaming* de radio. En este ejemplo la actividad lee paquetes de un buffer que ha recibido previamente información desde Internet utilizando el protocolo RTP (*Real-time Transport Protocol*). A partir de aquí se transfieren repetidamente los *tokens* de eventos RTP/MP3 hacia las acciones de decodificación y de reproducción de sonido dentro de su *thread*.

En general, cualquier escenario que requiera de un flujo continuado de recepción y procesamiento de información es candidato a modelarse mediante *streaming*.



Figura 12.21. Ejemplo de streaming para una radio por Internet

multicasting

En los ejemplos que hemos visto hasta ahora un emisor simplemente enviaba objeto/s a un solo receptor. En ocasiones puede ser útil especificar que un emisor puede enviar un objeto a múltiples receptores (*multicast*) o que un receptor reciba de múltiples emisores (*multireceive*). Para especificar esta situación utilizaremos una notación similar a la usada para el streaming.

En el ejemplo de la figura 12.22 un sistema de noticias envía periódicamente informaciones a un servidor de *News* que las difunde mediante *multicast* a un conjunto de clientes (subscriptores) que a su vez envían de vuelta el mensaje de confirmación al servidor. El uso del estereotipo <<multicast>> y <<multireceive>> en este ejemplo facilita la multidifusión entre las dos secciones ubicadas en distintos lugares. Los objetos enviados son replicados hacia varios clientes subscriptores que a su vez pueden reenviar la confirmación de vuelta al recolector de mensajes del emisor inicial.

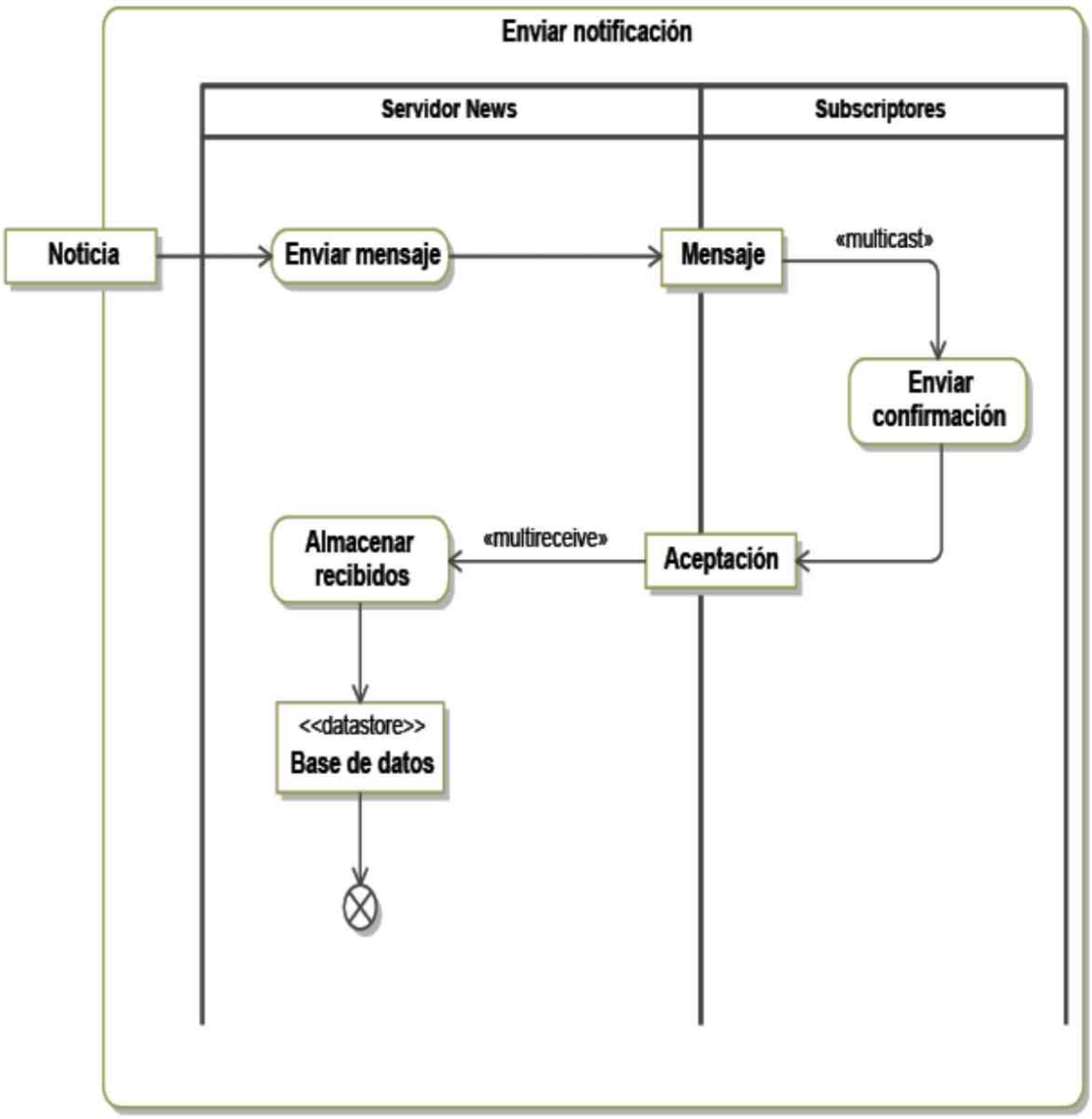


Figura 12.22. Ejemplo de «multicast» y «multireceive»

caso de estudio: ajedrez

En el diagrama de la figura 12.23 se representa el diagrama de actividades para el caso de uso "*Hacer jugada*" visto en el capítulo dos.

La actividad comienza con la selección de pieza sobre el tablero y el movimiento a la posición de destino. Después de dibujar la pieza en su correspondiente escaque se procede a detectar si está realizando algún mate y si está siendo amenazada por alguna pieza rival. Finalmente, en caso de estar amenazada por otra pieza se ofrece consejo al usuario. Si el jugador prefiere pedir consejo a la IA, el sistema le proporcionará una jugada estratégica de ayuda para realizar, en caso contrario se finaliza la actividad.

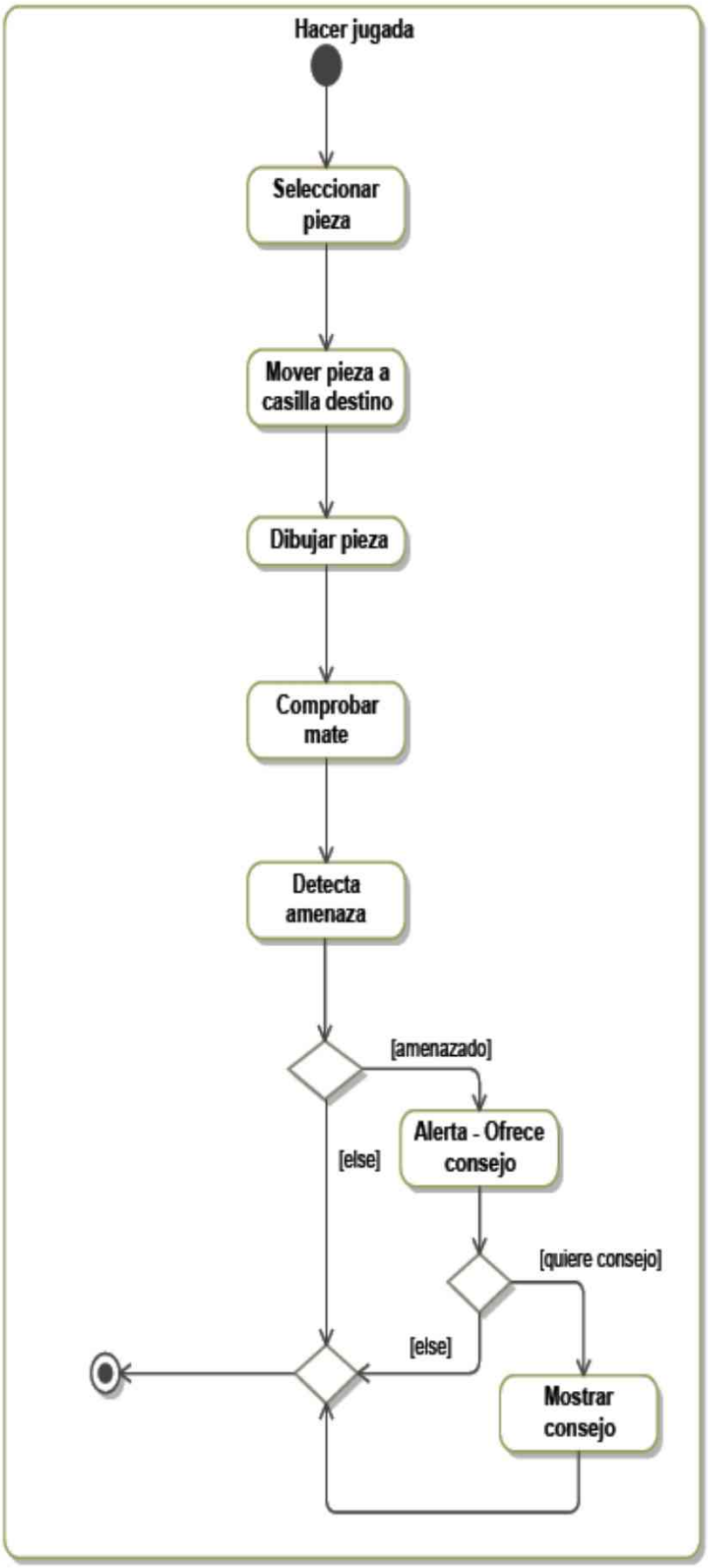

Hacer jugada
Seleccionar pieza
Mover pieza a casilla destino
Dibujar pieza
Comprobar mate
Detecta amenaza
[amenazado]
[else]
Alerta - Ofrece consejo
[quiere consejo]
[else]
Mostrar consejo

Figura 12.23. Diagrama de actividades para el caso de uso: “Hacer jugada”

caso de estudio: mercurial

Nos encontramos ahora con una situación algo diferente con respecto al ejemplo anterior. En el caso de la figura 12.24 se modela la especificación del caso de uso de *Mercurial "Listar ficheros"*. Sin embargo, en este ejemplo hemos optado por proponer una situación en la que concurren las peticiones de dos usuarios simultáneamente.

El escenario comienza modelando una partición con dos carriles: *Programador A* y *Programador B*. Ambos arrancan sus actividades en un nodo de inicio diferente para desembocar en un nodo de concurrencia. En este momento las acciones de procesar comandos generan en paralelo sus respectivos objetos. Dichos objetos *Comando* serán pasados al proceso de interpretación de los *ClientesProgramadores* que serán los encargados de enviar la petición a la fachada de comunicaciones.

En otro instante de tiempo se inician, en el carril del servidor, las actividades paralelas de recepción del mensaje de petición y de generación de los respectivos listados. Una vez generados los listados son devueltos a sus clientes en forma de objeto serializado por el *socket*.

Cuando los programadores A y B reciben sendos objetos comienzan a imprimir en sus pantallas los listados. Finalmente cada cauce de ejecución termina de forma independiente con respecto a sus otros flujos. Para indicar esto utilizaremos el nodo de final de flujo (nodo con un aspa).

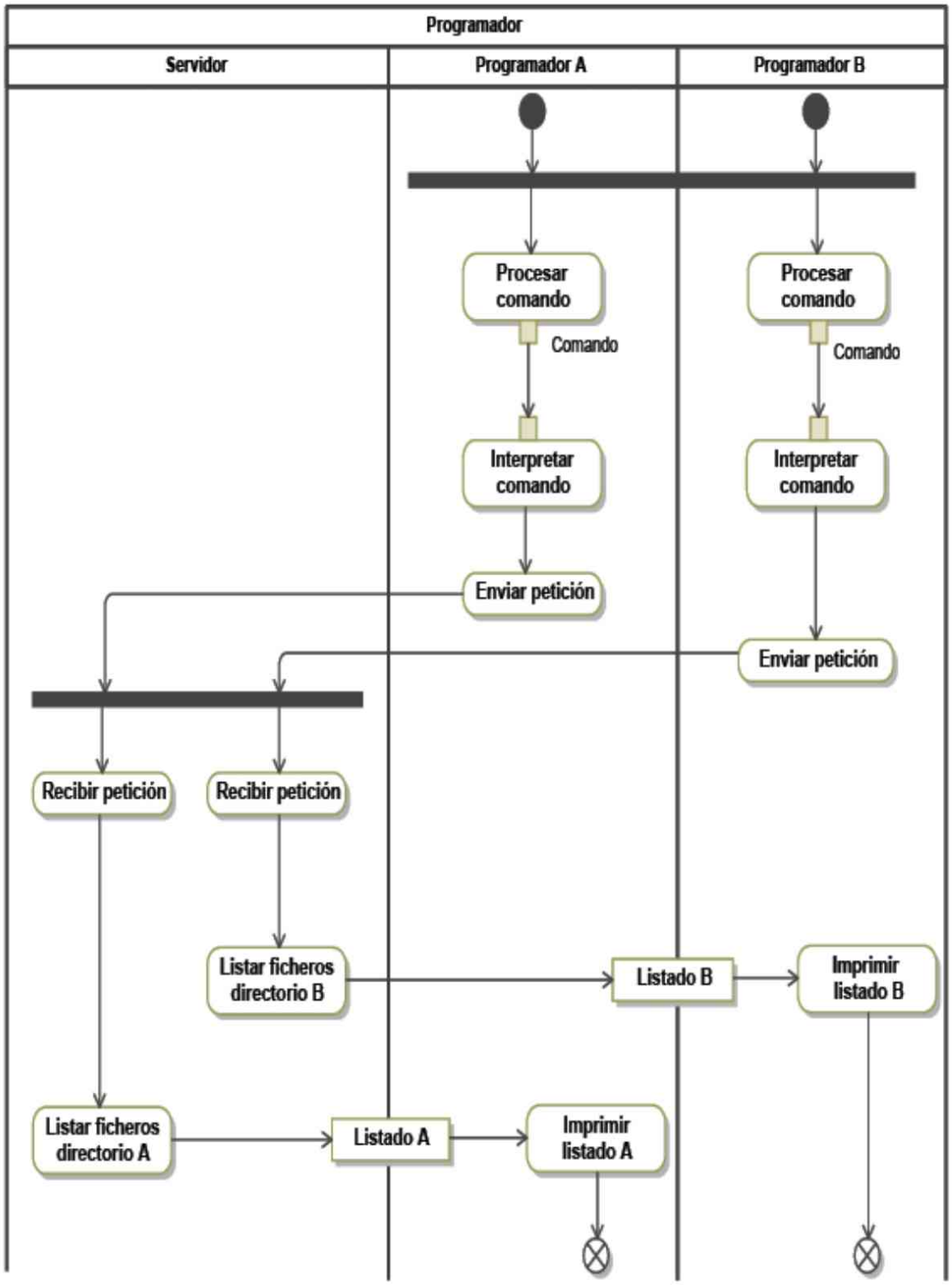


Figura 12.24. Diagrama de actividades para el caso de uso: "Listar ficheros"

caso de estudio: servicio de cifrado remoto

Como en el ejemplo anterior, en el diagrama de la figura 12.26 recurriremos igualmente a la utilización de carriles (*swimlanes*) con el fin de modelar un escenario entre dos clientes y un servidor que envían y recibe dos mensajes cifrados respectivamente. Tanto el *Cliente1* como el *Cliente2* comienzan con un nodo *fork* de paralelismo en el que se simula la selección de algoritmo y escritura del mensaje para cifrar desde la interfaz de usuario. Seguidamente, el mensaje se transmite en forma de objeto pin a sendos cifradores, para posteriormente transitar a la actividad de envío de los mismos por la red. Es importante destacar en este punto la utilización de excepciones de envío por *sockets* y las actividades que procesan las situaciones anómalas mediante la captura del objeto *OSError*. Los objetos con el contenido cifrado se sitúan en el borde del carril en forma de parámetro de salida entre los clientes y el servidor.



Figura 12.25. Notación UML de excepción en diagrama de actividades

En el lado del servidor, que se mantiene a la escucha, se recurre de nuevo a un nodo de paralelización una vez son recibidas las conexiones en el mismo. Cuando los datos son recibidos por el *socket* del servidor se procede a determinar el algoritmo de cifrado concurrentemente, mediante el mismo procedimiento visto en el diagrama de estados del capítulo anterior. En el caso del *Cliente1* se produce una excepción al intentar descifrar un mensaje cifrado de procedencia desconocida con un algoritmo simétrico. La situación se resuelve

capturando la excepción y sometiendo al mensaje a un algoritmo de descifrado híbrido como última opción antes de mostrar un mensaje de error que no llega a producirse en este diagrama.

Por último, ambos flujos de actividad se reagrupan en un nodo de sincronización antes de la finalización total.

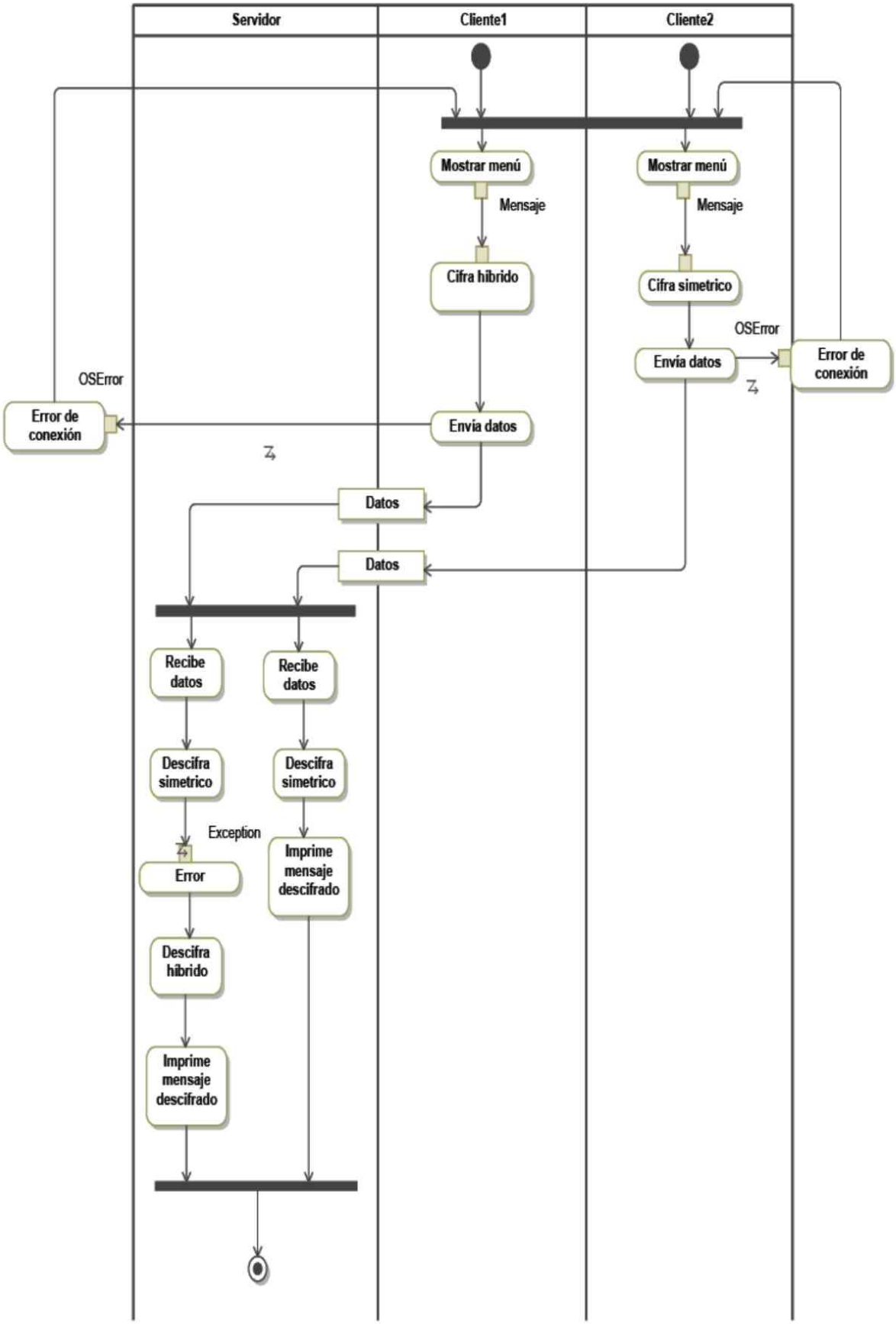


Figura 12.26. Envío de mensajes cifrados por dos clientes al servidor

Diagramas de estructura compuesta

«El uso de unas estructuras de datos bien escogidas suele ser un factor crucial en el diseño de algoritmos eficientes [...]».

(G. Brassard - P.Bratley: Fundamentos de Algoritmia, Capítulo 5).

Las clases que se representan en los diagramas de clases suelen contener otras clases mediante el símbolo de composición o agregación para conformar un concepto o entidad compuesta. Con frecuencia, dichas clases contienen otras clases de forma recursiva. Una desventaja de los diagramas de clases es que la capacidad sintáctica para definir composiciones es limitada y suele ocurrir que una clase que comparta entidades con otras clases posea cierta ambigüedad en sus asociaciones. Por este motivo, la especificación de UML nos facilita una alternativa eficaz para representar estos casos de composición y superar las limitaciones anteriormente descritas.

La principal ventaja de los *diagramas de estructura compuesta* es la capacidad para mostrar la estructura interna de un clasificador, como puede ser una clase. La potencia expresiva de los diagramas de estructura compuesta para definir una recursión de clases deja claramente expuesta la relación de composición en una situación real. Otra gran ventaja de estos diagramas es la posibilidad de indicar mediante conexiones la semántica de las relaciones existentes entre los clasificadores cuando se usan en un determinado contexto o situación.

estructura básica

Básicamente el *diagrama de estructura compuesta* está formado por los siguientes elementos:

- **Clase estructurada:** Es el contenedor principal del resto de los objetos, el clasificador en cuestión que se intenta modelar para representar su contenido interno.
- **Parte:** Representa el clasificador contenido por la clase mediante el uso de la *composición* o *agregación*. Se simboliza con un rectángulo de línea continua.
- **Propiedad:** Se denomina *propiedad* a aquellas entidades internas de la clase que no están contenidas mediante composición o agregación. Indican generalmente referencias externas definidas con asociaciones y se representan mediante un rectángulo de línea discontinua.
- **Conector:** Permite enlazar varias *partes* en el interior de una clase estructurada. Los *conectores* representan aquí las asociaciones del diagrama de clases.
- **Puertos:** Como veremos más adelante existen dos tipos de puertos: *de comportamiento* y *de servicios*. Esta notación permite representar la relación de una clase estructurada con su entorno y sus partes internas. Los puertos se simbolizan mediante un cuadrado en el borde de la parte contenedora.

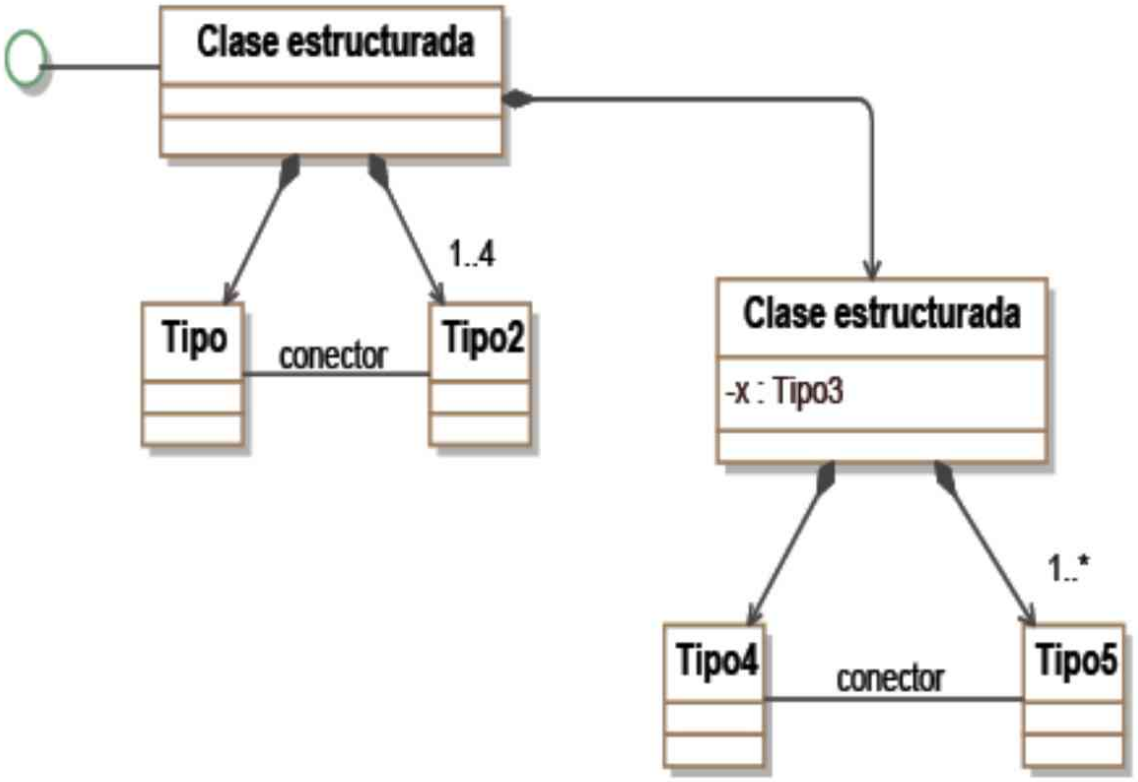


Figura 13.1. Diagrama de clases de ejemplo

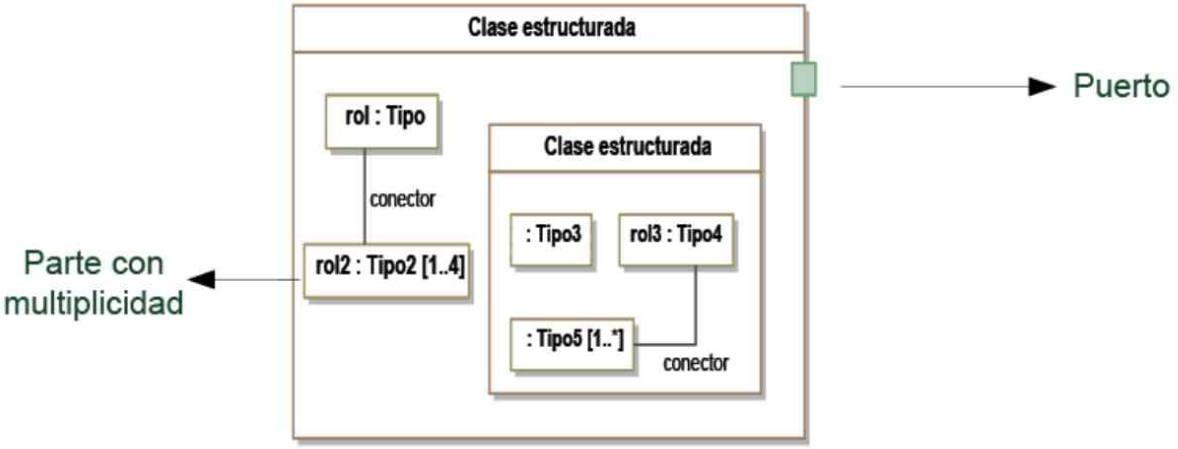


Figura 13.2. Diagrama de estructura compuesta correspondiente a la figura 13.1

En el diagrama de la figura 13.2 se muestra un ejemplo de una clase estructurada. El interior consta de dos partes con una clase estructurada que a su vez contiene otras dos partes. La notación en expresión regular para la parte

es la siguiente:

[nombreDeParte] : Tipo [`[´a..b`]´ | `[´b`]´]

donde el rol o el nombre de la parte es opcional y el tipo obligatorio. Adicionalmente pueden añadirse los límites inferior (*a*) y superior (*b*) de la multiplicidad.

Por último, el ejemplo también muestra el puerto implícito por el cual el clasificador se comunica con el exterior.

Para comprender un caso real de diagramas de estructura compuesta vamos a suponer cómo se representaría internamente la clase *Display* explicada en la figura 6.12 del capítulo dedicado al diagrama de clases.

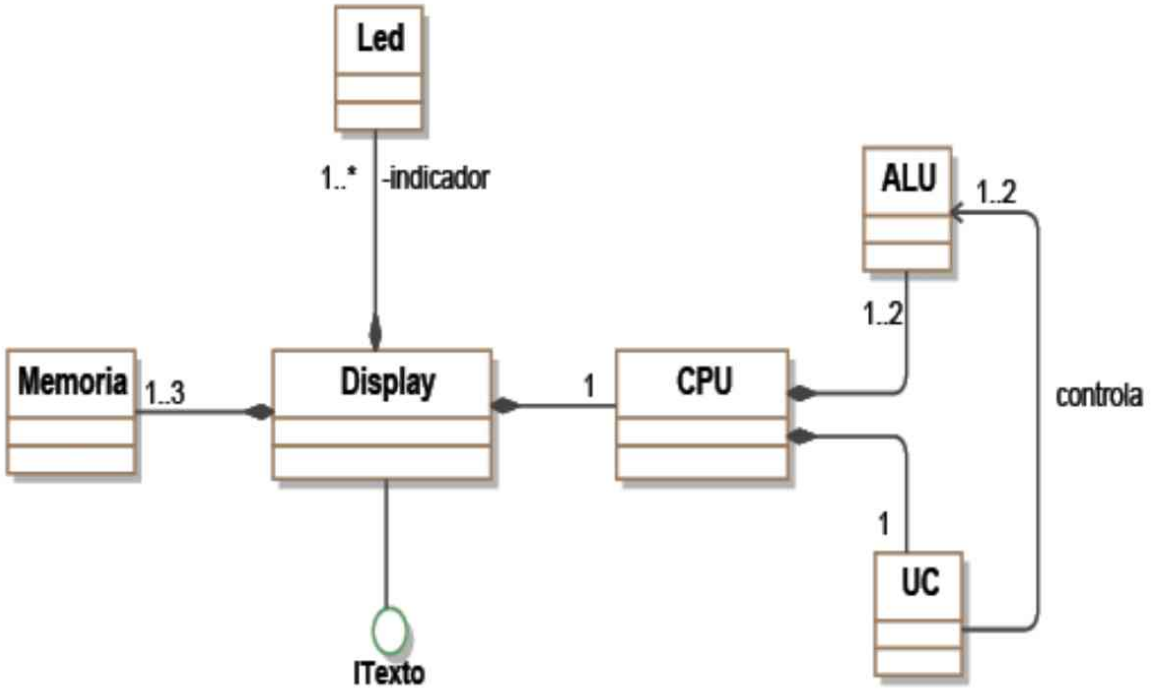


Figura 13.3. Diagrama de clases ampliado de "Display"

En la figura 13.3 se representa el diagrama de clases ampliado de un *display* compuesto por un conjunto de varios *leds* para iluminar un texto, una CPU para procesar el movimiento de la cadena y tres memorias de pocos kB para almacenar datos. Con estas premisas el diagrama correspondiente a la figura

13.3 es el mostrado a continuación:

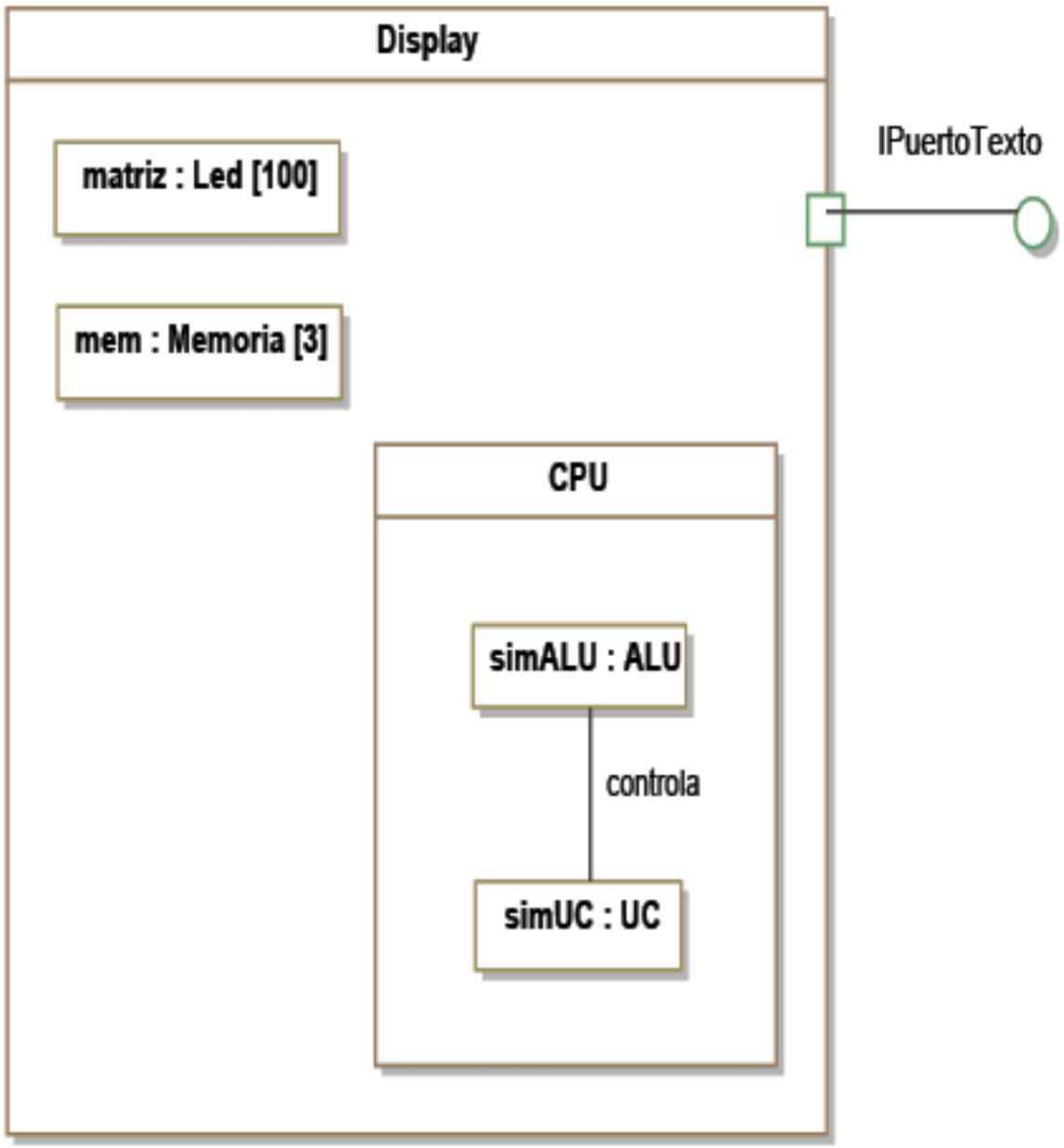


Figura 13.4. Diagrama de estructura compuesta para "Display"

La figura 13.4 es la estructura interna equivalente al diagrama mostrado en la figura 13.3, si bien ahora se visualizan instancias de las clases de la aplicación. Como se puede apreciar, la propiedad *matriz* de *Display* está compuesta por 100 *leds* de color que permiten la visualización del texto. Esta propiedad, que se encontraba indefinida en el diagrama de clases, pasa ahora a tomar un valor discreto. Lo mismo hay que decir para la memoria, que se ajusta a su

valor máximo, mientras que la CPU muestra sus dos componentes principales: la ALU (unidad aritmética-lógica) y la UC (unidad de control). La ALU y la UC están enlazadas mediante un conector equivalente a la asociación *controla*, puesto que es la unidad de control la que gobierna la ejecución de las instrucciones en la ALU.

Por último nos encontramos el puerto que presenta la interfaz *IPuertoTexto* por la cual otro objeto o componente puede enviar el *String* de la cadena de texto a visualizar.

Puertos

Como se ha comentado al comienzo del capítulo, los puertos permiten que una clase estructurada pueda comunicarse con su entorno y con otras partes internas. Generalmente los puertos se ubicarán en el borde externo de la clase estructurada; aunque también es posible ubicarlos en las partes internas para la comunicación entre ellas. Los dos tipos más usuales de puertos en UML 2.x son los siguientes:

- *De servicio*: Son los tipos de puerto por defecto. Permiten especificar los servicios que proporciona o requiere el clasificador.
- *De comportamiento*: El puerto se conecta mediante una línea a un estado de la clase estructurada que representa el comportamiento interno de la misma. Este tipo de puerto no se tratará en este libro.

En el ejemplo de la figura 13.4, el puerto *IPuertoTexto* es un tipo de puerto de servicio ya que ofrece una interfaz para pasar la cadena de texto a la matriz de *leds*.

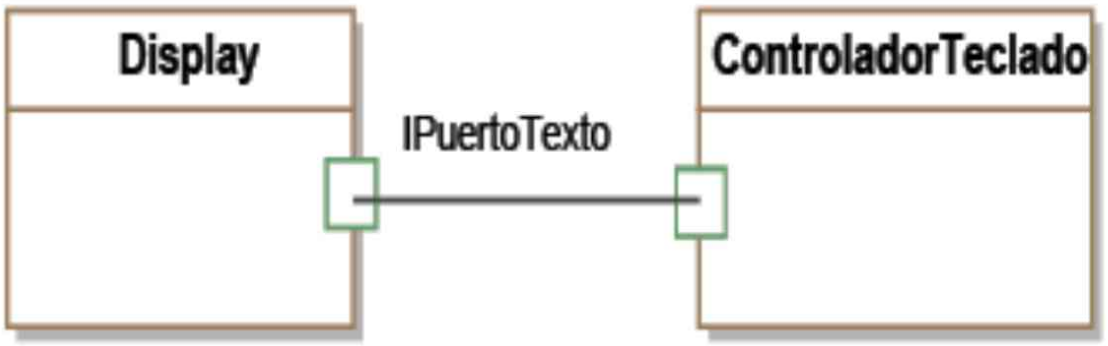


Figura 13.5. Figura de interconexión de dos clases estructuradas

En la figura anterior la clase *ControladorTeclado* requiere la interfaz proporcionada por el puerto del *Display* (*IPuertoTexto*).
En la figura 13.6 se muestra el mismo ejemplo anterior pero indicado con

otra notación más explícita:

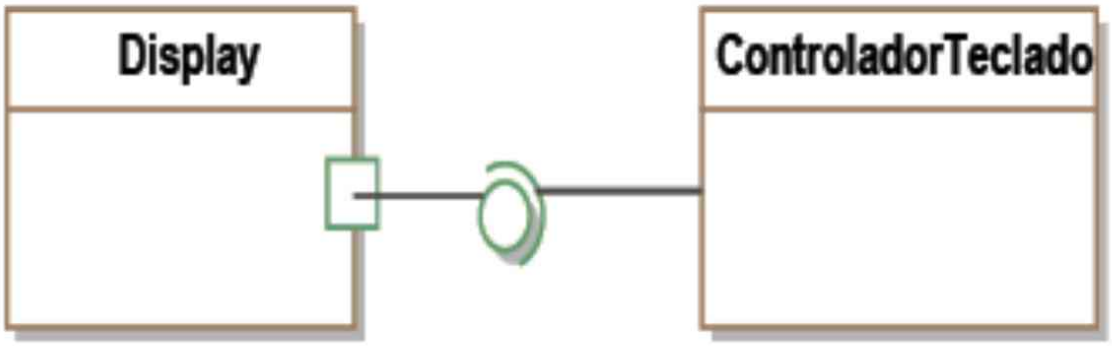


Figura 13.6. Notación explícita para el ejemplo de la figura 13.5

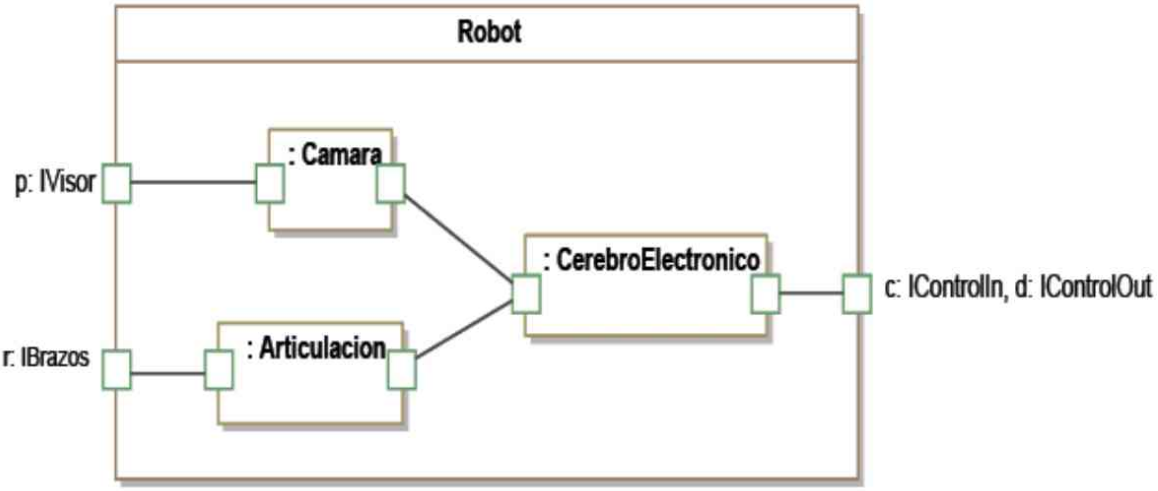


Figura 13.7. Diagrama de estructura compuesta con varios puertos

La conexión a través de puertos permite recrear estructuras compuestas más complejas, como este diagrama de un *Robot* con un cerebro electrónico que recibe órdenes del exterior, una cámara que analiza imágenes de un sensor y las articulaciones que permiten controlar objetos externos.

Generalmente una clase estructurada que presenta un puerto puede ser representada de forma implícita (figura 13.7) o de forma explícita con todas las interfaces que proporciona y requiere (figura 13.8):

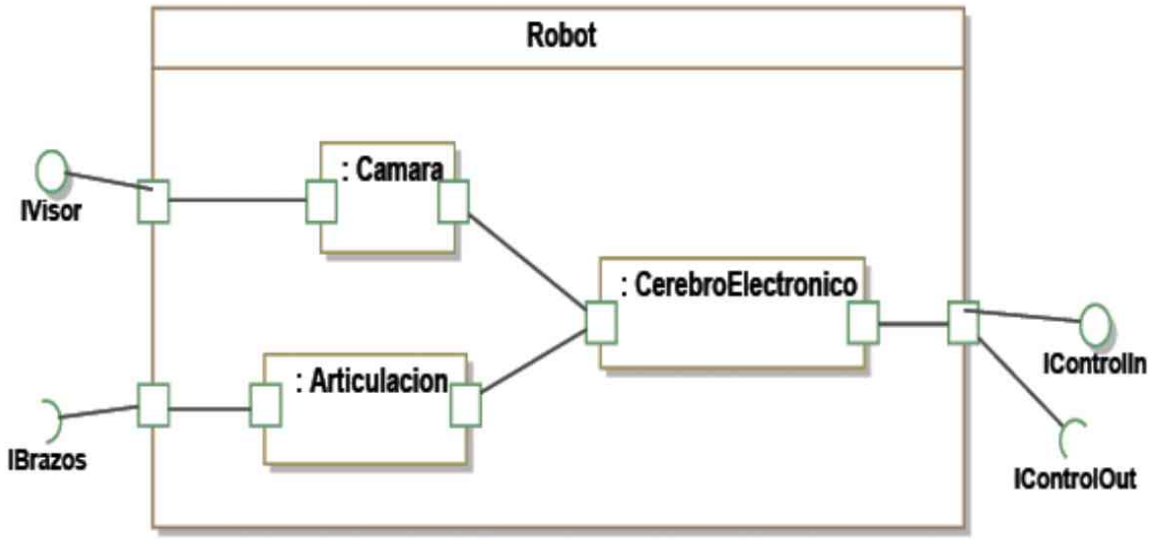


Figura 13.8. Clase estructurada con puerto explícito

colaboraciones

Otra característica destacada en los diagramas de estructura compuesta son las colaboraciones. Mediante las colaboraciones es posible describir los roles que protagonizan cada elemento para llevar a cabo una tarea o funcionalidad en el contexto del modelo conceptual. Dicha colaboración permite crear un *patrón* que será posteriormente aplicado en un contexto concreto mediante los d*iagramas de uso de colaboración*.

La notación de un diagrama de colaboración se representa mediante una elipse con un nombre que la identifica en la parte superior. En el interior de la elipse se representan los diferentes roles que participan en la colaboración unidos mediante conectores. Estos conectores modelan las interacciones que podrían suceder entre instancias.

A modo de ejemplo supongamos el diagrama de colaboración para la clase *Robot* vista en la figura 13.7:

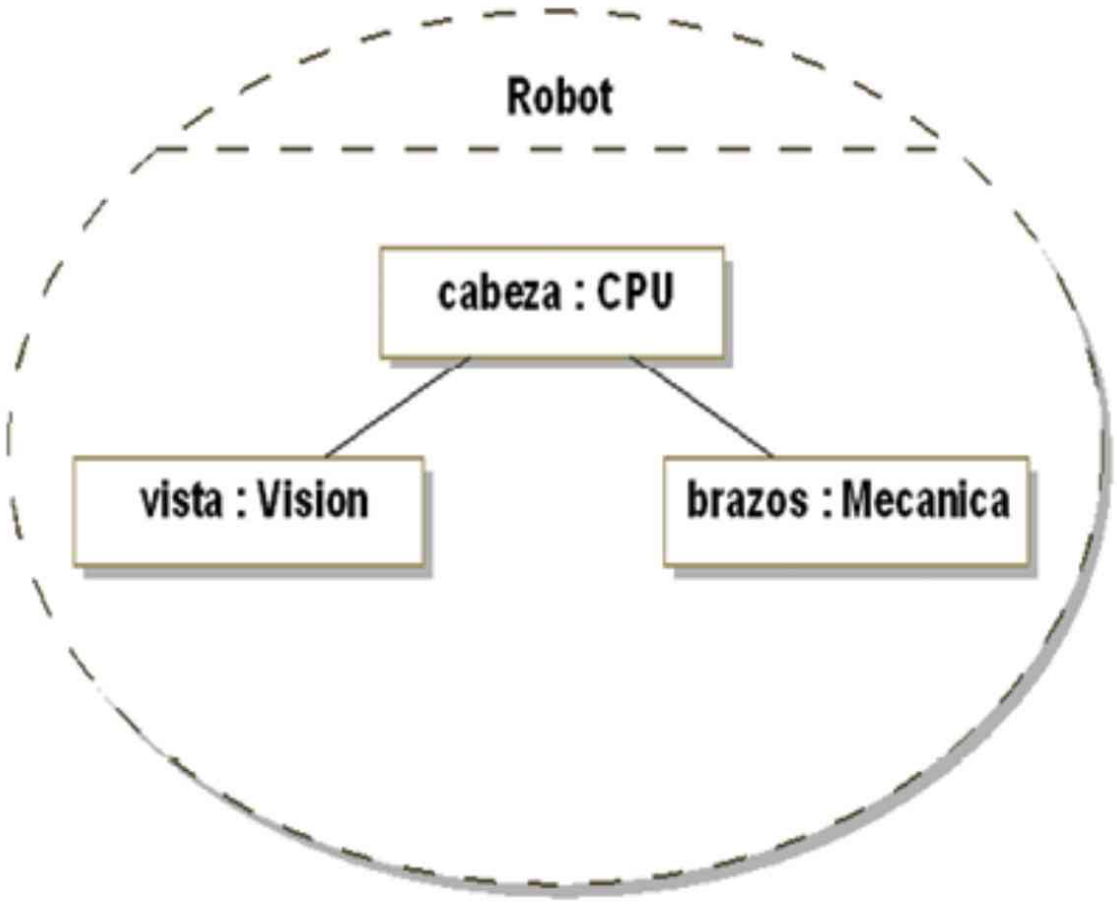


Figura 13.9. Colaboración para el ejemplo del Robot (figura 13.7)

La figura 13.9 muestra un caso de colaboración para la clase estructurada del *Robot* y su representación diagramática. En el interior se encuentran los tres roles que participan en la colaboración unidos mediante dos conectores. El siguiente paso es modelar un escenario donde estos roles sean aplicados a un caso concreto.

uso de la colaboración

El punto de llegada del diagrama de colaboración visto en el apartado anterior es asociar cada rol con instancias o partes representadas en un diagrama de estructura compuesta de forma que se pueda aplicar ese patrón a un determinado contexto. Así, para el ejemplo de la figura 13.9 podríamos considerar el siguiente uso:

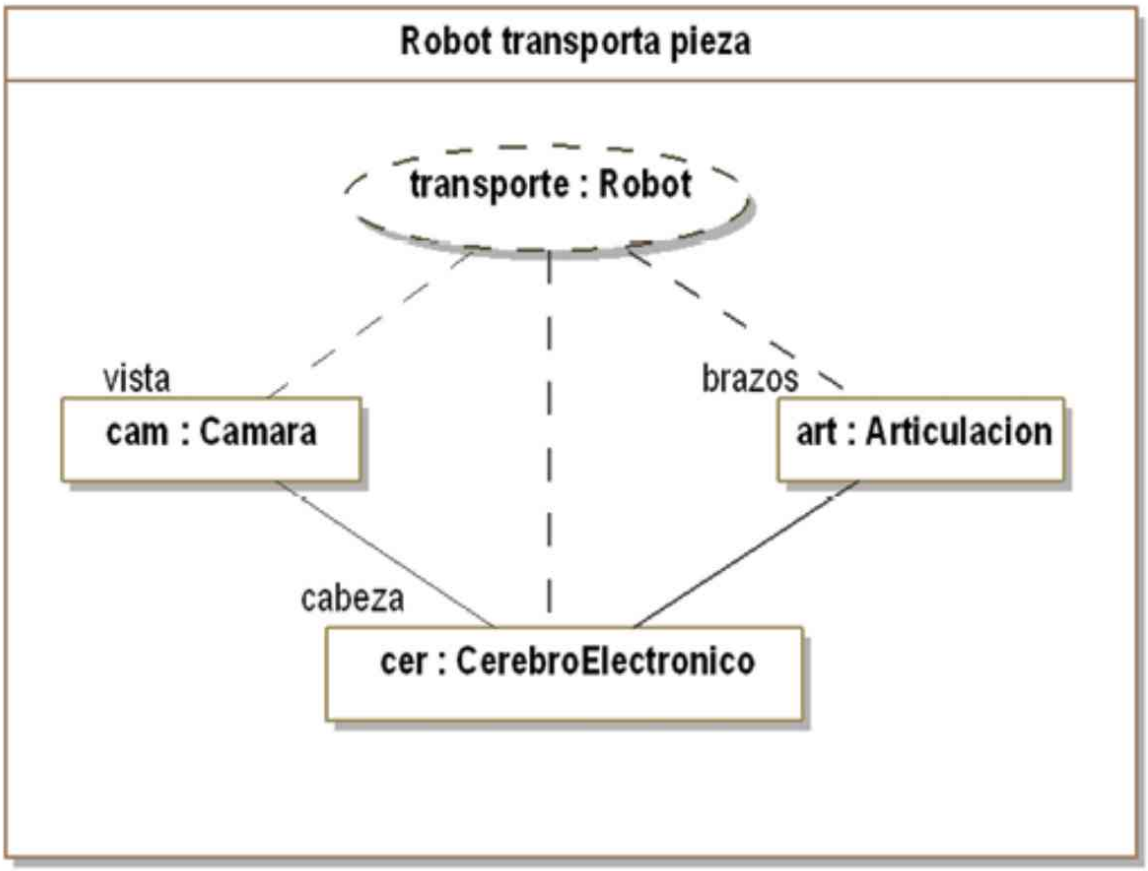


Figura 13.10. Diagrama de uso de colaboración de la figura 13.9

La figura 13.10 es la representación del *uso de la colaboración* para el caso particular de un robot transportando una pieza. En el diagrama del ejemplo se asocia cada rol del *uso de colaboración* (representado por una elipse en línea discontinua) con las partes concretas del diagrama de estructura compuesta. Este diagrama ilustra el comportamiento que tendrían las citadas partes en un determinado contexto de la clase.

caso de estudio: ajedrez

Diagrama de estructura compuesta

En la figura 13.12 podemos apreciar la descomposición en partes de la clase del modelo de ajedrez. En líneas discontinuas vemos las partes que no están relacionadas mediante una asociación de tipo composición. Las partes que están relacionadas con composición se representan con línea continua. Por ejemplo, en la clase *Jugador_humano* solo el *array* de Piezas[16] está asociado mediante composición:

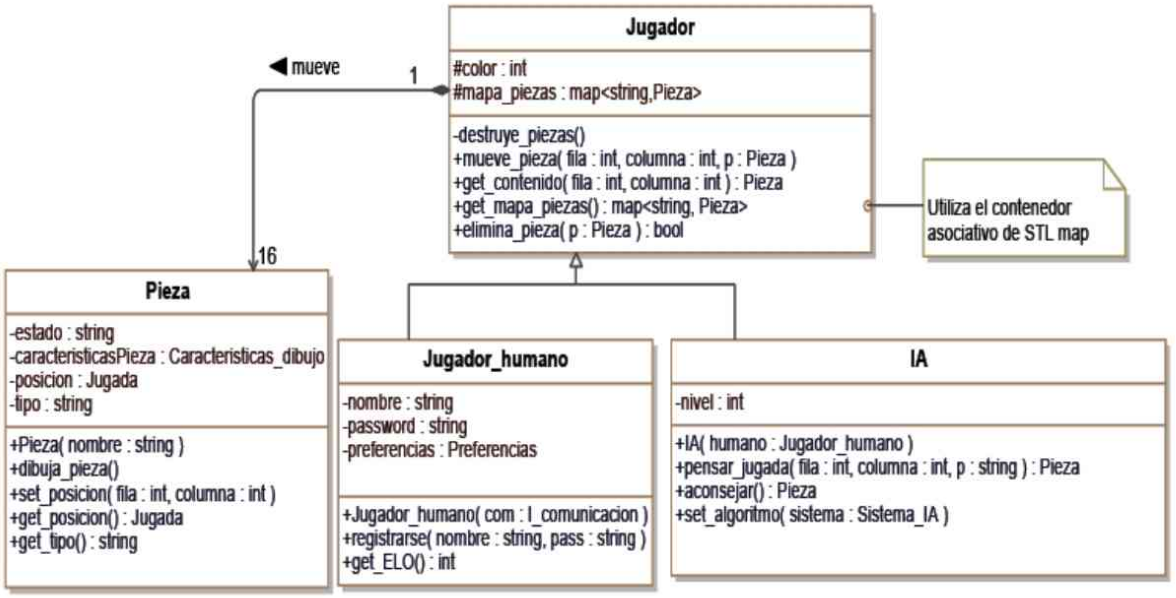


Figura 13.11. Diagrama de clases relacionado con la clase Jugador_humano

Mientras que en la figura 13.11 se representa el diagrama de clases para la clase *Jugador_humano*, en el diagrama 13.12 se puede apreciar el diagrama de estructura compuesta para la misma, lógicamente con mayor contenido semántico en su representación. Fíjese que todos sus elementos se han definido como partes al estar definidos implícitamente como atributos de composición en el diagrama de clases. En este ejemplo se hace uso por primera vez de la notación de clase estructurada con una cardinalidad de 16 elementos para el atributo *piezas* de la clase base *Jugador*.

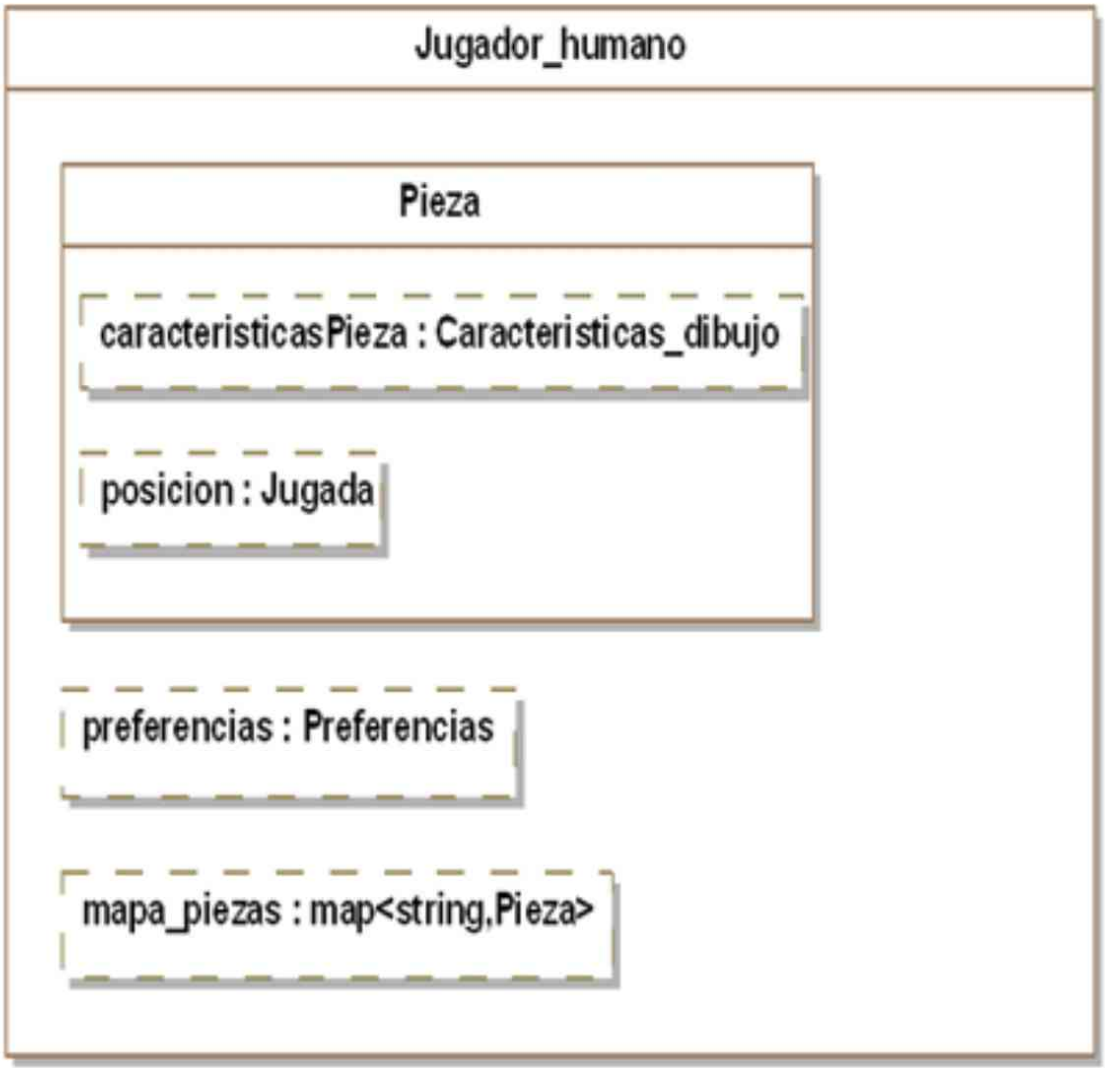


Figura 13.12. Diagrama de estructura compuesta para la clase Jugador_humano

Colaboración

La colaboración de la figura 13.13 muestra la interrelación de los roles que protagonizan cada una de las instancias del modelo. Como ya sabemos, la relación del sistema de IA requiere la intervención de un algoritmo y la obtención de un mapa de información del rival por parte de éste.

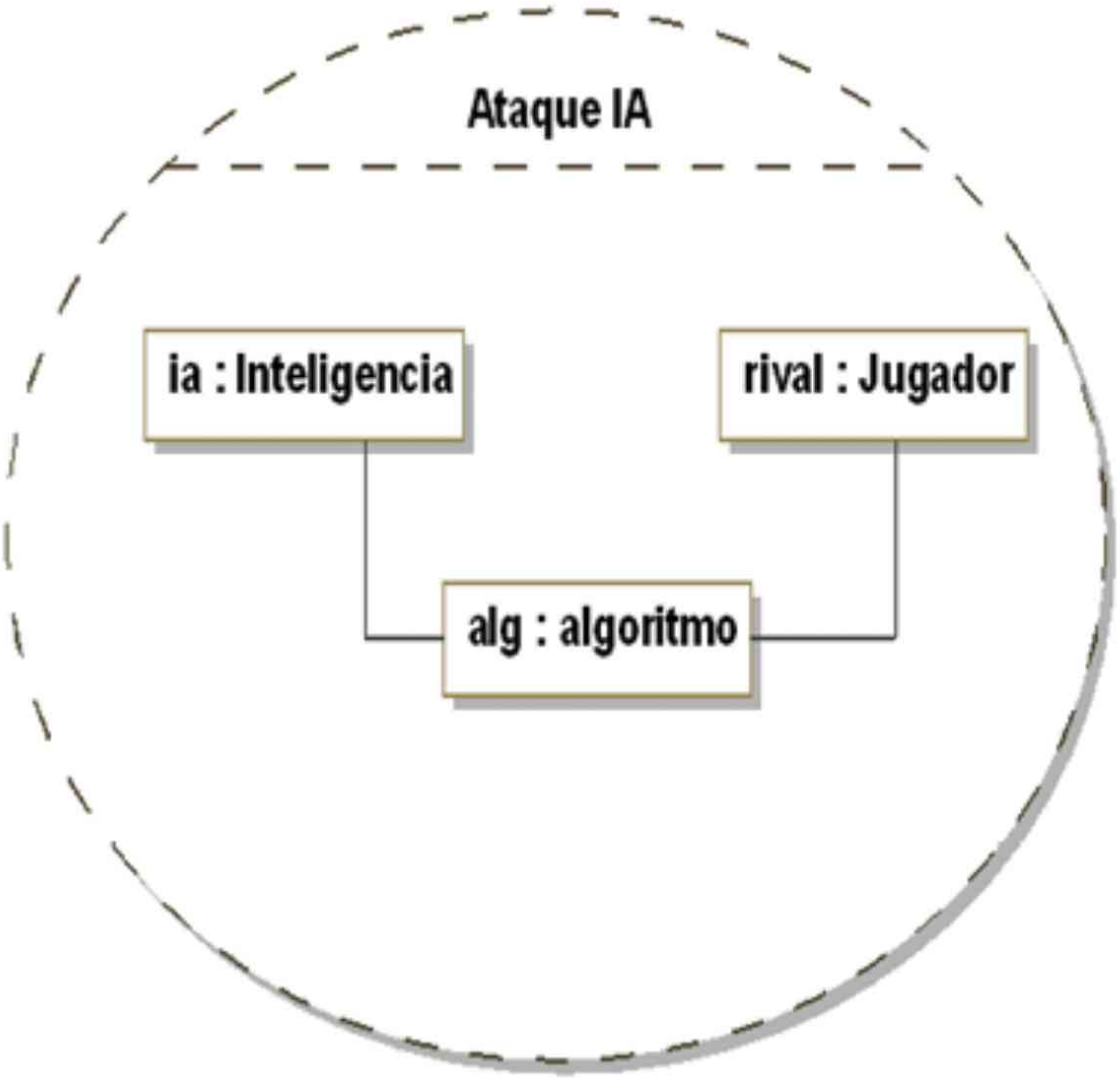


Figura 13.13. Colaboración que modela el ataque de la Inteligencia Artificial

Uso de la colaboración

El paso final es la aplicación de la colaboración vista en el esquema anterior a un caso concreto. En el diagrama 13.14 se muestra el contexto de aplicación. En el uso *turnoIA* que se muestra arriba en una elipse relaciona cada uno de los roles de instancias del modelo de clases con los roles de la colaboración vista en el apartado anterior.

