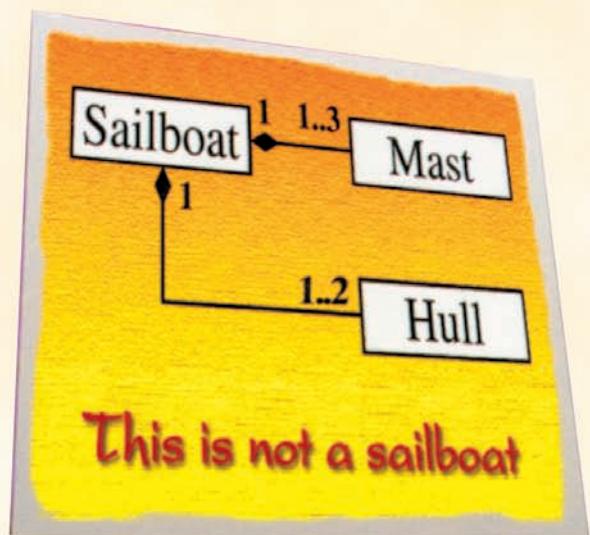
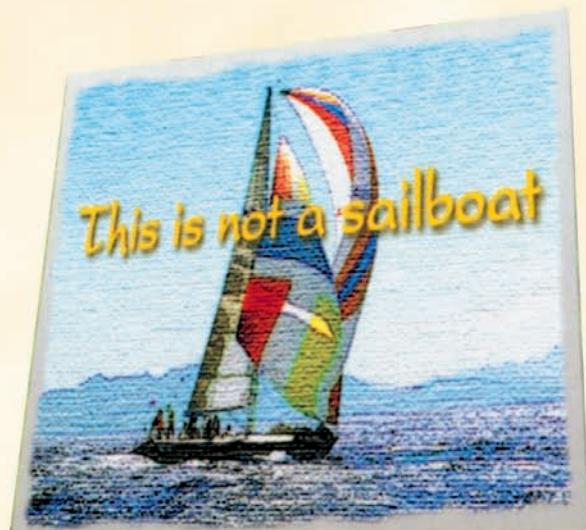


2^a Edición

UML y Patrones

Una introducción al análisis y diseño orientado a objetos y al proceso unificado

www.librosite.net/larman



PEARSON
Prentice Hall

Craig Larman

UML Y PATRONES

UNA INTRODUCCIÓN AL ANÁLISIS
Y DISEÑO ORIENTADO A OBJETOS
Y AL PROCESO UNIFICADO

Segunda edición

CRAIG LARMAN

Traducción:

Begoña Moros Valle
Universidad de Murcia

Supervisión de la traducción y revisión técnica:

Jesús García Molina
Universidad de Murcia



Madrid • México • Santafé de Bogotá • Buenos Aires • Caracas • Lima
Montevideo • San Juan • San José • Santiago • São Paulo • White Plains

Datos de catalogación bibliográfica
LARMAN, C.
<i>UML Y PATRONES. Una introducción al análisis y diseño orientado a objetos y al proceso unificado.</i> Segunda edición
PEARSON EDUCACIÓN, S.A., Madrid, 2003
ISBN eBook: 978-84-832-2927-9 Materia: Informática 681.3
Formato 195 × 250
Páginas: 624

Todos los derechos reservados.

Queda prohibida, salvo excepción prevista en la ley, cualquier forma de reproducción, distribución, comunicación pública y transformación de esta obra sin contar con autorización de los titulares de propiedad intelectual. La infracción de los derechos mencionados puede ser constitutiva de delito contra la propiedad intelectual (*arts. 270 y sgts. Código Penal*).

DERECHOS RESERVADOS

© 2003 respecto a la segunda edición en español por:

PEARSON EDUCACIÓN, S.A.

Núñez de Balboa, 120
28006 Madrid

LARMAN, C.

UML Y PATRONES. Una introducción al análisis y diseño orientado a objetos y al proceso unificado. Segunda edición

ISBN: 84-205-3438-2

Depósito Legal: M-

PRENTICE HALL es un sello editorial autorizado de PEARSON EDUCACIÓN, S.A.

Traducido de:

APPLYING UML AND PATTERNS: An Introduction to Object-Oriented Analysis and Design and the Unified Process, Second edition, by Craig Larman.

Published by Pearson Education, Inc. Publishing as Prentice Hall PTR.

Prentice Hall, Inc.

© 2002 All rights reserved.

ISBN: 0-13-092569-1

Edición en español:

Equipo editorial:

Editor: David Fayerman Aragón

Técnico editorial: Ana Isabel García

Equipo de producción:

Director: José Antonio Clares

Técnico: José Antonio Hernán

Diseño de cubierta: equipo de diseño de PEARSON EDUCACIÓN, S.A.

Composición: COPIBOOK, S.L.

Impreso por:

IMPRESO EN ESPAÑA - PRINTED IN SPAIN



Este libro ha sido impreso con papel y tintas ecológicos

Para Julie

Sin su apoyo, esto no habría sido posible.

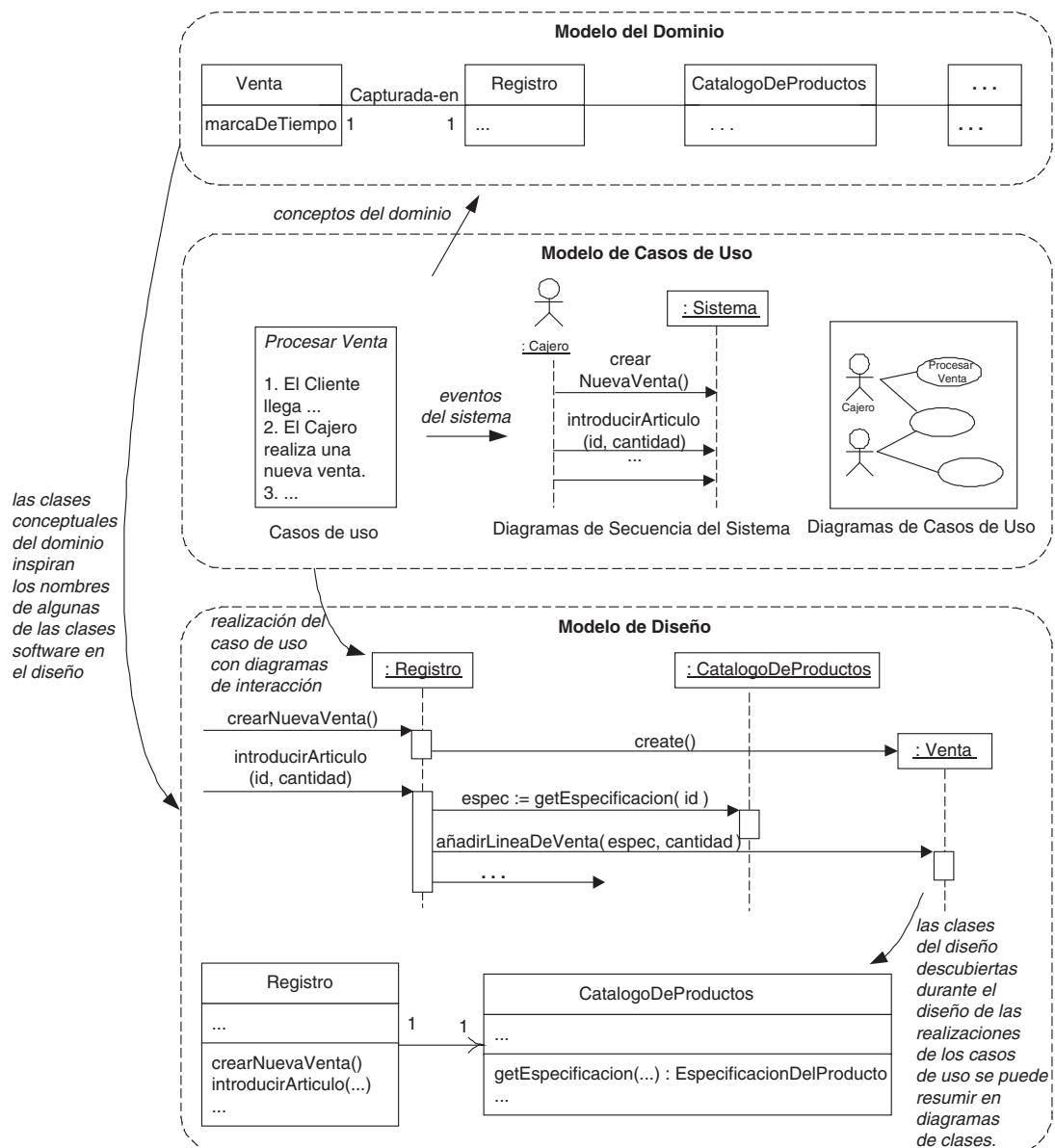
Para Haley y Hannah

Gracias por aguantar a un papá distraído, ¡otra vez!

Muestra de los artefactos del Proceso Unificado y evolución temporal (c-comenzar; r-refinar)

Disciplina	Artefacto Iteración →	Inicio I1	Elab. E1...En	Const. C1...Cn	Trans. T1...T2
Modelado del Negocio	Modelo del Dominio		c		
Requisitos	Modelo de Casos de Uso Visión Especificación Complementaria Glosario	c c c c	r r r r		
Diseño	Modelo de Diseño Documento de Arquitectura SW Modelo de Datos		c c c	r r	
Implementación	Modelo de Implementación		c	r	r
Gestión del Proyecto	Plan de Desarrollo SW	c	r	r	r
Pruebas	Modelo de Pruebas		c	r	
Entorno	Marco de Desarrollo	c	r		

Muestra de las relaciones entre los artefactos del Proceso Unificado



Patrones de Software Generales para la Asignación de Responsabilidades (GRASP)

<i>Patrón</i>	<i>Descripción</i>
Experto en Información	<p>¿Un principio general del diseño de objetos y la asignación de responsabilidades?</p> <p>Asigne una responsabilidad al experto en información, —la clase que tiene la información necesaria para llevar a cabo la responsabilidad.</p>
Creador	<p>¿Quién crea? (Nótese que la Factoría es una solución alternativa frecuente.)</p> <p>Asigne a la clase B la responsabilidad de crear una instancia de la clase A si se cumple alguno de los puntos siguientes:</p> <ul style="list-style-type: none"> 1. B contiene a A 2. B agrega a A 3. B tiene los datos de inicialización de A 4. B registra a A 5. B utiliza estrechamente a A
Controlador	<p>¿Quién gestiona un evento del sistema?</p> <p>Asigne la responsabilidad de gestionar un mensaje de un evento del sistema a una clase que represente una de estas opciones:</p> <ul style="list-style-type: none"> 1. Representa el sistema global, dispositivo o un subsistema (controlador de fachada). 2. Representa un escenario de caso de uso en el que tiene lugar el evento del sistema (controlador de caso de uso o sesión).
Bajo Acoplamiento (evaluativo)	<p>¿Cómo dar soporte a las bajas dependencias y al incremento de la reutilización?</p> <p>Asigne responsabilidades de manera que el acoplamiento (innecesario) se mantenga bajo.</p>
Alta Cohesión (evaluativo)	<p>¿Cómo mantener manejable la complejidad?</p> <p>Asigne responsabilidades de manera que la cohesión permanezca alta.</p>
Polimorfismo	<p>¿Quién es el responsable cuando el comportamiento varía en función del tipo?</p> <p>Cuando las alternativas o comportamientos relacionados varían según el tipo (clase), asigne la responsabilidad del comportamiento —utilizando operaciones polimórficas— a los tipos para los que varía el comportamiento.</p>
Fabricación Pura	<p>¿Quién es el responsable cuando está desesperado, y no quiere violar los principios de alta cohesión y bajo acoplamiento?</p> <p>Asigne un conjunto altamente cohesivo de responsabilidades a una clase de “comportamiento” artificial o de conveniencia que no representa un concepto del dominio del problema —algo inventado—, para dar soporte a la alta cohesión, bajo acoplamiento y la reutilización.</p>
Indirección	<p>¿Cómo asignar responsabilidades para evitar el acoplamiento directo?</p> <p>Asigne la responsabilidad a un objeto intermedio para mediar entre otros componentes o servicios, de manera que no se acoplan directamente.</p>
Variaciones Protegidas	<p>¿Cómo asignar responsabilidades a los objetos, subsistemas, y sistemas de manera que las variaciones o inestabilidad en estos elementos no influya de manera no deseable en otros elementos?</p> <p>Identifique los puntos de variaciones predecibles o inestabilidad; asigne las responsabilidades para crear una “interfaz” estable alrededor de ellos.</p>

Diagrama de Secuencia

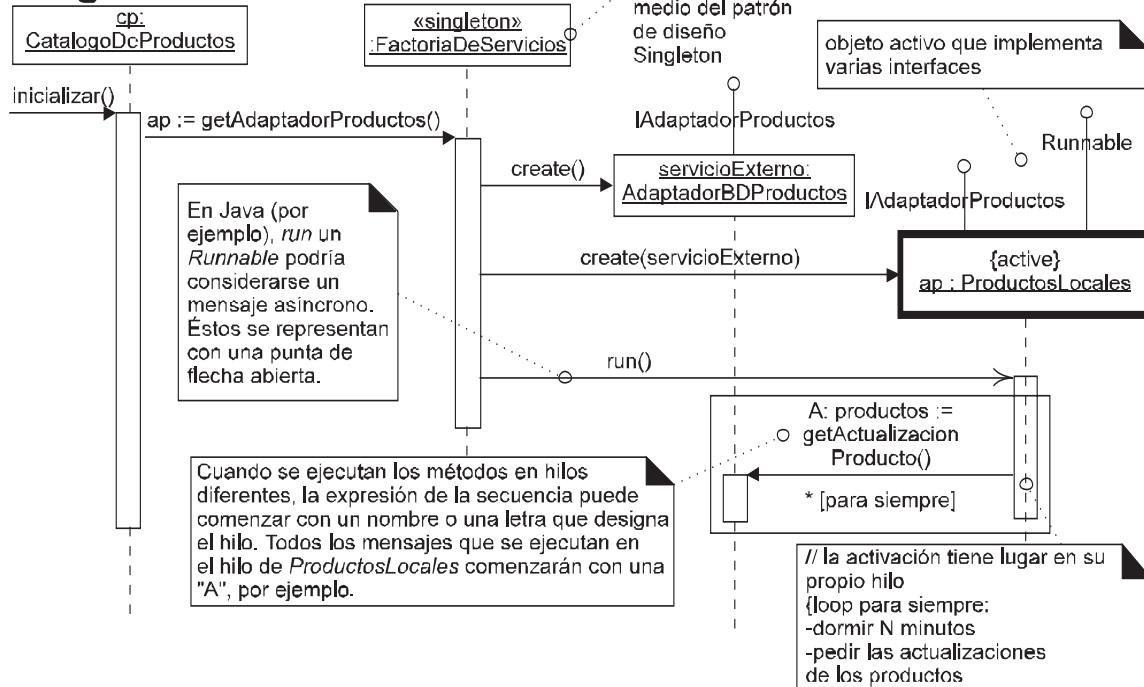
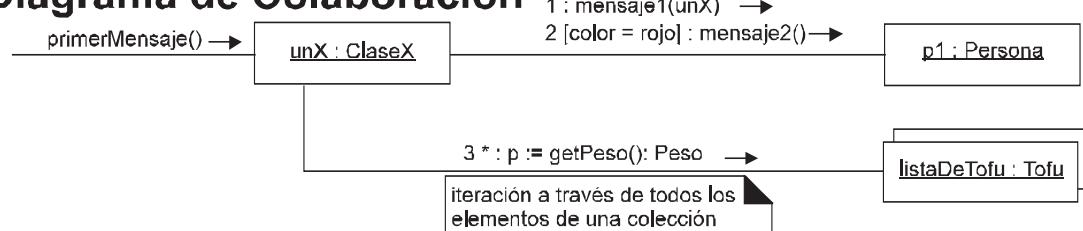
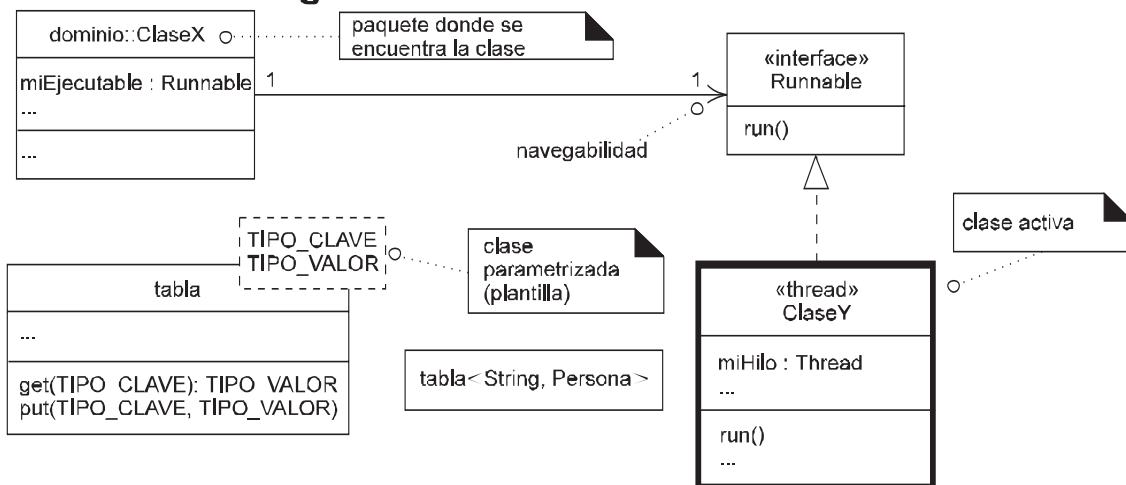


Diagrama de Colaboración



Notación del Diagrama de Clase Adicional



RESUMEN DE CONTENIDO

Parte 1: Introducción

1.	Análisis y diseño orientado a objetos	3
2.	Desarrollo iterativo y el Proceso Unificado	13
3.	Caso de estudio: el sistema de punto de venta NuevaEra	27

Parte 2: Inicio

4.	Inicio	33
5.	Comprensión de los requisitos	39
6.	Modelo de Casos de Uso: escritura de requisitos en contexto	43
7.	Identificación de otros requisitos	79
8.	Del inicio a la elaboración	103

Parte 3: Elaboración en la Iteración 1

9.	Modelo de Casos de Uso: representación de los diagramas de secuencia del sistema	113
10.	Modelo del Dominio: visualización de conceptos	121
11.	Modelo del Dominio: añadir asociaciones	145
12.	Modelo del Dominio: añadir atributos	157
13.	Modelo de Casos de Uso: añadir detalles con los contratos de las operaciones	167
14.	De los requisitos al diseño en esta iteración	181
15.	Notación de los diagramas de interacción	185
16.	GRASP: diseño de objetos con responsabilidades	201
17.	Modelo de Diseño: realización de los casos de uso con los patrones GRASP	231
18.	Modelo de Diseño: determinación de la visibilidad	261

VIII RESUMEN DE CONTENIDO

19. Modelo de Diseño: creación de los diagramas de clases de diseño	267
20. Modelo de Implementación: transformación de los diseños en código ..	281

Parte 4: Elaboración en la Iteración 2

21. La Iteración 2 y sus requisitos	291
22. GRASP: más patrones para asignar responsabilidades	305
23. Diseño de las realizaciones de casos de uso con los patrones de diseño GoF ..	321

Parte 5: Elaboración en la Iteración 3

24. La Iteración 3 y sus requisitos	359
25. Relaciones entre casos de uso	361
26. Modelado de la generalización	369
27. Refinamiento del modelo del dominio	383
28. Añadir nuevos DSSs y contratos	401
29. Modelado del comportamiento con diagramas de estado	407
30. Diseño de la arquitectura lógica con patrones	417
31. Organización de los paquetes de los modelos de diseño e implementación ..	443
32. Introducción al análisis arquitectural y el SAD	451
33. Diseño de más realizaciones de casos de uso con objetos y patrones	473
34. Diseño de un framework de persistencia con patrones	501

Parte 6: Temas especiales

35. Sobre el dibujo de diagramas y las herramientas	531
36. Introducción a cuestiones relacionadas con la planificación iterativa y el proyecto	539
37. Comentarios acerca del desarrollo iterativo y el UP	553
38. Más notación UML	567

CONTENIDO

Presentación	XIX
Prólogo	XXI

Parte 1: Introducción

Capítulo 1: Análisis y diseño orientado a objetos	3
Aplicación de UML y patrones en el A/DOO	3
Asignación de responsabilidades	5
¿Qué es análisis y diseño?	6
¿Qué son el análisis y diseño orientados a objetos?	6
Un ejemplo	7
UML	10
Lecturas adicionales	10
Capítulo 2: Desarrollo iterativo y el Proceso Unificado	13
La idea más importante del UP: desarrollo iterativo	14
Conceptos y buenas prácticas del UP adicionales	18
Las fases del UP y términos orientados a la planificación	19
Las disciplinas del UP (eran flujos de trabajo)	20
Adaptación del proceso y el Marco de Desarrollo	22
El UP ágil	23
El ciclo de vida “en cascada” secuencial	24
No se entendió el UP cuando...	25
Lecturas adicionales	25
Capítulo 3: Caso de estudio: el sistema de punto de venta NuevaEra	27
El sistema de punto de venta NuevaEra	27
Capas arquitectónicas y el énfasis del caso de estudio	28
Estrategia del libro: aprendizaje y desarrollo iterativo	29

Parte 2: Inicio

Capítulo 4: Inicio	33
Inicio: una analogía	34
La fase de inicio podría ser muy breve	35
¿Qué artefactos podrían crearse en la fase de inicio?	35
No se entendió la fase de inicio cuando...	36
Capítulo 5: Comprensión de los requisitos	39
Tipos de requisitos	40
Lecturas adicionales	41
Capítulo 6: Modelo de Casos de Uso: escritura de requisitos en contexto	43
Objetivos e historias	44
Antecedentes	44
Casos de uso y valor añadido	45
Casos de uso y requisitos funcionales	46
Tipos de casos de uso y formatos	46
Ejemplo completo: Procesar Venta	47
Explicación de las secciones	52
Objetivos y alcance de un caso de uso	56
Descubrimiento de actores principales, objetivos y casos de uso	60
Enhorabuena: se han escrito los casos de uso y no son perfectos	64
Escritura de casos de uso en un estilo esencial independiente de la interfaz de usuario	65
Actores	67
Diagramas de casos de uso	68
Requisitos en contexto y lista de características de bajo nivel	70
Los casos de uso no son orientados a objetos	71
Casos de uso en el UP	72
Caso de estudio: casos de uso en la fase de inicio de NuevaEra	76
Lecturas adicionales	76
Artefactos UP y contexto del proceso	77
Capítulo 7: Identificación de otros requisitos	79
Ejemplos del PDV NuevaEra	80
Ejemplo NuevaEra: Especificación Complementaria (Parcial)	80
Comentario: Especificación Complementaria	84
Ejemplo NuevaEra: Visión (Parcial)	87
Comentario: Visión	90
Ejemplo NuevaEra: un Glosario (Parcial)	94
Comentario: Glosario (Diccionario de Datos)	95
Especificaciones fiables: ¿un Oxímoron?	96
Artefactos disponibles en el sitio web del proyecto	97
¿Poco UML durante la fase de inicio?	97
Otros artefactos de requisitos en el UP	97

Lecturas adicionales	99
Artefactos UP y contexto del proceso	100
Capítulo 8: Del inicio a la elaboración	103
Punto de control: ¿qué sucedió en el inicio?	104
En la elaboración	105
Planificación de la siguiente iteración	106
Requisitos y énfasis de la iteración 1: habilidades de A/DOO fundamentales ..	108
¿Qué artefactos podrían crearse en la elaboración?	108
No se entendió la elaboración cuando.....	110
Parte 3: Elaboración en la Iteración 1	
Capítulo 9: Modelo de Casos de Uso: representación de los diagramas de secuencia del sistema	113
Comportamiento del sistema	114
Diagramas de secuencia del sistema	114
Ejemplo de un DSS	115
DSS entre sistemas	115
DSS y los casos de uso	116
Eventos del sistema y los límites del sistema	116
Asignación de nombres a los eventos y operaciones	117
Mostrar el texto del caso de uso	118
Los DSS y el Glosario	118
DSS en el UP	118
Lecturas adicionales	119
Artefactos del UP	120
Capítulo 10: Modelo del Dominio: visualización de conceptos	121
Modelos del Dominio	122
Identificación de las clases conceptuales	126
Clases conceptuales candidatas para el dominio de ventas	129
Guías para el modelado del negocio	130
Resolución de clases conceptuales similares: Registro vs. “TPDV”	132
Modelado del mundo <i>irreal</i>	133
Clases conceptuales de especificación o descripción	133
Notación UML, modelos y métodos: perspectivas múltiples	136
Reducción del salto en la representación	138
Ejemplo: el Modelo del Dominio del PDV NuevaEra	140
Modelos del Dominio en el UP	140
Lecturas adicionales	142
Artefactos del UP	142
Capítulo 11: Modelo del Dominio: añadir asociaciones	145
Asociaciones	145
Notación de las asociaciones en UML	146

Localización de las asociaciones—lista de asociaciones comunes	147
Guías para las asociaciones	148
Roles	149
¿Cómo de detalladas deben ser las asociaciones?	150
Asignación de nombres a las asociaciones	151
Múltiples asociaciones entre dos tipos	152
Asociaciones e implementación	152
Asociaciones del Modelo del Dominio del PDV NuevaEra	153
Modelo del Dominio del PDV NuevaEra	154
Capítulo 12: Modelo del Dominio: añadir atributos	157
Atributos	157
Notación de los atributos en UML	158
Tipos de atributos válidos	158
Clases de tipos de datos no primitivos	160
Deslizarse al diseño: ningún atributo como clave ajena	162
Modelado de cantidades y unidades de los atributos	162
Atributos en el Modelo del Dominio de NuevaEra	163
Multiplicidad de la LineaDeVenta al Articulo	163
Conclusión del Modelo del Dominio	164
Capítulo 13: Modelo de Casos de Uso: añadir detalles con los contratos de las operaciones	167
Contratos	167
Ejemplo de contrato: introducirArticulo	168
Secciones del contrato	169
Postcondiciones	169
Discusión: postcondiciones de introducirArticulo	171
La escritura de los contratos da lugar a actualizaciones en el Modelo del Dominio	172
¿Cuándo son útiles los contratos? ¿Contratos vs. casos de uso?	173
Guías: contratos	173
Ejemplo del PDV NuevaEra: contratos	174
Cambios en el Modelo del Dominio	175
Contratos, operaciones y UML	176
Contratos de las operaciones en el UP	177
Lecturas adicionales	178
Capítulo 14: De los requisitos al diseño en esta iteración	181
Iterativamente hacer lo correcto, y hacerlo correcto	181
¿No lleva eso semanas en hacerse? No, no exactamente	182
Pasar al diseño de objetos	182
Capítulo 15: Notación de los diagramas de interacción	185
Diagramas de secuencia y colaboración	186
Ejemplo de diagrama de colaboración: realizarPago	187
Ejemplo de diagrama de secuencia: realizarPago	187
Los diagramas de interacción son importantes	188

Notación general de los diagramas de interacción	189
Notación básica de los diagramas de colaboración	190
Notación básica de los diagramas de secuencia	195
Capítulo 16: GRASP: diseño de objetos con responsabilidades	201
Responsabilidades y métodos	202
Responsabilidades y los diagramas de interacción	203
Patrones	204
GRASP: Patrones de Principios Generales para Asignar Responsabilidades ..	205
Notación del diagrama de clases UML	206
Experto en Información (o Experto)	207
Creador	211
Bajo Acoplamiento	214
Alta Cohesión	217
Controlador	221
Diseño de objetos y tarjetas CRC	229
Lecturas adicionales	230
Capítulo 17: Modelo de Diseño: realización de los casos de uso con los patrones GRASP	231
Realizaciones de casos de uso	232
Comentarios sobre los artefactos	232
Realizaciones de los casos de uso para la iteración de NuevaEra	236
Diseño de objetos: crearNuevaVenta	236
Diseño de objetos: introducirArticulo	239
Diseño de objetos: finalizarVenta	243
Diseño de objetos: realizarPago	247
Diseño de objetos: ponerEnMarcha	252
Conexión de la capa de UI con la capa del dominio	255
Realizaciones de casos de uso en el UP	258
Resumen	258
Capítulo 18: Modelo de Diseño: determinación de la visibilidad	261
Visibilidad entre objetos	261
Visibilidad	262
Representación de la visibilidad en UML	266
Capítulo 19: Modelo de Diseño: creación de los diagramas de clases de diseño	267
Cuándo crear los DCD	267
Ejemplo de DCD	268
Terminología del DCD y el UP	268
Clases del Modelo de Dominio vs. clases del Modelo de Diseño	269
Creación de un DCD del PDV NuevaEra	269
Notación para los detalles de los miembros	277
DCD, dibujo y herramientas CASE	278
DCD en el UP	278
Artefactos del UP	280

Capítulo 20: Modelo de Implementación: transformación de los diseños en código	281
Programación y el proceso de desarrollo	282
Transformación de los diseños en código	284
Creación de las definiciones de las clases a partir de los DCDs	284
Creación de métodos a partir de los diagramas de interacción	287
Clases contenedoras/colecciones en el código	288
Manejo de excepciones y de errores	289
Definición del método Venta--crearLineaDeVenta	289
Orden de implementación	290
Programar probando primero	291
Resumen de la transformación de los diseños en código	292
Introducción a la solución del programa	292
Parte 4: Elaboración en la Iteración 2	
Capítulo 21: La Iteración 2 y sus requisitos	299
Énfasis de la Iteración 2: diseño de objetos y patrones	299
De la Iteración 1 a la 2	300
Requisitos de la Iteración 2	301
Refinamiento de los artefactos orientados al análisis en esta iteración	303
Capítulo 22: GRASP: más patrones para asignar responsabilidades	305
Polimorfismo	306
Fabricación Pura	308
Indirección	312
Variaciones Protegidas	313
Capítulo 23: Diseño de las realizaciones de casos de uso con los patrones de diseño GoF	321
Adaptador (GoF)	322
Descubrimientos del “análisis” durante el diseño: Modelo del Dominio	324
Factoría (GoF)	326
Singleton (GoF)	328
Conclusiones del problema de los servicios externos con diversas interfaces	331
Estrategia (GoF)	332
Composite (GoF) y otros principios de diseño	337
Fachada (GoF)	346
Observador/Publicar-Suscribir/Modelo de Delegación de Eventos (GoF)	348
Conclusión	356
Lecturas adicionales	356
Parte 5: Elaboración en la Iteración 3	
Capítulo 24: La Iteración 3 y sus requisitos	359
Requisitos de la Iteración 3	359
Énfasis de la Iteración 3	359

Capítulo 25: Relaciones entre casos de uso	361
La relación de inclusión (<i>include</i>)	362
Terminología: casos de uso concretos, abstractos, base y adicional	364
La relación de extensión (<i>extend</i>)	365
La relación de generalización (<i>generalize</i>)	366
Diagramas de casos de uso	367
Capítulo 26: Modelado de la generalización	369
Nuevos conceptos para el Modelo del Dominio	369
Generalización	371
Definición de superclases y subclases conceptuales	373
Cuándo definir una clase conceptual	375
Cuándo definir una superclase conceptual	377
Jerarquías de clases conceptuales del PDV NuevaEra	377
Clases conceptuales abstractas	380
Modelado de los cambios de estado	381
Jerarquías de clases y herencia en el software	382
Capítulo 27: Refinamiento del modelo del dominio	383
Clases asociación	383
Agregación y composición	385
Intervalos de tiempo y precios de los productos: arreglar un “error” de la Iteración 1	390
Nombres de los roles de asociación	390
Roles como conceptos vs. roles en asociaciones	391
Elementos derivados	392
Asociaciones calificadas	393
Asociaciones reflexivas	394
Elementos ordenados	394
Utilización de paquetes para ordenar el Modelo del Dominio	394
Capítulo 28: Añadir nuevos DSSs y contratos	401
Nuevos diagramas de secuencia del sistema	401
Nuevas operaciones del sistema	403
Nuevos contratos de operaciones del sistema	404
Capítulo 29: Modelado del comportamiento con diagramas de estados	407
Eventos, estados y transiciones	407
Diagramas de estados	408
¿Diagramas de estados en el UP?	409
Diagramas de estados de casos de uso	409
Diagramas de estados de casos de uso para la aplicación del PDV	410
Clases que se benefician de los diagramas de estados	411
Representación de eventos externos e internos	412
Notación adicional de los diagramas de estados	413
Lecturas adicionales	415

Capítulo 30: Diseño de la arquitectura lógica con patrones	417
Arquitectura del software	418
Patrón de arquitectura: Capas (<i>Layers</i>)	420
Principio de Separación Modelo-Vista	440
Lecturas adicionales	442
Capítulo 31: Organización de los paquetes de los modelos de diseño e implementación	443
Guías para la organización de paquetes	444
Notación adicional de los paquetes en UML	450
Lecturas adicionales	450
Capítulo 32: Introducción al análisis arquitectural y el SAD	451
Análisis arquitectural	452
Tipos y vistas de la arquitectura	454
La ciencia: identificación y análisis de los factores de la arquitectura	454
Ejemplo: tabla de factores parcial de la arquitectura del PDV NuevaEra	457
El arte: resolución de los factores de la arquitectura	459
Resumen de los temas del análisis arquitectural	466
Análisis arquitectural en el UP	467
Lecturas adicionales	471
Capítulo 33: Diseño de más realizaciones de casos de uso con objetos y patrones	473
Mantenimiento de los servicios ante los fallos mediante servicios locales; rendimiento con el almacenamiento local	473
Manejo de fallos	479
Mantenimiento de los servicios ante los fallos mediante un Proxy (GoF)	484
Diseño para los requisitos no funcionales o de calidad	488
Acceso a los dispositivos físicos externos con adaptadores; comprar vs. construir	488
Factoría Abstracta (GoF) para familias de objetos relacionados	490
Gestión de pagos con Polimorfismo y Hacerlo Yo Mismo	493
Conclusión	498
Capítulo 34: Diseño de un framework de persistencia con patrones ..	501
El problema: objetos persistentes	502
La solución: un servicio de persistencia a partir de un framework de persistencia	502
Frameworks	503
Requisitos para el servicio y framework de persistencia	503
Ideas claves	504
Patrón: Representación de Objetos como Tablas	504
Perfil (<i>Profile</i>) de modelado de datos en UML	505
Patrón: Identificador de Objeto	505
Acceso al servicio de persistencia con una Fachada	506
Correspondencia de los objetos: patrón Conversor (<i>Mapper</i>) de Base de Datos o Intermediario (<i>Broker</i>) de Base de Datos	507

Diseño del framework con el patrón Método Plantilla	509
Materialización con el patrón Método Plantilla	510
Configuración de conversores con una FactoriaDeConversores	515
Patrón: Gestión de Caché	515
Reunir y ocultar sentencias SQL en una clase	516
Estados transaccionales y el patrón Estado	517
Diseño de una transacción con el Patrón Command	520
Materialización perezosa con un Proxy Virtual	522
Cómo representar las relaciones en tablas	524
Superclase ObjetoPersistente y separación de intereses	525
Cuestiones sin resolver	526

Parte 6: Temas especiales

Capítulo 35: Sobre el dibujo de diagramas y las herramientas	529
Diseño especulativo y razonamiento visual	529
Sugerencias para dibujar los diagramas de UML en el proceso de desarrollo	530
Herramientas y características de ejemplo	533
Ejemplo dos	534
Capítulo 36: Introducción a cuestiones relacionadas con la planificación iterativa y el proyecto	537
Priorización de los requisitos	538
Priorización de los riesgos del proyecto	541
Planificación adaptable vs. predictiva	541
Planes de Fase y de Iteración	543
Plan de Iteración: ¿qué hacemos en la siguiente iteración?	543
Traza de los requisitos a través de las iteraciones	544
La (in)validez de las primeras estimaciones	546
Organización de los artefactos del proyecto	547
Algunas cuestiones de la planificación de la iteración del equipo	548
No se entendió la planificación en el UP cuando.....	549
Lecturas adicionales	550
Capítulo 37: Comentarios acerca del desarrollo iterativo y el UP	551
Buenas prácticas y conceptos del UP adicionales	551
Las fases de construcción y transición	553
Otras prácticas interesantes	554
Motivos para fijar la duración de una iteración	555
El ciclo de vida secuencial en “cascada”	555
Ingeniería de usabilidad y diseño de interfaces de usuario	561
El Modelo de Análisis del UP	561
El producto del RUP	562
Los desafíos y mitos de la reutilización	563

XVIII CONTENIDO

Capítulo 38: Más notación UML	565
Notación general	565
Diagramas de implementación	566
Clase plantilla (parametrizada, genérica)	567
Diagramas de actividades	567
Bibliografía	571
Glosario	577
Índice alfabético	583

PRESENTACIÓN

Programar es divertido, pero desarrollar software de calidad es difícil. Entre las ideas espléndidas, los requisitos o la “visión”, y un producto software funcionando, hay mucho más que programar. El análisis y el diseño que definen cómo solucionar el problema, qué programar, y la expresión de este diseño de forma que sea fácil de comunicar, revisar, implementar y evolucionar constituyen la parte central de este libro. Esto es lo que aprenderás.