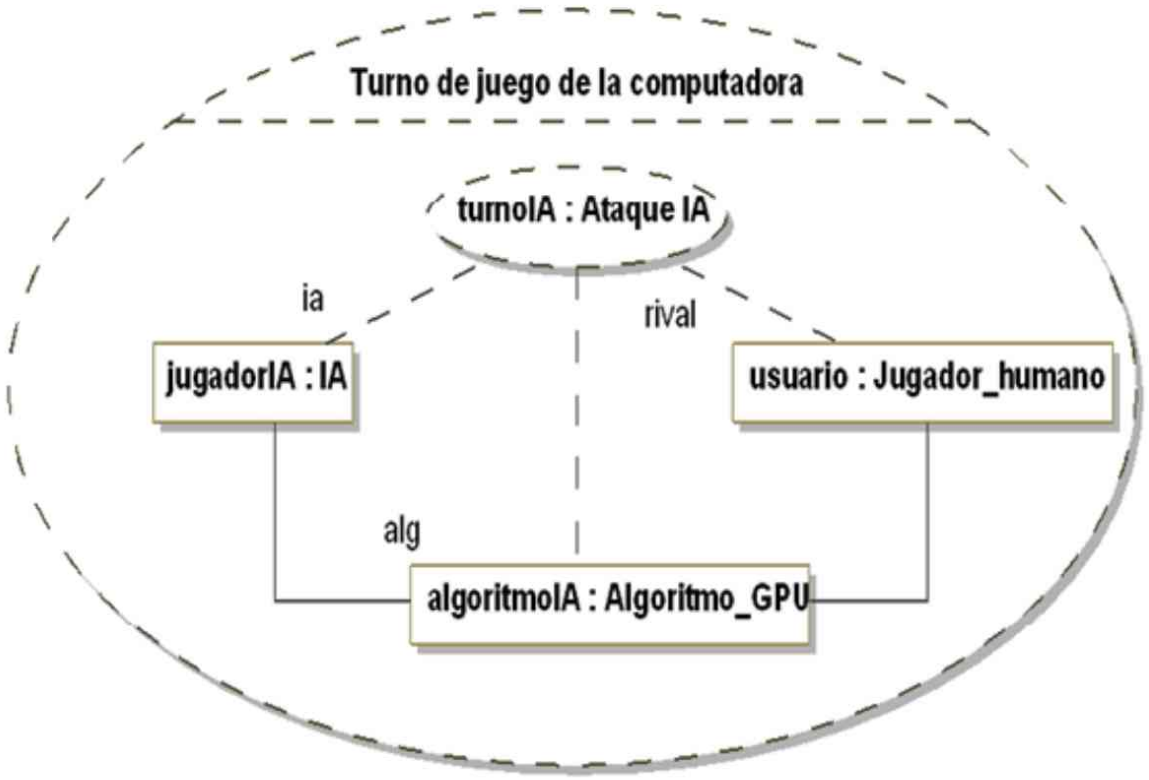


Figura 13.14. Uso de colaboración para el ataque de la computadora

caso de estudio: mercurial

Diagrama de estructura compuesta

En la figura 13.15 se representa la estructura compuesta para la clase *Usuario*. Esta clase más elemental muestra un puerto explícito que implementa dos interfaces:

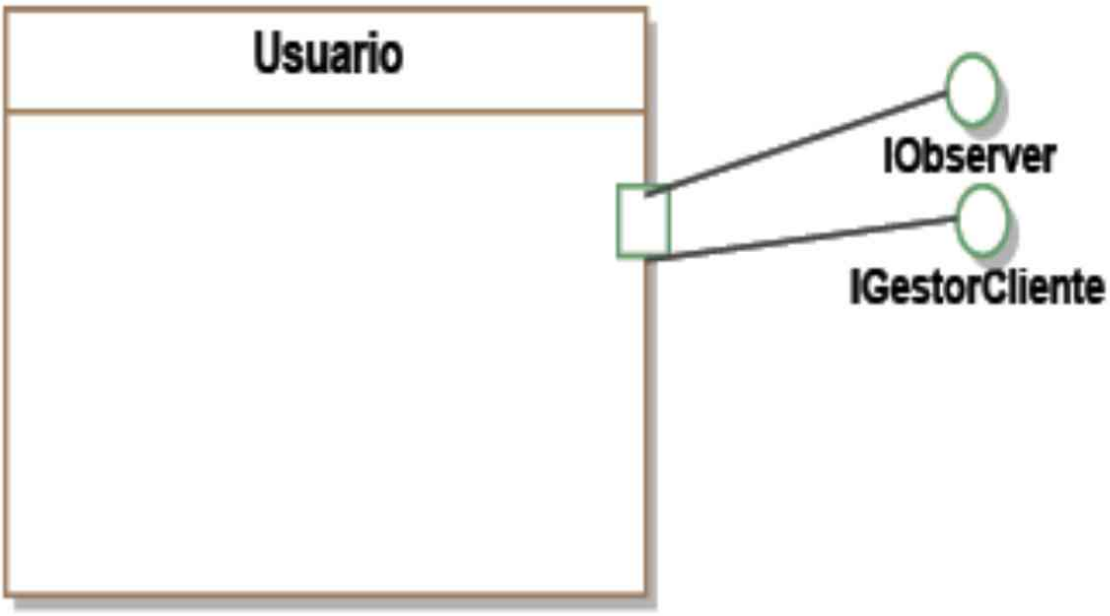


Figura 13.15. Diagrama de estructura compuesta para la clase Usuario

Colaboración

La figura 13.16 muestra la colaboración de los roles entre un programador y el sistema de archivos remoto en el servidor a través de la red.

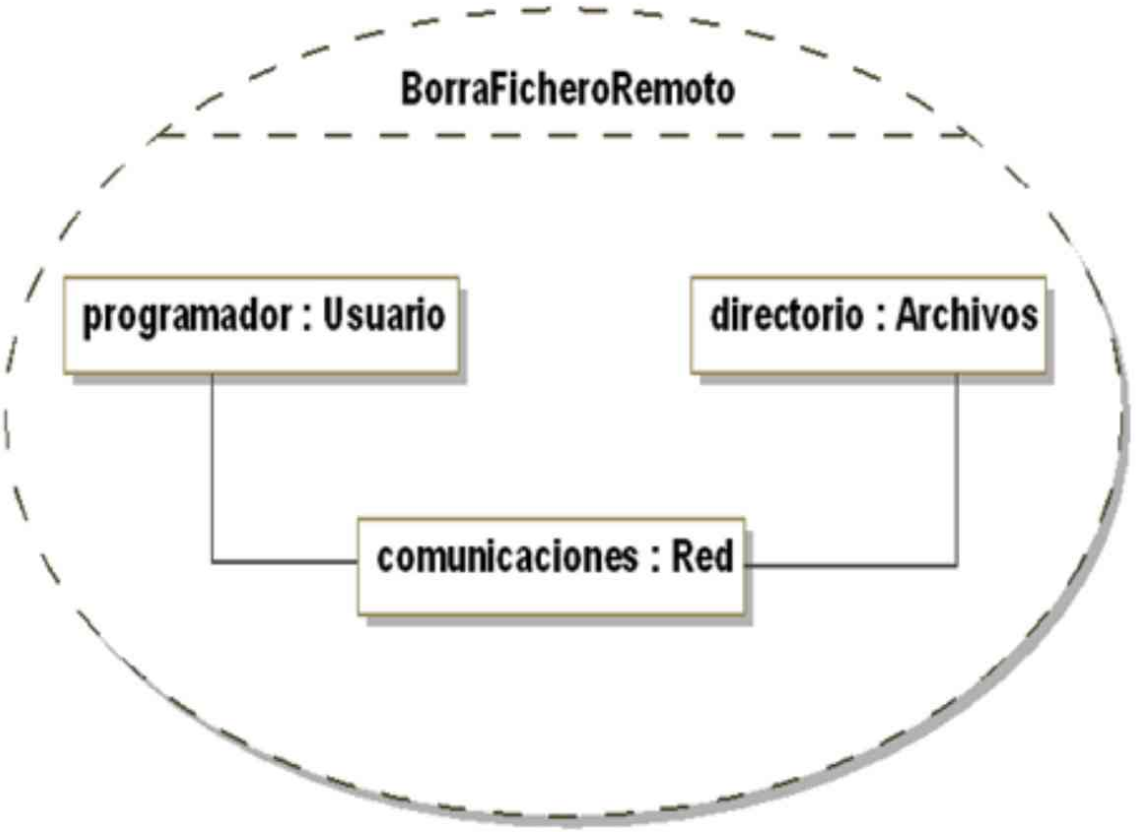


Figura 13.16. Diagrama de colaboración para la clase Usuario

Uso de la colaboración

Finalmente, se aplica la colaboración en el uso *"Programador borra fichero remoto"*, el cual relaciona las instancias del modelo de clases con los roles definidos en el diagrama de la figura 13.16.

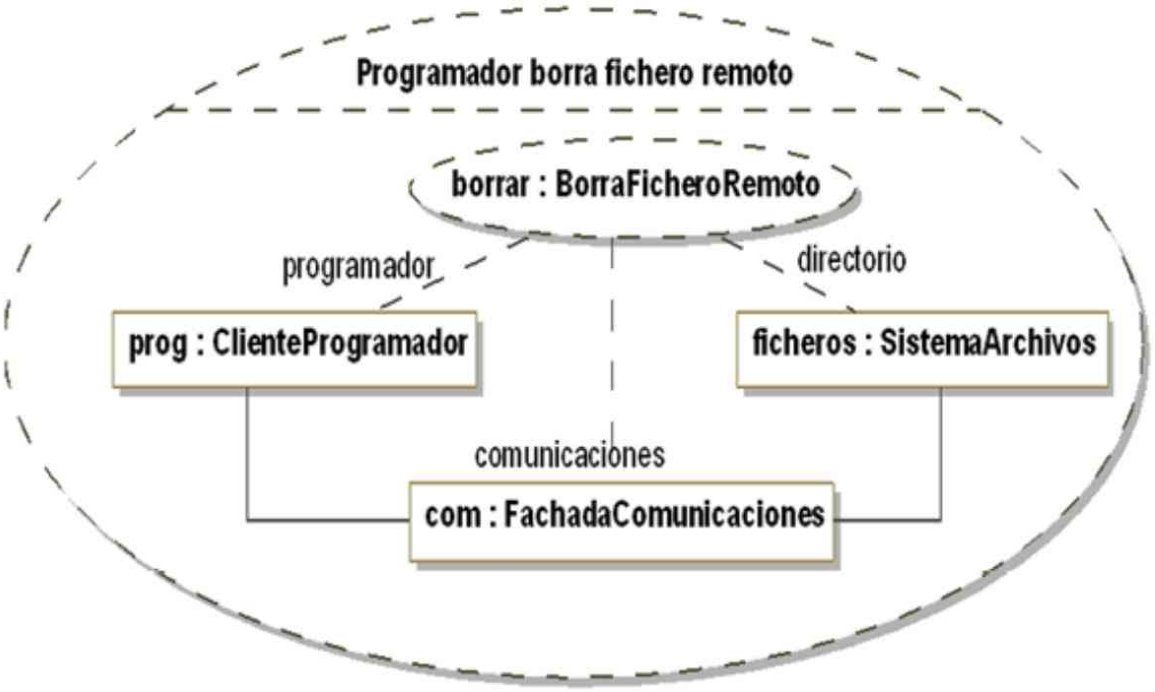


Figura 13.17. Uso de la colaboración para el borrado de un fichero en el servidor

caso de estudio: servicio de cifrado remoto (cliente)

Diagrama de estructura compuesta

En la figura 13.18 se observa claramente la composición 1..2 para las partes de la clase *Cifrador* y la propiedad para asociación con la fachada de comunicaciones. Se aprecia, así mismo, la implementación de la interfaz *IMenuCifrado* en la clase *Usuario*.

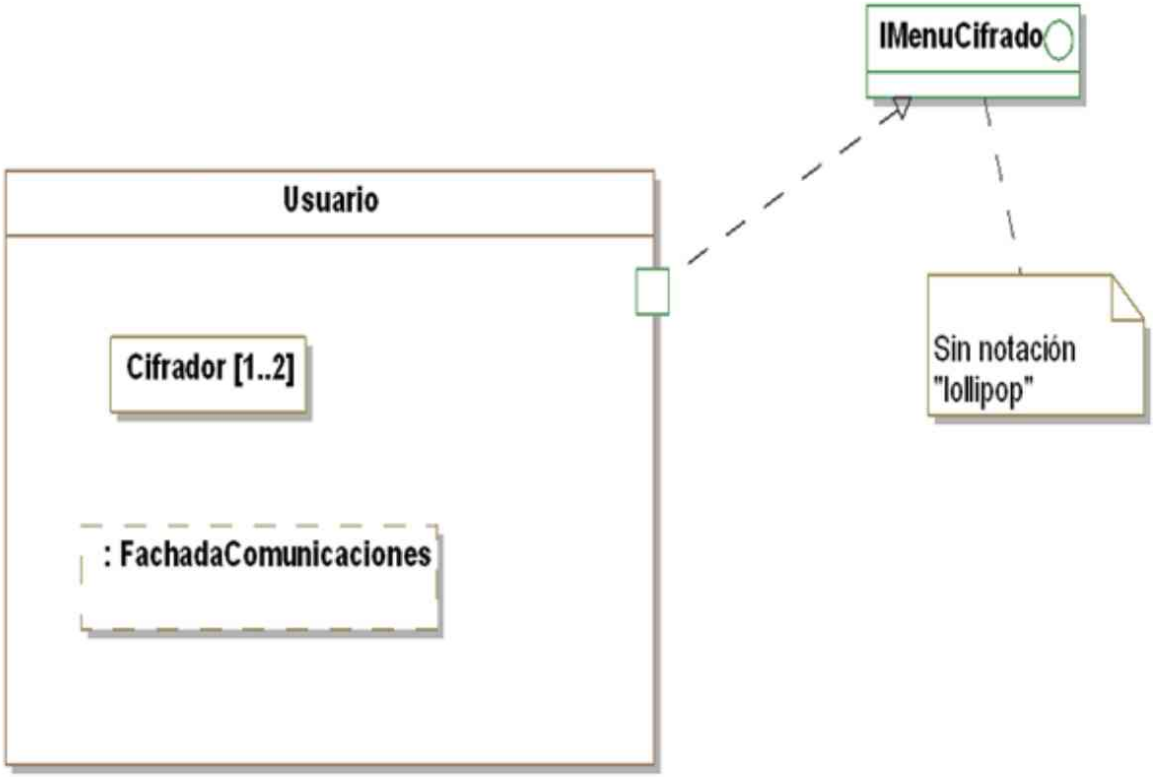


Figura 13.18. Diagrama de estructura compuesta para la clase Usuario novel

Colaboración

La figura 13.19 ilustra la colaboración de los roles que intervienen en el contexto del cifrado del mensaje, en el que intervienen el cliente, un determinado algoritmo y el sistema de comunicaciones.

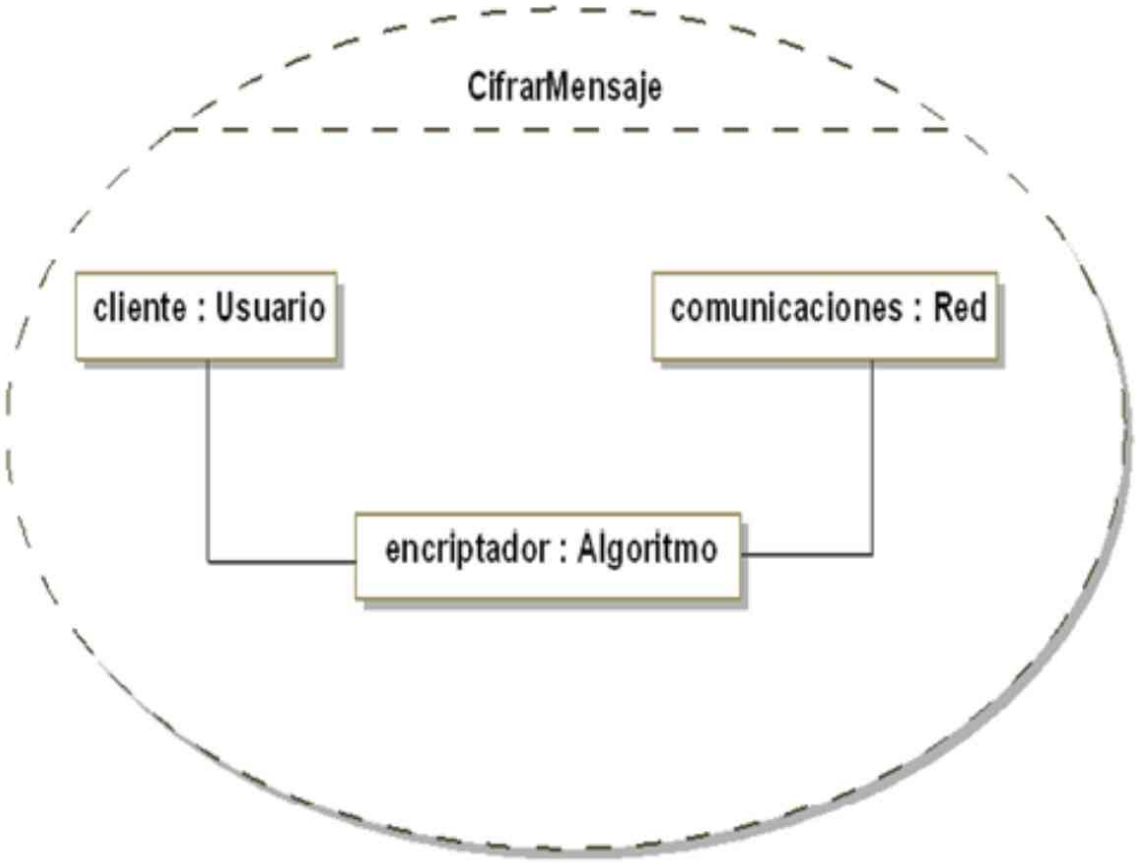


Figura 13.19. Diagrama de colaboración para cifrar mensaje

Uso de la colaboración

Finalmente, se aplica el patrón sobre el contexto determinado por el diagrama de estructura compuesta visto en la figura 13.18.

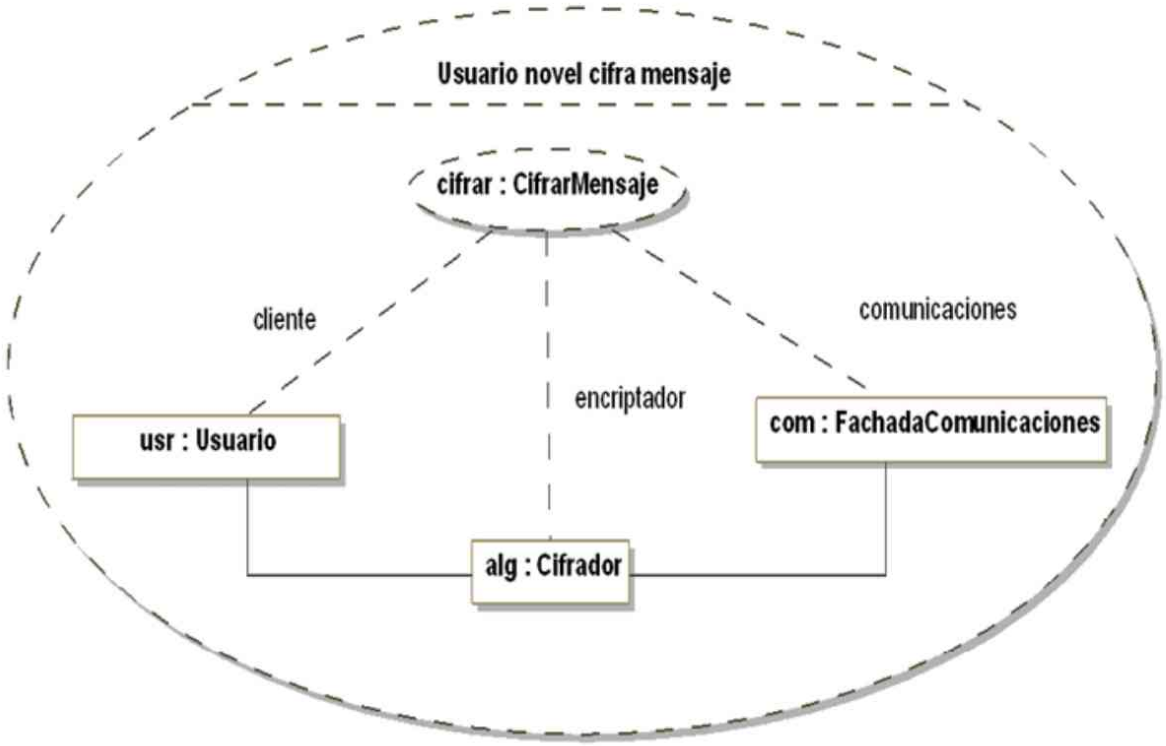


Figura 13.20. Uso del patrón de la colaboración

caso de estudio: servicio de cifrado remoto (servidor)

Diagrama de estructura compuesta

La necesidad de utilización de la herencia en el servidor para especializar a la clase *Usuario* hace que ésta también se refleje en el citado diagrama de estructura compuesta.

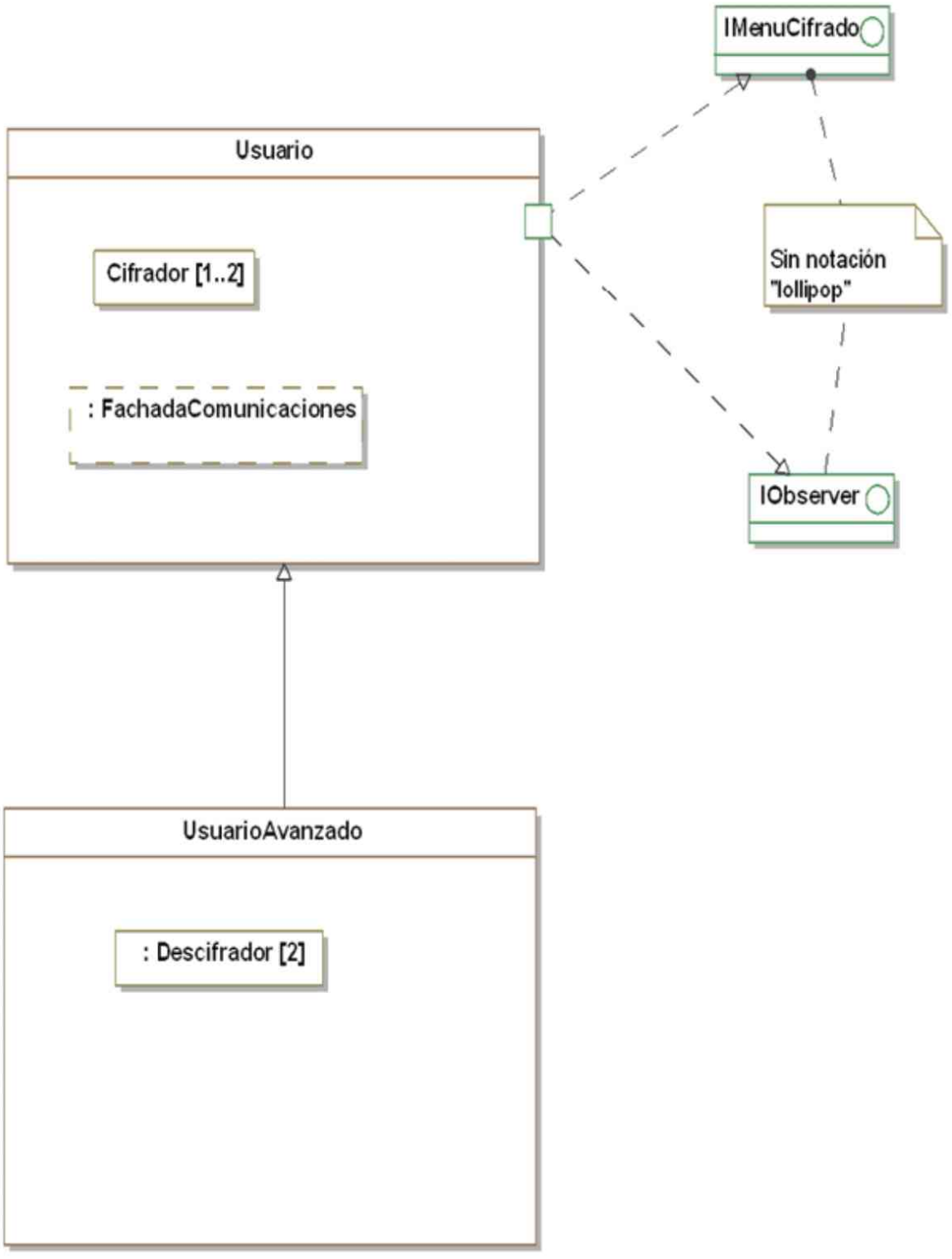


Figura 13.21. Diagrama de estructura compuesta para el servidor

Colaboración

El ejemplo de la figura 13.22 es el caso complementario al visto para el cifrado en el lado cliente.

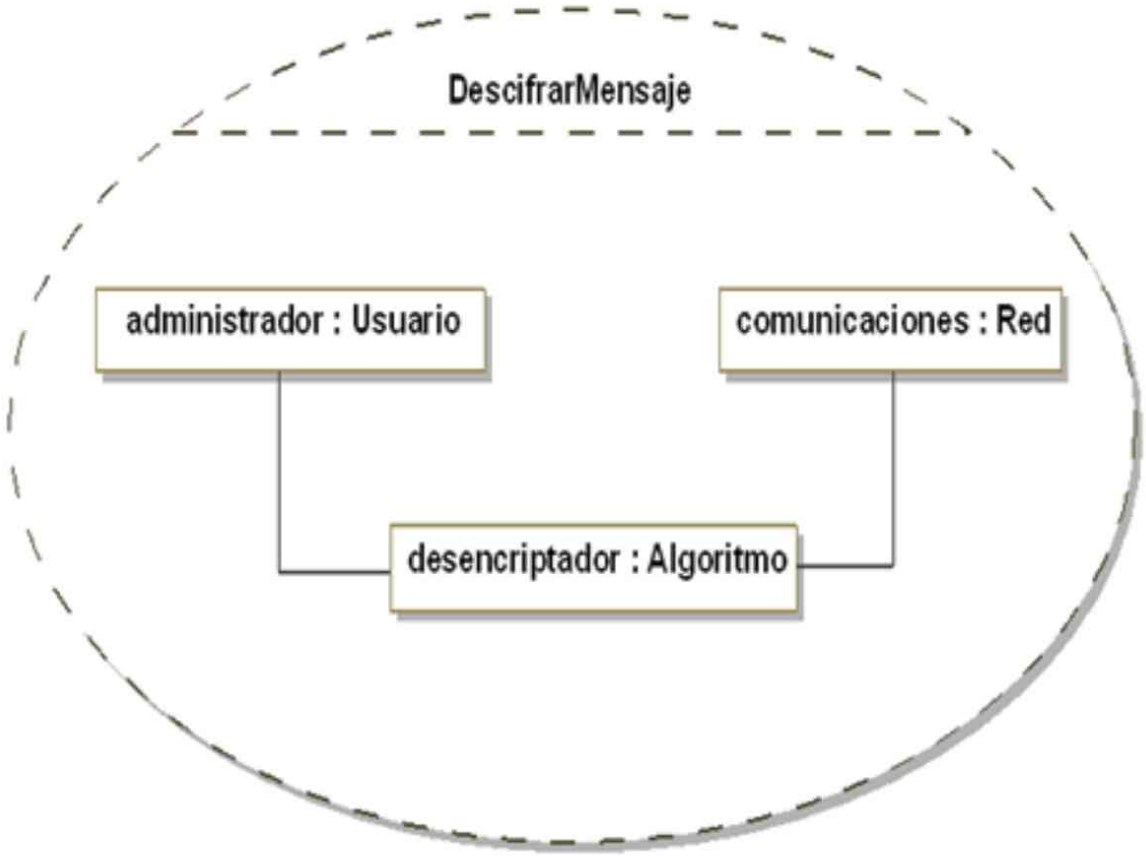


Figura 13.22. Diagrama de colaboración para descifrar mensaje

Uso de la colaboración

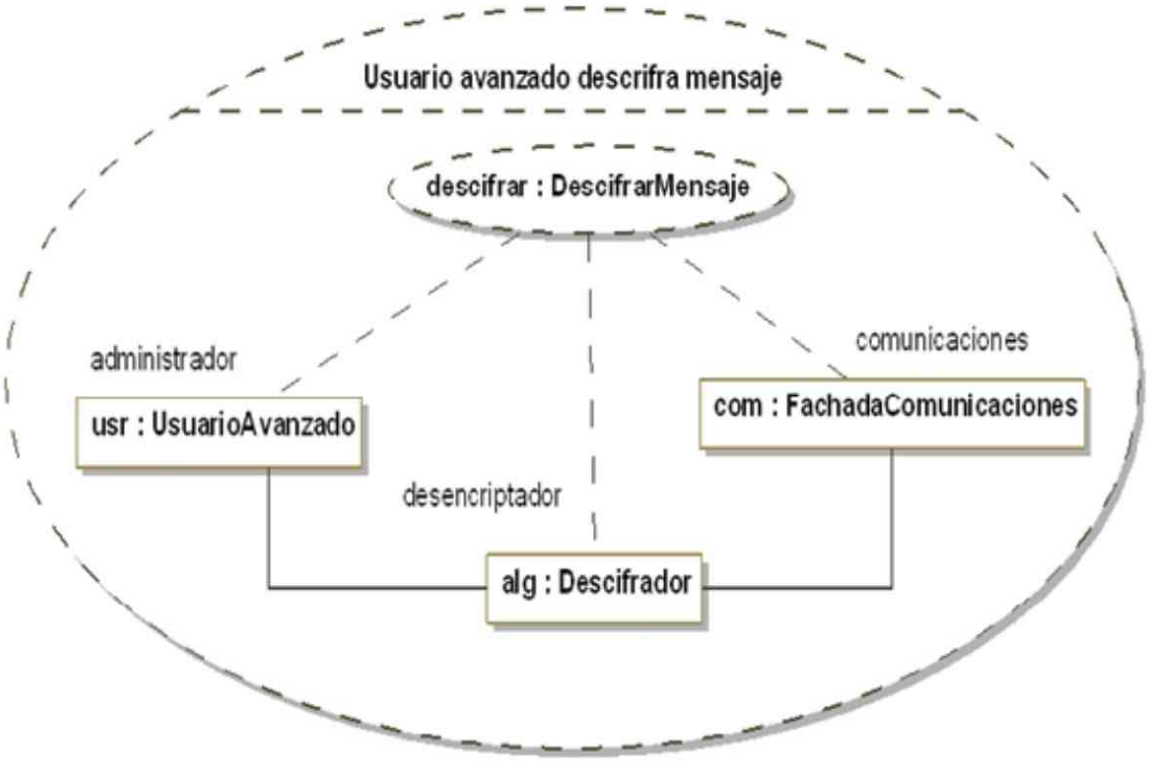


Figura 13.23. Diagrama de uso de la colaboración para descifrar mensaje

OCL (Object constraint language)

«Palabras nuevas y recién creadas tendrán aceptación, si van fluyendo desde la fuente griega y si se traen con precaución».

(Horacio: Arte poética, "Neologismos").

OCL (*Object Constraint Language* o lenguaje de restricción de objetos) es un lenguaje perteneciente a la especificación de UML que permite a los diseñadores de software aplicar restricciones y consultas a sus modelos de objetos. Estas restricciones y consultas, como veremos más adelante, se escriben en forma reglas con una sintaxis y semántica propias. Las limitaciones de los diagramas de clases para expresar algunas propiedades de un modelo es un factor crucial para el uso de este lenguaje.

OCL como tal es un lenguaje declarativo que va siempre unido a los diagramas UML para ampliar su semántica. Especifica mediante unas reglas sintácticas las consultas que requiere sobre un modelo y no la forma o el procedimiento explícito para llevarlas a cabo. Con OCL nunca se podrá cambiar el valor de un elemento del modelo de objetos, quedando intacto el sistema cuando se realiza una consulta.

OCL no es un lenguaje de programación imperativo como lo podría ser Java o C++. No es posible realizar instrucciones de control como *for*, *while*, etc. o llamadas a procedimientos en la forma en que se entiende para un lenguaje de programación convencional. En general, OCL se aplica en herramientas CASE para realizar comprobaciones de integridad y restricciones sobre los modelos UML.

estructura básica

Como todo lenguaje informático las expresiones OCL que se añaden al modelo tienen una determinada sintaxis formal. Una expresión OCL está compuesta de[31]:

> [package *nombrePaquete*] context [*nombreContexto*:] elementoUML (expresión [*nombreExpresión*]:cuerpo)+
> [endpackage]

En la anterior expresión se identifican las siguientes partes:

- **package nombrePaquete**: Define el paquete o el espacio de nombres al cual pertenece el clasificador. Si se utiliza debe finalizar con *endpackage*.
- **context nombreContexto:elementoUML**: Representa el elemento del modelo en estudio al cual se le aplicarán las expresiones, generalmente clasificadores u operaciones de clases.
- **expresión nombreExpresión:cuerpo**: Conjunto de cláusulas OCL donde se especifican las restricciones y las consultas sobre el modelo.

Las expresiones OCL deben indicarse en los diagramas entre signos de llaves:

{expresiónOCL}

En el lenguaje OCL se pueden añadir comentarios que facilitan la revisión y el mantenimiento por otros desarrolladores. Dichos comentarios no son procesados por el intérprete de OCL. Concretamente existen dos tipos de comentarios:

> -- Comentario de una sola línea de código OCL
> ó

```
/* Comentario de varias
líneas en un fragmento de código OCL */
```

tipos y operadores

OCL es un lenguaje fuertemente tipado y con una notación simple. Incluye cuatro tipos datos primitivos: **Boolean** (valores lógicos), **Integer** (números enteros), **String** (cadenas alfanuméricas) y **Real** (valores fraccionarios). Dichos tipos y operadores están destinados a utilizarse dentro del cuerpo de una expresión de consulta OCL.

Con la finalidad de simplificar la exposición del conjunto de operadores que se pueden aplicar a cada tipo primitivo proponemos a continuación la siguiente tabla:

Tipo	Operadores aplicables
Boolean	**=, <>, and, or, xor, implies, if-then-else**
Integer y Real	**=, <>, <, >, <=, >=, +, -, *, /,** **mod, div, abs, max, min, round, floor**
String	**=, <>, concat, size, toLower, toUpper,** **toInteger, toReal, subString**

Tabla 14.1. Operadores para cada tipo de datos primitivo

De igual forma la precedencia de dichos operadores es la siguiente:

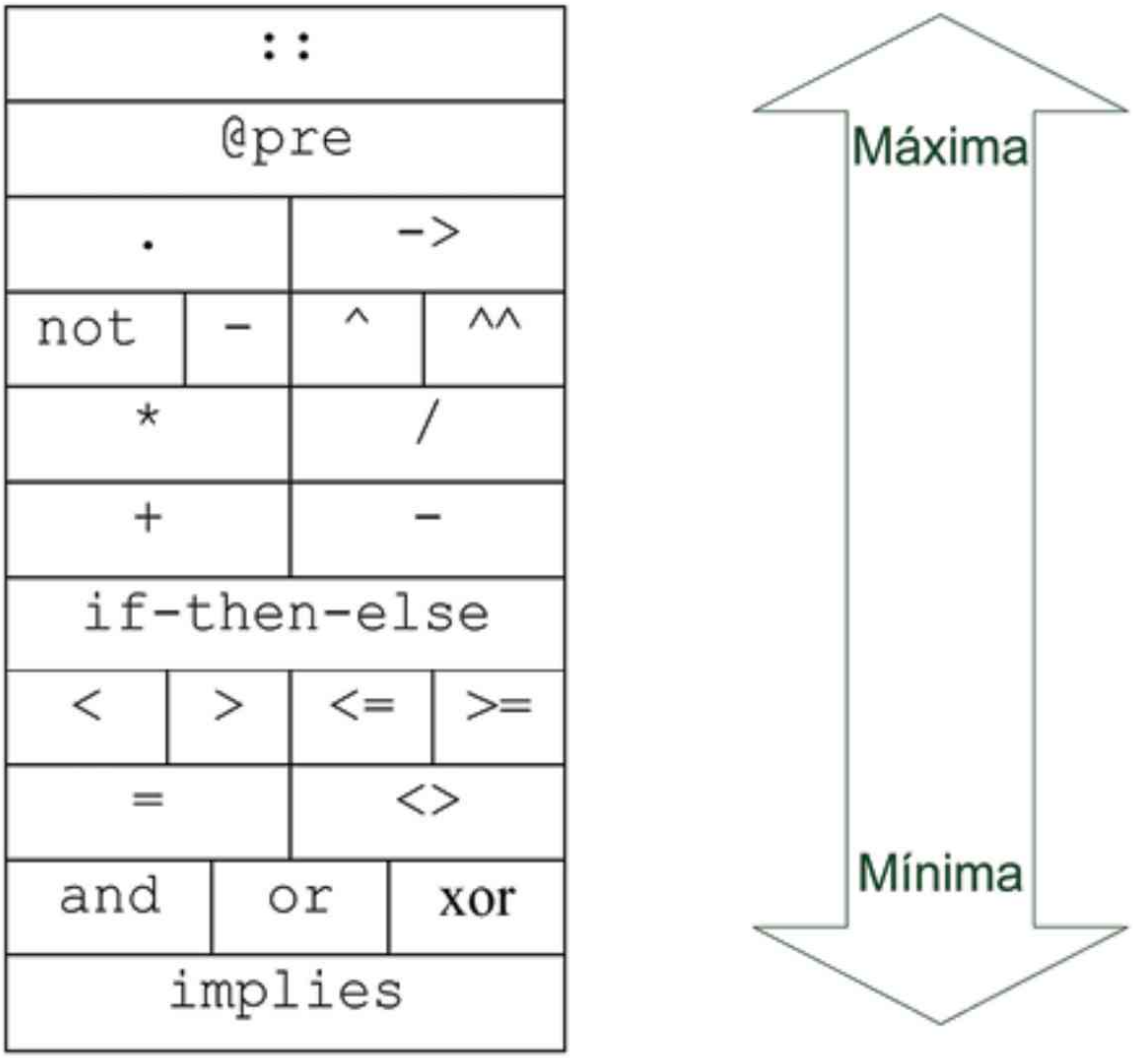


Tabla 14.2. Precedencia de operadores

En OCL es posible convertir de un tipo a otro mediante *casting* (retipado) o proyección de tipos. Para ello utilizaremos el nombre de tipo OclType, sabiendo de antemano que todos los tipos en OCL derivan del supertipo OclAny, incluido OclType

Por ejemplo:

```
Fragata.oclAsType(Barco) -- Devuelve objeto Barco
```

Convertirá el objeto *Fragata* al tipo *Barco*.

Obviamente las conversiones deben ser de subtipos a supertipos o viceversa.

Otra posibilidad es comparar dos tipos mediante la operación oclIsTypeOf que devuelve true si son iguales o false en caso contrario.

Fragata.oclIsTypeOf(Galeon) -- *Devuelve false*

Aunque para mayor precisión podemos recurrir al operador oclIsKindOf que permite determinar si el tipo del objeto en cuestión es del mismo tipo o un subtipo.

Fragata.oclIsKindOf(Barco) -- *Devuelve true*

Con la finalidad de comparar dos objetos entre sí recurriremos a los operadores matemáticos = y <>, de esta forma es posible determinar la semejanza entre objetos.

MatrizAlfa = MatrizBeta –- *Devolverá falso*

ó

MatrizAlfa <> MatrizBeta –- *Devolverá true*

Recuerde que este conjunto de tipos y operadores aquí expuesto conformarán los átomos de las expresiones comúnmente utilizadas en las consultas OCL.

modelo de referencia: "academia"

El siguiente diagrama (ver figura 14.1) nos servirá como modelo de referencia para aplicar el conjunto de expresiones OCL que se explicarán a lo largo de los próximos apartados. En la citada figura se representa un modelo de clases para el contexto de una academia de formación a alumnos. La academia consta de varios niveles en donde un alumno elige una serie de asignaturas a estudiar. Los profesores de la academia serán los responsables de realizar un conjunto de exámenes a los alumnos para certificar sus conocimientos. A cada examen se le aplicará una nota fraccionaria y la fecha en la que se efectuó. Un sistema informatizado de la academia recoge información del rendimiento de cada alumno en cada asignatura, almacenando para ello un valor con la tasa en números fraccionarios. Finalmente, en caso de un cambio de la Ley, una asignatura podría verse en el cumplimiento de incrementar el número de horas lectivas.

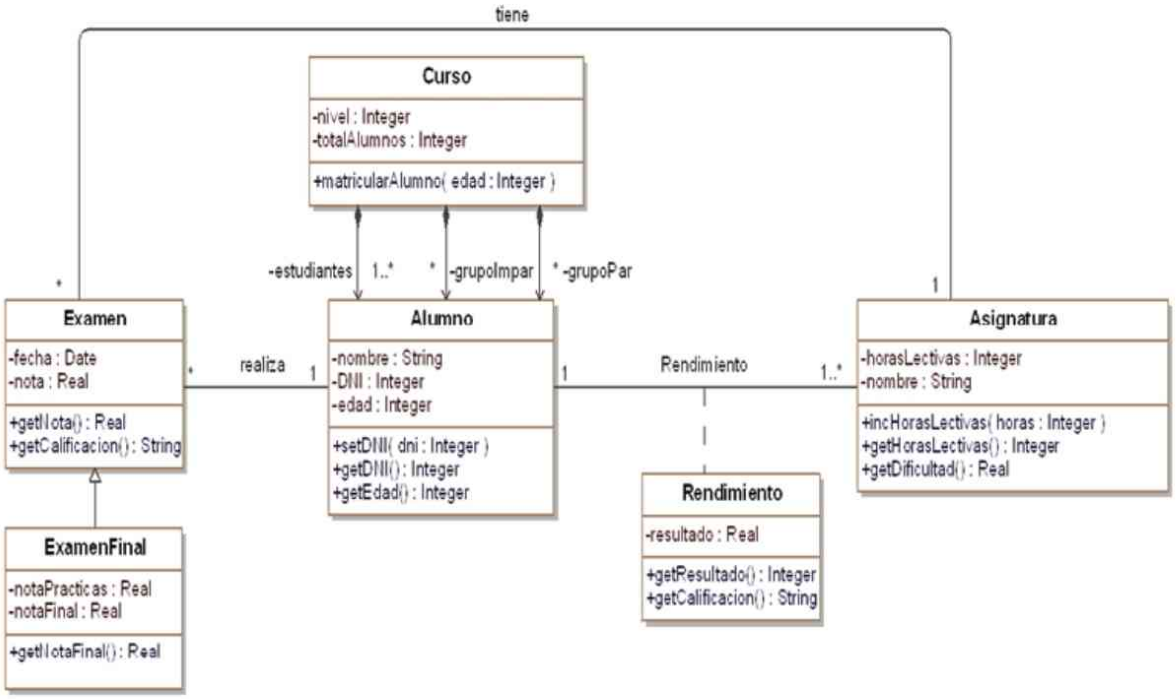


Figura 14.1. Diagrama de clases para una academia

expresiones de restricción

En OCL existen exactamente ocho tipos diferentes de expresiones clasificadas en dos categorías: de *restricción* y de *definición*. En esta sección nos centraremos únicamente en las expresiones de restricción, que son aquellas que restringen alguna propiedad en los clasificadores o en las operaciones. Las expresiones de restricción se dividen en tres clases *"inv"*, *"pre"* y *"post"* como veremos a continuación.

Expresión inv:

La operación se aplica únicamente a clasificadores y especifica que la expresión indicada es un invariante que debe ser siempre cierto para todas las instancias del clasificador.

En el ejemplo de la academia podríamos tener la siguiente situación en OCL:

Listado 14.1 Ejemplo con dos invariantes

```
context Curso
/* Debe existir al menos un alumno matriculado
y a lo sumo 250*/
inv requisitoAlumnos:
(self.totalAlumnos > 0) and
(self.totalAlumnos <= 250)
-- Se exige un nivel menor de 5
inv nivelExigido:
self.nivel < 5
```

Aquí las invariantes restringen a todas las instancias de la clase *Curso* a cumplir los requisitos impuestos a sus atributos. En este caso se exige a la academia a que haya un número determinado de alumnos y un nivel menor de cinco.

Expresión pre

La operación “pre” se utiliza únicamente para restringir el comportamiento en los métodos y debe ser siempre cierta antes de la ejecución de la operación.

```
context Curso::matricularAlumno(edad: Integer)
-- Debe ser mayor de edad
pre edadAlumno:
edad >= 18
```

Expresión post

Al igual que en el apartado anterior, la operación "post" permite aplicar una condición de restricción que debe ser cierta después de la terminación del método u operación de la clase.

context Alumno::setDNI(dni: **Integer**)
-- *El número almacenado debe ser un número válido*
post DNIválido:
(self.DNI <> NaN) **and** (**self**.DNI > 0)

En el ejemplo anterior debe cumplirse como postcondición que el DNI almacenado sea un número natural válido.

Como hemos comentado anteriormente, *pre*: y *post*: sólo pueden utilizarse en los métodos de una clase.

Operador @pre

Por medio del operador “pre” podemos acceder al valor que tenía un atributo antes de que se ejecutara el método. Una característica importante a tener en cuenta de este operador es que debe aplicarse únicamente en postcondiciones.

context Asignatura::incHorasLectivas(horas: **Integer**)
-- *Si hay un cambio en la Ley, incrementa las horas*
post cambioLey:
self.horasLectivas = **self**.horasLectivas@**pre** + horas

En este ejemplo, después de ejecutar el método en la sección “post” se puede acceder al valor previo de *horasLectivas* mediante el uso del operador @pre.

expresiones de definición

Como la misma palabra indica, las expresiones de definición nos permiten definir nuevas operaciones, iniciar valores o añadir variables globales y locales en el contexto del clasificador. Son concretamente cinco operaciones que sumadas a las tres vistas anteriormente conforman el núcleo de expresiones disponibles en OCL. Estas operaciones son: “body”, “init”, “def”, “let” y “derive”. Empezaremos a explicar la utilización de cada una de ellas.

Expresión body

La expresión "body" se aplica a las operaciones de consulta (*get*) y permite definir el valor que devuelve el método.

Listado 14.2 Ejemplo de utilización de la expresión body

```
context Examen::getCalificacion():String
/* Retorna la nota alfanumérica o si el alumno no se ha presentado */
body notaDelAlumno:
if (self.nota >= 0) and (self.nota < 5) then
"suspenso"
else if (self.nota >= 5) and (self.nota < 7) then
"aprobado"
else if (self.nota >= 7) and (self.nota < 9) then
"notable"
else if (self.nota >=9) then
"sobresaliente"
else "no presentado"
endif
```

En el ejemplo anterior se aplica el operando "body" con la finalidad de devolver la calificación alfanumérica correspondiente al valor numérico de la nota.

Cada *String* definido al final de las sentencias condicionales del listado 14.2 representa el valor alfanumérico devuelto por la operación, por lo que no es necesario en OCL el uso de una sentencia de retorno.

Expresión init

Utilizaremos “init” cuando sea necesario inicializar el valor de algún atributo ya que solo se aplicará a éste.

context Asignatura::horasLectivas

init:

4

Expresión def

Mediante la expresión "def" es posible añadir atributos y operaciones auxiliares a un clasificador de UML. Su expresión opuesta es "let".

Listado 14.3 Ejemplo de definición de dos variables

```
context Asignatura
-- Define un invariante
inv:
horasLectivas >= 1
-- Define nuevas variables de aplicación de la Ley
def:
cambioLey = false
hLey = -2
context Asignatura::incHorasLectivas(horas: Integer)
-- Si hay un cambio en la Ley, incrementa las horas
post:
if (cambioLey = true) then
self.horasLectivas = self.horasLectivas@pre + horas
else
self.horasLectivas = self.horasLectivas@pre + hLey
endif
```

Como se puede apreciar en el listado 14.3 es posible definir nuevas variables y métodos en el contexto global del código OCL. En el ejemplo visto anteriormente se definen dos nuevas variables (*cambioLey* y *hLey*) que serán utilizadas fuera del ámbito, más concretamente en el contexto de la operación *incHorasLectivas*.

Expresión let

Esta expresión permite declarar variables locales dentro de una expresión OCL. Hay que tener en cuenta que el acceso a la variable únicamente tiene lugar en el ámbito de una expresión, por lo tanto no puede ser accedida desde otros contextos.

```
context Asignatura::incHorasLectivas(horas: Integer)
-- Si hay un cambio en la Ley, incrementa las horas
post cambioLey:
let inicialHoras:Integer = self.horasLectivas@pre in
self.horasLectivas = inicialHoras + horas
```

En este ejemplo se crea una nueva variable local llamada *inicialHoras* que tomará el valor previo de las horas lectivas y se aplicará en la expresión definida después de la palabra reservada “in”.

Expresión derive

Con la expresión “derive” podemos asignar valores a atributos derivados. Por ejemplo, en el siguiente listado se especifica el valor de la dificultad de una asignatura mediante el cálculo del porcentaje de horas lectivas.

context Asignatura::dificultad:**Real**
derive:
-- *Se especifica un nuevo valor a dificultad*
(horasLectivas / 20) * 100
context Asignatura::getDificultad():**Real**
body:
dificultad

Colecciones

Nociones básicas

OCL proporciona estructuras de datos para colecciones. Estas colecciones agrupan un número determinado de objetos de cualquier tipo y proporcionan un repertorio de operaciones de acceso, consulta y eliminación.

OCL ofrece cuatro tipos de colecciones: **Set** (para conjuntos), **OrderedSet** (para conjuntos ordenados), **Bag** (cojuntos de datos repetidos y desordenados) y **Sequence** (secuencia de objetos ordenados)[32].

A modo de resumen proporcionamos la siguiente tabla con las características principales de cada tipo de colección.

Tipo de colección	Valores duplicados	Valores ordenados
Set	No	No
OrderedSet	No	Sí
Bag	Sí	No
Sequence	Sí	Sí

Tabla 14.3. Estructuras de datos de colección en OCL y sus propiedades

Por ejemplo un tipo *Set* podrían ser los diferentes tipos de colores:

```
Set{"Verde", "Rojo", "Azul"}
```

y un *OrderedSet* las notas musicales:

```
OrderedSet{"Do", "Re", "Mi", "Fa", "Sol", "La", "Si"}
```

La notación para las operaciones aplicadas a las colecciones tiene la siguiente sintaxis:

```
colección->operación([parámetros]):valor devuelto
```

en donde *colección* hace referencia a una de las estructuras de datos vistas en

la tabla 14.3 y donde *operación* se refiere a alguno de los métodos aplicables a cada uno de estos tipos. Finalmente el valor devuelto hace referencia a un objeto colección del mismo tipo que la colección de entrada o un valor de tipo primitivo (Boolean/Integer).

Es importante saber que las operaciones aplicadas a una colección nunca cambian los valores de la misma, sino que devuelven otra colección nueva.

Operaciones básicas

En los siguientes apartados veremos cada una de las diferentes operaciones aplicadas a colecciones y ordenadas por su funcionalidad.

conversión

Cada colección puede ser transformada o convertida a otro tipo de objeto colección por medio de las operaciones de conversión.
Por ejemplo si tenemos la siguiente colección en forma de *Bag*:

```
Bag{1,2,2,3}->asSet()
```

Se transformará en un Set{1,2,3}
En general las operaciones de transformación tienen la siguiente sintaxis:

Operación de conversión	Devuelve
colección->asSet():Set()	**Set**
colección->asOrderedSet():OrderedSet()	**OrderedSet**
colección->asBag():Bag()	**Bag**
colección->asSequence():Sequence()	**Sequence**

Tabla 14.4. Operaciones de conversión y sus valores devueltos

comparación

Las operaciones de comparación permiten comparar dos colecciones elemento a elemento. Tienen la siguiente sintaxis:

```
colección = colección ó colección <> colección
```

Por ejemplo:

```
Set{1,2,3} = Set {1,2,3,4} devuelve false
```

y

Set{1,2,3,4} <> Set{1,2,3,4} devuelve *false*

En general, la operación "=" aplicada a *Set, Bag OrderedSet* y *Sequence* devolverá *true* si ambos operandos contienen los mismos elementos y *false* en caso contrario.

consulta

Por medio de las operaciones de consulta podemos indagar sobre las propiedades y los diferentes valores contenidos en las colecciones, por ejemplo, averiguar el número total de elementos o si existe un determinado objeto dentro de un conjunto.

En este caso, si aplicamos la operación *size* nos devolverá el número de elementos en la colección:

Sequence{1,2,3,4,5}->size()

nos proporciona el valor numérico 5.

Otra posibilidad es averiguar si un determinado objeto pertenece a una colección:

Bag{"Juan","Pedro","Beatriz","María"}->includes("Juan")

devolverá *true*.

A continuación se proporciona una tabla con un listado de funciones de consulta aplicables a las colecciones:

Operación de consulta	Significado
colección->size():Integer	Devuelve número de elementos
colección->sum():Objeto	Devuelve la suma

	de los elementos si soportan la operación "+"
colección->isEmpty():Boolean	Devuelve *true* si la colección está vacía
colección->notEmpty():Boolean	Devuelve *true* si la colección *no* está vacía
colección->includes(x: Objeto):Boolean	Devuelve *true* si la colección incluye el objeto "x"
colección->excludes(x: Objeto):Boolean	Devuelve *true* si la colección excluye el objeto "x"
colección->includesAll(x: Colección):Boolean	Devuelve *true* si colección incluye a todo "x"
colección->excludesAll(x: Colección):Boolean	Devuelve *true* si colección excluye a todo "x"
colección->count(x: Objeto):Integer	Devuelve el número de veces que se encuentra "x" en la colección

Tabla 14.5. Operaciones de consulta

acceso

Las operaciones de acceso se aplican únicamente a los tipos *OrderedSet* y a *Sequence* para obtener el valor ubicado en una determinado índice de la

colección.

De esta forma la aplicación de la expresión OCL:

```
OrderedSet{23,45,67,80,123}->indexOf(123)
```

nos devolverá 5, puesto que las colecciones comienzan a numerarse en uno.

En la siguiente tabla se recoge el resumen de las operaciones de acceso:[33]

Operaciones de acceso[33]	Significado
colección->first():Objeto	Devuelve el primer elemento
colección->last():Objeto	Devuelve el último elemento
colección->at(i: Integer):Objeto	Devuelve el elemento en la posición "i"
colección->indexOf(x: Objeto):Integer	Devuelve la posición del objeto "x"

Tabla 14.6. Operaciones de acceso

algebraicas

Las operaciones de álgebra de conjuntos nos permitirán aplicar acciones de selección de elementos en las colecciones. Para ilustrar las operaciones que veremos a continuación utilizaremos los diagramas de *Venn* como base pedagógica para explicar las operaciones de álgebra de conjuntos.

Unión

La unión viene expresada por el siguiente diagrama de *Venn*:

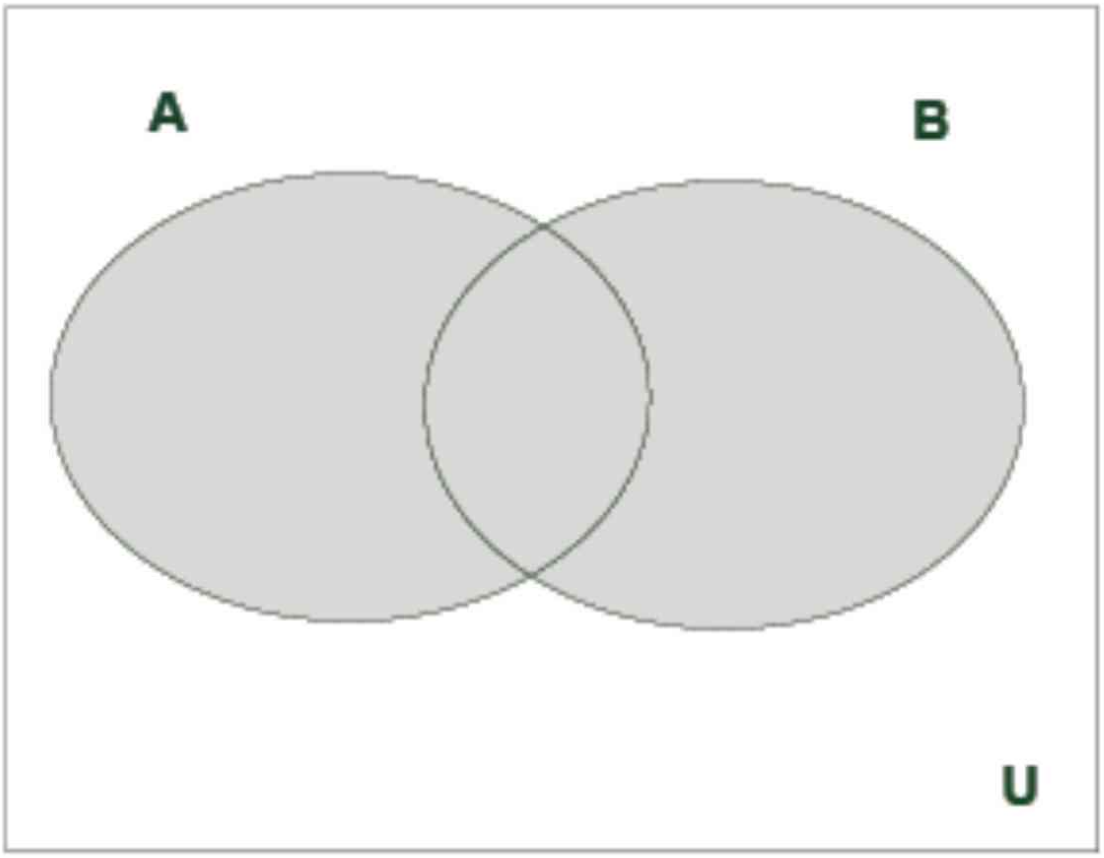


Figura 14.2. Unión (A □ B)

mientras que en OCL se representa por la siguiente expresión:

```
A->union(B):C
```

y donde A, B y C son aplicables a *Set*, *OrderedSet*, *Bag* y *Sequence*.
Por ejemplo si tuviéramos:

```
Sequence{“a”, “b”, “b”}->union(Sequence{“d”, “e”})
```

El resultado es Sequence{“a”,“b”,“d”,“e”} y donde los tipos de datos deben ser, obviamente, iguales.

Intersección

Se representa como:

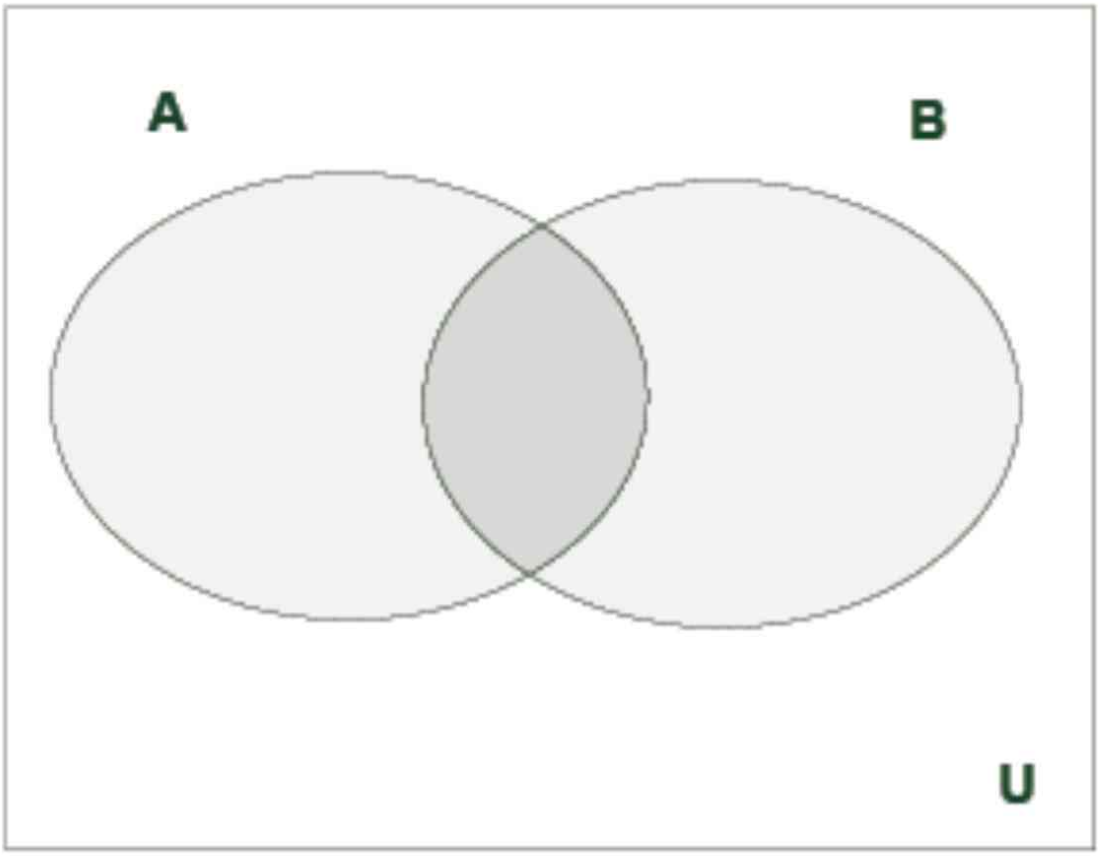


Figura 14.3. Intersección (A □ B)

En OCL se expresa de forma:

```
A->intersection(B):C
```

y donde A, B y C deben ser del tipo *Set* u *OrderedSet.*
Por ejemplo:

```
Set{“a”, “b”, “c”}->intersection(Set{“a”, “h”, “j”})
```

Dará como resultado Set{“a”}.

Diferencia

La diferencia de conjuntos devuelve el conjunto original menos los elementos contenidos en el conjunto destino.

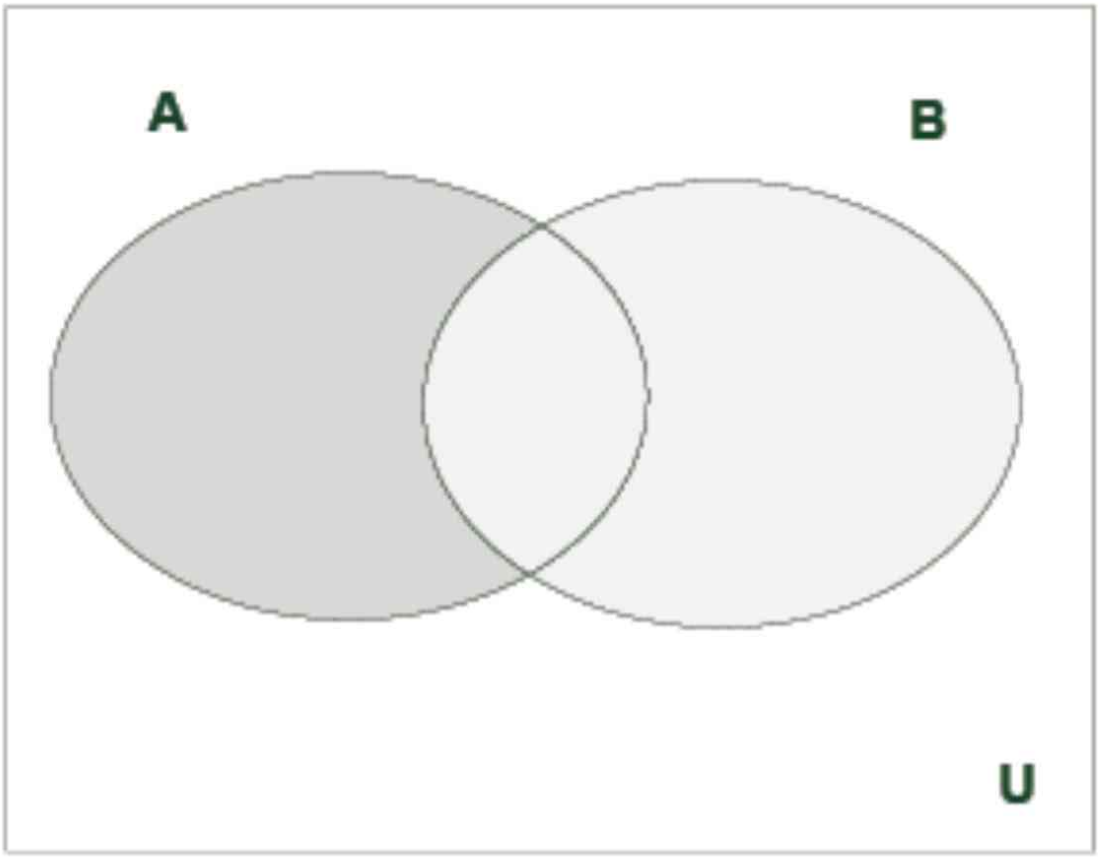


Figura 14.4. Diferencia (A - B)

La sintaxis OCL es la siguiente:

A-B:C

y en donde A, B y C pueden ser *Set* u *OrderedSet*.