El Lenguaje Unificado de Modelado (UML) se ha convertido en el lenguaje aceptado universalmente para los planos del diseño software. UML es el lenguaje visual utilizado a lo largo de este libro para ilustrar las ideas de diseño, poniendo énfasis en cómo aplican los desarrolladores realmente los elementos UML utilizados con más frecuencia, más que en características oscuras del lenguaje.

La importancia de los patrones en la creación de sistemas complejos es reconocida desde hace tiempo en otras disciplinas. Los patrones de diseño software son los que nos permiten describir fragmentos de diseño y reutilizar ideas de diseño, ayudando a beneficiarse de la experiencia de otros. Los patrones dan nombre y forma a heurísticas abstractas, reglas y buenas prácticas de técnicas orientadas a objetos. Ningún ingeniero razonable quiere partir de una pizarra en blanco, y este libro ofrece una paleta de patrones de diseño que pueden utilizarse fácilmente.

Pero el diseño de software parece un poco árido y misterioso cuando no se presenta en el contexto de un proceso de ingeniería del software. Y sobre este tema, estoy encantado de que para su segunda edición, Craig Larman haya elegido adoptar e introducir el Proceso Unificado, mostrando cómo puede aplicarse de un modo relativamente simple y poco ceremonioso. Al presentar el caso de estudio a través de un proceso centrado en la arquitectura, dirigido por el riesgo e iterativo, los consejos de Craig tienen un contexto realista; expone la dinámica de lo que ocurre en realidad en el desarrollo de software, y muestra las fuerzas externas que entran en juego. Las actividades del diseño están conectadas a otras tareas, y ya no aparecen como actividades puramente cerebrales de transformaciones sistemáticas o intuición creativa. Y Craig y yo estamos convencidos de los beneficios de un desarrollo iterativo, que verás ilustrado con todo detalle a lo largo del libro.

Así es que para mí, este libro tiene la mezcla correcta de ingredientes. Aprenderás un método sistemático para abordar el Análisis y Diseño Orientado a Objetos (A/DOO) de un gran profesor, un metodologista brillante, y un “gurú de la OO” que lo ha enseñado a miles de personas alrededor del mundo. Craig describe el método en el contexto del Pro-

ceso Unificado. Presenta gradualmente patrones de diseño más sofisticados —esto hará el libro muy útil y fácil de usar cuando te enfrentes con retos de diseño del mundo real. Y utiliza la notación más ampliamente aceptada.

Me siento honrado de haber tenido la oportunidad de trabajar directamente con el autor de este importante libro. Disfruté leyendo la primera edición, y me encantó que me pidiera que revisara el borrador de su segunda edición. Nos encontramos varias veces e intercambiamos muchos correos electrónicos. He aprendido mucho de Craig, incluso sobre nuestro propio proceso de trabajo en el Proceso Unificado y cómo mejorarlo y aplicarlo en varios contextos organizacionales. Estoy seguro de que aprenderás mucho, bastante, al leer este libro, incluso si ya estás familiarizado con el A/DOO. Y, como yo, a menudo volverás a él, para refrescar tu memoria o para comprender mejor las explicaciones y experiencias de Craig.

En un proceso iterativo, el resultado de la segunda iteración mejora la primera. Similarmente, la escritura madura, supongo; incluso si tienes la primera edición, disfrutarás y te beneficiarás de la segunda.

¡Feliz lectura!

*Philippe Kruchten
Rational Fellow
Rational Software Canada
Vancouver, BC*

PRÓLOGO

¡Gracias por leer este libro! Ésta es una introducción práctica al análisis y diseño orientado a objetos (A/DOO), y a aspectos relacionados de desarrollo iterativo. Estoy agradecido de que la primera edición fuese recibida por todo el mundo como una introducción sencilla al A/DOO, traducida a muchos idiomas. Por tanto, esta segunda edición refina y se construye sobre el contenido de la primera, más que reemplazarla. Quiero dar las gracias sinceramente a todos los lectores de la primera edición.

Aquí tienes los beneficios que te proporcionará el libro.

Diseñar sistemas
de objetos robustos
y de fácil
mantenimiento

Seguir un mapa
a través de los
requisitos, análisis,
diseño
y codificación

Usar UML para
ilustrar los modelos
de análisis y diseño

Mejorar los diseños
aplicando los
patrones de diseño
GRASP y
los de la "pandilla
de los cuatro"

Aprendizaje
eficiente
siguiendo una
presentación
refinada

Aprendizaje
mediante
un ejercicio
realista

Primero, el uso de la tecnología de objetos ha proliferado en el desarrollo de software, y el dominio del A/DOO es crítico para crear sistemas de objetos robustos y de fácil mantenimiento.

Segundo, si eres nuevo en el A/DOO, te preguntarás comprensiblemente cómo avanzar por este tema tan complejo; este libro presenta un mapa bien definido —el Proceso Unificado— de manera que te puedas mover en un proceso paso a paso, desde los requisitos al código.

Tercero, el Lenguaje Unificado de Modelado (UML) ha emergido como la notación estándar para el modelado; por tanto, te resultará útil familiarizarte con él. Este libro enseña las técnicas del A/DOO utilizando la notación UML.

Cuarto, los patrones de diseño comunican los estilos y soluciones consideradas como “buenas prácticas”, que los expertos en el diseño orientado a objetos utilizan para la creación de sistemas. En este libro aprenderás a aplicar patrones de diseño, incluyendo los populares patrones de la “pandilla de los cuatro” (*gang-of-four*) y los patrones GRASP que comunican los principios fundamentales de asignación de responsabilidades en el diseño orientado a objetos. Aprender y aplicar patrones acelerará el dominio del análisis y el diseño.

Quinto, la estructura y el enfoque del libro se basa en años de experiencia en la enseñanza y asesoramiento a miles de personas en el arte del A/DOO. Refleja esa experiencia proporcionando un enfoque eficiente, probado y refinado para aprender la materia de manera que se optimiza tu inversión en leer y aprender.

Sexto, examina de manera exhaustiva un único caso de estudio —para ilustrar de manera realista el proceso de A/DOO completo—, y examina en profundidad detalles espinosos del problema; es un ejercicio realista.

Traducción a código **Séptimo**, muestra cómo obtener código Java a partir de los artefactos del diseño de objetos.

Diseño de una arquitectura de capas **Octavo**, explica cómo diseñar una arquitectura de capas y relaciona la capa de interfaz gráfica de usuario con las capas del dominio y servicios técnicos.

Diseño de un framework **Por último**, muestra cómo diseñar un framework orientado a objetos aplicándolo a la creación de uno para el almacenamiento persistente en una base de datos.

Objetivos

El objetivo global es:

Ayudar a los estudiantes y a los desarrolladores a crear diseños orientados a objetos mediante la aplicación de un conjunto de principios y heurísticas explicables.

Estudiando y aplicando la información y las técnicas que se presentan aquí, adquirirás experiencia en la comprensión de un problema en términos de sus procesos y conceptos, y en el diseño de una solución utilizando objetos.

A quiénes está dirigido este libro

Este libro es una *introducción* al A/DOO, al análisis de requisitos relacionado, y al desarrollo iterativo, con el Proceso Unificado como ejemplo de un proceso; la intención no es ser un texto avanzado. Va destinado a la siguiente audiencia:

- Desarrolladores y estudiantes con experiencia en un lenguaje de programación orientado a objetos, pero que son nuevos —o relativamente nuevos— en el análisis y diseño orientado a objetos.
- Estudiantes de informática o cursos de ingeniería del software que estudien la tecnología de objetos.
- Aquellos familiarizados con el A/DOO que quieran aprender la notación UML, aplicar patrones, o que quieran mejorar y perfeccionar sus habilidades de análisis y diseño.

Requisitos

Se asumen —y se necesitan— algunos conocimientos previos para aprovechar este libro:

- Conocimiento y experiencia en un lenguaje de programación orientado a objetos como Java, C#, C++ o Smalltalk.
- Conocimiento de los conceptos generales de la tecnología de objetos, como clase, instancia, interfaz, polimorfismo, encapsulación y herencia.

No se definen los conceptos fundamentales de la tecnología de objetos.

Ejemplos Java

En general, el libro presenta ejemplos de código en Java o plantea implementaciones Java, debido a que su uso está muy extendido. Sin embargo, las ideas presentadas son aplicables a la mayoría —si no a todos— los lenguajes orientados a objetos.

Organización del libro

La estrategia global en la organización de este libro es introducir las cuestiones de análisis y diseño en un orden similar al de un proyecto de desarrollo de software a través de una fase de “inicio” (término del Proceso Unificado) seguido por tres iteraciones (ver Figura P.1).

1. Los capítulos de la fase de inicio introducen los fundamentos del análisis de requisitos.
2. La iteración 1 introduce el A/DOO básico y cómo asignar responsabilidades a los objetos.
3. La iteración 2 se centra en el diseño de objetos, especialmente en introducir algunos “patrones de diseño” muy utilizados.
4. La iteración 3 introduce una variedad de temas, como el análisis de la arquitectura y el diseño de frameworks.

El Libro

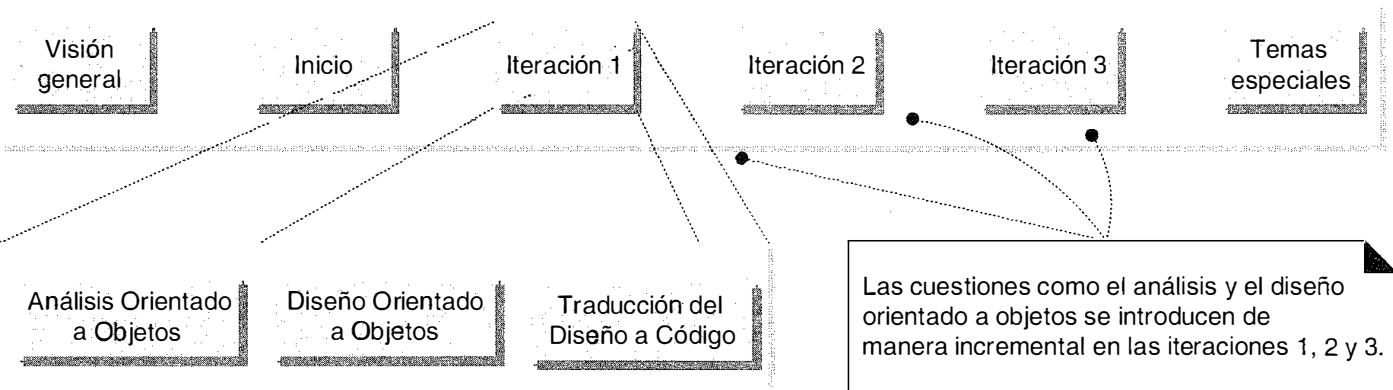


Figura P.1. El libro está organizado como un proyecto de desarrollo.

Recursos web relacionados

- Por favor, consúltese www.craiglarman.com para artículos relacionados con la tecnología de objetos, patrones y procesos.
- Se pueden encontrar recursos para los profesores en www.phptr.com/larman.

Mejoras a la primera edición

Aunque mantiene el mismo núcleo central que la primera edición, la segunda ha refinado muchos aspectos como:

- Se actualizan los casos de uso para seguir el enfoque de [Cockburn01] que ha tenido bastante aceptación.
- Se utiliza el bien conocido Proceso Unificado (UP) como ejemplo de proceso iterativo dentro del cual introducir el A/DOO. Por tanto, todos los artefactos se nombran de acuerdo con la terminología del UP, como Modelo de Dominio.
- Nuevos requisitos en el caso de estudio, que nos encaminan a la tercera iteración.
- Tratamiento actualizado de los patrones de diseño.
- Introducción al análisis arquitectural.
- Introducción de Variaciones Protegidas como un patrón GRASP.
- Balance al 50% entre los diagramas de secuencia y colaboración.
- Actualizada a la notación UML más reciente.
- Discusión de algunos aspectos prácticos de dibujo utilizando pizarras o herramientas CASE para UML.

Reconocimientos

En primer lugar, un agradecimiento muy especial a mis amigos y colegas de Valtech, desarrolladores de objetos a nivel mundial y expertos en el desarrollo iterativo, quienes de alguna manera contribuyeron a apoyar, o revisar el libro, incluyendo a Chris Tarr, Michel Ezran, Tim Snyder, Curtis Hite, Celso González, Pascal Roques, Ken DeLong, Brett Schuchert, Ashley Johnson, Chris Jones, Thomas Liou, Darryl Gebert, Frank Rodorigo, Jean-Yves Hardy, y muchos más que no puedo nombrar.

A Philippe Kruchten por escribir la presentación del libro, revisar y ayudarme de muchas formas.

A Martin Fowler y Alistair Cockburn por muchas discusiones esclarecedoras sobre procesos y diseños, comentarios y revisiones.

A John Vlissides y Cris Kobryn por sus amables comentarios.

A Chelsea Systems y John Gray por ayudarme con algunos requisitos inspirados en su sistema PDV con tecnología Java ChelseaStore.

A Pete Coad y Dave Astels de TogetherSoft por su apoyo.

Muchas gracias a los otros revisores, incluyendo Steve Adolph, Bruce Anderson, Len Bass, Gary K. Evans, Al Goerner, Luke Hohmann, Eric Lefebvre, David Nunn, y Robert J. White.

Gracias a Paul Becker de Prentice-Hall por creer que la primera edición sería un proyecto que merecía la pena, y a Paul Petralia y Patti Guerrieri por dirigir la segunda.

Por último, agradezco especialmente a Graham Glass por abrir una puerta.

Semblanza del autor

Craig Larman trabaja como Director de Procesos para Valtech, una compañía consultora internacional con sucursales en Europa, Asia y América del Norte, especializada en

desarrollo de sistemas de negocio electrónico, tecnologías de objetos y desarrollo iterativo con el Proceso Unificado.

Desde mediados de los ochenta, Craig ha ayudado a miles de desarrolladores a aplicar el análisis, diseño y programación orientada a objetos, y a organizaciones a adoptar las prácticas del desarrollo iterativo.

Después de una carrera fracasada como músico callejero, construyó sistemas en APL, PL/I, y CICS en los setenta. Desde comienzos de los ochenta —después de una completa recuperación— comenzó a interesarse por la inteligencia artificial (teniendo poca propia), procesamiento del lenguaje natural y representación del conocimiento, y construyó sistemas de conocimiento con máquinas Lisp, Prolog y Smalltalk. Toca mal la guitarra eléctrica en su banda *Los Requisitos Cambiantes* (antes se llamaba *Los Requisitos*, pero algunos miembros de la banda cambiaron...) a la que dedica su tiempo libre.

Es licenciado en informática por la Universidad de Simon Fraser en Vancouver, Canadá.

Contacto

Se puede contactar con Craig en clarman@ieee.org y www.craiglarman.com. Serán bienvenidas las preguntas de los lectores y profesores, y peticiones de conferencias, ase-soramiento y consultoría.

Convenciones tipográficas

Esto es un **término nuevo** en una frase. Esto es un nombre de *Clase* o un *método* en una frase. Esto es una referencia a un autor [Bob67]. El operador de resolución de alcance independiente del lenguaje “--” se utiliza para indicar una clase y su método asociado como sigue: *NOMBREClase--NOMBREMetodo*.

Notas de producción

El manuscrito de este libro se creó con Adobe FrameMaker. Todos los dibujos se hicieron con Microsoft Visio. La fuente del texto es New Century Schoolbook. Las imágenes impresas finales se generaron con ficheros PDF utilizando Adobe Acrobat Distilled, a partir del PostScript generado por un controlador AGFA.

Parte 1

INTRODUCCIÓN

ANÁLISIS Y DISEÑO ORIENTADO A OBJETOS

El cambio de enfoque (a patrones) tendrá un efecto profundo y duradero en el modo en el que escribimos los programas.

Ward Cunningham y Ralph Johnson

Objetivos

- Comparar y contrastar análisis y diseño.
 - Definir análisis y diseño orientado a objetos (A/DOO).
 - Ilustrar con un pequeño ejemplo.
-

1.1. Aplicación de UML y patrones en el A/DOO

¿Qué significa tener un buen diseño de objetos? Este libro es una herramienta para ayudar a los desarrolladores y a los estudiantes a aprender las habilidades fundamentales en el análisis y diseño orientado a objetos (A/DOO). Estas técnicas son esenciales para la creación de software de fácil mantenimiento, robusto y bien definido, utilizando las tecnologías y lenguajes de objetos como Java, C++, Smalltalk y C#.

El proverbio “tener un martillo no te hace un arquitecto” es especialmente cierto con respecto a la tecnología de objetos. Conocer un lenguaje orientado a objetos (como Java) es necesario pero un primer paso insuficiente para crear sistemas de objetos. También es preciso conocer cómo “pensar en objetos”.

Esto es una introducción al A/DOO mientras aplicamos el Lenguaje Unificado de Modelado (UML), patrones y el Proceso Unificado. No se debe entender como un texto avanzado; se centra en el dominio de los fundamentos, como el modo de asignar responsabilidades a los objetos, la notación UML utilizada con más frecuencia, y patrones de

Utilización de UML

diseño comunes. Al mismo tiempo, principalmente en los capítulos posteriores, el material progresará a unos pocos temas de nivel intermedio, tal como el diseño de frameworks.

El libro no trata únicamente de UML. UML es una notación visual estándar. Tan útil como aprender una notación, hay más cosas orientadas a objetos importantes que aprender; concretamente, cómo pensar en objetos —cómo diseñar sistemas orientados a objetos—. UML no es A/DOO o un método, es simplemente una notación. No es tan útil aprender a hacer diagramas UML sintácticamente correctos y quizás una herramienta CASE para UML, si no se es capaz de crear un diseño excelente o evaluar y mejorar uno existente. Ésta es la habilidad más difícil y valiosa. En consecuencia, este libro es una introducción al diseño de objetos.

Con todo necesitamos un lenguaje para el A/DOO y los “planos de software”, tanto como una herramienta para pensar, como una forma de comunicación con otros. Por tanto, explora cómo *aplicar* UML al servicio del A/DOO, y cubre la notación UML utilizada con más frecuencia. Se presta especial atención a ayudar a que la gente aprenda el arte y la ciencia de construir sistemas de objetos, más que en la notación.

Aplicación de patrones y asignación de responsabilidades

¿Cómo deberíamos asignar las **responsabilidades** a las clases de objetos? ¿Cómo deberían interaccionar los objetos? ¿Qué clases deberían hacer qué? Éstas son preguntas claves en el diseño de un sistema. Ciertas soluciones contrastadas a problemas de diseño se pueden expresar (y se han expresado) como principios, heurísticas o **patrones** de buenas prácticas —llamados fórmulas de solución de problemas que codifican principios de diseño ejemplares—. Este libro, al enseñar cómo aplicar patrones, favorece un aprendizaje más rápido y un uso propio de experto de estos estilos de diseño de objetos fundamentales.

Un caso de estudio

Esta introducción al A/DOO se ilustra con un **único caso de estudio** que se sigue a lo largo del libro, profundizando en el análisis y diseño lo suficiente para tener en cuenta, y solucionar, algunos de los horribles detalles que se deben considerar y solucionar en un problema real.

Casos de uso y análisis de requisitos

El A/DOO (y todo el diseño software) está fuertemente relacionado con la actividad que es un requisito previo del **análisis de requisitos**, que incluye escribir **casos de uso**. Por tanto, el caso de estudio comienza con una introducción a este tema, a pesar de que realmente no es orientado a objetos.

Un ejemplo de proceso iterativo, el Proceso Unificado

Dadas muchas posibles actividades desde los requisitos hasta la implementación, ¿cómo debería proceder un desarrollador o un equipo? El análisis de requisitos y el A/DOO requieren que se presenten en el contexto de algún proceso de desarrollo. En este caso, se utiliza como *ejemplo de proceso de desarrollo iterativo* el bien conocido **Proceso Unificado**, en cuyo marco se presentan estos temas. Sin embargo, los temas de análisis y diseño que se cubren son comunes a muchos enfoques, y aprenderlos en el contexto del Proceso Unificado no invalida su aplicabilidad a otros métodos.

En conclusión, este libro ayuda a los estudiantes y desarrolladores:

- A aplicar principios y patrones para crear mejores diseños de objetos.
- A seguir un conjunto de actividades comunes en el análisis y diseño, basado en el Proceso Unificado como un ejemplo.
- A crear los diagramas utilizados con más frecuencia en la notación UML.

Esto se ilustra en el contexto de un único caso de estudio.



Figura 1.1. Temas y habilidades que se cubren.

Muchas otras habilidades son importantes

La construcción de software conlleva innumerables habilidades y pasos más allá del análisis de requisitos, el A/DOO, y la programación orientada a objetos. Por ejemplo, la ingeniería de usabilidad y el diseño de interfaces de usuario son claves para el éxito; de igual modo que el diseño de bases de datos.

Sin embargo, esta introducción se centra en el A/DOO, y no pretende cubrir todas las cuestiones del desarrollo de software. Es una parte de un dibujo más grande.

1.2. Asignación de responsabilidades

Hay muchas posibles actividades y artefactos en una introducción al A/DOO, y un gran número de principios y directrices. Suponga que debemos elegir una única habilidad práctica entre todos los temas aquí expuestos —una habilidad como una “isla desierta”—. ¿Cuál sería?

Una habilidad clave y fundamental en el A/DOO es la asignación cuidadosa de responsabilidades a los componentes software.

¿Por qué? Porque es una actividad que debe efectuarse —o mientras se dibuja un diagrama UML o programando— e influye fuertemente sobre la robustez, mantenimiento y reutilización de los componentes software.

Por supuesto, hay otras habilidades necesarias en el A/DOO, pero en esta introducción se hace hincapié en la asignación de responsabilidades porque suele ser una habi-

lidad que requiere esfuerzo el llegar a dominarla y al mismo tiempo tiene una importancia vital. En un proyecto real, un desarrollador podría no tener la oportunidad de abordar ninguna otra actividad de análisis o diseño —el proceso de desarrollo con “prisas por codificar”—. Pero incluso en esta situación, la asignación de responsabilidades es inevitable.

En consecuencia, los pasos de diseño de este libro se centran en los principios de asignación de responsabilidades.

Se presentan y aplican nueve principios fundamentales en el diseño de objetos y asignación de responsabilidades. Se organizan en una ayuda al aprendizaje denominada patrones GRASP.

1.3. ¿Qué es análisis y diseño?

El **Análisis** pone énfasis en una *investigación* del problema y los requisitos, en vez de ponerlo en una solución. Por ejemplo, si se desea un nuevo sistema de información informatizado para una biblioteca, ¿cómo se utilizará?

“Análisis” es un término amplio, es más adecuado calificarlo, como *análisis de requisitos* (un estudio de los requisitos) o *análisis de objetos* (un estudio de los objetos del dominio).

El **Diseño** pone énfasis en una *solución conceptual* que satisface los requisitos, en vez de ponerlo en la implementación. Por ejemplo, una descripción del esquema de una base de datos y objetos software. Finalmente, los diseños pueden ser implementados.

Como con el análisis, es más apropiado calificar el término como *diseño de objetos* o *diseño de bases de datos*.

El análisis y el diseño se han resumido en la frase *hacer lo correcto (análisis), y hacerlo correcto (diseño)*.

1.4. ¿Qué son el análisis y el diseño orientados a objetos?

Durante el **análisis orientado a objetos**, se presta especial atención a encontrar y describir los objetos —o conceptos— en el dominio del problema. Por ejemplo, en el caso del sistema de información de la biblioteca, algunos de los conceptos son *Libro*, *Biblioteca*, y *Socio*.

Durante el **diseño orientado a objetos**, se presta especial atención a la definición de los objetos software y en cómo colaboran para satisfacer los requisitos. Por ejemplo, en el sistema de la biblioteca, un objeto software *Libro* podría tener un atributo *título* y un método *obtenerCapítulo* (ver Figura 1.2).

Por último, durante la implementación o programación orientada a objetos, los objetos de diseño se implementan, como la clase Java *Libro*.

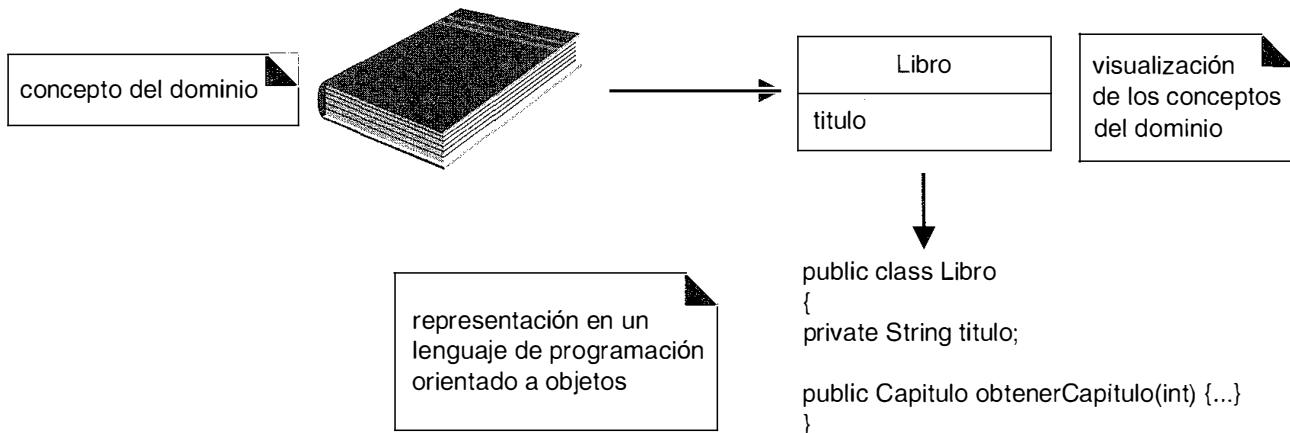


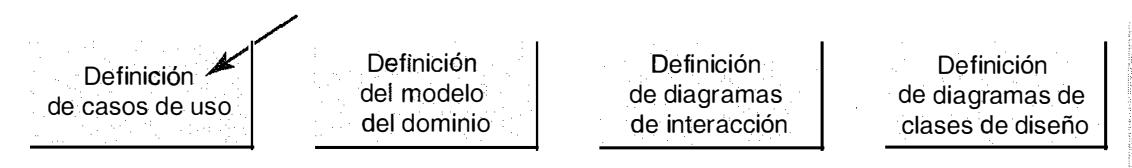
Figura 1.2. La orientación a objetos presta especial atención a la representación de los objetos.

1.5. Un ejemplo

Antes de sumergirnos en los detalles del análisis de requisitos y el A/DOO, esta sección presenta, de modo superficial, unos pocos pasos y diagramas claves, utilizando un ejemplo sencillo —un “juego de dados” en el que un jugador lanza dos dados—. Si el total es siete, gana; en otro caso, pierde.

Definición de los casos de uso

El análisis de requisitos podría incluir una descripción de los procesos del dominio relacionados, que podrían representarse como **casos de uso**.

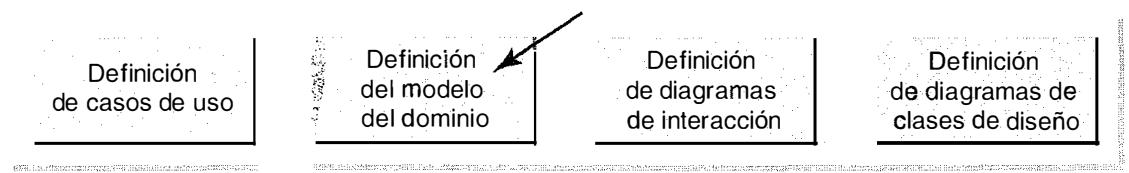


Los casos de uso no son artefactos orientados a objetos —son simplemente historias escritas—. Sin embargo, son una herramienta muy popular en análisis de requisitos y son una parte importante del Proceso Unificado. Por ejemplo, aquí está una versión breve del caso de uso *Jugar una Partida de Dados*:

Jugar una partida de dados: Un jugador recoge y lanza los dados. Si el valor de las caras de los dados suman siete, gana; en otro caso, pierde.

Definición de un modelo del dominio

La finalidad del análisis orientado a objetos es crear una descripción del dominio desde la perspectiva de la clasificación de objetos. Una descomposición del dominio conlleva una identificación de los conceptos, atributos y asociaciones que se consideran significativas. El resultado se puede expresar en un **modelo del dominio**, que se ilustra mediante un conjunto de diagramas que muestran los objetos o conceptos del dominio.



Por ejemplo, la Figura 1.3 muestra un modelo del dominio parcial.

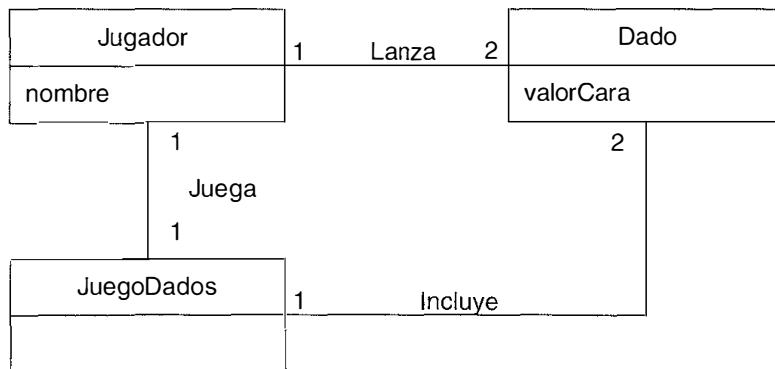


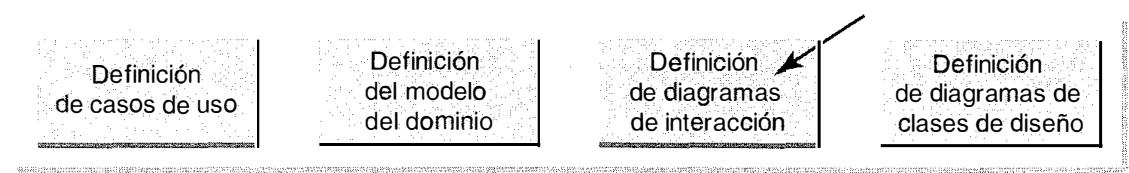
Figura 1.3. Modelo del dominio parcial del juego de dados.

Este modelo ilustra los conceptos importantes *Jugador*, *Dado* y *JuegoDados*, con sus asociaciones y atributos.

Nótese que un modelo del dominio no es una descripción de los objetos software, es una visualización de los conceptos en el dominio del mundo real.

Definición de los diagramas de interacción

La finalidad del diseño orientado a objetos es definir los objetos software y sus colaboraciones. Una notación habitual para ilustrar estas colaboraciones es el **diagrama de interacción**. Muestra el flujo de mensajes entre los objetos software y, por tanto, la invocación de métodos.



Por ejemplo, supongamos que se desea la implementación de un juego de dados. El diagrama de interacción de la Figura 1.4 ilustra los pasos esenciales del juego, enviando mensajes a las clases *JuegoDados* y *Dado*.

Nótese que aunque en el mundo real un *jugador* lanza los dados, en el diseño software el objeto *JuegoDados* “tira” los dados (es decir, envía mensajes a los objetos *Dado*). Los diseños de los objetos software y los programas se inspiran en los dominios del mundo real, pero *no* son modelos directos o simulaciones del mundo real.

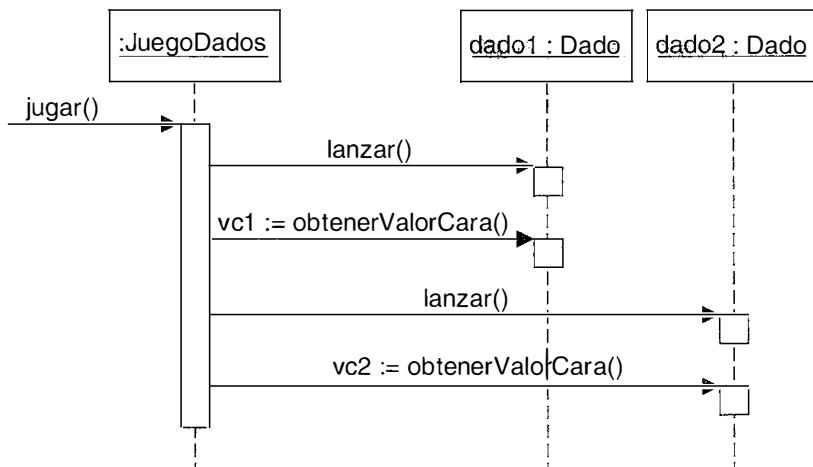
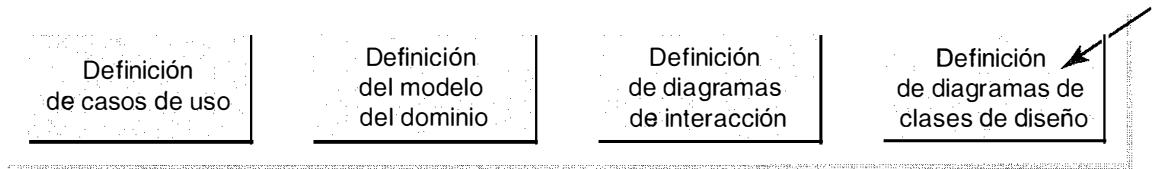


Figura 1.4. Diagrama de interacción que muestra los mensajes entre los objetos software.

Definición de los diagramas de clases de diseño

Además de la vista *dinámica* de las colaboraciones entre los objetos que se muestra mediante los diagramas de interacción, es útil crear una vista *estática* de las definiciones de las clases mediante un **diagrama de clases de diseño**.



Por ejemplo, en el juego de dados, un estudio del diagrama de interacción nos conduce al diagrama de clases de diseño parcial que se muestra en la Figura 1.5. Puesto que se envía el mensaje `jugar` al objeto `JuegoDados`, `JuegoDados` requiere un método `jugar`, mientras la clase `Dado` requiere los métodos `lanzar` y `obtenerValorCara`.

A diferencia del modelo de dominio, este diagrama no muestra conceptos del mundo real, sino clases software.

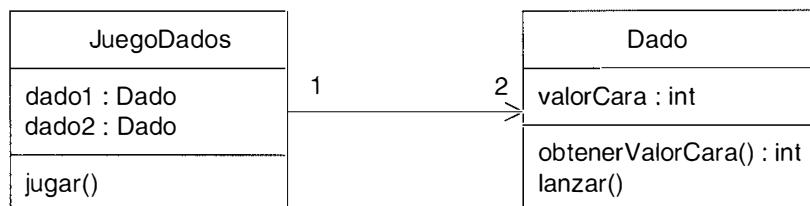


Figura 1.5. Diagrama de clases de diseño parcial.

Resumen

El juego de dados es un problema sencillo, presentado para centrar nuestra atención en unos pocos pasos y artefactos del análisis y diseño. Para no complicar la introducción, no se ha explicado toda la notación UML que se ha mostrado. En los siguientes capítulos se explorará el análisis y el diseño y estos artefactos con más detalle.

1.6. UML

Citando textualmente:

El Lenguaje Unificado de Modelado (UML) es un lenguaje para especificar, visualizar, construir y documentar los artefactos de los sistemas software, así como para el modelado del negocio y otros sistemas no software [OMG01].

UML se ha convertido en la notación visual estándar *de facto* y *de iure* para el modelado orientado a objetos. Comenzó como una iniciativa de Grady Booch y Jim Rumbaugh en 1994 para combinar las notaciones visuales de sus dos populares métodos —los métodos de Booch y OMT (*Object Modeling Technique*)—. Más tarde se les unió Ivar Jacobson, el creador del método Objectory, y el grupo comenzó a ser conocido como los *tres amigos*. Muchos otros contribuyeron a UML, quizás más notablemente Cris Kobryn, que lidera el proceso de refinamiento que todavía continúa.

UML fue adoptado en 1997 como estándar por el OMG (*Object Management Group*, organización que promueve estándares para la industria), y continúa siendo refinado en nuevas versiones.

Este libro no cubre cada pequeño aspecto de UML, que es una notación muy grande (algunos dicen, demasiado grande¹). Se centra en los diagramas que se utilizan con más frecuencia, las características utilizadas más comúnmente en esos diagramas, y la notación central que no es probable que cambie en futuras versiones de UML.

¿Por qué no veremos mucho UML durante unos pocos capítulos?

Éste no es simplemente un libro de notación UML, sino que trata de algo más amplio como es la aplicación de UML, patrones y un proceso iterativo en el contexto de desarrollo de software. UML se aplica principalmente durante el A/DOO, precedido, normalmente, por el análisis de requisitos. Por tanto, los capítulos iniciales presentan una introducción a las cuestiones importantes de casos de uso y análisis de requisitos, seguidos entonces de capítulos de A/DOO y más detalles de UML.

1.7. Lecturas adicionales

UML Distilled, de Martin Fowler, es un libro que ha tenido mucho éxito, y que merece la pena leer, que describe de forma concisa la notación UML más importante.

The Rational Unified Process-An Introduction de Philippe Kruchten es una introducción concisa y muy conocida al Proceso Unificado (y su refinamiento en el Proceso Unificado de Rational).

¹ Entre los objetivos de la versión 2.0 de UML se incluye la simplificación y reducción de la notación. Este libro presenta la parte de UML más utilizada que es muy probable que se mantenga tras la simplificación.

Para una discusión más detallada sobre la notación UML (versión 1.3), merecen la pena *The Unified Modeling Language Reference Manual* y *The Unified Modeling Language User Guide*, de Booch, Jacobson y Rumbaugh. Nótese que la intención de estos textos no era enseñar cómo hacer el modelado de objetos o el A/DOO —son referencias de la notación de los diagramas UML—.

Para una descripción de la versión actual de UML, es necesario la especificación on-line *OMG Unified Modeling Language Specification* en www.omg.org. Los trabajos de revisión de UML y las versiones que van a estar disponibles pronto se pueden encontrar en www.celgent.com/uml.

Hay muchos libros sobre patrones software, pero el libro que más ha influido en el área es el clásico *Design Patterns* de Gamma, Helm, Johnson y Vlissides. Es una lectura verdaderamente obligada para aquellos que estudian el diseño de objetos. Sin embargo, no es un texto introductorio y es mejor que se lea después de conocer bien los fundamentos del diseño y la programación de objetos.

Capítulo 2

DESARROLLO ITERATIVO Y EL PROCESO UNIFICADO

Las personas son más importantes que cualquier proceso.

Buenas personas con un buen proceso siempre actuarán mejor que buenas personas sin procesos.

Grady Booch

Objetivos

- Explicar la motivación del orden y contenido de los capítulos siguientes.
 - Definir un proceso iterativo y adaptable.
 - Definir los conceptos fundamentales del Proceso Unificado.
-

Introducción

El desarrollo iterativo es un enfoque para el desarrollo de software que requiere un entrenamiento y poseer ciertos conocimientos, y juega un papel central en el modo en que se presenta el A/DOO en este libro. El Proceso Unificado es un ejemplo de proceso iterativo para proyectos que utilizan el A/DOO, y da forma a la presentación del libro. En consecuencia, es útil leer este capítulo para clarificar estos conceptos fundamentales y su influencia en la estructura del libro.

Este capítulo resume algunas ideas claves; el lector se puede dirigir al Capítulo 37 para una discusión más detallada del UP y las prácticas de los procesos iterativos.

De manera informal, un **proceso de desarrollo de software** describe un enfoque para la construcción, desarrollo y, posiblemente, mantenimiento del software. El **Proceso Unificado** [JBR99] se ha convertido en un proceso de desarrollo de software de gran

éxito para la construcción de sistemas orientados a objetos. En particular, se ha adoptado ampliamente el **Proceso Unificado de Rational** o **RUP** (*Rational Unified Process*) [Kruchten00], un refinamiento detallado del Proceso Unificado.

El Proceso Unificado (UP) combina las prácticas comúnmente aceptadas como “buenas prácticas”, tales como ciclo de vida iterativo y desarrollo dirigido por el riesgo, en una descripción consistente y bien documentada. Por tanto, se utiliza en este libro como ejemplo de proceso para introducir el A/DOO.

Este libro comienza con una introducción al UP por dos motivos:

1. El UP es un proceso *iterativo*. El desarrollo iterativo es una práctica de gran valor que influye en el modo en el que se introduce el A/DOO en este libro y en cómo debe ser aplicado.
2. Las prácticas del UP proporcionan una estructura organizada de ejemplo para discutir sobre cómo hacer —y cómo aprender— el A/DOO.

Este texto proporciona una introducción al UP, no lo aborda por completo. Se centra en las ideas y artefactos comunes relacionados con una introducción al A/DOO y el análisis de requisitos.

¿Y si no me interesa el UP?

El UP se utiliza como ejemplo de proceso con el que explorar el análisis de requisitos y el A/DOO, puesto que es necesario introducir el tema en el contexto de algún proceso, y el uso del UP (o el refinamiento del RUP) está relativamente bastante extendido. Además, el UP presenta actividades comunes y buenas prácticas. No obstante, las ideas centrales de este libro —como casos de uso y patrones de diseño— son independientes de cualquier proceso particular, y pueden aplicarse a muchos.

2.1. La idea más importante del UP: desarrollo iterativo

El UP fomenta muchas buenas prácticas, pero una destaca sobre las demás: el **desarrollo iterativo**. En este enfoque, el desarrollo se organiza en una serie de mini-proyectos cortos, de duración fija (por ejemplo, cuatro semanas) llamados **iteraciones**; el resultado de cada uno es un sistema que puede ser probado, integrado y ejecutado. Cada iteración incluye sus propias actividades de análisis de requisitos, diseño, implementación y pruebas.

El ciclo de vida iterativo se basa en la ampliación y refinamiento sucesivos del sistema mediante múltiples iteraciones, con retroalimentación cíclica y adaptación como elementos principales que dirigen para converger hacia un sistema adecuado. El sistema crece incrementalmente a lo largo del tiempo, iteración tras iteración, y por ello, este enfoque también se conoce como **desarrollo iterativo e incremental** (ver Figura 2.1).

Las primeras ideas sobre procesos iterativos se conocieron como desarrollo en espiral y desarrollo evolutivo [Boehm88, Gilb88].

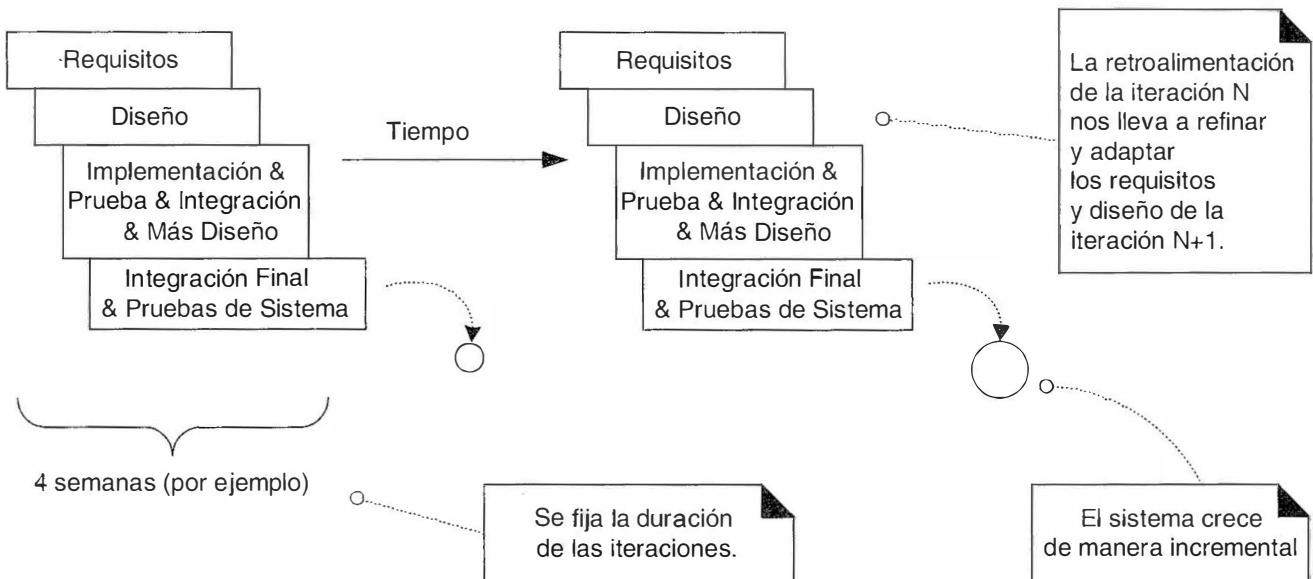


Figura 2.1. Desarrollo iterativo e incremental.

Ejemplo:

Como ejemplo (no como receta), en una iteración de dos semanas, a mitad de camino en un proyecto, quizás el lunes se dedique a distribuir y clarificar las tareas y requisitos de la iteración, mientras que una persona hace ingeniería inversa pasando el código de la última iteración a diagramas UML (mediante una herramienta CASE), e imprime y muestra los diagramas interesantes. El martes se dedica a diseñar por parejas en las pizarras, dibujando diagramas UML imprecisos que se capturan con cámaras digitales, y a escribir algo de pseudocódigo y notas de diseño. Durante los ocho días restantes, se dedica el tiempo a implementar, probar (unidad, aceptación, usabilidad...), ampliar el diseño, integrar, llevar a cabo construcciones diarias, pruebas de sistema y estabilización del sistema parcial. Otras actividades comprenden presentaciones y evaluaciones con el personal involucrado en el proyecto (*stakeholders*¹), y planificación de la siguiente iteración.

Nótese que en este ejemplo no hay prisa por codificar, ni una etapa de diseño más larga de lo que uno piensa que debería ser, en la que se pretende perfeccionar todos los detalles del diseño antes de programar. Se anticipa "un poco" del diseño con modelado basado en diagramas, utilizando dibujos UML realizados con rapidez y a grandes rasgos; quizás los desarrolladores dedican medio día, o un día entero, a trabajar diseñando en parejas.

El resultado de cada iteración es un sistema ejecutable, pero incompleto; no está preparado para ser puesto en producción. El sistema podría no estar listo para su puesta en producción hasta después de muchas iteraciones; por ejemplo, 10 ó 15.

La salida de una iteración *no* es un prototipo experimental o desecharable, y el desarrollo iterativo no es prototipado. Más bien, la salida es un subconjunto con calidad de producción del sistema final.

Aunque, en general, cada iteración aborda nuevos requisitos y amplía el sistema incrementalmente, una iteración podría, ocasionalmente, volver sobre el software que ya existe y mejorarlo; por ejemplo, una iteración podría centrarse en mejorar el rendimiento de un subsistema, en lugar de extenderlo con nuevas características.

¹ *N. del T.:* Este término se refiere a todas las personas que de uno u otro modo están involucradas en un proyecto (directivos, usuarios, analistas, programadores...).

Aceptando los cambios: retroalimentación y adaptación

El subtítulo de un libro que trata el desarrollo iterativo es *Aceptar el Cambio* [Beck00]. Esta frase evoca una aptitud clave del desarrollo iterativo: En lugar de luchar contra el inevitable cambio que ocurre en el desarrollo de software intentando (normalmente sin éxito) especificar, congelar y “firmar” de manera completa y correcta a partir de un conjunto de requisitos fijos y diseñar antes de implementar, el desarrollo iterativo se basa en una aptitud de aceptación del cambio y la adaptación como motores inevitables y, de hecho, esenciales.

Esto no quiere decir que el desarrollo iterativo y el UP fomenten un proceso dirigido por “una adición de características” de manera incontrolada y reactiva. Los siguientes capítulos explorarán cómo el UP llega a un equilibrio entre la necesidad —por un lado— de llegar a un acuerdo y estabilizar un conjunto de requisitos, y —por otro lado— la realidad de los requisitos cambiantes, cuando el personal involucrado clarifica su visión o cambia el mercado.

Cada iteración conlleva la elección de un pequeño conjunto de requisitos y, rápidamente, diseñar, implementar y probar. En las primeras iteraciones, la elección de los requisitos y el diseño podrían no ser exactamente lo que se desea al final. Pero el acto de dar un pequeño paso con rapidez, antes de capturar todos los requisitos y que el diseño completo se haya definido de forma especulativa, nos lleva a una rápida retroalimentación —de los usuarios, desarrolladores y pruebas (tales como pruebas de carga y usabilidad)—.

Tener retroalimentación en una etapa temprana vale su peso en oro; más que las *especulaciones* sobre los requisitos y diseños correctos, la retroalimentación, a partir de la construcción y prueba realista de algo, aporta un conocimiento práctico y crucial, y una oportunidad de modificar o adaptar la comprensión de los requisitos o el diseño. Los usuarios finales tienen la oportunidad de ver rápidamente una parte del sistema y decir: “Sí, esto es lo que pedí, pero ahora que lo pruebo, lo que realmente quiero es algo un poco distinto”². Este proceso de “si... pero” no es un signo de fallo; sino, ciclos estructurados frecuentes y tempranos de “si... peros”, son un modo habilidoso de hacer progresar y descubrir qué es lo que tiene un valor real para el personal involucrado. Si, como se ha mencionado, esto no es un consentimiento, un desarrollo caótico y reactivo en el que los desarrolladores cambian continuamente de dirección —es posible llegar a un término medio—.

Además de clarificar los requisitos, actividades como la prueba de carga probarán si el diseño y la implementación parcial están en el camino correcto, o si en la siguiente iteración, se necesita un cambio en la arquitectura básica. Cuanto antes se resuelvan y *prueben* las decisiones de diseño críticas y arriesgadas mejor —el desarrollo iterativo proporciona los mecanismos para esto—.

En consecuencia, el trabajo se desarrolla a lo largo de una serie de ciclos estructurados de construir-retroalimentar-adaptar. No sorprende que la desviación del sistema del

² O más probable: “¡No entendió lo que quería!”

“verdadero camino” (en términos de sus requisitos y diseño finales) en las primeras iteraciones será mayor que en las últimas. A lo largo del tiempo, el sistema converge hacia este camino, como se ilustra en la Figura 2.2.

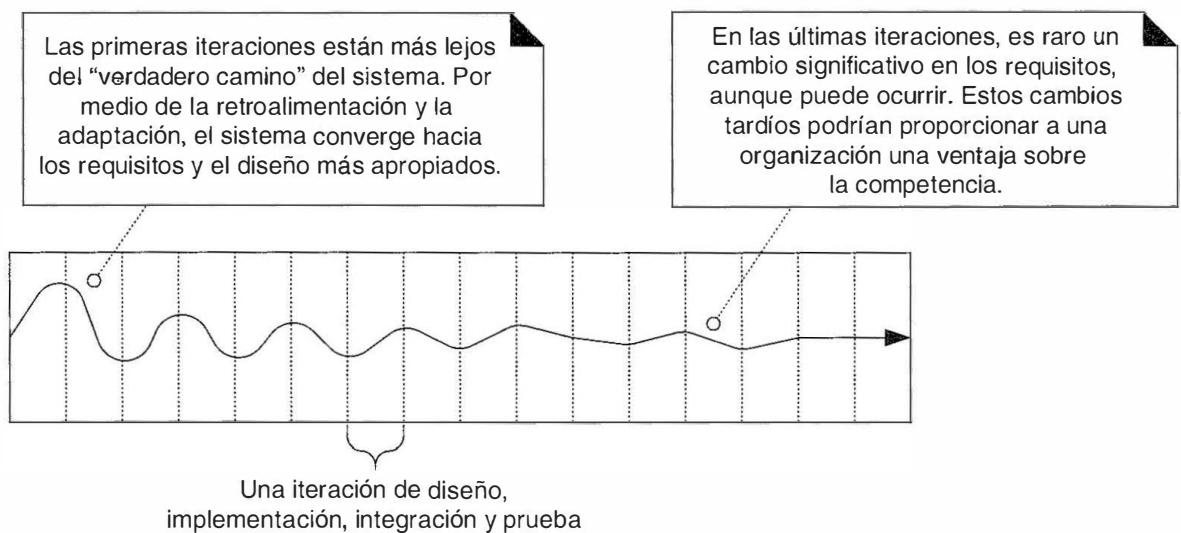


Figura 2.2. La retroalimentación iterativa y la adaptación nos conducen hacia el sistema deseado. La inestabilidad de los requisitos y el diseño disminuyen a lo largo del tiempo.

Beneficios del desarrollo iterativo

Los beneficios del desarrollo iterativo incluyen:

- Mitigación tan pronto como sea posible de riesgos altos (técnicos, requisitos, objetivos, usabilidad y demás)
- Progreso visible en las primeras etapas.
- Una temprana retroalimentación, compromiso de los usuarios y adaptación, que nos lleva a un sistema refinado que se ajusta más a las necesidades reales del personal involucrado.
- Gestión de la complejidad; el equipo no se ve abrumado por la “parálisis del análisis” o pasos muy largos y complejos.
- El conocimiento adquirido en una iteración se puede utilizar metódicamente para mejorar el propio proceso de desarrollo, iteración a iteración.

Longitud de una iteración y fijación de la duración

El UP (y desarrolladores con experiencia en aplicar procesos iterativos) recomiendan que la longitud de una iteración sea de dos a seis semanas. Pasos pequeños, rápida retroalimentación, y adaptación son las ideas fundamentales del desarrollo iterativo; iteraciones largas destruyen la motivación principal del desarrollo iterativo e incrementan el riesgo del proyecto. Menos de dos semanas y es difícil completar el trabajo suficiente para obtener resultados significativos y retroalimentación; más de seis u ocho semanas, y la complejidad se hace bastante abrumadora, y se retrasa la retroalimentación. Con iteraciones muy largas pierde su sentido el desarrollo iterativo. Lo corto es bueno.

Una idea clave es que se **fija la duración** de las iteraciones. Por ejemplo, si se elige que la siguiente iteración dure cuatro semanas, entonces el sistema parcial debería integrarse, probarse y estabilizarse en la fecha planificada —los retrasos son frustrantes—. Si parece que será difícil cumplir con el plazo fijado, la respuesta recomendada es eliminar tareas o requisitos de la iteración, e incluirlos en una iteración posterior, más que retrasar la fecha de terminación prevista. El Capítulo 37 resume los motivos para fijar la duración.

Equipos muy grandes (por ejemplo, varios cientos de desarrolladores) podrían requerir iteraciones de más de seis semanas para compensar los costes fijos de coordinación y comunicación; pero no se recomienda que sea más de tres a seis semanas. Por ejemplo, la exitosa sustitución de los noventa del sistema de control aéreo canadiense, se desarrolló siguiendo un ciclo de vida iterativo y otras prácticas del UP. Necesitó 150 programadores que se organizaron en iteraciones de seis meses³. Pero, nótense que incluso en el caso de una iteración de proyecto global de seis meses, un equipo encargado de un subsistema formado por 10 ó 20 desarrolladores puede dividir su trabajo en una serie de seis iteraciones de un mes.

Una iteración de seis meses es la excepción, en el caso de grandes equipos, no la regla. Reiterando lo dicho, el UP recomienda que la duración de una iteración sea entre dos y seis semanas.

2.2. Conceptos y buenas prácticas del UP adicionales

La idea fundamental para apreciar y utilizar el UP es el desarrollo iterativo, fijando iteraciones cortas, y adaptable.

Otra idea del UP implícita, pero muy importante, es el uso de las tecnologías de objetos, entre las que se encuentra el A/DOO y la programación orientada a objetos.

Algunos conceptos claves y buenas prácticas del UP son:

- Abordar cuestiones de alto riesgo y muy valiosas en las primeras iteraciones
- Involucrar continuamente a los usuarios para evaluación, retroalimentación y requisitos
- Construir en las primeras iteraciones una arquitectura que constituya un núcleo central consistente
- Verificar la calidad continuamente; pruebas muy pronto, con frecuencia y de manera realista
- Aplicar casos de uso
- Modelar software visualmente (con UML)
- Gestionar los requisitos con cuidado
- Manejar peticiones de cambio y gestión de configuraciones

El lector puede acudir al Capítulo 37 para una descripción más detallada de estas prácticas.

³ Philippe Kruchten, quien también dirigió el desarrollo del RUP, trabajó como arquitecto jefe en el proyecto.

2.3. Las fases del UP y términos orientados a la planificación

Un proyecto UP organiza el trabajo y las iteraciones en cuatro fases fundamentales:

1. Inicio: visión aproximada, análisis del negocio, alcance, estimaciones imprecisas.

2. Elaboración: visión refinada, implementación iterativa del núcleo central de la arquitectura, resolución de los riesgos altos, identificación de más requisitos y alcance, estimaciones más realistas.

3. Construcción: implementación iterativa del resto de requisitos de menor riesgo y elementos más fáciles, preparación para el despliegue.

4. Transición: pruebas beta, despliegue.

Estas fases se definen de una manera más completa en los capítulos siguientes.

Esto *no* se corresponde con el antiguo ciclo de vida “en cascada” o secuencial, en el que primero se definían todos los requisitos y, después, se realizaba todo, o la mayoría, del diseño.

La fase de Inicio no es una fase de requisitos; sino una especie de fase de viabilidad, donde se lleva a cabo sólo el estudio suficiente para decidir si continuar o no.

De igual modo, la fase de Elaboración no es la fase de requisitos o de diseño; sino que es una fase donde se implementa, de manera iterativa, la arquitectura que constituye el núcleo central y se mitigan las cuestiones de alto riesgo.

La Figura 2.3 ilustra los términos orientados a la planificación comunes del UP. Nótese que un ciclo de desarrollo (que termina con el lanzamiento de un sistema a producción) se compone de muchas iteraciones.

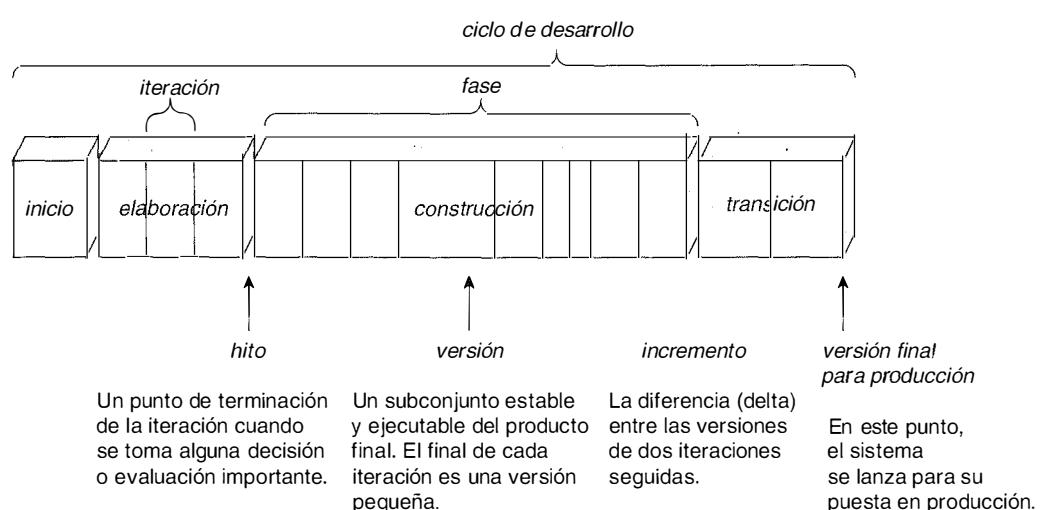


Figura 2.3. Términos orientados a la planificación en el UP.

2.4. Las disciplinas del UP (eran flujos de trabajo)

El UP describe actividades de trabajo, como escribir casos de uso, en **disciplinas** (llamadas originalmente **flujos de trabajo**)⁴. Informalmente, una disciplina es un conjunto de actividades (y artefactos relacionados) en un área determinada, como las actividades en el análisis de requisitos. En el UP, un **artefacto** es el término general para cualquier producto del trabajo: código, gráficos Web, esquema de base de datos, documentos de texto, diagramas, modelos, etcétera.

Hay varias disciplinas en el UP; este libro se centra en algunos artefactos de las siguientes tres:

- **Modelado del Negocio.** Cuando se desarrolla una única aplicación, esto incluye el modelado de los objetos del dominio. Cuando se está haciendo análisis del negocio a gran escala o reingeniería de procesos del negocio, esto incluye el modelado dinámico de los procesos del negocio de toda la empresa.
- **Requisitos.** Análisis de los requisitos para una aplicación, como escritura de casos de uso e identificación de requisitos no funcionales.
- **Diseño.** Todos los aspectos de diseño, incluyendo la arquitectura global, objetos, bases de datos, red y cosas parecidas.

Una lista más extensa de las disciplinas del UP se muestra en la Figura 2.4

En el UP, **Implementación** significa programar y construir el sistema, no despliegue. La disciplina **Entorno** se refiere a establecer las herramientas y adaptar el proceso al proyecto —esto es organizar la herramienta y el entorno del proceso—.

Disciplinas y fases

Como se ilustra en la Figura 2.4, durante una iteración, el trabajo se desarrolla en la mayoría o todas las disciplinas. Sin embargo, el esfuerzo relativo en estas disciplinas cambia a lo largo del tiempo. Las primeras iteraciones, naturalmente, tienden a aplicar un esfuerzo relativo mayor a los requisitos y al diseño, y en las posteriores disminuye, cuando los requisitos y el diseño central se estabilizan, mediante un proceso de retroalimentación y adaptación.

Relacionando esto con las fases del UP (inicio, elaboración...), la Figura 2.5 muestra el esfuerzo relativo de cambio con respecto a las fases; por favor, nótese que son una sugerencia, no es literal. En la elaboración, por ejemplo, las iteraciones tienden a tener un nivel relativamente alto de trabajo de requisitos y diseño, aunque, sin ninguna duda, también algo de implementación. Durante la construcción, se le da una importancia mayor a la implementación y más ligera al análisis de requisitos.