Set{“b”, “c”, “a”}-Set{“a”}

Devolverá Set{“b”,“c”}.

Diferencia simétrica

La diferencia simétrica es la unión de los dos conjuntos menos la intersección de ambos conjuntos.

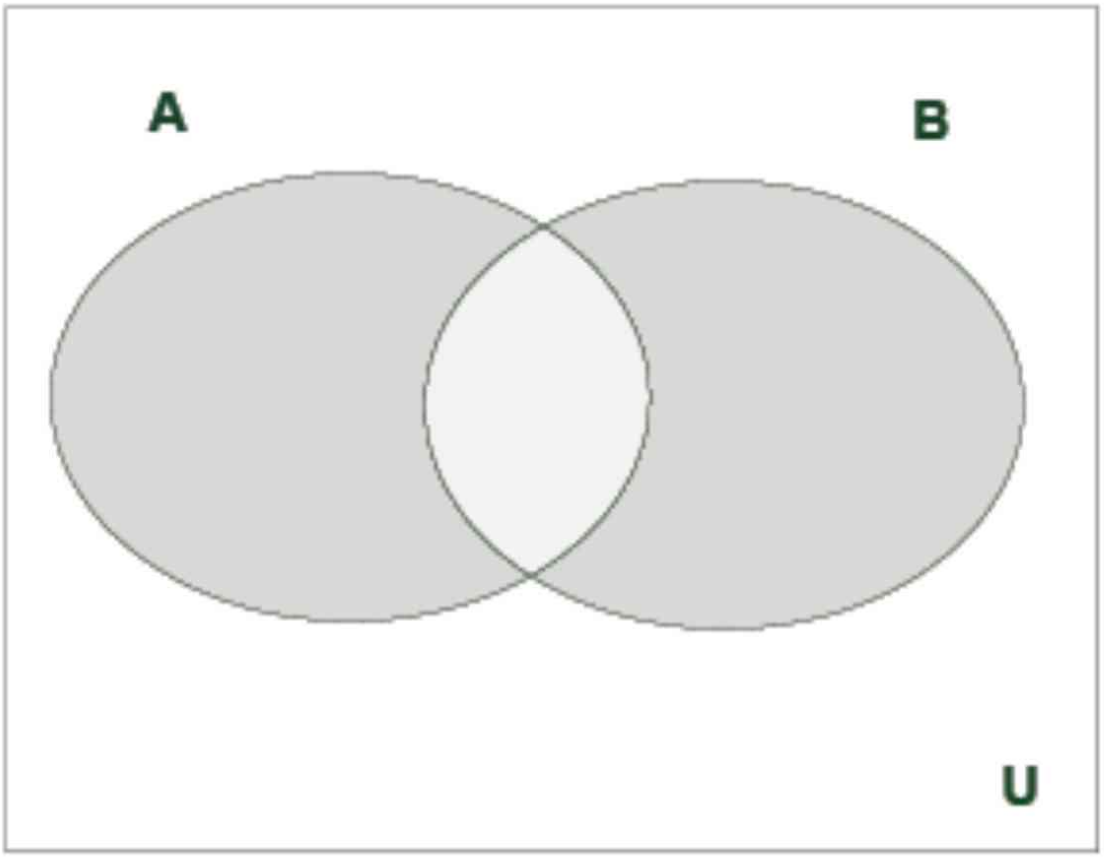


Figura 14.5. Intersección (A □ B) - (A □ B)

Se representa mediante OCL como:

```
A->symmetricDifference(B):C
```

Donde A, B y C pueden ser *Set* u *OrderedSet*.
A modo de ejemplo supongamos el siguiente caso:

```
Set{“a”, “b”, “c”}->symmetricDifference(Set{“b”, “d”})
```

Se obtendrá como resultado Set{“a”,“c”,“d”}.

Producto cartesiano

El producto cartesiano de dos conjuntos A y B (A × B) contiene todos los pares ordenados (a_i, b_j) resultado de la combinación de cada elemento de A

con cada elemento de B.
Por ejemplo si tenemos los siguientes elementos:
A = {1,2,3} y B = {"a","b"} su producto cartesiano es:
A × B = {(1,a), (1,b), (2,a), (2,b), (3,a), (3,b)} = 3 x 2 = 6.
En OCL esta operación se representa como:

```
A->product(B):C
```

En donde A y B pueden ser del tipo *Set*, *OrderedSet*, *Bag* y *Sequence* y C es obligatoriamente un *Set* del tipo *Tuple*[34] que contiene los binomios del producto cartesiano.
Ejemplo:

```
Sequence{1,2,3}->product(Bag{"a","b"})
```

Esto dará como resultado el conjunto de tuplas:

```
Set{Tuple{1,"a"},Tuple{1,"b"},Tuple{2,"a"},Tuple{2,"b"},Tuple{3,"a"},Tuple{3,"b"}}
```

Inclusión / exclusión
En OCL es posible la inclusión por medio de la operación:

```
colección->including(x: Objeto):colección
```

por ejemplo:

```
Set{"a", "b", "c"}->including("d")
```

devolverá un Set{"a", "b", "c", "d"}.
de igual forma se puede utilizar la operación:

colección->excluding(x: Objeto):colección

Inserción

Es posible añadir un elemento al final de una colección con la operación:

A->append(x: Objeto): A

y al inicio con el operador:

A->preppend(x: Objeto): A

o en una posición indexada mediante:

A->insertAt(i: Integer, x: Objeto): A

En todos los casos A debe ser del tipo *Sequence* u *OrderedSet*.

Extracción

La extracción de una subsecuencia a partir de una secuencia se realiza con la operación:

Sequence->subSequence(i: Integer, j:Integer): Sequence

Siempre y cuando se aplique a una tipo *Sequence*.
Igualmente para la extracción de un subconjunto ordenado (*OrderedSet*) se procede de forma análoga:

OrderedSet->subOrderedSet(i: Integer, j:Integer): OrderedSet

iteración

Gracias a las operaciones de iteración es posible recorrer todos los elementos

de una colección mientras se comprueba una expresión condicional.
La estructura de este tipo de operaciones para colecciones está formada por:

colección->opIteración([variable:Tipo]|expresión)

En la expresión anterior, la colección formada por un *Set*, *OrderedSet*, *Bag* o *Sequence* va seguida de una operación de iteración cuyos parámetros son la variable de iteración separada mediante " | " de la expresión de guarda. En dicha expresión se podrá utilizar opcionalmente la variable de iteración.

select, reject, sortedBy y collect
En algunas ocasiones es necesario consultar una colección para extraer un subconjunto de la misma que cumpla una determinada condición. Para esta finalidad disponemos de los operadores *Select*, *Reject*, *SortedBy* y *Collect*
La utilización de *Select* permite devolver una colección que contiene aquellos elementos para los cuales se cumple la condición. Por ejemplo:

context Alumno
inv:
self.examen->**select**(e | e.nota > 5)

ó en versión reducida:

self.examen->**select**(nota > 5)

devolverán una colección con todos los alumnos cuya nota sea mayor que cinco. Fíjese que para ello se utiliza el atributo "examen" de la clase *Alumno* que contiene una referencia al array de objetos del tipo *Examen*. Esto es debido a la asociación uno-a-muchos entre "Alumno" y "Examen".
De forma diametralmente opuesta opera *Reject*, selecciona los elementos que

precisamente no cumplen la condición. Del mismo modo *SortedBy* devuelve una colección de elementos que además de cumplir la guarda también están ordenados. Por último *Collect* devuelve un Bag con el resultado de aplicar la guarda sobre cada elemento de la colección. Por ejemplo:

self.examen->**collect**(nota = 10)

devuelve todos los exámenes con nota igual a diez incluso más de una vez.

exist, forAll, isUnique

Suponga el siguiente ejemplo:

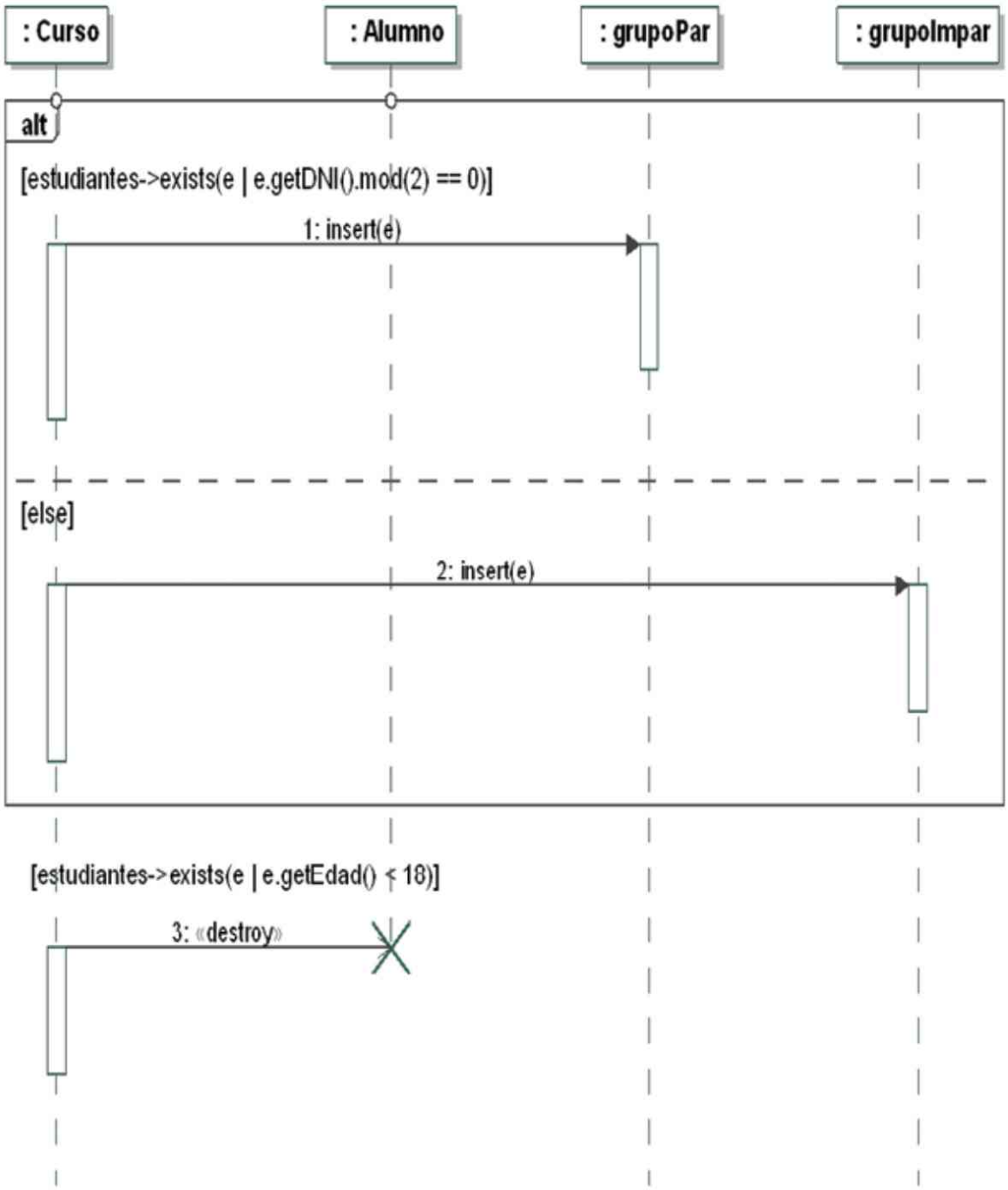


Figura 14.6. Diagrama de secuencias para inclusión de alumno en grupo

En el anterior diagrama de secuencias se muestra el caso de inclusión de un alumno en un grupo u otro dependiendo de la numeración de su DNI. En concreto, cuando el alumno posee un DNI par se le incluye en el grupo de la misma denominación, mientras que si lo posee impar se le incluye en el grupo contrario.

En el primer mensaje del fragmento combinado *alt* muestra la llamada a la iteración OCL *exists*. Lo que realiza esta operación es únicamente recorrer todos los elementos de la colección que cumplan el criterio de guarda y devolver *true*. En este caso accede a la colección de estudiantes de la composición de *Curso* para obtener todos los alumnos con DNI par e insertarlos en la colección de destino. Finalmente, en la última interacción entre *Curso* y *Alumno* se vuelve a invocar la operación *exists* con la finalidad de borrar de la academia todos los alumnos menores de edad.

Otra posibilidad de iteración en colecciones es la operación *forAll*. Dicha operación permite comprobar si todos los elementos de una colección cumplen un determinado criterio. En caso de que todos los elementos de la colección cumplan la condición el operador devolverá *true*.

Por ejemplo supongamos el siguiente caso:

```
context Alumno
inv:
self.asignaturas->forAll(a | a.horasLectivas = 2)
```

devolverá cierto si todas las asignaturas tienen dos horas lectivas.

Y si quisiéramos averiguar si todos los nombres de las asignaturas son distintos escribiríamos lo siguiente:

```
context Alumno
inv:
self.asignaturas->forAll(a1, a2 | a1 <> a2 implies a1.nombre <> a2.nombre)
```

Debe considerarse que esta última expresión realiza la comprobación a todos los pares del producto cartesiano de los nombres de las asignaturas.

Por último *isUnique* permite comprobar si la expresión de guarda es única

para todos los valores de la variable de iteración. Por ejemplo:

```
context Curso
inv:
self.estudiantes->isUnique(e | e.DNI)
```

devolverá *true* si el DNI de un alumno no está repetido.

navegación

Gracias a la navegación en OCL podemos ir desde un objeto fuente a otro objeto destino mediante el acceso a clasificadores, atributos, operaciones de consulta y asociaciones.

Por ejemplo, para acceder a un atributo de un clasificador como la clase *Curso* utilizaremos la palabra reservada *self*. De esta forma si quisiéramos acceder al atributo *nivel* utilizaríamos la siguiente expresión:

self.nivel

de igual manera si quisiéramos acceder a un método haríamos lo siguiente:

self.matricularAlumno(edadAlumno)

También podemos utilizar las asociaciones para movernos por el diagrama. (Si consideramos que estamos en el contexto *Alumno)*:

self.asignaturas.nombre

donde *asignaturas* es un conjunto (*Set*) de objetos *Asignatura,* pero al acceder a *nombre* la consulta nos devolverá un colección (*Bag*) de nombres de asignaturas. Es decir:

self.asignaturas (Devolverá un *Set* de objetos Asignatura)
self.asignaturas.nombre (Devolverá un *Bag* con todos los nombres de asignaturas)

Es importante recordar que con la finalidad de navegar a través de las asociaciones debemos utilizar siempre el operador punto. En general, cuando la multiplicidad de la asociación es mayor que uno, este operador nos devolverá

el conjunto de objetos (*Set*) al cual hace referencia, sin embargo si la multiplicidad es 1, éste nos devolverá un único objeto.

Por ejemplo si quisiéramos acceder al método *getHorasLectivas()* de la clase *Asignatura* desde una instancia de *Examen* utilizaríamos la siguiente expresión:

```
self.alumno.asignaturas.getHorasLectivas()
```

que retorna una colección (Bag) de valores devueltos por el método get*HorasLectivas()*. Visto de forma más detallada apreciamos que a partir de un examen se obtiene el alumno que lo ha realizado en forma de *Set* (cardinalidad 1). De ahí se accede al *Set* de asignaturas (cardinalidad *) para finalmente devolver un *Bag* de *Integers* con los resultados concretos de las horas lectivas en esas asignaturas en las que el alumno se encuentra matriculado.

ocl en los diagramas de estado

La utilidad de OCL también se extiende a otros diagramas UML, especialmente a diagramas de secuencias, actividades y estados.

OCL proporciona una importante operación de comparación para los diagramas de estado.

oclInState(x: OclState):Boolean

que devuelve un *booleano* indicando si un determinado contexto se encuentra en el estado "x".

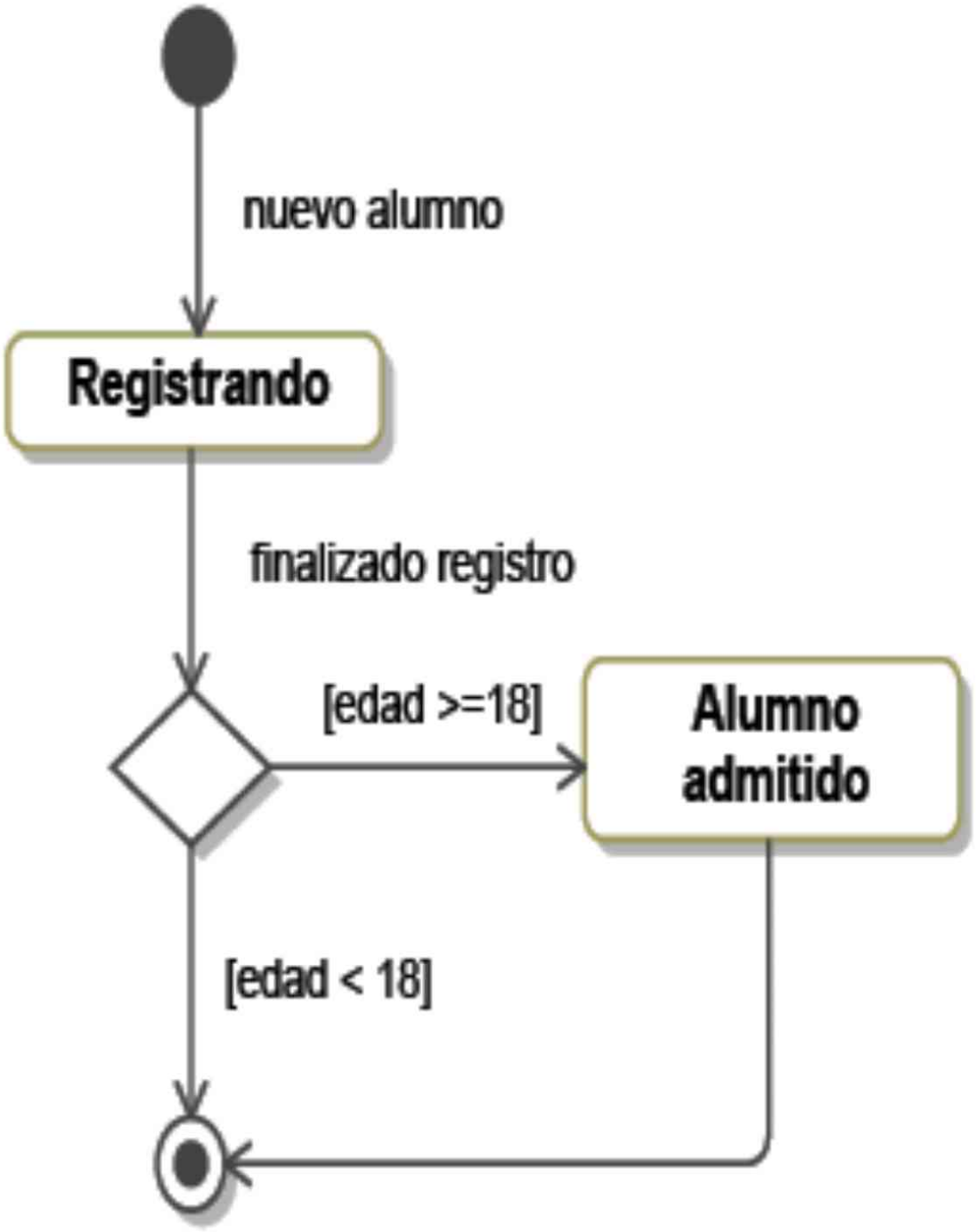


Figura 14.7. Diagrama de estados del proceso de matriculación

En el ejemplo de la figura 14.7 se recogen los estados por los que se transita para matricular a un alumno y que están definidos en el comportamiento de la clase *Curso*. Por tanto, una posible sentencia OCL dentro del contexto de esta clase puede ser la siguiente:

```
context Curso
inv:
oclInState(AlumoAdmitido) implies (self.edad >= 18)
context Curso::matricularAlumno(edad:Integer)
post:
oclInState(Registrando) implies (self.edad >= 18)
```

En el primer contexto de la sentencia OCL se especifica un invariante para la clase *Curso* que debe cumplirse cuando se encuentre en el estado *Alumno admitido*. De igual forma, en el segundo contexto se define una postcondición para la operación *matricularAlumno* en el caso de encontrarse en el estado *Registrando*.

caso de estudio: ajedrez

En este apartado veremos la aplicación práctica del lenguaje de restricciones OCL a algunos casos basados en el modelo del diagrama de clases del juego de ajedrez:

```
package Ajedrez
context Heuristica
inv:
(self.puntuación >= 0.0)
endpackage
```

En este ejemplo el uso de la expresión "*inv*" permite aplicar un invariante al atributo *puntuación* de la clase *Heurística* ubicada en el paquete *Ajedrez*. Dicho invariante impone la restricción de que el valor de la puntuación sea mayor o igual que cero.

```
package Gestion_juego
context Control_juego::on_destino(x_dest:Integer, y_dest:Integer)
pre:
(x_dest >= 0) and (y_dest >= 0)
endpackage
```

En este caso se aplica la precondición "*pre*" a la operación *on_destino()* de la clase *Control_juego* ubicada en el paquete *Gestion_juego*. La precondición impone la restricción de ser mayor o igual que cero a los parámetros del método.

```
package Ajedrez
context Pieza
inv:
```

```
Pieza::allInstances->forAll(p1, p2 | p1 <> p2 implies
(p1.posicion.fila <> p2.posicion.fila) and
(p1.posicion.columna <> p2.posicion.columna)
endpackage
```

El invariante utiliza la operación de ámbito de clase *allInstances*[35], que devuelve una colección *Set* con el conjunto de todas las instancias de clase del tipo *Pieza*. Finalmente, la operación *forAll* realiza el producto cartesiano de todos los pares de filas y columnas para comprobar que son distintos entre ellos. En el ejemplo anterior se comprueba que no es posible tener dos piezas en la misma posición.

caso de estudio: mercurial

En la aplicación *Mercurial* también podemos imponer restricciones OCL a distintos elementos y clasificadores del modelo de clases:

```
package GestorCliente
context Usuario
inv:
Usuario::allInstances->isUnique(user | user.login)
endpackage
```

Como en el caso del ajedrez, *allInstances* devuelve un *Set* con el conjunto de todas las instancias del objeto Usuario; mientras que la operación de colección *isUnique* devolverá *true* si el *login* del usuario es único en todas las instancias de *Usuario*.

```
package Interprete
context FacahadaInterprete::interpretaComando(comando:String)
pre:
(comando.size() > 0) and (comando.size() < 255)
endpackage
```

La precondición impone la restricción al método *interpretaComando()* para que la cadena del parámetro *comando* esté comprendida entre 0 y 255 caracteres.

```
package GestorArchivos
context Fichero
inv:
(self.fichero <> null) implies
(self.nombre.size() <> 0)
```

endpackage

Si un fichero es distinto de *null* en la clase *Fichero* del paquete *GestorArchivos*, implica que debe contener un nombre identificativo que no esté vacío.

caso de estudio: servicio de cifrado Remoto

En el caso de la aplicación de cifrado remoto, tanto en el lado cliente como en el lado servidor, se puede incluir en el cuerpo de la operación inicio de menú una consulta para determinar el tipo de opción de cifrado.

```
package Menu
context MenuServidor::getModo(opcion: Integer):String
body opcionTexto:
if (opcion = 1) then "Cifrado simétrico"
else if (opcion = 2) then "Cifrado híbrido"
else if (opcion = 3) then "Poner en espera"
else if (opcion = 4) then "Salir"
endif
endpackage
```

En el paquete cliente se inicializa el número de accesos a 0 para el atributo del usuario novel.

```
package Cliente
context Usuario::numero_accesos
init:
0
endpackage
```

También, en el lado cliente, se establece la precondición para el tamaño del mensaje y el tamaño de la clave de cifrado simétrico. Por último, se incrementa el número de accessos a la operación de cifrado para el usuario novel en la postcondición.

```
package Cliente
```

context Usuario::cifra_simetrico(msg:**String**, clave:**String**):**String**

pre requisitos:

(msg.size() > 0) **and** (clave.size() = 16)

post accesos:

self.numero_accesos = **self**.numero_accesos@**pre** + 1

endpackage

---

31 El signo " + " representa la repetición del elemento una o muchas veces.

32 Todas las colecciones comienzan a numerarse en 1 y terminan en n.

33 Colección debe de ser únicamente "OrderedSet" o "Sequence" para las tres primeras.

La operación *indexOf* (x) únicamente pude aplicarse a "OrderedSet".

34 Estructura de datos de OCL. Alberga una estructura de tupla con campos.

35 La operación OCL allInstances() es estática y devuelve todos los objetos instanciados. No se recomienda su uso debido al poco soporte proporcionado por algunas herramientas OCL.

ingeniería directa en java

«Los límites de mi lenguaje son los límites de mi mundo».

(Ludwig Wittgenstein: Tractatus, 5.6).

Entramos en la fase de implementación del ciclo de vida de un proyecto software. Después de entender las nociones de diseño detallado en UML, pasamos a estudiar la parte relacionada con la transformación de los elementos de los diagramas a código Java. Si bien la parte de *ingeniería directa* no es un aspecto tratado expresamente en la especificación de la OMG y en otros libros de esta materia, nosotros haremos hincapié en sus fundamentos. En la práctica dicha transformación depende de cada lenguaje de programación y sus características de orientación a objetos, por lo que es importante que conozca tanto como pueda sus características en dicho ámbito.

En este capítulo veremos cómo aplicar la *ingeniería directa* a cada uno de los diagramas estáticos y de comportamiento vistas anteriormente, con la idea de crear código ejecutable siguiendo los preceptos de UML de una forma eficaz y eficiente.

Java es un lenguaje más puramente orientado a objetos que C++ y se presta mejor a la implementación de los diseños realizados en UML. Otros aspectos más puramente orientados a objetos que se han incluido en Java, como las *interfaces* o los *paquetes*, nos permitirán entender mejor los conceptos aprendidos en temas anteriores. Además, Java tiene una sintaxis más asequible e intuitiva que C++ por lo que se ha decidido comenzar por este lenguaje como modelo pedagógico y de referencia para entender los conceptos clave de la fase de postmodelado.

diagramas de clases

Las clases, como se trató en el capítulo seis, son modelos estáticos que permiten representar un concepto del dominio. Mediante sus atributos y operaciones (métodos) es posible definir una unidad de programa que modelará la estructura y comportamiento de un conjunto de objetos: las instancias de clase. En las siguientes secciones trataremos la conversión de cada una de las estrategias de un diagrama de clases a código Java.

Representación de una clase

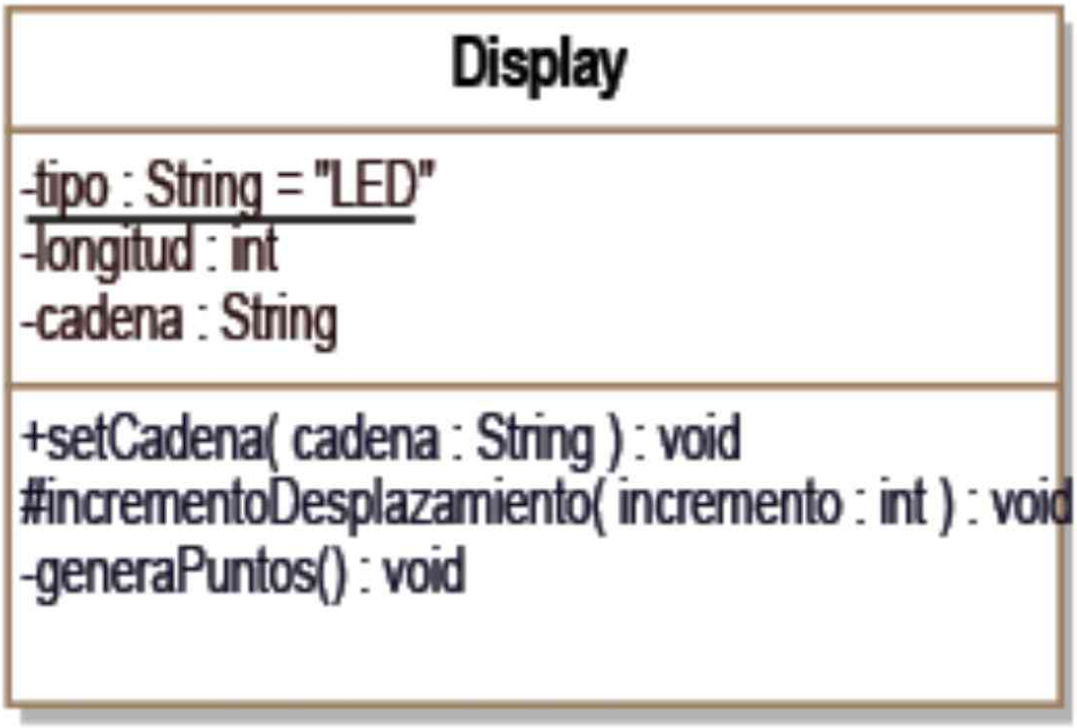


Figura 15.1. Ejemplo de clase en UML

Listado 15.1 Implementación de la clase Display

```
// Implementación de la clase Display
public class Display
{
private static String tipo = “LED”;
private int longitud;
private String cadena;
// Constructor
public void Display()
{
}
```

```
public void setCadena(String cadena)
{
this.cadena = cadena;
System.out.println(cadena);
}
protected void incrementaDesplazamiento(int incremento)
{
}
private void generaPuntos()
{
}
}
```

El listado 15.1 implementa el modelo de clase que se especificó en la figura 15.1. En el código se muestra los atributos privados junto con el constructor de la clase y sus tres métodos *public*, *protected* y *private*[36].

Asociaciones

Con una asociación podemos especificar que una clase tiene visibilidad o referencia a nivel de atributo a otros objetos. Por ejemplo, en las siguientes asociaciones un atributo mantiene una referencia/s hacia otro/s objeto/s.



Figura 15.2. Asociación simple

```
class Sistema
{
private Dispositivo dispositivo;
}
```

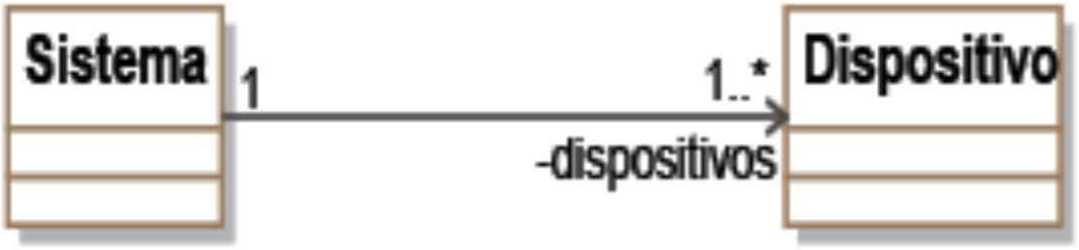


Figura 15.3. Asociación con cardinalidad 1..*

```
class Sistema
{
private Vector<Dispositivo> dispositivos = new Vector<Dispositivo>();
```

}



Figura 15.4. Asociación con cardinalidad restringida

```
public class Sistema
{
private Dispositivo[] dispositivos = new Dispositivo[8]
}
```

Herencia

Mediante la herencia podemos aplicar relaciones jerárquicas entre clases, derivando los atributos de la clase padre a las clases descendientes. Java no permite la herencia múltiple entre clases, ya que esta posibilidad está reservada únicamente a las *interfaces*. En el siguiente ejemplo mostraremos el uso de la herencia entre clases mediante la palabra reservada *extends*:

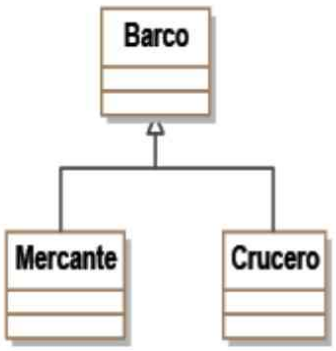


Figura 15.5. Herencia

```
public class Barco
{
}
class Mercante extends Barco
{
}
class Crucero extends Barco
{
}
```

Mientras que la herencia de *interface* se realiza con la palabra reservada *implements*:

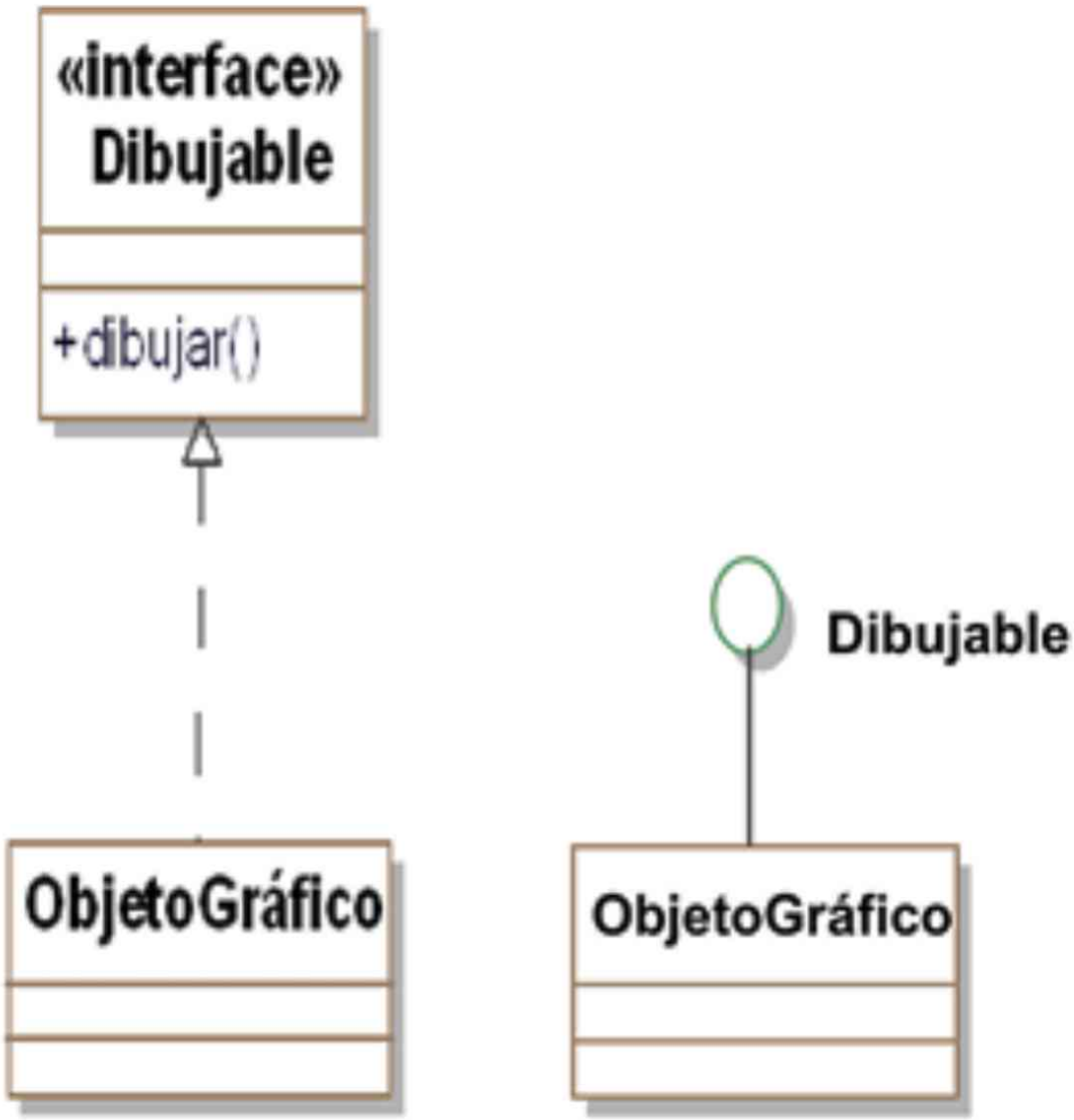


Figura 15.6. Herencia de interface (dos versiones de representación)

Listado 15.2 Implementación de interface

```
interface Dibujable
{
public void dibujar();
}
class ObjetoGrafico implements Dibujable
```

```
{
public void dibujar()
{
(...)
}
}
```

Agregación

La agregación implica una asociación en el que el todo contiene a las partes de forma no estricta, es decir, las partes contenidas pueden ser compartidas por varias partes contenedoras. De igual manera, las partes contenidas tienen su propio ciclo de vida como objetos, por lo que la parte contenedora no es responsable de su gestión.

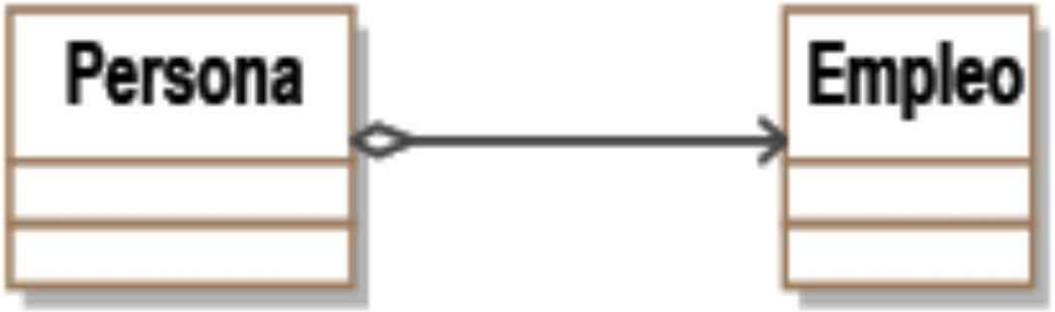


Figura 15.7. Agregación simple

```
public class Persona
{
private Empleo empleo;
}
```

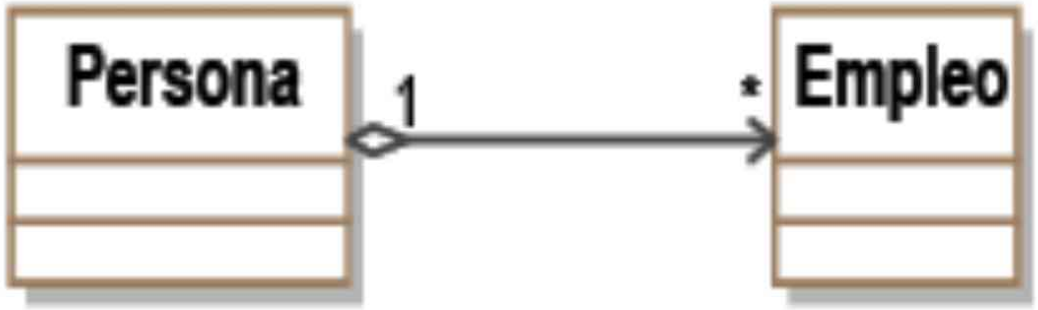


Figura 15.8. Agregación múltiple

Una posibilidad de implementación es mediante un *HashSet*, aunque puede implementarse también con **vectores** para un acceso indexado.

```
public class Persona // Alternativa 1
{
private Set<Empleo> empleos = new HashSet<Empleo>();
}
public class Persona // Alternativa 2
{
private Vector<Empleo> empleos = new Vector<Empleo>();
}
```

Composición

Como se explicó en el apartado 6.2.2 la composición implica que el concepto de la parte contenedora no tiene sentido sin las partes contenidas, por ejemplo un automóvil con ruedas, motor y chasis. La parte contenedora es responsable de la gestión del ciclo de vida de las partes contenidas, lo que implica que al destruir el objeto contenedor deben destruirse sus objetos contenidos.

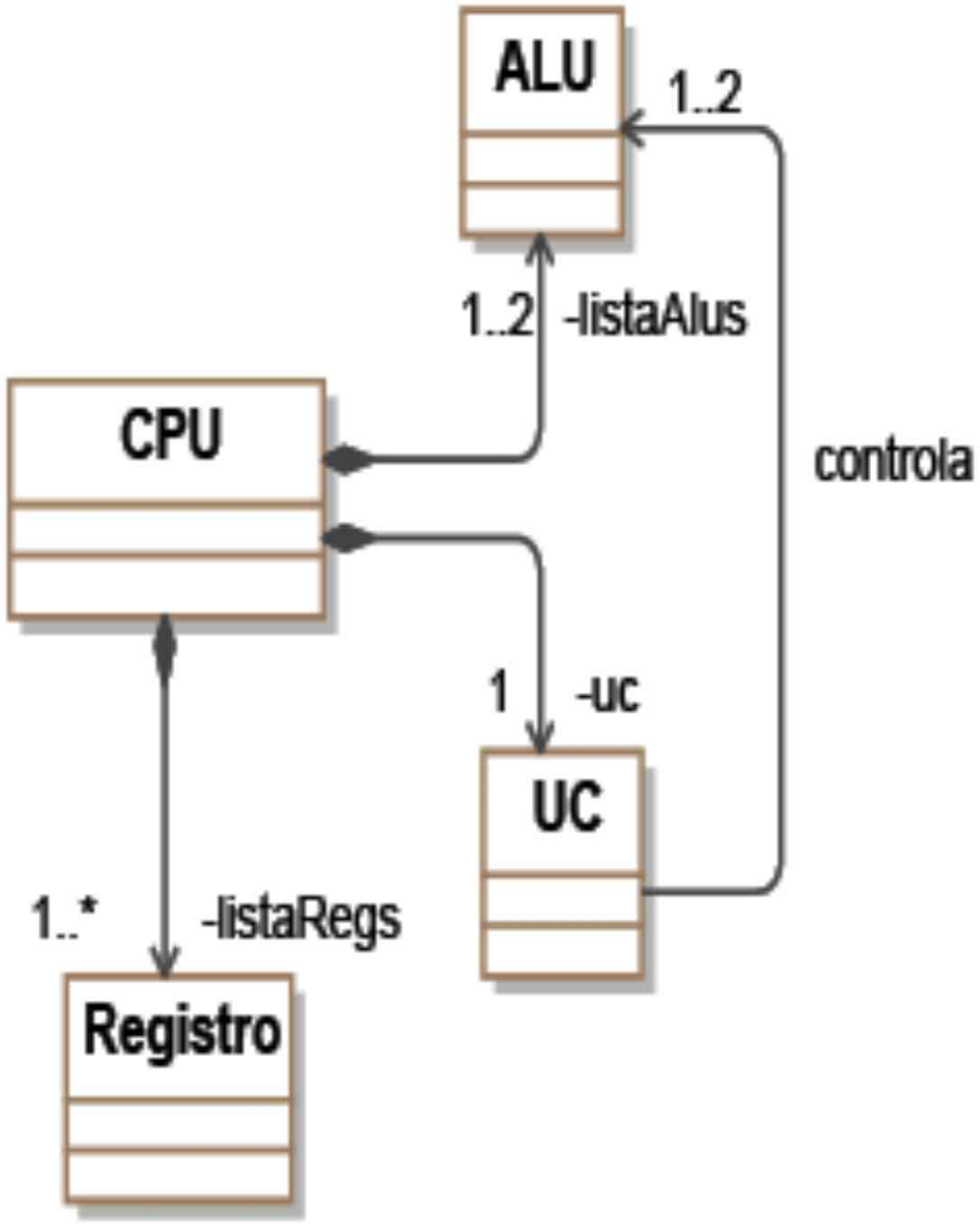


Figura 15.9. Ejemplo de composición

Como la composición es algo restrictiva es preferible decantarse hacia el uso de la agregación en detrimento de la composición.

Al igual que en la agregación, la solución de implementación aquí propuesta no es la única pues podría implementarse también con **vectores**.

Listado 15.3 Implementación de la figura 15.9

```
// Clase ALU
class ALU
{
String unidad;
}
// Clase Unidad de Control
class UC
{
private ALU[] alus = new ALU[2];
public void setALUs(ALU alu1, ALU alu2)
{
alus[0] = alu1;
alus[1] = alu2;
}
public void imprimeTipoALU()
{
System.out.println(alus[0].unidad);
System.out.println(alus[1].unidad);
}
}
// Clase Registro
class Registro
{
public String tipo;
}
// Clase CPU
public class CPU
```

```
{
// Crea composiciones
private UC uc;
private Set<ALU> listaAlus;
private Set<Registro> listaRegs;
public CPU()
{
uc = new UC();
listaAlus = new HashSet<ALU>(2);
listaRegs= new HashSet<Registro>();
Registro reg = new Registro();
reg.tipo = "PG 1";
listaRegs.add(reg);
ALU alu1 = new ALU();
alu1.unidad = "Suma";
ALU alu2 = new ALU();
alu2.unidad = "Resta";
listaAlus.add(alu1);
listaAlus.add(alu2);
// Pasa referencias
uc.setALUs(alu1, alu2);
uc.imprimeTipoALU();
}
}
```

Clases abstractas

Las clases abstractas permiten definir un comportamiento y una estructura común para un conjunto de objetos. No se pueden instanciar y deben ser heredadas por otras clases, aunque pueden tener métodos concretos. En UML las identificaremos como clases con un nombre y algunos métodos en cursiva:

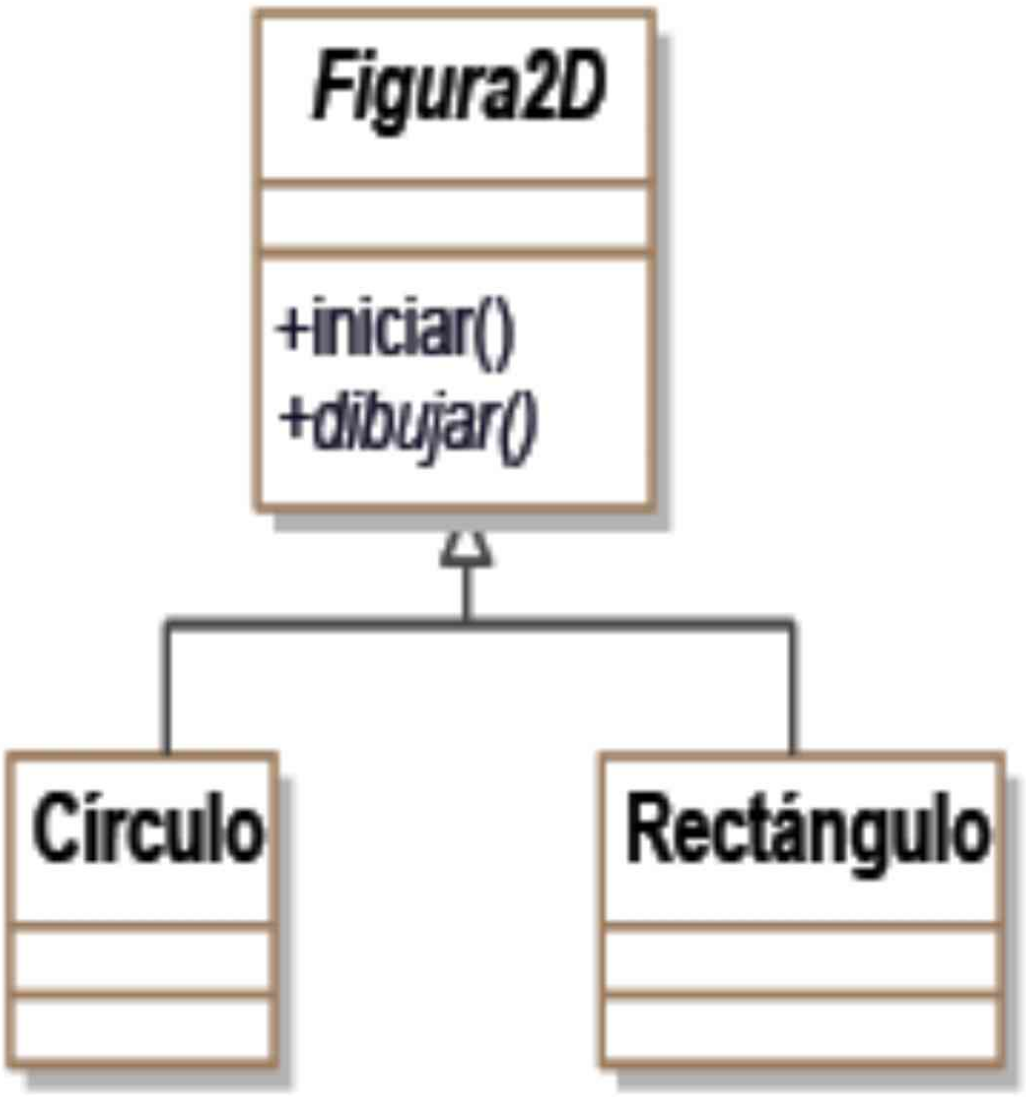


Figura 15.10. Ejemplo de clase abstracta con herencia

En el ejemplo de la figura 15.10 se ha modelado una jerarquía de clases

basada en la herencia de la clase abstracta *Figura2D*. Como se puede observar, la clase *Figura2D* está definida con nombre en cursiva y un método virtual también en cursiva. A continuación mostramos cómo implementar el modelo de la figura 15.10:

Listado 15.4 Implementación de la figura 15.10

```
// Clase abstracta
public abstract class Figura2D
{
public Figura2D()
{
}
public void iniciar()
{
}
// Método abstracto
public abstract void dibujar();
}
// Primera clase heredada
class Circulo extends Figura2D
{
public void dibujar()
{
System.out.println("Dibujando círculo");
}
}
// Segunda clase heredada
class Rectangulo extends Figura2D
```

```
{
public void dibujar()
{
System.out.println("Dibujando rectángulo");
}
}
// Clase principal
class Inicio
{
public static void main (String[] args)
{
Figura2D figura2d = new Circulo();
figura2d.dibujar();
}
}
```

Clases internas

En Java es posible definir clases contenidas dentro de otras clases mediante las llamadas clases internas. Aunque en el capítulo seis no dedicamos una sección específica para ellas, aquí haremos una referencia a su notación:



Figura 15.11. Clase interna

```
public class Buscador
{
private class Evento
{
(...)
}
(...)
}
```

Clases asociación

En el capítulo seis definimos la utilización de las clases asociación para tipos de relación muchos-a-muchos, donde una clase intermedia mantiene información de ambas conexiones.

Supongamos la siguiente relación:



Figura 15.12. Clase asociación

NOTA

Téngase en cuenta que las asignaciones duplicadas no se deben añadir a la lista de asignaciones en una aplicación real.

La correspondiente implementación se codificará de la siguiente manera:

Listado 15.5 Implementación de clase asociación

```
import java.util.Map;
import java.util.HashMap;
import java.util.Iterator;
```

```
// Clase ingeniero
class Ingeniero
{
private int numero;
private HashMap<String, Asignacion> asignaciones;
public Ingeniero(int numero)
{
asignaciones = new HashMap<String,Asignacion>();
this.numero = numero;
}
public boolean setAsignacion(Asignacion asignacion)
{
// Evita repetidos
if (asignaciones.get(asignacion.getId()) ==
null)
{
asignaciones.put
(asignacion.getId(),asignacion);
System.out.println("es true" +
asignacion.getId());
return true;
}
return false;
}
public String toString()
{
return "Ingeniero " + numero;
}
```

```
public String getID()
{
return "ING" + numero;
}
public void imprimeProyectos()
{
System.out.println(this + "
con los proyectos: ");
Iterator i = asignaciones.entrySet().iterator();
while(i.hasNext())
{
Map.Entry entrada = (Map.Entry)i.next();
System.out.println(((Asignacion)
entrada.getValue()).getProyecto());
}
}
}
// Clase asociación
public class Asignacion
{
private Ingeniero ingeniero;
private Proyecto proyecto;
private int presupuesto;
private String id;
public void addRelacion(Ingeniero ingeniero,
Proyecto proyecto, int presupuesto)
{
// Genera clave
```

```
id = ingeniero.getID() +
proyecto.getID();
this.ingeniero = ingeniero;
this.proyecto = proyecto;
this.presupuesto = presupuesto;
if (ingeniero.setAsignacion(this))
proyecto.setAsignacion(this);
}
public String getId()
{
return id;
}
public Proyecto getProyecto()
{
return proyecto;
}
public Ingeniero getIngeniero()
{
return ingeniero;
}
}
// Clase proyecto
class Proyecto
{
private int numero;
private HashMap<String, Asignacion> asignaciones;
public Proyecto(int numero){
asignaciones = new HashMap<String, Asignacion>();
```

```
this.numero = numero;
}
public void setAsignacion(Asignacion asignacion)
{
asignaciones.put
(asignacion.getId(),asignacion);
}
public String toString()
{
return "Proyecto " + numero;
}
public String getID()
{
return "PRY" + numero;
}
public void imprimeIngenieros()
{
System.out.println(this + " con los
ingenieros: ");
Iterator i =
asignaciones.entrySet().iterator();
while(i.hasNext())
{
Map.Entry entrada =
(Map.Entry)i.next();
System.out.println(((Asignacion)
entrada.getValue()).getIngeniero());
}
```

```
}
}
// Clase principal
class Inicio
{
public static void main (String[] args) throws Exception
{
Ingeniero ingenieros[] = new Ingeniero[3];
for (int i = 0; i < 3 ; i++)
ingenieros[i] = new Ingeniero(i);
Proyecto proyectos[] = new Proyecto[3];
for (int i = 0; i < 3 ; i++)
proyectos[i] = new Proyecto(i);
Asignacion asignaciones[] = new Asignacion[3];
for (int i = 0; i < 3 ; i++)
asignaciones[i] = new Asignacion();
asignaciones[0].addRelacion(ingenieros[0],
proyectos[1],1000);
asignaciones[1].addRelacion(ingenieros[1],
proyectos[0],2000);
asignaciones[2].addRelacion(ingenieros[1],
proyectos[1],3000);
ingenieros[1].imprimeProyectos();
proyectos[1].imprimeIngenieros();
}
}
```

Imprimirá:

Ingeniero 1 con los proyectos:
Proyecto 1
Proyecto 0
Proyecto 1 con los ingenieros:
Ingeniero 1
Ingeniero 0

En el listado anterior se ha mostrado el código que implementa la clase asociación; aunque hay que tener en cuenta evitar claves duplicadas dentro de las referencias al objeto que relaciona a los ingenieros con los proyectos.

Asociación calificada

Mediante la asociación calificada podemos identificar un ítem utilizando una clave a modo de parámetro que se le proporciona a la clase de entrada:

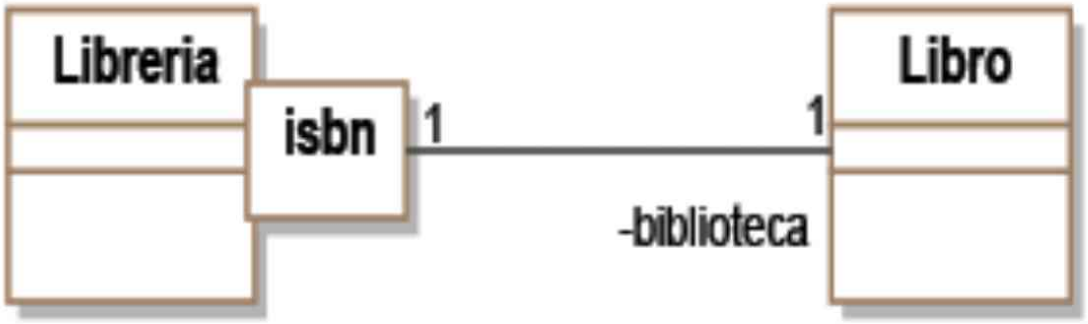


Figura 15.13. Asociación calificada

Listado 15.6 Implementación de calificación

```
public class Libreria
{
private HashMap<Integer,Libro> biblioteca;
public Libreria()
{
biblioteca = new HashMap<Integer,Libro>();
}
public void addLibro(int isbn, String autor)
{
Libro libro = new Libro();
libro.setDatos(autor);
biblioteca.put(isbn,libro);
}
public String getLibro(int isbn)
{
```

```
Libro libro = biblioteca.get(isbn);
return libro.getAutor();
}
}
```

En el código podemos ver la utilización de la clave *isbn* en el método *getLibro()* para la localización de objetos libro dentro de la asociación.

diagramas de secuencias

Los diagramas de secuencias permiten modelar el comportamiento dinámico de la aplicación. Las interacciones entre objetos muestran el orden de las llamadas a operaciones y los valores que devuelven para llevar a cabo una funcionalidad del sistema. Muchos diagramas de secuencias modelan fragmentos algorítmicos en el que varios objetos intercambias mensajes y argumentos.

Interacción básica

Supongamos el siguiente diagrama del capítulo siete:

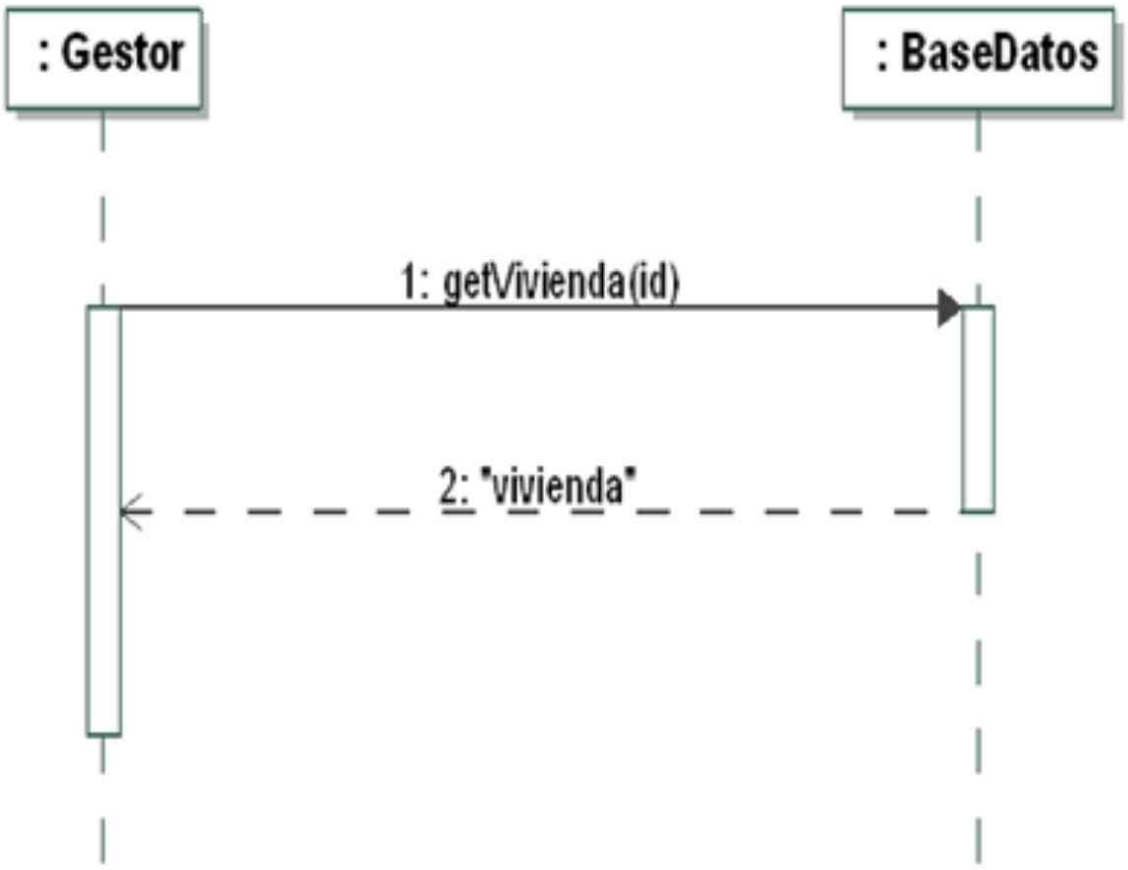


Figura 15.14. Diagrama de secuencias

El correspondiente código equivaldría a:

```
String vivienda = bd.getVivienda(7);
```

esta llamada estaría ubicada dentro de alguno de los métodos de la clase *Gestor* a lo largo del cauce de ejecución de su caso de uso.

Creación, destrucción, automensajes y recursividad

El siguiente ejemplo muestra un caso de mensaje reflexivo y una secuencia de invocaciones recursivas sobre un objeto que emula un algoritmo.

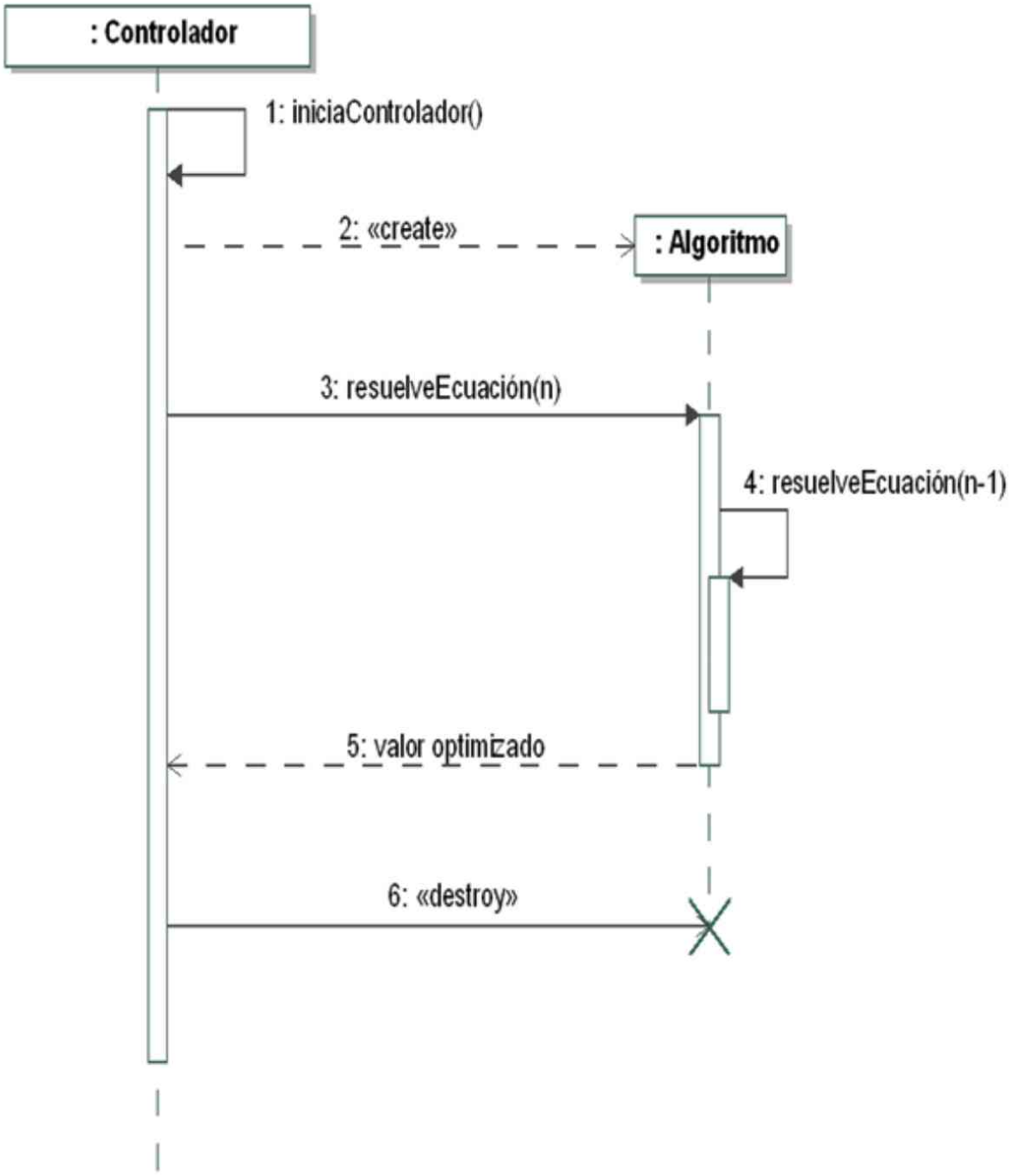


Figura 15.15. Diagrama de secuencias de un controlador que solicita una optimización

Listado 15.7 Implementación del diagrama 15.15

```
// Clase algoritmo
class Algoritmo
{
public int resuelveEcuacion(int n)
{
if (n <= 1)
{
return 1;
}
else
{
return n * resuelveEcuacion(n - 1);
}
}
}
// Clase controlador
public class Controlador
{
public Controlador()
{
// Secuencia de llamadas
iniciaControlador();
}
private void iniciaControlador()
{
Algoritmo algoritmo = new Algoritmo();
System.out.println(
algoritmo.resuelveEcuacion(8));
```

```
}
}
```

Fíjese cómo en el constructor de *Controlador* se ha especificado la secuencia de creación del objeto *Algoritmo*, seguido de la llamada reflexiva y la invocación al método recursivo de *Algoritmo*.