⁴ En el 2001, el antiguo término del UP “flujo de trabajo” se sustituyó por el nuevo término “disciplina” para armonizar con un esfuerzo de estandarización internacional denominado OMG SPEM; debido al significado anterior en el UP, muchos continúan utilizando el término flujo de trabajo para referirse a disciplina, aunque no es estrictamente correcto. El término “flujo de trabajo” comenzó a tener un significado nuevo pero ligeramente distinto en el UP: en un proyecto particular, es una secuencia particular de actividades (quizás *entre* disciplinas)—un flujo de trabajo—.

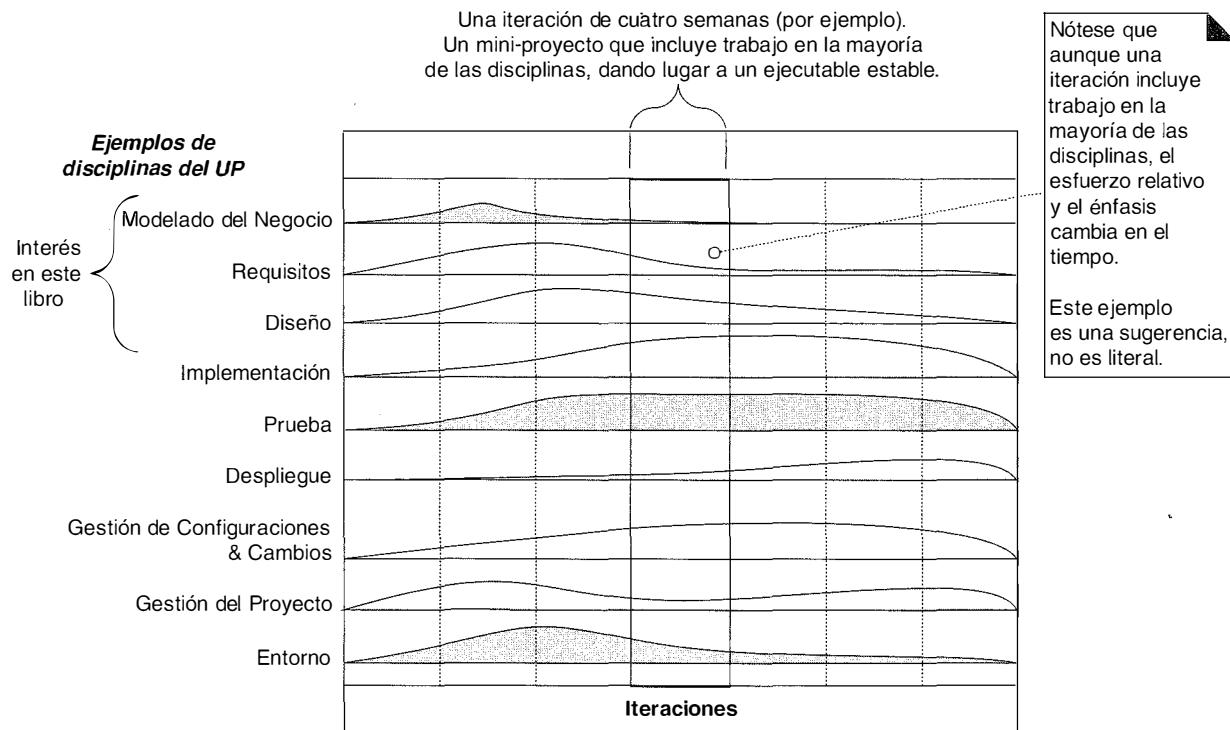
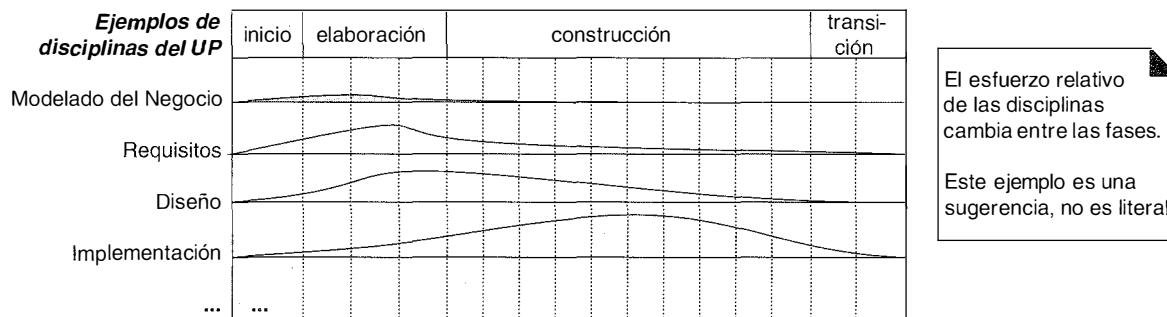
Figura 2.4. Disciplinas del UP⁵.

Figura 2.5. Disciplinas y fases.

Estructura del libro y fases y disciplinas del UP

Con respecto a las fases y disciplinas, ¿cuál es el enfoque del caso de estudio?

Respuesta:

El caso de estudio presta una atención especial a las fases de inicio y elaboración. Se centra en algunos artefactos de las disciplinas del Modelado del Negocio, Requisitos y Diseño, donde se aplican, fundamentalmente, el análisis de requisitos, el A/DOO, los patrones y UML.

⁵ Diagrama adaptado del producto RUP.

Los primeros capítulos presentan las actividades de la fase de inicio; los capítulos posteriores exploran varias iteraciones en la fase de elaboración. La siguiente lista y la Figura 2.6 describen la organización con respecto a las fases del UP.

1. Los capítulos de la fase de inicio introducen los temas esenciales del análisis de requisitos.
2. La iteración 1 presenta los fundamentos del A/DOO y cómo asignar responsabilidades a los objetos.
3. La iteración 2 está enfocada al diseño de objetos, en concreto a introducir algunos de los “patrones de diseño” más utilizados.
4. La iteración 3 presenta una variedad de temas, como el análisis de la arquitectura y el diseño de frameworks.

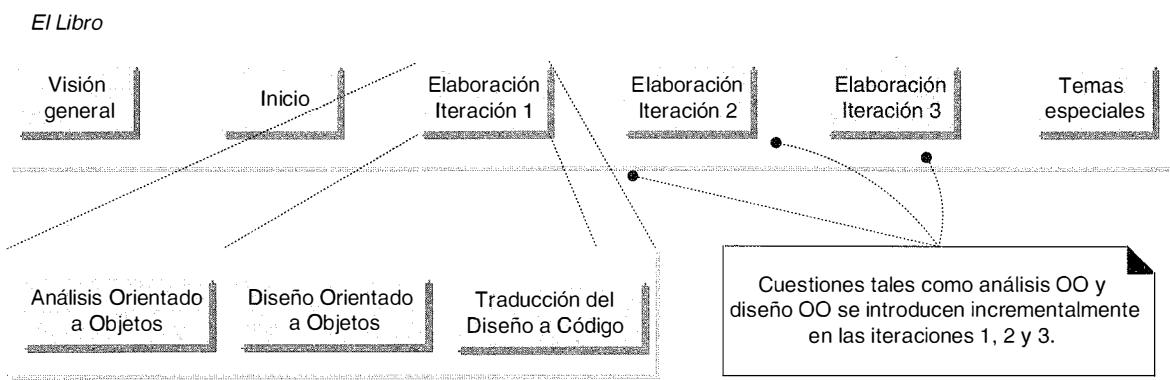


Figura 2.6. La organización del libro está relacionada con las fases e iteraciones del UP.

2.5. Adaptación del proceso y el Marco de Desarrollo

Artefactos opcionales

Algunas prácticas y principios del UP deben seguirse siempre, como el desarrollo iterativo y dirigido por el riesgo, y la verificación continua de la calidad.

Sin embargo, es importante saber que en el UP todas las actividades y artefactos (modelos, diagramas, documentos...) son *opcionales* —bueno, ¡quizás el código no!—. El conjunto de artefactos posibles del UP debería entenderse como un conjunto de medicinas en una farmacia. Exactamente igual que uno no toma medicinas indiscriminadamente, sino que las elige según la dolencia, en un proyecto UP, un equipo debería seleccionar un pequeño subconjunto de artefactos que sirvan para tratar sus problemas y necesidades particulares. En general, centrarse en un *pequeño* conjunto de artefactos que demuestran tener un gran valor práctico.

El Marco de Desarrollo

La elección de los artefactos del UP para un proyecto podría recogerse en un documento breve denominado **Marco de Desarrollo** (un artefacto en la disciplina Entorno). Por

ejemplo, la Tabla 2.1 podría ser el Marco de Desarrollo que describe los artefactos para el caso de estudio “Proyecto NuevaEra” presentado en este libro.

Los capítulos siguientes describen la creación de algunos de estos artefactos como el Modelo del Dominio, Modelo de Casos de Uso y el Modelo de Diseño.

Los artefactos de ejemplo que se presentan en este caso de estudio de ningún modo son suficientes, o adecuados, para todos los proyectos. Por ejemplo, un sistema de control de una máquina encontraría útil realizar muchos diagramas de estados. Un sistema de comercio electrónico basado en el web podría centrar su atención en los prototipos de interfaz de usuario. Un proyecto de desarrollo nuevo, en un campo en el que no se tenga experiencia tendría necesidades muy diferentes, en cuanto a los artefactos de diseño, de las de un proyecto de integración de sistemas.

Tabla 2.1 Ejemplo de Marco de Desarrollo de artefactos UP. c-comenzar; r-refinar

Disciplina	Artefacto Iteración →	Inicio I1	Elab. E1...En	Const. C1...Cn	Trans. T1...T2
Modelado del Negocio	Modelo del Dominio	c			
Requisitos	Modelo de Casos de Uso Visión Especificación Complementaria Glosario	c c c c	r r r r		
Diseño	Modelo de Diseño Documento de Arquitectura SW Modelo de Datos		c c c	r r	
Implementación	Modelo de Implementación		c	r	r
Gestión del Proyecto	Plan de Desarrollo SW	c	r	r	r
Pruebas	Modelo de Pruebas		c	r	
Entorno	Marco de Desarrollo	c	r		

2.6. El UP ágil

Los metodólogos distinguen entre procesos pesados y ligeros, y procesos predictivos y adaptables. Un **proceso pesado** es un término peyorativo, que pretende sugerir un proceso con las siguientes cualidades [Fowler00]:

- Muchos artefactos creados en un ambiente burocrático.
- Rigididad y control.
- Planificación detallada, muy larga y elaborada.
- Predictivo más que adaptable.

Un **proceso predictivo** es aquel que intenta planificar y predecir en detalle las actividades y asignación de recursos (personal) en un intervalo relativamente largo de tiempo, tal como la totalidad de un proyecto. Los procesos predictivos normalmente si-

guen un ciclo de vida en “cascada” o secuencial —primero, definición de todos los requisitos; segundo, definición del diseño detallado; y tercero, implementación—. Frente a esto, un **proceso adaptable** es aquel que acepta el cambio como motor inevitable y fomenta la adaptación flexible; normalmente siguen un ciclo de vida iterativo. Un **proceso ágil** implica un proceso adaptable y ligero, listo para responder rápidamente a las necesidades cambiantes.

La intención de los autores del UP no era que fuese pesado o predictivo, aunque su amplio conjunto de actividades y artefactos opcionales, comprensiblemente, ha llevado a algunos a tener esta impresión. Más bien, la intención fue que se adoptara y aplicara con el espíritu de un proceso ágil —**UP ágil**—. Algunos ejemplos de cómo se pone esto en práctica:

- Optar por un conjunto pequeño de actividades y artefactos del UP. Algunos proyectos se beneficiarán más que otros, pero, en general, manténgalo simple.
- Puesto que el UP es iterativo, no se completan los requisitos y diseños antes de la implementación. Surgen de modo adaptable a lo largo de una serie de iteraciones, basadas en la retroalimentación.
- No hay un plan *detallado* para todo el proyecto. Hay un plan de alto nivel (denominado el **Plan de Fase**) que estima la fecha de terminación del proyecto y otros hitos importantes, pero no detalla los pasos de grano fino de estos hitos. Un plan detallado (llamado **Plan de Iteración**) sólo planifica con gran detalle una iteración por adelantado. La planificación detallada se lleva a cabo de manera adaptable de iteración en iteración. En el Capítulo 36 se pueden encontrar algunos comentarios sobre la planificación de proyectos iterativos y la justificación de este enfoque.

El caso de estudio hace hincapié en un número relativamente pequeño de artefactos y en el desarrollo iterativo, en el espíritu de un UP ágil.

2.7. El ciclo de vida “en cascada” secuencial

A diferencia del ciclo de vida iterativo del UP, una antigua alternativa era el ciclo de vida “en cascada”, lineal o secuencial [Royce70]. En su acepción habitual, definía los pasos más o menos de la siguiente forma:

1. Determinar, registrar y acordar un conjunto de requisitos completo y fijo.
2. Diseñar un sistema basado en estos requisitos.
3. Implementar en base al diseño.

Un estudio de dos años, presentado en el *MIT Sloan Management Review*, sobre los proyectos software con éxito, identificó cuatro factores comunes para el éxito; el primero de la lista era el desarrollo iterativo, en lugar de un proceso en cascada [MacCormack01]⁶.

⁶ Los otros eran: 2) al menos una incorporación diaria de nuevo código para la construcción del sistema completo, y rápida retroalimentación de los cambios del diseño (a través de las pruebas); 3) equipo con experiencia en expedir múltiples productos; y 4) centrarse pronto en la construcción y prueba de una arquitectura consistente. Tres de estos cuatro factores son prácticas explícitas en el UP.

Una descripción breve de estos problemas, y cómo pueden mitigarse mediante el desarrollo iterativo, se presentará en el Capítulo 37.

2.8. No se entendió el UP cuando...

Aquí presentamos algunos signos que indican que no se ha entendido lo que significa adoptar el UP y el desarrollo iterativo con la idea de agilidad que se pretende en el UP.

- Se piensa que inicio = requisitos, elaboración = diseño, y construcción = implementación (esto es, imponiendo un ciclo de vida en espiral sobre el UP).
- Se piensa que el objetivo de la elaboración es definir modelos de manera completa y cuidadosa, los cuales se traducen a código durante la construcción.
- Se intenta definir la mayoría de los requisitos antes de comenzar el diseño o la implementación.
- Se intenta definir la mayoría del diseño antes de comenzar a implementar; se intenta definir completamente y acordar una arquitectura antes de programar y probar iterativamente.
- Se dedica mucho tiempo a realizar trabajo sobre los requisitos y el diseño antes de comenzar a programar.
- Se cree que una iteración adecuada es de cuatro meses de duración, en lugar de cuatro semanas (excluyendo proyectos de cientos de desarrolladores).
- Se piensa que realizar los diagramas UML y las actividades de diseño constituyen el momento para definir diseños y modelos de manera completa y precisa con gran detalle, y se cree que programar es una simple traducción mecánica de éstos en código.
- Se piensa que adoptar el UP significa hacer muchas de las actividades posibles y crear muchos documentos, y se piensa o experimenta el UP como un proceso formal y exigente con muchos pasos que seguir.
- Se intenta planificar un proyecto en detalle desde el principio hasta el final; intenta predecir, de manera especulativa, todas las iteraciones y lo que debería ocurrir en cada una de ellas.
- Se quieren planes y estimaciones creíbles para los proyectos antes de que termine la fase de elaboración.

2.9. Lecturas adicionales

Una introducción al UP y su refinamiento en el RUP, que merece la pena leer es *The Rational Unified Process-An Introduction* de Philippe Kruchten, el arquitecto líder del RUP.

Se puede encontrar una descripción del UP original en *The Unified Software Development Process* de Jacobson, Booch y Rumbaugh. Merece la pena estudiarlo, pero se recomienda primero la introducción de Kruchten puesto que es más pequeña y concisa, y el RUP actualiza y refina el original UP.

Rational Software vende un producto basado en el web con la documentación del RUP on-line, el cual permite conocer con todo detalle todas las actividades y artefactos del RUP y las plantillas para la mayoría de los artefactos. En el Capítulo 37 se presentará una breve discusión. Una organización puede abordar un proyecto UP utilizando únicamente mentores y libros como fuentes de aprendizaje, pero algunos encuentran la documentación del RUP una ayuda útil para el aprendizaje y el proceso.

Las actividades del UP también se describen de manera poco estricta en una serie de libros editados por Ambler y Constantine (por ejemplo, *The Unified Process: Elaboration Phase* [Ambler00]). Estos libros contienen artículos publicados a lo largo de los años en la revista *Software Development*, clasificados en sus respectivas fases y actividades en términos de una taxonomía del UP. Nótese que los artículos no se escribieron originalmente para el UP, aunque, sin lugar a dudas contienen consejos útiles. También se advierte un pequeño error en las series: describen la fase de elaboración del UP como una fase en la que se crean prototipos desechables, de este modo reduce la necesidad para prestar atención a cuidar la programación o el diseño. Esto no es exacto; durante la elaboración se crean diseños y código con calidad de producción (aunque parcial). Ambler reconoce la imprecisión y puede corregirlo en las ediciones siguientes⁷.

Para obtener información sobre otros métodos ágiles se recomienda la serie de libros **Extreme Programming** (XP) [Beck00, BF00, JAH00], tales como *Extreme Programming Explained*. Algunas de las prácticas de XP se mencionan en capítulos posteriores. La mayoría de las prácticas de XP (tales como programar probando primero y desarrollo iterativo) son compatibles —o idénticas— a las prácticas del UP, y recomiendo su adopción en un proyecto UP. Nótese que XP no inventó (ni pidió que se hiciese) el desarrollo iterativo, con iteraciones cortas de duración fija, y adaptable, que ha sido una práctica del UP y otros métodos iterativos durante años. Dos diferencias destacables —no es una lista completa— entre el UP y XP son: 1) el UP recomienda escribir de manera incremental los casos de uso y un documento de requisitos no funcionales (XP no); y, 2) el UP recomienda dedicar más tiempo a realizar los diagramas del diseño visual (como medio día o un día entero) cerca del comienzo de una iteración, antes de la programación. Los líderes de XP recomiendan muy poco, unos 30 minutos.

Highsmith justifica el valor del desarrollo iterativo en *Adaptative Software Development* [Highsmith00].

⁷ Ambler, comunicación privada.

Capítulo 3

CASO DE ESTUDIO: EL SISTEMA DE PUNTO DE VENTA NUEVAERA

Pocas cosas son más difíciles de soportar que un buen ejemplo.

Mark Twain

Introducción

Este capítulo describe brevemente el caso de estudio. Si se conoce el dominio del problema, podría saltarse. De hecho, se eligió este problema porque es familiar, pero suficientemente rico, con problemas interesantes de arquitectura y diseño, y, por tanto, nos permite concentrarnos en cómo llevar a cabo el análisis y diseño, en lugar de explicar el problema y el dominio.

3.1. El sistema de punto de venta NuevaEra

El caso de estudio es el sistema de punto de venta (PDV) NuevaEra. En este dominio del problema, en apariencia sencillo, veremos que hay requisitos muy interesantes y problemas de diseño que solucionar. Además, es un problema real; las organizaciones realmente escriben sistemas PDV utilizando la tecnología de objetos.

Un sistema PDV es una aplicación informática utilizada (en parte) para registrar ventas y realizar pagos; normalmente se utiliza en tiendas. Incluye componentes hardware, como un ordenador y un lector de códigos de barras, y software para ejecutar el sistema. Interactúa con varias aplicaciones de servicios, como un servicio de cálculo de impuestos y un control de inventario, de terceras partes. Estos sistemas deben ser, relativamente, tolerantes a fallos; es decir, incluso si los servicios remotos no están disponibles temporalmente (como el sistema de inventario), todavía deben ser capaces de

capturar las ventas y gestionar, al menos, los pagos en efectivo (de manera que no se impida que el negocio funcione de manera adecuada).

Un sistema PDV, progresivamente, debe soportar múltiples y variados terminales e interfaces del lado del cliente, como un terminal con un navegador Web de una arquitectura “cliente delgado” (*thin client*), un ordenador personal normal con una interfaz gráfica de usuario hecha con las clases Swing de Java, entrada de datos mediante una pantalla táctil, PDAs inalámbricos, etcétera.

Además, estamos creando un sistema PDV comercial que se venderá a diferentes clientes con necesidades dispares en términos de procesamiento de reglas del negocio. Cada cliente deseará un conjunto exclusivo de lógica a ejecutar en ciertos puntos predecibles en escenarios de uso del sistema, como cuando se inicia una venta o cuando se añade una nueva línea. Por tanto, necesitaremos un mecanismo para proporcionar esta flexibilidad y personalización.

Utilizando una estrategia de desarrollo iterativo, vamos a realizar las fases de requisitos y análisis, diseño e implementación orientados a objetos.

3.2. Capas arquitectónicas y el énfasis del caso de estudio

Un sistema de información orientado a objetos típico se diseña en función de varias capas arquitectónicas o subsistemas (ver Figura 3.1). A continuación, presentamos algunos ejemplos, no una lista completa:

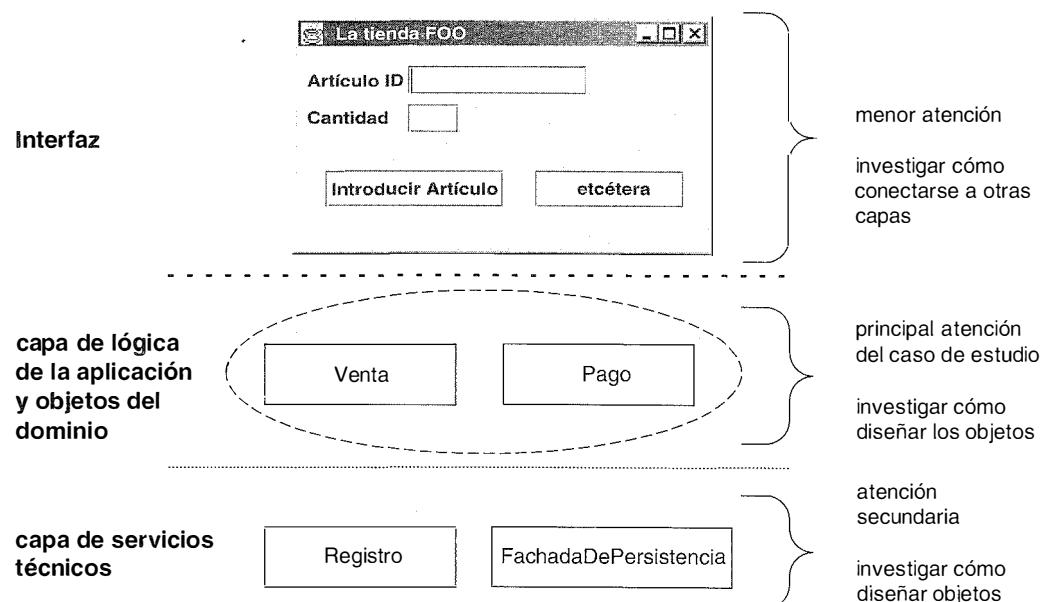


Figura 3.1. Capas y objetos de ejemplo en un sistema orientado a objetos, y el enfoque del caso de estudio.

- **Interfaz de Usuario:** interfaz gráfica; ventanas.
- **Lógica de la aplicación y Objetos del Dominio:** objetos software que representan conceptos del dominio (por ejemplo una clase software denominada *Venta*) que satisfacen los requisitos de la aplicación.
- **Servicios técnicos:** objetos de propósito general y subsistemas que proporcionan servicios técnicos de apoyo, como conexión con una base de datos o registrar los errores. Normalmente, estos servicios son independientes de la aplicación y se pueden reutilizar entre varios sistemas.

El A/DOO es, por lo general, más relevante para modelar la lógica de la aplicación y las capas de servicios técnicos.

El caso de estudio NuevaEra se centra principalmente en los objetos del dominio del problema, asignándoles responsabilidades para satisfacer los requisitos de la aplicación. El diseño orientado objetos también se aplica para crear un subsistema de servicio técnico para interactuar con una base de datos.

En este enfoque de diseño, la capa de interfaz de usuario tiene muy poca responsabilidad; se dice que es *delgada*. Las ventanas *no* contienen código que ejecute o procese la lógica de la aplicación, sino que, las peticiones de realización de tareas se envían a otras capas.

3.3. Estrategia del libro: aprendizaje y desarrollo iterativo

Este libro se organiza para seguir una estrategia de desarrollo iterativo. El A/DOO se aplica al sistema PDV NuevaEra en varias iteraciones; la primera iteración es para determinar algunas funciones básicas. Las iteraciones posteriores expanden la funcionalidad del sistema (ver Figura 3.2). Junto con el desarrollo iterativo, se introduce incrementalmente y de manera iterativa, la *presentación* de las cuestiones del análisis y diseño, la notación UML y los patrones. En la primera iteración, se presenta un núcleo básico de cuestiones de análisis, diseño y notación. La segunda iteración amplía con nuevas ideas, la notación UML y los patrones. Y, de igual modo, la tercera iteración.

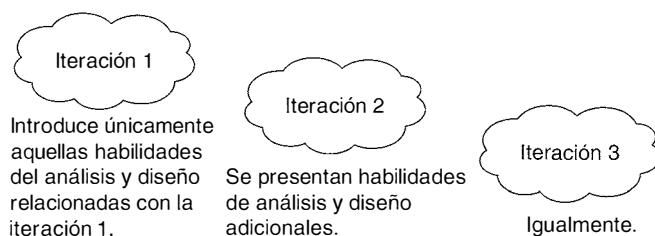


Figura 3.2. El camino de aprendizaje sigue iteraciones.



Parte 2
INICIO

Capítulo 4

INICIO

Le mieux est l'ennemi du bien (Lo mejor es enemigo de lo bueno.)

Voltaire

Objetivos

- Definir la etapa de inicio.
 - Motivar los capítulos siguientes de esta sección.
-

Introducción

Este capítulo define la fase de inicio de un proyecto. Si las ideas del proceso no son del interés del lector, o prefiere centrarse en primer lugar en aprender la actividad práctica principal de esta fase —modelado de casos de uso— entonces puede saltarse este capítulo.

La mayoría de los proyectos requieren una etapa inicial breve en la que se estudian los siguientes tipos de preguntas:

- ¿Cuál es la visión y el análisis del negocio para este proyecto?
- ¿Es viable?
- ¿Comprar y/o construir?
- Estimación aproximada del coste: ¿cuesta 10K-100K o millones de euros?
- ¿Deberíamos abordarlo o no seguir?

Para definir la visión y obtener una estimación (de la que no te puedes fiar) del orden de magnitud es necesario llevar a cabo alguna exploración de los requisitos. Sin embargo, el objetivo de la etapa de inicio no es definir todos los requisitos, o generar una estimación creíble o plan de proyecto. Aún a riesgo de simplificar demasiado, la idea es hacer la investigación justa para formar una opinión racional y justificable del propósito global y la viabilidad del nuevo sistema potencial, y decidir si merece la pena invertir en un estudio más profundo (el objetivo de la fase de elaboración).

Por tanto, la fase de inicio debería ser relativamente corta en la mayoría de los proyectos, una duración de una a unas pocas semanas. De hecho, en muchos proyectos, si la duración es superior a una semana, entonces se pierde la idea fundamental de la etapa de inicio: decidir si merece la pena una investigación seria (durante la elaboración), no llevar a cabo esta investigación.

La fase de inicio en una frase:

Vislumbrar el alcance del producto, visión y análisis del negocio.

El principal problema resuelto en una frase:

¿Está de acuerdo el personal involucrado en la visión del proyecto, y merece la pena invertir en un estudio serio?

4.1. Inicio: una analogía

En el negocio del petróleo, cuando se está considerando una nueva zona, algunos de los pasos que hay que abordar son:

1. Decidir si hay evidencias suficientes o un análisis del negocio que justifique perforaciones de exploración.
2. De ser así, llevar a cabo medidas y perforaciones de exploración.
3. Proporcionar información del alcance y estimación.
4. Pasos adicionales...

La fase de inicio es como el primer paso en esta analogía. En el primer paso, no se predice cuánto petróleo hay, o el coste o esfuerzo para extraerlo. Es prematuro —no hay suficiente información—. Aunque estaría bien ser capaz de responder a las preguntas de “cuánto” y “cuándo” sin el coste de la exploración, en el negocio del petróleo se entiende que no es realista.

En términos del UP, el paso de exploración realista se corresponde con la fase de elaboración. La fase de inicio que la precede es parecida a un estudio de viabilidad para decidir si incluso merece la pena invertir en perforaciones de exploración. Sólo después de la exploración (elaboración) tenemos los datos y los conocimientos para hacer, de algún modo, planes y estimaciones creíbles. Por tanto, en el desarrollo iterativo y el UP, los planes y estimaciones de la fase de inicio no deben considerarse fiables. Simplemente proporcionan una percepción del orden de magnitud del grado de esfuerzo, para ayudar en la decisión de continuar o no.

4.2. La fase de inicio podría ser muy breve

El propósito de la fase de inicio es establecer una visión común inicial de los objetivos del proyecto, determinar si es viable y decidir si merece la pena llevar a cabo algunas investigaciones serias en la fase de elaboración. Si se ha decidido de antemano que el proyecto se hará sin ninguna duda, y es claramente viable (quizás porque el equipo ha desarrollado proyectos parecidos antes), entonces la fase de inicio será especialmente breve. Podría incluir los primeros talleres de requisitos¹, planificación de la primera iteración y, entonces, rápidamente, cambiar a la elaboración.

4.3. ¿Qué artefactos podrían crearse en la fase de inicio?

La Tabla 4.1 presenta un listado de los artefactos comunes de la fase de inicio (o principio de la elaboración) e indica las cuestiones que deben abordarse. Los capítulos siguientes estudiarán algunos de ellos con más detalle, especialmente el Modelo de Ca-

Tabla 4.1. Ejemplo de artefactos de la fase de inicio.

Artefacto [†]	Comentario
Visión y Análisis del Negocio	Describe los objetivos y las restricciones de alto nivel, el análisis del negocio y proporciona un informe para la toma de decisiones.
Modelo de Casos de Uso	Describe los requisitos funcionales y los no funcionales relacionados.
Especificación Complementaria	Describe otros requisitos.
Glosario	Terminología clave del dominio.
Lista de Riesgos & Plan de Gestión del Riesgo	Describe los riesgos del negocio, técnicos, recursos, planificación, y las ideas para mitigarlos o darles respuesta.
Prototipos y pruebas-de-conceptos	Para clarificar la visión y validar las ideas técnicas.
Plan de Iteración	Describe qué hacer en la primera iteración de la elaboración.
Fase Plan de & Plan de Desarrollo de Software	Estimación de poca precisión de la duración y esfuerzo de la fase de elaboración. Herramientas, personas, formación y otros recursos.
Marco de Desarrollo	Una descripción de los pasos del UP y los artefactos adaptados para este proyecto. El UP siempre se debe adaptar al proyecto.

[†] Estos artefactos se completan sólo parcialmente en esta fase. Se refinrarán de manera iterativa en las siguientes iteraciones. El nombre en mayúsculas indica que es un artefacto UP con ese nombre oficial.

¹ *N. del T.*: Se ha traducido “requirements workshop” por “taller de requisitos” para indicar una reunión de discusión sobre requisitos.

sos de Uso. Una idea clave con respecto al desarrollo iterativo es comprender que estos artefactos sólo se completan parcialmente en esta fase, se refinarán en iteraciones posteriores, e incluso no deberían crearse a menos que se considere probable que añadirán valor práctico real. Y, puesto que estamos en el inicio, la investigación y el contenido de los artefactos deberían ser ligeros.

Por ejemplo, el Modelo de Casos de Uso (que se describirá en los capítulos siguientes) podría listar los *nombres* de la mayoría de los casos de uso y actores esperados, pero quizás sólo describiría en detalle el 10% de los casos de uso —hecho con el propósito de desarrollar una visión de alto nivel y sin detalles del alcance, objetivo y riesgos del sistema.

Nótese que en la fase de inicio se podrían realizar algunas tareas de programación con el objeto de crear prototipos de “pruebas de conceptos”, para clarificar unos pocos requisitos mediante (generalmente) prototipos orientados a la interfaz de usuario, y hacer experimentos de programación para cuestiones técnicas críticas.

¿No es eso mucha documentación?

Hay que recordar que los artefactos se deberían considerar opcionales. Se deben elegir sólo aquellos que realmente añadan valor al proyecto, y desecharlos si no se prueba que merezcan la pena.

Lo importante de un artefacto no es el documento o el diagrama en sí mismo, sino el pensamiento, análisis y disposición activa (y entonces se registra, para evitar re-inveniciones o tener que repetir las cosas de palabra). Como dijo el general Eisenhower: “Al preparar una batalla siempre he encontrado que los planes no son útiles, pero planificar es indispensable” [Nixon90, BF00].

Hay que almacenar los artefactos digitalmente y on-line —disponibles en el sitio web del proyecto— en lugar de en papel.

Obsérvese también que los artefactos del UP de los proyectos anteriores se pueden utilizar en otros posteriores. Es normal que haya muchas similitudes entre los proyectos en cuanto a los artefactos de riesgos, gestión del proyecto, pruebas y entorno. Todos los proyectos UP organizarán (o deberían organizar) los artefactos de la misma manera, con los mismos nombres (Lista de Riesgos, Marco de Desarrollo, etc.). Esto simplifica la búsqueda de artefactos reutilizables de los proyectos precedentes en las nuevas aplicaciones del UP.