La destrucción del objeto *Algoritmo* es implícita dentro del constructor del controlador.

Saltos condicionales

Los saltos condicionales se pueden implementar en los diagramas de secuencias UML mediante un fragmento *alt*.

En el siguiente ejemplo mostramos la ejecución de una secuencia basada en los pulsos de un objeto *Timer*. Dependiendo del número de *ticks* se realizará la llamada a un dispositivo u a otro. Finalmente, cuando se superan los cinco segundos se procede a detener el *timer*.

En la figura 15.16 se muestra el diagrama de secuencias que modela este contexto en el que se ha requerido el uso de una región "alt".

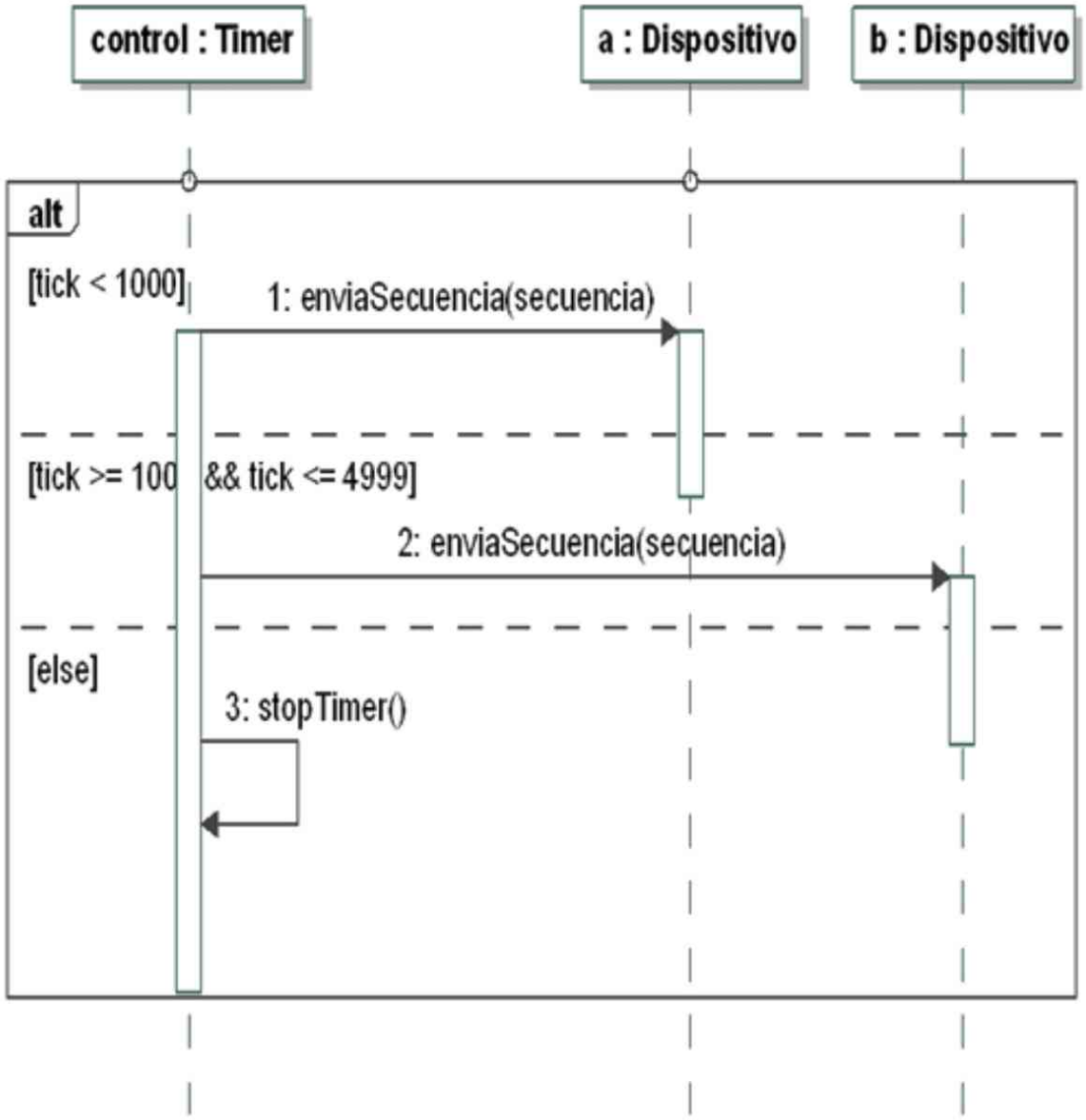


Figura 15.16. Diagrama de secuencias con fragmento condicional

Listado 15.8 Implementación del salto condicional

```
// Clase dispositivo
class Dispositivo
{
public void enviaSecuencia(int secuencia)
{
}
}
// Clase timer
public class Timer
{
private int tick;
private int secuencia;
private Dispositivo a;
private Dispositivo b;
public Timer()
{
tick = 0;
a = new Dispositivo();
b = new Dispositivo();
startTimer();
}
// Evento del timer. Implementación fragmento ALT
private void onTimer()
{
tick++;
```

```
if (tick < 1000)
a.enviaSecuencia(secuencia);
else if (tick >= 1000 && tick <=4999)
b.enviaSecuencia(secuencia);
else stopTimer();
}
}
```

Iteraciones

Examinemos ahora la implementación de un bucle en un diagrama de secuencias. Como vimos en el capítulo siete, la manera de aplicar las iteraciones en UML era mediante un fragmento *loop*. Dichos fragmentos loop pueden adoptar diversas formas de sentencias iterativas dependiendo de si se trata de un bucle *while, for, forEach* o *repeat*.

En el siguiente ejemplo se modela un escenario de consulta de una posible biblioteca. Cuando el usuario solicita al sistema una búsqueda, el objeto *MotorBusqueda* realizará una iteración sobre las tuplas devueltas anteriormente por la base de datos. En caso de encontrar una coincidencia con el libro buscado por el lector se devolverá un objeto *Libro*. Para realizar esta última acción se requiere el uso de la cláusula *break* de UML con la finalidad de romper el flujo iterativo (aunque en el listado 15.9 se termina implícitamente mediante *return*).

Finalmente, en caso de no encontrar ninguna coincidencia, el motor de búsqueda del sistema bibliotecario devolverá *null*.

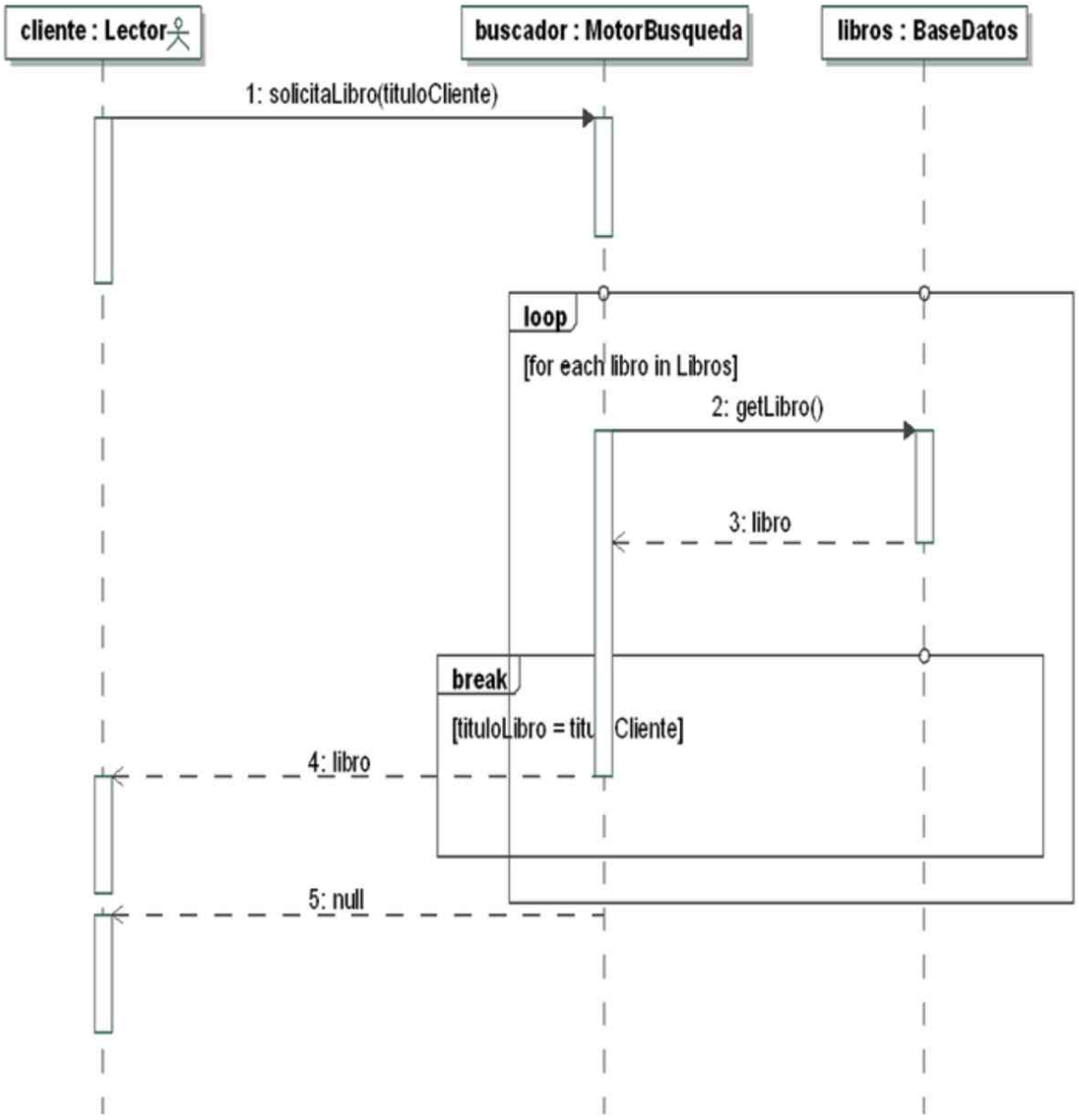


Figura 15.17. Diagrama de secuencias con fragmento iterativo

Listado 15.9 Implementación de la iteración 15.17

```
// Clase motor de búsqueda
public class MotorBusqueda
{
// Almacena un registro de tuplas (recordset)
private Vector<Libro> registroLibros = new Vector<Libro>();
```

```
// Secuencia del diagrama
public Libro solicitaLibro(String tituloCliente)
{
for (Libro libro : registroLibros) {
System.out.println("Libro:" + libro.getTitulo());
if (libro.getTitulo().equals(tituloCliente))
return libro; //break
}
return null;
}
}
// Clase principal
class Inicio
{
public static void main (String[] args)
{
MotorBusqueda motor = new MotorBusqueda();
if (motor.solicitaLibro("Hiperión") != null)
System.out.println("Libro encontrado");
else System.out.println("No encontrado");
}
}
```

diagramas de estado

Los diagramas de estado permiten modelar el comportamiento interno de una aplicación que transita entre diferentes etapas. Los diagramas de estado se utilizan con más frecuencia para controlar la secuencia de fases que se suceden en la vida de un objeto. Para implementar estos diagramas nos serán bastante útiles los patrones de diseño vistos en el capítulo nueve, en concreto el patrón *State*. La asociación de una clase cliente a dicho patrón permite gestionar la máquina de estados que modela el comportamiento del objeto.

En el ejemplo que presentamos en la figura 15.18 se ilustra la secuencia de estados por los que transita un manejador de dispositivo (*driver*) para la lectura de un dato en el puerto de E/S de una controladora. La clase que gestiona la lectura del puerto contiene seis estados exactamente y se inicia con la comprobación del registro de estado. Cuando este registro informa al manejador de la existencia de un dato, pasa a almacenarlo en el buffer para lectura por la aplicación de usuario. Solo en el caso de error en la lectura del dato y desbordamiento del buffer se procede a detener el autómata.

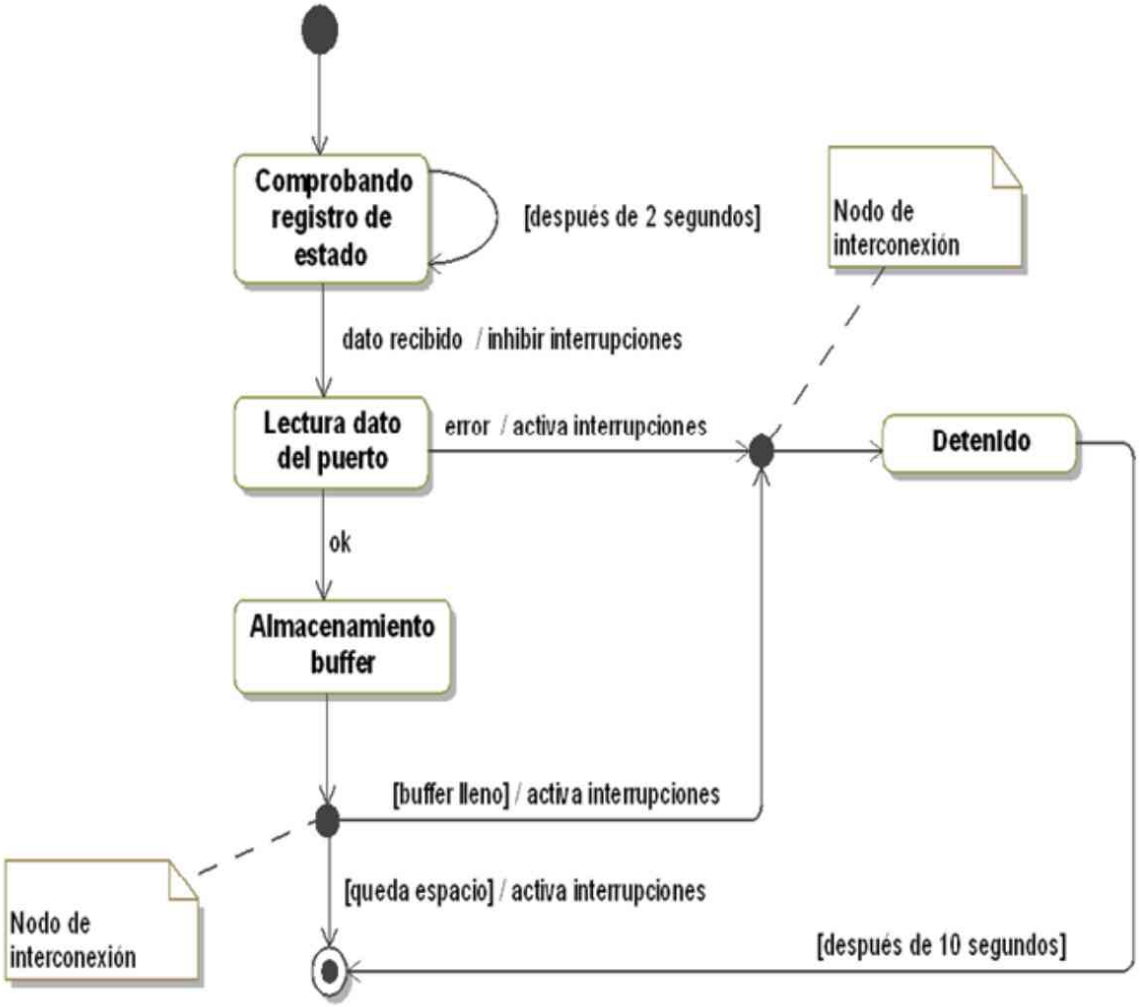


Figura 15.18. Diagrama de estado del procesamiento de un dato por un manejador de dispositivo

Listado 15.10 Implementación del diagrama de estado de la figura 15.18

```
// Clase de gestión del driver
public class Driver
{
// Declaración de estados
private Estado estadoActual;
private EstInicio estInicio;
private EstComprobando estComprobando;
```

```
private EstLeyendo estLeyendo;
private EstAlmacenando estAlmacenando;
private EstDetenido estDetenido;
private EstFinal estFinal;
// Clase abstracta Estado
abstract class Estado
{
public String nombreEstado;
// Posibles eventos
public void iniciar()
{
checkError("EstInicio");
}
public void pasanDosSegundos()
{
checkError("EstComprobando");
}
public void datoRecibido()
{
checkError("EstComprobando");
}
public void errorLectura()
{
checkError("EstLeyendo");
}
public void lecturaOK()
{
checkError("EstLeyendo");
```

```
}
public void bufferLleno()
{
checkError("EstAlmacenando");
}
public void quedaEspacioBuffer()
{
checkError("EstAlmacenando");
}
public void pasanDiezSegundos()
{
checkError("EstDetenido");
}
public void imprimeEstado()
{
System.out.println(nombreEstado);
}
// Comprueba si en estado correcto
public void checkError(String supuesto)
{
if (!nombreEstado.equals(supuesto))System.out.println("Error estado incor-
recto");
}
}
// Implementa los estados concretos
class EstInicio extends Estado
{
public EstInicio()
```

```
{
nombreEstado = “EstInicio”;
}
public void iniciar()
{
System.out.println(“iniciar()”);
estadoActual = estComprobando;
}
}
// Comprobando registro de estado
class EstComprobando extends Estado
{
public EstComprobando()
{
nombreEstado = “EstComprobando”;
}
public void pasanDosSegundos()
{
System.out.println(“pasanDosSegundos()”);
estadoActual = estComprobando;
}
public void datoRecibido()
{
System.out.println(“datoRecibido()”);
estadoActual = estLeyendo;
}
}
// Lectura dato del puerto
```

```
class EstLeyendo extends Estado
{
public EstLeyendo()
{
nombreEstado = “EstLeyendo”;
}
public void errorLectura()
{
System.out.println(“ErrorLectura()”);
estadoActual = estDetenido;
}
public void lecturaOK()
{
System.out.println(“lecturaOK()”);
estadoActual = estAlmacenando;
}
}
// Almacenamiento buffer
class EstAlmacenando extends Estado
{
public EstAlmacenando()
{
nombreEstado = “EstAlmacenando”;
}
public void bufferLleno()
{
System.out.println(“BufferLleno()”);
estadoActual = estDetenido;
```

```
}
public void quedaEspacioBuffer()
{
System.out.println("quedaEspacioBuffer()");
estadoActual = estFinal;
}
}
// Detenido
class EstDetenido extends Estado
{
public EstDetenido()
{
nombreEstado = "EstDetenido";
}
public void pasanDiezSegundos()
{
System.out.println("pasanDiezSegundos()");
estadoActual = estFinal;
}
}
class EstFinal extends Estado
{
public EstFinal()
{
nombreEstado = "EstFinal";
}
}
public Driver()
```

```
{
estInicio = new EstInicio();
estComprobando = new EstComprobando();
estLeyendo = new EstLeyendo();
estAlmacenando = new EstAlmacenando();
estDetenido = new EstDetenido();
estFinal = new EstFinal();
// Estado inicial
estadoActual = estInicio;
}
// Eventos que ocurren en el driver
public void iniciar()
{
estadoActual.iniciar();
}
public void pasanDosSegundos()
{
estadoActual.pasanDosSegundos();
}
public void datoRecibido()
{
estadoActual.datoRecibido();
inhibeInterrupciones();
}
public void errorLectura()
{
estadoActual.errorLectura();
activaInterrupciones();
```

```
}
public void lecturaOK()
{
estadoActual.lecturaOK();
}
public void bufferLleno()
{
estadoActual.bufferLleno();
activaInterrupciones();
}
public void quedaEspacioBuffer()
{
estadoActual.quedaEspacioBuffer();
activaInterrupciones();
}
public void pasanDiezSegundos()
{
estadoActual.pasanDiezSegundos();
}
public void imprimeEstado()
{
estadoActual.imprimeEstado();
}
}
```

Buena parte del código se ha implementado siguiendo las directrices del patrón *State* que nos ha facilitado la construcción de la jerarquía de estados. Durante la ejecución, cada estado concreto se crea estableciendo su identificador, mientras que los métodos se encargan de gestionar los eventos de

salida. Finalmente, desde un subsistema externo de gestión del *driver* se llamarán a los métodos que emularán los diferentes eventos.

caso de estudio: mercurial

Como resumen del capítulo de ingeniería directa en Java y con la finalidad de sintetizar todo lo aquí explicado, se implementará el diagrama de secuencias visto en el apartado 7.7 utilizando el modelo estructural y arquitectónico de *Mercurial*. Dejaremos para el próximo capítulo la implementación en C++ del caso de estudio del ajedrez.

Tal como se definió en el diagrama del apartado 5.4.5, los paquetes que se crearán para nuestra aplicación *Mercurial* son: *Comunicaciones, Interprete, GestorCliente* y *GestorArchivos*. No obstante, debido a la falta de espacio nos centraremos únicamente en aquellos paquetes relacionados con el diagrama de secuencias para borrar un fichero local (figura 7.18). Empezaremos a desglosar cada uno de los paquetes explicando la función específica de cada una de sus clases e interfaces:

Listado 15.11 Paquete interprete :: clase FachadaInterprete

```
package mercurial.interprete;
// Inclusión de paquetes
import mercurial.gestorcliente.*;
import mercurial.gestorarchivos.*;
// Clase de fachada
public class FachadaInterprete
{
private IGestorCliente cliente;
public FachadaInterprete()
{
cliente = new ClienteProgramador();
}
public void entradaTextoSistema()
```

```
{
// Inicia sesión
interpretaComando1("init");
// Borra fichero local
interpretaComando2("delete a/b/p2.c");
}
public void setComando(Comando comando)
{
}
// Inicia sesión
private void interpretaComando1(String comando)
{
// Llama al patrón interpréte
Comando com1 = new Comando();
com1.setId(1);
com1.setComando("init");
cliente.setComando(com1);
}
// Borra fichero local
private void interpretaComando2(String comando)
{
// Llama al patrón interpréte
Comando com2 = new Comando();
com2.setId(2);
com2.setComando("delete");
com2.addParametros("raiz");
com2.addParametros("a");
com2.addParametros("b");
```

```
com2.addParametros("p2.c");
cliente.setComando(com2);
}
}
// Clase principal
class Inicio
{
public static void main (String[] args) throws Exception
{
FachadaInterprete interprete = new
FachadaInterprete();
// Comienzo de la interacción
interprete.entradaTextoSistema();
}
}
```

En el listado 15.11 se muestra la clase principal del paquete *Interprete* que es el responsable de implementar el patrón de diseño con mismo nombre. Su objetivo es el análisis sintáctico y semántico de las cadenas de texto con comandos CVS. La clase *FachadaInterprete* es el punto de interconexión con el patrón y por donde se reciben los objetos *Comando* con la orden ya desglosada. Una vez recibido el objeto comienza la llamada al subsistema representado en el paquete *GestorCliente*:

Listado 15.12 Paquete GestorCliente :: clase Comando

```
package mercurial.gestorcliente;
// Inclusión de paquetes
import java.util.Vector;
```

```
// Clase comando desglosado
public class Comando
{
private int id;
private String comando;
private Vector<String> parametros;
// Constructor
public Comando()
{
parametros = new Vector<String>();
}
// Identificador
public void setId(int id)
{
this.id = id;
}
// Tipo de comandos
public void setComando(String comando)
{
this.comando = comando;
}
// Parámetros del comando
public void addParametros(String parametro)
{
parametros.add(parametro);
}
// Métodos de acceso
public int getID()
```

```
{
return id;
}
public String getComando()
{
return comando;
}
public Vector<String> getParametros()
{
return parametros;
}
}
```

Este simple listado muestra la estructura de la clase *Comando* que servirá para intercambiar órdenes entre subsistemas. La clase *Comando* abstrae los elementos ya analizados y desglosados por el patrón *Interpreter*.

Listado 15.13 Paquete GestorCliente :: interface IGestorCliente

```
package mercurial.gestorcliente;
public interface IGestorCliente
{
public void setComando(Comando comando);
}
```

El listado 15.13 muestra otro elemento principal del paquete *GestorCliente*. Se trata de la interfaz *IGestorCliente* que permite el desarrollo del subsistema *GestorCliente* así como la intercomunicación entre paquetes y clases heterogéneas.

En el siguiente listado se expondrá la clase base para los roles principales de la aplicación: programador y administrador. La clase *Administrador* no se ha implementado al no ser necesaria en la descripción del diagrama de secuencias. Sin embargo, nos será de utilidad la clase *Usuario* para implementar el rol del programador:

Listado 15.14 Paquete GestorCliente :: clase Usuario

```
package mercurial.gestorcliente;
// Clase base paquete GestorCliente
public class Usuario implements IGestorCliente
{
private String login;
private String password;
// Comandos básicos
public void setComando(Comando comando)
{
if (comando.getComando().equals("init"))
iniciarSesion();
else if (comando.getComando().equals("close"))
cerrarSesion();
}
public void iniciarSesion()
{
// Conecta con fachada comunicaciones
System.out.println("Sesión iniciada");
}
public void listarFicheros()
{
```

```
}
public void cerrarSesion()
{
System.out.println("Cerrando sesión");
}
public void registarse()
{
}
}
```

Listado 15.15 Paquete GestorCliente :: clase ClienteProgramador

```
package mercurial.gestorcliente;
// Inclusión de paquetes
import java.util.Vector;
import java.util.Iterator;
import mercurial.gestorarchivos.*;
public class ClienteProgramador extends Usuario
{
private IDirectorioAbstracto directorioMain;
public ClienteProgramador()
{
// Crea estructura de directorios
Directorio directorioRaiz = new Directorio();
directorioRaiz.setNombre("raiz");
Directorio directorioA = new Directorio();
directorioA.setNombre("a");
Directorio directorioB = new Directorio();
directorioB.setNombre("b");
```

```
Directorio directorioC = new Directorio();
directorioC.setNombre("c");
Fichero fichero1 = new Fichero();
fichero1.setNombre("p2.c");
Fichero fichero2 = new Fichero();
fichero2.setNombre("p4.c");
directorioMain = directorioRaiz;
// Añade contenido (directorios y archivos) a lista
directorioRaiz.add(directorioA);
directorioA.add(directorioB);
directorioB.add(directorioC);
directorioB.add(fichero1);
directorioC.add(fichero2);
}
// Establecimiento de comandos del programador
public void setComando(Comando comando)
{
// Comandos básicos
super.setComando(comando);
if (comando.getComando().equals("delete"))
{
try{
borrarFichero(comando.getParametros().
lastElement(), comando.getParametros());
} catch (Exception ex)
{
System.out.println("Fichero no encontrado");
}
```

```
}
}
// Añade fichero remoto
public void anadirFichero(String fichero, Vector<String> pathFichero)
{
}
// Método para borrar un fichero local
public void borrarFichero(String fichero,
Vector<String> pathFichero) throws Exception
{
Iterator i = pathFichero.iterator();
i.next(); // Salta el raíz (siempre está)
while (directorioMain != null && i.hasNext()
&& !directorioMain.getNombre().equals(fichero))
{
System.out.println("Accediendo a: " +
directorioMain.getNombre());
directorioMain = directorioMain.getItem((String)i.next());
}
// Se ha encontrado el fichero
if (directorioMain != null &&
directorioMain.getNombre().equals(fichero))
((Fichero)directorioMain).
deleteFile(directorioMain.getNombre());
else throw new Exception();
// Error, no se encuentra fichero
}
}
```

La clase *ClienteProgramador* es la responsable de implementar los casos de uso relacionados con el rol del programador. De este código nos interesa únicamente el método que se encarga de borrar el fichero local mediante el bucle ya visto en el diagrama de secuencias. Para esta implementación se necesita conocer previamente el patrón *Composite* visto en el capítulo nueve, y que aquí viene representado en el paquete *GestorArchivos* con la interface *IDirectorioAbstracto, Directorio* y *Fichero*.

Listado 15.16 Paquete GestorArchivos :: interface IDirectorioAbstracto

```
package mercurial.gestorarchivos;
// Implementación del patrón composite (clase Component)
public interface IDirectorioAbstracto
{
public String getNombre();
public void setNombre(String nombre);
public IDirectorioAbstracto getItem(String siguiente);
}
```

El listado 15.16 muestra la interfaz *IDirectorioAbstracto* que representa a la clase abstracta *Component* vista en la sección 9.6.2 y sirve de base para la estructura del patrón.

Listado 15.17 Paquete GestorArchivos :: clase Directorio

```
package mercurial.gestorarchivos;
// Inclusión de paquetes
import java.util.Vector;
import java.util.Iterator;
// Implementación del patrón composite (clase Composite)
```

```
public class Directorio implements IDirectorioAbstracto
{
private int numFicheros;
private String nombre;
private Vector<IDirectorioAbstracto> items;
// Constructor
public Directorio()
{
items = new Vector<IDirectorioAbstracto>();
}
public String getNombre()
{
return nombre;
}
public void setNombre(String nombre)
{
this.nombre = nombre;
}
// Añade lista de contenido directorio
public void add(IDirectorioAbstracto nombre)
{
items.addElement(nombre);
}
// Obtiene elemento concreto de un directorio
public IDirectorioAbstracto getItem(String siguiente)
{
for (IDirectorioAbstracto directorioEncontrado : items)
{
```

```
String nombreEncontrado = directorioEncontrado.getNombre();
if (nombreEncontrado.equals(siguiente))
return directorioEncontrado;
}
return null;
}
}
```

La clase *Directorio* equivale a la clase concreta *Composite* del esquema del patrón. Permite implementar el contenido de un directorio (directorios hijos y ficheros) en el mismo orden jerárquico de un sistema de archivos. El método *getItem()* permite extraer un elemento concreto dentro del conjunto de hijos de un directorio padre y devolverlo a la iteración de borrado (clase *ClienteProgramador*). El elemento extraído debe coincidir con el ítem actual de la ruta absoluta del archivo.

Listado 15.18 Paquete GestorArchivos :: clase Fichero

```
package mercurial.gestorarchivos;
// Implementación del patrón composite (clase Leaf)
public class Fichero implements IDirectorioAbstracto
{
private String nombre;
public String getNombre()
{
return nombre;
}
public void setNombre(String nombre)
{
```

```
this.nombre = nombre;
}
public IDirectorioAbstracto getItem(String siguiente)
{
return null;
}
public void deleteFile(String fichero)
{
System.out.println("Borrando fichero: " + fichero);
}
}
```

Finalmente, la clase *Fichero* implementa los nodos terminales del patrón *Composite*, es decir, los objetos que en teoría representan las clases *Leaf*.

---

36 Nótese que cada uno tiene diferente visibilidad.

ingeniería directa en c++

«[...] La mente es su propio lugar y puede hacer en ella un Cielo del Infierno y del Infierno un Cielo.[...]».

(John Milton: El paraíso perdido, Libro I, v.253).

Después de implementar los conceptos fundamentales de UML en Java comenzaremos a abordarlos también aquí para el lenguaje de programación C++. Si entendió las explicaciones y los listados de código fuente que se presentaron en el capítulo quince, la adaptación a C++ le resultará mucho más sencilla. Conviene entender los conceptos clave de UML en un lenguaje más puramente orientado a objetos para así familiarizarse directamente con la asociación *UML – código* sin las complejidades añadidas de la sintaxis.

En este capítulo haremos un recorrido por los diagramas que están más estrechamente relacionados con la ingeniería de un proyecto. Por este motivo, se presentarán al comienzo los ejemplos vistos en el capítulo anterior y su relación con el lenguaje C++. Al final del capítulo se implementará el diagrama de secuencias asociado a una jugada de ajedrez como consolidación de conocimientos.

Aunque C++ no es un lenguaje puramente orientado a objetos, es muy factible para el desarrollo de aplicaciones nativas de alto rendimiento: buscadores a gran escala, lenguajes de alto nivel, motores de bases de datos o aplicaciones gráficas, incluso muchas tecnologías de base se programan en este lenguaje. Esto es debido a que C++ no es un lenguaje interpretado por ninguna máquina virtual y el código que genera es directamente código de la máquina. Además, C++ tiene la gran ventaja del equilibrio entre el nivel hardware y el alto nivel de abstracción, lo que lo hace una herramienta muy versátil y poderosa.

diagramas de clases

Como se vio en el capítulo seis, los diagramas de clases definen un modelo estructural basado en clases, jerarquías y asociaciones que permiten definir una visión estática del sistema. A continuación estudiaremos cada uno de los clasificadores y componentes principales de este diagrama desde una perspectiva de la sintaxis de C++.

Representación de una clase

La clase es la piedra angular de la programación orientada a objetos. Su existencia en cualquier lenguaje facilita en gran medida la construcción de software complejo.

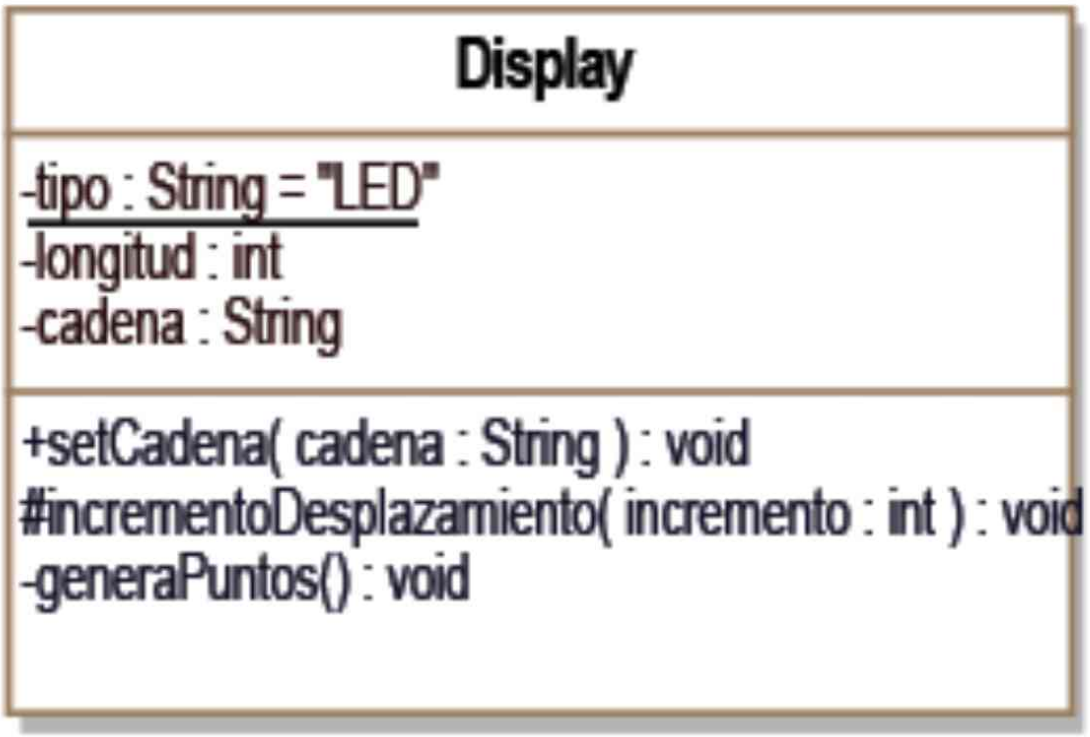


Figura 16.1. Ejemplo de clase en UML

Listado 16.1 Implementación de la clase Display :: fichero display.h

```
#ifndef display_h
#define display_h
#include <string>
#include <iostream>
using namespace std;
class Display
{
private:
```

```
static string tipo;
int longitud;
string cadena;
public:
Display();
void set_cadena(const string& cadena);
protected:
void incrementa_desplazamiento(int incremento);
private:
void genera_puntos();
};
#endif
```

Listado 16.2 Implementación de la clase Display :: fichero display.cpp

```
#include "display.h"
using namespace std;
string Display::tipo = "LED";
Display::Display()
{
}
void Display::set_cadena(const string& cadena)
{
this->cadena = cadena;
cout << cadena << endl;
}
void Display::incrementa_desplazamiento(int incremento)
{
}
```

```
void Display::genera_puntos()
{
}
```

El listado 16.1 muestra el código fuente del fichero de cabecera asociado a la clase *Display*. En las secciones *public*, *protected* y *private* se aplican las propiedades de acceso a cada uno de los atributos y operaciones. La implementación del código de la clase se realiza en el fichero *display.cpp* que aparece en el listado 16.2. A diferencia del código Java explicado en el capítulo catorce, la implementación de una clase extensa o subsistema en C++ requiere del uso de dos ficheros (.h y .cpp) para la definición y la implementación respectivamente.

Asociaciones

Para especificar la visibilidad o conocimiento a nivel de atributo de otros objetos en una clase en C++ se requiere el uso de punteros o referencias. Así, para el diagrama de clases siguiente:

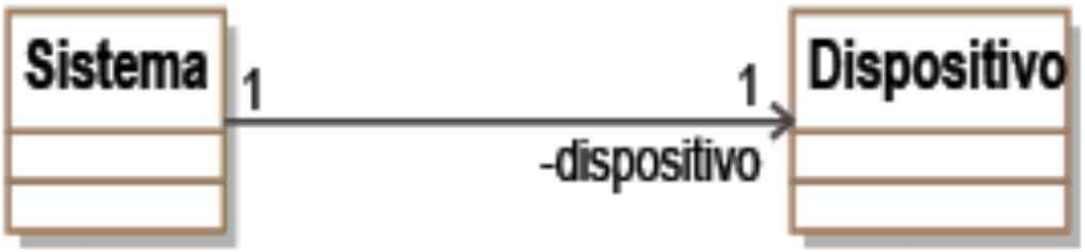


Figura 16.2. Asociación simple

El código correspondería a:

Listado 16.3 Definición de asociación simple (opción referencia)

```
class Sistema
{
private:
Dispositivo& dispositivo_ref;
public:
Sistema(Dispositivo& dispositivo);
void haz_algo_ref();
};
```

Listado 16.4 Definición de asociación simple (opción puntero)

```
class Sistema
{
private:
```

```
Dispositivo* dispositivo_ptr;
public:
Sistema(Dispositivo* dispositivo);
void haz_algo_ptr();
};
```



Figura 16.3. Asociación con cardinalidad 1..*

La asociación múltiple requiere de la estructura de datos *vector* de la STL:

Listado 16.5 Definición de asociación múltiple

```
class Sistema
{
private:
vector<Dispositivo*> dispositivos;
public:
Sistema();
~Sistema();
void inserta_dispositivo(Dispositivo* dispositivo);
void imprime_dispositivos();
};
```



Figura 16.4. Asociación con cardinalidad restringida

Sin embargo, para una multiplicidad limitada se puede recurrir a un *array*:

Listado 16.6 Definición de asociación con multiplicidad limitada

```
class Sistema
{
private:
Dispositivo* dispositivos[8];
public:
Sistema();
void inserta_dispositivo(Dispositivo* dispositivo, int indice);
void imprime_dispositivos();
};
```

Herencia simple

Por medio de la herencia podemos establecer relaciones jerárquicas entre clases. En el árbol producido por la herencia, las clases base proporcionan o restringen atributos y métodos a sus clases derivadas. Gracias a la herencia es posible la compartición de información y el comportamiento desde las clases base a las clases derivadas. Es así, por tanto, como se puede extender la funcionalidad de las diferentes entidades participantes en una aplicación. En general, las ventajas de la herencia son muchas [Barnes17]:

- Se evita la duplicación de código.
- Facilita la reutilización del código.
- Facilita el mantenimiento.
- Aumenta la capacidad de ampliación del sistema.

En el siguiente ejemplo se muestra la equivalencia entre la herencia en el diagrama de clases y el código C++:

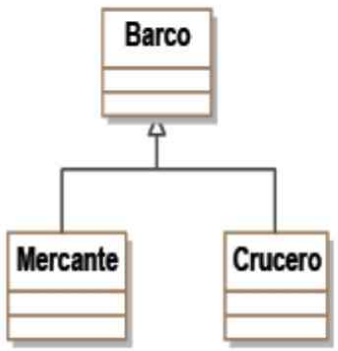


Figura 16.5. Simple

```
class Barco
{
};
class Mercante : public Barco
{
};
class Crucero : public Barco
{
};
```

Herencia múltiple

En C++ es posible indicar que una clase herede de varias clases diferentes al mismo tiempo. A esto se le denomina herencia múltiple, y como dijimos en el capítulo catorce, es una característica de este lenguaje que no posee Java. A este respecto existe mucha controversia sobre la ambigüedad y complejidad para el proyecto que supone esta técnica implementada en lenguajes como C++, Python o Perl. Sin embargo, la utilización de herencia simple y patrones de diseño en vez de la herencia múltiple de clases es una alternativa cada vez más eficaz para el modelado de aplicaciones consistentes.

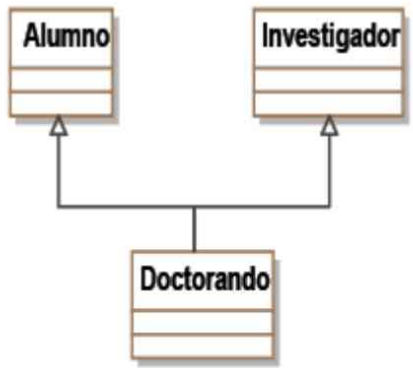


Figura 16.6. Múltiple

```
class Alumno
{
};
class Investigador
{
};
class Doctorando : public Alumno, public Investigador
{
};
```

Agregación

La agregación en C++ se implementa mediante la declaración de un vector que mantiene las referencias a los objetos contenidos. La clase contenedora no gestiona el ciclo de vida de los objetos contenidos.



Figura 16.7. Agregación simple

```
class Persona
{
private:
Empleo* empleo;
};
```

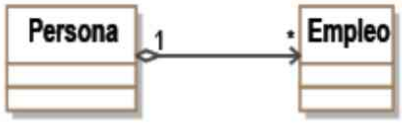


Figura 16.8. Agregación múltiple

```
class Persona
{
private:
vector<Empleo*> empleos;
};
```

Como puede observarse en el código de la figura 16.8, la implementación de la agregación múltiple se realiza a través de un vector STL (Standard Template Library).

Composición

En la composición, los objetos mantenidos por el contenedor deben ser eliminados cuando éste es destruido, es decir, el objeto contenedor es responsable de la vida de sus objetos compuestos. La composición en C++ puede implementarse tanto con *arrays* como con vectores; aunque es importante no olvidar eliminar los objetos que contienen al borrar el objeto padre.

Supongamos de nuevo el ejemplo explicado en el capítulo catorce:

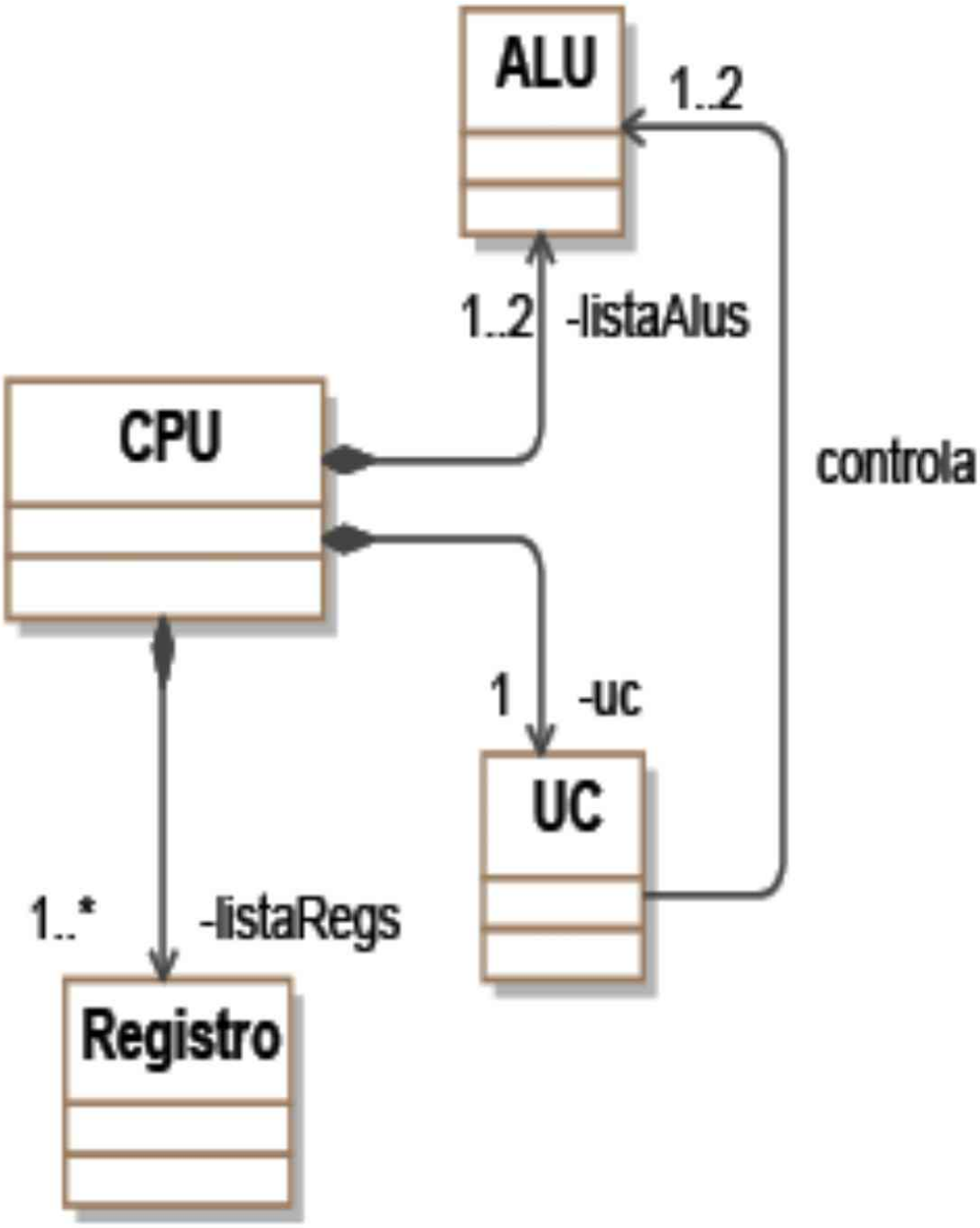


Figura 16.9. Composición

Listado 16.7 Implementación de la composición :: fichero cpu.h

```
#ifndef cpu_h
#define cpu_h
```

```
#include <vector>
#include <string>
#include <iostream>
using namespace std;
// Clase ALU
class ALU
{
public:
string unidad;
public:
~ALU();
};
// Clase Unidad de Control
class UC
{
private:
ALU* alus[2];
public:
~UC();
void set_ALUS(ALU* alu1, ALU* alu2);
void imprime_tipo_ALU();
};
// Clase Registro
class Registro
{
public:
string tipo;
public:
```

```
~Registro();
};
// Clase CPU
class CPU
{
// Crea composiciones
private:
UC* uc;
ALU* alus[2];
vector<Registro*> lista_regs;
public:
CPU();
~CPU();
private:
void destruye_componentes();
};
#endif
```

Listado 16.8 Implementación de la composición :: fichero cpu.cpp

```
#include "cpu.h"
using namespace std;
// Destructor ALU
ALU::~ALU()
{
cout << "Destruyendo ALU: " << unidad << endl;
}
void UC::set_ALUS(ALU* alu1, ALU* alu2)
{
```

```
alus[0] = alu1;
alus[1] = alu2;
}
void UC::imprime_tipo_ALU()
{
cout << alus[0]->unidad << endl;
cout << alus[1]->unidad << endl;
}
// Destructor Unidad de Control
UC::~UC()
{
cout << "Destruyendo UC" << endl;
}
// Destructor de registros
Registro::~Registro()
{
cout << "Destruyendo registro: " << tipo << endl;
}
CPU::CPU()
{
uc = new UC();
Registro* reg = new Registro();
reg->tipo = "PG 1";
lista_regs.push_back(reg);
reg = new Registro();
reg->tipo = "PG 2";
lista_regs.push_back(reg);
alus[0] = new ALU();
```

```
alus[0]->unidad = "Suma";
alus[1] = new ALU();
alus[1]->unidad = "Resta";
// Pasa referencias
uc->set_ALUS(alus[0], alus[1]);
uc->imprime_tipo_ALU();
}
// Elimina composiciones
void CPU::destruye_componentes()
{
delete uc;
delete alus[0];
delete alus[1];
vector<Registro*>::iterator it;
for (it=lista_regs.begin();it!=lista_regs.end();it++)
delete *it;
}
// Destructor CPU
CPU::~CPU()
{
destruye_componentes();
}
int main()
{
CPU* cpu = new CPU();
delete cpu;
return 0;
}
```

Clases abstractas e interfaces

Como en Java, las clases abstractas nos permitirán definir un comportamiento heterogéneo para cada clase derivada. En C++, una clase que contiene al menos una función virtual pura se considera una clase abstracta. Cuando se requiera implementar una *interface* de UML en C++ se definirán todos los métodos como funciones virtuales puras, es decir, del siguiente modo:

```
class Interface
{
public:
virtual void metodo1(parámetros) = 0;
virtual void metodo2(parámetros) = 0;
. = 0;
. = 0;
. = 0;
};
```

Supongamos de nuevo el ejemplo visto en la sección 15.1.6 sobre la jerarquía de un modelo de clases para la construcción de figuras:

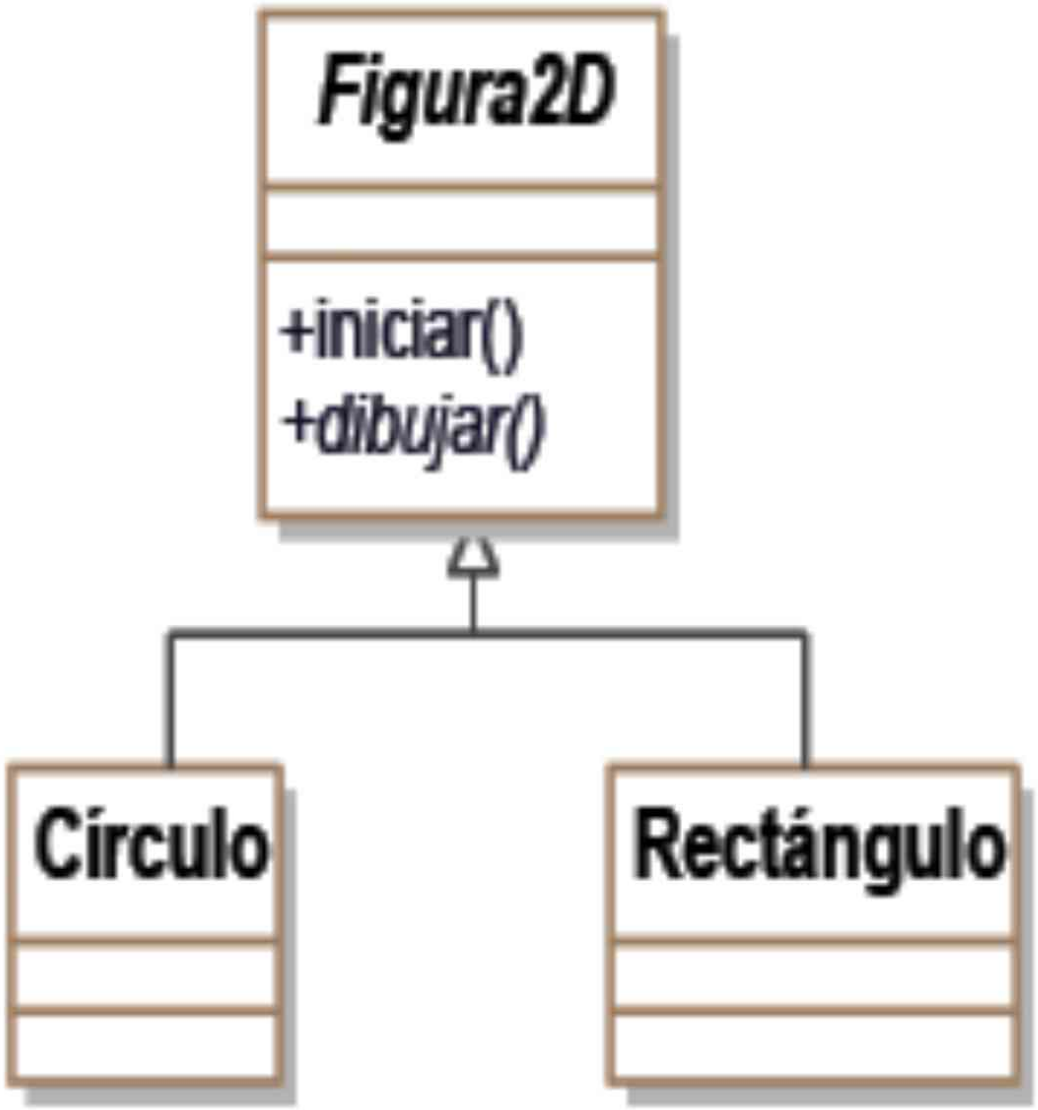


Figura 16.10. Ejemplo de clase abstracta con herencia

La implementación del diagrama de clases de la figura 16.10 supondrá la definición de la clase *Figura2D* con un método virtual puro, lo que la convertirá automáticamente en una clase abstracta. Las clases *Circulo* y *Rectangulo* deberán derivar de la clase base para implementar el comportamiento correcto.

Listado 16.9 Implementación de la figura 16.10 :: fichero figura2d.h

```
#ifndef figura2d_h
#define figura2d_h
#include <string>
#include <iostream>
// Clase abstracta
class Figura2D
{
public:
virtual void iniciar();
// Método abstracto
virtual void dibujar() = 0;
};
// Primera clase heredada
class Circulo : public Figura2D
{
public:
virtual void dibujar();
};
// Segunda clase heredada
class Rectangulo : public Figura2D
{
public:
virtual void dibujar();
};
#endif
```

Listado 16.10 Implementación de la figura 16.10 :: fichero figura2d.cpp

```
#include "figura2d.h"
```

```
using namespace std;
void Figura2D::iniciar()
{
(...)
}
void Circulo::dibujar()
{
cout << “Dibujando círculo” << endl;
}
void Rectangulo::dibujar()
{
cout << “Dibujando rectángulo” << endl;
}
int main()
{
Figura2D* figura2d = new Rectangulo();
figura2d->dibujar();
cin.get();
delete figura2d;
return 0;
}
```

Clases internas o anidadas



Figura 16.11. Clase interna

En C++ también es posible de declaración de clases internas, tal como vemos en el ejemplo del listado 16.11.

Listado 16.11 Implementación de la clase interna en C++ :: fichero internas.cpp

```
#include <iostream>
using namespace std;
class Buscador
{
public:
class Evento
{
public:
void imprime_evento()
{
std::cout << "Imprimiendo evento..." << std::endl;
}
};
```

```
};
int main() {
Buscador::Evento obj;
obj.imprime_evento();
}
```

Clases asociación

En C++ también es posible la implementación de clases de asociación, en la que existe una clase intermedia que mantiene la correspondencia en una asociación muchas-a-mu-chos.



Figura 16.12. Clase asociación

NOTA

Téngase en cuenta que las asignaciones duplicadas no se deben añadir a la lista de asignaciones en una aplicación real.

Listado 16.12 Implementación de la clase asociación :: fichero asignacion.h

```
#ifndef asignacion_h
#define asignacion_h
#include <string>
```

```
#include <map>
#include <iostream>
#include <sstream>
using namespace std;
// Prototipos
class Asignacion;
class Proyecto;
// Clase ingeniero
class Ingeniero
{
private:
int numero;
map<string, Asignacion*> asignaciones;
public:
Ingeniero(int numero);
bool set_asignacion(Asignacion* asignacion);
string to_string();
string get_ID();
void imprime_proyectos();
};
// Clase asociación
class Asignacion
{
private:
Ingeniero* ingeniero;
Proyecto* proyecto;
int presupuesto;
string id;
```

```
public:
void add_relacion(Ingeniero* ingeniero, Proyecto*
proyecto, int presupuesto);
string get_ID();
Proyecto* get_proyecto();
Ingeniero* get_ingeniero();
};
// Clase proyecto
class Proyecto
{
private:
int numero;
map<string, Asignacion*> asignaciones;
public:
Proyecto(int numero);
void set_asignacion(Asignacion* asignacion);
string to_string();
string get_ID();
void imprime_ingenieros();
};
#endif
```

Listado 16.13 Implementación de la clase asociación :: fichero asignacion.cpp

```
#include "asignacion.h"
using namespace std;
Ingeniero::Ingeniero(int numero)
{
```

```
this->numero = numero;
}
bool Ingeniero::set_asignacion(Asignacion* asignacion)
{
// Evita repetidos
if (asignaciones[asignacion->get_ID()] == NULL)
{
asignaciones[asignacion->get_ID()] = asignacion;
return true;
}
return false;
}
string Ingeniero::to_string()
{
stringstream ss;
ss << numero;
return "Ingeniero " + ss.str();
}
string Ingeniero::get_ID()
{
stringstream ss;
ss << numero;
return "ING" + ss.str();
}
void Ingeniero::imprime_proyectos()
{
cout << to_string() << " con los proyectos " << endl;
map<string, Asignacion*>::iterator it =
```

```
asignaciones.begin();
for (it = asignaciones.begin(); it !=
asignaciones.end(); it++)
{
cout << (*it).second->get_proyecto()->to_string() <<
endl;
}
}
void Asignacion::add_relacion(Ingeniero* ingeniero, Proyecto* proyecto,
int presupuesto)
{
// Genera clave
id = ingeniero->get_ID() + proyecto->get_ID();
this->ingeniero = ingeniero;
this->proyecto = proyecto;
this->presupuesto = presupuesto;
if (ingeniero->set_asignacion(this))
{
proyecto->set_asignacion(this);
}
}
string Asignacion::get_ID()
{
return id;
}
Proyecto* Asignacion::get_proyecto()
{
return proyecto;
```

```
}
Ingeniero* Asignacion::get_ingeniero()
{
return ingeniero;
}
Proyecto::Proyecto(int numero)
{
this->numero = numero;
}
void Proyecto::set_asignacion(Asignacion* asignacion)
{
asignaciones[asignacion->get_ID()] = asignacion;
}
string Proyecto::to_string()
{
stringstream ss;
ss << numero;
return “Proyecto “ + ss.str();
}
string Proyecto::get_ID()
{
stringstream ss;
ss << numero;
return “PRY” + ss.str();
}
void Proyecto::imprime_ingenieros()
{
cout << to_string() << “ con los ingenieros “ << endl;
```

```
map<string, Asignacion*>::iterator it =
asignaciones.begin();
for (it = asignaciones.begin(); it !=
asignaciones.end(); it++)
{
cout << (*it).second->get_ingeniero()->to_string() <<
endl;
}
}
int main()
{
Ingeniero* ingenieros[3];
for (int i = 0; i < 3 ; i++)
{
ingenieros[i] = new Ingeniero(i);
}
Proyecto* proyectos[3];
for (int i = 0; i < 3 ; i++)
{
proyectos[i] = new Proyecto(i);
}
Asignacion* asignaciones[3];
for (int i = 0; i < 3 ; i++)
{
asignaciones[i] = new Asignacion();
}
asignaciones[0]->add_relacion(ingenieros[0],
proyectos[0],1000);
```

```
asignaciones[1]->add_relacion(ingenieros[0],
proyectos[1],2000);
asignaciones[2]->add_relacion(ingenieros[0],
proyectos[2],3000);
ingenieros[0]->imprime_proyectos();
proyectos[1]->imprime_ingenieros();
for (int i = 0; i < 3 ; i++)
{
delete ingenieros[i];
}
for (int i = 0; i < 3 ; i++)
{
delete proyectos[i];
}
for (int i = 0; i < 3 ; i++)
{
delete asignaciones[i];
}
cin.get();
return 0;
}
```

Imprimirá:

```
Ingeniero 0 con los proyectos
Proyecto 0
Proyecto 1
Proyecto 2
Proyecto 1 con los ingenieros
```

Ingeniero o

Asociación calificada

C++ nos permite también implementar la asociación calificada. Ésta nos facilita la búsqueda de un ítem utilizando un parámetro proporcionado a la clase.

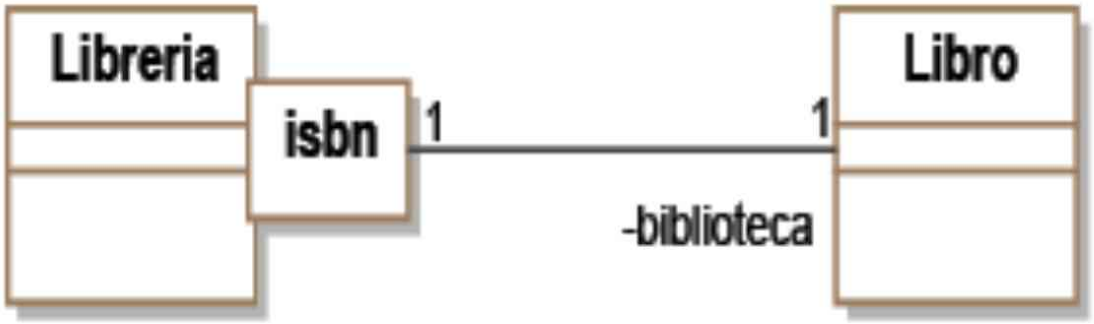


Figura 16.13. Asociación calificada

Listado 16.14 Asociación calificada (implementación):: fichero libreria.h

```
#ifndef libreria_h
#define libreria_h
#include <string>
#include <map>
#include <iostream>
#include <stdexcept>
using namespace std;
class Libro
{
private:
string autor;
public:
void set_datos(const string& autor);
string get_autor();
```

```
};
class Libreria
{
private:
map<int, Libro*> biblioteca;
public:
~Libreria();
void add_libro(int isbn, const string& autor);
string get_libro(int isbn);
private:
void borra_libros();
};
#endif
```

Listado 16.15 Asociación calificada (implementación):: fichero libreria.cpp

```
#include "libreria.h"
using namespace std;
void Libro::set_datos(const string& autor)
{
this->autor = autor;
}
string Libro::get_autor()
{
return autor;
}
void Libreria::add_libro(int isbn, const string& autor)
{
Libro* libro = new Libro();
```

```
libro->set_datos(autor);
biblioteca[isbn] = libro;
}
string Libreria::get_libro(int isbn)
{
Libro* libro = biblioteca[isbn];
return libro->get_autor();
}
void Libreria::borra_libros()
{
map<int, Libro*>::iterator it;
for (it = biblioteca.begin(); it != biblioteca.end();
it++)
delete (*it).second;
}
Libreria::~Libreria()
{
borra_libros();
}
int main()
{
Libreria lib;
lib.add_libro(12345678, “Miguel de Cervantes”);
lib.add_libro(4567890, “Friedrich Hölderlin”);
cout << lib.get_libro(4567890) << endl;
cin.get();
return 0;
}
```

diagramas de secuencias

Los diagramas de secuencias nos permiten modelar el comportamiento e interacciones entre objetos de un subsistema para llevar a cabo una funcionalidad especificada en los casos de uso.