4.4. No se entendió la fase de inicio cuando...

- La duración es mayor de “unas pocas” semanas en la mayoría de los proyectos.
- Se intenta definir la mayoría de los requisitos.
- Se espera que los planes y estimaciones sean fiables.
- Se define la arquitectura; en lugar de hacerlo de manera iterativa en la fase de elaboración.

- Se cree que la secuencia adecuada de trabajo debería ser: 1) definición de los requisitos; 2) diseño de la arquitectura; 3) implementación.
- No hay artefacto de Análisis del Negocio o Visión.
- No se identificaron la mayoría de los nombres de los casos de uso y los actores.
- Se escribieron todos los casos de uso en detalle.
- Ninguno de los casos de uso se escribió en detalle; cuando del 10-20% se deberían escribir con detalle para obtener algún conocimiento realista del alcance del problema.

Capítulo 5

COMPRENSIÓN DE LOS REQUISITOS

El nuestro es un mundo donde la gente no sabe lo que quiere y está deseando atravesar el infierno para conseguirlo.

Don Marquis

Objetivos

- Definir el modelo FURPS+.
 - Relacionar los tipos de requisitos con los artefactos del UP.
-

Introducción

No todos los requisitos se crean igual. Este capítulo presenta la clasificación de requisitos FURPS+.

Los **requisitos** son capacidades y condiciones con las cuales debe ser conforme el sistema —y más ampliamente, el proyecto [JBR99]—. El primer reto del trabajo de los requisitos es encontrar, comunicar y recordar (que normalmente significa registrar) lo que se necesita realmente, de manera que tenga un significado claro para el cliente y los miembros del equipo de desarrollo.

El UP fomenta un conjunto de buenas prácticas, una de las cuales es la *gestión de requisitos*. Esto no hace referencia a la actitud del ciclo de vida en cascada de definir completamente y estabilizar los requisitos en la primera fase del proyecto, sino más bien —en el contexto de que inevitablemente los deseos del personal involucrado son

cambiantes y poco claros— “un enfoque sistemático para encontrar, documentar, organizar y seguir la pista de los requisitos cambiantes de un sistema” [RUP]; en concreto, haciéndolo con destreza y sin ser descuidado. Fíjese en la palabra *cambiantes*; el UP acepta el cambio en los requisitos como un motor fundamental del proyecto. Otro término importante es *encontrar*; es decir, elicitar cuidadosamente mediante técnicas tales como escritura de casos de uso y talleres de requisitos.

Como se indica en la Figura 5.1, un estudio sobre los costes en proyectos reales en diferentes empresas reveló que el 37% de ellos estaban relacionados con los requisitos, de manera que las cuestiones de requisitos constituyen la principal causa de problemas [Standish94]. En consecuencia, es importante adquirir dominio en la gestión de requisitos. La respuesta del ciclo de vida en cascada a este dato sería intentar con ahínco pulir, estabilizar y fijar los requisitos antes de cualquier diseño o implementación, pero la historia demuestra que es una batalla perdida. La respuesta iterativa es utilizar un proceso que acepte el cambio y la retroalimentación como motores centrales en el descubrimiento de los requisitos.

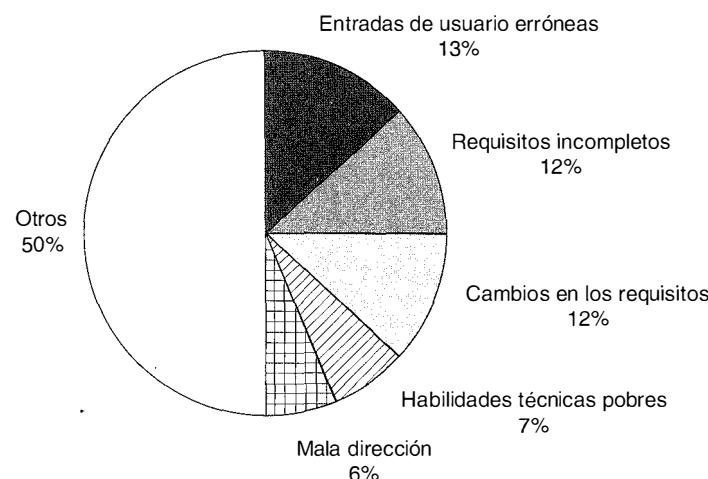


Figura 5.1. Factores del coste en proyectos software reales.

5.1. Tipos de requisitos

En el UP, los requisitos se clasifican de acuerdo con el modelo FURPS+ [Grady92], un útil nemotécnico que significa los siguientes cinco tipos de requisitos¹:

- **Funcional (*Functional*):** características, capacidades y seguridad.
- **Facilidad de uso (*Usability*):** factores humanos, ayuda, documentación.
- **Fiabilidad (*Reliability*):** frecuencia de fallos, capacidad de recuperación de un fallo y grado de previsión.

¹ Hay varios sistemas de clasificación de requisitos y atributos de calidad publicados en libros y por organizaciones estándares, como el ISO 9126 (que es similar a la lista del FURPS+), y varias del Instituto de Ingeniería del Software (SEI, *Software Engineering Institute*); cualquiera de ellas se puede utilizar en un proyecto UP.

- **Rendimiento (Performance):** tiempos de respuesta, productividad, precisión, disponibilidad, uso de los recursos.
- **Soporte (Supportability):** adaptabilidad, facilidad de mantenimiento, internacionalización, configurabilidad.

El ‘+’ en FURPS+ indica requisitos adicionales, tales como:

- **Implementación:** limitación de recursos, lenguajes y herramientas, hardware...
- **Interfaz:** restricciones impuestas para la interacción con sistemas externos.
- **Operaciones:** gestión del sistema en su puesta en marcha.
- **Empaquetamiento**
- **Legales:** licencias, etcétera.

Resulta útil utilizar las categorías del FURPS+ (o algún esquema de clasificación) como una lista para comprobar que se cubren los requisitos, de manera que reducimos el riesgo de no considerar alguna faceta importante del sistema.

Algunos de estos requisitos se denominan colectivamente **atributos de calidad, requisitos de calidad**, o las “-ilities”² de un sistema. Éstos comprenden facilidad de uso (*usability*), fiabilidad (*reliability*), rendimiento (*performance*) y soporte (*supportability*). Lo normal es dividir los requisitos en **funcionales** (comportamiento) y **no funcionales** (todo lo demás); a algunos no les gusta esta amplia generalización [BCK98], pero se utiliza de manera muy extendida.

Los requisitos funcionales se estudian y recogen en el Modelo de Casos de Uso, el tema del siguiente capítulo, y en la lista de características del sistema del artefacto Visión. Los otros requisitos se pueden recoger en los casos de usos con los que están relacionados, o en el artefacto Especificación Complementaria. El artefacto Visión resume los requisitos de alto nivel que se elaboran en estos otros documentos. El Glosario agrupa y clarifica los términos que se utilizan en los requisitos. El Glosario en el UP también comprende el concepto de **diccionario de datos**, que reúne los requisitos relacionados con los datos, como reglas de validación, valores aceptables, etcétera. Los prototipos son un mecanismo para clarificar qué es lo que se quiere o es posible.

Como veremos cuando estudiemos el análisis arquitectural, los requisitos de calidad influyen fuertemente en la arquitectura de un sistema. Por ejemplo, un requisito de alto rendimiento y alta fiabilidad influirá en la elección de componentes software y hardware, y en sus configuraciones. La necesidad de fácil adaptación, debido a cambios frecuentes en los requisitos funcionales, igualmente dará forma al diseño del software.

5.2. Lecturas adicionales

En los capítulos siguientes se cubrirán referencias relacionadas con casos de uso. Se recomiendan como punto de partida para el estudio de los requisitos, textos sobre requisitos orientados a los casos de uso, como *Writing Effective Use Cases* [Cockburn01], en lugar de textos más generales (y normalmente, tradicionales).

² *N. del T.:* Terminación plural en inglés de los nombres de tales requisitos.

Hay un amplio debate sobre requisitos —y una extensa variedad de temas de ingeniería del software— bajo el paraguas de Cuerpo de Conocimientos en Ingeniería del Software (**SWEBOK**, *Software Engineering Body of Knowledge*) disponible en www.swebok.org.

El SEI (www.sei.cmu.edu) tiene varias propuestas relacionadas con los requisitos de calidad. El ISO 9126, el IEEE Std. 830, y el IEEE Std. 1061 son estándares relacionados con los requisitos y atributos de calidad, y están disponibles en la web en varios sitios.

Algunas advertencias con respecto a los libros generales sobre requisitos, incluso aquellos que proponen cubrir los casos de uso, desarrollo iterativo o, de hecho, incluso los requisitos en el UP:

1. La mayoría están escritos bajo la influencia de un ciclo de vida en cascada planteando la definición “precisa” de los requisitos por adelantado, antes de pasar al diseño y la implementación. Esto no pretende invalidar su valor más amplio o, a menudo, profundas y útiles visiones de los requisitos independientes del método, sino que pretende aclarar que no constituyen una visión fiable del desarrollo iterativo. Esto es debido a que la experiencia de los autores proviene principalmente de proyectos de ciclo de vida en cascada, trabajando para refinar, definir cuidadosamente y precisamente los requisitos, y terminar con la fase de requisitos antes de continuar con el diseño. Aquellos libros que también mencionan el desarrollo iterativo, lo abordan muy superficialmente, quizás añadiendo material “iterativo” apelando a las tendencias modernas. Por tanto, los libros y artículos de requisitos deberían leerse con cautela; uno podría sentirse seguro con la idea de intentar definir de manera cuidadosa todos los requisitos en la fase inicial, que no es consistente con un proceso iterativo.
2. Muchos libros generales sobre requisitos que también dicen incluir casos de uso, lo hacen muy superficialmente, o entienden mal el significado real de requisitos dirigidos por casos de uso. Esto podría deberse a que los autores poseen una experiencia de muchos años en métodos de requisitos tradicionales, y recientemente, han intentado incluir los casos de uso en sus métodos anteriores, sin darse cuenta de que la idea principal de los casos de uso, tal y como la concibe Ivar Jacobson y el UP, es hacer los casos de uso el elemento central del enfoque de requisitos global —sustituyendo a otros documentos de requisitos como elemento central—; los casos de uso impregnán y dirigen el trabajo de requisitos, en lugar de considerarse como una técnica de apoyo auxiliar a nivel inferior o medio, añadida a los documentos o enfoques de requisitos tradicionales.

En resumen, los libros generales sobre requisitos ofrecen consejos útiles sobre técnicas y cuestiones para recopilar los requisitos, escritos por personas expertas, pero normalmente, los presentan en el contexto de un proceso en cascada, sin tener un gran conocimiento de las implicaciones más profundas de los casos de uso. Cualquier consejo sobre el proceso que sea una variante de: “trate de definir la mayoría de los requisitos, y entonces pase al diseño y la implementación” no es consistente con el desarrollo iterativo y el UP.

MODELO DE CASOS DE USO: ESCRITURA DE REQUISITOS EN CONTEXTO

El primer paso indispensable para conseguir las cosas que quieras de la vida: decidir qué quieras.

Ben Stein

Objetivos

- Identificar y escribir casos de uso.
 - Relacionar los casos de uso con los objetivos de los usuarios y los procesos de negocio básicos.
 - Utilizar los formatos breve, informal y completo, en un estilo esencial.
 - Relacionar el trabajo con casos de uso con el desarrollo iterativo.
-

Introducción

Merece la pena estudiar este capítulo durante la primera lectura del libro porque los casos de uso son un mecanismo ampliamente utilizado para descubrir y registrar los requisitos (especialmente los funcionales); influencian muchos aspectos de un proyecto, incluyendo el A/DOO. Merece la pena tanto saber sobre los casos de uso como crearlos.

La escritura de casos de uso —historias del uso de un sistema— es una técnica excelente para entender y describir los requisitos. Este capítulo explora los conceptos claves de los casos de uso y presenta casos de uso de ejemplo para la aplicación NuevaEra.

El UP define el **Modelo de Casos de Uso** en la disciplina Requisitos. Básicamente, es el conjunto de todos los casos de uso; es un modelo de la funcionalidad y entorno del sistema.

6.1. Objetivos e historias

Los clientes y los usuarios finales tienen objetivos (también conocidos como *necesidades*) y quieren sistemas informáticos que les ayuden a conseguirlos, que varían desde registrar las ventas hasta estimar el flujo de petróleo de futuros pozos. Hay varias formas de capturar estos objetivos y requisitos del sistema; las mejores son simples y familiares, porque esto hace que sea más fácil —especialmente para clientes y usuarios finales— contribuir a su definición o evaluación. Eso reduce el riesgo de perder el hilo.

Los casos de uso son un mecanismo para ayudar a mantenerlo simple y entendible para todo el personal involucrado. De manera informal, son historias del uso de un sistema para alcanzar los objetivos. A continuación presentamos un ejemplo de caso de uso en *formato breve*:

Procesar Venta: Un cliente llega a una caja con artículos para comprar. El cajero utiliza el sistema PDV para registrar cada artículo comprado. El sistema presenta una suma parcial y detalles de cada línea de venta. El cliente introduce los datos del pago, que el sistema valida y registra. El sistema actualiza el inventario. El cliente recibe un recibo del sistema y luego se va con los artículos.

A menudo, los casos de uso necesitan una elaboración mayor que ésta, pero la esencia es descubrir y registrar los requisitos funcionales, mediante la escritura de historias del uso de un sistema, para ayudar a cumplir los objetivos de varias de las personas involucradas; esto es, los *casos de uso*¹. Se supone que no es una idea difícil, aunque, de hecho podría ser difícil descubrir o decidir lo que es necesario, y escribirlo de manera coherente con un nivel de detalle útil.

Se ha escrito mucho acerca de los casos de uso, y si bien es útil, existe el riesgo entre las personas inteligentes y creativas, de oscurecer una idea sencilla con niveles de sofisticación. Normalmente es posible distinguir a un modelador de casos de uso novato (o a un analista serio de Tipo A) preocupándose en exceso con cuestiones secundarias como diagramas de casos de uso, relaciones de casos de uso, paquetes de casos de uso, atributos opcionales, etcétera, en lugar de escribir las historias. En otras palabras, el poder del mecanismo de casos de uso es la capacidad tanto de aumentar como de disminuir, en términos de sofisticación y formalidad, dependiendo de la necesidad.

6.2. Antecedentes

La idea de utilizar los casos de uso para describir los requisitos funcionales fue introducida en 1986 por Ivar Jacobson [Jacobson92], uno de los contribuidores principales al UML y UP. La idea de caso de uso de Jacobson ha tenido una gran influencia y ha sido ampliamente reconocida; siendo sus principales virtudes la simplicidad y utilidad. Aunque muchos han contribuido en este campo, se puede sostener que el siguiente paso más coherente, comprensible e influyente en la definición de qué son (o deberían ser) los casos de uso y cómo escribirlos, procede de Alistair Cockburn, resumido en su popular texto *Writing Effective Use Cases* [Cockburn01], basado en sus primeros trabajos y escritos publicados de 1992 en adelante. Esta introducción, por tanto, se basa y es consistente con este último trabajo.

¹ El término original en sueco se traduce literalmente como “caso de costumbre”.

6.3. Casos de uso y valor añadido

En primer lugar, algunas definiciones informales: un **actor** es algo con comportamiento, como una persona (identificada por un rol), sistema informatizado u organización; por ejemplo, un cajero.

Un **escenario** es una secuencia específica de acciones e interacciones entre los actores y el sistema objeto de estudio; también se denomina **instancia de caso de uso**. Es una historia particular del uso de un sistema, o un camino a través del caso de uso; por ejemplo, el escenario de éxito de compra de artículos con pago en efectivo, o el escenario de fallo al comprar debido al rechazo de la transacción de pago con la tarjeta de crédito.

Informalmente entonces, un **caso de uso** es una colección de escenarios con éxito y fallo relacionados, que describe a los actores utilizando un sistema para satisfacer un objetivo. Por ejemplo, a continuación presentamos un caso de uso en *formato informal* que incluye algunos escenarios alternativos:

Gestionar Devoluciones

Escenario principal de éxito: Un cliente llega a una caja con artículos para devolver. El cajero utiliza el sistema PDV para registrar cada uno de los artículos devueltos...

Escenarios alternativos:

Si se pagó con tarjeta de crédito, y se rechaza la transacción de reembolso a su cuenta, informar al cliente y pagarle en efectivo.

Si el identificador del artículo no se encuentra en el sistema, notificar al Cajero y sugerir la entrada manual del código de identificación (quizás esté alterado).

Si el sistema detecta fallos en la comunicación con el sistema de contabilidad externo...

El RUP proporciona una definición alternativa, aunque similar, de un caso de uso:

Un conjunto de instancias de caso de uso, donde cada instancia es una secuencia de acciones que un sistema ejecuta, produciendo un resultado observable de valor para un actor particular [RUP].

La expresión “*un resultado observable de valor*” es sutil pero importante, porque destaca el hecho de que el comportamiento del sistema debería preocuparse de proporcionar valor al usuario.

Una actitud clave en el trabajo con casos de uso es centrarse en la pregunta: “¿Cómo puedo, utilizando el sistema, proporcionar un valor observable al usuario, o cumplir sus objetivos?” en lugar de, simplemente, pensar en los requisitos del sistema en términos de una “lista de la lavandería” de características o funciones.

Quizás parece obvio destacar que se proporcione un valor observable para el usuario, pero la industria de software está plagada de proyectos fracasados que no proporciona-

ron lo que la gente realmente necesitaba. El enfoque de la lista de características y funciones para capturar los requisitos, puede contribuir a este resultado negativo, puesto que no fomenta que el personal involucrado considere los requisitos en un contexto amplio de uso del sistema, en un escenario para alcanzar algún resultado observable de valor, o algún objetivo. Por el contrario, los casos de uso sitúan las características y funciones en un contexto orientado al objetivo. De ahí el título del capítulo².

Ésta es la idea clave que Jacobson intentaba transmitir con el concepto de caso de uso: trabaja con los requisitos centrándote en cómo puede un sistema añadir valor y cumplir los objetivos.

6.4. Casos de uso y requisitos funcionales

Los casos de uso son requisitos; ante todo son requisitos funcionales que indican qué hará el sistema. En términos de los tipos de requisitos FURPS+, los casos de uso se refieren fundamentalmente a la “F” (funcional o de comportamiento), pero también pueden utilizarse para otros tipos, especialmente cuando esos otros tipos están estrechamente relacionados con un caso de uso. En el UP —y métodos más modernos— los casos de uso son el mecanismo principal que se recomienda para su descubrimiento y definición. Los casos de uso definen una promesa o contrato de la manera en la que se comportará un sistema.

Para ser claros: los casos de uso *son* requisitos (aunque no todos los requisitos). Algunos piensan en requisitos sólo como listas de características y funciones de la forma “el sistema deberá hacer...”. No es así, y una idea clave de los casos de uso es (por lo general) reducir la importancia o el uso de listas de características detalladas al estilo antiguo y más bien, escribir casos de uso para los requisitos funcionales. Veremos más sobre este punto en una sección posterior.

Los casos de uso son documentos de texto, no diagramas, y el modelado de casos de uso es, sobre todo, una acción de escribir texto, no dibujar. Sin embargo, UML define un diagrama de casos de uso para ilustrar los nombres de casos de uso y actores, y sus relaciones.

6.5. Tipos de casos de uso y formatos

Casos de uso de caja negra y las responsabilidades del sistema

Los casos de uso de caja negra son la clase más común y recomendada; no describen el funcionamiento interno del sistema, sus componentes o diseño, sino que se describe el sistema en base a las *responsabilidades* que tiene, que es una metáfora común y unificadora en el pensamiento orientado a objetos —los elementos software tienen responsabilidades y colaboran con otros elementos que tienen responsabilidades—.

² Procede del libro titulado muy apropiadamente *Uses Cases: Requirements in Context* [GK00] (el título del capítulo se adaptó con permiso de los autores).

A través de la definición de las responsabilidades del sistema con casos de uso de caja negra, es posible especificar *qué* debe hacer el sistema (los requisitos funcionales) sin decidir *cómo* lo hará (el diseño). De hecho, la definición de “análisis” frente al “diseño” se resume algunas veces como el “qué” frente al “cómo”. Éste es un tema importante en un buen desarrollo de software: evite durante el análisis de requisitos tomar decisiones acerca del “cómo”, y especifique el comportamiento externo del sistema, como una caja negra. Después, durante el diseño, cree una solución que satisfaga la especificación.

<i>Estilo de caja negra</i>	<i>No</i>
El sistema registra la venta	El sistema escribe la venta en una base de datos... o (incluso peor). El sistema genera una sentencia SQL INSERT para la venta...

Tipos de formalidad

Los casos de uso se escriben con formatos diferentes, dependiendo de la necesidad. Además del tipo de *visibilidad*, de caja negra frente a caja blanca, los casos de uso se escriben con varios grados de *formalidad*:

- **Breve:** resumen conciso de un párrafo, normalmente del escenario principal con éxito. El anterior ejemplo de *Procesar Venta* era breve.
- **Informal:** formato de párrafo en un estilo informal. Múltiples párrafos que comprenden varios escenarios. El anterior ejemplo de *Gestionar Devoluciones* era informal.
- **Completo:** el más elaborado. Se escriben con detalle todos los pasos y variaciones, y hay secciones de apoyo como precondiciones y garantías de éxito.

El ejemplo siguiente es un caso de uso en formato completo del caso de estudio NuevaEra.

6.6. Ejemplo completo: Procesar Venta

Los casos de uso completos muestran más detalles y están estructurados; son útiles para entender en profundidad los objetivos, tareas y requisitos. En el caso de estudio del PDV NuevaEra, se crearían en uno de los primeros talleres de requisitos con la colaboración del analista del sistema, expertos en la materia de estudio y los desarrolladores.

El formato usecases.org

Hay varias plantillas disponibles para los casos de uso completos. Sin embargo, quizás el formato más ampliamente extendido y compartido es la plantilla disponible en www.usecases.org. El siguiente ejemplo muestra este estilo.

El lector debe darse cuenta que éste es el principal ejemplo de caso de uso detallado para el caso de estudio del libro; muestra muchos elementos y cuestiones comunes.

Caso de uso UC1: Procesar Venta

Actor principal: Cajero.

Personal involucrado e intereses:

- Cajero: quiere entradas precisas, rápidas, y sin errores de pago, ya que las pérdidas se deducen de su salario.
- Vendedor: quiere que las comisiones de las ventas estén actualizadas.
- Compañía: Quiere registrar las transacciones con precisión y satisfacer los intereses de los clientes. Quiere asegurar que se registran los pagos aceptados por el Servicio de Autorización de Pagos. Quiere cierta tolerancia a fallos que permita capturar las ventas incluso si los componentes del servidor (ej. validación remota de crédito) no están disponibles. Quiere actualización automática y rápida de la contabilidad y el inventario.
- Agencia Tributaria del Gobierno: quiere recopilar los impuestos de cada venta. Podrían ser múltiples agencias: nacional, provincial y local.
- Servicio de Autorización de Pagos: Quiere recibir peticiones de autorización digital con el formato y protocolo correctos. Quiere registrar de manera precisa las cuentas por cobrar de la tienda.

Precondiciones: El cajero se identifica y autentica.

Garantías de éxito (Postcondiciones): Se registra la venta. El impuesto se calcula de manera correcta. Se actualizan la contabilidad y el inventario. Se registran las comisiones. Se genera el recibo. Se registran las autorizaciones de pago aprobadas.

Escenario principal de éxito (o Flujo Básico):

1. El Cliente llega a un terminal PDV con mercancías y/o servicios que comprar.
2. El Cajero comienza una nueva venta.
3. El Cajero introduce el identificador del artículo.
4. El Sistema registra la línea de la venta y presenta la descripción del artículo, precio y suma parcial. El precio se calcula a partir de un conjunto de reglas de precios.

El Cajero repite los pasos 3-4 hasta que se indique.

5. El Sistema presenta el total con los impuestos calculados.
6. El Cajero le dice al Cliente el total y pide que le pague.
7. El Cliente paga y el Sistema gestiona el pago.
8. El Sistema registra la venta completa y envía la información de la venta y el pago al sistema de Contabilidad externo (para la contabilidad y las comisiones) y al sistema de Inventario (para actualizar el inventario).
9. El Sistema presenta el recibo.
10. El Cliente se va con el recibo y las mercancías (si es el caso).

Extensiones (o Flujos Alternativos):

*a. En cualquier momento el Sistema falla:

Para dar soporte a la recuperación y registro correcto, asegura que todos los estados y eventos significativos de una transacción puedan recuperarse desde cualquier paso del escenario.

1. El Cajero reinicia el Sistema, inicia la sesión, y solicita la recuperación al estado anterior.
 2. El Sistema reconstruye el estado anterior.
- 2a. El Sistema detecta anomalías intentando la recuperación:
1. El Sistema informa del error al Cajero, registra el error, y pasa a un estado limpio.
 2. El Cajero comienza una nueva venta.

- 3a. Identificador no válido:
 1. El Sistema señala el error y rechaza la entrada.
- 3b. Hay muchos artículos de la misma categoría y tener en cuenta una única identidad del artículo no es importante (ej. 5 paquetes de hamburguesas vegetales):
 1. El Cajero puede introducir el identificador de la categoría del artículo y la cantidad.
- 3-6a. El Cliente le pide al Cajero que elimine un artículo de la compra:
 1. El Cajero introduce el identificador del artículo para eliminarlo de la compra.
 2. El Sistema muestra la suma parcial actualizada.
- 3-6b. El Cliente le pide al Cajero que cancele la venta:
 1. El Cajero cancela la venta en el Sistema.
- 3-6c. El Cajero detiene la venta:
 1. El sistema registra la venta para que esté disponible su recuperación en cualquier terminal PDV.
- 4a. El Sistema genera el precio de un artículo que no es el deseado (ej. el Cliente se queja por algo y se le ofrece un precio más bajo):
 1. El Cajero introduce el precio alternativo.
 2. El Sistema presenta el precio nuevo.
- 5a. El sistema encuentra algún fallo para comunicarse con el servicio externo del sistema de cálculo de impuestos.
 1. El Sistema reinicia el servicio en el nodo PDV y continúa.
 - 1a. El Sistema detecta que el servicio no se reinicia.
 1. El Sistema señala el error.
 2. El Cajero podría calcular e introducir manualmente el impuesto, o cancelar la venta.
- 5b. El Cliente dice que le son aplicables descuentos (ej. empleado, cliente preferente):
 1. El Cajero señala la petición de descuento.
 2. El Cajero introduce la identificación del Cliente.
 3. El Sistema presenta el descuento total, basado en las reglas de descuento.
- 5c. El Cliente dice que tiene crédito en su cuenta, para aplicar a la venta:
 1. El Cajero señala la petición de crédito.
 2. El Cajero introduce la identificación del Cliente.
 3. El Sistema aplica el crédito hasta que el precio = 0, y reduce el crédito que queda.
- 6a. El Cliente dice que su intención era pagar en efectivo pero que no tiene suficiente:
 - 1a. El Cliente utiliza un método de pago alternativo.
 - 1b. El Cliente le dice al Cajero que cancele la venta. El Cajero cancela la venta en el Sistema.
- 7a. Pago en efectivo:
 1. El Cajero introduce la cantidad de dinero en efectivo entregada.
 2. El Sistema muestra la cantidad de dinero a devolver y abre el cajón de caja.
 3. El Cajero deposita el dinero entregado y devuelve el cambio al Cliente.
 4. El Sistema registra el pago en efectivo.
- 7b. Pago a crédito:
 1. El Cliente introduce la información de su cuenta de crédito.
 2. El Sistema envía la petición de autorización del pago al Sistema externo de Servicio de Autorización de Pagos, y solicita la aprobación del pago.
 - 2a. El Sistema detecta un fallo en la colaboración con el sistema externo:
 1. El Sistema señala el error al Cajero.
 2. El Cajero le pide al Cliente un modo de pago alternativo.
 3. El Sistema recibe la aprobación del pago y lo notifica al Cajero.
 - 3a. El Sistema recibe la denegación del pago:
 1. El Sistema señala la denegación al Cajero.
 2. El Cajero le pide al Cliente un modo de pago alternativo.

4. El Sistema registra el pago a crédito, que incluye la aprobación del pago
 5. El Sistema presenta el mecanismo de entrada para la firma del pago a crédito.
 6. El Cajero le pide al Cliente que firme el pago a crédito. El Cliente introduce la firma.
- 7c. Pago con cheque...
 7d. Pago a cuenta...
 7e. El Cliente presenta cupones:
1. Antes de gestionar el pago, el Cajero recoge cada cupón y el Sistema reduce el pago como sea oportuno. El sistema registra los cupones utilizados por razones de contabilidad.
 - 1a. El cupón introducido no es válido para ninguno de los artículos comprados
 1. El Sistema señala el error al Cajero.
- 9a. Hay rebajas en los artículos:
1. El Sistema presenta los formularios de rebaja y los recibos de descuento para cada artículo con una rebaja.
- 9b. El Cliente solicita un vale-regalo (sin precio visible):
1. El Cajero solicita el vale-regalo y el Sistema lo proporciona.

Requisitos especiales:

- Interfaz de Usuario con pantalla táctil en un gran monitor de pantalla plana. El texto debe ser visible a un metro de distancia.
- Tiempo de respuesta para la autorización de crédito de 30 segundos el 90% de las veces.
- De algún modo, queremos recuperación robusta cuando falla el acceso a servicios remotos, como el sistema de inventario.
- Internacionalización del lenguaje del texto que se muestra.
- Reglas de negocio que se puedan añadir en tiempo de ejecución en los pasos 3 y 7.
- ...

Lista de tecnología y variaciones de datos:

- 3a. El identificador del artículo se introduce mediante un escáner láser de código de barras (si está presente el código de barras) o a través del teclado.
 3b. El identificador del artículo podría ser cualquier esquema de código UPC, EAN, JAN o SKU.
 7a. La entrada de la información de la cuenta de crédito se lleva a cabo mediante un lector de tarjetas o el teclado.
 7b. La firma de los pagos a crédito se captura en un recibo de papel. Pero en dos años, pronosticamos que muchos clientes querrán que se capture la firma digital.

Frecuencia: Podría ser casi continuo.

Temas abiertos:

- ¿Cuáles son las variaciones de la ley de impuestos?
- Explorar las cuestiones de recuperación de servicios remotos.
- ¿Cuál es la adaptación que se tiene que hacer para diferentes negocios?
- ¿Un cajero debe llevarse el dinero de la caja cuando salga del sistema?
- ¿Puede utilizar el cliente directamente el lector de tarjetas o tiene que hacerlo el cajero?

Este caso de uso es más bien ilustrativo que exhaustivo (aunque está basado en los requisitos de un sistema PDV real). Sin embargo, muestra suficiente detalle y complejidad para exponer de manera realista que los casos de uso completos pueden documentar muchos detalles de los requisitos. Este ejemplo nos servirá bien como modelo para muchos problemas de los casos de uso.

La variación de dos-columnas

Algunos prefieren el formato en dos columnas o conversacional, que destaca el hecho de que se establece una interacción entre los actores y el sistema. Se propuso por primera vez por Rebecca Wirfs-Brock en [Wirfs-Brock93], y también promueven su uso Constantine y Lockwood para ayudar en el análisis e ingeniería de usabilidad [CL99]. A continuación presentamos el mismo contenido utilizando el formato en dos columnas:

Caso de uso UC1: Procesar Venta

Actor principal:como antes...	
Escenario principal de éxito: Acción del actor (o intención)	Responsabilidad del Sistema
1. El Cliente llega a un terminal PDV con mercancías y/o servicios que comprar.	
2. El Cajero comienza una nueva venta.	
3. El Cajero introduce el identificador del artículo.	
<i>El Cajero repite los pasos 3-4 hasta que se indique</i>	
6. El Cajero le dice al Cliente el total y pide que le pague.	
7. El Cliente paga.	
...	...

¿Cuál es el mejor formato?

No existe un mejor formato; unos prefieren el estilo de una columna, otros el de dos columnas. Las secciones se pueden añadir y quitar; los nombres de los títulos podrían cambiar. Ninguna de estas cosas es especialmente importante; la clave es escribir los detalles del escenario principal de éxito y sus extensiones de alguna forma. [Cockburn1] resume muchos formatos utilizables:

Práctica personal

Ésta es mi práctica, no una recomendación. Durante algunos años, utilicé el formato en dos columnas debido a su clara separación visual de la conversación. Sin embargo, he vuelto al estilo en una columna ya que es más compacto y más fácil de formatear, y el pequeño valor de la separación visual de la conversación, para mí, no supera estos beneficios. Encuentro que es todavía sencillo identificar visualmente las diferentes partes en la conversación (Cliente, Sistema,...) si cada parte y las respuestas del Sistema se asignan normalmente a sus propios pasos.

6.7. Explicación de las secciones

Elementos del prólogo

Son posibles muchos elementos opcionales en el prólogo. Sitúe al principio sólo los elementos que son importantes que se lean antes del escenario principal de éxito. Mueve el material de “encabezamiento” ajeno, al final de los casos de uso.