Interacción básica

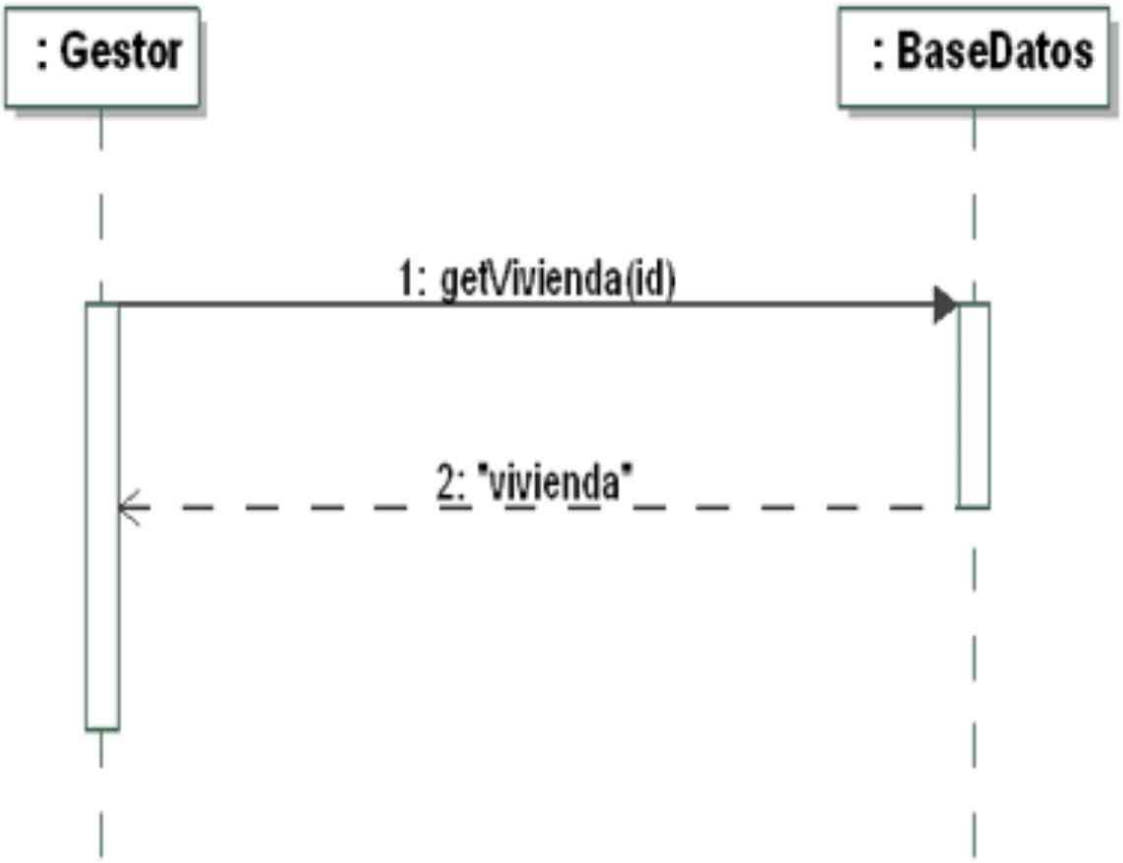


Figura 16.14. Diagrama de secuencias

El correspondiente código C++ asociado al diagrama puede tener dos variantes, según se disponga de un puntero a objeto o el objeto en sí:

```
Vivienda* viv = bd.get_vivienda(7); // Desde el objeto
```

ó

```
Vivienda* viv = bd->get_vivienda(7); // Desde el puntero
```

Estas llamadas se ubicarían dentro de algún método de la clase *Gestor*.

Creación, destrucción, automensajes y recursividad

Mostraremos ahora una secuencia de interacción en C++ donde se aprecia la creación, los mensajes reflexivos, la recursividad y la correspondiente destrucción del objeto.

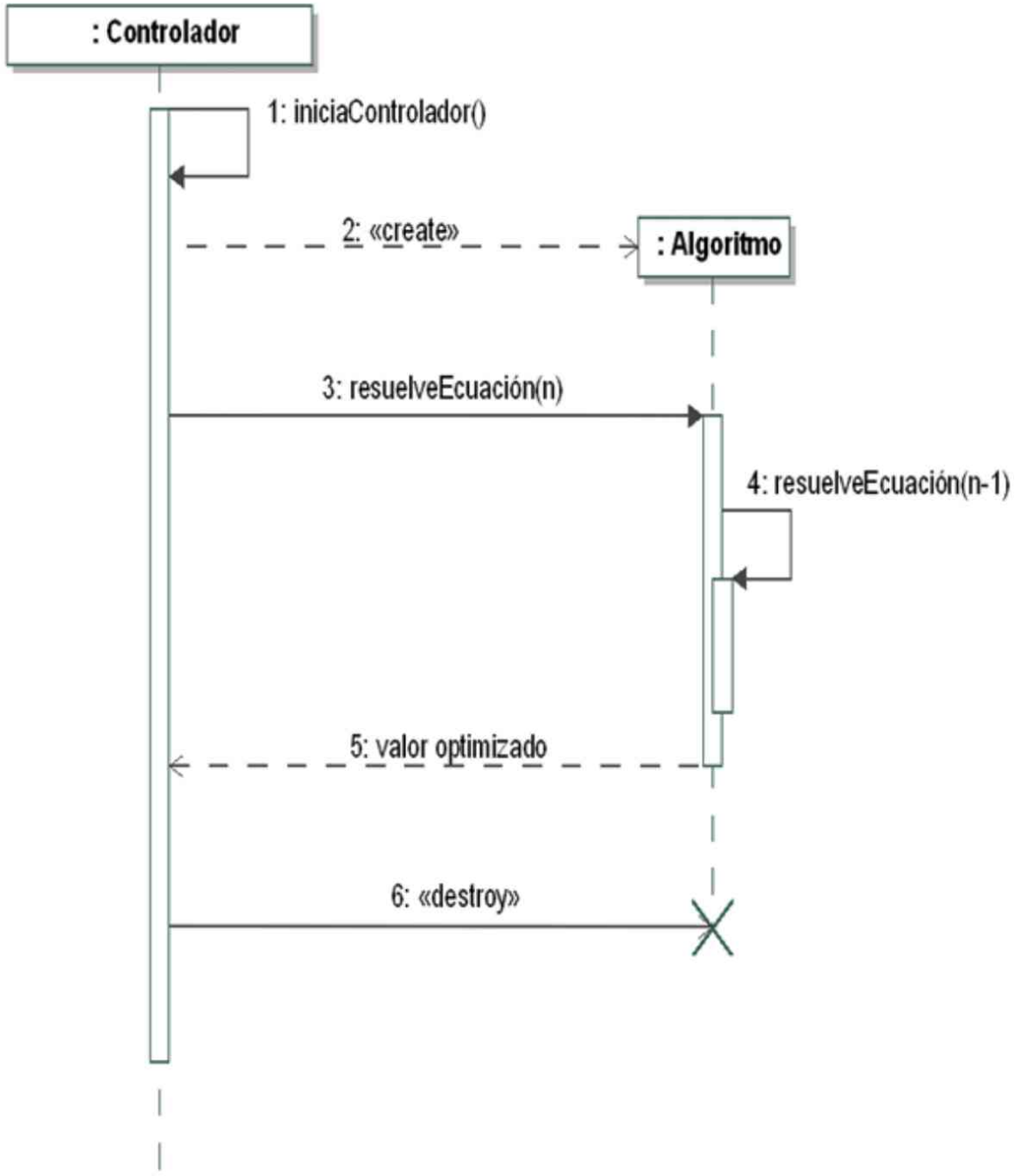


Figura 16.15. Diagrama de secuencias de un controlador que solicita una optimización

Listado 16.16 Implementación de la figura 16.15 :: fichero recursividad.h

```
#ifndef recursividad_h
#define recursividad_h
#include <vector>
#include <string>
#include <iostream>
using namespace std;
class Algoritmo
{
public:
int resuelve_ecuacion(const int n);
};
class Controlador
{
private:
Algoritmo* algoritmo;
public:
Controlador();
private:
void inicia_controlador();
};
#endif
```

Listado 16.17 Implementación de la figura 16.15 :: fichero recursividad.cpp

```
#include "recursividad.h"
using namespace std;
int Algoritmo::resuelve_ecuacion(const int n)
{
```

```
if (n <= 1)
{
return 1;
} else
{
return n * resuelve_ecuacion(n - 1);
}
}
void Controlador::inicia_controlador()
{
Algoritmo* algoritmo = new Algoritmo();
cout << algoritmo->resuelve_ecuacion(8);
delete algoritmo;
}
Controlador::Controlador()
{
inicia_controlador();
}
```

Puesto que estamos en C++ y hemos usado punteros, la invocación de destrucción del objeto *Algoritmo* debe ser explícita mediante *delete*.

Saltos condicionales

La implementación de los saltos condicionales se modela en UML mediante los fragmentos *alt*. Para ilustrar la ingeniería directa de un caso de sentencia condicional en UML aplicaremos el ejemplo visto en la sección 15.2.3.

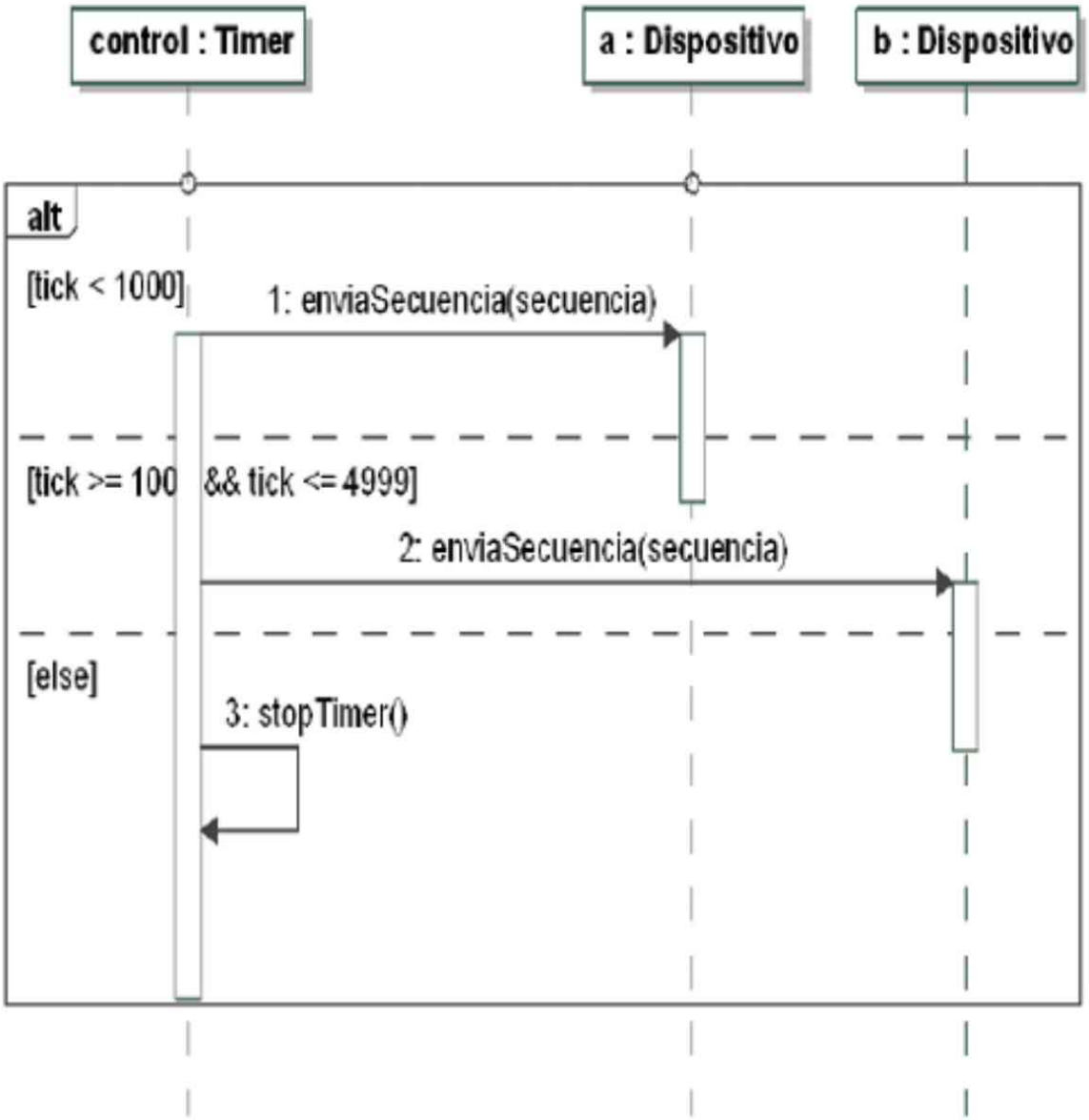


Figura 16.16. Diagrama de secuencias con fragmento condicional

Listado 16.18 Implementación de la figura 16.16 :: fichero *condicional.h*

```
#ifndef condicional_h
#define condicional_h
```

```
#include <vector>
#include <string>
#include <iostream>
// Clase dispositivo
class Dispositivo
{
public:
void envia_secuencia(int secuencia);
};
// Clase timer
class Timer
{
private:
int tick;
int secuencia;
Dispositivo* a;
Dispositivo* b;
public:
Timer();
~Timer(); // Destrutor
// Evento del timer
private:
void on_timer();
};
#endif
```

Listado 16.19 Implementación de la figura 16.16 :: fichero condicional.cpp

```
#include "condicional.h"
```

```
using namespace std;
void Dispositivo::envia_secuencia(int secuencia)
{
(...)
}
Timer::Timer()
{
tick = 0;
secuencia = 0;
a = new Dispositivo();
b = new Dispositivo();
start_timer();
}
Timer::~Timer()
{
delete a;
delete b;
}
void Timer::on_timer()
{
tick++;
if (tick < 1000)
{
a->envia_secuencia(secuencia);
}
else if (tick >= 1000 && tick <= 4999)
{
b->envia_secuencia(secuencia);
```

```
}
else stop_timer();
}
```

Iteraciones

C++ es un lenguaje ideal para iterar, ya que el código queda muy optimizado por el compilador. Al igual que en las sentencias condicionales tratadas en la sección anterior, los bucles se modelan también mediante un fragmento del tipo *loop*.

Veamos a continuación un ejemplo de un fragmento *loop* con la implementación de la sentencia *forEach*, tal como se vio en la sección 15.2.4:

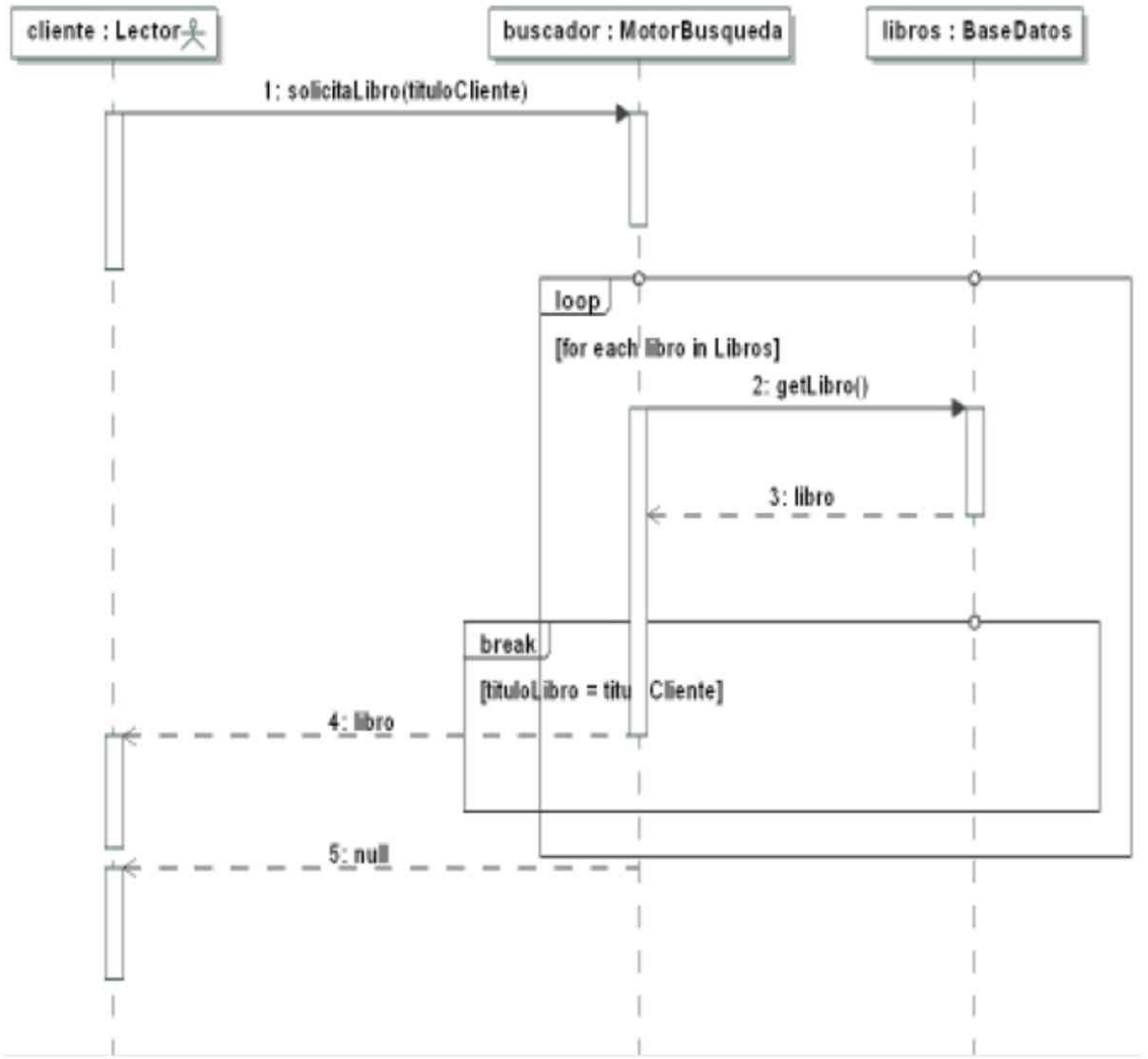


Figura 16.17. Diagrama de secuencias con fragmento iterativo

Listado 16.20 Implementación de iteración de la figura 16.17 :: busqueda.cpp

```
Libro* Motor_busqueda::solicita_libro(const string&
tituloCliente)
{
vector<Libro*>::iterator it;
for (it = registro_libros.begin(); it !=
registro_libros.end(); it++)
{
cout << "Libro: " << (*it)->get_titulo() <<
endl;
if ((*it)->get_titulo() == tituloCliente)
{
return *it; //break
}
}
return NULL;
}
int main()
{
Motor_busqueda motor;
if (motor.solicita_libro("Hiperión") != NULL)
{
cout << "Libro encontrado" << endl;
}
else
{
cout << "No encontrado" << endl;
}
cin.get();
```

```
return o;
}
```

En cuanto se ejecuta la guarda del fragmento *break* se procede a devolver el ítem solicitado al objeto actor.

diagramas de estado

La implementación de un diagrama de estados en C++ es muy similar a la vista anteriormente en Java. Para ello recurriremos al patrón *State* visto en el capítulo nueve sobre patrones de diseño. Su implementación requiere una versión ligeramente modificada de la que se explica en el código fuente de [Gamma95], utilizando clases abstractas y punteros miembro estáticos para mantener las referencias de las instancias de clases.

A modo de ejemplo explicaremos el código asociado al diagrama 16.18 que consta de seis estados: *inicio, comprobando, leyendo, almacenando, detenido* y *final*. Estos seis estados están relacionados con eventos ocurridos en el manejador de un dispositivo (*driver*) que procede a atender la petición de interrupción del hardware. El listado 16.20 recoge el código fuente con la implementación del autómata.

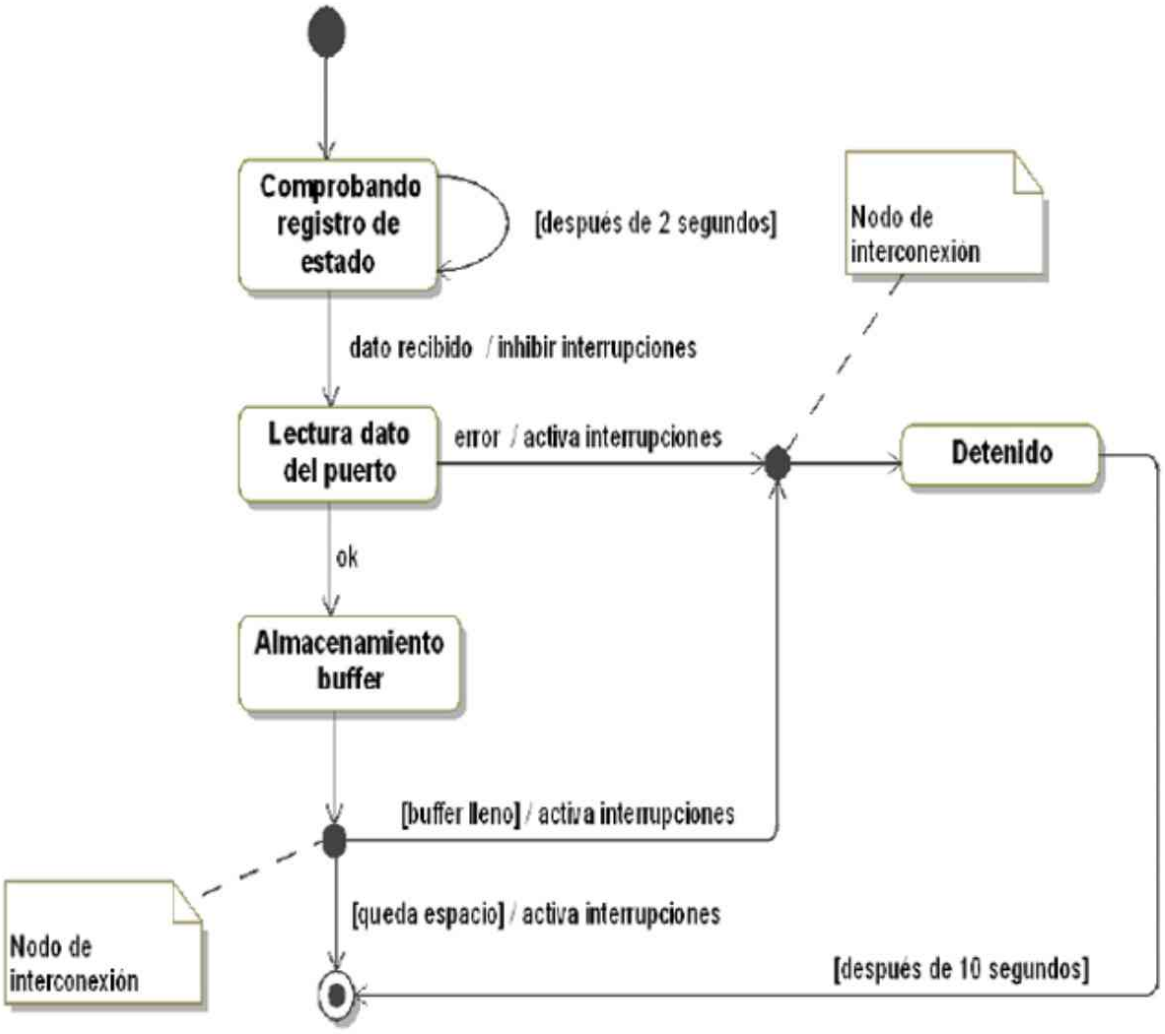


Figura 16.18. Diagrama de estado del procesamiento de un dato por un manejador de dispositivo

Listado 16.21 Implementación del diagrama de estados de la figura 16.18

```
#ifndef estados_h
#define estados_h
#include <vector>
#include <string>
#include <iostream>
using namespace std;
class Driver;
```

```
class Estado
{
protected:
string nombre_estado;
public:
Estado();
// Posibles eventos
virtual void iniciar(Driver* driver);
virtual void pasan_dos_segundos(Driver* driver);
virtual void dato_recibido(Driver* driver);
virtual void error_lectura(Driver* driver);
virtual void lectura_OK(Driver* driver);
virtual void buffer_lleno(Driver* driver);
virtual void queda_espacio_buffer(Driver* driver);
virtual void pasan_diez_segundos(Driver* driver);
virtual void imprime_estado(Driver* driver);
// Comprueba si en estado correcto
virtual void check_error(const string &supuesto);
protected:
void cambia_estado(Driver* driver, Estado* estado);
};
// Eventos concretos
class Est_inicio : public Estado
{
private:
static Estado* instancia;
public:
Est_inicio();
```

```
static Estado* get_instancia();
virtual void iniciar(Driver* driver);
};
class Est_comprobando : public Estado
{
private:
static Estado* instancia;
public:
Est_comprobando();
static Estado* get_instancia();
virtual void pasan_dos_segundos(Driver* driver);
virtual void dato_recibido(Driver* driver);
};
class Est_leyendo : public Estado
{
private:
static Estado* instancia;
public:
Est_leyendo();
static Estado* get_instancia();
virtual void error_lectura(Driver* driver);
virtual void lectura_OK(Driver* driver);
};
class Est_almacenando : public Estado
{
private:
static Estado* instancia;
public:
```

```
Est_almacenando();
static Estado* get_instancia();
virtual void buffer_lleno(Driver* driver);
virtual void queda_espacio_buffer(Driver* driver);
};
class Est_detenido : public Estado
{
private:
static Estado* instancia;
public:
Est_detenido();
static Estado* get_instancia();
virtual void pasan_diez_segundos(Driver* driver);
};
class Est_final : public Estado
{
private:
static Estado* instancia;
public:
Est_final();
static Estado* get_instancia();
};
// Clase de gestión del driver
class Driver
{
// Declaración de estados
private:
Estado* estado_actual;
```

```
public:
Driver();
// Eventos que ocurren en el driver
virtual void iniciar();
virtual void pasan_dos_segundos();
virtual void dato_recibido();
virtual void error_lectura();
virtual void lectura_OK();
virtual void buffer_lleno();
virtual void queda_espacio_buffer();
virtual void pasan_diez_segundos();
virtual void imprime_estado();
void cambia_estado(Estado* estado);
};
#endif
```

Listado 16.22 Implementación del diagrama de estados de la figura 16.18

```
#include "estados.h"
using namespace std;
Estado::Estado()
{
}
void Estado::iniciar(Driver* driver)
{
check_error("EstInicio");
}
void Estado::pasan_dos_segundos(Driver* driver)
{
```

```
check_error("EstComprobando");
}
void Estado::dato_recibido(Driver* driver)
{
check_error("EstComprobando");
}
void Estado::error_lectura(Driver* driver)
{
check_error("EstLeyendo");
}
void Estado::lectura_OK(Driver* driver)
{
check_error("EstLeyendo");
}
void Estado::buffer_lleno(Driver* driver)
{
check_error("EstAlmacenando");
}
void Estado::queda_espacio_buffer(Driver* driver)
{
check_error("EstAlmacenando");
}
void Estado::pasan_diez_segundos(Driver* driver)
{
check_error("EstDetenido");
}
void Estado::imprime_estado(Driver* driver)
{
```

```
cout << nombre_estado << endl;
}
void Estado::check_error(const string& supuesto)
{
if (nombre_estado != supuesto)
{
cout << "Error estado incorrecto" << endl;
}
}
void Estado::cambia_estado(Driver* driver, Estado* estado)
{
driver->cambia_estado(estado);
}
Estado* Est_inicio::instancia = NULL;
Est_inicio::Est_inicio()
{
nombre_estado = "EstInicio";
}
void Est_inicio::iniciar(Driver* driver)
{
cout << "iniciar()" << endl;
cambia_estado(driver, Est_comprobando::get_instancia());
}
Estado* Est_inicio::get_instancia()
{
if (instancia == NULL)
{
return (instancia = new Est_inicio());
```

```
} else return instancia;
}
Estado* Est_comprobando::instancia = NULL;
Est_comprobando::Est_comprobando()
{
nombre_estado = “EstComprobando”;
}
void Est_comprobando::pasan_dos_segundos(Driver* driver)
{
cout << “pasan_dos_segundos()” << endl;
}
void Est_comprobando::dato_recibido(Driver* driver)
{
cout << “dato_recibido()” << endl;
cambia_estado(driver, Est_leyendo::get_instancia());
}
Estado* Est_comprobando::get_instancia()
{
if (instancia == NULL)
{
return (instancia = new Est_comprobando());
} else return instancia;
}
Estado* Est_leyendo::instancia = NULL;
Est_leyendo::Est_leyendo()
{
nombre_estado = “EstLeyendo”;
}
```

```
void Est_leyendo::error_lectura(Driver* driver)
{
cout << “error_lectura()” << endl;
cambia_estado(driver, Est_detenido::get_instancia());
}
void Est_leyendo::lectura_OK(Driver* driver)
{
cout << “lectura_OK()” << endl;
cambia_estado(driver, Est_almacenando::get_instancia());
}
Estado* Est_leyendo::get_instancia()
{
if (instancia == NULL)
{
return (instancia = new Est_leyendo());
} else return instancia;
}
Estado* Est_almacenando::instancia = NULL;
Est_almacenando::Est_almacenando()
{
nombre_estado = “EstAlmacenando”;
}
void Est_almacenando::buffer_lleno(Driver* driver)
{
cout << “buffer_lleno()” << endl;
cambia_estado(driver, Est_detenido::get_instancia());
}
void Est_almacenando::queda_espacio_buffer(Driver* driver)
```

```
{
cout << “queda_espacio_buffer()” << endl;
cambia_estado(driver, Est_final::get_instancia());
}
Estado* Est_almacenando::get_instancia()
{
if (instancia == NULL)
{
return (instancia = new Est_almacenando());
} else return instancia;
}
Estado* Est_detenido::instancia = NULL;
Est_detenido::Est_detenido()
{
nombre_estado = “EstDetenido”;
}
void Est_detenido::pasan_diez_segundos(Driver* driver)
{
cout << “pasan_diez_segundos()” << endl;
cambia_estado(driver, Est_final::get_instancia());
}
Estado* Est_detenido::get_instancia()
{
if (instancia == NULL)
{
return (instancia = new Est_detenido());
} else return instancia;
}
```

```
Estado* Est_final::instancia = NULL;
Est_final::Est_final()
{
nombre_estado = “EstFinal”;
}
Estado* Est_final::get_instancia()
{
if (instancia == NULL)
{
return (instancia = new Est_final());
} else return instancia;
}
Driver::Driver()
{
// Estado inicial
estado_actual = Est_inicio::get_instancia();
}
void Driver::iniciar()
{
estado_actual->iniciar(this);
}
void Driver::pasan_dos_segundos()
{
estado_actual->pasan_dos_segundos(this);
}
void Driver::dato_recibido()
{
estado_actual->dato_recibido(this);
```

```
inhibe_interrupciones();
}
void Driver::error_lectura()
{
estado_actual->error_lectura(this);
activa_interrupciones();
}
void Driver::lectura_OK()
{
estado_actual->lectura_OK(this);
}
void Driver::buffer_lleno()
{
estado_actual->buffer_lleno(this);
activa_interrupciones();
}
void Driver::queda_espacio_buffer()
{
estado_actual->queda_espacio_buffer(this);
activa_interrupciones();
}
void Driver::pasan_diez_segundos()
{
estado_actual->pasan_diez_segundos(this);
}
void Driver::imprime_estado()
{
estado_actual->imprime_estado(this);
```

```
}
void Driver::cambia_estado(Estado* estado)
{
estado_actual = estado;
}
int main()
{
Driver* driver = new Driver()
// Comienza transiciones
driver->imprime_estado();
driver->iniciar();
driver->imprime_estado();
driver->imprime_estado();
driver->pasan_dos_segundos();
driver->imprime_estado();
driver->imprime_estado();
driver->pasan_dos_segundos();
driver->imprime_estado();
driver->imprime_estado();
driver->dato_recibido();
driver->imprime_estado();
driver->imprime_estado();
driver->error_lectura();
driver->imprime_estado();
driver->imprime_estado();
driver->pasan_diez_segundos();
driver->imprime_estado();
delete Est_inicio::get_instancia();
```

```
delete Est_comprobando::get_instancia();
delete Est_leyendo::get_instancia();
delete Est_almacenando::get_instancia();
delete Est_detenido::get_instancia();
delete Est_final::get_instancia();
delete driver;
}
```

caso de estudio: ajedrez

Terminaremos el capítulo explicando la implementación de los diagramas de secuencias de los apartados 7.6.1 y 7.6.2 donde se recogen diversos casos de uso asociados a una partida. Para la implementación de las interacciones es necesario definir primeramente las clases que se modelaron en la figura 6.21 con sus atributos y métodos. Este modelo estático nos servirá de base para dar forma al código asociado al modelo dinámico que representan los diagramas de secuencias.

Los diagramas de paquetes vistos en la figura 5.25 nos permitirán organizar el agrupamiento de los ficheros que implementan las clases en unidades lógicas. Así mismo el diagrama de componentes que refleja la figura 5.11 nos orientará con mayor facilidad al establecimiento de las interfaces.

Finalmente, cabe decir que el código aquí expuesto es un simple prototipo de prueba que no implementa íntegramente toda la lógica de negocio que requiere una aplicación de esta envergadura. La idea de partida es, por tanto, aproximar los conceptos de UML a una aplicación real y sin ánimo de entrar en demasiados detalles irrelevantes para el ámbito de este libro.

A continuación se exponen los ficheros de código fuente ordenados por sus correspondientes paquetes:

Listado 16.23 Paquete comunicaciones – fichero de cabecera

```
#ifndef comunicaciones_h
#define comunicaciones_h
#include <iostream>
#include <string>
#include <vector>
using namespace std;
namespace comunicaciones
```

```
{
// Interfaz I_observer
class I_observer
{
public:
virtual void notifica_mensaje(const string&
mensaje) = 0;
};
// Interfaz I_comunicacion
class I_comunicacion
{
public:
virtual void enviar_mensaje(const string&
mensaje) = 0;
virtual void recibe_mensaje(const string&
mensaje) = 0;
};
class Notificador
{
private:
vector<I_observer*> lista_observers;
public:
void adjunta(I_observer* observer);
void notifica(const string& mensaje);
};
// Implementa interfaz I_comunicacion
class Fachada_comunicaciones : public I_comunicacion
{
```

```
private:
Notificador notificador;
public:
void enviar_mensaje(const string& mensaje);
void recibe_mensaje(const string& mensaje);
Notificador& get_notificador();
};
}
#endif
```

Listado 16.24 Paquete comunicaciones – código fuente

```
#include "comunicaciones.h"
using namespace comunicaciones;
void Fachada_comunicaciones::enviar_mensaje(const string& mensaje)
{
cout << "Enviando mensaje: " << mensaje << endl;
}
void Fachada_comunicaciones::recibe_mensaje(const string& mensaje)
{
cout << "Frontera_comunicaciones::recibe_mensaje(): "
<< mensaje << endl;
notificador.notifica(mensaje);
}
Notificador& Fachada_comunicaciones::get_notificador()
{
return notificador;
}
void Notificador::adjunta(I_observer* observer)
```

```
{
lista_observers.push_back(observer);
}
void Notificador::notifica(const string& mensaje)
{
vector<I_observer*>::iterator it;
for(it = lista_observers.begin(); it !=
lista_observers.end(); it++)
(*it)->notifica_mensaje(mensaje);
}
```

En este paquete se implementa el patrón *Observer*, relacionado con las clases que gestionan el envío y notificación a los oyentes de la recepción de mensajes por el *socket*. Dichas clases son piezas clave de la arquitectura cliente/servidor de la aplicación de ajedrez, pues las facilidades de juego en red multijugador así como el registro en el servidor son proporcionadas por ellas. La definición de la interface *I_observer* es exportada por el paquete para su implementación en las clases que requiren de notificaciones de recepción de mensajes. Igualmente, la interface *I_comunicacion* se implementará en las clases que necesiten de envío de mensajes y gestión de las comunicaciones.
Aunque aquí no se ha implementado, este paquete en concreto está estrechamente relacionado con su homólogo en la parte del servidor. Su función es conectarse a él y gestionar la transmisión de mensajes.

Listado 16.25 Paquete GUI – fichero de cabecera

```
#ifndef gui_h
#define gui_h
#include <iostream>
```

```
#include "../comunicaciones/comunicaciones.h"
#include "../ajedrez/ajedrez.h"
using namespace comunicaciones;
using namespace ajedrez;
namespace GUI
{
// Interfaz I_gui
class I_gui
{
public:
virtual void on_destino(int x_dest, int
y_dest) = 0;
virtual void on_origen(int x_orig, int y_orig)
= 0;
virtual void on_consejo() = 0;
};
// Implementa interfaz I_observer
class Gui : public comunicaciones::I_observer
{
private:
Tablero& tablero_agr;
public:
Gui(Tablero& tablero);
void notifica_mensaje(const string& mensaje);
void dibuja_tablero();
void dibuja_marcador();
void dibuja_interfaz();
void alerta();
```

```
};
}
#endif
```

Listado 16.26 Paquete GUI – código fuente

```
#include "gui.h"
using namespace std;
using namespace GUI;
Gui::Gui(Tablero& tablero):tablero_agr(tablero)
{
}
void Gui::notifica_mensaje(const string& mensaje)
{
cout << "Gui::notifica_mensaje(): " << mensaje <<
endl;
}
void Gui::dibuja_interfaz()
{
cout << "Gui::dibua_interfaz()" << endl;
}
void Gui::dibuja_tablero()
{
cout << "Gui::dibuja_tablero()" << endl;
tablero_agr.dibuja_tablero();
}
void Gui::dibuja_marcador()
{
cout << "Gui::dibuja_marcador()" << endl;
```

```
}
void Gui::alerta()
{
cout << "Gui::alerta()" << endl;
}
```

El paquete *GUI* se encarga de definir la interfaz *I_gui* que será implementada en el motor de juego para recibir los mensajes de pulsación del ratón y del teclado. También se define la clase *Gui* que es la encargada de gestionar las funciones de los gráficos del juego y la interfaz de usuario.

Listado 16.27 Paquete Gestion_juego – fichero de cabecera

```
#ifndef gestion_juego_h
#define gestion_juego_h
#include <iostream>
#include "../comunicaciones/comunicaciones.h"
#include "../gui/gui.h"
#include "../ajedrez/ajedrez.h"
using namespace ajedrez;
using namespace GUI;
using namespace comunicaciones;
using namespace std;
namespace gestion_juego
{
class Time
{
};
// Implementa interfaces I_gui e I_observer
```

```
class Control_juego : public GUI::I_gui, public
comunicaciones::I_observer
{
private:
Time tiempo_limite;
Time tiempo_a;
Time tiempo_b;
int turno;
int fil_orig;
int col_orig;
int fil_dest;
int col_dest;
GUI::Gui& gui_ref;
comunicaciones::I_comunicacion& com_ref;
ajedrez::Jugador_humano& humano_ref;
ajedrez::IA& ia_ref;
public:
Control_juego(GUI::Gui& gui, comunicaciones::I_comunicacion& com,
ajedrez::Jugador_humano& humano, ajedrez::IA& ia);
// Heredadas de I_gui
void on_destino(int x_dest, int y_dest);
void on_origen(int x_orig, int y_orig);
void on_consejo();
// Heredadas de I_observer
void notifica_mensaje(const string& mensaje);
// Propias
void aconsejar();
void inicia_partida();
```

```
void termina_partida();
void buscar_contrincante();
void turno_ia();
private:
void hacer_movimiento(int fil_orig, int
col_orig, int fil_dest, int col_dest);
bool detecta_amenaza(int fila, int columna);
bool jugada_correcta(int fila, int columna,
const string& pieza);
bool mate(int fila, int columna);
int convierte_xy(int cord) ;
};
}
#endif
```

Listado 16.28 Paquete Gestion_juego – código fuente

```
#include "gestion_juego.h"
using namespace gestion_juego;
Control_juego::Control_juego(GUI::Gui& gui, comunicaciones::I_-
comunicacion& com, ajedrez::Jugador_humano& humano, ajedrez::IA&
ia):gui_ref(gui), com_ref(com), humano_ref(humano), ia_ref(ia)
{
}
void Control_juego::on_origen(int x_orig, int y_orig)
{
cout << "Control_juego::on_origen()" << endl;
cout << "x_orig: " << x_orig << endl;
cout << "y_orig: " << y_orig << endl;
```

```
// Realiza conversion coordenadas de pantalla a filas
y columnas
fil_orig = convierte_xy(y_orig);
col_orig = convierte_xy(x_orig);
}
void Control_juego::on_destino(int x_dest, int y_dest)
{
cout << "Control_juego::on_destino()" << endl;
cout << "x_dest: " << x_dest << endl;
cout << "y_dest: " << y_dest << endl;
fil_dest = convierte_xy(y_dest);
col_dest = convierte_xy(x_dest);
hacer_movimiento(fil_orig, col_orig, fil_dest,
col_dest);
}
void Control_juego::on_consejo()
{
cout << "Control_juego::on_consejo()" << endl;
aconsejar();
}
void Control_juego::notifica_mensaje(const string& mensaje)
{
cout << "Control_juego::notifica_mensaje(): " <<
mensaje << endl;
}
void Control_juego::aconsejar()
{
cout << "Control_juego::aconsejar()" << endl;
```

```
cout << “Aconsejando” << endl;
cout << “Debe mover la pieza: “ <<
ia_ref.aconsejar()->get_tipo() << endl;
}
void Control_juego::hacer_movimiento(int fil_orig, int col_orig, int
fil_dest, int col_dest)
{
cout << “Control_juego::hacer_movimiento()” << endl;
cout << “haciendo movimiento desde “ << fil_orig <<
“,” << col_orig << “ a “ << fil_dest << “,” <<
col_dest << endl;
Pieza* pieza =
humano_ref.get_contenido(fil_orig,col_orig);
if (jugada_correcta(fil_dest, col_dest,
pieza->get_tipo()))
{
humano_ref.mueve_pieza(fil_dest, col_dest, *pieza);
gui_ref.dibuja_tablero();
if (mate(fil_dest, col_dest))
{
pieza = ia_ref.get_contenido(fil_dest,col_dest);
cout << “Se ha dado mate a pieza: “ <<
pieza->get_tipo() << endl;
}
else if (detecta_amenaza(fil_dest, col_dest))
gui_ref.alerta();
}
}
```

```
void Control_juego::inicia_partida()
{
cout << “Control_juego::iniciando_partida()” << endl;
}
void Control_juego::termina_partida()
{
cout << “Control_juego::termina_partida()” << endl;
}
void Control_juego::buscar_contrincante()
{
cout << “Control_juego::buscando_contrincante()” <<
endl;
}
void Control_juego::turno_ia()
{
cout << “Control_juego::turno_ia()” << endl;
Pieza* pieza =
ia_ref.pensar_jugada(fil_dest,col_dest,”Torre2”);
cout << “IA ha decidio mover la pieza: “ <<
pieza->get_tipo() << endl;
fil_dest = pieza->get_posicion().fila;
col_dest = pieza->get_posicion().columna;
if (jugada_correcta(fil_dest, col_dest,
pieza->get_tipo()))
{
ia_ref.mueve_pieza(fil_dest, col_dest, *pieza);
pieza = humano_ref.get_contenido(fil_dest, col_dest);
if (detecta_amenaza(fil_dest, col_dest))
```

```
cout << “La pieza de la IA está siendo
amenazada” << endl;
if (mate(fil_dest, col_dest))
{
cout << “Jaque al “ << pieza->get_tipo()
<< endl;
if (humano_ref.elimina_pieza(pieza))
delete pieza;
termina_partida();
}
}
}
bool Control_juego::detecta_amenaza(int fila, int columna)
{
cout << “Control_juego::detectando amenaza()” <<
endl;
return (((fila == 7) && (columna == 4)) ? true :
false);
}
bool Control_juego::jugada_correcta(int fila, int columna, const string&
pieza)
{
cout << “Control_juego::jugada correcta()” << endl;
return true;
}
bool Control_juego::mate(int fila, int columna)
{
cout << “Control_juego::mate()” << endl;
```

```
return (((fila == 7) && (columna == 4)) ? false :
true);
}
int Control_juego::convierte_xy(int cord)
{
cout << "Control_juego::convierte_xy()" << endl;
return cord / 100;
}
```

Este paquete representa el núcleo principal del juego donde se centraliza la gestión y la coordinación de las diversas acciones a llevar a cabo durante el progreso de las partidas. Controla, entre otras cosas, el cambio de turno, los tiempos de cada jugador, la verificación de jugadas, el inicio y la terminación de la partida, la coordinación de las jugadas de la IA así como el asesoramiento de jugadas maestras.

Después de la recepción de los mensajes de posición del ratón y su correspondiente conversión a filas y columnas, el código recoge la mayor parte de las interacciones llevadas a cabo en las secciones 7.6.1 (método *hacer_movimiento*) y 7.6.2 (método *turno_ia*). Como nota curiosa de su complejidad, este paquete implementa dos interfaces y mantiene cuatro referencias a diferentes clases del modelo.

Listado 16.29 Paquete Ajedrez – fichero de cabecera

```
#ifndef ajedrez_h
#define ajedrez_h
#include "../comunicaciones/comunicaciones.h"
#include <vector>
#include <map>
```

```
#include <string>
#include <iostream>
#include <algorithm>
using namespace std;
using namespace comunicaciones;
namespace ajedrez
{
class Jugada
{
public:
int fila;
int columna;
};
class Pieza
{
private:
string estado;
Jugada posicion;
string tipo;
public:
Pieza(const string& nombre);
void dibuja_pieza();
void set_posicion(int fila, int columna);
const Jugada& get_posicion();
const string& get_tipo();
};
class Jugador
{
```

```
protected:
int color;
map<string, Pieza*> mapa_piezas;
vector<Pieza*> piezas;
public:
~Jugador();
void mueve_pieza(int fila, int columna, Pieza&
p);
Pieza* get_contenido(int fila, int columna);
map<string, Pieza*>* get_mapa_piezas();
bool elimina_pieza(Pieza* pieza);
private:
void destruye_piezas();
};
class Jugador_humano : public Jugador
{
private:
string nombre;
string password;
I_comunicacion& com_ref;
public:
Jugador_humano(I_comunicacion& com);
void registrarse(const string& nombre, const string& password);
int get_ELO();
};
class Sistema_IA;
class IA : public Jugador
{
```

```
private:
int nivel;
Sistema_IA* algoritmo_agr;
public:
IA(Jugador_humano& humano);
Pieza* pensar_jugada(int fila, int columna,
string p);
Pieza* aconsejar();
void set_algoritmo(Sistema_IA* sistema);
};
class Heuristica
{
private:
float puntuacion;
public:
virtual float get_puntuacion() = 0;
};
class Final : public Heuristica
{
public:
float get_puntuacion();
};
class Sistema_IA
{
protected:
Jugador_humano& humano_ref;
IA& ia_ref;
Heuristica& heu_ref;
```

```
public:
Sistema_IA(Jugador_humano& humano, IA& ia,
Heuristica& heu);
virtual Pieza* aconsejar() = 0;
virtual Pieza* calcula_jugada(int fila, int
columna, const string& p) = 0;
};
class Algoritmo_secuencial : public Sistema_IA
{
public:
Pieza* aconsejar();
Pieza* calcula_jugada(int fila, int columna,
const string& p);
};
class Algoritmo_GPU : public Sistema_IA
{
public:
Algoritmo_GPU(Jugador_humano& humano, IA& ia,
Heuristica& heu);
Pieza* aconsejar();
Pieza* calcula_jugada(int fila, int columna,
const string& p);
};
// Interfaz I_tablero
class I_tablero
{
public:
virtual void dibuja_tablero() = 0;
```

```
};
// Implementa interfaces I_observer e I_tablero
class Tablero : public I_observer, public I_tablero
{
public:
void notifica_mensaje(const string& mensaje);
void dibuja_tablero();
};  –
}
#endif
```

Listado 16.30 – Paquete Ajedrez – código fuente

```
#include "ajedrez.h"
using namespace ajedrez;
Pieza::Pieza(const string& nombre):tipo(nombre)
{
}
void Pieza::dibuja_pieza()
{
cout << "Pieza::dibuja_pieza()" << endl;
}
void Pieza::set_posicion(int fila, int columna)
{
cout << "Pieza::set_posicion()" << endl;
posicion.fila = fila;
posicion.columna = columna;
}
const Jugada& Pieza::get_posicion()
```

```
{
return posicion;
}
const string& Pieza::get_tipo()
{
return tipo;
}
Jugador::~Jugador()
{
destruye_piezas();
}
void Jugador::mueve_pieza(int fila, int columna, Pieza& p)
{
cout << "Jugador::mueve_pieza()" << endl;
p.set_posicion(fila, columna);
p.dibuja_pieza();
}
// Obtiene la pieza que ocupa la posición fila,columna
Pieza* Jugador::get_contenido(int fila, int columna)
{
cout << "Jugador::get_contenido()" << endl;
map<string, Pieza*>::iterator it;
for(it = mapa_piezas.begin(); it !=
mapa_piezas.end(); it++)
if (((*it).second->get_posicion().fila ==
fila)&& ((*it).second->get_posicion().columna
== columna))
return (*it).second;
```

```
return NULL;
}
map<string, Pieza*>* Jugador::get_mapa_piezas()
{
cout << "Jugador::get_mapa_piezas()" << endl;
return NULL;
}
void Jugador::destruye_piezas()
{
vector<Pieza*>::iterator it;
for (it = piezas.begin(); it != piezas.end(); it++)
delete *it;
}
bool Jugador::elimina_pieza(Pieza* pieza)
{
cout << "Jugador::elimina_pieza()" << endl;
vector<Pieza*>::iterator it = find(piezas.begin(),
piezas.end(), pieza);
if (it != piezas.end())
{
piezas.erase(it);
return true;
} else return false;
}
Jugador_humano::Jugador_humano(I_comunicacion& com):com_ref(com)
{
Pieza* torre2 = new Pieza("Torre2");
torre2->set_posicion(2,4);
```

```
piezas.push_back(torre2);
mapa_piezas["Torre2"] = torre2;
Pieza* peon4 = new Pieza("Peon4");
peon4->set_posicion(5,3);
piezas.push_back(peon4);
mapa_piezas["Peon4"] = peon4;
Pieza* rey = new Pieza("Rey");
rey->set_posicion(1,2);
piezas.push_back(rey);
mapa_piezas["Rey"] = rey;
}
void Jugador_humano::registrarse(const string& nombre, const string& password)
{
cout << "Jugador::registrarse()" << endl;
this->nombre = nombre;
this->password = password;
}
int Jugador_humano::get_ELO()
{
cout << "Jugador::get_ELO()" << endl;
return 10;
}
IA::IA(Jugador_humano& humano)
{
Pieza* caballo1 = new Pieza("Caballo1");
caballo1->set_posicion(3,1);
piezas.push_back(caballo1);
```

```
mapa_piezas["Caballo1"] = caballo1;
}
void IA::set_algoritmo(Sistema_IA* sistema)
{
algoritmo_agr = sistema;
}
Pieza* IA::pensar_jugada(int fila, int columna, string p)
{
cout << "IA::pensar_jugada()" << endl;
return algoritmo_agr->calcula_jugada(fila, columna,
p);
}
Pieza* IA::aconsejar()
{
cout << "IA::aconsejar()" << endl;
return algoritmo_agr->aconsejar();
}
float Final::get_puntuacion()
{
cout << "Final::get_puntuacion()" << endl;
return 4.3f;
}
Sistema_IA::Sistema_IA(Jugador_humano& humano, IA& ia, Heuristica&
heu):humano_ref(humano), ia_ref(ia), heu_ref(heu)
{
}
Pieza* Algoritmo_secuencial::aconsejar()
{
```

```
return NULL;
}
Pieza* Algoritmo_secuencial::calcula_jugada(int fila, int columna, const string& p)
{
return NULL;
}
Algoritmo_GPU::Algoritmo_GPU(Jugador_humano& humano, IA& ia, Heuristica& heu):Sistema_IA(humano, ia, heu)
{
}
Pieza* Algoritmo_GPU::aconsejar()
{
cout << "Algoritmo_GPU::aconsejar()" << endl;
return humano_ref.get_contenido(5,3);
}
Pieza* Algoritmo_GPU::calcula_jugada(int fila, int columna, const string& p)
{
cout << "Algoritmo_GPU::calcula_jugada()" << endl;
map<string, Pieza*>* mapa =
humano_ref.get_mapa_piezas();
if (heu_ref.get_puntuacion() > 1.3f)
{
// Procesa en paralelo
Pieza* pieza = ia_ref.get_contenido(3,1);
// Procesa en paralelo
pieza->set_posicion(1,2);
```

```
return pieza;
}
return NULL;
}
void Tablero::notifica_mensaje(const string& mensaje)
{
cout << "Tablero::notifica_mensaje(): " << mensaje <<
endl;
}
void Tablero::dibuja_tablero()
{
cout << "Tablero::dibuja_tablero()" << endl;
}
```

Finalmente, en el paquete *Ajedrez* se engloba la lógica de negocio relacionada con las entidades del juego completo. Incluye la clase que gestiona las piezas para dibujarlas, posicionarlas y mantener su información. También implementa la jerarquía de clases que representan a los jugadores: *los humanos* y *la inteligencia artificial*. Para esta última se incluye la clase abstracta que define la interfaz para el acceso al algoritmo de cálculo de jugadas y asesoramiento. Esta jerarquía no es más que una aplicación del patrón *Strategy* visto en el capítulo nueve, la cual contiene dos clases heredadas en las que se implementa el algoritmo secuencial y el algoritmo paralelo basado en GPGPU respectivamente.

ingeniería directa en python

«[...] Moisés hizo una serpiente de bronce y la puso sobre un mástil, y si alguien había sido mordido por una serpiente, miraba fijamente la serpiente de bronce y vivía».

(Números 21:4-9).

En este capítulo abordamos la implementación de la notación UML de cada tipo de diagrama mediante el lenguaje de programación Python. Pyhton es un lenguaje emergente y posicionado, junto a Java y C/C++, en lo más alto de los "rankings" de los lenguajes de programación más demandados por las empresas actualmente.

Aunque Python no posee la misma eficiencia que el código C/C++, su ventaja de lenguaje altamente legible y su filosofía multiparadigma y multiplataforma, lo hace una herramienta muy competitiva en el mundo del desarrollo de software. Una de sus características positivas es la de una sintaxis menos compleja que la de Java o C++, lo que permite escribir código de forma productiva y fácilmente, sin los añadidos tediosos de otros lenguajes más antiguos.

Puesto que Python está influido por los lenguajes estructurados C y Java, entre otros, sus facilidades y características versátiles para la programación orientada a objetos lo hacen candidato perfecto para la ingeniería directa de UML.

Comenzaremos este capítulo explicando la conversión de UML a código Pyhton para cada uno de sus diagramas y finalizaremos detallando la fase de implementación de la aplicación de cifrado remoto que hemos ido modelando en los capítulos anteriormente vistos.

diagramas de clases

Los diagramas de clases son modelos estáticos que definen en detalle las principales entidades del dominio de la aplicación. Mediante la especificación de los campos o atributos y métodos en el diagrama, tenemos información sufienciente para convertir esta estructura en código Python.

Representación de una clase

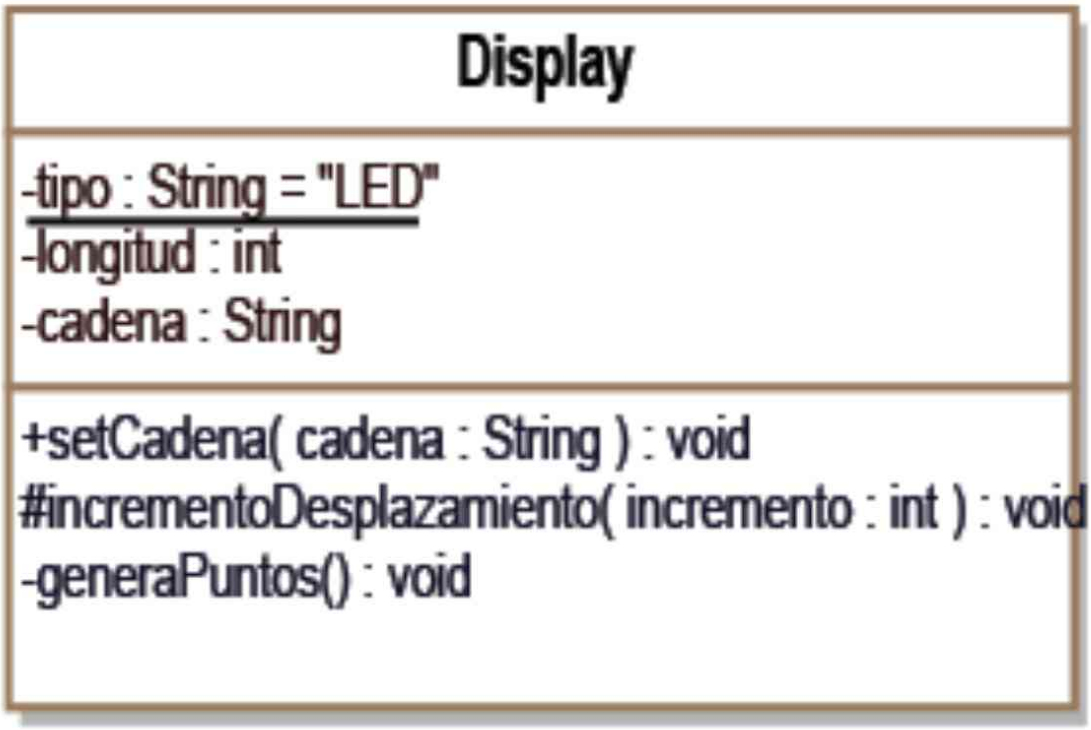


Figura 17.1. Ejemplo de clase en UML

Listado 17.1 Implementación de la clase Display

```
class Display:
__tipo = "LED"
#Constructor
def __init__(self):
pass
#Método público
def set_cadena(self, cadena):
self.__cadena = cadena
print(self.__cadena)
#Método protegido
```

```
    def _incrementa_desplazamiento(self, incremento):
    pass
    #Método privado
    def __genera_puntos(self):
    pass
```

En el listado 17.1 podemos apreciar la simplicidad del código con respecto a Java y C++ para implementar la clase *Display*.

Asociaciones

Las asociaciones, tal como se explicó en los capítulos 15 y 16, permiten la visibilidad o referencia a nivel de atributo de otros objetos. En Python es posible también tener asociaciones 1:1 y 1:*.

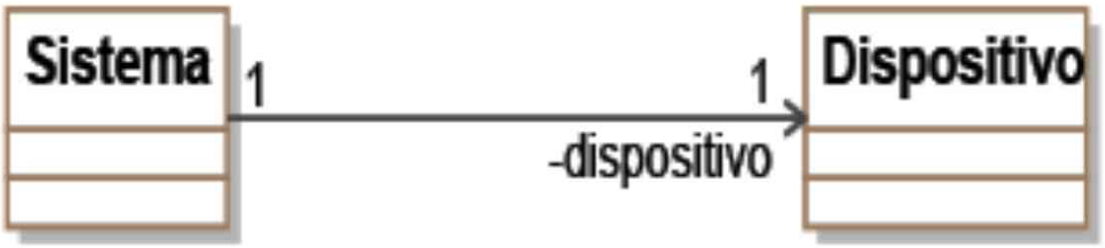


Figura 17.2. Asociación simple

Puesto que en Python no es necesario especificar el tipo de dato junto al identificador debido al tipado dinámico, esta opción queda postergada a la utilización en un método:

Listado 17.2 Definición de asociación simple en constructor

```
class Sistema:
def __init__(self, dispositivo):
self.__dispositivo = dispositivo
```

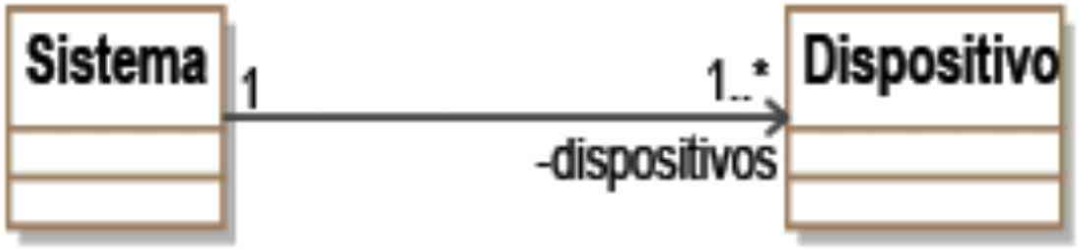


Figura 17.3. Asociación con cardinalidad 1..*

Para el caso uno a muchos crearemos una lista de dispositivos:

Listado 17.3 Definición de asociación múltiple en constructor

```
class Sistema:
def __init__(self):
self.__dispositivos = []
def add_dispositivo(self, dispositivo):
self.__dispositivos.append(dispositivo)
```

Herencia simple

Las ventajas de la herencia son múltiples en programación orientada a objetos. Nos permite evitar la duplicación de código, la reutilización del mismo, así como mejorar la mantenibilidad y la ampliabilidad de la aplicación en futuras versiones.

Diagrama de clases y el código Python:

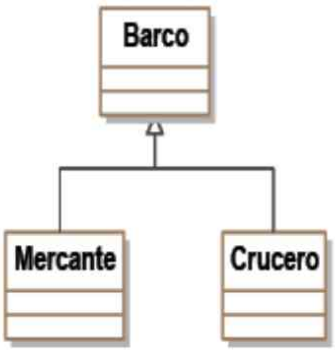


Figura 17.4. Herencia simple

```
class Barco:
pass
class Mercante(Barco):
pass
class Crucero(Barco):
pass
```

Herencia múltiple

Como en C++, Python también posee el mecanismo de la herencia múltiple, pero como comentábamos en el capítulo anterior, existe mucha controversia entre los expertos en relación al uso de este tipo de herencia, ya que suele conducir a problemas de implementación y de diseño, lo que convierte a la aplicación en un sistema complejo. Por ello, siempre que se pueda es preferible decantarse por herencia simple y patrones de diseño que simplifiquen en gran medida esta situación.