Actor principal: El actor principal que recurre a los servicios del sistema para cumplir un objetivo.

Importante: Personal involucrado y lista de intereses

Esta lista es más importante y práctica de lo que podría parecer a primera vista. Sugiere y delimita qué es lo que debe hacer el sistema. Citando a Cockburn:

El [sistema] funciona siguiendo un contrato entre el personal involucrado, donde los casos de usos detallan la parte de comportamiento del contrato... El caso de uso, como contrato de comportamiento, captura *todo y sólo* el comportamiento relacionado con la satisfacción de los intereses del personal involucrado [Cockburn01].

Esto contesta la pregunta: ¿Qué debería estar en un caso de uso? La respuesta es: lo que satisface los intereses de todo el personal involucrado. Además, empezando con el personal involucrado y sus intereses antes de escribir el resto del caso de uso, tenemos un modo de recordarnos cuáles deberían ser las responsabilidades más detalladas del sistema. Por ejemplo, ¿habría identificado una responsabilidad para gestionar la comisión del vendedor si no hubiese listado en primer lugar la persona involucrada vendedor y sus intereses? En el mejor de los casos al final, pero quizás lo habría echado en falta durante la primera sesión de análisis. El punto de vista del interés del personal involucrado proporciona un procedimiento metódico y completo para el descubrimiento y registro de todos los comportamientos requeridos.

Personal involucrado e intereses:

- Cajero: quiere entradas precisas, rápidas, y sin errores de pago, ya que las pérdidas se deducen de su salario.
- Vendedor: quiere que las comisiones de las ventas estén actualizadas.
- ...

Precondiciones y garantías de éxito (postcondiciones)

Las **precondiciones** establecen lo que *siempre debe cumplirse* antes de comenzar un escenario en el caso de uso. Las precondiciones *no* se prueban en el caso de uso, sino que son condiciones que se asumen que son verdad. Normalmente, una precondición implica un escenario de otro caso de uso que se ha completado con éxito, como inicio de sesión o el más general “el cajero se identifica y autentica”. Nótese que hay condiciones

que deben ser verdad, pero no tienen un valor práctico para que se escriban como “el sistema tiene energía”. Las precondiciones comunican suposiciones importantes de las que el escritor del caso de uso piensa que los lectores deberían ser avisados.

Las **garantías de éxito** (o **postcondiciones**) establecen qué debe cumplirse cuando el caso de uso se completa con éxito —o bien el escenario principal de éxito o algún camino alternativo—. La garantía debería satisfacer las necesidades de todo el personal involucrado.

Precondiciones: El cajero se identifica y autentica.

Garantías de éxito (Postcondiciones): Se registra la venta. El impuesto se calcula de manera correcta. Se actualizan la Contabilidad y el Inventario. Se registran las comisiones. Se genera el recibo.

Escenario principal de éxito y pasos (o Flujo Básico)

También recibe el nombre de escenario del “camino feliz”, o más prosaico “Flujo Básico”. Describe el camino de éxito típico que satisface los intereses del personal involucrado. Nótese que, a menudo, *no* incluye ninguna condición o bifurcación. Aunque no es incorrecto o ilegal, se puede suponer que es más comprensible y extensible ser muy consistente, y postergar todo el manejo de caminos condicionales a la sección Extensiones.

Sugerencia

Posponga todas las sentencias condicionales y de bifurcación a la sección Extensiones.

El escenario recoge los pasos, que pueden ser de tres tipos:

1. Una interacción entre actores³.
2. Una validación (normalmente a cargo del sistema).
3. Un cambio de estado realizado por el sistema (por ejemplo, registrando o modificando algo).

El primer paso de un caso de uso, normalmente no está incluido en esta clasificación, sino que indica el evento que desencadena el comienzo del escenario.

Un estilo habitual es poner con mayúsculas los nombres de los actores para facilitar la identificación. También podemos observar el estilo que se utiliza para indicar una repetición.

Escenario principal de éxito (o Flujo Básico):

1. El Cliente llega a un terminal PDV con mercancías para comprar.
2. El Cajero comienza una nueva venta.
3. El Cajero introduce el identificador del artículo.
4. ...
- El Cajero repite los pasos 3-4 hasta que se indique.
5. ...

³ Nótese que el sistema que se está estudiando en sí mismo debería considerarse un actor cuando juega un rol de actor colaborando con otros sistemas.

Extensiones (o Flujos Alternativos)

Las extensiones son muy importantes. Indican todos los otros escenarios o bifurcaciones, tanto de éxito como de fracaso. Podemos observar que en el ejemplo de caso de uso completo, la sección Extensiones es considerablemente más larga y compleja que la correspondiente al Escenario Principal de Éxito; esto es normal y de esperar. También se conocen como “Flujos Alternativos”.

En la escritura de casos de uso completos, la combinación del camino feliz y los escenarios de extensión deberían satisfacer “casi” todos los intereses del personal involucrado. Este punto está limitado, puesto que algunos intereses se podrían capturar mejor como requisitos no funcionales escritos en la Especificación Complementaria en lugar de en los casos de uso.

Los escenarios de extensión son bifurcaciones del escenario principal de éxito y, por tanto, pueden ser etiquetados de acuerdo con él. Por ejemplo, en el Paso 3 del escenario principal podría haber un identificador de artículo inválido, bien porque no se introdujo correctamente o bien porque el sistema no lo conoce. Una extensión se etiqueta como “3a”; primero identifica la condición y después la respuesta. Una extensión alternativa al Paso 3 se etiqueta como “3b” y así sucesivamente.

Extensiones (o Flujos Alternativos):

3a. Identificador no válido:

1. El Sistema señala el error y rechaza la entrada.

3b. Hay muchos artículos de misma categoría y tener en cuenta una única identidad del artículo no es importante (ej. 5 paquetes de hamburguesas vegetales):

1. El Cajero puede introducir el identificador de la categoría del artículo y la cantidad.

Una extensión tiene dos partes: la condición y el manejo.

Guía: Escriba la condición como algo que pueda ser *detectado* por el sistema o un actor. Para contrastar:

5a. El Sistema detecta un fallo en la comunicación con el servicio externo del sistema de cálculo de impuestos:

5a. El sistema de cálculo de impuestos externo no funciona:

Se prefiere el primer estilo porque se trata de algo que el sistema puede detectar; el último es una inferencia.

El manejo de la extensión se puede resumir en un paso, o incluir una secuencia, como en este ejemplo, que también ilustra la notación utilizada para indicar que una condición puede tener lugar en una serie de pasos:

3-6a. El Cliente le pide al Cajero que elimine un artículo de la compra:

1. El Cajero introduce el identificador del artículo para eliminarlo de la compra.
2. El Sistema muestra la suma parcial actualizada.

Al final del manejo de la extensión, por defecto, el escenario se une de nuevo con el escenario principal de éxito, a menos que la extensión indique otra cosa (como interrumpir el sistema).

Algunas veces, un punto de extensión particular es bastante complejo, como en la extensión “pago a crédito”. Esto puede ser un motivo para expresar la extensión como un caso de uso aparte.

Este ejemplo de extensión también muestra la notación para expresar fallos en las extensiones.

7b. Pago a crédito:

1. El Cliente introduce la información de su cuenta de crédito.
2. El Sistema envía la petición de autorización del pago al Sistema externo de Servicio de Autorización de Pagos, y solicita la aprobación del pago.
 - 2a. El Sistema detecta un fallo en la colaboración con el sistema externo:
 1. El Sistema señala el error al Cajero.
 2. El Cajero le pide al Cliente un modo de pago alternativo.
- 3 ...

Si es deseable describir una condición de extensión que puede ser posible durante cualquiera (o al menos la mayoría) de los pasos, se pueden utilizar las etiquetas *a, *b,...

***a. En cualquier momento el Sistema falla:**

- Para dar soporte a la recuperación y registro correcto, asegura que todos los estados y eventos significativos de una transacción puedan recuperarse desde cualquier paso del escenario.
1. El Cajero reinicia el Sistema, inicia la sesión, y solicita la recuperación al estado anterior.
 2. El Sistema reconstruye el estado anterior.

Requisitos especiales

Si un requisito no funcional, atributo de calidad o restricción se relaciona de manera específica con un caso de uso, se recoge en el caso de uso. Esto incluye cualidades tales como rendimiento, fiabilidad y facilidad de uso, y restricciones de diseño (a menudo, en dispositivos de entrada/salida) que son obligados o se consideran probables.

Requisitos especiales:

- Interfaz de usuario con pantalla táctil en un gran monitor de pantalla plana. El texto debe ser visible a un metro de distancia.
- Tiempo de respuesta para la autorización de crédito de 30 segundos el 90% de las veces.
- Internacionalización del lenguaje del texto que se muestra.
- Reglas de negocio que se puedan añadir en ejecución en los pasos 3 y 7.

Un consejo clásico del UP es registrar estos requisitos con el caso de uso, y constituye un lugar razonable al escribir primero el caso de uso. Sin embargo, muchos expertos encuentran útil reunir al final todos los requisitos no funcionales en la Especificación Complementaria, para favorecer la gestión del contenido, comprensión y legibilidad por-

que, normalmente, se tienen que considerar estos requisitos en conjunto durante el análisis arquitectural.

Lista de tecnología y variaciones de datos

A menudo, encontramos variaciones técnicas en *cómo* se debe hacer algo, pero no en qué, y es importante registrarlo en el caso de uso. Un ejemplo típico es una restricción técnica impuesta por el personal involucrado con respecto a las tecnologías de entrada o salida de datos. Por ejemplo, uno de los interesados podría decir, “El sistema PDV debe soportar la entrada de la cuenta de crédito utilizando un lector de tarjetas y el teclado”. Nótese que éstos son ejemplos de decisiones o restricciones de diseño anticipadas; en general, deben evitarse decisiones de diseño prematuras, pero algunas veces son obvias o inevitables, especialmente en relación con las tecnologías de entrada/salida.

También es necesario entender las variaciones en los esquemas de los datos, como utilizar UPCs o EANs para los identificadores de los artículos, codificados mediante el código de barras.

Esta lista es el lugar para situar tales variaciones. También es útil registrar las variaciones en los datos que podrían capturarse en un paso particular.

Lista de tecnología y variaciones de datos

- 3a. El identificador del artículo se introduce mediante un escáner láser de código de barras (si está presente el código de barras) o a través del teclado.
- 3b. El identificador del artículo podría ser cualquier esquema de código UPC, EAN, JAN o SKU.
- 7a. La entrada de la información de la cuenta de crédito se lleva a cabo mediante un lector de tarjetas o el teclado.
- 7b. La firma de los pagos a crédito se captura en un recibo de papel. Pero en dos años, pronosticamos que muchos clientes querrán que se capture la firma digital.

Sugerencia

Esta sección no debería contener múltiples pasos para representar la variación de comportamiento en diferentes casos. Si es necesario, dígalo en la sección Extensiones.

6.8. Objetivos y alcance de un caso de uso

¿Cómo deberían descubrirse los casos de uso? Es típico no estar seguro si algo es un caso de uso válido (o de manera más práctica, útil). Las tareas se pueden agrupar a muchos niveles de granularidad, desde uno o unos pocos pasos pequeños, hasta actividades de nivel de empresa.

¿A qué nivel y alcance deberían expresarse los casos de uso?

Las siguientes secciones presentan las ideas sencillas de los procesos y objetivos del negocio elementales, como marco para la identificación de los casos de uso de una aplicación.

Casos de uso para los procesos del negocio elementales

¿Cuál de éstos es un caso de uso válido?

- Negociar un Contrato con el Proveedor
- Gestionar las Devoluciones
- Iniciar Sesión

Podría argumentarse que todos ellos son casos de uso *a diferentes niveles*, dependiendo de los límites del sistema, actores y objetivos. La evaluación de estos candidatos se presenta después de una introducción a los procesos del negocio elementales.

En lugar de preguntar en general: “¿qué es un caso de uso válido?”, una pregunta más relevante para el caso de estudio PDV es: ¿cuál es el nivel útil para expresar los casos de uso en el análisis de requisitos de la aplicación?

Guía: El caso de uso EBP

Para el análisis de requisitos de una aplicación informática, céntrese en los casos de uso al nivel de **procesos del negocio elementales** (EBPs, *Elementary Business Processes*).

EBP es un término que procede del campo de la ingeniería de procesos del negocio⁴, y se define como:

Una tarea realizada por una persona en un lugar, en un instante, como respuesta a un evento del negocio, que añade un valor cuantificable para el negocio y deja los datos en un estado consistente, ej. Autorizar Crédito, o Solicitar Precio (se perdió la fuente original).

Esto se puede tomar demasiado literalmente: ¿Está mal considerar un caso de uso como un EBP si se requieren dos personas, o si una persona tiene que pasear? Probablemente no; sin embargo, la impresión general de la definición es casi correcta. No se trata de un pequeño paso como “eliminar una línea de pedido” o “imprimir el documento”. Sino que el escenario principal de éxito está formado probablemente por cinco o diez pasos. No tarda días y múltiples sesiones, como “negociar un contrato con el proveedor”; es una tarea que se aborda en una única sesión. Probablemente dura entre unos pocos minutos y una hora. Como con la definición del UP, hace hincapié en añadir valor observable y cuantificable al negocio, y llega a un acuerdo en el que el sistema y los datos se encuentran en un estado estable y consistente.

⁴ EBP es parecido al término **tarea de usuario** en la ingeniería de usabilidad, aunque el significado es menos estricto en ese dominio.

Un error típico de los casos de uso es definir muchos casos de uso a un nivel muy bajo; es decir, como un paso simple, subfunción o subtarea en un EBP.

Violaciones razonables de la guía EBP

Aunque los casos de uso “base” de una aplicación deberían satisfacer la guía EBP, normalmente es útil crear “sub” casos de uso separados que representan subtareas, o pasos, en un caso de uso base. Pueden existir casos de uso que no sean EBP; potencialmente existen muchos a un nivel inferior. La guía sólo se utiliza para encontrar el nivel dominante de casos de uso en el análisis de requisitos de una aplicación; esto es, el nivel en el que nos tenemos que centrar para nombrarlos y escribirlos.

Por ejemplo, una subtarea o extensión como “pago a crédito” podría repetirse en varios casos de uso base. Es conveniente separarlo en un caso de uso propio (que no satisface la guía EBP) y conectarlo a varios casos de uso base, para evitar duplicaciones del texto.

En el Capítulo 25 estudiaremos las cuestiones acerca de las relaciones de casos de uso.

Casos de uso y objetivos

Los actores tienen objetivos (o necesidades) y utilizan las aplicaciones para ayudarles a satisfacerlos. En consecuencia, un caso de uso de nivel EBP se denomina caso de uso de nivel de **objetivo de usuario**, para remarcar que sirve (o debería servir) para satisfacer un objetivo de un usuario del sistema, o el actor principal.

Esto nos lleva a recomendar el siguiente procedimiento:

1. Encontrar los objetivos de usuario.
2. Definir un caso de uso para cada uno.

Esto supone un ligero cambio de énfasis para el modelador de casos de uso. En lugar de preguntar: “¿Cuáles son los casos de uso?”, uno comienza preguntando: “¿Cuáles son tus objetivos?”. De hecho, el nombre del caso de uso para un objetivo de usuario debería reflejar dicho objetivo, para resaltar este punto de vista. Objetivo: capturar o procesar una venta; caso de uso: *Procesar Venta*.

Nótese que debido a esta simetría, la guía EBP se puede aplicar igualmente para decidir si un objetivo o un caso de uso se encuentran a un nivel adecuado.

Por tanto, he aquí una idea clave con respecto a la investigación de los objetivos de usuario frente a la investigación de casos de uso:

Imagine que estamos juntos en un taller de requisitos. Nos podríamos preguntar:

- “¿Qué haces?” (una pregunta bastante orientada al caso de uso) o;
- “¿Cuáles son tus objetivos?”

Es más probable que las respuestas a la primera pregunta reflejen soluciones y procedimientos actuales, y las complicaciones asociadas a ellos.

Las respuestas a la segunda pregunta, especialmente combinada con una investigación para ascender en la jerarquía de objetivos (“¿cuál es el objetivo de ese objetivo?”), abren la visión de soluciones nuevas y mejoradas, centradas en añadir valor al negocio, y llegar al corazón de lo que el personal involucrado quiere del sistema que se está estudiando.

Ejemplo: aplicación de la guía EBP

Un analista de sistemas responsable del descubrimiento de los requisitos del sistema NuevaEra, está investigando los objetivos de usuario. La conversación transcurre de esta forma durante un taller de requisitos:

Analista de sistemas: “¿Cuáles son algunos de sus objetivos en el contexto de uso de un sistema PDV?”

Cajero: “Uno, iniciar la sesión rápidamente. También, capturar las ventas.”

Analista de sistemas: “¿Cuál cree que es el objetivo de nivel más alto que motiva el inicio de sesión?”

Cajero: “Intento identificarme en el sistema, de este modo puede validar que estoy autorizado para utilizar el sistema que captura ventas y otras tareas.”

Analista de sistemas: “¿Más alto que ése?”

Cajero: “Evitar robos, alteración de datos, y mostrar información privada de la compañía.”

Obsérvese que la estrategia del analista de buscar de manera ascendente en la jerarquía de objetivos para encontrar los objetivos de usuario de un nivel superior que todavía satisfagan la guía EBP, para obtener el objetivo real detrás de la acción, y también para entender el contexto de los objetivos.

“Evitar robos...” es de un nivel superior a un objetivo de usuario; podría llamarse objetivo de empresa, y no es un EBP. Por tanto, aunque puede inspirar nuevas formas de pensar en el problema y las soluciones (como eliminar sistemas PDV y cajeros completamente), lo vamos a dejar a un lado por ahora.

Disminuyendo el nivel del objetivo a “identificarme en el sistema y ser validado” se acerca al nivel de objetivo de usuario. ¿Pero es el nivel EBP? No añade valor observable o cuantificable al negocio. Si el responsable del comercio pregunta: “¿Qué hiciste hoy?” y dices “Inicié la sesión 20 veces！”, no se impresionaría. En consecuencia, es un objetivo secundario, siempre disponible para hacer algo útil, y no es un EBP u objetivo de usuario. En cambio, “capturar una venta” cumple el criterio para ser un EBP u objetivo de usuario.

Otro ejemplo podría ser, en algunas tiendas existe un proceso denominado “abrir caja”, en el cual un cajero inserta su propia bandeja del cajón de caja en el terminal, inicia la sesión, e indica al sistema el dinero que hay en caja. *Abrir caja* es un caso de uso de nivel EBP (o nivel de objetivo de usuario); el paso de inicio de sesión, en lugar de ser un caso de uso de nivel EBP, es un objetivo de subfunción para llevar a cabo el objetivo de abrir caja.

Objetivos y casos de uso de subfunción

Aunque “identificarme y ser validado” (o “iniciar sesión”) se ha eliminado como objetivo de usuario, es un objetivo de nivel más bajo, denominado **objetivo de subfunción** —subobjetivos que dan soporte a un objetivo de usuario—. Sólo deberían escribirse casos de uso de manera ocasional para estos objetivos de subfunción, aunque es un problema típico que observan los expertos cuando se les pide que evalúen y mejoren (normalmente que simplifiquen) un conjunto de casos de uso.

No es ilegal escribir casos de uso para objetivos de subfunción, pero no siempre es útil, ya que añade complejidad al modelo de casos de uso; puede haber cientos de objetivos de subfunción —o casos de uso de subfunción— en un sistema.

Un punto importante es que el número y granularidad de los casos de uso influyen en el tiempo y la dificultad para entender, mantener y gestionar los requisitos.

El motivo válido, más común para representar un objetivo de subfunción como un caso de uso, es cuando la subfunción se repite o es una precondición en muchos casos de uso de nivel de objetivos de usuario. Este hecho se cumple probablemente en el caso de “identificarme y ser validado”, que es una precondición de la mayoría, si no todos, los otros casos de uso de nivel de objetivos de usuario.

En consecuencia, podría escribirse como el caso de uso *Autenticar Usuario*.

Objetivos y casos de uso pueden ser compuestos

Normalmente, los objetivos son compuestos, desde el nivel de empresa (“ser rentable”), que incluyen muchos objetivos intermedios a nivel de uso de la aplicación (“se capturan las ventas”), que a su vez incluyen objetivos de subfunción dentro de las aplicaciones (“la entrada es válida”).

De manera análoga, los casos de uso se pueden escribir a niveles diferentes para satisfacer estos objetivos, y pueden estar compuestos de casos de uso de nivel inferior.

Estos diferentes niveles de objetivos y casos de uso son una fuente típica de confusión en la identificación del nivel adecuado de los casos de uso de una aplicación. La guía EBP proporciona una orientación para eliminar casos de uso de nivel excesivamente bajo.

6.9. Descubrimiento de actores principales, objetivos y casos de uso

Los casos de uso se definen para satisfacer los objetivos de usuario de actores principales. Por tanto, el procedimiento básico es:

1. Elegir los límites del sistema. ¿Es sólo una aplicación software, el hardware y la aplicación como un todo, que lo utiliza más de una persona o una organización completa?

2. Identificar los actores principales —aquellos que tienen objetivos de usuario que se satisfacen mediante el uso de los servicios del sistema—.
3. Para cada uno, identificar sus objetivos de usuario. Elevarlos al nivel de objetivos de usuario más alto que satisfaga la guía EBP.
4. Definir los casos de uso que satisfagan los objetivos de usuario; nombrarlos de acuerdo con sus objetivos. Normalmente, los casos de uso del nivel de objetivo de usuario se corresponderán uno a uno con los objetivos de usuario, aunque hay al menos una excepción, como se verá.

Paso 1: Elegir el límite del sistema

Para este caso de estudio, el propio sistema PDV es el sistema que se está diseñando; todo lo que queda fuera de él, está fuera de los límites del sistema, incluyendo el cajero, el servicio de autorización de pagos, etcétera.

Si no está clara la definición de los límites del sistema que se está diseñando, se puede aclarar definiendo lo que está fuera —los actores principales externos y de apoyo—. Una vez identificados los actores externos, los límites se vuelven más claros. Por ejemplo, ¿se encuentra la responsabilidad de autorización de pagos completa en los límites del sistema? No, hay un actor del servicio externo de autorización de pagos.

Pasos 2 y 3: Identificar los actores principales y objetivos

Es artificial establecer de manera estricta que la identificación de los actores principales es antes que los objetivos de usuario; en un taller de requisitos, la gente pone en común sus ideas y dan lugar a una mezcla de ambos. Algunas veces los objetivos ponen de manifiesto a los actores, o viceversa.

Guía: Centrar la discusión en los actores principales en primer lugar, ya que establece el marco para las investigaciones posteriores.

Preguntas útiles para encontrar los actores principales y objetivos

Además de los actores principales y objetivos de usuario obvios, las preguntas siguientes ayudan a identificar otros que se podrían haber pasado:

¿Quién arranca y para el sistema?	¿Quién se encarga de la administración del sistema?
¿Quién gestiona a los usuarios y la seguridad?	¿Es un actor el “tiempo” porque el sistema hace algo como respuesta a un evento de tiempo?
¿Existe un proceso de control que reinicie el sistema si falla?	¿Quién evalúa la actividad o el rendimiento del sistema?

¿Cómo se gestionan las actualizaciones de software?
¿Actualizaciones automáticas o no?

¿Quién evalúa los registros?
¿Se recuperan de manera remota?

Actores principales y de apoyo

Recordemos que los actores principales tienen objetivos de usuario que se satisfacen mediante el uso de los servicios del sistema. Acuden al sistema para que les ayude. Al contrario que los *actores de apoyo*, que proporcionan servicios al sistema que se está diseñando. Por ahora, nos centraremos en encontrar los actores principales, no los de apoyo.

Recordemos también que los actores principales pueden ser —entre otras cosas— otros sistemas informáticos, como procesos software “guardianes” (“watchdog”).

Sugerencia

Desconfía si ninguno de los actores principales se corresponde con un sistema informático externo.

La lista actor-objetivo

Recoge los actores principales y sus objetivos de usuario en una lista actor-objetivo. En términos de los artefactos del UP, debería corresponderse con una sección del artefacto Visión (que se describirá en el capítulo siguiente).

Por ejemplo:

Actor	Objetivo	Actor	Objetivo
Cajero	procésar ventas procesar alquileres gestionar las devoluciones abrir caja cerrar caja ...	Administrador del Sistema	añadir usuarios modificar usuarios eliminar usuarios gestionar seguridad gestionar las tablas del sistema ...
Director	poner en marcha suspender operación ...	Sistema de Actividad de Ventas	analizar los datos de ventas y rendimiento
...

El Sistema de Actividad de Ventas es una aplicación remota a la que se le solicitará con frecuencia datos de ventas desde cada nodo PDV en la red.

Dimensión de la planificación del proyecto

En la práctica, esta lista tiene columnas adicionales para la prioridad, esfuerzo y riesgo; esto se tratará brevemente en el Capítulo 36.

La complicada realidad

Esta lista parece ordenada, pero la realidad de su creación es cualquier cosa salvo eso. Son necesarias muchas “tormentas de ideas” y discusiones durante los talleres de requisitos. Consideremos el ejemplo anterior que ilustraba la aplicación de la regla EBP al objetivo de “iniciar sesión”. Durante el taller, mientras se crea esta lista, el cajero podría proponer “iniciar sesión” como un objetivo de usuario. El analista de sistemas profundizará y subirá el nivel del objetivo, más allá del mecanismo de bajo nivel de inicio de sesión (el cajero posiblemente estaba pensando en utilizar un cuadro de diálogo en una GUI), al nivel de “identificar y autenticar al usuario”. Pero, el analista entonces se da cuenta de que no cumple la guía EBP, y lo descarta como objetivo de usuario. Por supuesto, la realidad es incluso algo diferente a esto puesto que un analista con experiencia cuenta con un conjunto de heurísticas, procedentes de experiencias o estudios anteriores, una de las cuales es “la autenticación de los usuarios es rara vez un EBP”, y de este modo es probable que se elimine rápidamente.

El actor principal y los objetivos de usuario dependen del límite del sistema

¿Por qué es el cajero, y no el cliente, el actor principal del caso de uso de *Procesar Venta*? ¿Por qué no aparece el cliente en la lista actor-objetivo?

La respuesta depende del límite del sistema que se esté diseñando, como se ilustra en la Figura 6.1. Si consideramos la empresa o el servicio de caja como un sistema agregado, el cliente es un actor principal, con el objetivo de obtener artículos o servicios, y marcharse. Sin embargo, desde el punto de vista de únicamente el sistema PDV (que es la elección del límite del sistema para este caso de estudio), éste atiende el objetivo del cajero (y de la tienda) de procesar la venta del cliente.

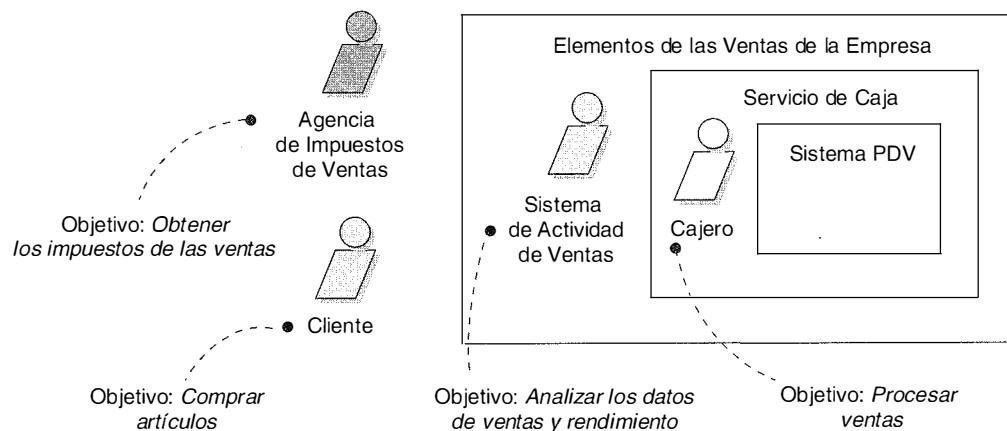


Figura 6.1. Actores principales y objetivos con diferentes límites del sistema.

Actores y objetivos por medio del análisis de eventos

Otro enfoque para ayudar en la búsqueda de los actores, objetivos y casos de uso, es identificar los eventos externos. ¿Cuáles son, de dónde proceden y por qué? A menudo,

un grupo de eventos pertenecen al mismo objetivo o caso de uso de nivel EBP. Por ejemplo:

<i>Evento Externo</i>	<i>Parte del Actor</i>	<i>Objetivo</i>
introducir línea de venta	Cajero	procesar una venta
introducir el pago	Cliente o Cajero	procesar una venta
...	...	

Paso 4: Definir los casos de uso

Por lo general, definimos un caso de uso de nivel EBP por cada objetivo de usuario. Nombramos el caso de uso de manera similar al objetivo de usuario; por ejemplo, Objetivo: procesar una venta; Caso de Uso: *Procesar Venta*.

También, nombre los casos de uso comenzando con un verbo.

Una excepción típica a un caso de uso por objetivo, es agrupar objetivos separados CRUD⁵ (crear, recuperar, actualizar, eliminar) en un caso de uso CRUD, llamado, por convención, *Gestionar<X>*. Por ejemplo, los objetivos “editar usuario”, “eliminar usuario”, etcétera, todos se satisfacen en el caso de uso *Gestionar Usuarios*.

“Definir casos de uso” comprende varios niveles de esfuerzo, variando desde unos pocos minutos, para recoger simplemente los nombres, hasta semanas, con el fin de escribir las versiones en formato completo. La última sección del proceso UP de este capítulo, sitúa este trabajo —cuándo y cuánto— en el contexto del desarrollo iterativo y del UP.

6.10. Enhorabuena: se han escrito los casos de uso y no son perfectos

La necesidad de comunicación y participación

El equipo del PDV NuevaEra va escribiendo casos de uso en múltiples talleres de requisitos, a través de una serie de iteraciones de desarrollo cortas, añadiendo incrementalmente al conjunto, y refinando y adaptando en base a la retroalimentación. Los expertos en la materia de estudio, los cajeros y programadores participan activamente en el proceso de escritura. No existen intermediarios entre los cajeros, otros usuarios y los desarrolladores, sino que se establece una comunicación directa.

Está bien, pero no es suficiente. Las especificaciones de requisitos escritas dan la impresión de ser correctas; pero no lo son. Los casos de uso y otros requisitos todavía

⁵ *N. del T.*: Acrónimo de Create-Retrieve-Update-Delete.

no serán correctos —garantizado—. Carecen de información crítica y contienen afirmaciones erróneas. La solución no es la aptitud del proceso “en cascada” de esforzarse mucho por recopilar los requisitos perfectos y completos al principio, aunque, por supuesto, lo hacemos lo mejor que podemos en el tiempo disponible. Pero nunca será suficiente.

Se necesita un enfoque diferente. Una buena parte del enfoque es el desarrollo iterativo, pero se necesita algo más: *comunicación personal continua*. Comunicación y participación cercana y continua —diaria— entre los desarrolladores y alguien que entienda el dominio y pueda tomar decisiones sobre los requisitos. Alguno de los programadores se puede acercar y en cuestión de segundos obtener aclaraciones, en cualquier momento que surja una duda. Por ejemplo, las prácticas de XP [Beck00] incluyen una excelente recomendación: *los usuarios deben tener dedicación a tiempo completo al proyecto, permaneciendo en la sala del proyecto*.

6.11. Escritura de casos de uso en un estilo esencial independiente de la interfaz de usuario

¡Nuevo y mejorado! Razones a favor de utilizar las huellas dactilares

Investigar y preguntar acerca de los objetivos, en lugar de las tareas y procedimientos fomenta que se centre la atención en la esencia de los requisitos —la intención detrás de ellos—. Por ejemplo, durante un taller de requisitos, el cajero podría decir que uno de sus objetivos es “iniciar la sesión”. El cajero, probablemente, estaría pensando en una GUI, cuadro de diálogo, identificador de usuario (ID) y contraseña (*password*). Éste es un mecanismo para alcanzar un objetivo, en lugar del objetivo en sí mismo. Mediante la investigación ascendente de la jerarquía de objetivos (“¿Cuál es el objetivo del objetivo?”), el analista del sistema llega hasta el objetivo independiente del mecanismo: “identificarse y conseguir la autenticación”, o incluso un objetivo de nivel superior: “prevenir robos...”.

Este proceso de descubrimiento puede abrir la visión a soluciones nuevas y mejoradas. Por ejemplo, los teclados y ratones con lectores biométricos, normalmente para las huellas dactilares son ahora habituales y baratos. Si el objetivo es “identificación y autenticación”, ¿por qué no hacerlo rápido y fácil, utilizando un lector biométrico en el teclado? Pero, la respuesta adecuada conlleva también un análisis de usabilidad, como conocer el perfil de los usuarios típicos del sistema. ¿Qué pasa si tienen los dedos cubiertos de grasa? ¿Tienen dedos?