Figura 17.5. Herencia múltiple

```
class Alumno:
pass
class Investigador:
pass
class Doctorando(Alumno,Investigador):
pass
```

Implementación de interface

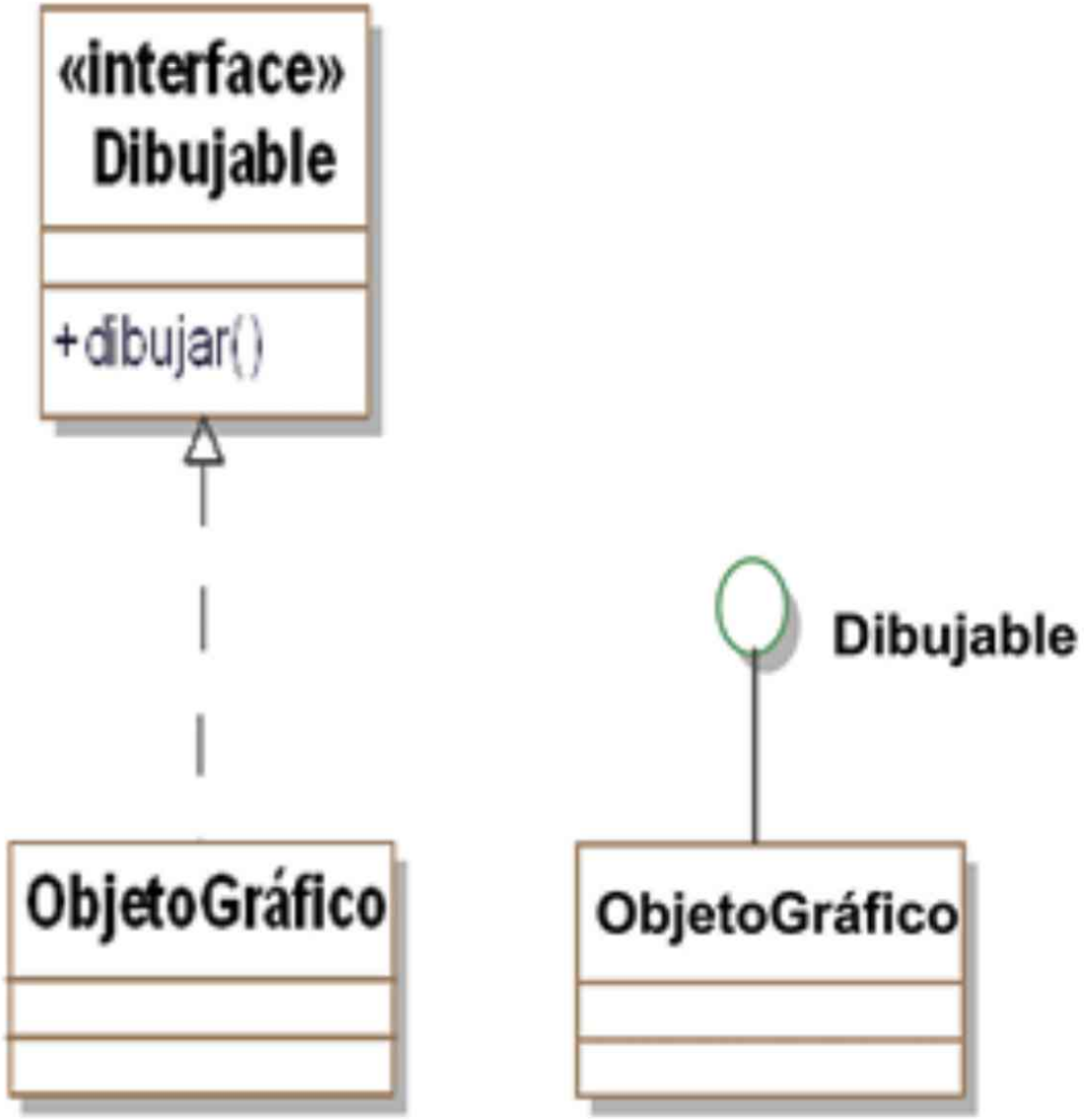


Figura 17.6. Herencia de interface (dos versiones de representación)

Listado 17.4 Implementación de una interface

```
import abc
class Dibujable(metaclass = abc.ABCMeta):
@abc.abstractmethod
def dibujar(self):
```

```
    pass
class ObjetoGrafico(Dibujable):
def dibujar(self):
(...) #Implementación
```

Con el fin de crear una interface en Pyhon es necesario importar el paquete *abc* y especificar la interface con la siguiente definición: metaclass = abc.ABCMeta. Finalmente, recurriremos al decorador de Python @abc.abstractmethod con el propósito de indiciar al intérprete que estamos utilizando un método de interface abstracto.

Agregación

Con la agregación tenemos un tipo de asociación donde la clase contiene las partes de forma no estricta, es decir, las partes pueden ser compartidas por otras clases del sistema, teniendo éstas su propio ciclo de vida no dependiente de la clase contenedora.



Figura 17.7. Agregación simple

```
class Persona:
def __init__(self, empleo):
self.__empleo = empleo
```

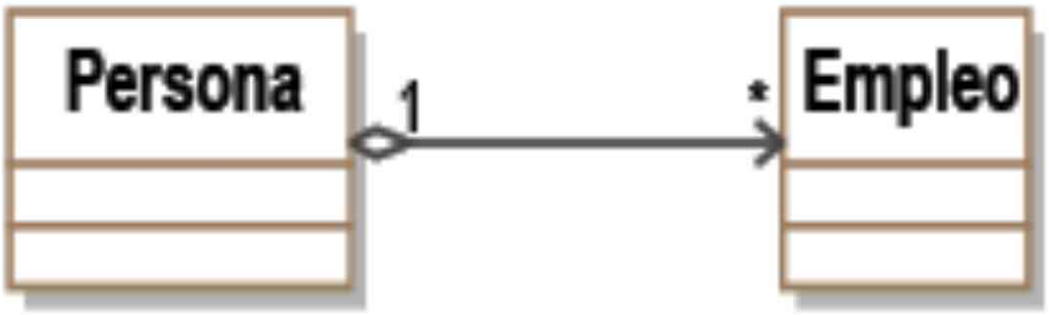


Figura 17.8. Agregación múltiple

La agregación múltiple en Python puede implementarse con listas:

```
class Persona:
def __init__(self):
self.__empleos = []
def add_empleo(self, empleo):
self.__empleos.append(empleo)
```

Composición

La composición es un tipo de asociación estricta en el que el todo contiene a las partes de forma inseparable, es decir, al desaparecer la parte contenedora deben destruirse también los objetos relacionados con las partes contenidas. Además, la relación que supone las partes con el todo son semánticamente un conjunto indivisible, como el ejemplo visto en el capítulo 4 sobre el libro con sus respectivas partes: índice, páginas, portada, figuras, etc.

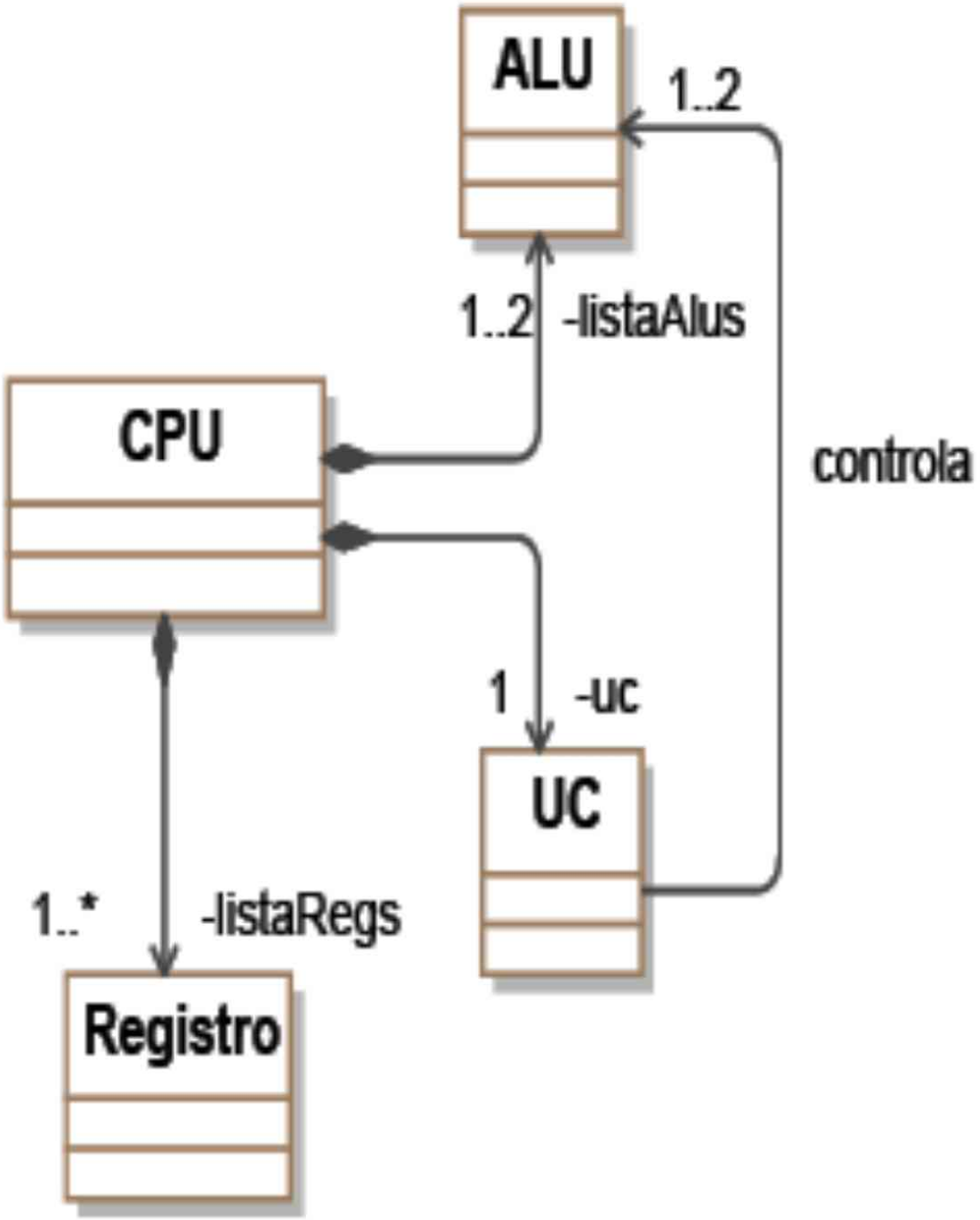


Figura 17.9. Diagrama de clases con composiciones

La composición se puede implementar en Python de la misma forma que en los ejemplos de la asociación y la agregación, esto es, mediante listas, ya que la forma de implementación en Python es independiente de la semántica de este tipo de símbolo UML.

Listado 17.5 Implementación de la figura 17.9

```
# Clase ALU
class ALU:
pass
# Clase Unidad de Control
class UC:
def __init__(self):
self.__alus = []
def set_ALUS(self, alu1, alu2):
self.__alus.append(alu1)
self.__alus.append(alu2)
def imprime_tipo_ALU(self):
print(self.__alus[0].unidad)
print(self.__alus[1].unidad)
# Clase Registro
class Registro:
pass
#Clase CPU
class CPU:
def __init__(self):
# Crea composiciones
self.__lista_alus = []
self.__lista_regs = []
self.__uc = UC();
reg = Registro()
reg.tipo = "PG 1"
self.__lista_regs.append(reg)
alu1 = ALU()
```

```
alu1.unidad = “Suma”
alu2 = ALU()
alu2.unidad = “Resta”
self.__lista_alus = [alu1, alu2];
# Pasa referencias
self.__uc.set_ALUS(alu1,alu2);
self.__uc.imprime_tipo_ALU()
CPU()
```

Clases abstractas

Las clases abstractas permiten crear clases que solo funcionan como superclases y de las cuales no se pueden crear instancias. Permiten definir un comportamiento común a un conjunto de subclases por medio del polimorfismo. Aunque es preferible utilizar las interfaces por su ligereza, flexibilidad y ampliabilidad, optaremos por la abstracción cuando necesitemos una entidad intermedia en la clase concreta y la pura interfaz abstracta. En el siguiente ejemplo se implementará una clase abstracta *Figura2D* que será heredada en las clases *Circulo* y *Rectangulo*.

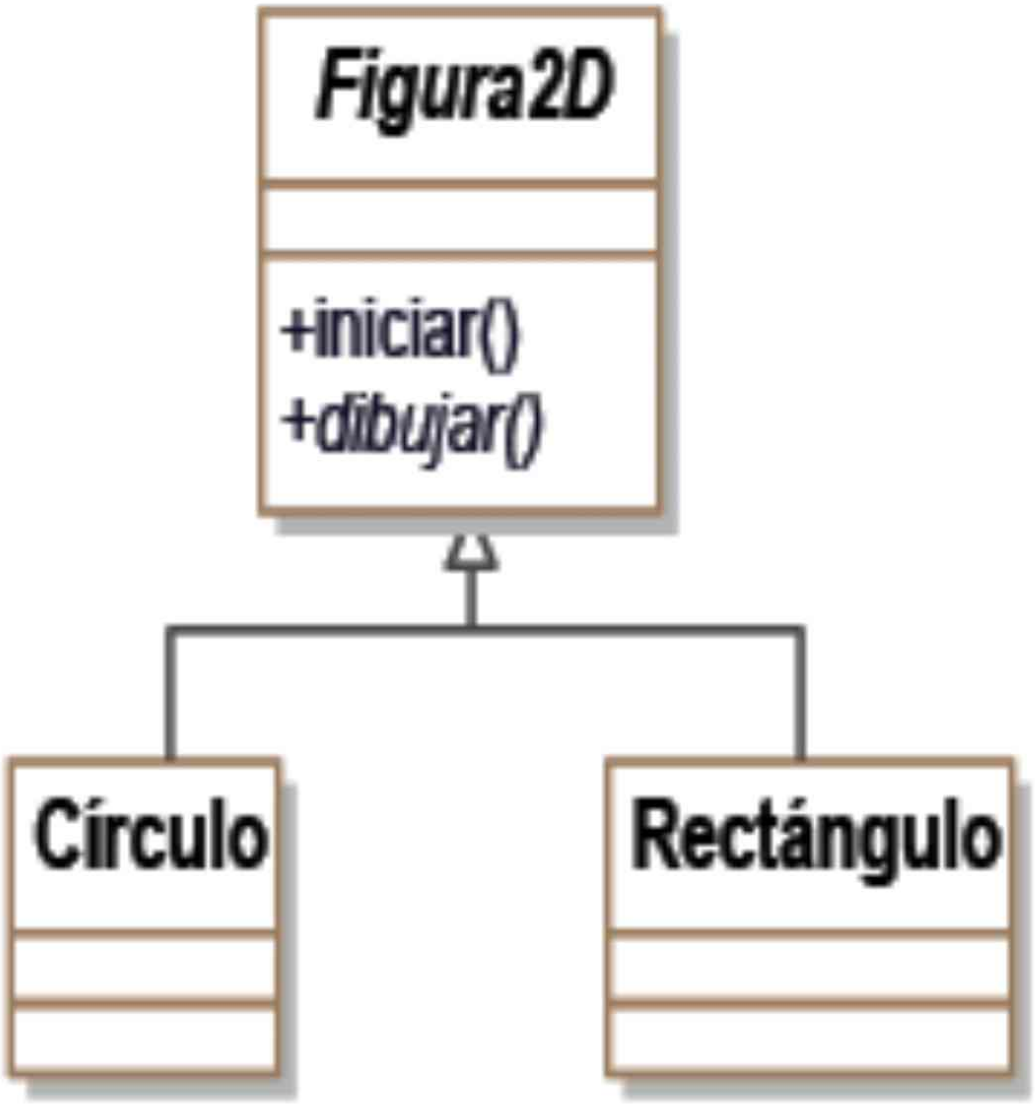


Figura 17.10. Herencia de clase abstracta

Listado 17.6 Implementación de la figura 17.10

```
import abc
# Clase abstracta
class Figura2D(metaclass = abc.ABCMeta):
def __init__(self):
pass
```

```
def iniciar(self):
print("Iniciando...")
#Metodo abstracto
@abc.abstractmethod
def dibujar(self):
pass
# Primera clase concreta
class Circulo(Figura2D):
def dibujar(self):
super().iniciar()
print("Dibujando círculo")
# Segundo clase concreta
class Rectangulo(Figura2D):
def dibujar(self):
super().iniciar()
print("Dibujando rectángulo")
figura2D = Rectangulo()
figura2D.dibujar()
```

Como puede observar, para implementar una clase abstracta en Python es necesario importar el paquete **abc** y utilizar el decorador @abc.abstractmethod con el fin de especificar el método virtual.

Clases internas o anidadas

Al igual que vimos en los dos capítulos anteriores, el lenguaje Python también ofrece facilidades para implementar clases internas o anidadas.

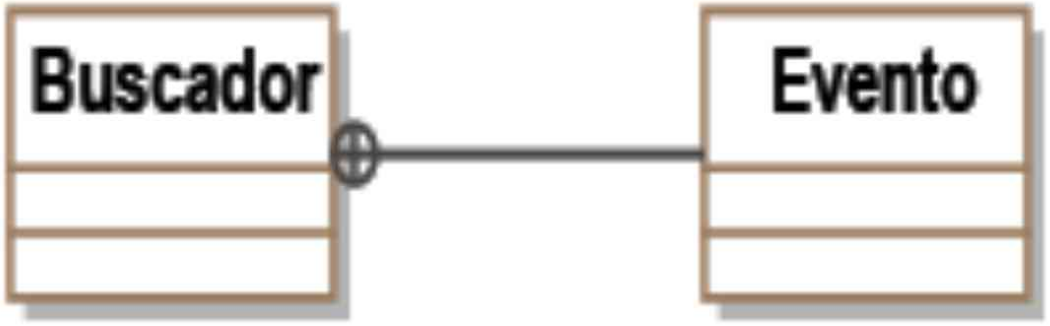


Figura 17.11. Clase interna

```
class Buscador:
class Evento:
def imprime_evento(self):
print("Generando evento...")
Buscador().Evento().imprime_evento()
```

Clases asociación

En este tipo de clases se relaciona un conjunto de objetos muchos a muchos mediante la utilización de la estructura de datos diccionario de Python.



Figura 17.12. Clase asociación

NOTA

Téngase en cuenta que las asignaciones duplicadas no se deben añadir a la lista de asignaciones en una aplicación real.

Listado 17.7 Implementación de la figura 17.12

```
# Clase ingeniero
class Ingeniero:
def __init__(self, numero):
self.__numero = numero
self.__asignaciones = {}
def set_asignacion(self,asignacion):
```

```
# Evita duplicados
if not asignacion.get_id() in self.__asignaciones:
self.__asignaciones[asignacion.get_id()] = asignacion
print(f"es true {asignacion.get_id()}")
return True
return False
def get_id(self):
return "ING" + str(self.__numero)
def __str__(self):
return "Ingeniero " + str(self.__numero)
def imprime_proyectos(self):
print(self.__str__() + " con los proyectos: ")
for i in self.__asignaciones:
print(self.__asignaciones[i].get_proyecto())
# Clase asociación
class Asignacion:
def add_relacion(self, ingeniero, proyecto, presupuesto):
# Genera clave
self.__id = ingeniero.get_id() + proyecto.get_id()
self.__ingeniero = ingeniero
self.__proyecto = proyecto
self.__presupuesto = presupuesto
if self.__ingeniero.set_asignacion(self):
self.__proyecto.set_asignacion(self)
return True
return False
def get_id(self):
return self.__id
```

```
def get_proyecto(self):
return self.__proyecto
def get_ingeniero(self):
return self.__ingeniero
# Clase proyecto
class Proyecto:
def __init__(self, numero):
self.__numero = numero
self.__asignaciones = {}
def set_asignacion(self, asignacion):
self.__asignaciones[asignacion.get_id()] = asignacion
def get_id(self):
return "PRY" + str(self.__numero)
def __str__(self):
return "Proyecto " + str(self.__numero)
def imprime_ingenieros(self):
print(self.__str__() + " con los ingenieros: ")
for i in self.__asignaciones:
print(self.__asignaciones[i].get_ingeniero())
ingenieros = [Ingeniero(0), Ingeniero(1), Ingeniero(2)]
proyectos = [Proyecto(0), Proyecto(1), Proyecto(2)]
asignaciones = [Asignacion(), Asignacion(), Asignacion()]
asignaciones[0].add_relacion(ingenieros[0], proyectos[1], 1000)
asignaciones[1].add_relacion(ingenieros[1], proyectos[0], 2000)
asignaciones[2].add_relacion(ingenieros[1], proyectos[1], 3000)
ingenieros[1].imprime_proyectos()
proyectos[1].imprime_ingenieros()
```

Imprimirá:

Ingeniero 1 con los proyectos:

Proyecto 0

Proyecto 1

Proyecto 1 con los ingenieros:

Ingeniero 0

Ingeniero 1

Asociación calificada

Python permite también la asociación calificada mediante el uso de diccionarios. Estos nos facilitan la búsqueda de un elemento mediante una clave.

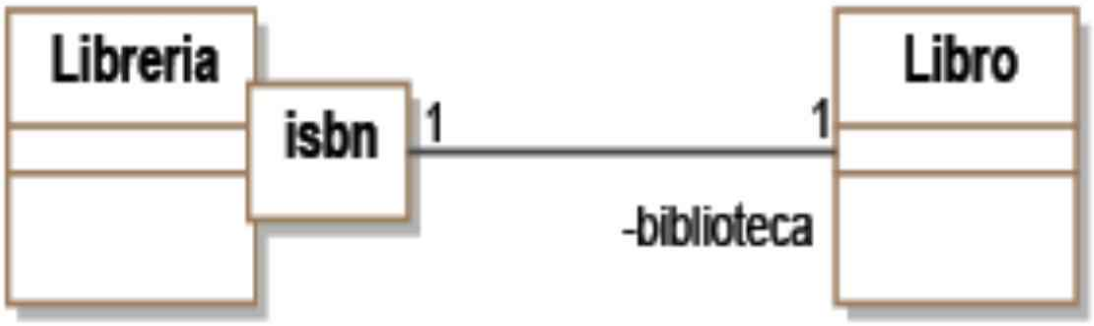


Figura 17.13. Asociación calificada

Listado 17.8 Implementación de la asociación calificada

```
# Clase libro
class Libro:
def set_datos(self, autor, titulo):
self.__autor = autor
self.__titulo = titulo
def get_autor(self):
return self.__autor
# Clase librería
class Libreria:
def __init__(self):
self.__biblioteca = {}
def add_libro(self, isbn, autor, titulo):
libro = Libro()
libro.set_datos(autor, titulo)
```

```
self.__biblioteca[isbn] = libro
def get_libro(self, isbn):
libro = self.__biblioteca[isbn]
return libro.get_autor()
libreria = Libreria()
libreria.add_libro(987654321,"Antonio Machado","Campos de Castilla")
print(f"El libro buscado es de: {libreria.get_libro(987654321)}")
```

diagramas de secuencias

Los diagramas de secuencias modelan el comportamiento dinámico de la aplicación para un determinado caso de uso. Generalmente modelan algoritmos o transacciones en el que intervienen un conjunto de actores y objetos.

Interacción básica

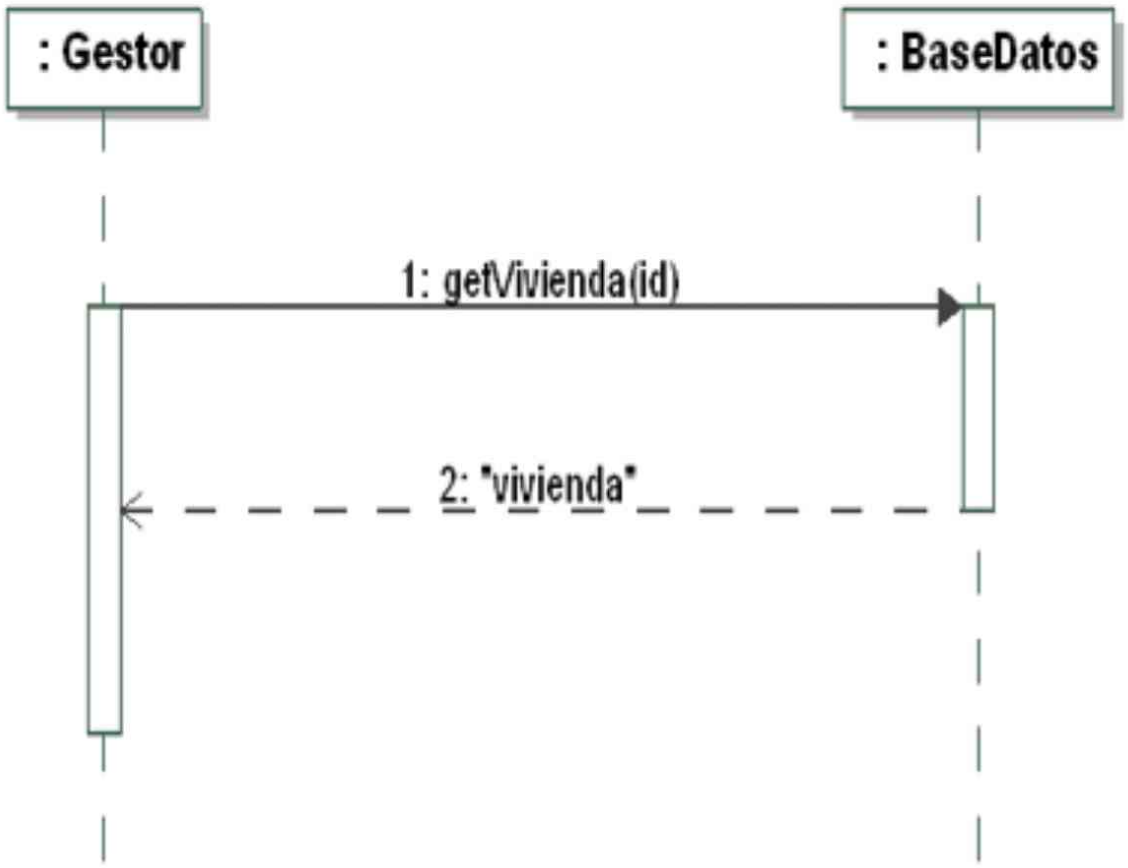


Figura 17.14. Diagrama de secuencias

El diagrama de la figura 17.14 se implementaría en Python mediante la siguiente sintáxis básica de llamada a operación de un objeto:

```
vivienda = bd.get_vivienda(7)
```

La llamada a bd.get_vivienda(7) estará ubicada en alguno de los métodos que modelan el caso de uso de la clase *Gestor*, siendo bd la variable que referencia al objeto :*BaseDatos*.

Creación, destrucción, automensajes y recursividad

Mostraremos ahora una secuencia de interacción en Python donde se aprecia la creación, los mensajes reflexivos, la recursividad y la correspondiente destrucción del objeto.

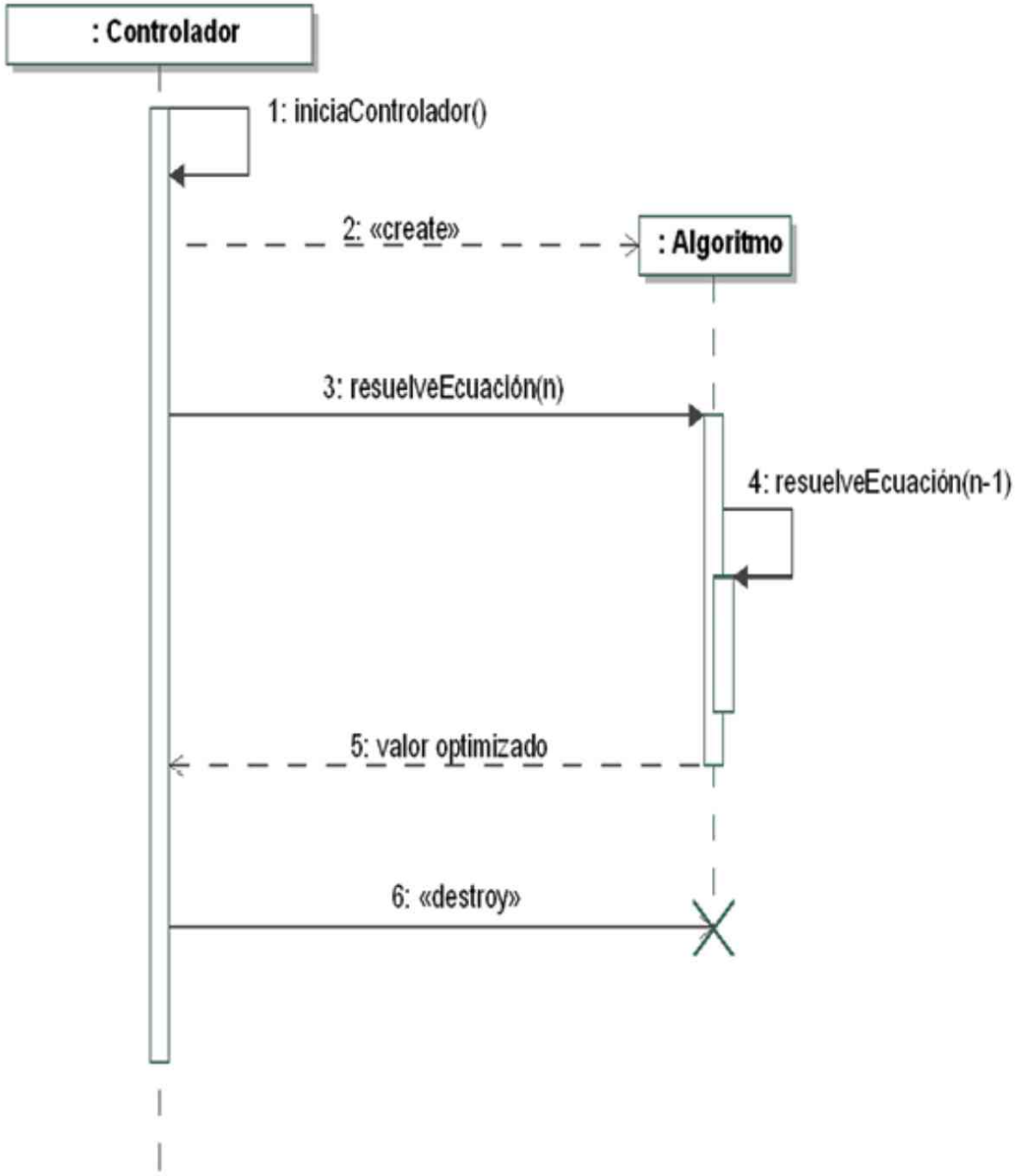


Figura 17.15. Diagrama de secuencias de un controlador que solicita una optimización

Listado 17.9 Implementación de la figura 17.15

```
# Clase algoritmo
class Algoritmo:
def resuelve_ecuacion(self, n):
if (n <= 1):
return 1
else: # Llamada recursiva
return n * self.resuelve_ecuacion(n - 1)
# Clase controlador
class Controlador:
def __init__(self):
self.__inicia_controlador()
def __inicia_controlador(self):
algoritmo = Algoritmo()
print(algoritmo.resuelve_ecuacion(8))
Controlador()
```

El listado 17.9 implementa la creación del objeto *Algoritmo* y la llamada reflexiva a *inicia_controlador()* de este objeto. Una vez creado el objeto *Algoritmo* se realiza la llamada recursiva a *resuelve_ecuacion(n)* como se ilustra en el diagrama de la figura 17.15. La destrucción del objeto *Algoritmo* es implícita y realizada por el recolector de basura de Python.

Saltos condicionales

La implementación de los saltos condicionales se modela en UML mediante los fragmentos *alt*. Este tipo de semántica también es posible implementarla en Python de una forma eficaz.

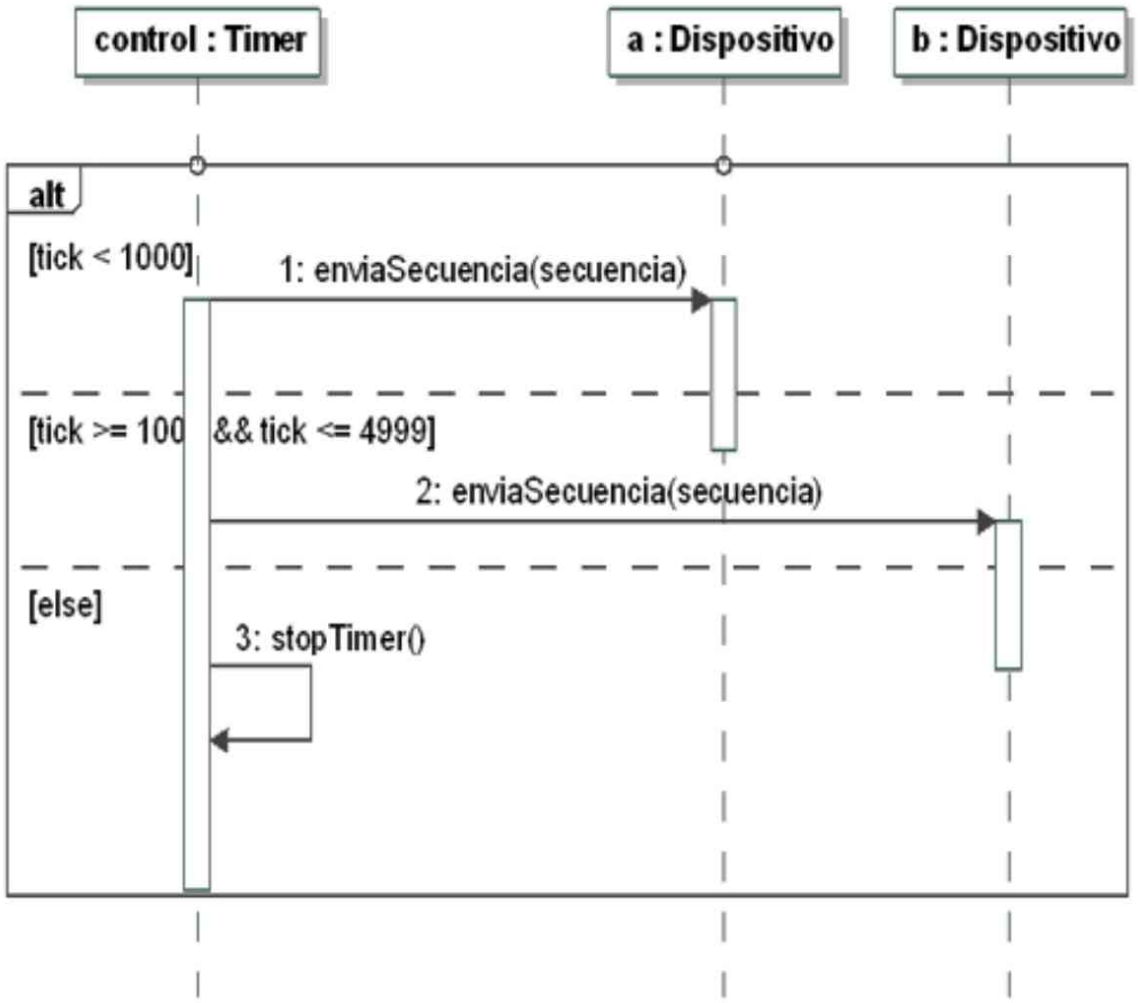


Figura 17.16. Diagrama de secuencias con fragmento condicional

Listado 17.10 Implementación de la figura 17.16

```
# Clase dispositivo
class Dispositivo:
def envia_secuencia(self, secuencia):
print(f"Enviando secuencia: {secuencia}")
```

```
# Clase Timer
class Timer:
def __init__(self):
self.__tick = 0
self.__a = Dispositivo()
self.__b = Dispositivo()
self.start_timer()
def start_timer(self):
self._on_timer()
def stop_timer(self):
print("Timer detenido")
# Región ALT UML. Evento del timer
def _on_timer(self):
self.__tick += 1
if (self.__tick < 1000):
self.__a.envia_secuencia(self.__tick)
elif (self.__tick >= 1000 and self.__tick <= 4999):
self.__b.envia_secuencia(self.__tick)
else:
self.stop_timer()
Timer()
```

En el fragmento ALT de la figura 17.16 se realiza una llamada al método *envía_secuencia()* de la clase *Dispositivo* cuando el evento del timer se encuentra dentro del rango del primer segundo y cuando se encuentra en el rango del primer al cuarto segundo posteriormente. Cuando se supera el quinto segundo se detiene el objeto *timer*.

Iteraciones

Como vimos en el capítulo siete, la manera de aplicar las iteraciones en UML era mediante un fragmento *loop*. Dichos fragmentos loop pueden adoptar diversas formas de sentencias iterativas dependiendo de si se trata de un bucle *while* o *for de* Python.

En el siguiente ejemplo se modela un escenario de consulta de una posible biblioteca. Cuando el usuario solicita al sistema una búsqueda, el objeto *MotorBusqueda* realizará una iteración sobre las tuplas devueltas anteriormente por la base de datos. En caso de encontrar una coincidencia con el título del libro buscado por el lector se devolverá un objeto *Libro*. Para realizar esta última acción se requiere el uso de la cláusula *break* de UML con la finalidad de romper el flujo iterativo (aunque en el listado 17.11 se termina implícitamente mediante *return*).

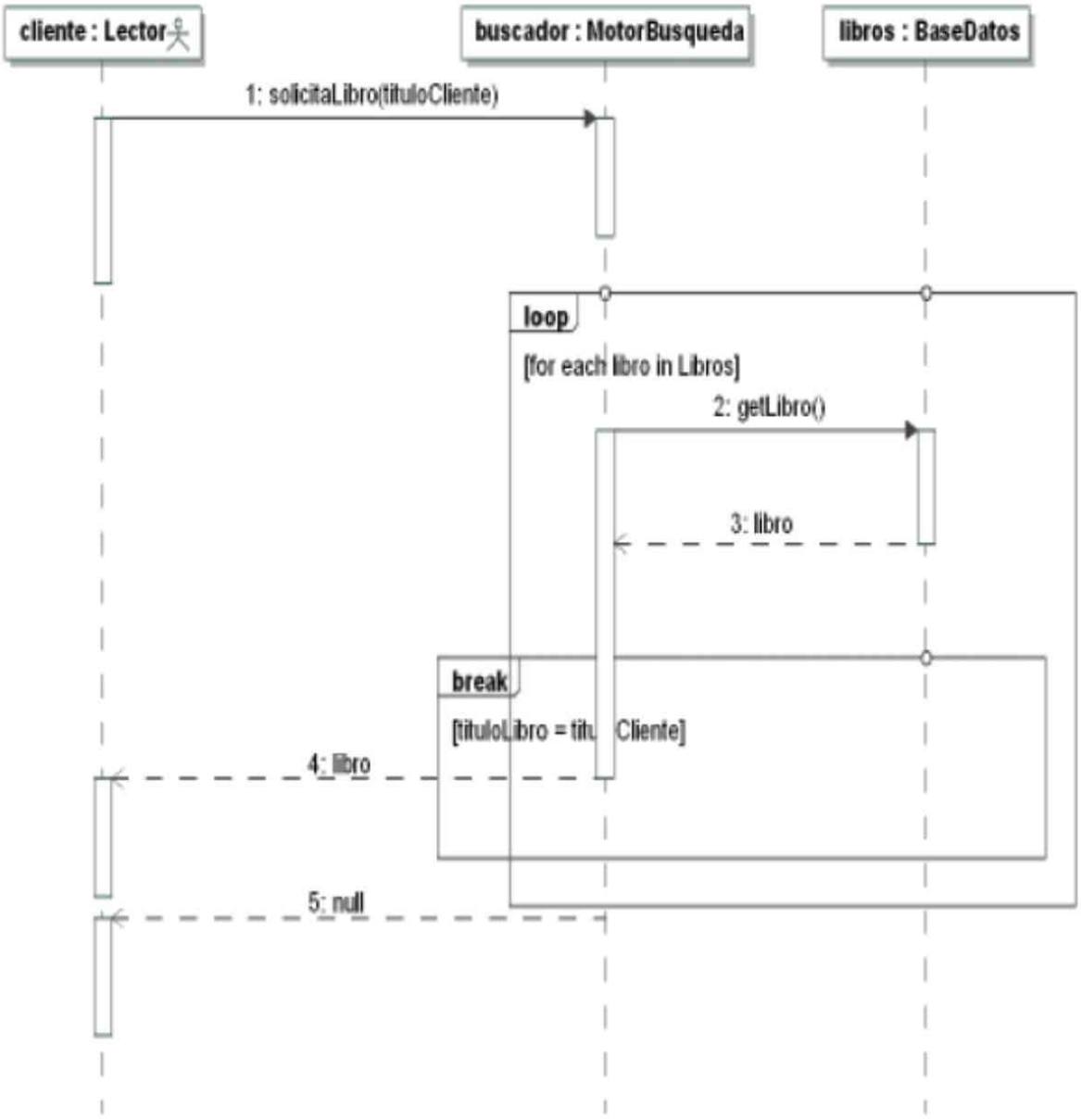


Figura 17.17. Diagrama de secuencias con fragmento iterativo

Listado 17.11 Implementación de la iteración 17.17

```
# Clase libro
class Libro:
def __init__(self, titulo):
self.__titulo = titulo
def get_titulo(self):
```

```
return self.__titulo.lower()
# Clase motor de búsqueda
class MotorBusqueda:
__registro_libros = []
def add_libro(self, libro):
MotorBusqueda.registro_libros.append(libro)
# Secuencia Loop del diagrama
def solicita_libro(self, titulo_cliente):
return [libro for libro in self.__registro_libros if (libro.get_titulo() == titulo_cliente.get_titulo())]
# Aplicación
class Inicio:
motor = MotorBusqueda()
motor.add_libro(Libro("Lazarillo de Tormes"))
motor.add_libro(Libro("Hamlet"))
libro = Libro("lazarillo de Tormes")
if (motor.solicita_libro(libro)):
print("Libro encontrado")
else:
print("Libro no encontrado")
```

diagramas de estado

De la misma forma que se explicó el funcionamiento del manejador de dispositivo en la sección 16.3, aquí vamos a reutilizar el patrón *State*, ligeramente modificado, para implementar el diagrama de estados de la figura 17.18 mediante código Python. Apréciese la utilización de la propiedad protegida *driver* con el decorador @property y el método de mutado mediante el decorador @driver.setter. Las transiciones se realizan por medio del método definido en la clase *Driver:* def cambia_estado(self, estado), el cual actualiza el estado actual y procede a pasar una referencia a la instancia de la clase de contexto a este método. Finalmente, serán las clases concretas que heredan de la clase *Estado* las que llamaran a dicho método con el objetivo de realizar las transiciones.

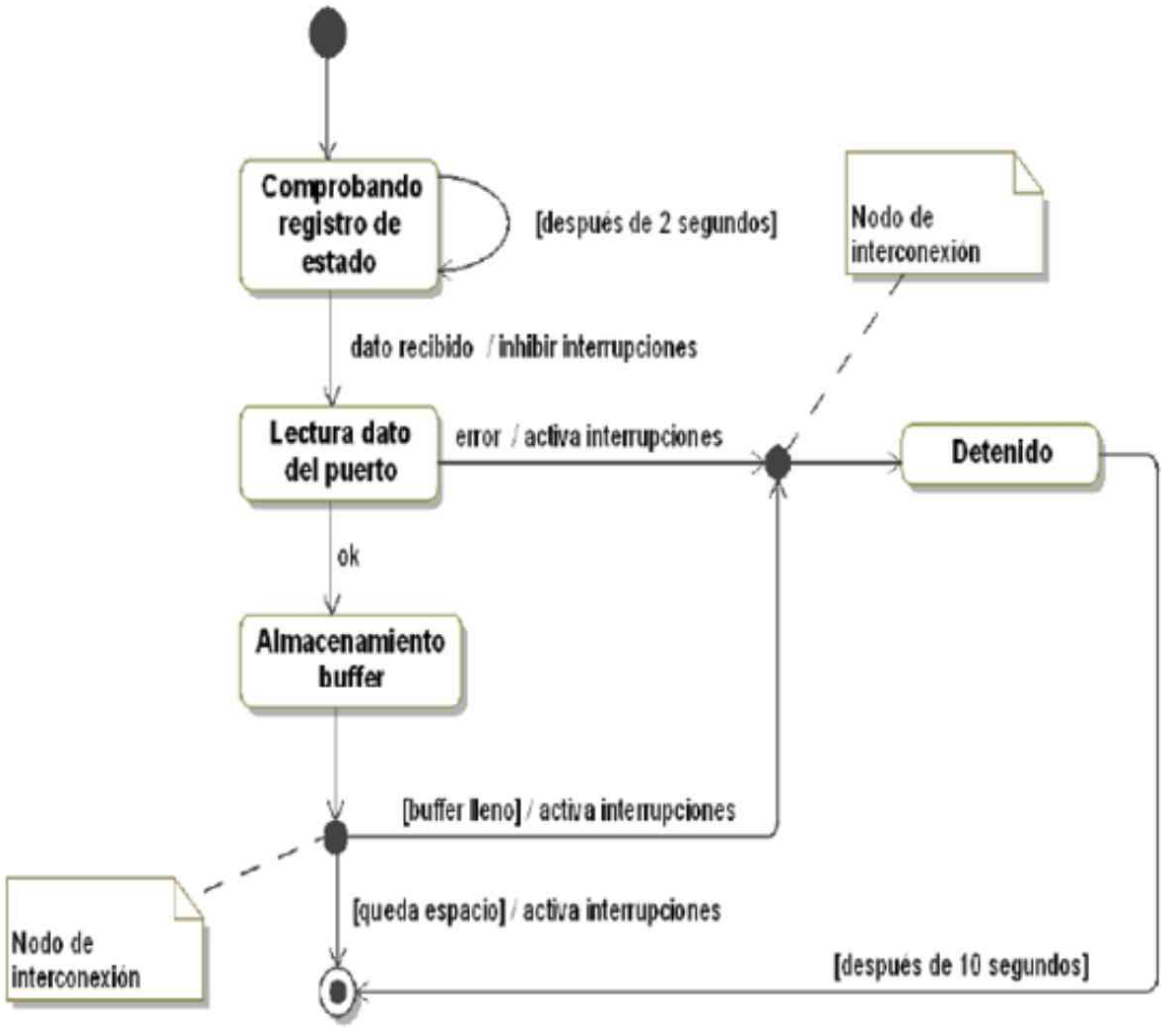


Figura 17.18. Diagrama de estado del procesamiento de un dato por un manejador de dispositivo

Listado 17.12 Implementación del diagrama de estados de la figura 17.18

```
import abc
#Clase abstracta para gestión del estado
class Estado(metaclass = abc.ABCMeta):
@property
def driver(self):
return self._driver
@driver.setter
def context(self, driver):
self._driver = driver
def iniciar(self):
self.check_error("EstInicio")
def pasan_dos_segundos(self):
self.check_error("EstComprobando")
def dato_recibido(self):
self.check_error("EstComprobando")
def error_lectura(self):
self.check_error("EstLeyendo")
def lectura_OK(self):
self.check_error("EstLeyendo")
def buffer_lleno(self):
self.check_error("EstAlmacenando")
def queda_espacio_buffer(self):
self.check_error("EstAlmacenando")
def pasan_diez_segundos(self):
```

```
self.check_error("EstDetenido")
def imprime_estado(self):
print(f"{self.nombre_estado}")
# Comprueba si en estado correcto
def check_error(self, supuesto):
if (self.nombre_estado != supuesto):
print("Error estado incorrecto")
# Estado de inicio
class EstInicio(Estado):
def __init__(self):
self.nombre_estado = "EstInicio"
def iniciar(self):
print("iniciar()")
self._driver.cambia_estado(EstComprobando())
# Estado comprobando
class EstComprobando(Estado):
def __init__(self):
self.nombre_estado = "EstComprobando"
def pasan_dos_segundos(self):
print("pasan_dos_segundos()")
self._driver.cambia_estado(EstComprobando())
def dato_recibido(self):
print("dato_recibido()")
self._driver.cambia_estado(EstLeyendo())
#Estado leyendo
class EstLeyendo(Estado):
def __init__(self):
self.nombre_estado = "EstLeyendo"
```

```
def error_lectura(self):
print("error_lectura()")
self._driver.cambia_estado(EstDetenido())
def lectura_OK(self):
print("lecturaOk()")
self._driver.cambia_estado(EstAlmacenando())
#Estado almacenando
class EstAlmacenando(Estado):
def __init__(self):
self.nombre_estado = "EstAlmacenando"
def buffer_lleno(self):
print("buffer_lleno()")
self._driver.cambia_estado(EstDetenido())
def queda_espacio_buffer(self):
print("queda_espacio_buffer()")
self._driver.cambia_estado(EstFinal())
#Estado detenido
class EstDetenido(Estado):
def __init__(self):
self.nombre_estado = "EstDetenido"
def pasan_diez_segundos(self):
print("pasan_diez_segundos()")
self._driver.cambia_estado(EstFinal())
#Estado final
class EstFinal(Estado):
def __init__(self):
self.nombre_estado = "EstFinal"
# Clase contexto
```

```
class Driver:
__estado_actual = None
def __init__(self, estado):
self.cambia_estado(estado)
def cambia_estado(self, estado):
Driver.__estado_actual = estado
Driver.__estado_actual._driver = self
def iniciar(self):
Driver.__estado_actual.iniciar()
def pasan_dos_segundos(self):
Driver.__estado_actual.pasan_dos_segundos()
def dato_recibido(self):
Driver.__estado_actual.dato_recibido()
inhibe_interrupciones()
def error_lectura(self):
Driver.__estado_actual.error_lectura()
activa_interrupciones()
def lectura_OK(self):
Driver.__estado_actual.lectura_OK()
def buffer_lleno(self):
Driver.__estado_actual.buffer_lleno()
activa_interrupciones()
def queda_espacio_buffer(self):
Driver.__estado_actual.queda_espacio_buffer()
activa_interrupciones()
def pasan_diez_segundos(self):
Driver.__estado_actual.pasan_diez_segundos()
def imprime_estado(self):
```

```
Driver.__estado_actual.imprime_estado()
```

caso de estudio: servicio de cifrado remoto

Para concluir el capítulo se presentarán los códigos fuente de la implementación en Python tanto del lado cliente como del lado servidor del servicio de cifrado remoto. Se utilizarán los diferentes diagramas desarrollados en los pasados capítulos para la presente fase del ciclo de vida.

Lado cliente

Si echamos un vistazo al diagrama de paquetes visto en el capítulo cinco, observaremos que la aplicación cliente se distribuye en los siguientes nombres de paquetes: *Menu*, *Cliente*, *Comunicaciones* y *Cifrador*. El paquete *Comunicaciones* es un paquete que será reutilizado igualmente en el lado servidor, pues su fin es el de realizar todas las funcionalidades de red vía *socket*.

Listado 17.13 Paquete Menu :: clase MenuCliente

```
from cliente.cliente import Usuario, Opcion
class MenuCliente:
# Mapa asociativo de selección
def __init__(self):
self.__menu_mapa = {
"1" : self.__simetrico,
"2" : self.__hibrido,
"3" : self.__salir,
}
# Introduce datos usuario normal
def __pedir_datos(self):
nombre = input("Introduzca su nombre: ")
login = input("login: ")
password = input("password: ")
return nombre, login, password
# Opción de cifrado simétrico
def __simetrico(self):
self.usuario.set_opcion(Opcion.SIMETRICO)
# Opción de cifrado híbrido
def __hibrido(self):
self.usuario.set_opcion(Opcion.HIBRIDO)
```

```
# Sale de la aplicación
def __salir(self):
raise SystemExit()
# Imprime menú
def __imprime_menu(self):
print(
"""
Menu Cliente Cifrador
---------------------
1. Cifrado simétrico
2. Cifrado híbrido
3. Salir
"""
)
# Muestra menu y selecciona opción
def inicio(self):
try:
opc = ""
print("Bienvenido al cliente del servicio de cifrado remoto")
print("------------------------------------------------\n")
nombre, login, password = self.__pedir_datos()
self.usuario = Usuario(nombre,login,password)
while True:
self.__imprime_menu()
opc = input("Introduzca un número de opción: ")
try:
func = self.__menu_mapa[opc]
except KeyError:
```

```
print("{} no es una opción válida".format(opc))
else:
func()
finally:
print(f"Gracias por enviar su mensaje cifrado {nombre}")
```

En el código 17.13 se implementa la clase *MenuCliente* tal cual se vió en el diagrama de clases y se solicita al usuario sus datos y la elección del algoritmo. Ahora procedemos a crear la interface *IComunicacion* requerida para la clase *FachadaComunicaciones*.

Listado 17.14 Paquete Comunicaciones :: interface IComunicacion

```
import socket
from .notificador import Notificador
# Interfaz IComunicación
class IComunicacion(metaclass = abc.ABCMeta):
@abc.abstractmethod
def conecta(self):
pass
@abc.abstractmethod
def desconecta(self):
pass
@abc.abstractmethod
def envia_msg(self, msg):
pass
@abc.abstractmethod
def envia_clave(self, clave):
pass
@abc.abstractmethod
def envia_nonce(self, nonce):
```

```
    pass
    @abc.abstractmethod
    def envia_tag(self, tag):
    pass
    @abc.abstractmethod
    def adjunta(self, observer):
    pass
    @abc.abstractmethod
    def activa_servidor(self):
    pass
```

Listado 17.15 Paquete Comunicaciones :: clase FachadaComunicaciones

```
    # Fachada de comunicaciones. Implementa IComunicación.
    class FachadaComunicaciones(IComunicacion):
    # Inicia Notificador
    def __init__(self):
    self.notificador = Notificador()
    # Conecta socket
    def conecta(self):
    # Crea el socket TCP/IP
    self.__sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    direccion_servidor = ("localhost", 10000)
    print(f"conectando a {direccion_servidor}")
    self.__sock.connect(direccion_servidor)
    # Desconecta socket
    def desconecta(self):
    print("cerrando conexion")
    self.__sock.close()
```

```
# Envía mensaje cifrado
def envia_msg(self,msg):
print ("enviando mensaje cifrado...")
self.__sock.sendall(msg)
# Envía clave
def envia_clave(self, clave):
print("enviando clave sesión cifrada...")
self.__sock.sendall(clave)
# Envía nonce
def envia_nonce(self, nonce):
print("enviando nonce...")
self.__sock.sendall(nonce)
# Envía tag
def envia_tag(self, tag):
print("enviando tag...")
self.__sock.sendall(tag)
# Adjunta observador
def adjunta(self, observer):
self.notificador.adjunta(observer)
# Activa el servidor
def activa_servidor(self):
# Crea socket TCP/IP para el servidor
self.__sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
direccion_servidor = ('localhost', 10000)
print(f"Iniciando en {direccion_servidor}")
try:
self.__sock.bind(direccion_servidor)
# Escucha conexiones entrantes
```

```
        self.__sock.listen(1)
        # Espera conexión con cliente
        print("Esperando conexión...")
        conexion, direccion_cliente = self.__sock.accept()
        print (f"Conectado a: {direccion_cliente}")
        # Recibe datos
        mensaje_cifrado = conexion.recv(1024)
        clave = conexion.recv(1024)
        nonce = conexion.recv(1024)
        tag = conexion.recv(1024)
        self.notificador.notifica(mensaje_cifrado, clave, nonce, tag)
    except:
        print("Error activando escucha en servidor (posiblemente exista otro)")
    finally:
        # Cierra el socket y la conexión
        self.__sock.close()
        print("Conexión cerrada")
```

La clase *FachadaComunicaciones* implementa la interfaz *IComunicaciones* para realizar todas las operaciones de envío, recepción y escucha TCP/IP vía socket tanto en el cliente como en el servidor.

Listado 17.16 Paquete Cliente :: interfaz IMenuCifrado y clase Usuario

```
import abc
from enum import Enum
from cifrador.cifrador import CifradorSimetrico, CifradorAsimetrico
from comunicaciones.comunicaciones import FachadaComunicaciones
# Enumeración de opciones
class Opcion(Enum):
SIMETRICO = 1
```

```
HIBRIDO = 2
# Interfaz de Fachada Menu
class IMenuCifrado(metaclass = abc.ABCMeta):
@abc.abstractmethod
def set_opcion(self, opcion):
pass
# Clase para el usuario normal
class Usuario(IMenuCifrado):
def __init__(self, nombre, login, password):
self.__simetrico = None
self.__asimetrico = None
self.__numero_accesos = 0
self.__nombre = nombre
self._login = login
self._password = password
# Instancia la Fachada de Comunicaciones
self._conexion = FachadaComunicaciones()
# Envía datos cifrados
def __envia_datos(self, msg, clave, nonce, tag):
try:
self._conexion.conecta()
except:
print("Error de conexion")
return
try:
self._conexion.envia_msg(msg)
self._conexion.envia_clave(clave)
self._conexion.envia_nonce(nonce)
```

```
self._conexion.envia_tag(tag)
except:
print("Error enviando datos por socket")
finally:
self._conexion.desconecta()
def _introduce_mensaje(self):
return input("Introduzca el mensaje a cifrar: ")
# Genera clave de 16 bytes automáticamente
def _introduce_clave(self):
clave = CifradorSimetrico.genera_clave()
print (f"Introduzca la clave de cifrado: {clave}")
return clave
# Cifra con algoritmo simétrico AES
def cifra_simetrico(self, msg, clave):
if self.__simetrico == None:
print("Creando cifrador simétrico...")
self.__simetrico = CifradorSimetrico()
self.__numero_accesos = self.__numero_accesos + 1
self.__simetrico.cifra(msg, clave)
msg, nonce, tag = self.__simetrico.get_msg()
print (f"Mensaje cifrado= {msg}")
self.__envia_datos(msg, clave, nonce, tag)
# Cifra con algoritmo híbrido AES-RSA
def cifra_hibrido(self, msg):
if self.__asimetrico == None:
print("Creando cifrador asimétrico...")
self.__asimetrico = CifradorAsimetrico()
if self.__simetrico == None:
```

```
print("Creando cifrador simétrico...")
self.__simetrico = CifradorSimetrico()
print("Por favor, espere...")
self.__asimetrico.crea_claves()
self.__clave_sesion = self.__simetrico.genera_clave()
self.__asimetrico.cifra(self.__clave_sesion)
self.clave_sesion_cifrada = self.__asimetrico.get_msg()
self.__numero_accesos = self.__numero_accesos + 1
self.__simetrico.cifra(msg,self.__clave_sesion)
msg, nonce, tag = self.__simetrico.get_msg()
print (f"Mensaje cifrado= {msg}")
self.__envia_datos(msg, self.clave_sesion_cifrada, nonce, tag)
# Define opción
def set_opcion(self, opcion):
if opcion == Opcion.SIMETRICO:
msg = self._introduce_mensaje()
clave = self._introduce_clave()
self.cifra_simetrico(msg, clave)
elif opcion == Opcion.HIBRIDO:
msg = self._introduce_mensaje()
self.cifra_hibrido(msg)
```

El paquete *Cliente* desempeña toda la lógica de la aplicación, realizando cuantas llamadas sean necesarias a los dos tipos algoritmos de cifrado según el caso solicitado por el usuario. También implementa la interfaz *IMenuCifrado* con el fin de recibir las peticiones del menú.

Listado 17.17 Paquete Cifrador :: Patrón Strategy con clases cifradoras

```
import abc
from Crypto.Random import get_random_bytes
```

```
from Crypto.Cipher import AES, PKCS1_OAEP
from Crypto.PublicKey import RSA
# Patrón Strategy. Interfaz Cifrador
class Cifrador(metaclass = abc.ABCMeta):
@abc.abstractmethod
def cifra(self, msg, clave):
pass
# Implementa cifrador simétrico
class CifradorSimetrico(Cifrador):
@staticmethod
def genera_clave():
return get_random_bytes(16)
def cifra(self, msg, clave):
cifrador = AES.new(clave, AES.MODE_EAX)
self.__nonce = cifrador.nonce
self.__msg_cifrado, self.__tag = cifrador.encrypt_and_di-
gest(msg.encode("UTF-8"))
# Devuelve datos necesarios de cifrado
def get_msg(self):
return self.__msg_cifrado, self.__nonce, self.__tag
# Implementa cifrador híbrido
class CifradorAsimetrico(Cifrador):
def cifra(self, msg, clave = None):
# Encripta la clave de sesión con la clave pública RSA
clave_publica = RSA.import_key(open("../servidor/publica.pem").read())
cifrador_rsa = PKCS1_OAEP.new(clave_publica)
self.__clave_sesion_cifrada = cifrador_rsa.encrypt(msg)
# Crea par de claves publica y privada
```

```
def crea_claves(self):
clave = RSA.generate(2048)
clave_publica = clave.publickey().exportKey()
with open("publica.pem", "wb") as f:
f.write(clave_publica)
clave_privada = clave.exportKey()
with open("privada.pem", "wb") as f:
f.write(clave_privada)
# Devuelve datos necesarios de cifrado
def get_msg(self):
return self.__clave_sesion_cifrada
```

Por último, el paquete *Cifrador* gestiona el cifrado simétrico AES y el cifrado híbrido AES-RSA solicitado por la clase *Usuario* mediante la reutilización del patrón Strategy.

Lado servidor

En el lado servidor se reutilizarán los paquetes *Comunicaciones* y el paquete *Cifrador* anteriormente implementados. Tan solo se añadirá el patrón Observer en esta sección. En total, los paquetes que cuenta el lado servidor y que se irán desarrollando a continuación son: *Menu*, *Servidor*, *Comunicaciones*, *Cifrador* y *Descifrador*.

Listado 17.18 Paquete Menu :: clase MenuServidor

```
from servidor.servidor import UsuarioAvanzado, Opcion
# Patrón Singleton y fachada de Menú
class MenuServidor:
_singleton = None
def __new__(cls, *args, **kwargs):
if not cls._singleton:
cls._singleton = super(MenuServidor, cls
).__new__(cls, *args, **kwargs)
return cls._singleton
# Mapa asociativo de selección
def __init__(self):
self.__menu_mapa = {
"1" : self.__simetrico,
"2" : self.__hibrido,
"3" : self.__espera,
"4" : self.__salir,
}
# Introduce datos para usuario avanzado
def __pedir_datos(self):
nombre = input("Introduzca su nombre: ")
login = input("login: ")
```

```
password = input("password: ")
simbolico = input("Introduzca su nombre simbólico: ")
while True:
filtrar = input("¿Desea filtrar spam en mensaje cifrado (s/n)?").lower()
if filtrar == "s" or filtrar == "n":
break
return nombre, login, password, simbolico, filtrar
# Opción cifrado simétrico
def __simetrico(self):
self.usuario.set_opcion(Opcion.SIMETRICO)
# Opción cifrado híbrido
def __hibrido(self):
self.usuario.set_opcion(Opcion.HIBRIDO)
# Pone en espera al servidor
def __espera(self):
self.usuario.espera_mensaje_cifrado()
# Sale de la aplicación
def __salir(self):
raise SystemExit()
# Imprime menú
def __imprime_menu(self):
print(
"""
Menu Servidor
---------------------
1. Cifrado simétrico
2. Cifrado híbrido
3. Poner en espera
```

```
4. Salir
"""
)
# Imprime menú y selecciona opción
def inicio(self):
try:
opc = ""
print("Bienvenido al servidor de cifrado remoto")
print("-------------------------------------------------\n")
nombre, login, password, simbolico, filtrar = self.__pedir_datos()
self.usuario = UsuarioAvanzado(nombre, login, password, simbolico, filtrar)
while True:
self.__imprime_menu()
opc = input("Introduzca un número de opción: ")
try:
func = self.__menu_mapa[opc]
except KeyError:
print("{} no es una opción válida".format(opc))
else:
func()
finally:
print(f"Gracias por enviar su mensaje cifrado {simbolico}")
```

En el código del listado 17.18 se implementa, en primer lugar, el patrón Singleton con el fin de no invocar a varias instancias de la aplicación servidor. Posteriormente se realiza la implementación del resto de la clase *MenuServidor* que se encargará de cifrar un mensaje y enviarlo por red o poner el servicio en espera para la recepción de mensajes encriptados y su posterior

descifrado automático dependiendo del tipo de algoritmo.

Listado 17.19 Paquete Comunicaciones :: Patrón Observer

```
import abc
# Patrón Observer. Clase Notificador
class Notificador:
def __init__(self):
self.__observers = []
# Adjunta observador
def adjunta(self, observer):
self.__observers.append(observer)
# Notifica a los observadores
def notifica(self, mensaje_cifrado, clave, nonce, tag):
for observer in self.__observers:
observer.notifica(mensaje_cifrado, clave, nonce, tag)
# Interfaz Observer
class IObserver(metaclass = abc.ABCMeta):
@abc.abstractmethod
def notifica(self, mensaje_cifrado, clave, nonce, tag):
pass
```

En el listado 17.19 se completa el paquete *Comunicaciones* mediante la implementación del patrón Observer. Dicho patrón servirá para redirigir los mensajes recibidos por la clase *FachadaComunicaciones* al usuario avanzado que se añadirá como observador en la lista de observadores de la clase *Notificador*.