Escritura en estilo esencial

Esta idea se ha resumido en varias guías de casos de uso como “no considere la interfaz de usuario; céntrese en la intención”. [Cockburn01]. La motivación y notación la ha estudiado Larry Constantine de manera más completa, en el contexto de la creación de interfaces de usuario (UIs) mejores y de la ingeniería de usabilidad [Constantine94,

CL99]. Constantine denomina al estilo de escritura **esencial** cuando evita los detalles de UI y se centra en la intención real del usuario⁶.

En un estilo de escritura esencial, la narración se expresa al nivel de las *intenciones* de los usuarios y *responsabilidades* del sistema, en lugar de sus acciones concretas. Son independientes de los detalles acerca de la tecnología y los mecanismos, especialmente aquellos relacionados con la UI.

Escriba casos de uso en un estilo esencial; no considere la interfaz de usuario y céntrese en la intención del actor.

Todos los ejemplos de casos de uso anteriores de este capítulo, como *Procesar Venta*, se escribieron con la intención de seguir un estilo esencial.

Nótese que el diccionario define *objetivo* como sinónimo de intención [MW89], lo que ilustra la conexión entre la idea del estilo *esencial* de Constantine y el punto de vista orientado al objetivo que se puso de relieve anteriormente en este capítulo. De hecho, muchos pasos de las *intenciones* de los actores, también se pueden caracterizar como *objetivos* de subfunción.

Ejemplos de contraste

Estilo esencial

Supongamos que el caso de uso *Gestionar Usuarios* requiere identificación y autenticación. El estilo esencial, inspirado en Constantine, utiliza el formato en dos columnas. Sin embargo, se puede escribir en una columna.

...	Intención del Actor	Responsabilidad del Sistema
	1. El Administrador se identifica	2. Autenticar la identidad
	3. ...	

En el formato de una columna esto se muestra como:

...
1. El Administrador se identifica
2. El Sistema autentica la identidad
3. ...

La solución de diseño para estas intenciones y responsabilidades está muy abierta: lectores biométricos, interfaces gráficas de usuario (GUIs), etcétera.

⁶ El término procede de los “modelos esenciales” en el *Ánalisis de Sistemas Esenciales* [MP84].

Estilo concreto—evitar durante el trabajo de requisitos inicial

En contraste, hay un estilo de **caso de uso concreto**. En este estilo, se incluyen en el texto del caso de uso las decisiones acerca de la interfaz de usuario. El texto podría incluso mostrar instantáneas de las pantallas de las ventanas, discutir la navegación de las ventanas, la manipulación de los elementos de la GUI, etcétera. Por ejemplo:

-
- ...
1. El Administrador introduce su ID y contraseña en el cuadro de diálogo (ver Dibujo 3).
 2. El Sistema autentica al Administrador.
 3. El Sistema muestra la ventana de “edición de usuarios” (ver Dibujo 4).
 4. ...

Estos casos de uso concretos podrían ser útiles para ayudar en el diseño de la GUI detallada y concreta en etapas posteriores, pero no son adecuados durante el trabajo del análisis de requisitos inicial. Durante el trabajo de requisitos inicial, “no considere la interfaz de usuario, céntrese en la intención”.

6.12. Actores

Un actor es cualquier cosa con comportamiento, incluyendo el propio sistema que se está estudiando (*SuD*, *System under Discussion*) cuando solicita los servicios de otros sistemas⁷. Los actores principales y de apoyo aparecerán en los pasos de acción del texto del caso de uso. Los actores no son solamente roles que juegan personas, sino también organizaciones, software y máquinas. Hay tres tipos de actores externos con relación al SuD:

- **Actor principal:** tiene objetivos de usuario que se satisfacen mediante el uso de los servicios del SuD. Por ejemplo, el cajero.
 - ¿Por qué se identifica? Para encontrar los objetivos de usuario, los cuales dirigen los casos de uso.
- **Actor de apoyo:** proporciona un servicio (por ejemplo, información) al SuD. El servicio de autorización de pago es un ejemplo. Normalmente se trata de un sistema informático, pero podría ser una organización o una persona.
 - ¿Por qué se identifica? Para clarificar las interfaces externas y los protocolos.
- **Actor pasivo:** está interesado en el comportamiento del caso de uso, pero no es principal ni de apoyo; por ejemplo, la agencia tributaria del gobierno.
 - ¿Por qué se identifica? Para asegurar que *todos* los intereses necesarios se han identificado y satisfecho. Los intereses de los actores pasivos algunas veces son sutiles o es fácil no tenerlos en cuenta, a menos que estos actores sean identificados explícitamente.

⁷ Éste fue un refinamiento y mejora para las definiciones de actores alternativas, incluyendo las de las primeras versiones de UML y el UP [Cockburn97]. Las antiguas definiciones excluían, de manera inconsistente, el SuD como actor, incluso cuando recurriía a servicios de otros sistemas. Todas las entidades podrían jugar múltiples *roles*, incluyendo el SuD.

6.13. Diagramas de casos de uso

UML proporciona notación para los diagramas de casos de uso con el fin de ilustrar los nombres de los casos de uso y los actores, y las relaciones entre ellos (ver Figura 6.2).

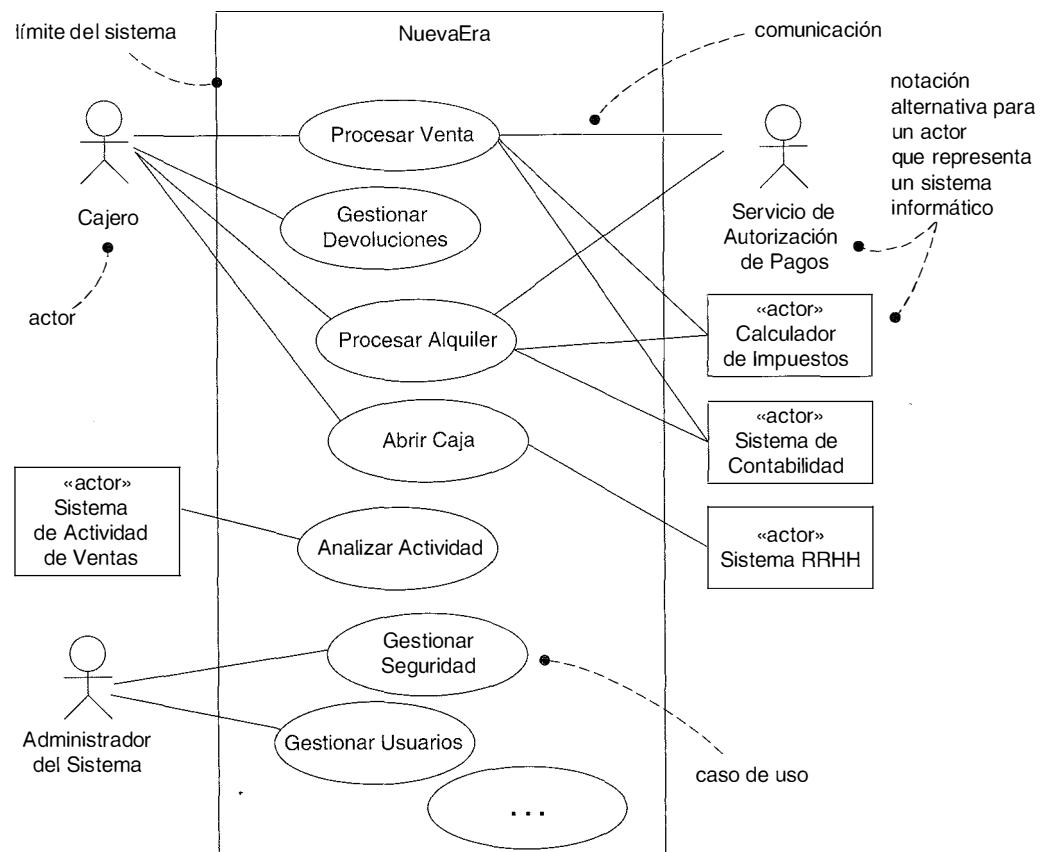


Figura 6.2. Diagrama de contexto de casos de uso parcial.

Los diagramas de caso de uso y las relaciones entre los casos de uso son secundarios en el trabajo con los casos de uso. Los casos de uso son documentos de texto. Trabajar con los casos de uso significa escribir texto.

Un signo típico de un modelador de casos de uso novato (o un estudiante) es la preocupación por los diagramas de casos de uso y las relaciones entre los casos de uso, en lugar de escribir texto. Los expertos mundiales en casos de uso, como Anderson, Fowler, Cockburn, entre otros, minimizan la importancia de los diagramas de casos de uso y las relaciones entre los casos de uso, y en lugar de eso se centran en escribir. Con eso como advertencia, un simple diagrama de casos de uso proporciona un conciso diagrama de contexto visual del sistema, que muestra los actores externos y cómo utilizan el sistema.

Sugerencia

Dibuje un diagrama de casos de uso sencillo junto con la lista actor-objetivo.

Un diagrama de casos de uso es una excelente representación del contexto del sistema; conforma un buen **diagrama de contexto**, esto es, muestra los límites de un sistema, lo que permanece fuera de él, y cómo se utiliza. Sirve como herramienta de comunicación que resume el comportamiento de un sistema y sus actores. La Figura 6.2 presenta una muestra de diagrama de contexto de casos de uso *parcial* para el sistema NuevaEra.

Sugerencias en la realización de los diagramas

La Figura 6.3 muestra algunos consejos sobre los diagramas. Nótese que la caja del actor contiene el símbolo «actor». Este símbolo se denomina **estereotipo UML**; se trata de un mecanismo para clasificar un elemento en cierto modo. El nombre de un estereotipo se escribe entre comillas francesas —signo ortográfico especial formado por un *único* carácter (no “<<” y “>>”) conocido sobre todo por su uso en la tipografía francesa para indicar una cita—.

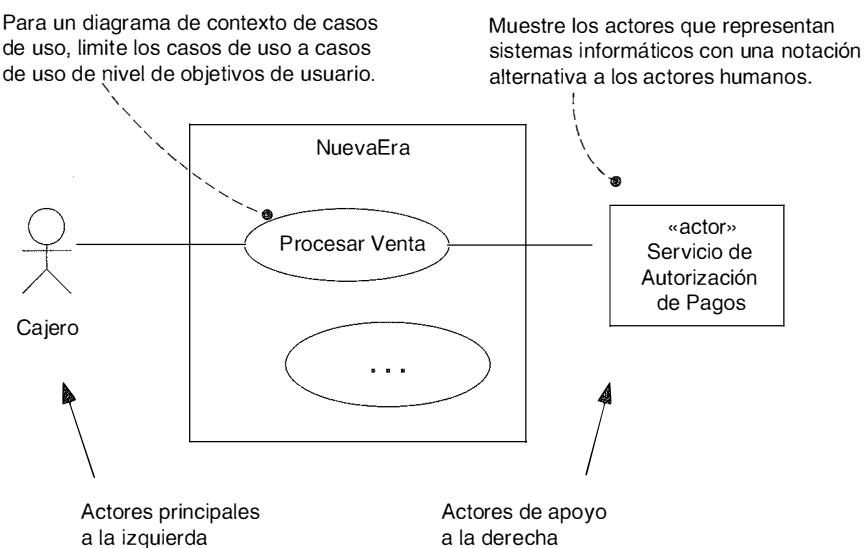


Figura 6.3. Sugerencias sobre la notación.

Para clarificar, algunos prefieren destacar los actores que se corresponden con sistemas informáticos externos con una notación alternativa, como se ilustra en la Figura 6.4.

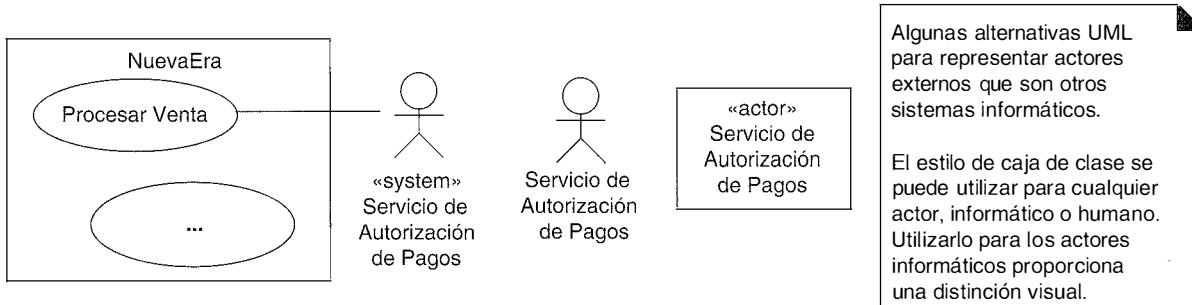


Figura 6.4. Notación alternativa para los actores.

Advertencia sobre el exceso de diagramas

Reiteramos que la importancia del trabajo de los casos de uso es escribir texto, no diagramas o centrarse en las relaciones entre los casos de uso. Si una organización dedica muchas horas (o peor, días) trabajando en los diagramas de casos de uso y debatiendo las relaciones en los casos de uso, en lugar de centrarse en escribir texto, se ha desaprovechado el esfuerzo.

6.14. Requisitos en contexto y listas de características de bajo nivel

Como queda implícito en el título del libro *Uses Cases: Requirements in Context* [GK00], una motivación clave de la idea de caso de uso es considerar y organizar los requisitos en el contexto de los objetivos y escenarios de uso de un sistema. Eso es bueno —mejora la cohesión y comprensión—. Sin embargo, los casos de uso no son los únicos artefactos de requisitos necesarios. Algunos requisitos no funcionales, reglas de dominio y contexto, y otros elementos difíciles de ubicar, se capturan mejor en la Especificación Complementaria, que se describirá en el siguiente capítulo.

Una idea detrás de los casos de uso es sustituir las listas de características de bajo nivel (típicas de los métodos de requisitos tradicionales) por los casos de uso (con algunas excepciones). Estas listas tendían a ser como sigue, normalmente agrupadas por áreas funcionales:

<i>ID</i>	<i>Característica</i>
CARAC1.9	El Sistema aceptará entradas de los identificadores de los artículos.
...	...
CARAC2.4	El Sistema registrará los pagos a crédito en el sistema de cuentas por cobrar.

De tales listas detalladas de las características de bajo nivel se puede aprovechar algo. Sin embargo, la lista completa no se recoge en media página; es más probable que ocupe docenas o cientos de páginas. Esto nos conduce a algunos obstáculos, que los casos de uso ayudan a abordar. Éstos incluyen:

- Listas de funciones largas y detalladas no relacionan los requisitos en un contexto cohesivo; las diferentes funciones y características aparecen progresivamente como una “lista de la lavandería” inconexa de elementos. En cambio, los casos de uso sitúan los requisitos en el contexto de las historias y objetivos de uso del sistema.
- Si se utilizan tanto los casos de uso como las listas de características detalladas, hay duplicación. Más trabajo, mayor volumen para escribir y leer, más problemas de consistencia y sincronización.

Sugerencia

Esfuércese en sustituir las listas detalladas de características de bajo nivel por casos de uso.

Son aceptables las listas de características de alto nivel del sistema

Es típico y útil resumir la funcionalidad del sistema con una lista breve de características de alto nivel, denominadas características del sistema, en un documento de Visión. A diferencia de las 100 páginas de características detalladas de bajo nivel, la lista de características del sistema tiende a incluir sólo unas pocas docenas de elementos. La lista proporciona un resumen muy conciso de la funcionalidad del sistema, independiente de la vista del caso de uso. Por ejemplo:

Resumen de características del sistema

- capturar las ventas.
- autorización del pago (crédito, débito, cheque).
- administración del sistema para los usuarios, seguridad, tablas de códigos y constantes, etcétera.
- procesamiento automático de ventas sin conexión, cuando fallan los componentes externos.
- transacciones en tiempo real, en base a los estándares industriales, con sistemas de terceras partes, que incluye inventario, contabilidad, recursos humanos, cálculo de impuestos y servicios de autorización de pagos.
- definición y ejecución de reglas de negocio adaptadas y que se pueden añadir en ejecución en puntos típicos, fijados en los escenarios de proceso.
- ...

Esto se estudiará en el siguiente capítulo.

¿Cuándo son apropiadas las listas de características detalladas?

Algunas veces los casos de uso no encajan realmente; algunas aplicaciones exigen un punto de vista dirigido por las características. Por ejemplo, los servidores de aplicaciones, productos de bases de datos y otros sistemas *middleware* o *back-end* necesitan ante todo ser considerados y evolucionar en términos de las *características* (“Necesitamos el soporte de XML en la siguiente versión”). Los casos de uso no se ajustan de manera natural a estas aplicaciones o al modo en que necesitan evolucionar en términos de las fuerzas del mercado.

6.15. Los casos de uso no son orientados a objetos

No hay nada orientado a objetos en los casos de uso; uno no está realizando un análisis orientado a objetos si escribe casos de uso. Esto no es un defecto sino una aclaración. De hecho, los casos de uso constituyen una herramienta para el análisis de requisitos ampliamente aplicable, que se puede utilizar en proyectos no orientados a objetos, lo cual

incrementa su utilidad como método de requisitos. Sin embargo, como estudiaremos, los casos de uso son una entrada fundamental en las actividades clásicas de A/DOO.

6.16. Casos de uso en el UP

Los casos de uso son vitales y centrales en el UP, que fomentan el **desarrollo dirigido por casos de uso**. Esto implica:

- Los requisitos se recogen principalmente en casos de uso (el Modelo de Casos de Uso); otras técnicas de requisitos (como las listas de funciones) son secundarias, si es que se utilizan.
- Los casos de uso son una parte importante de la planificación iterativa. El trabajo de una iteración se define —en parte— eligiendo algunos escenarios de caso de uso, o casos de uso completos. Los casos de uso son una entrada clave para hacer estimaciones.
- Las **realizaciones de los casos de uso** dirigen el diseño. Es decir, el equipo diseña objetos y subsistemas que colaboran para ejecutar o realizar los casos de uso.
- Los casos de uso, a menudo, influyen en la organización de los manuales de usuario.

El UP diferencia entre casos de uso del sistema y del negocio. Los **casos de uso del sistema** son los que se han estudiado en este capítulo, como *Procesar Venta*. Se crean en la disciplina Requisitos, y forman parte del Modelo de Casos de Uso.

Los **casos de uso del negocio** se escriben con menos frecuencia. Si se hace, se crean en la disciplina Modelado del Negocio, como parte de un esfuerzo de reingeniería de los procesos de negocio a gran escala, o para ayudar a entender el contexto de un nuevo sistema en el negocio. Describen una secuencia de acciones de un negocio como un todo para cumplir un objetivo de un **actor del negocio** (un actor en el entorno del negocio, como un cliente o un proveedor). Por ejemplo, en un restaurante, un caso de uso del negocio es *Servir una Comida*.

Casos de uso y especificación de requisitos a lo largo de las iteraciones

Esta sección reitera la idea clave del UP y el desarrollo iterativo: medir el tiempo y el nivel de esfuerzo de la especificación de requisitos a lo largo de las iteraciones. La Tabla 6.1 presenta una muestra (no una receta) de la estrategia del UP sobre el modo de desarrollar los requisitos.

Nótese que el equipo técnico comienza construyendo los fundamentos de la producción del sistema cuando, quizás, sólo se han detallado el 10% de los requisitos, y de hecho, se retrasa de manera deliberada la continuación del trabajo de los requisitos concertados hasta casi el final de la primera iteración de elaboración.

Ésta es la diferencia clave entre el proceso iterativo y el proceso en cascada: el desarrollo con calidad de producción de los fundamentos del sistema comienza rápidamente, mucho antes de conocer todos los requisitos.

Obsérvese que cerca del final de la primera iteración de elaboración, hay un segundo taller de requisitos, durante el que quizás, el 30% de los casos de uso se escriben en

Tabla 6.1. Muestra del esfuerzo de los requisitos a lo largo de las primeras iteraciones; no es una receta.

Disciplina	Artefacto	Comentarios y nivel de esfuerzo de los requisitos				
		Inicio 1 semana	Elab 1 4 semanas	Elab 2 4 semanas	Elab 3 3 semanas	Elab 4 3 semanas
Requisitos.	Modelo de Casos de Uso.	2 días de taller de requisitos. Se identifican por el nombre la mayoría de los casos de uso y se resumen en un párrafo breve. Sólo el 10% se escribe en detalle.	Cerca del final de esta iteración, tiene lugar un taller de requisitos de 2 días. Se obtiene un mejor entendimiento y retroalimentación a partir del trabajo de implementación, entonces se completa el 30% de los casos de uso en detalle.	Cerca del final de esta iteración, tiene lugar un taller de requisitos de dos días. Se obtiene una mejor comprensión y retroalimentación a partir del trabajo de implementación, entonces se completa el 50% de los casos de uso en detalle.	Repetir, se completa el 70% de todos los casos de uso en detalle.	Repetir con la intención declarificar y escribir en detalle del 80-90% de los casos de uso. Sólo una pequeña parte de éstos se construyen durante la elaboración; el resto se aborda durante la construcción.
Diseño.	Modelo de Diseño.	Nada.	Diseño de un pequeño conjunto de requisitos de alto riesgo significativos desde el punto de vista de la arquitectura.	Repetir.	Repetir.	Repetir. Deberían ahora estabilizarse los aspectos de alto riesgo significativos para la arquitectura.
Implementación.	Modelo de Implementación (código, etc.)	Nada.	Implementar esto.	Repetir. Se construye el 5% del sistema final.	Repetir. Se construye el 10% del sistema final.	Repetir. Se construye el 15% del sistema final.
Gestión del Proyecto.	Plan de Desarrollo de SW.	Estimación muy imprecisa del esfuerzo total.	La estimación comienza a tomar forma.	Un poco mejor...	Un poco mejor...	Ahora se pueden establecer racionalmente la duración global del proyecto, los hitos más importantes, estimación del coste y esfuerzo.

detalle. Este análisis de requisitos escalonado se beneficia de la retroalimentación a partir de la construcción de un poco del núcleo del software. La retroalimentación incluye la evaluación del usuario, pruebas y “conocimiento de lo que no conocemos” mejorado. Es decir, el acto de construir software rápidamente hace que surjan suposiciones y preguntas que necesitan aclararse.

Momento de la creación de los artefactos del UP

La Tabla 6.2 muestra algunos de los artefactos del UP y un ejemplo de la planificación de sus momentos de comienzo y refinamiento. El Modelo de Casos de Uso comienza en la fase de inicio, con quizás sólo el 10% de los casos de uso escritos con algo de detalle. La mayoría se escriben incrementalmente a lo largo de las iteraciones de la fase de elaboración, de manera que, al final de la elaboración se ha escrito un gran cuerpo de casos de uso detallados y otros requisitos (en la Especificación Complementaria), proporcionando una base realista para hacer una estimación precisa hasta el final del proyecto.

Tabla 6.2. Muestra de los artefactos UP y evolución temporal. c - comenzar; r - refiniar.

Disciplina	Artefacto Iteración →	Inicio I1	Elab. E1...En	Const. C1...Cn	Trans. T1...T2
Modelado del Negocio	Modelo del Dominio		c		
Requisitos	Modelo de Casos de Uso Visión Especificación Complementaria Glosario	c c c c	r r r r		
Diseño	Modelo de Diseño Documento de Arquitectura SW Modelo de Datos		c c c	r r	
Implementación	Modelo de Implementación		c	r	r
Gestión del Proyecto	Plan de Desarrollo SW	c	r	r	r
Pruebas	Modelo de Pruebas		c	r	
Entorno	Marco de Desarrollo	c	r		

Casos de uso en la fase de inicio

La siguiente discusión detalla la información presentada en la Tabla 6.1.

No todos los casos de uso se escriben en formato completo durante la fase de inicio. Más bien, suponga que se lleva a cabo un taller de requisitos durante dos días al comienzo del estudio de NuevaEra. La primera parte del día se dedica a identificar los objetivos y el personal involucrado, y especular sobre lo que queda dentro y fuera del alcance del proyecto. Se escribe una tabla de casos de uso actor-objetivo y se presenta con el proyector del ordenador. Se inicia el diagrama de contexto de casos de uso. Tras unas pocas horas, quizás se identifican unos 20 objetivos de usuario (y por tanto, casos de uso de nivel de usuario), que incluye *Procesar Venta*, *Gestionar Devoluciones*, etcétera. La mayoría de los casos de uso interesantes, complejos o arriesgados, se escriben en formato breve; cada uno escrito con una duración media de dos minutos. El equipo comienza a formarse un esquema de alto nivel de la funcionalidad del sistema.

Después de esto, entre el 10% y el 20% de los casos de uso que representan las funciones complejas principales, o que son especialmente arriesgadas en alguna dimensión, se escriben en formato completo; el equipo investiga con algo más de profundidad para entender mejor la magnitud, complejidad y demonios ocultos en el proyecto, a través de una pequeña muestra de casos de uso interesantes. Quizás esto puede implicar dos casos de uso: *Procesar Venta* y *Gestionar Devoluciones*.

Se utiliza una herramienta de gestión de requisitos integrada con un procesador de texto para la escritura, y el trabajo se muestra por medio de un proyector mientras el equipo colabora en el análisis y la escritura. Se escriben las listas de *Intereses y Personal Involucrado* para estos casos de uso, para descubrir requisitos más refinados (y quizás costosos) funcionales y no funcionales —o cualidades del sistema— claves, como la fiabilidad y el rendimiento.

El objetivo del análisis no es completar los casos de uso de manera exhaustiva, sino dedicar unas horas a comprenderlos mejor.

El promotor del proyecto necesita decidir si merece la pena un estudio profundo (esto es, la fase de elaboración). La intención del trabajo de inicio no es hacer este estudio, sino adquirir una idea de poca fidelidad (y claramente propensa a errores) acerca del alcance, riesgo, esfuerzo, viabilidad técnica, y análisis del negocio, para decir avanzar, dónde comenzar si se hace, o si parar.

Quizás la fase de inicio del proyecto NuevaEra duró cinco días. La combinación del taller de requisitos de dos días y su análisis de casos de uso breve, y otros estudios durante la semana, condujeron a tomar la decisión de continuar con la fase de elaboración para el sistema.

Casos de uso en la elaboración

La siguiente discusión detalla la información presentada en la Tabla 6.1.

Se trata de una fase de múltiples iteraciones de duración fija (por ejemplo, cuatro iteraciones) en las cuales se construyen incrementalmente partes del sistema arriesgadas, de alto valor o significativas desde el punto de vista de la arquitectura, y se identifican y clasifican la “mayoría” de los requisitos. La retroalimentación de los pasos concretos de programación influye e informa del conocimiento de los requisitos por parte del equipo, que se refina de manera iterativa y adaptable. Quizás se aborda un taller de requisitos de dos días en cada iteración —cuatro talleres—. Sin embargo, no se estudian todos los casos de uso en cada taller. Se priorizan; los primeros talleres se centran en un subconjunto de los casos de uso más importantes.

En cada siguiente taller de requisitos breve, es el momento de adaptar y refinar la visión de los requisitos principales, que serán inestables en las primeras iteraciones y se irán estabilizando en las últimas. Por tanto, hay una interacción iterativa entre el descubrimiento de los requisitos y la construcción de partes del software.

Durante cada taller de requisitos se refinan los objetivos de usuario y la lista de casos de uso. Se escriben, y reescriben, la mayoría de los casos de uso, en formato completo. Al final de la elaboración, se escriben en detalle del “80 al 90%” de los casos de uso. Para el sistema PDV con 20 casos de uso de nivel de objetivo de usuario, 15 o más de los más complejos y arriesgados deberían investigarse, escribirse y reescribirse en formato completo.

Nótese que la elaboración conlleva programar partes del sistema. Al final de esta etapa, el equipo NuevaEra no sólo debería tener una mejor definición de los casos de uso, sino también algo de software ejecutable de calidad.

Casos de uso en la construcción

La etapa de construcción está compuesta de iteraciones de duración fija (por ejemplo, 20 iteraciones de dos semanas cada una) que se centra en completar el sistema, una vez que las principales cuestiones arriesgadas e inestables se han establecido en la elaboración. Tendrá lugar todavía la escritura de casos de uso menores y quizás talleres de requisitos,

pero mucho menos de lo que se hizo en la elaboración. En esta etapa, la mayoría de los requisitos funcionales y no funcionales principales deberían haberse estabilizado de manera iterativa y adaptable. La intención no es dar a entender que los requisitos se congelan o el estudio termina, sino que el grado de cambio es mucho menor.

6.17. Caso de estudio: casos de uso en la fase de inicio de NuevaEra

Como se ha descrito en la sección anterior, no todos los casos de uso se escriben en el formato completo durante la fase de inicio. El Modelo de Casos de Uso de esta fase para el caso de estudio podría detallarse como sigue:

<i>Completo</i>	<i>Informal</i>	<i>Breve</i>
Procesar Venta	Procesar Alquiler	Abrir Caja
Gestionar Devoluciones	Analizar Actividad de Ventas	Cerrar Caja
	Gestionar Seguridad	Gestionar Usuarios
	...	Poner en Marcha
		Suspender Operación
		Gestionar Tablas del Sistema
		...

6.18. Lecturas adicionales

La guía de casos de uso de mayor éxito, traducida a varios idiomas es *Writing Effective Use Cases* [Cockburn01]⁸. Por buenas razones se ha convertido en el libro de casos de uso más ampliamente leído y seguido y, por tanto, se recomienda como referencia fundamental. Este capítulo se basa, y es consistente, con su contenido. Sugerencia: no rechace el libro como consecuencia del uso de iconos para los diferentes niveles de casos de uso por parte de los autores, o el énfasis temprano en los niveles y la taxonomía de casos de uso. Los iconos son opcionales y no muy importantes. Y aunque el debate acerca de los niveles y objetivos podría parecer al principio que distrae la atención a aquellos nuevos en los casos de uso, los que han trabajado con ellos durante algún tiempo estiman que el nivel y alcance de los casos de uso son cuestiones prácticas claves, porque su incomprendión es una fuente típica de complicaciones en el modelado de casos de uso.

“Structuring Use Cases with Goals” [Cockburn97] es el artículo sobre casos de uso más citado, disponible on-line en www.usecases.org.

Use Cases: Requirements in Context [GK00] es otro texto útil. Destaca el importante punto de vista —como establece el título— de que los casos de uso no son únicamente

⁸ Nótese que Cockburn rima con *slow burn* (*N. del T.*: En una comunicación personal, el autor nos ha comentado que introdujo esta aclaración a petición de A. Cockburn para señalar que su apellido no se pronuncia como cock-burn y evitar chistes por la connotación sexual.)

otro artefacto de los requisitos, sino que constituyen el vehículo central que dirige el trabajo de los requisitos y la información.

Otra lectura que merece la pena destacar es *Applying Use Cases: A Practical Guide* [SW98], escrita por un profesor y experto en casos de uso que entiende y comunica cómo aplicar los casos de uso en un ciclo de vida iterativo.

6.19. Artefactos UP y contexto del proceso

Como se ilustra en la Figura 6.5 los casos de uso influyen en muchos artefactos UP.

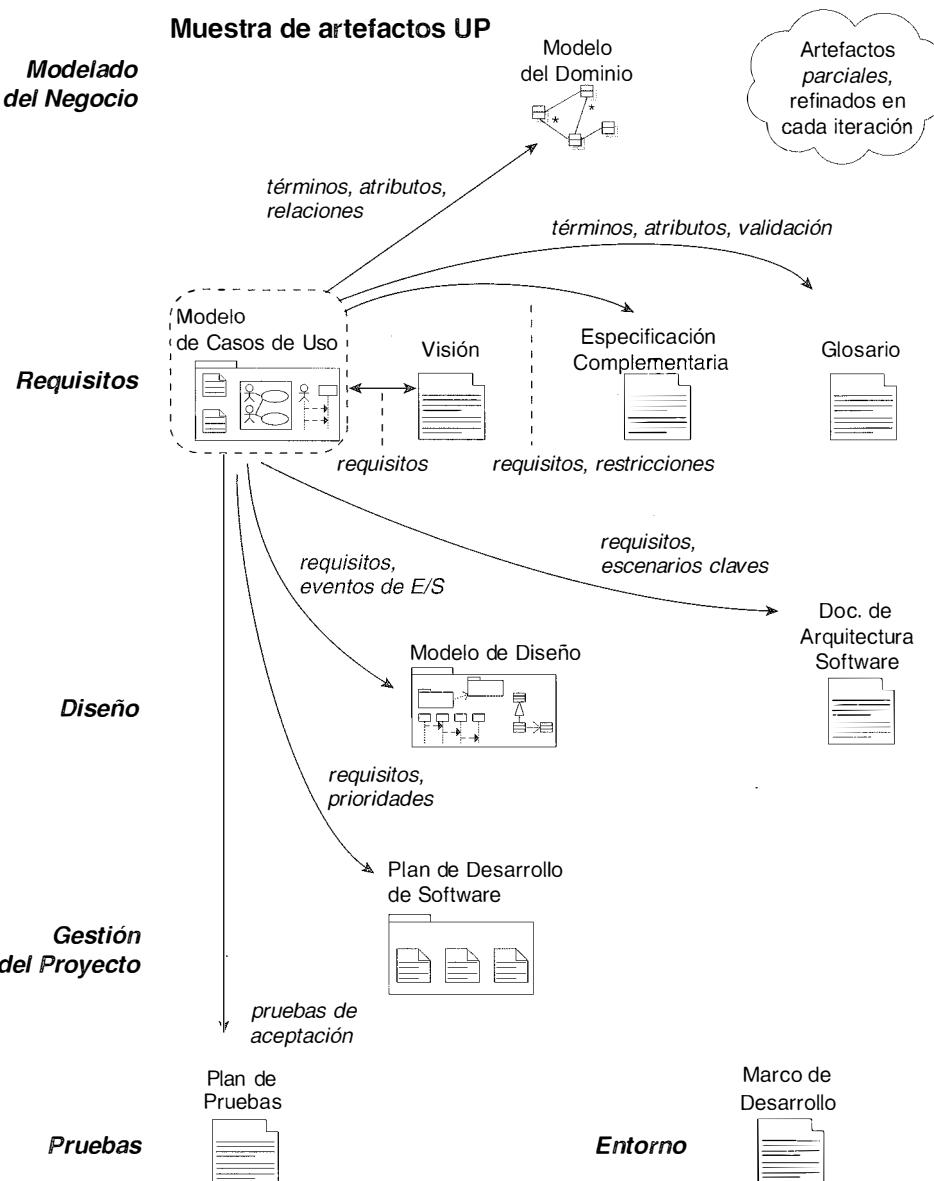


Figura 6.5. Muestra de la influencia entre los artefactos UP.