Listado 17.20 Paquete Servidor :: interfaz IMenuCifrado y clase UsuarioAvanzado

```
import abc
from enum import Enum
from cifrador.cifrador import CifradorSimetrico, CifradorAsimetrico
```

```
from descifrador.descifrador import DescifradorSimetrico, DescifradorAsimetrico
from comunicaciones.comunicaciones import FachadaComunicaciones
from comunicaciones.notificador import IObserver
# Enumeración opciones
class Opcion(Enum):
SIMETRICO = 1
HIBRIDO = 2
# Enumeración rango del usuario avanzado
class Rango(Enum):
APRENDIZ = 1
COMPAÑERO = 2
MAESTRO = 3
# Interfaz IMenuCifrado
class IMenuCifrado(metaclass = abc.ABCMeta):
@abc.abstractmethod
def set_opcion(self, opcion):
pass
# Clase usuario normal. Implementa IMenuCifrado e IObserver
class Usuario(IMenuCifrado, IObserver):
def __init__(self, nombre, login, password):
self.__simetrico = None
self.__asimetrico = None
self.__numero_accesos = 0
self.__nombre = nombre
self._login = login
self._password = password
self._conexion = FachadaComunicaciones()
```

```
# Envía datos cifrados
def __envia_datos(self, msg, clave, nonce, tag):
try:
self._conexion.conecta()
except:
print("Error de conexion")
return
try:
self._conexion.envia_msg(msg)
self._conexion.envia_clave(clave)
self._conexion.envia_nonce(nonce)
self._conexion.envia_tag(tag)
except:
print("Error enviando datos por socket")
finally:
self._conexion.desconecta()
def _introduce_mensaje(self):
return input("Introduzca el mensaje a cifrar: ")
# Genera clave simétrica de 16 bytes AES automáticamente
def _introduce_clave(self):
clave = CifradorSimetrico.genera_clave()
print (f"Introduzca la clave de cifrado: {clave}")
return clave
# Cifra con algoritmo simétrico AES
def cifra_simetrico(self, msg, clave):
if self.__simetrico == None:
print("Creando cifrador simétrico...")
self.__simetrico = CifradorSimetrico()
```

```
self.__numero_accesos = self.__numero_accesos + 1
self.__simetrico.cifra(msg, clave)
msg, nonce, tag = self.__simetrico.get_msg()
print (f"Mensaje cifrado= {msg}")
self.__envia_datos(msg, clave, nonce, tag)
# Cifra con algoritmo híbrido AES-RSA
def cifra_hibrido(self, msg):
if self.__asimetrico == None:
print("Creando cifrador asimétrico...")
self.__asimetrico = CifradorAsimetrico()
if self.__simetrico == None:
print("Creando cifrador simétrico...")
self.__simetrico = CifradorSimetrico()
print("Por favor, espere...")
self.__clave_sesion = self.__simetrico.genera_clave()
self.__asimetrico.cifra(self.__clave_sesion)
self.clave_sesion_cifrada = self.__asimetrico.get_msg()
self.__numero_accesos = self.__numero_accesos + 1
self.__simetrico.cifra(msg,self.__clave_sesion)
msg, nonce, tag = self.__simetrico.get_msg()
print (f"Mensaje cifrado= {msg}")
self.__envia_datos(msg, self.clave_sesion_cifrada, nonce, tag)
# Define opción
def set_opcion(self, opcion):
if opcion == Opcion.SIMETRICO:
msg = self._introduce_mensaje()
clave = self._introduce_clave()
self.cifra_simetrico(msg, clave)
```

```
elif opcion == Opcion.HIBRIDO:
msg = self._introduce_mensaje()
self.cifra_hibrido(msg)
# Clase para el usuario avanzado. Hereda de usuario normal con restricciones.
class UsuarioAvanzado(Usuario):
def __init__(self, nombre, login, password, simbolico, filtrar):
super().__init__(nombre, login, password)
self.__rango = Rango.COMPAÑERO
self.__simbolico = simbolico
self.__filtrar = filtrar
self._conexion.adjunta(self) # Adjunta observador
CifradorAsimetrico().crea_claves()
# Activa servidor
def espera_mensaje_cifrado(self):
self._conexion.activa_servidor()
# Filta spam en mensaje descifrado
def _filtra_mensaje(self, mensaje_descifrado):
print("Spam filtrado")
# Parte del patrón Observer. Notificación.
def notifica(self, mensaje_cifrado, clave, nonce, tag):
print("Recibiendo mensaje...")
print("Probando estrategia simétrica AES...")
# Prueba con estrategia simétrica AES
try:
descifrador_simetrico = DescifradorSimetrico()
descifrador_simetrico.descifra(mensaje_cifrado, clave, nonce, tag)
self.mensaje_descifrado = descifrador_simetrico.get_msg()
print(f"Posible mensaje descifrado: {self.mensaje_descifrado}")
```

```
except:
print("mensaje no cifrado con algoritmo simétrico AES");
print("Probando estrategia híbrida AES-RSA...")
# Prueba con estrategia asimétrica AES-RSA
try:
descifrador_asimetrico = DescifradorAsimetrico()
descifrador_asimetrico.descifra(clave)
clave_descifrada = descifrador_asimetrico.get_msg()
descifrador_simetrico.descifra(mensaje_cifrado, clave_descifrada, nonce, tag)
self.mensaje_descifrado = descifrador_simetrico.get_msg()
print(f"Posible mensaje descifrado: {self.mensaje_descifrado}")
except:
print("mensaje no cifrado con algoritmo híbrido AES-RSA")
# Comprueba que requiere filtrado spam
if self.__filtrar == "s":
self._filtra_mensaje(self.mensaje_descifrado)
```

El listado 17.20 implementa la clase *UsuarioAvanzado* como una especialización de la clase *Usuario*. Dicha especialización se define mediante la técnica de la herencia. En general, el paquete *Servidor* realiza toda la gestión de la lógica de negocio para los usuarios, tanto si desean cifrar como descifrar. Así mismo, la clase que implementa el usuario avanzado es la encargada de implementar la interfaz *IObserver* para la recepción de mensajes procedentes de la red. También esta clase implementa la interfaz *IMenuCifrado* que recibirá las invocaciones de la fachada del menú. Como puede apreciar, el método notifica (para la notificación) se encarga de gestionar la desencriptación automática de los mensajes cifrados recurriendo al manejo de rutinas de tratamiento de excepciones.

Listado 17.21 Paquete Descifrador :: Patrón Strategy con clases Descifradoras

```
import abc
from Crypto.Cipher import AES, PKCS1_OAEP
from Crypto.PublicKey import RSA
# Patrón Strategy. Interfaz Descifrador.
class Descifrador(metaclass = abc.ABCMeta):
@abc.abstractmethod
def descifra(self, msg, clave, nonce, tag):
pass
# Implementa descrifrador simétrico
class DescifradorSimetrico(Descifrador):
def descifra(self, msg, clave, nonce, tag):
descifrador_aes = AES.new(clave, AES.MODE_EAX, nonce)
mensaje_descifrado = descifrador_aes.decrypt_and_verify(msg,tag)
self.__mensaje_descifrado = mensaje_descifrado.decode("UTF-8")
# Devuelve mensaje descifrado
def get_msg(self):
return self.__mensaje_descifrado
# Implementa descifrador asimétrico
class DescifradorAsimetrico(Descifrador):
def descifra(self, msg, clave = None, nonce = None, tag = None):
clave_privada = RSA.import_key(open("privada.pem").read())
# Desencripta la clave de sesión con la clave privada RSA
descifrador_rsa = PKCS1_OAEP.new(clave_privada)
self.__mensaje_descifrado = descifrador_rsa.decrypt(msg)
# Devuelve mensaje descifrado
def get_msg(self):
return self.__mensaje_descifrado
```

Terminamos la implementación de la aplicación del lado servidor con el listado 17.21, cuya finalidad es realizar la desencriptación de los mensajes cifrados provenientes de la red mediante la reutilización del patrón Strategy para cada tipo de algoritmo.

Anexo A

programación orientada a objetos

A.1 breve reseña histórica

Los comienzos de la Programación Orientada a Objetos se remontan al diseño del lenguaje SIMULA 67 inventado por *Krinsten Nygaard* y *Ole-Johan Dahl* en el centro de cálculo noruego de Oslo. Más tarde *Alan Kay*, que era en aquel entonces profesor de la *Universidad de Utah*, observó las capacidades de SIMULA e ideó una gama de computadoras para la visualización de gráficos para los cuales el lenguaje SIMULA se adecuaba a su programación. Posteriormente planteó esta idea a la compañía Xerox Parc a principios de 1970 donde se proyectó el ordenador Dynabook y se implementó la primera versión del lenguaje Smalltalk con el fin de venderlos conjuntamente. La idea inicial de Smalltalk era la de un sistema dinámico en donde todo eran objetos y se permitiera manipularlos mediante mensajes en tiempo de ejecución sobre una máquina virtual.

El auge de la Programación Orientada a Objetos se produjo en los años ochenta cuando *Bjarne Stroustrup* extendió la estructura del lenguaje C (creado por *Dennis Ritchie*) para crear C++, el cual poseía muchas de las características de la Programación Orientada a Objetos, pero sin llegar a ser un lenguaje puro en este aspecto. Más tarde, en los años noventa, los ingenieros de *Sun* junto a *James Gosling* crearon un lenguaje simplificado de C++ llamado *Java* orientado a la programación en Internet y el video en tiempo real.

Actualmente el éxito de la Programación Orientada a Objetos es indudable y la mayoría de los lenguajes de programación modernos como C#, PHP, Ruby, Haskell, Python, etc. incorporan orientación a objetos.

A.2 características de la poo

Enunciaremos ahora las principales características de los lenguajes orientados a objetos y en la que ya existe un claro consenso:

- La **modularidad**consiste en subdividir un programa en trozos más pequeños para conseguir un objetivo de forma más sencilla. Se debe dividir de forma que cada parte sea independiente de las demás y que tenga que interactuar lo mínimo posible con el resto, siendo prácticamente autónomo en la labor que realiza para conseguir el objetivo final del programa. A las correspondientes divisiones del programa se les denomina módulos.
- La **abstracción**consigue centrarse en las partes relevantes o esenciales de una entidad obviando las menos importantes. Consiste básicamente en definir los aspectos esenciales de un objeto del mundo real o un dominio, creando una estructura de agentes que compartan y extiendan su funcionalidad y atributos e interactúen en un sistema mediante mensajes.
- La **encapsulación** permite agrupar las partes de un programa que realizan funciones similares en un mismo conjunto abstracto.
- La **ocultación de información** permite mostrar únicamente al exterior la información necesaria y únicamente exponer al exterior sus funciones públicas (interfaz), mientras que la información que es relevante a nivel interno permanece oculta. Con esta idea en mente es posible cambiar el comportamiento interno de un objeto sin tener que cambiar el comportamiento del resto del programa cliente.
- El **polimorfismo**permite que varios objetos heterogéneos posean un comportamiento diferente. Dicho comportamiento se invoca con un determinado mensaje de llamada compartido bajo el mismo nombre.
- Con la **herencia** es posible crear una jerarquía de clases relacionadas

permitiendo que las clases superiores doten o restrinjan acceso a sus atributos a las clases hijas. Permite, entre otras cosas, la compartición de la información y el comportamiento desde las entidades superiores a sus descendientes.

- La **sobrecarga** es una característica de la Programación Orientada a Objetos que permite que funciones o métodos con comportamiento diferente compartan el mismo nombre. La sobrecarga de operadores en C++ permite añadir un comportamiento a medida a los operadores estándar del lenguaje.
- La **recolección de basura**permite no preocuparse por la destrucción de objetos ya que estos se liberan automáticamente por la máquina virtual cuando sus referencias se encuentran desasignadas o se sale del ámbito de declaración.

A.3 clases y objetos

En esta sección se tratarán las piezas básicas de diseño de la Programación Orientada a Objetos y sin las cuales no tendría mucho sentido hablar de ella. El **Objeto** es una de las partes principales de la construcción de una aplicación orientada a objetos, mientras que la **Clase** es la plantilla base para crear, instanciar o replicar dichos objetos. Una clase puede representar una entidad del mundo real, o puede significar una entidad abstracta. Lo importante es conocer la relación recíproca que existe entre la clase y el objeto, pues sin la primera no es posible la instanciación del segundo. Los objetos contienen una serie de propiedades que lo caracterizan y que pueden tener diferentes modos de acceso. Además, los objetos pueden comunicarse con otros objetos o pueden cambiar su estado interno por medio de los métodos (funciones miembro en C++). A nivel de clase se denomina a estos conceptos como atributos —cuando representan las características que tendrá el objeto— .

A.4 programación orientada a objetos en c++

A.4.1 Clases y objetos

En C++ se puede definir una clase mediante la palabra reservada **class** seguida del nombre de la clase. Para ello suponga que necesitamos crear una clase para almacenar información sobre objetos de un juego de ajedrez:

```
// Definición de la clase
class Pieza
{
private:
int color;
int altura;
int anchura;
std::string nombre;
```

Después de la definición de la clase Pieza le siguen una serie de características que definen el color, el nombre y las dimensiones de la pieza. Estas variables definidas dentro de la clase se les denominan **atributos**. Dichos atributos pueden tener diferentes modos de acceso: *private*, *protected*, *public* según el modo que queramos otorgar a la clase en cuestión. Con el modificador de acceso "**public**" permitimos que cualquier objeto acceda a este atributo, mientras que con el modificador "**private**" evitamos que otros objetos accedan al miembro, y donde únicamente los objetos creados desde su misma clase pueden acceder a él desde dentro de la clase. En el ejemplo los atributos son *private* y esto evitará que sean accedidos desde fuera de la clase[37].

Otra parte fundamental en una clase es el constructor, el cual tiene la siguiente sintaxis:

```
public:
```

```
Pieza()
{
color = 1;
altura = 10;
anchura = 5;
nombre = "Alfil1";
}
```

Los constructores se definen con el mismo nombre de la clase y generalmente se utilizan para iniciar los atributos del objeto de la clase, una vez reservada la memoria por el sistema operativo.

Otra característica de los objetos en C++ son los métodos o funciones miembro, que permiten comunicarse con el objeto y cambiar el valor interno de los atributos. En el siguiente ejemplo se muestra un método de la clase *Pieza* que permite visualizar el nombre de la misma:

```
// Método para imprimir el nombre
void get_nombre()
{
std::cout << nombre;
}
```

Los métodos pueden tener argumentos pasados por valor o por referencia, y devolver valores:

```
// Método para cambiar el estado
bool cambia_estado(int color, int altura, int anchura, std::string& nombre)
{
this->color = color;
```

```
this->altura = altura;
this->anchura = anchura;
this->nombre = nombre;
return true;
}
```

Toda función miembro debe estar antepuesta por el nombre de la clase seguida de :: si se define fuera del ámbito de la definición de la clase, aunque en el ejemplo no ha sido necesario.
Por último existen dos formas de instanciar una clase. Ésta puede ser de forma estática o dinámica. En la primera forma únicamente se necesita declarar la variable especificando su tipo de clase seguido de un nombre:

```
int main()
{
Pieza pieza;
```

En la segunda forma se escribe de forma similar pero mediante el uso de punteros y la palabra reservada **new**.

```
Pieza* pieza2 = new Pieza();
```

Sin embargo, es importante no olvidar invocar al destructor de la clase de este objeto mediante el uso de la palabra **delete**. Esto provocará que se libere la memoria del objeto reservado en una zona de memoria especial llamada *heap* y se ejecuten las sentencias declaradas dentro del destructor:

```
delete pieza2;
```

y el destructor definido como:

```
~Pieza()
{
std::cout << "Destruyendo la pieza" << std::endl;
}
```

Fíjese que el destructor viene precedido por **~**. En el caso del objeto "pieza1", declarado estáticamente, no es necesario liberarlo mediante *delete* puesto que se libera implícitamente.

Por último exponemos el código completo para la función **main** del ejemplo anterior:

Listado A.1 Construcción y destrucción en C++

```
int main()
{
Pieza pieza1; // Se instancian los objetos
Pieza* pieza2 = new Pieza();
std::cout << "Imprimiendo el nombre de la pieza 1" << std::endl;
pieza1.get_nombre(); // Llamada a los métodos
if (pieza1.cambia_estado(2,20,10,"Torre2"))
std::cout << "Pieza1 cambiada" << std::endl;
std::cout << "Imprimiendo el nombre de la pieza 1" << std::endl;
pieza1.get_nombre();
std::cout << "Imprimiendo el nombre de la pieza 2" << std::endl;
pieza2->get_nombre();
delete pieza2; // Se destruye la pieza2
return 0;
}
```

A.4.2 Herencia

Cuando hablamos de herencia nos viene a la mente la herencia genética transmitida de padres a hijos o la herencia de bienes entre familiares. Nada más lejos de la realidad, la herencia en un lenguaje de programación viene a ser algo similar. Con la característica de la herencia en C++ es posible compartir atributos y métodos de clases padre a clases hijas. A continuación se presenta un ejemplo sencillo de herencia basado en el caso visto anteriormente:

Listado A.2 Herencia en C++

```
#include <string>
#include <iostream>
class Pieza
{
private:
int color;
int altura;
int anchura;
protected:
std::string nombre;
public:
Pieza(int color, int altura, int anchura)
{
this->color = color;
this->altura = altura;
this->anchura = anchura;
}
void get_nombre()
```

```
{
std::cout << nombre << std::endl;
}
};
class Peon : public Pieza
{
public:
Peon(int color, int altura, int anchura):Pieza(color, altura, anchura)
{
nombre = "Peon7";
}
void mueve_escaque(int x, int y)
{
std::cout << "Moviendo a posicion: " << x << " " << y << std::endl;
}
};
int main()
{
Peon peon(3, 10, 5); // Se instancia el objeto
peon.get_nombre();
peon.mueve_escaque(7,3);
return 0;
}
```

La nueva clase *Peón* hereda las características y el comportamiento definidos en la clase base *Pieza*. Cuando esto ocurre decimos que la clase *Peón* deriva de la clase *Pieza* y hereda el atributo nombre declarado como **protected**. Según el modificador de acceso que especifiquemos a la hora de derivar la clase podemos encontrarnos con los siguientes casos:

Herencia pública (class Peon : public Pieza)		
Modificador en base	**Acceso desde clase derivada**	**Acceso público (a Peón)**
Public	SI	SI
Private	NO	NO
Protected	SI	NO

Tabla A.1 Herencia pública

Herencia protegida (class Peon : protected Pieza)		
Modificador en base	**Acceso desde clase derivada**	**Acceso público (a Peón)**
Public	SI	NO
Private	NO	NO
Protected	SI	NO

Tabla A.2 Herencia protegida

Herencia privada (class Peon : private Pieza)		
Modificador en base	**Acceso desde clase derivada**	**Acceso público (a Peón)**
Public	SI	NO
Private	NO	NO
Protected	SI	NO

Tabla A.3 Herencia privada

Para completar la explicación de este ejemplo, fíjese que en el constructor de la clase derivada (*Peón*) hemos realizado una llamada al constructor de la clase base. Esto es así con la finalidad de poder pasar parámetros desde el constructor de la clase derivada a los atributos de la clase base.

En C++ existe la posibilidad de realizar una herencia múltiple en la que una

clase hija puede derivar de varias clases padre, al modo siguiente:

```
class A : public B, private C, protected D
{
(...)
```

No obstante, para entender mejor esta idea suponga el siguiente código de ejemplo:

Listado A.3 Herencia múltiple en C++

```
// Clase para realizar estadísticas
class Estadistica
{
protected:
int movimientos;
public:
Estadistica()
{
movimientos = 0;
}
void imprime_estadisticas()
{
std::cout << "Total movimientos: " << movimientos << std::endl;
}
};
// Hereda de la nueva clase
class Peon : public Pieza, public Estadistica
{
public:
```

```
Peon(int color, int altura, int anchura):Pieza(color, altura, anchura)
{
nombre = "Peon7";
}
void mueve_escaque(int x, int y)
{
std::cout << "Moviendo a posicion: " << x << " " << y << std::endl;
movimientos++;
}
};
```

Se ha creado una nueva clase llamada *Estadistica* que es adaptada por herencia en la clase *Peón*. Según se vio en la tabla A.1, los atributos y los métodos de la clase *Estadistica* son transmitidos a la clase hija, la cual es la responsable de operar con sus atributos y funciones miembro. Para concluir el ejemplo añadimos aquí el código completo de la función *main*:

```
int main()
{
Peon peon(3, 10, 5); // Se instancia el objeto
peon.get_nombre(); // Método heredado de Pieza
peon.mueve_escaque(7,3); // Método propio
peon.imprime_estadisticas(); // Heredado de Estadística
return 0;
}
```

A.4.3 Polimorfismo

Como la misma etimología de la palabra indica, el polimorfismo ocurre cuando una cosa puede adoptar múltiples formas. En el caso de la Programación Orientada a Objetos el polimorfismo tiene lugar cuando una función miembro posee el mismo nombre e impone un comportamiento diferente a cada objeto. Esta situación ocurre cuando objetos diferentes, con diferentes contenidos y operaciones, comparten un mismo comportamiento basándose en las características comunes de las funciones virtuales en la clase o clases base. Para entender la idea de polimorfismo es necesario tener clara la noción de función *virtual* que proporciona la *ligadura dinámica*. Por medio de las funciones virtuales podemos especificar que un determinado método se invocará según el tipo de objeto al que se refiera el apuntador. Las funciones virtuales se añaden como punteros en las llamadas *tablas virtuales* o *vtable* en inglés. Normalmente un compilador crea una tabla virtual separada por cada clase. Estos punteros nos indicarán a qué bloque de código de función debemos llamar cuando se haga referencia a la *vtable*. Cuando se crea un objeto de clase se crea un puntero a dicha tabla (VPTR). Este puntero es un miembro oculto dentro de la clase. Cuando se llama al constructor de la clase se inicia el valor de dicho puntero para apuntar a la correspondiente tabla virtual. Para ilustrar esta idea suponga el siguiente código de ejemplo:

```
class Base // Clase abstracta
{
public:
int dato_base;
virtual void mi_funcion() = 0;
};
class Derivada : public Base
{
```

```
public:
int dato_derivada;
void mi_funcion()
{
std::cout << "Llamada sobrecargada";
}
};
```

Tal como se muestra en el siguiente código, al invocar una función virtual de un objeto de la clase derivada a través de un puntero de la clase base ocurre lo siguiente:

```
int main()
{
Derivada derivada;
Base* base = &derivada;
base->mi_funcion();
return 0;
}
```

Imprimirá:

```
Llamada sobrecargada
```

No obstante, para entender mejor los conceptos explicados anteriormente se expone a continuación un ejemplo básico de polimorfismo en el que se crea una clase base para Pieza con dos métodos virtuales: uno para mover entre escaques y otro para imprimir información interna de la clase. Por último, se heredan tres clases de Pieza: *Peón, Caballo* y *Torre*. Estas tres clases implementarán los métodos virtuales citados anteriormente y dotarán de

comportamiento al resto del programa.

Supongamos la siguiente clase base para *Pieza*:

Listado A.4 Polimorfismo en C++

```
#include <string>
#include <iostream>
class Pieza
{
private:
int color;
int altura;
int anchura;
protected:
std::string nombre;
int x,y;
public:
Pieza(int color, int altura, int anchura)
{
this->color = color;
this->altura = altura;
this->anchura = anchura;
}
void get_nombre()
{
std::cout << nombre << std::endl;
}
// Mueve pieza a nueva posición
virtual void mueve_escaque()=0;
```

```
// Imprime los datos de la pieza
virtual void imprime_datos()
{
std::cout << "Color: " << color << std::endl;
std::cout << "Altura: " << altura << std::endl;
std::cout << "Anchura: " << anchura << std::endl;
}
};
```

Posteriormente se crean las tres clases que heredarán de Pieza y sobrescribirán los métodos virtuales:

```
class Peon : public Pieza
{
private:
bool amenaza_peon; // Si amenaza a otra pieza
public:
Peon(int color, int altura, int anchura):Pieza(color, altura, anchura)
{
nombre = "Peon7";
x = 7;
y = 2;
amenaza_peon = false;
}
void mueve_escaque()
{
// Movimiento del peón
y++;
amenaza_peon = true;
```

```
}
void imprime_datos()
{
std::cout << "¿Se encuentra amenazando?: " << (amenaza_peon ?
"SI" : "NO") << std::endl;
Pieza::imprime_datos();
}
};
class Torre : public Pieza
{
private:
bool amenaza_torre; // Si amenaza a otra pieza
public:
Torre(int color, int altura, int anchura):Pieza(color, altura, anchura)
{
nombre = "Torre1";
x = 1;
y = 1;
amenaza_torre = false;
}
void mueve_escaque()
{
// Movimiento de la torre
y+= 7;
amenaza_torre = true;
}
void imprime_datos()
{
```

```
std::cout << "¿Se encuentra amenazando?: " << (amenaza_torre ?
"SI" : "NO") << std::endl;
Pieza::imprime_datos();
}
};
class Caballo : public Pieza
{
private:
bool amenaza_caballo; // Si amenaza a otra pieza
public:
Caballo(int color, int altura, int anchura):Pieza(color, altura, anchura)
{
nombre = "Caballo1";
x = 2;
y = 1;
amenaza_caballo = false;
}
void mueve_escaque()
{
// Movimiento del caballo
x++;
y+= 2;
amenaza_caballo = true;
}
void imprime_datos()
{
std::cout << "¿Se encuentra amenazando?: " << (amenaza_caballo ?
"SI" : "NO") << std::endl;
```

```
Pieza::imprime_datos();
}
};
```

Si nos fijamos en una de ellas, por ejemplo en la clase *Torre*, es posible distinguir los dos métodos virtuales sobrecargados: *mueve_escaque* e *imprime_datos*. Estos ya no se han especificado con la palabra reservada *virtual* pues es aquí donde se sobrecargan los métodos virtuales definidos en la clase *Pieza*. Para comprender definitivamente el funcionamiento del polimorfismo suponga el siguiente código de ejemplo en la función de entrada *main*:

Listado A.5 Polimorfismo en C++ (continuación)

```
int main()
{
Peon peon(1, 10, 5);
Torre torre(2, 20, 5);
Caballo caballo(3, 10, 10);
Pieza* pieza = &torre;
// Llamada a mueve_escaque de Torre
pieza->mueve_escaque();
// Llamada a imprime_datos de Torre y Pieza
pieza->imprime_datos();
// Conversión
Torre* torre = dynamic_cast<Torre*>(pieza);
torre->mueve_escaque();
torre->imprime_datos();
return 0;
}
```

Primero declaramos tres objetos de las clases *Peon*, *Torre* y *Caballo*, iniciándolos con valores previamente establecidos de color, altura y anchura. Cuando realizamos la llamada a Pieza* pieza = &torre estamos tomando un puntero de la clase base sobre un objeto de la clase derivada. Esto es totalmente factible en Programación Orientada a Objetos. Con dicho puntero es ahora posible invocar al método virtual del objeto derivado. En el caso del ejemplo la primera llamada a *mueve_escaque* hará que se invoque el método en *Torre*, puesto que la función virtual de su tabla está establecida como *Torre::mueve_escaque*. En el segundo caso se invocará a *Torre::imprime_datos* pero con una retrollamada a la función virtual de la clase base. En este caso es una solución mixta a la llamada a la función virtual, pues aunque la entrada en la tabla de métodos virtuales apunta a *imprime_datos* la llamada a *Pieza-::imprime_datos* hará que se invoque a su homóloga en la clase base.
Si compilamos y ejecutamos el ejemplo, obtendremos la siguiente salida para el ejemplo anterior:

```
¿Se encuentra amenazando la torre?: SI
Color: 2
Altura: 20
Anchura: 5
```

Otro aspecto de la Programación Orientada a Objetos es la posibilidad de conversión entre punteros de objetos, también conocida en el argot como *cast* o *retipado*. En el caso del ejemplo que nos atañe se ha realizado una conversión del puntero de la clase base *Pieza*, que en realidad apunta a un objeto derivado, a un puntero a un objeto de la clase derivada *Torre*. Fíjese que la llamada a los métodos virtuales con dicho puntero produce la misma salida por pantalla.

¿Se encuentra amenazando la torre?: SI

Color: 2

Altura: 20

Anchura: 5

A.5 Programación orientada a objetos en java

A.5.1 Clases y objetos

Las clases y los objetos en Java siguen las mismas ideas conceptuales que en C++. La clase es la parte nuclear en la Programación Orientada a Objetos y permite agrupar los datos (atributos) y funciones (métodos) que operan sobre esos datos dentro de una estructura abstracta y encapsulada. Por otro lado, el objeto es la pieza básica de construcción de las aplicaciones orientadas a objetos y es el resultado de la instanciación de la clase. En Java las clases se definen de forma muy similar a C++, siguiendo el siguiente patrón:

```
[public] class NombreClase {
// Atributos y métodos
}
```

El especificador **public** es opcional y si no se escribe la clase, tiene visibilidad por defecto (**package**), y puede ser vista por el resto de las clases del paquete.
Una característica importante de los objetos en Java es que todos heredan de la clase **Object**, que es una clase especial que permite realizar operaciones como la copia, conversiones, identificación, etc.
En Java se pueden definir varias clases dentro de un fichero *.java*, pero sólo una puede ser "*public*" y debe nombrarse como el fichero. De no ser así, el compilador generaría una excepción sin llegar a generar el *bytecode*.
Para entender mejor la idea de clase en Java se muestra aquí un ejemplo de definición de la misma:

Listado A.6 Clases y objetos en Java

```
package ejemplo1;
// Definición de la clase Pieza
class Pieza
{
```

```
private int color;
private int altura;
private int anchura;
protected String nombre;
public Pieza()
{
color = 1;
altura = 10;
anchura = 5;
nombre = "Alfil1";
}
// Métodos
public boolean cambiaEstado(int color, int altura, int anchura, String nombre)
{
this.color = color;
this.altura = altura;
this.anchura = anchura;
this.nombre = nombre;
return true;
}
public void getNombre()
{
System.out.println(nombre);
}
}
```

Como puede apreciarse en el código anterior, prácticamente no hay muchas diferencias con C++ en lo que respecta a la definición de la clase. En la

primera línea se define el paquete al que pertenece el fichero principal del mismo. Se definen varios atributos de forma similar a como se realizó en C++ y se especifica el constructor donde de inician los atributos. Después se definen dos métodos (*cambiaEstado* y *getNombre*) para modificar el estado de la pieza y obtener su nombre respectivamente.

En la arquitectura de máquina virtual de Java no existe el concepto de destructor como sucedía en C++. En su lugar, el sistema de recolección de basura se encargará de liberar la memoria asignada dinámicamente mediante la cláusula *new*. Por este mismo motivo en el método *cambiaEstado* no ha sido necesario definir el operador de acceso a un componente de puntero de clase (->), en su lugar únicamente ha sido necesario utilizar el operador punto para acceder a los atributos color, altura, anchura y nombre. De esta forma, la creación y la llamada a los métodos se realiza de la siguiente manera:

Listado A.7 Clases y objetos en Java (continuación)

```
public class Ejemplo1
{
public static void main(String[] args)
{
// Se instancian los objetos
Pieza pieza = new Pieza();
System.out.println("Imprimiendo el nombre de la pieza");
pieza.getNombre(); // Llamada a los métodos
if (pieza.cambiaEstado(2,20,10,"Torre2"))
System.out.println("Pieza cambiada");
pieza.getNombre();
}
}
```

En el código anterior se muestra la clase *Ejemplo1*, que es la clase que identifica al fichero. La parte importante está en el punto de entrada a la aplicación. En ésta se define el método estático *main* que es generado de manera estática en memoria, sin necesidad de reserva dinámica, y es compartido por todas las clases. Éste será el comienzo de ejecución de la aplicación y donde se crea el objeto *Pieza*, mediante el operador *new*. Ya no es necesario el uso de punteros en esta sentencia del lenguaje, tal como se ha comentado anteriormente, por lo que no utilizaremos el operador de indirección de C++. Por último, las llamadas a los métodos se realizan escribiendo el identificador del objeto, seguido del operador punto y finalizando con el nombre del método y sus argumentos.

A.5.2 Paquetes

Los paquetes en Java son unas unidades lógicas asociadas a directorios del sistema y que permiten la agrupación de clases, la ordenación y la evitación de conflictos de nombres. Es posible la importación de un paquete en un programa Java mediante el uso de la directiva *import*. De esta forma, pueden importarse clases como:

```
import java.util.Vector
```

Aquí se importa la clase contenedora de utilidad Vector.
Si queremos que un fichero en el que hemos definido varias clases pertenezca a un determinado paquete, debemos especificarlo con la palabra reservada *package*.

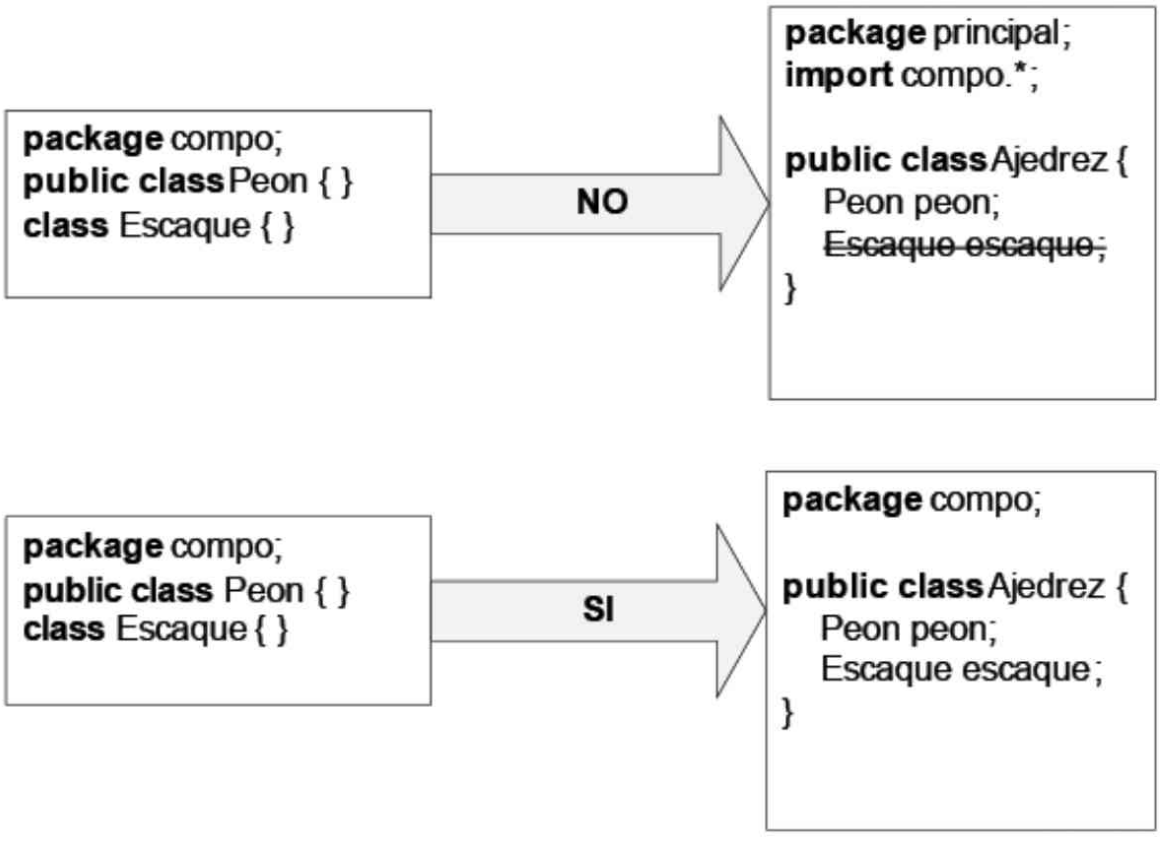


Figura A.1. Ejemplo de importación con paquetes

Es importante agrupar todas las clases que pertenecen al mismo paquete

dentro del mismo directorio, así podrán ser accedidas entre ellas. Cuando se invoca a un paquete mediante la directiva *import* realmente a lo que se está haciendo referencia es al fichero *.class* donde se encuentran las clases compiladas.

A.5.3 Herencia

El concepto de herencia no difiere con respecto al de C++. Cuando una clase hija hereda de otra en Java incorpora todas las características de la clase padre. La diferencia más substancial en cuanto a C++ son los especificadores de acceso de "*extends*", pues Java carece de restricciones en cuanto al modo de herencia que siempre es del tipo *public*.
Para definir que una clase hereda de otra en Java se utiliza la sintaxis:

```
class Hija extends Padre
```

En Java no está permitida la herencia múltiple como sí ocurría en C++, por lo que una clase únicamente puede heredar de una sola clase padre. En la sección siguiente veremos que esto sí es posible a nivel de interfaz.
A modo de ejemplo se muestra aquí el código descrito en la sección A.4.2, pero en versión Java:

Listado A.8 Herencia en Java

```
package ejemplo2;
class Pieza
{
private int color;
private int altura;
private int anchura;
protected String nombre;
public Pieza(int color, int altura, int anchura)
{
this.color = color;
this.altura = altura;
this.anchura = anchura;
```

```
}
public void getNombre()
{
System.out.println(nombre);
}
};
// Realización de la herencia
class Peon extends Pieza
{
public Peon(int color, int altura, int anchura)
{
super(color, altura, anchura);
nombre = "Peon7";
}
public void mueveEscaque(int x, int y)
{
System.out.println("Moviendo a posicion: "+ x + " " + y);
}
};
public class Ejemplo2
{
public static void main(String[] args)
{
// Se instancia el objeto
Peon peon = new Peon(3, 10, 5);
peon.getNombre();
peon.mueveEscaque(7,3);
}
```

```
}
```

Como se puede apreciar en el ejemplo, la clase *Peón* hereda de *Pieza*, incorporando todos sus atributos y métodos. Por otro lado, en el constructor de la clase Peón se ha realizado la transferencia de parámetros de atributos a la clase padre mediante el uso de la palabra reservada **super**. Esta sentencia es utilizada tanto en constructores como en métodos convencionales para invocar al método inmediatamente superior de cuya clase hereda.

A.5.4 Interfaces

Una característica de Java y de la que carece C++ es del concepto de *Interfaz*. Esta útil herramienta de la Programación Orientada Objetos se utiliza para que el sistema o la jerarquía de clases se adapte a unos patrones establecidos por las interfaces, las cuales especifican las características de los métodos que deben heredar las clases hijas (entre otros usos). Los métodos de las interfaces carecen de definición. Tampoco tienen constructor y definen sus atributos constantes, públicos y estáticos de forma automática, por lo que pueden omitirse estos modificadores de acceso.

La forma de heredar una interfaz es mediante la palabra reservada **implements** de la siguiente forma:

```
public class Peon extends Pieza implements Grafico
{
(...)
```

y en el fichero *Grafico.java* tendremos:

```
import javax.swing.*;
public interface Grafico
{
public void dibuja(int x, int y, Graphics g);
public void anima(Graphics g);
}
```

Implementando la interfaz Grafico en *Peon* podemos aplicar una serie de comportamientos comunes a cada una de las piezas del ajedrez. La clase *Peon* está obligada a implementar los métodos de *Grafico*.

A.5.5 Polimorfismo

Aunque las clases abstractas y las interfaces no son necesarias para aplicar el polimorfismo en Java, es importante tener claro su utilización. Estas clases se caracterizan porque permiten definir funciones miembros abstractas que no tienen implementación e imponen un comportamiento predefinido a un conjunto de clases heterogéneas heredadas. Cuando una clase define un método como **abstract** está obligada a definir la clase como **abstract**. Obviamente y dada su naturaleza, las clases abstractas no se pueden instanciar para crear un objeto de ella.

La sintaxis de las clases abstractas no difiere mucho en cuanto al modo en que se realizan las interfaces. En el siguiente ejemplo se muestra el uso de una clase abstracta para implementar la característica de polimorfismo en el juego del ajedrez:

Listado A.9 Polimorfismo en Java

```
package ejemplo3;
abstract class Pieza
{
private int color;
private int altura;
private int anchura;
protected String nombre;
protected int x,y;
public Pieza(int color, int altura, int anchura)
{
this.color = color;
this.altura = altura;
this.anchura = anchura;
```

```
}
public void getNombre()
{
System.out.println(nombre);
}
// Mueve pieza a nueva posición
public abstract void mueveEscaque();
// Imprime los datos de la pieza
public void imprimeDatos()
{
System.out.println("Color: " + color);
System.out.println("Altura: " + altura);
System.out.println("Anchura: " + anchura);
}
};
```

Aquí se define la clase abstracta *Pieza*, que será la que establecerá el comportamiento y las características que heredarán cada una de las figuras hijas. El hecho de crear el método *mueveEscaque* implica un comportamiento común en cada clase heredada para moverla de una posición a otra del tablero. Como se puede apreciar, las clases abstractas proporcionan la capacidad de declarar métodos con código junto con métodos abstractos que carecen del mismo. La diferencia fundamental con respecto a las interfaces es la posibilidad de combinar métodos con código dispuesto para la herencia en compañía de miembros abstractos que permiten aplicar un comportamiento a las clases heredadas.

A continuación se aplicará la herencia a cada una de las figuras concretas.

Listado A.10 Polimorfismo en Java (continuación)

```
class Peon extends Pieza
{
private boolean amenazaPeon; // Si amenaza a otra pieza
public Peon(int color, int altura, int anchura)
{
super(color, altura, anchura);
nombre = "Peon7";
x = 7;
y = 2;
amenazaPeon = false;
}
public void mueveEscaque()
{
// Movimiento del peon
y++;
amenazaPeon = true;
}
public void imprimeDatos()
{
System.out.println("¿Se encuentra amenazando el peon?:
" + (amenazaPeon ? "SI" : "NO"));
super.imprimeDatos();
}
};
class Torre extends Pieza
{
private boolean amenazaTorre; // Si amenaza a otra pieza
public Torre(int color, int altura, int anchura)
```

```
{
super(color, altura, anchura);
nombre = “Torre1”;
x = 1;
y = 1;
amenazaTorre = false;
}
public void mueveEscaque()
{
// Movimiento de la torre
y+= 7;
amenazaTorre = true;
}
public void imprimeDatos()
{
System.out.println(“¿Se encuentra amenazando la torre?” + (amenazaTorre
?
“SI” : “NO”));
super.imprimeDatos();
}
};
class Caballo extends Pieza
{
boolean amenazaCaballo; // Si amenaza a otra pieza
public Caballo(int color, int altura, int anchura)
{
super(color, altura, anchura);
nombre = “Caballo1”;
```

```
x = 2;
y = 1;
amenazaCaballo = false;
}
public void mueveEscaque()
{ // Movimiento del caballo
x++;
y+= 2;
amenazaCaballo = true;
}
public void imprimeDatos()
{
System.out.println("¿Se encuentra amenazando el caballo?:
" + (amenazaCaballo ? "SI" : "NO"));
super.imprimeDatos();
}
};
```

En el código anterior se definen los constructores y se concretan los métodos abstractos donde se realiza cada uno de los movimientos dependiendo de si es peón, torre o caballo. Fíjese que en el método *imprimeDatos* no ha sido necesario definirlo virtual como se hizo en C++; Java incorpora este comportamiento por defecto, invocando primero a la clase derivada y posteriormente con el uso de **super** a la clase base, como se muestra a continuación:

Listado A.11 Polimorfismo en Java (continuación)

```
public class Ejemplo3
{
```

```
public static void main(String[] args)
{
Peon peon = new Peon(1, 10, 5);
Torre torre = new Torre(2, 20, 5);
Caballo caballo = new Caballo(3, 10, 10);
Pieza pieza = torre;
// Llamada a mueveEscaque de Torre
pieza.mueveEscaque();
// Llamada a imprimeDatos de Torre y Pieza
pieza.imprimeDatos();
Torre torre = (Torre)pieza; // Conversión
torre.mueveEscaque();
torre.imprimeDatos();
}
}
```

En el método *main* del ejemplo anterior se comienza instanciando los tres tipos de piezas que conformarán el programa, iniciándolos con sus respectivos valores. Luego se asigna una referencia del objeto torre al puntero del tipo *Pieza*. Éste será el encargado de realizar las llamadas a los métodos sobrecargados *mueveEscaque* e *imprimeDatos*. Por último se realiza una conversión (retipado) desde el puntero del tipo *Pieza* a un puntero del tipo *Torre* y se llama a sus respectivos métodos.

A.6 Programación orientada a objetos en python

A.6.1 Clases y objetos

La clase es el modelo conceptual de una entidad del mundo real y puede ser una idea concreta como por ejemplo, un coche, o abstracta, como la empatía entre dos personas. Por medio de la clase podemos establecer una plantilla con la que crear objetos, es decir, podemos instanciar dicha clase. La clase es la parte más importante del modelo estructural o estático y por tanto pieza fundamental de la Programación Orientada a Objetos. Python permite definir clases de una forma muy sencilla y reducida, siguiendo la siguiente sintaxis:

Listado A.12 Sintaxis clase, atributos y métodos en Python

```
class NombreClase:
atributo_estatico = None
def __init__(self):
self.atributo_publico = 1
self._atributo_protegido = 2
self.__atributo_privado = 3
NombreClase.atributo_estatico = 0
def metodo_publico(self, parametro_formal):
pass
def _metodo_protegido(self, parametro_formal):
pass
def __metodo_privado(self, parametro_formal):
pass
class Heredada(NombreClase):
def acceso(self):
print(self.atributo_publico)
print(self._atributo_protegido)
```

```
objeto = NombreClase()
objeto.metodo_publico(None)
Heredada().acceso()
```

Es importante recordar que en Python el anidamiento de sentencias se consigue mediante la tabulación o el sangrado del código. La sintaxis de una clase en Python puede apreciarse en el listado A.12. El constructor de la misma se define mediante la palabra clave __init__ , mientras que el modificador de acceso *protegido* es mediante un único guión bajo: _ y el acceso *privado* con dos guiones bajos: __ , tanto para los atributos como para los métodos.

Un atributo estático o de clase se define fuera del constructor y los métodos, tal como se aprecia en el listado A.12. La forma que tiene Pyhton de agregar a la clase atributos miembro es mediante la palabra reservada self, seguida de un punto y el atributo correspondiente con el modo de acceso. Todos los métodos pertenecientes a la clase deben contener la palabra reservada self al principio de la lista de parámetros formales.

En el listado A.12 se ha incluido un ejemplo de herencia simple en la clase *Heredada* con respecto a *NombreClase*. Trataremos de forma más precisa la herencia en los siguientes apartados. De momento solo es importante fijarse en el método público *acceso* para comprobar el nivel de visibilidad de los atributos heredados.

En el siguiente listado se presenta un ejemplo real de clase en Python para un juego de ajedrez.

Listado A.13 Sintaxis clase, atributos y métodos en Python

```
# Definición de la clase Pieza
class Pieza:
```

```
    # Constructor
    def __init__(self):
    self.color = 1
    self.altura = 10
    self.anchura = 5
    self._nombre = "Alfil"
    # Método público
    def cambia_estado(self, color, altura, anchura, nombre):
    self.color = color
    self.altura = altura
    self.anchura = anchura
    self._nombre = nombre
    return True
    #Métodos selectores y mutadores
    @property
    def nombre(self):
    return self._nombre
    @nombre.setter
    def nombre(self, nombre):
    self._nombre = nombre
    pieza = Pieza()
    pieza.cambia_estado(7, 10, 20, "Torre")
    pieza.nombre = "Caballo"
    print(pieza.nombre)
```

El listado A.13 es un ejemplo de clase real para una pieza del juego de ajedrez. Se han establecido métodos selectores y mutadores mediante el decorador built-in de Python @property. Finalmente, se ha instanciado la clase *Pieza,* se ha llamado a su método *cambia_estado* y se ha procedido a cambiar el

atributo *nombre* usando la función property con el fin de crear métodos *setter* y *getter*.

Como en Java, la destrucción del objeto del tipo *Pieza* se realiza de forma automática por el recolector de basura del intérprete de Python. Una vez que un objeto pierde las referencias de otros objetos hacia él, éste es liberado automáticamente de memoria dinámica.

A.6.2 Paquetes

En Python, a los ficheros de código fuente con extensión .py se les denomina módulos, mientras que a los directorios conteniendo a un conjunto de estos se les denomina paquetes. Los diagramas de paquetes que vimos en el capítulo cinco son un ejemplo teórico de los paquetes aquí utilizados.

De cara a utilizar paquetes en Python, debemos asegurarnos de que existe un fichero (normalmente vacío) con el nombre __init__.py en el directorio del paquete. De no existir dicho fichero no se podrá realizar la importación del paquete.

Suponga el siguiente árbol de directorios:

```
directorio_principal/
menu.py
tienda/
__init__.py
cliente.py
productos.py
basedatos/
__init__.py
gestor.py
driver.py
```

En cada subdirectorio se ha ubicado un fichero __init__.py con el fin de proceder a la importación. En Python existen varias formas de importar un módulo, función o clase. Por ejemplo:

Importación absoluta desde cualquier módulo:

```
import tienda.productos
producto = tienda.productos.Producto()
```

o también como:

```
from tienda.productos import Producto
producto = Producto()
```

o también:

```
from tienda import productos
producto = productos.Producto()
```

Importación relativa:

Si nos encontráramos dentro del paquete tienda.basedatos, podríamos importar la siguiente clase:

```
from ..productos import Producto()
```

A.6.3 Herencia

La herencia en Python sigue el mismo concepto que en C++ y Java. Con este mecanismo de Programación Orientada a Objetos podemos reutilizar el código de otra clase y evitar la duplicación del mismo. Además, nos facilita una herramienta versátil para la ampliación y la mantenibilidad del código fuente. Para definir la herencia en Python entre dos clases se utiliza la siguiente sintaxis:

```
class Hija(Padre):
```

En Python está permitida la herencia múltiple como sucede igualmente en C++. La sintaxis para realizarla es muy similar a la herencia simple:

```
class Hija(Padre1, Padre2, ...,Padre_n):
```

A continuación se propone un ejemplo real que ilustra la utilización de la herencia simple:

Listado A.14 Sintaxis clase, atributos y métodos en Python

```
# Definición de la clase Pieza
class Pieza:
# Constructor
def __init__(self, color, altura, anchura):
self.color = color
self.altura = altura
self.anchura = anchura
#Métodos selectores y mutadores
@property
def nombre(self):
```

```
return self._nombre
@nombre.setter
def nombre(self, nombre):
self._nombre = nombre
class Peon(Pieza):
def __init__(self, color, altura, anchura):
super().__init__(color, altura, anchura)
self._nombre = "Peon7"
def mueve_escaque(self, x, y):
print(f"Moviendo a posición: {x},{y}")
peon = Peon(3,10,5)
print(peon.nombre)
peon.mueve_escaque(7,3)
```

En el listado A.14 se ejemplifica el uso de la herencia simple de la subclase *Peon* con respecto a la superclase *Pieza*. La clase *Peon* hereda los atributos públicos y protegidos definidos en la clase padre y permite la construcción del objeto *Peon* mediante la llamada al constructor de la superclase usando la palabra reservada super() con los parámetros requeridos: *color*, *altura* y *anchura*.

A.6.4 Interfaces

Como ocurría en C++, las interfaces en Python deben de ser definidas mediante clases abstractas con métodos sin cuerpo. Para ello es necesario importar el módulo ABC, el cual contiene una serie de facilidades para tal propósito.
La sintaxis para realizar una interfaz en Python es la siguiente:

```
import abc
class Interfaz(metaclass = abc.ABCMeta):
@abc.abstractmethod
def método_1(self, parametro1, ..., parametro_n):
pass
@abc.abstractmethod
def método_2(self, parametro1, ..., parametro_n):
pass
```

Finalmente, para realizar la implementación, únicamente es necesaria la utilzación de la herencia múltiple o simple y la reescritura de los métodos sin cuerpo definidos previamente mediante el decorador @abc.abstractmethod

```
class Concreta(Interfaz):
def método_1(self, parametro1, ..., parametro_n):
(...) # implementación
def método_2(self, parametro1, ..., parametro_n):
(...) # implementación
```

A.6.5 Polimorfismo

En un sentido literal, polimorfismo significa la capacidad de un objeto de adquirir diferentes formas. En Python, el polimorfismo nos permite imponer un comportamiento diferente a cada objeto gobernándolos mediante un mismo método en la clase padre que es replicado con diferente cuerpo en las clases hijas con el fin de dirigir dicho comportamiento.

Proponemos el mismo ejemplo aplicado a Java y C++ en el contexto de un juego de ajedrez con tres tipos de piezas: Peón, Torre y Caballo con los que hay que operar.

Listado A.15 Ejemplo de polimorfismo en Python

```
import abc
# Definición de la clase Pieza
class Pieza(metaclass = abc.ABCMeta):
# Constructor
def __init__(self, color, altura, anchura):
self.__color = color
self.__altura = altura
self.__anchura = anchura
@abc.abstractmethod
def mueve_escaque(self):
pass
def imprime_datos(self):
print(f"Color: {self.__color}")
print(f"Altura: {self.__altura}")
print(f"Anchura: {self.__anchura}")
#Métodos selectores y mutadores
@property
```

```
def nombre(self):
return self._nombre
@nombre.setter
def nombre(self, nombre):
self._nombre = nombre
# Clase Peon
class Peon(Pieza):
def __init__(self, color, altura, anchura):
super().__init__(color, altura, anchura)
self._nombre = "Peon7"
self.x = 7
self.y = 2
self.__amenaza_peon = False
# Método sobrecargado
def mueve_escaque(self):
self.y += 1
self.__amenaza_peon = True
# Método sobrecargado
def imprime_datos(self):
print(f"¿Se encuentra amenazando el peón? {'SI' if self.__amenaza_peon
else 'NO'}")
super().imprime_datos()
# Clase Torre
class Torre(Pieza):
def __init__(self, color, altura, anchura):
super().__init__(color, altura, anchura)
self._nombre = "Torre1"
self.x = 1
```

```
self.y = 1
self.__amenaza_torre = False
# Método sobrecargado
def mueve_escaque(self):
self.y += 7
self.__amenaza_torre = True
# Método sobrecargado
def imprime_datos(self):
print(f"¿Se encuentra amenazando la torre? {'SI' if self.__amenaza_torre else 'NO'}")
super().imprime_datos()
# Clase Caballo
class Caballo(Pieza):
def __init__(self, color, altura, anchura):
super().__init__(color, altura, anchura)
self._nombre = "Caballo1"
self.x = 2
self.y = 1
self.__amenaza_caballo = False
# Método sobrecargado
def mueve_escaque(self):
self.x += 1
self.y += 2
self.__amenaza_caballo = True
# Método sobrecargado
def imprime_datos(self):
print(f"¿Se encuentra amenazando el caballo? {'SI' if self.__amenaza_caballo else 'NO'}")
```

```
    super().imprime_datos()
peon = Peon(1,10,5)
torre = Torre(2, 20, 5)
caballo = Caballo(3, 10, 10)
lista_piezas = [peon, torre, caballo]
# Realiza polimorfismo
for pieza in lista_piezas:
pieza.mueve_escaque()
pieza.imprime_datos()
```

Se ha optado por la utilización de una clase abstracta *Pieza* para la realización del polimorfismo en este ejemplo. Si bien no son necesarias las clases abstractas y las interfaces para el polimorfismo, en el ejemplo anterior se facilita la implementación del comportamiento deseado.

Si se fija en las tres últimas líneas del listado A.15, podrá comprobar la utilización del polimorfismo para invocar a un conjunto de piezas con un mismo método definido en la clase padre abstracta *Pieza*, lo que impone un comportamiento diferente a cada tipo de objeto hijo durante las llamadas de la iteración for.

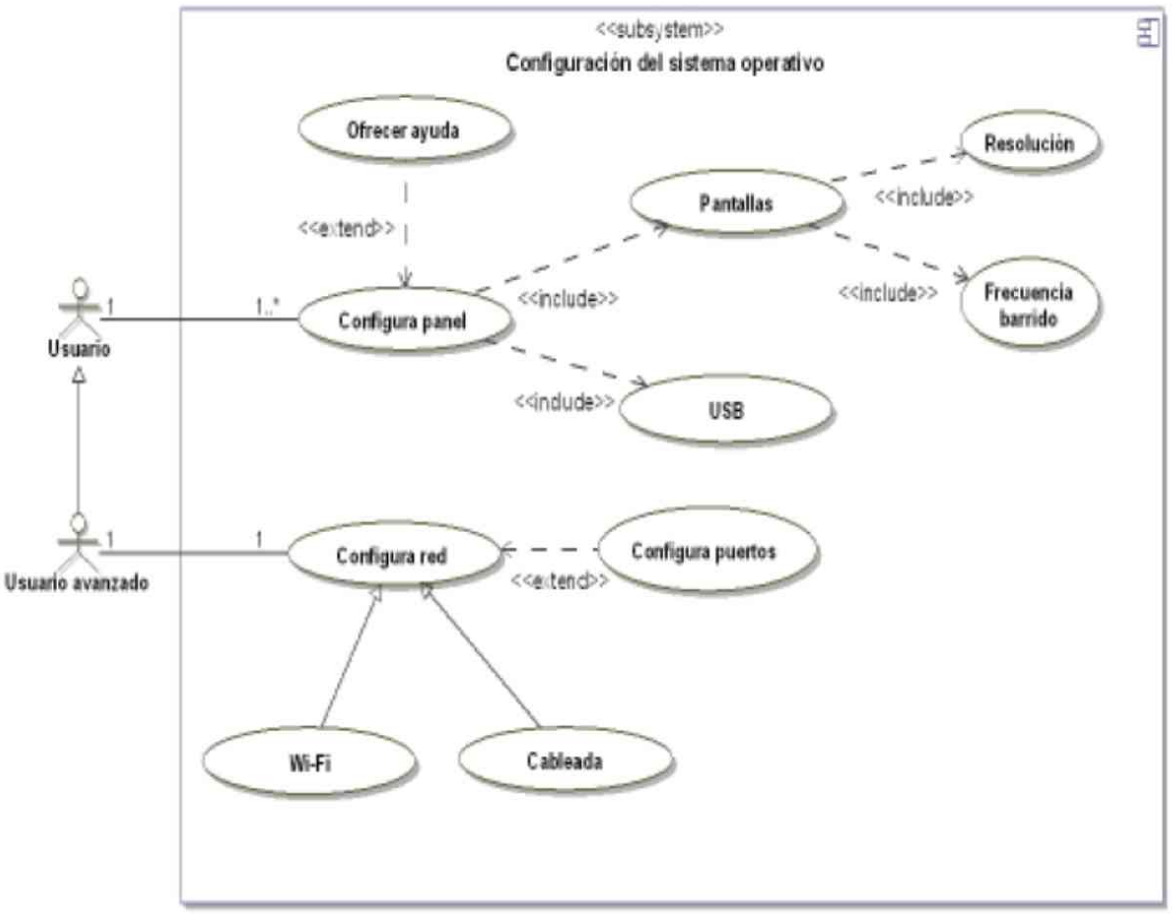
---

37. El modificador de acceso "protected" se explicará en más detalle en la sección de la herencia.

Anexo B

resumen de notación uml

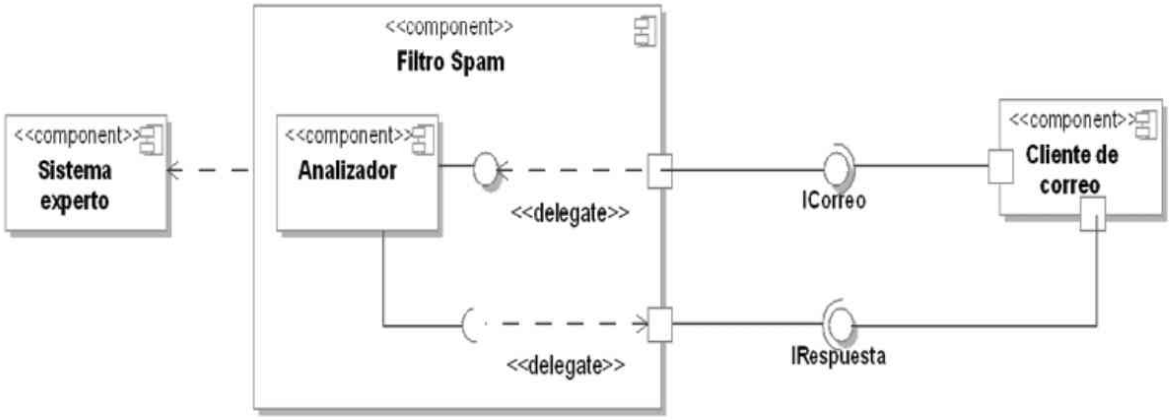
B.1 diagrama de casos de uso



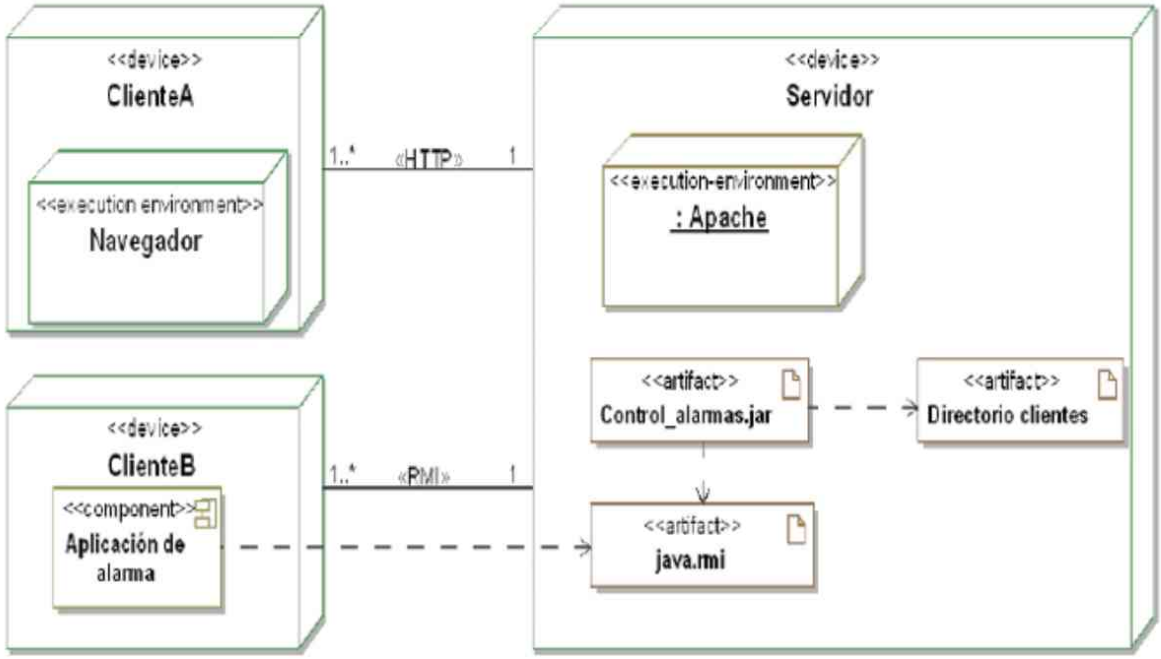
B.2 diagrama de robustez



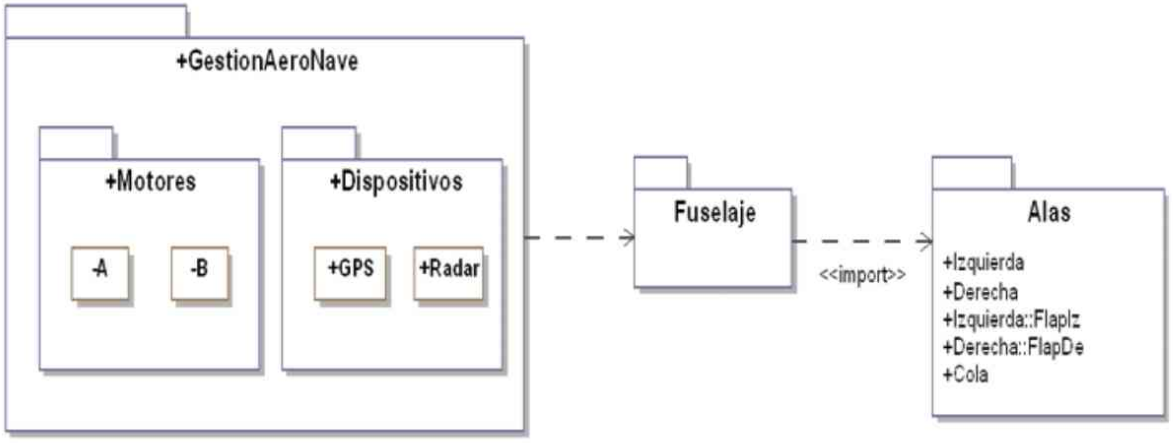
B.3 diagrama de componentes



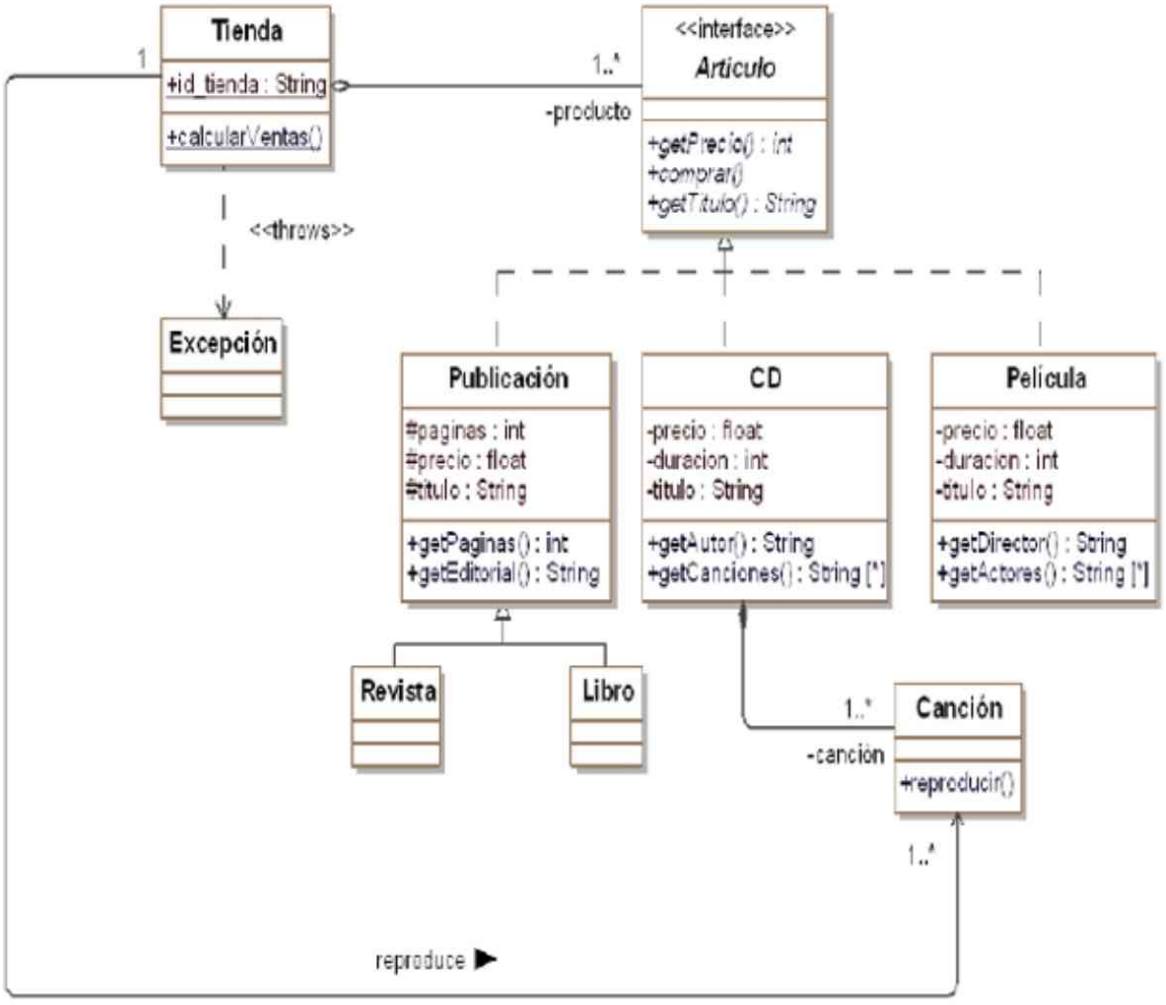
B.4 diagrama de despliegue



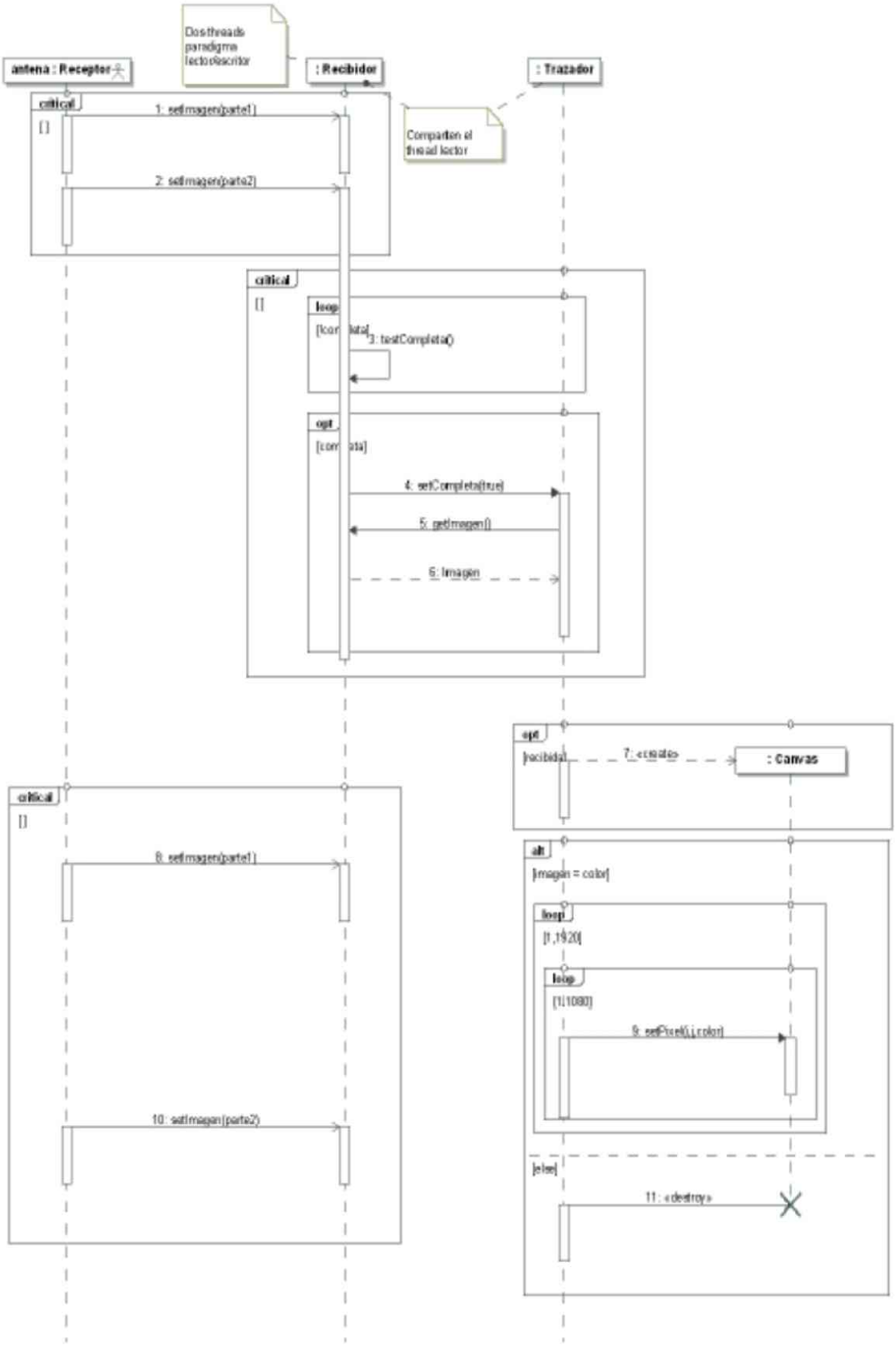
B.5 diagrama de paquetes



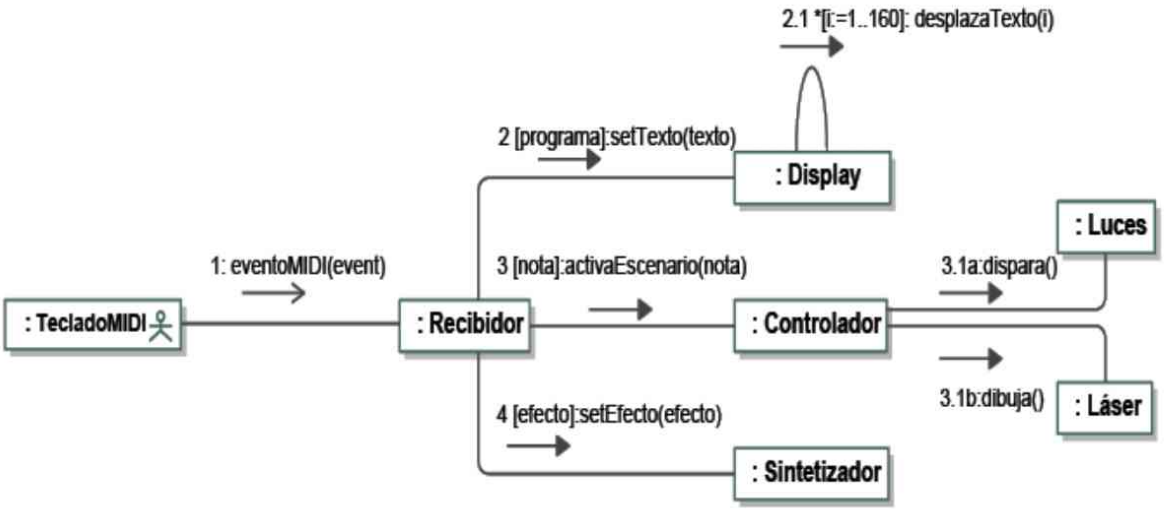
B.6 diagrama de clases



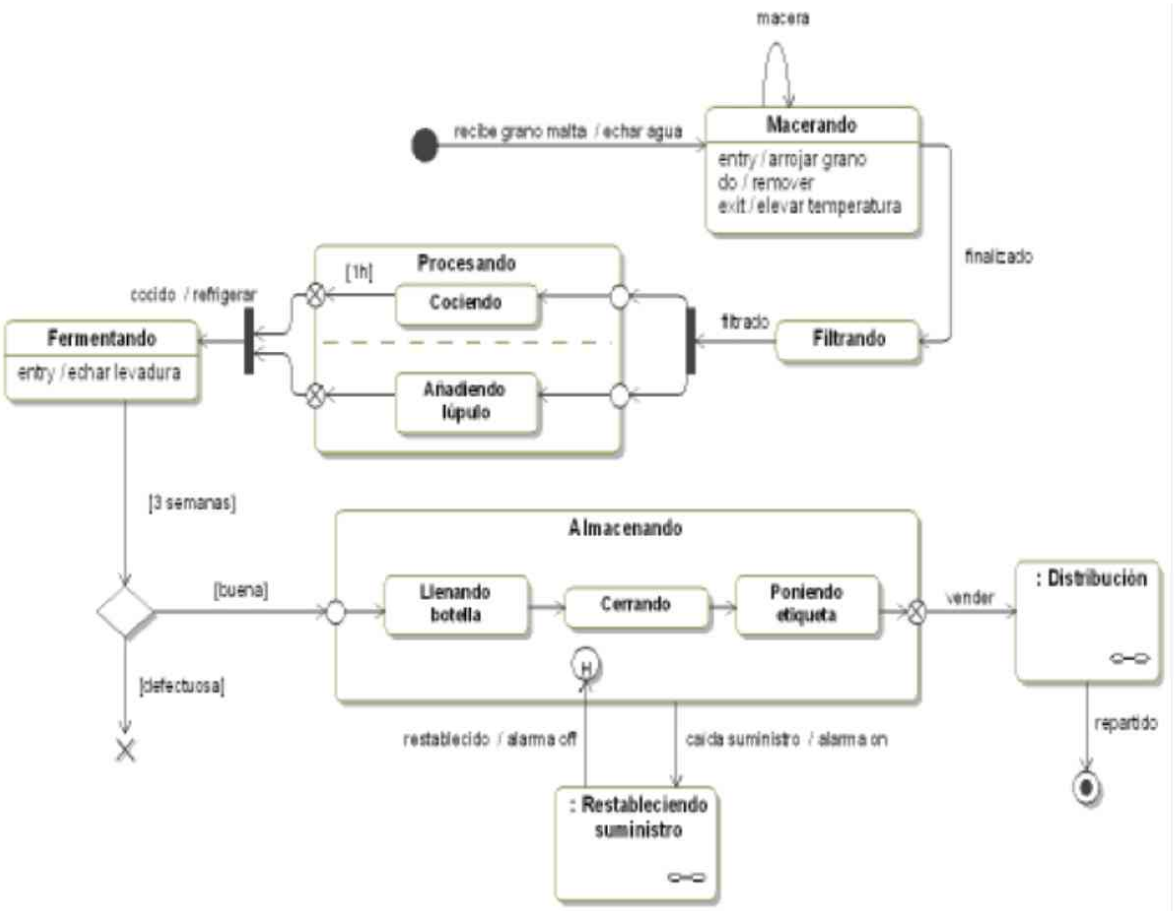
B.7 diagrama de secuencias



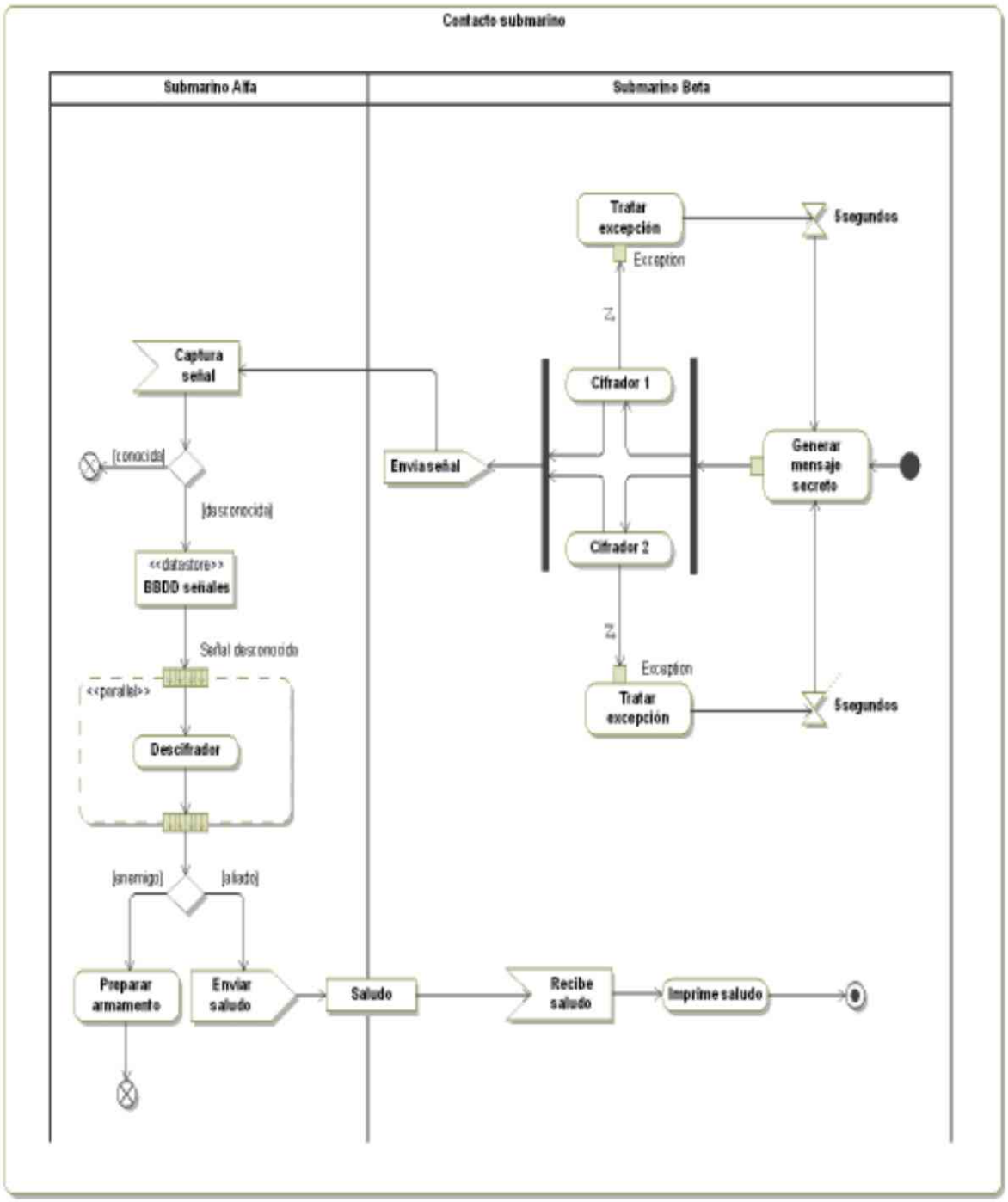
B.8 diagrama de comunicación



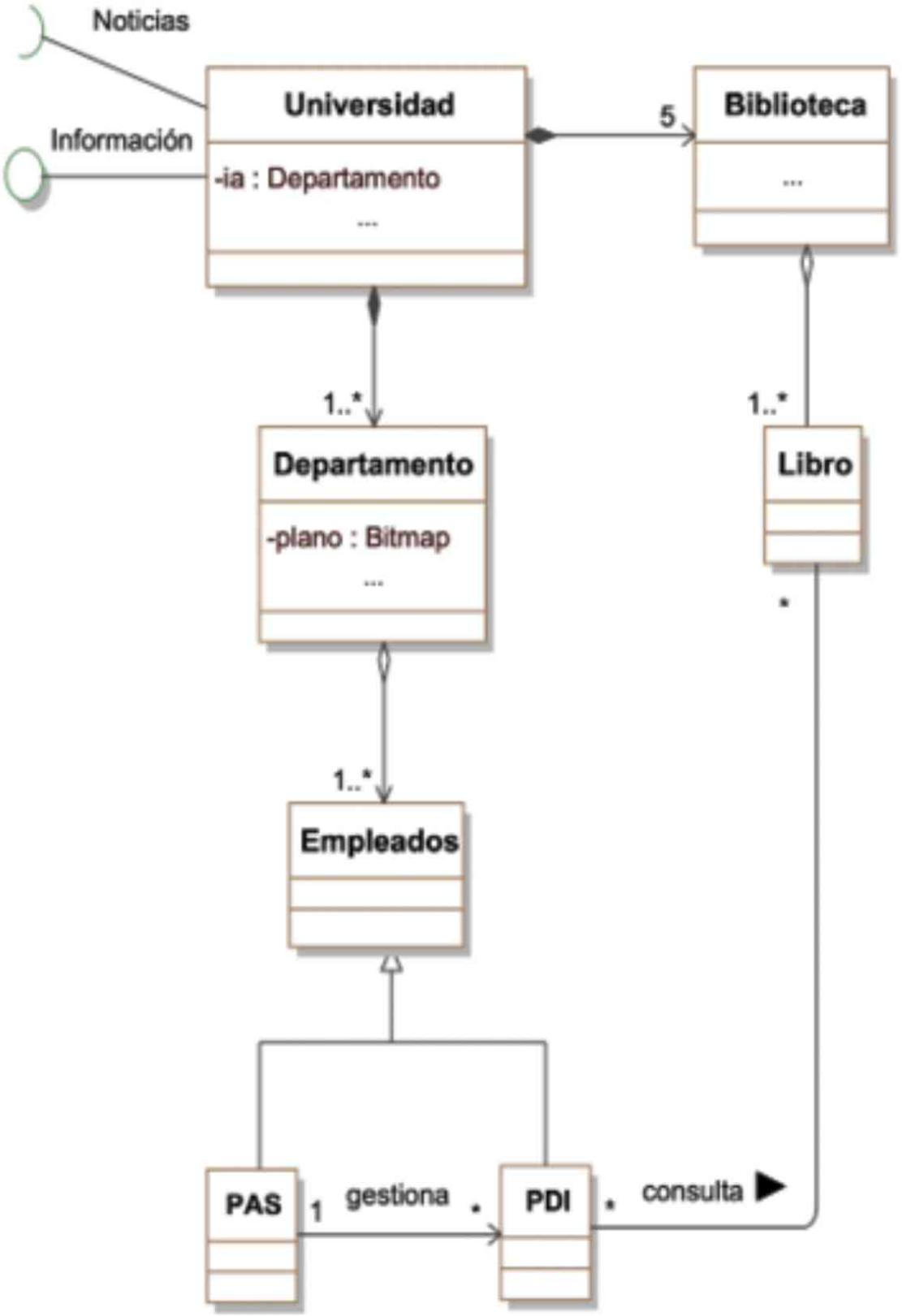
B.9 diagrama de estados

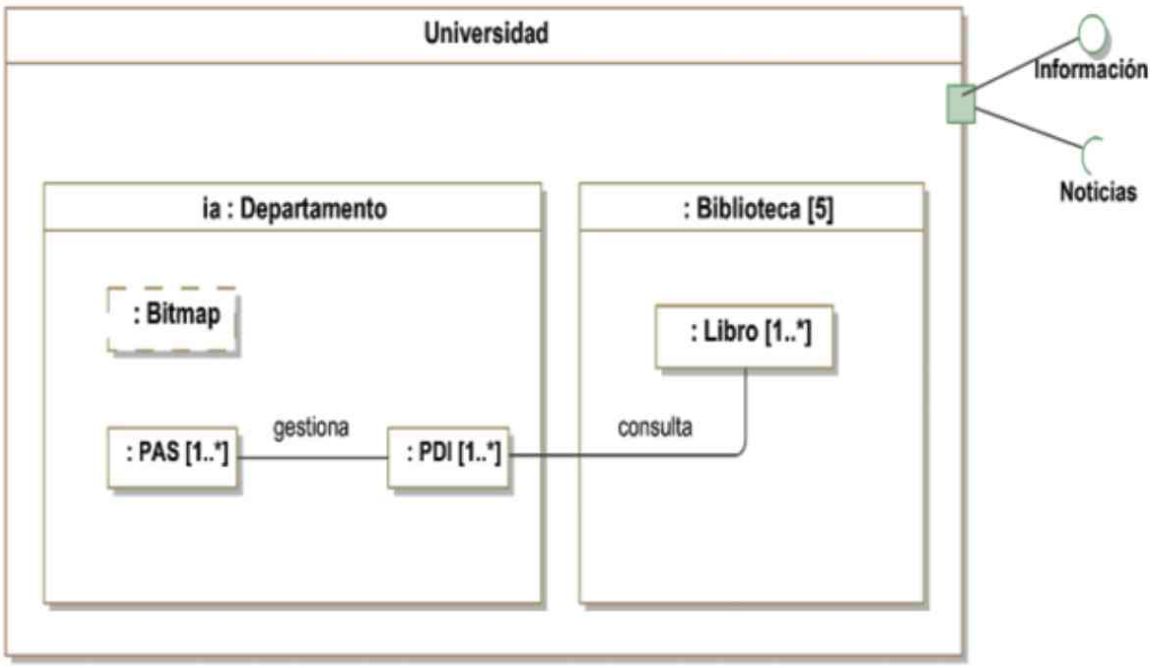


B.10 diagrama de actividades



B.11 diagrama de estructura compuesta




Universidad
Información
Noticias
ia : Departamento
: Bitmap
: PAS [1..*]
gestiona
: PDI [1..*]
: Biblioteca [5]
: Libro [1..*]
consulta

bibliografía y referencias web

[OMG1] *Unified Modeling Language: Version 2.5.1*, (2017).

[OMG2] *Response to the UML 2.0 - OCL RfP (ad/2000-09-03)*, (2003).

[Mellor04] Mellor J.S, Scott K., Uhl A. & Weise D. *MDA Distilled: Principles of Model-Driven Architecture*. Addison-Wesley, (2004).

[Fowler03] Fowler, M. UML *Distilled: A Brief Guide to the Standard Object Modeling Language (3rd Edition)*. Addison-Wesley, (2003).

[Nutshell05] Pilone, D. *UML 2.0 in a Nutshell, (2nd Edition)*. O'Reilly, (2005).

[Arlow07] Arlow J. & Neustadt I. *UML 2 and the Unified Process: Practical Object-Ori-ented Analysis and Design, (2nd Edition)*. Addison-Wesley, (2007).

[Beck99] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison Wesley, (1999).

[Gamma95] Gamma E., Helm R., Johnson R. & Vlissides J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, (1995).

[Shalloway04] Allan Shalloway & James R. Trott. *Design Patterns Explained: A New Perspective on Object-Oriented Design*. Addison-Wesley, (2004).

[Eckel03] Eckel B. *Thinking in Patterns: Problem-Solving Techniques using Java*. (2003).

[Rose99] Rosenberg D. & Scott K. *Use Case Driven Modelling with UML: a practical approach*. Addison-Wesley, (1999).

[Malveau01] Malveau R., Thomas J. & Mowbray PhD. *Software Architect Bootcamp*. Prentice-Hall, (2001).

[Sommerville03] Sommerville I. *Software Engineering (8th Edition)*. Addison-Wesley, (2006).

[Pressman02] Roger S. Pressman. *Ingeniería del Software: un enfoque práctico*. Mc-Graw Hill, (2002).

[Pressman14] Roger. S. Pressman & Bruce R. Maxim. *Software Engineering: A*

Practitioner's Approach (8th Edition). Mc-Graw Hill, (2014).

[Marick02] Marick B. *Software Testing Patterns*, 2002.

[Sutherland17] Ken Schwaber & Jeff Sutherland. La Guía de Scrum, (2017). CC-BY-SA-4.0.

[Shaw93] Garlan D. & Shaw M. *An Introduction to Software Architecture*, World Scientific Publishing Company, (1993).

[Meyer00] Meyer B. *Construcción de Software Orientado a Objetos (2ª Edición)*. Prentice Hall, (2000).

[Larman02] Larman C. *UML y Patrones: Una introducción al análisis y diseño orientado a objetos y al proceso unificado*. Pearson/Prentice-Hall, (2002).

[Perdita07] Perdita S. & Pooley R. *Utilización de UML en Ingeniería del Software con Objetos y Componentes (2ª Edición)*. Pearson/Addison-Wesley, (2007).

[Brambilla17] *Brambilla M., Cabot J., Wimmer M. Model-Driven Software Engineering in Practice: Second Edition, (2017)*. Morgan & Claypool Publishers.

[Arias07] Arias C. M. *Addenda de la asignatura Análisis, Diseño y Mantenimiento del Software*. UNED, (2007).

[Jacobson00] Ivar Jacobson, Grady Booch & James Rumbaugh. *El proceso unificado de desarrollo de software*. Addison Wesley, 2000.

[Cuevas03] Cuevas A.G. *Gestión del Proceso Software*. Centro de Estudios Ramón Areces, (2003).

[Casas] Casas C.J., Arenas F.M. *OCL (Object Constraint Language)*. Universidad Politécnica de Valencia. Facultad de Informática.

[Minguet03] Minguet M.J.M. & Ballesteros H.J.F. *La Calidad del Software y su Medida*. Centro de Estudios Ramón Areces, (2003).

[Gar87] Garvin D. *Competing on the Eight Dimensions of Quality*. Harvard Business Review, Noviembre 1987, pp. 101-109.

[Martin04] Martin R.C. *UML para Programadores Java*.

Pearson/Prentice-Hall, (2004).

[Joyanes99] Joyanes A.L. *Programación en C++: Algoritmos, estructuras de datos y objetos*. McGraw Hill, (1999).

[Stroustrup02] Stroustrup B. *El lenguaje de Programación C++, (Edición Especial)*. Addison-Wesley, (2002).

[Carballeira04] García S.J.D., Pérez M.J.M., García S.L.M., Pérez C.J. & Carballeira F.G. *Problemas resueltos de Programación en Lenguaje C++*. Thomson, (2004).

[Jalón00] Jalón J.G., Rodríguez J.I, Mingo I., Imaz A., Brazález A., Larzabal A., Calleja J. & García J. *Aprenda Java como si estuviera en primero*. Escuela Superior de Ingenieros Industriales de San Sebastián, Universidad de Navarra, (2000).

[Kurose04] Kurose F.J. & Ross K.W. *Redes de Computadores: Un enfoque descendente basado en Internet*. Pearson/Addison-Wesley, (2004).

[Botaro09] Botaro E.A. & Jiménez D.J.J. *Sistemas Operativos: Conceptos Fundamentales*. Servicio de Publicaciones Universidad de Cádiz, (2009).

[Lac19] Laczkó Dávid. *Example Plug-in Framework*. CC BY-SA 4.0.

[Barnes17] David J. Barnes, Michael Kölling. *Programación Orientada a Objetos con Java usando BlueJ*. 6ª Edición. Pearson, (2017).

[Phillips18] Dusty Phillips. *Python 3 Object-Oriented Programming*. Third Edition. Packt, (2018).

Referencias web

- [UMLDiagrams]
  UML Diagrams.
  http://www.uml-diagrams.org/
- [Abiztar]
  Diagrama de Estructura Compuesta.
  https://www.abiztar.com.mx/articulos/componiendo-lo-descompuesto-diagrama
- [SparxUML]
  UML Tutorial.
  http://www.sparxsystems.com/uml-tutorial.html
- [CiclosVida]
  Modelos de ciclo de vida.
  http://ciclodevidasoftware.wikispaces.com/CICLO+DE+VIDA+EN+ESPIRAL

material adicional

El material adicional de este libro puede descargarlo en nuestro portal web: *http://www.ra-ma.es*.

Debe dirigirse a la ficha correspondiente a esta obra, dentro de la ficha encontrará el enlace para poder realizar la descarga.

Cuando descomprima el fichero obtendrá los archivos que complementan al libro para que pueda continuar con su aprendizaje.

INFORMACIÓN ADICIONAL Y GARANTÍA

- RA-MA EDITORIAL garantiza que estos contenidos han sido sometidos a un riguroso control de calidad.
- Los archivos están libres de virus, para comprobarlo se han utilizado las últimas versiones de los antivirus líderes en el mercado.
- RA-MA EDITORIAL no se hace responsable de cualquier pérdida, daño o costes provocados por el uso incorrecto del contenido descargable.
- Este material es gratuito y se distribuye como contenido complementario al libro que ha adquirido, por lo que queda terminantemente prohibida su venta o distribución.

ÍNDICE ALFABÉTICO

Símbolos

.NET, 104
@abc.abstractmethod, 415, 418, 419, 433, 434, 436, 437, 440, 441, 444, 474, 475
@pre, 302, 303
@property, 428, 429, 470, 471, 473, 475
<<access>>, 119
<<artifacts>>, 107
<<bind>>, 138
<<component>>, 106
<<create>>, 152
<<datastore>>., 262
<<destroy>>, 152
<<device>>, 112
<<execution environment>>, 112
<<extend>>, 58
<<import>>, 119
<<include>>, 56
<<interface>>, 106, 135
<<iterative>>, 266
<<merge>>, 119
<<multicast>>, 274
<<multireceive>>, 274
<<parallel>>, 266
<<streaming>>, 266
<<throws>>, 137

<<use>>, 106, 119, 137

A

Abs, 299
Abstracción, 22, 44, 45, 51, 83, 84, 105, 134, 142, 182, 196, 203, 204, 359, 446
Abstract, 189, 330, 331, 344, 465
Abstracta, 37, 45, 83, 96, 124, 127, 186, 189, 191, 193, 196, 199, 202, 204, 205, 207, 209, 223, 225, 229, 231, 330, 344, 355, 368, 369, 410, 418, 419, 429, 447, 454, 459, 465, 466, 469, 476
AbstractCreator, 193
AbstractExpression, 205
Abstract Factory (factoría abstracta), 189, 190
AbstractProduct, 189, 193
AbstractStrategy, 202
Acceso, 79, 117, 119, 125, 127, 144, 184, 190, 305, 309, 317, 352, 361, 410, 446, 447, 448, 451, 460, 462
Access, 119
Acciones, 152, 235
Acoplamiento, 134, 182, 185, 191, 212, 217, 218, 219, 221, 222, 223
Activación, 154
Actores, 49, 50, 51, 52, 61, 72, 73, 78, 79, 94, 95, 152, 263
Actores primarios, 52, 53, 54, 55, 57, 59, 62, 63, 66, 67, 68, 69
Actores secundarios, 52, 63
AENOR (Asociación Española de Normalización), 38
AES (Advanced Encryption Standard), 64, 66, 147, 171, 180, 254, 436, 437, 438, 442, 443, 444
Agente, 49, 50, 113
Agregación, 87, 88, 132, 133, 195, 196, 281, 327, 328, 364, 365, 415, 416
Alan Kay, 445

Alcance, 88, 91, 125, 126, 139
Álgebra de conjuntos, 309
Algol, 44
Algoritmo, 64, 66, 67, 68, 69, 76, 81, 92, 94, 98, 142, 147, 165, 167, 171, 180, 202, 213, 225, 237, 254, 267, 278, 289, 292, 338, 339, 380, 381, 404, 405, 408, 409, 410, 424, 433, 436, 440, 442, 443, 444
Alt, 159, 160, 161, 316, 339, 340, 382, 425
ALU, 284, 328, 329, 366, 367, 368
Ámbito, 23, 124, 125, 127, 163, 304, 305, 320, 323, 395, 449
Análisis, 18, 23, 29, 486
Análisis de amenazas, 41
Análisis de requisitos, 42
Análisis de riesgos, 43
Análisis de software seguro, 42
Anti-patrón, 182
Append, 313
Arquitectura, 23, 45, 47, 96, 99, 100, 101, 102, 103, 109, 113, 121, 182, 183, 184, 186, 397, 460
Arquitectura basada en capas, 101
Arquitectura dirigida por modelos, 44
Arquitecturas formadas por tuberías, 100
Arrays, 267, 365
Artefactos, 29, 32, 34, 112, 114, 115, 116
Asbag, 307
Aseguramiento de la calidad del software (sqa), 39
Asíncronas, 152
Asistencia técnica, 37
Asociación, 18, 56, 85, 95, 115, 124, 128, 129, 130, 132, 141, 144, 174, 284, 288,

316, 317, 325, 327, 332, 336, 337, 343, 359, 362, 363, 371, 372, 373, 377, 413, 415, 419, 422
Asociación calificada, 336
asOrderedSet, 307
asSequence, 307
asSet, 307
at, 309
ATM, 101
Atributo, 85, 86, 124, 125, 127, 129, 141, 204, 289, 302, 304, 317, 319, 325, 361, 447, 451
Autómatas, 233, 244
Autómatas finitos, 233

B

Bag, 305, 306, 307, 308, 310, 312, 314, 315, 317, 318
Barrera de sincronización, 244, 260
Base de datos, 47, 55, 79, 114, 159, 163, 165, 184, 262, 341, 426
Bass, 99
Bind, 138
Bjarne Stroustrup, 445
BNF, 204, 205
Body, 303
Boehm, 24, 26
Bohem, 38
Booch, 21
Boolean, 298, 299, 306, 308, 318
Break, 162, 163, 341, 343, 385, 426
Broker, 185
Buffer, 238, 274, 343, 347, 387, 388, 389, 391, 393

Builder (constructor virtual), 191, 194

C

C, 17, 19, 29, 44, 46, 86, 105, 114, 116, 118, 119, 125, 134, 135, 138, 156, 270, 297, 310, 311, 312, 323, 349, 359, 360, 361, 363, 364, 365, 371, 377, 379, 381, 384, 385, 414, 422, 423, 445, 446, 447, 448, 450, 452, 455, 458, 459, 460, 461, 462, 464, 468, 486, 487
C++, 17, 19, 29, 46, 86, 105, 114, 116, 119, 125, 134, 135, 138, 156, 270, 297, 323, 349, 359, 360, 361, 363, 364, 365, 371, 377, 379, 381, 384, 385, 414, 422, 423, 445, 446, 447, 448, 450, 452, 455, 458, 459, 460, 461, 462, 464, 468, 487
Calidad, 26
Calidad del software, 36
Calificación, 129, 303, 336
Capa de acceso a datos, 184
Capa de interfaz, 184
Capa de negocio, 78, 184
Capas, 183
Cardinalidad, 85, 92, 113, 128, 131, 289, 318, 325, 326, 362, 363, 413
Carriles, 263
CASE, 24, 36, 44, 45, 173, 297
Casos de aseguramiento, 42, 43
Casos de uso, 29, 49, 50, 51, 52, 56, 58, 71, 74, 75, 76, 78, 84, 92, 94, 95, 123, 151, 152, 233, 355, 379, 394
Casting o retipado, 299
Christopher Alexander, 182
Ciclo de vida, 18, 23, 24, 26, 27, 28, 49, 78, 83, 99, 112, 187, 234, 323, 327, 328, 364, 487
Cifrado asimétrico, 64

Cifrado híbrido, 64, 171, 172, 432, 438, 439
Cifrador, 96, 97, 122, 147, 229, 231, 292, 432, 437, 438
Cifraro simétrico, 64
Clase, 78, 86, 91, 94, 104, 105, 117, 124, 125, 127, 128, 129, 130, 131, 134, 135, 136, 137, 138, 141, 142, 144, 152, 153, 174, 184, 189, 190, 191, 193, 194, 196, 197, 199, 201, 202, 203, 204, 205, 206, 209, 234, 237, 239, 240, 241, 281, 282, 283, 284, 285, 286, 287, 288, 289, 290, 291, 302, 317, 318, 319, 320, 321, 324, 325, 326, 330, 331, 332, 336, 338, 343, 349, 351, 352, 355, 356, 357, 360, 361, 364, 369, 370, 371, 372, 373, 377, 379, 397, 398, 410, 412, 447, 448, 449, 451, 452, 453, 454, 455, 457, 458, 459, 460, 461, 462, 464, 465, 466, 468
Clase abstracta, 105, 125, 134, 208, 330, 368, 386, 465, 466
Clase asociación, 130, 332, 419
Clase estructurada, 282, 286
Clases internas, 331
Cliente/servidor, 75, 79, 112, 113, 397
CMM (Modelo de la Madurez de la Capacidad), 38
COCOMO (COnstructive COst MOdel), 38
Codificación, 24
Código, 19, 22, 24, 45, 46, 47, 50, 60, 102, 114, 134, 138, 154, 190, 242, 260, 298, 304, 323, 324, 325, 336, 337, 338, 349, 355, 359, 361, 363, 365, 379, 384, 386, 394, 395, 396, 398, 400, 403, 406, 414, 449, 452, 453, 454, 457, 460, 461, 463, 466, 468
Cohesión, 134, 212, 217, 219, 221
Colaboraciones, 286
Colecciones, 305
Collect, 314
COM+, 104
Command (comando, orden), 197

Comparación, 251, 307, 318
Component, 195
Composición, 87, 88, 92, 123, 132, 133, 134, 195, 242, 281, 282, 288, 328, 365, 366, 367, 416
Composite (objeto compuesto), 195
Concat, 299
Concepto, 22, 83, 84, 85, 86, 87, 88, 94, 105, 182, 281, 324, 328, 460, 462, 464
ConcreteBuilder, 191
ConcreteCommand, 198
ConcreteCreator, 193, 194
ConcreteFactory, 189
ConcreteObserver, 200
ConcreteProduct, 193, 194
ConcreteState, 203
ConcreteStrategy, 202
ConcreteSubject, 200
Concurrencia, 155, 159, 175, 242, 246, 248, 258, 260, 273, 277
Concurrentes, 175, 243, 246, 260, 262, 271, 272, 273
Conector, 271, 282
Conformidad, 37
Consulta, 20, 155, 167, 297, 298, 303, 305, 308, 317, 341, 426
Context, 202, 203, 205, 298
Controlador, 74, 77, 176, 187, 219, 338, 339, 380, 381, 424
Conversión, 22, 24, 307, 324, 403, 458, 468
CORBA, 22, 46, 104, 185
Count, 308
CPU, 209, 284, 329, 366, 367, 368

Creador, 217
Create, 152
Criptomonedas, 222
Crisis del software, 38
Critical, 163

D

David Garvin, 37
David Harel, 233
DCOM, 185
Def, 304
Delete, 156, 350, 368, 370, 376, 378, 381, 383, 394, 402, 407, 449, 450
Dennis Ritchie, 445
Dependencias, 29, 108, 114, 118, 121, 136, 137
Derive, 305
Desarrollo de software seguro, 41
Descifrador, 97, 98, 122, 149, 231, 438, 444
Despliegue, 112, 113, 114
Destroy, 152
Device, 112, 113, 114
Diagrama de casos de uso, 29, 61
Diagrama de clases, 29, 75, 123, 135, 138, 140, 142, 143, 145, 146, 148, 173, 174, 201, 206, 282, 283, 284, 288, 301, 319, 324, 361, 363, 369, 414
Diagrama de colaboración, 286, 287
Diagrama de componentes, 105, 114, 121, 142, 394
Diagrama de comunicación, 173, 174, 176, 178
Diagrama de despliegue, 29, 113, 114, 115
Diagrama de estados, 234, 240, 241, 247, 251, 253, 255, 385, 386, 389
Diagrama de paquetes, 29, 119, 120, 121, 122

Diagrama de secuencias, 49, 152, 153, 159, 164, 165, 169, 173, 179, 199, 316, 340, 341, 349, 352, 355, 359
Diagramas asociados al comportamiento, 233
Diagramas de actividad, 151, 255, 256, 258, 262, 267
Diagramas de actividad, 29
Diagramas de componentes, 29, 104, 106, 108, 116
Diagramas de comportamiento, 49, 151, 235
Diagramas de estado de comportamiento, 236
Diagramas de estado de protocolo, 236
Diagramas de estructura compuesta, 29, 281
Diagramas de interacción, 29, 151
Diagramas de robustez, 29, 74, 75, 78, 152
Diagramas de secuencias, 29, 75, 151, 152, 174, 318, 337, 339, 379, 394, 423
Diagramas de uso de colaboración, 286
Diagramas de Venn, 309
Diccionario de la Real Academia de la Lengua (DRAE), 36
Diferencia, 50, 58, 75, 88, 99, 132, 141, 182, 311, 361, 462, 466
Diferenciación, 37
Dimensiones de la calidad, 37
Director, 191
Diseño, 18, 23, 29, 45, 80, 81, 486
Diseño arquitectónico, 27, 42, 99, 112, 116, 147
Diseño detallado, 23, 27, 78, 83, 116, 173, 183, 187, 323
Diseño global o arquitectónico, 23
Diseño orientado a objetos, 181, 486
Dispositivo, 24, 49, 112, 159, 161, 209, 325, 340, 343, 344, 362, 363, 382, 386, 428
Div, 299

DNS, 242
Do, 235, 306
Documentación, 23, 24
Dominio, 23, 45, 46, 49, 83, 84, 85, 88, 89, 90, 93, 95, 96, 97, 109, 123, 124, 128, 139, 151, 182, 202, 208, 324
DSL (Domain Specific Languages o Lenguajes específicos del dominio), 45
Duración, 37
Dynabook, 445

E

EJB, 46, 104
Encapsulación, 446
Enlaces, 78, 84, 114, 174
Entidad, 46, 72, 73, 74, 76, 78, 79, 80, 83, 86, 87, 94, 104, 112, 124, 154, 155, 248, 281, 447
Entornos críticos, 41
Entry, 235
Equipo de desarrollo (development team), 31, 40
Equipo Scrum, 32, 33, 34
Escenario, 51, 153, 161, 163, 165, 169, 173, 174, 175, 178, 179, 188, 199, 255, 260, 276, 287, 341, 426
Esclavos, 163, 175, 194
Espacio de nombres, 116, 124, 298
Especialización, 75, 86, 87
Espionaje, 41
Estado, 76, 204, 234, 244, 245, 344, 346, 387, 388, 389, 390, 391, 392, 393
Estado compuesto, 242, 243, 247, 248, 251
Estados compuestos ortogonales, 243
Estados compuestos simples, 242

Estáticos, 127, 324, 386
Estereotipo, 46, 105, 112, 119, 135, 137, 138, 142, 262, 266, 274
Estética, 37
Estímulo, 152, 154
Estructura conceptual, 63, 123
Estructura de datos, 94, 95, 102, 144, 196, 362
Estructuras, de, control, 258
Evento, 32, 33, 50, 83, 103, 151, 176, 200, 201, 207, 219, 225, 234, 235, 236, 240, 241, 348, 250, 255, 261, 267, 271, 274, 331, 341, 345, 349, 383, 386, 387, 388
Eventos de tiempo, 261
Eventos internos, 235
Excepciones, 137, 267, 270
Excludes, 308
Excludesall, 308
Excluding, 313
Execution environment, 112, 113
Exist, 315
Exit, 235
Experto, 216, 217
Expresión body, 303
Expresión def, 304
Expresión derive, 305
Expresiones de definición, 303
Expresiones de restricción, 301
Expresión init, 304
Expresión inv, 301
Expresión let, 305

Expresión post, 302
Expresión pre, 302
Expresión regular, 124, 283
Extend, 56, 58, 60, 75
Extends, 326, 331, 346, 462, 463, 464, 466, 467
Extracción, 313
Extreme programming (xp), 30

F

Fabricación pura, 221
Facade (fachada), 196, 197
Factory method (método factoría), 193
Fallos de software, 41
FFT, 267
Fiabilidad, 37, 38, 41
Filtros, 100, 101, 184, 190
First, 309
Flag, 246
Floor, 299
Flujos alternativos, 52, 54
Forall, 315
Fork (bifurcar), 244
Fortran, 44
Fragmento combinado, 159, 161, 316
Frameworks, 29, 45, 182, 190, 200
FTP, 157, 158, 159, 186

G

Generalización, 86, 87, 118, 133, 134, 135
GPL, 60

GPU, 142, 167, 405, 409, 410
GRASP, 18, 215, 216, 224, 226, 228, 229, 230, 231
GRL, 43
GUI, 142, 187, 208, 214, 397, 398, 399, 400

H

Hackers, 41
Hardware, 29, 45, 50, 51, 112, 190, 359, 386
HashMap, 336
HashSet, 328, 329
Haskell, 445
Herencia, 58, 86, 87, 97, 118, 124, 125, 133, 134, 135, 147, 149, 196, 214, 227, 294, 326, 330, 363, 364, 369, 414, 444, 446, 448, 450, 452, 453, 462, 463, 466, 470, 472, 473, 474
Herencia múltiple, 364, 414, 452
Herencia simple, 363, 414
Historia profunda, 249
Historias de usuario, 31, 32
Historia superficial, 248, 249, 251
HTML, 242, 260, 267
HTTP, 19, 115, 141, 186, 242, 487

I

IA, 76, 92, 94, 120, 142, 165, 167, 168, 209, 251, 275, 289, 399, 400, 402, 403, 404, 405, 408, 409
IEEE 610-1991, 37
Implements, 327, 357, 464
Implies, 299, 316, 318, 319, 320
Import, 119, 349, 351, 353, 356, 461, 462, 464
Include, 56, 57, 63, 75, 308, 360, 361, 366, 367, 369, 370, 372, 373, 377, 378,

380, 381, 382, 383, 386, 389, 395, 396, 397, 398, 399, 400, 403, 406, 450, 455
Includesall, 308
Including, 313
Inclusión / exclusión, 313
Incremento (increment), 34
indexOf, 309
Indirección, 221, 222
Informática, 21
Ingeniería directa, 23, 323, 349, 359, 382, 411
Ingeniero, 22, 27, 123, 181, 182, 233, 372, 373, 374, 375
Init, 304
Inserción, 313
Instancia, 75, 85, 125, 127, 152, 153, 156, 234, 240, 241, 317, 387, 388, 390, 391, 392, 394, 451, 453, 463
Integer, 298, 299, 302, 303, 304, 305, 306, 308, 309, 313, 314, 319, 336
Inteligencia artificial, 22, 76, 120, 208, 289
Interface, 29, 71, 104, 105, 106, 107, 108, 109, 110, 120, 121, 124, 134, 135, 136, 142, 144, 178, 182, 190, 200, 234, 286, 290, 323, 326, 327, 349, 352, 353, 355, 394, 397, 399, 403, 406, 415, 464, 465, 466
Interfaz, 23, 71, 75, 76, 79, 101, 104, 105, 108, 109, 110, 135, 144, 169, 178, 182, 184, 191, 193, 195, 196, 198, 200, 202, 208, 284, 285, 352, 355, 396, 397, 398, 410, 446, 462, 464
Interoperabilidad de tiempo-diseño, 45
Intérprete, 102, 144, 169, 179, 204, 298
Interpreter (intérprete), 204
Intersección, 310, 311
Intersection, 310
Inv, 301

Invocación explícita, 103
Invocación implícita, 103
Invocación recursiva, 156, 157
Invoker, 198
IP, 113
IS, 21, 22, 23, 100
isEmpty, 308
ISO 9001\
2000, 38
ISO 15504, 38
ISO (Organización Internacional para la Estandarización), 38
isUnique, 315
iteración, 54, 161, 162, 163, 178, 314, 316, 341, 342, 357, 384, 385, 426, 427, 432, 438
ITS4, 42

J

Jacobson, 21
James Gosling, 445
Java, 17, 19, 29, 103, 105, 114, 116, 120, 121, 125, 134, 135, 138, 156, 270, 297, 323, 324, 326, 331, 349, 359, 361, 364, 368, 385, 445, 459, 460, 461, 462, 463, 464, 465, 466, 468, 469, 470, 473, 474, 485, 486, 487
JEE, 114
Jeff Sutherland, 32, 486
Jerarquía, 92, 95, 125, 134, 141, 142, 144, 179, 196, 203, 210, 330, 349, 369, 410, 446, 464
Jini, 223
JNDI, 223
Join (unir), 244

K

Ken Schwaber, 32, 486
Kent Beck, 30
Krinsten Nygaard, 445

L

La norma ISO/IEC 9126, 38, 39
Last, 309
Leaf, 195
Lenguaje declarativo, 297
Let, 305
Ley de Demeter, 223
Ligadura dinámica, 134, 454
Línea de vida, 153
Linux, 60, 114, 184
Liskov, 223
Logarítmica, 30
Lógica de negocio, 29, 72, 75, 76, 187, 189, 200, 205, 395, 410
Lollipop, 105
Loop, 161, 162, 169, 269, 341, 384, 426

M

Mac OS X, 60
Maestro, 163
MagicDraw, 19, 45
Mainframe, 263
Malware, 41, 242
Mantenimiento, 24, 486
Map, 372, 373, 374, 375, 377, 378, 403, 404, 407, 409
Máquinas de estado de comportamiento, 234

Máquinas de estado de protocolo, 234
Máquina virtual, 102, 103, 359, 460
Matriz, 248, 264, 284, 285
Max, 299
McCall, 38
Mda, 44
MDA, 21, 22, 44, 45, 46, 47, 485
MDE (Model-Driven Engineering), 44
Mensaje, 49, 144, 152, 153, 154, 155, 156, 157, 159, 174, 175, 176, 177, 178, 202, 208, 241, 267, 274, 277, 316, 338, 395, 396, 397, 398, 399, 401, 406, 410, 446
Mensajes de creación, 156
Mensajes síncronos y asíncronos, 154
Merge, 119
Metaclass, 415, 418, 429, 433, 436, 437, 440, 441, 444, 474
Método, 21, 46, 124, 125, 127, 141, 144, 154, 155, 157, 169, 173, 179, 193, 199, 200, 201, 203, 204, 210, 211, 302, 303, 317, 318, 319, 320, 330, 337, 339, 355, 357, 369, 379, 403, 448, 454, 458, 460, 461, 464, 465, 466, 468
Metodologías ágiles, 30
Metodologías pesadas, 35
Métricas, 38, 43
Metrica Versión 3, 38
Min, 299
Minimax, 76, 94, 167
Mod, 299
Modelado, 21, 22, 45, 50, 83, 88, 89, 119, 139, 151, 157, 255, 330, 364
Modelado de amenazas, 43
Modelado de seguridad, 42

Modelo, 21, 22, 23, 24, 26, 29, 45, 46, 55, 58, 71, 72, 74, 75, 78, 83, 84, 85, 88, 91, 92, 94, 96, 108, 109, 112, 114, 116, 120, 121, 123, 128, 135, 139, 141, 142, 144, 151, 181, 187, 199, 201, 206, 208, 242, 248, 262, 286, 288, 289, 290, 291, 297, 298, 300, 319, 320, 323, 325, 330, 349, 360, 369, 394, 403
Modelo de amenazas, 42, 44
Modelo de análisis, 23, 71, 78
Modelo del dominio, 71, 72, 83, 84, 85, 91, 92, 109, 123, 141, 142
Modelo en cascada, 24
Modelo en espiral, 26
Modelo incremental, 25
Modelo-Vista-Controlador, 74, 78, 187
Modularidad, 446
MP3, 274
Multicast, 274, 275
Multiparadigma, 18, 411
Multiplataforma, 18, 47, 411
Multiplicidad, 85, 108, 113, 128, 131, 142, 317, 363
Multiprocesadores, 264
Multireceive, 274
Multithread, 155

N

Namespace, 116, 360, 361, 366, 367, 370, 372, 373, 377, 378, 380, 381, 383, 387, 389, 395, 396, 397, 398, 399, 400, 403, 406
Navegabilidad, 85, 128, 141
Navegación, 317
Neuronal, 248, 264
New, 156, 325, 326, 328, 329, 331, 336, 339, 341, 342, 343, 348, 350, 351, 367, 368, 370, 375, 376, 378, 381, 383, 390, 391, 392, 393, 408, 449, 450, 460, 461,

463, 468
Nodo de final de actividad, 274
Nodo de final de flujo, 273
Nodos condicionales, 238
Nodos de actividad, 255
Nodos de concurrencia, 260
Nodos de control, 255
Nodos de datos, 262
Nodos de decisión, 258
Nodos de expansión, 266
Nodos de interconexión, 237
Nodos de objeto, 255
Nodos inicial y final, 236
Nodos objeto, 262
No Hable con Extraños, 223
NonTerminalExpression, 205
Normalización y certificación, 38
notEmpty, 308

O

Object Management Group, 44
Object-Pascal, 445
Objeto, 61, 74, 75, 76, 78, 79, 80, 87, 99, 120, 125, 128, 129, 130, 131, 134, 153, 154, 155, 156, 157, 159, 161, 164, 165, 167, 169, 174, 175, 176, 178, 179, 185, 190, 191, 193, 195, 198, 199, 201, 202, 203, 204, 207, 208, 209, 210, 211, 233, 242, 253, 255, 262, 263, 264, 265, 274, 277, 299, 300, 306, 307, 308, 309, 317, 320, 325, 328, 336, 338, 339, 340, 341, 343, 351, 365, 379, 381, 385, 423, 426, 446, 447, 448, 449, 451, 453, 454, 458, 459, 461, 463, 465, 468
Objeto de control, 72

Objeto de frontera, 71
Objeto Entidad, 72
Observer, 200
Observer (observador), 200
OclAny, 299
Ocl en los diagramas de estado, 318
oclInState, 318
OCL (Object Constraint Language o lenguaje de restricción de objetos), 297
OclType, 299
Ocultación de información, 446
Ole-Johan Dahl, 445
OMG, 22, 44, 45, 234, 323
OOSE, 21
Operaciones, 125, 152, 306, 307, 308, 309
Operador @pre, 302
Opt, 159, 160, 161
OrderedSet, 305, 306, 307, 309, 310, 311, 312, 313, 314
OTAN, 38

P

P2P (Peer-to-Peer), 104
Package, 116, 298, 319, 320, 321, 349, 351, 352, 355, 356, 357, 459, 460, 462, 463, 465
Pair-programming, 31
Paquete, 29, 116, 117, 118, 119, 120, 121, 124, 125, 142, 274, 298, 319, 321, 349, 351, 352, 353, 355, 356, 357, 395, 396, 397, 398, 399, 400, 403, 406, 410, 432, 433, 434, 435, 437, 438, 440, 441, 444, 459, 460, 461, 462
Paralelismo, 29, 163, 242, 243, 260, 273
Parametrizable, 138

Parte, 18, 83, 173, 282
Particiones, 263
Patrón, 74, 78, 95, 100, 144, 163, 169, 182, 183, 184, 185, 187, 188, 189, 190, 191, 192, 193, 194, 195, 196, 197, 198, 199, 200, 201, 202, 203, 204, 205, 206, 207, 208, 209, 210, 211, 212, 213, 214, 224, 286, 343, 349, 350, 351, 352, 355, 357, 385, 397, 410, 459
Patrón Abstract Factory (factoría abstracta), 189
Patrón Builder (constructor virtual), 191
Patrón Command (comando, orden), 197
Patrón Composite (objeto compuesto), 195
Patrones arquitectónicos, 183
Patrones de análisis, 183
patrones de ataque, 42
Patrones de codificación, 183
Patrones de comportamiento, 197
Patrones de creación, 189
Patrones de diseño, 74, 181, 183, 187, 209, 233, 343, 364, 385
Patrones estructurales, 195
Patrón Facade (fachada), 196
Patrón Factory Method (método factoría), 193
Patrón Interpreter (intérprete), 204
Patrón Observer (observador), 200
Patrón State (estado), 203
Patrón Strategy (estrategia), 202
PCM, 274
Peer-to-peer, 186
Periférico, 24, 50
Perl, 364

Personal, 25, 30, 40
Phishing, 41, 42
PHP, 445
Pila del Producto (Product Backlog), 33, 34
Pila del sprint (sprint backlog), 34
PIM (Platform Indepenent Model), 45
Pin, 264, 270
Pizarra, 102, 184, 185
Planificación del sprint, 33
Plantilla (templates), 138
Plug-in, 184, 185
Poda alfa-beta, 76
Polimorfismo, 134, 220, 455, 446, 453, 454, 455, 457, 458, 465, 466, 468, 474
Post, 302
Postcondiciones, 52, 63, 257
Pre, 302
Precondiciones, 52, 63, 257
Preppend, 313
Prestaciones, 37
Private, 451
Proceso, 23, 24
Product, 189, 191, 312
Producto cartesiano, 312, 316, 320
Programación genérica, 138
Programación orientada a objetos, 17, 445, 446, 447, 454, 458, 459
Programación por pares, 31
Propiedad, 282
Propietario del producto (product owner), 33

Protected, 451
Proxy, 104
Pruebas, 24
Pruebas de unidad, 31, 32
Pruebas de validación, 32
Pseudoestados, 242, 248
PSM (Platform Specific Model), 45
Public, 451
Puerto, 107, 108, 196, 234, 237, 282, 283, 284, 285, 286, 290, 343, 346
Puntos Función, 38
Pyhton, 364, 411, 470

R

Rational, 21, 45
RATS, 42
Real, 274, 298, 299, 305
Realización, 135, 240
Receiver, 198
Recolección de basura, 446, 460
Recursividad, 196, 211, 338, 379, 380, 381, 423, 424
Recursivo, 206, 339
Red de comunicaciones, 114
Redes de Petri, 29, 255
Ref, 164, 165, 362, 399, 400, 401, 402, 404, 405, 408, 409
Refactorización, 31
Reflexiva, 95, 129, 131, 339
Reflexivo, 157, 176, 338
Regiones de expansión, 265
Regiones if, 268

Regiones loop, 269
Región interrumpible, 267
Reglas, 32
Reject, 314
Repositorios, 60, 102, 144
Requerimientos no funcionales, 112
Requisitos, 18, 23, 24, 26, 27, 29, 49, 52, 71, 83, 84, 96, 99, 109, 120, 123, 142, 257, 302
Responsabilidad, 31, 127, 149, 189, 193, 215, 216, 217, 219, 220, 229, 231, 263
Restricciones, 46, 72, 257, 297, 298, 319, 320, 462
Retrospectiva del sprint (sprint retrospective), 34
Reutilización, 44, 45, 47, 104, 182, 210
Revisión del sprint (sprint review), 34
Roles, 32, 128, 144, 194, 286, 287, 289, 290, 291, 352
Round, 299
Royce, 24
RSA, 64, 66, 147, 171, 180, 254, 436, 437, 438, 442, 443, 444
RTP (Real-time Transport Protocol), 274
Ruby, 46
Rumbaugh, 21
RUP (Rational Unified Process), 25

S

Saltos condicionales, 159, 176, 339, 382, 425
SAP, 101
Scrum Diario (Daily Scrum), 34
Scrum Master (Scrum Master), 33
Sebastián, Bargueño, y Novo, 37
Sección crítica, 163

Secuencia, 53, 54, 74, 141, 151, 153, 154, 159, 165, 167, 169, 173, 174, 175, 178, 179, 190, 203, 210, 242, 251, 256, 274, 305, 313, 338, 339, 340, 341, 343, 379, 382, 383, 384, 423
Seguridad, 43
Select, 314
Self, 175, 302, 303, 304, 305, 314, 315, 316, 317, 318, 319, 320
Semántica, 61, 74, 75, 78, 80, 87, 88, 91, 92, 124, 128, 133, 142, 160, 173, 174, 253, 255, 258, 281, 297
Semantic Task Force, 22
Sequence, 305, 306, 307, 308, 309, 310, 312, 313, 314
Servicio de cifrado remoto, 65, 111, 115, 121
Servicios Web (SOA)., 47
Set, 305, 306, 307, 310, 311, 312, 313, 314, 317, 318, 320, 328, 329
Símbolo, 87, 105, 127, 131, 132, 238, 264, 281
SIMULA, 445
Síncronas, 152
Singleton, 148, 194, 195, 212, 213, 438, 440
Sistema de Gestión de la Calidad (SGC), 38
Sistemas distribuidos, 185
Sistemas en tiempo real, 23
Size, 299, 308
SLAM, 42
SLIM, 38
SMTP, 113, 186
SOA, 186
SOAP, 186
Sobrecarga, 446
Socket, 105, 144, 178, 212, 254, 272, 277, 278, 397, 432, 433, 434, 435, 436,

442
Software, 17, 21, 23, 24, 71
Software en tiempo real, 274
SortedBy, 314
Spam, 68
Sprint (Sprint), 33
SQA, 39, 40, 41, 43
SQL injection, 42
StarUML, 19, 45
State (estado), 203
STL, 135, 362, 365
Strategy (estrategia), 202
Streaming, 266, 267, 274
String, 144, 208, 210, 211, 284, 298, 299, 303, 304, 320, 324, 329, 331, 336, 337, 338, 342, 345, 350, 351, 352, 355, 357, 460, 461, 463, 465, 468
Subclase, 127, 133, 202
Subject, 200
Submáquinas, 243, 244, 246, 248
subSequence, 313
Subsistema, 105, 109, 110, 120, 121, 144, 196, 206, 349, 351, 352, 361, 379
subString, 299
Sum, 308
Sun, 445
Superclase, 133, 142, 195, 204
Superestado, 242, 248, 250
Swimlanes, 263
symmetricDifference, 312

T

Tabla, 54, 63, 78, 85, 94, 95, 131, 163, 235, 298, 306, 308, 309, 453, 454, 458
Tablas virtuales o vtable, 454
Tarjetas CRC, 31
TCP/IP, 66, 101, 115, 434, 435
Terminado (Done), 33, 34
TerminalExpression, 205
Ternaria, 130
Test de aceptación, 32
This, 161, 175, 324, 351, 361, 373, 374, 375, 378, 392, 393, 408, 448, 450, 451, 455, 460, 463, 465
Thread, 156, 159
Timer, 159, 161, 340, 341, 382, 383
Timers, 50
toInteger, 299
tokens, 274
toLower, 299
toReal, 299
toUpper, 299
TraderService, 223
Transición, 234
Tres Amigos, 21
Tuple, 312

U

UC (unidad de control), 284
UDDI, 186, 223
UDP, 274
UML Partners, 22
UMLsec, 43

Unión, 120, 309, 310, 311
Universidad de Utah, 445
UNIX, 100, 113, 257
URL, 19, 141, 242, 260
Use, 106, 119, 137

V

Variaciones Protegidas (VP), 222
Vector, 135, 325, 342, 351, 352, 362, 364, 365, 366, 368, 380, 382, 385, 386, 395, 396, 403, 404, 407, 461
Videojuegos, 88, 182
Visibilidad, 108, 117, 125, 126, 128, 325, 361, 459
VPTR, 454
Vulnerabilidades, 41, 42

W

Web, 19, 47, 112, 186, 220, 241, 242, 260, 267
Windows, 19, 60, 114, 184
WSDL, 104, 186

X

Xerox, 445
XMI, 45
XML, 45, 114, 186