En el UP, el trabajo de los casos de uso es una actividad de la disciplina de requisitos que podría inicializarse durante un taller de requisitos. La Figura 6.6 presenta algunos consejos acerca del momento y el lugar para llevar a cabo este trabajo.

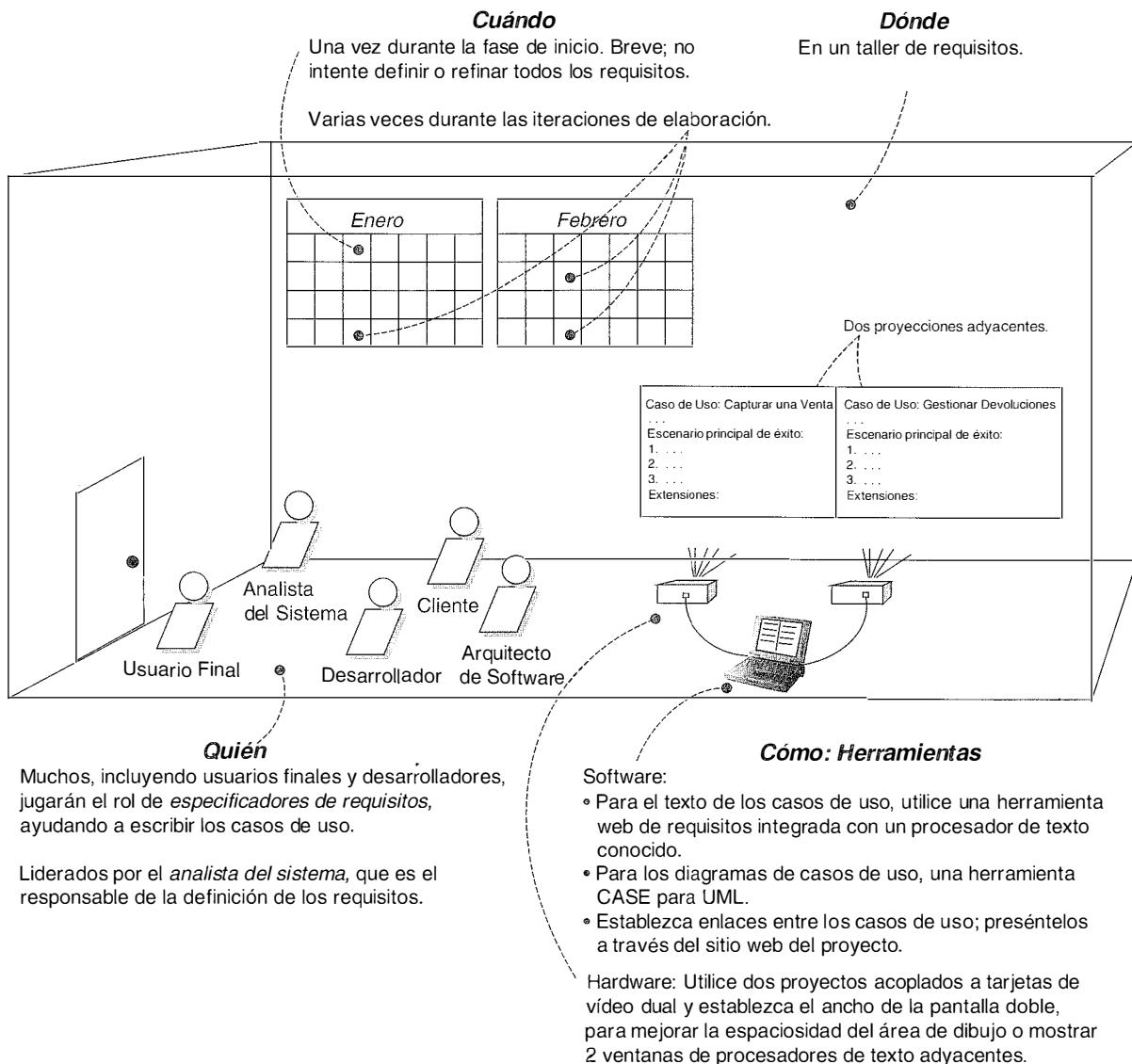


Figura 6.6. Proceso y establecimiento del contexto.

Capítulo 7

IDENTIFICACIÓN DE OTROS REQUISITOS

Cuando las ideas fallan, las palabras vienen muy bien.

Johann Wolfgang von Goethe

Objetivos

- Escribir una Especificación Complementaria, Glosario y Visión.
 - Comparar y contrastar las características del sistema con los casos de uso.
 - Relacionar la Visión con otros artefactos y con el desarrollo iterativo.
 - Definir los atributos de calidad.
-

Introducción

No es suficiente escribir casos de uso. Existen otros tipos de requisitos que son necesarios identificar, como los relacionados con documentación, empaquetado, soporte, licencia, etcétera. Éstos se recogen en la **Especificación Complementaria**.

El **Glosario** almacena los términos y definiciones; puede también jugar el rol de diccionario de datos.

La **Visión** resume la “visión” del proyecto. Sirve para comunicar de manera concisa las grandes ideas acerca de por qué se propuso el proyecto, cuáles son los problemas, quiénes son las personas involucradas, qué necesitan, y cuál podría ser la apariencia de la solución propuesta.

Citando literalmente:

La Visión define la vista que tienen las personas involucradas del producto que se va a desarrollar, especificada en términos de las necesidades y características clave de dichas personas. Al contener un esquema de los principales requisitos previstos, proporciona la base contractual para los requisitos técnicos más detallados [RUP].

7.1. Ejemplos del PDV NuevaEra

El objetivo de los siguientes ejemplos no es presentar de manera exhaustiva la Visión, el Glosario y Especificación Complementaria, ya que algunas de las secciones —aunque útiles para el proyecto— no son relevantes para los objetivos de aprendizaje¹. La finalidad del libro son las principales técnicas en el diseño de objetos, análisis de requisitos con casos de uso, análisis orientado a objetos, no los problemas del PDV o las sentencias de Visión. Por tanto, sólo se hace referencia brevemente a algunas secciones, para establecer las conexiones entre el trabajo previo y futuro, destacar las cuestiones que merecen la pena, proporcionar una idea del contenido, y avanzar rápidamente.

7.2. Ejemplo NuevaEra: Especificación Complementaria (Parcial)

Especificación Complementaria

Historia de revisiones

Versión	Fecha	Descripción	Autor
Borrador de Inicio	10 Enero, 2031	Primer borrador. Para refinarse principalmente durante la elaboración.	Craig Larman

Introducción

Este documento es el repositorio de todos los requisitos del PDV NuevaEra que no se capturan en los casos de uso.

Funcionalidad

(Funcionalidad común entre muchos casos de uso)

Registro y gestión de errores

Registrar todos los errores en almacenamiento persistente.

¹ El crecimiento gradual del alcance no es sólo un problema de los requisitos, sino de la *escritura* sobre requisitos.

Reglas de negocio conectables

En varios puntos de los escenarios de varios casos de uso (pendientes de ser definidos) soportar la capacidad de adaptar la funcionalidad del sistema con un conjunto arbitrario de reglas que se ejecutan en ese punto o evento.

Seguridad

Todo uso requiere la autenticación de los usuarios.

Facilidad de uso***Factores humanos***

El cliente será capaz de ver la información en un gran monitor del PDV. Por tanto:

- Se debe ver el texto fácilmente a una distancia de 1 metro.
- Evitar colores asociados con formas comunes de daltonismo.

Velocidad, comodidad y procesamiento libre de errores, son lo más importante en el procesamiento de ventas, ya que el comprador desea irse rápidamente, o perciben la experiencia de compra (y al vendedor) como menos positiva.

El cajero está mirando a menudo al cliente o los artículos, no a la pantalla del ordenador. Por tanto, se deben comunicar las señales y avisos con sonidos, en lugar de sólo mediante gráficos.

Fiabilidad***Capacidad de recuperación***

Si se produce algún fallo al usar un servicio externo (autorización de pago, sistema de contabilidad,...) intentar solucionarlo con una solución local (ej. almacenar y remitir) para, no obstante, completar una venta. Se necesita mucho más análisis aquí...

Rendimiento

Como se mencionó en los factores humanos, los compradores quieren completar el proceso de ventas *muy* rápido. Un cuello de botella potencial es la autorización de pagos externa. El objetivo es conseguir la autorización en menos de 1 minuto, el 90% de las veces.

Soporte***Adaptabilidad***

Los diferentes clientes del PDV NuevaEra tienen necesidades de procesamiento y reglas de negocio únicas en el procesamiento de una vena. Por tanto, en varios puntos definidos en el escenario (por ejemplo, cuando se inicia una nueva venta, cuando se añade una nueva línea de venta) se habilitarán reglas de negocio conectables.

Facilidad para cambiar la configuración

Clientes diferentes desean que varíe la configuración de la red para sus sistemas PDV, como clientes gruesos *versus* delgados, en dos capas *versus* N-capas, etcétera. Además, desean poder cambiar estas configuraciones, para reflejar sus cambios en el negocio y necesidades de rendimiento. Por tanto, el sistema será configurable para reflejar estas necesidades. Se necesita mucho más análisis en este área para descubrir las áreas y grado de flexibilidad, y el esfuerzo para conseguirla.

Restricciones de Implementación

La dirección de NuevaEra insiste en una solución utilizando las tecnologías Java, previendo que esto mejorará a largo plazo la portabilidad y soporte, además de facilitar el desarrollo.

Componentes adquiridos

- El sistema de cálculo de impuestos. Debe soportar sistemas de cálculo conectables de diferentes países.

Componentes de libre distribución

En general, recomendamos maximizar el uso de componentes Java de libre distribución en este proyecto.

Aunque es prematuro diseñar y elegir los componentes de manera definitiva, sugerimos los siguientes como candidatos posibles:

- Framework para registros JLog
- ...

Interfaces

Interfaces y hardware destacables

- Monitor con pantalla táctil (detectado por el sistema operativo como monitor corriente, y los movimientos de contacto como eventos del ratón)
- Escáner láser de código de barras (normalmente unido a un teclado especial, y el software considera las entradas escaneadas como entradas del teclado)
- Impresora de recibos
- Lector de tarjetas de crédito/débito
- Lector de firmas (pero no en la primera versión)

Interfaces software

Para la mayoría de los sistemas de colaboración externos (calculador de impuestos, contabilidad, inventario, ...) necesitamos ser capaces de conectar diversos sistemas y, por tanto, diversas interfaces.

Reglas de dominio (negocio)

ID	Regla	Grado de variación	Fuente
REGLA1	Se requiere la firma para pagos a crédito. Años la mayoría de los clientes	Se continuará solicitando la "firma" de los compradores, aunque en 2 años solicitarán la firma en un aparato de captura digital, y en 5 años esperamos que se demande la nueva "firma" digital única soportada por la ley americana.	La política de prácticamente todas las compañías de autorización de crédito.
REGLA2	Reglas sobre los impuestos. Hay que añadir un impuesto	Alto. La ley sobre impuestos cambia anualmente, en todos los niveles de gobierno.	ley

continúa

<i>ID</i>	<i>Regla</i>	<i>Grado de variación</i>	<i>Fuente</i>
	a las ventas. Ver los estatutos del gobierno para los detalles actuales.		
REGLA3	Las devoluciones de los pagos a crédito sólo pueden efectuarse como crédito en las cuentas de crédito de los compradores, no en efectivo.	Bajo.	Política de la compañía sobre la autorización de crédito.
REGLA4	Reglas de descuento al comprador. Ejemplo: Empleados: 20% Clientes preferentes: 10% Antiguos: 15%	Alto. Cada tienda utiliza reglas diferentes.	Política de la tienda.
REGLA5	Reglas de descuento de venta (nivel de transacción) Se aplica al total antes de los impuestos. Por ejemplo: 10% de descuento si el total es mayor de 100 €. 5% de descuento los lunes. 10% de descuento en las ventas de hoy entre las 10am y las 3pm. 50% de descuento en Tofu hoy desde las 9am hasta las 10am.	Alto. Cada tienda utiliza reglas distintas, y pueden cambiar diariamente o cada hora.	Política de la tienda.
REGLA6	Reglas de descuento de artículo (nivel de línea de venta). Ejemplo: 10% de descuento en tractores esta semana. Comprando 2 hamburguesas vegetales llévese 1 gratis.	Alto. Cada tienda utiliza reglas distintas, y pueden cambiar diariamente o cada hora.	Política de la tienda.

Cuestiones legales

Recomendamos algunos componentes de libre distribución si sus restricciones de licencia se pueden resolver para permitir la reventa de artículos que incluyen software de libre distribución. Todas las reglas de impuestos se tienen que aplicar, por ley, durante las ventas. Nótese que pueden cambiar con frecuencia.

Información en dominios de interés

Fijación de precios

Además de las reglas de fijación de precios que se describen en la sección de reglas del dominio, observe que los artículos tienen un *precio original*, y opcionalmente, un *precio rebajado*. El precio de los artículos (antes de los descuentos adicionales) es el precio rebajado, si existe. Las organizaciones mantienen el precio original incluso si hay un precio rebajado, por razones de contabilidad e impuestos.

Gestión de pagos a crédito y débito

Cuando el servicio de autorización de pagos aprueba un pago electrónico, de crédito o débito, son los responsables del pago al vendedor, no el comprador. En consecuencia, por cada pago, el vendedor necesita registrar la suma de dinero que le deben en las cuentas, desde el servicio de autorización. Normalmente, por las noches, el servicio de autorización llevará a cabo una transferencia electrónica de fondos a la cuenta de los vendedores por la suma de dinero total del día, menos un (pequeño) cargo por transacción que cobra el servicio.

Impuesto de ventas

Los cálculos de los impuestos de las ventas pueden ser muy complejos, y pueden cambiar regularmente como respuesta a la legislación en todos los niveles de gobierno. Por tanto, es aconsejable delegar el cálculo a software de terceras partes (de los que hay varios disponibles). Los impuestos se pueden deber a la ciudad, al gobierno regional y agencias nacionales. Algunos artículos podrían estar exentos de impuestos sin requisitos, o exentos dependiendo del comprador o el receptor objetivo (por ejemplo, un granjero o un niño).

Identificadores de artículos: UPCs, EANs, SKUs, código de barras y lectores de códigos de barras

El PDV NuevaEra necesita dar soporte a varios esquemas de identificador de artículos. UPCs (Códigos de Producto Universales, *Universal Product Codes*), EANs (Numeración de Artículos Europea, *European Article Numbering*) y SKUs (Unidades de Mantenimiento de Stock, *Stock Keeping Units*) son tres tipos comunes de sistemas de identificación para los artículos que se venden. El Número de Artículo Japonés (JANs, *Japanese Article Number*) es un tipo de la versión EAN.

SKUs son identificadores completamente arbitrarios definidos por el vendedor.

Sin embargo, los UPCs y EANs tienen un componente estándar y normativo. Diríjase a www.adams1.com/pub/russadam/upccode.html para obtener una buena visión general. También puede ver www.uc-council.org y www.ean-int.org.

7.3. Comentario: Especificación Complementaria

La Especificación Complementaria captura otros requisitos, información y restricciones que no se recogen fácilmente en los casos de uso o el Glosario, que comprende los atributos o requisitos de calidad “URPS+” de todo el sistema. Fíjese que los requisitos específicos de un caso de uso pueden (y probablemente deben) escribirse en primer lugar con el caso de uso, en una sección *Requisitos Especiales*, pero algunos prefieren también reunirlos en la Especificación Complementaria. Los elementos de la Especificación Complementaria podrían comprender:

- Requisitos FURPS+ —funcionalidad, facilidad de uso, fiabilidad, rendimiento y soporte—.
- Informes.
- Restricciones de software y hardware (sistemas operativos y de red...).
- Restricciones de desarrollo (por ejemplo, herramientas de proceso y desarrollo).
- Otras restricciones de diseño e implementación.

- Cuestiones de internacionalización (unidades, idiomas...).
- Documentación (usuario, instalación, administración) y ayuda.
- Licencia y otras cuestiones legales.
- Empaquetado.
- Estándares (técnico, seguridad, calidad).
- Cuestiones del entorno físico (por ejemplo, calor o vibración).
- Cuestiones operacionales (por ejemplo, ¿cómo se gestionan los errores o con qué frecuencia se hacen las copias de seguridad?).
- Reglas del dominio o negocio.
- Información en los dominios de interés (por ejemplo, ¿cuál es el ciclo completo de gestión de pagos a crédito?).

Las **restricciones** no son comportamientos, sino otro tipo de restricciones del diseño o el proyecto. También son requisitos, pero se denominan comúnmente “restricciones” para remarcar su influencia *restrictiva*. Por ejemplo:

Debe utilizar Oracle (tenemos un contrato de licencia con ellos)

Debe funcionar bajo Linux (disminuirá el coste)

Sugerencia

Las decisiones y restricciones de diseño tempranas (“elaboración prematura”) son casi siempre una mala idea, así que merece la pena desconfiar y cuestionarlas, especialmente durante la fase de inicio, cuando se ha analizado muy poco con cuidado. Algunas restricciones se imponen por causas inevitables, como restricciones legales, o una interfaz de un sistema externo existente al que se debe invocar.

Atributos de calidad

Algunos requisitos se denominan **atributos de calidad** [BCK98] (o “-ilities”) de un sistema. Éstos incluyen facilidad de uso (*usability*), fiabilidad (*reliability*), etcétera. Nótese que se refieren a cualidades del sistema, no a que estos requisitos sean necesariamente de calidad alta (la palabra está sobrecargada en inglés). Por ejemplo, la calidad de soporte (*supportability*) podría elegirse deliberadamente que fuese baja si el producto no se pretende que sirva a largo plazo.

Hay de dos tipos:

1. Observables en la ejecución (funcionalidad, facilidad de uso, fiabilidad, rendimiento, etc.).
2. No observables en ejecución (soporte, pruebas, etc.).

La funcionalidad se especifica en los casos de uso, como otros atributos de calidad relacionados con casos de uso específicos (por ejemplo, las cualidades de rendimiento en el caso de uso *Procesar Venta*).

Otros atributos de calidad FURPS+ del sistema se describen en la Especificación Complementaria.

Aunque la funcionalidad es un atributo de calidad válido, en su uso común, el término “atributo de calidad” casi siempre se refiere a “cualidades del sistema que no sean la funcionalidad”. En este libro, el término se utiliza de este modo. Esto no es exactamente lo mismo que los requisitos no funcionales, que es un término más amplio que incluye *todo* excepto la funcionalidad (por ejemplo, empaquetado y licencia).

Cuando nos ponemos nuestro “gorro de arquitecto”, los atributos de calidad de todo el sistema (y por tanto, la Especificación Complementaria donde se recogen) son especialmente interesantes porque —como veremos en el Capítulo 32— el análisis y el diseño de la arquitectura tienen que ver en gran medida con la identificación y resolución de los atributos de calidad en el contexto de los requisitos funcionales. Por ejemplo, suponga que uno de los atributos de calidad es que el sistema NuevaEra debe ser bastante tolerante a fallos cuando fallen los servicios remotos. Desde el punto de vista de la arquitectura, esto tendrá una influencia global en las decisiones de diseño a gran escala.

Entre los atributos de calidad existen interdependencias, lo que implica compromisos. Un ejemplo sencillo en el PDV podría ser, “muy fiable (tolerante a fallos)” y “fácil de probar” son opuestos, puesto que hay muchas formas sutiles de que pueda fallar un sistema distribuido.

Reglas del dominio (negocio)

Las reglas del dominio [Ross97, GK00] dictan el modo en el que podría operar un dominio o negocio. No son requisitos de ninguna aplicación, aunque, a menudo, los requisitos de una aplicación se ven afectados por las reglas del dominio. Las políticas de la compañía, leyes físicas y leyes gubernamentales, son reglas de dominio típicas.

Se denominan comúnmente **reglas del negocio**, que son el tipo más común, pero ese término está limitado, ya que existen aplicaciones software que no son de gestión de un negocio, como simulación del clima o logística militar. Una simulación del clima incluye “reglas del dominio”, relacionadas con las leyes y relaciones físicas, que afectan a los requisitos de la aplicación.

Con frecuencia, resulta útil identificar y registrar aquellas reglas del dominio que afectan a los requisitos, normalmente materializados en los casos de uso, porque pueden clarificar el contenido de un caso de uso ambiguo o incompleto. Por ejemplo, en el PDV NuevaEra, si alguien pregunta si en el caso de uso *Procesar Venta* debería escribirse una alternativa a los pagos a crédito que no necesite la captura de la firma, existe una regla del negocio (REGLA1) que aclara que no se permitirá por ninguna compañía de autorización de crédito.

Advertencia

Las reglas no son requisitos de la aplicación. No registre las características del sistema como reglas. Las reglas describen las restricciones y comportamientos del modo de trabajar del dominio, no de la aplicación.

Información en los dominios de interés

A menudo, a los expertos del dominio que se está estudiando, les resulta útil escribir (o proporcionar URLs con) explicaciones de dominios relacionados con el nuevo sistema software (ventas y contabilidad, la geofísica de los flujos subterráneos de petróleo/agua/gas,...), para presentar el contexto y ayudar a la comprensión por parte del equipo de desarrollo. Podría contener referencias importantes a expertos o a literatura, fórmulas, leyes u otras referencias. Por ejemplo, los misterios de los esquemas de codificación UPC y EAN, y la interpretación del código de barras, los deben entender, en cierta medida, el equipo de NuevaEra.

7.4. Ejemplo NuevaEra: Visión (Parcial)

Visión

Historia de revisiones

Versión	Fecha	Descripción	Autor
Borrador de Inicio	10 Enero, 2031	Primer borrador. Para refinarse principalmente durante la elaboración.	Craig Larman

El análisis del ejemplo es ilustrativo, pero ficticio

Introducción

Prevemos una aplicación de punto de venta (PDV) tolerante a fallos de próxima generación, PDV NuevaEra, con flexibilidad para poder soportar variación en las reglas del negocio del cliente, múltiples mecanismos de terminal e interfaz de usuario, y la integración con múltiples sistemas de terceras partes.

Orientación

Oportunidad del negocio

Los productos PDV existentes no son adaptables al negocio del cliente, en términos de permitir variar las reglas de negocio y los diseños de la red (por ejemplo cliente delgado o no, arquitectura en 2, 3 ó 4 capas). Además, no permiten su extensión de manera adecuada cuando se incrementan los terminales y crece el negocio. Y ninguno permite trabajar en línea o desconectados, adaptándose dinámicamente dependiendo de los fallos. Ninguno se integra fácilmente con muchos sistemas de terceras partes. Ninguno admite nuevas tecnologías como los PDAs móviles. El mercado se siente insatisfecho debido a este estado inflexible de cosas, y demandan un PDV que rectifique esta situación.

Enunciado del problema

Los sistemas PDV tradicionales son inflexibles, intolerantes a fallos, y difíciles de integrar con sistemas de terceras partes. Esto da lugar a problemas en el oportuno procesamiento de las ventas, en el establecimiento de procesos mejorados que no concuerdan con el software, y con los datos de contabilidad e inventario precisos y oportunos para dar soporte a la planificación y medidas, entre otras cuestiones. Esto afecta a los cajeros, encargados del almacén, administradores del sistema y a la gestión empresarial.

Enunciado de la posición en el mercado del producto

- Resumen conciso de a quién está dirigido el producto, las características relevantes, y qué lo diferencia de la competencia.

Alternativas y competencia...

Entender quiénes son los participantes y sus problemas

Descripción del personal involucrado

Demografía de mercado...

Resumen del personal involucrado (No usuarios)...

Resumen de Usuarios...

Objetivos de alto nivel y problemas claves del personal involucrado

El taller de requisitos de un día con los expertos en la materia que se está estudiando y otras personas involucradas, y encuestas a varios distribuidores, nos llevaron a los siguientes objetivos y problemas claves:

<i>Objetivo de alto nivel</i>	<i>Prioridad</i>	<i>Problemas e inquietudes</i>	<i>Soluciones actuales</i>
Procesamiento de ventas rápido, robusto e integrado.	Alta	<p>Se reduce la velocidad cuando se incrementa la carga.</p> <p>Pérdida de la capacidad de procesamiento de las ventas si los componentes fallan.</p> <p>Carencia de información actualizada y precisa de la contabilidad y otros sistemas debido a que no está integrado con los sistemas de contabilidad, inventario y RRHH. Da lugar a dificultades en las medidas y la planificación.</p> <p>Imposibilidad de adaptar las reglas de negocio a requisitos del negocio únicos.</p> <p>Dificultad al añadir nuevos tipos de terminales o interfaces de usuario (por ejemplo, PDAs móviles).</p>	Los productos PDV existentes permiten el procesamiento básico de ventas, pero no abordan estos problemas.
...

Objetivos de nivel de usuario

Podría ser la Lista Actor-Objetivo creada durante el modelado de casos de uso o un resumen más conciso

Los usuarios (y los sistemas externos) necesitan un sistema para satisfacer sus objetivos:

- *Cajero*: procesar las ventas, gestionar las devoluciones, abrir y cerrar caja.
- *Administrador del sistema*: gestionar los usuarios, gestionar la seguridad, gestionar las tablas del sistema.
- *Director*: poner en marcha, suspender operación.
- *Sistema de actividad de ventas*: analizar los datos de las ventas.
- ...

Entorno de usuario...

Visión general del producto

Perspectiva del producto

El PDV NuevaEra residirá, normalmente, en tiendas; si se utilizan terminales móviles se encontrarán muy próximos a la red de la tienda, en el interior o en el exterior. Proporcionará servicios al usuario, y colaborará con otros sistemas, como se indica en la Figura Visión-1.

Resumido a partir del diagrama de casos de uso

Los diagramas aparecen con diversos detalles, pero todos muestran los actores externos importantes para el sistema

Similar a la lista Actor-Objetivo, esta tabla relaciona objetivos, beneficios y soluciones, pero a un nivel más alto, no únicamente relacionado con los casos de uso

Resume el valor y características diferenciadoras del producto

Como se presenta abajo, las características del sistema constituyen un formato conciso para resumir la funcionalidad

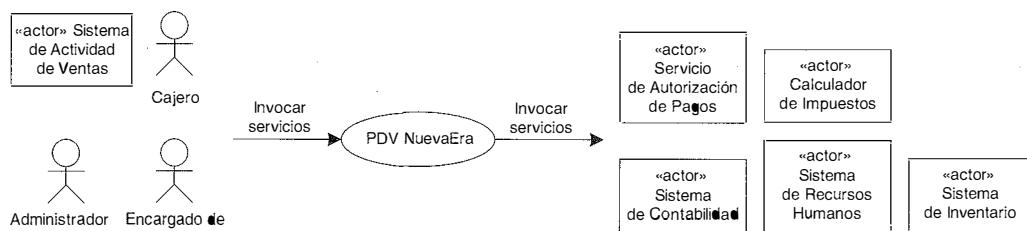


Figura Visión-1. Diagrama de contexto del sistema PDV NuevaEra.

Resumen de los beneficios

<i>Característica soportada</i>	<i>Beneficio del personal involucrado</i>
Funcionalmente, el sistema proporcionará todos los servicios típicos que requiere la organización de las ventas, incluyendo la entrada de las ventas, autorización de pagos, gestión de devoluciones, etcétera.	Servicios de punto de venta rápidos y automáticos.
Detección de fallos automática, cambiando a procesamiento local sin conexión cuando los servicios no estén disponibles.	Procesamiento de ventas continuado cuando fallan los componentes externos.
Reglas de negocio “conectables” en varios puntos del escenario durante el procesamiento de las ventas.	Configuración flexible de la lógica del negocio.
Transacciones en tiempo real con sistemas de terceras partes, haciendo uso de protocolos estándares industriales.	Información de ventas, contabilidad e inventario oportuna y precisa, para abordar las mediciones y la planificación.
...	...

Suposiciones y dependencias...

Coste y fijación del precio...

Licencia e instalación...

Resumen de las características del sistema

- Entrada de ventas.
- Autorización de pagos (crédito, débito, cheque).

- Administración del sistema de usuarios, seguridad, código y tablas de constantes, etcétera.
- Procesamiento automático de ventas sin conexión cuando fallen los componentes externos.
- Transacciones en tiempo real, basadas en estándares industriales, con sistemas de tercera partes, que incluye los servicios de inventario, contabilidad, recursos humanos, cálculo de impuestos, y autorización de pagos.
- Definición y ejecución de reglas del negocio “conectables” en puntos comunes y fijos de los escenarios de procesamiento.
- ...

Otros requisitos y restricciones

Abarca las restricciones de diseño, facilidad de uso, fiabilidad, rendimiento, soporte, empaquetado, documentación, etcétera: Diríjase a la Especificación Complementaria y a los casos de uso.

7.5. Comentario: Visión

¿Estamos solucionando el mismo problema? ¿El problema correcto?

El enunciado del problema

Durante el trabajo de requisitos en la fase de inicio, hay que colaborar para definir un enunciado del problema conciso; reducirá la posibilidad de que el personal involucrado esté intentando solucionar problemas ligeramente diferentes, y normalmente, se crea rápido. Algunas veces, este trabajo revela diferencias de opinión fundamentales en lo que las partes están tratando de conseguir.

Una alternativa al simple texto es el formato de tabla propuesto en la plantilla del RUP para el enunciado del problema:

El problema de	...
afecta	...
el impacto de lo cual es	...
una solución de éxito sería	...

Los objetivos de alto nivel y problemas claves del personal involucrado

Esta tabla resume los objetivos y problemas a un nivel más alto que los casos de uso de nivel de tarea, y muestra importantes objetivos no funcionales y de calidad que podrían pertenecer a un caso de uso o abarcar a muchos, tales como:

- *Necesitamos procesamiento de ventas tolerante a fallos.*
- *Necesitamos la capacidad de adaptar las reglas del negocio.*

¿Cuáles son los problemas y objetivos esenciales?

Es típico que el personal involucrado exprese sus objetivos en base a soluciones que imaginan, como “Necesitamos un programador a tiempo completo que adapte las reglas del negocio cuando las cambiemos”. Las soluciones algunas veces son inteligentes, porque comprenden bien el dominio del problema y las opciones. Pero, algunas veces, el personal involucrado llega a soluciones que no son las más apropiadas o que no tratan el problema subyacente más importante y esencial.

Por tanto, el analista del sistema, necesita estudiar la cadena de problemas y objetivos —como se presentó en el capítulo anterior sobre los objetivos y casos de uso— para aprender los problemas subyacentes, y su importancia e impacto relativos, y de este modo priorizar y solucionar las cuestiones más relevantes con una solución ingeniosa.

Métodos para facilitar la idea del grupo

Aunque queda fuera del alcance de esta discusión, especialmente durante actividades tales como definición de problemas de alto nivel e identificación de objetivos, tiene lugar el trabajo de investigación creativo en grupo. Presentamos algunas técnicas útiles de ayuda al grupo para el descubrimiento de los problemas y objetivos esenciales y para ayudar a la generación de ideas y su priorización: mapas mentales, diagramas causa-efecto (*fishbone*), diagramas pareto, “tormenta de ideas” (*brainstorming*), multivotación, votación por puntos (*dot voting*), técnica de grupo nominal, “escribir ideas en silencio” (*brainwriting*) y diagrama de afinidad. Estúdielos en la web. Prefiero aplicar varios de ellos durante el mismo taller, para descubrir problemas típicos y requisitos desde distintos ángulos.

Características del sistema—requisitos funcionales

Los casos de uso no son necesariamente el único modo para expresar los requisitos funcionales, por los siguientes motivos:

- Son detallados. A menudo, el personal involucrado quiere un resumen breve que identifique las funciones más destacables.
- ¿Y si listamos simplemente los nombres de los casos de uso (*Procesar Venta, Gestionar Devoluciones,...*) para resumir la funcionalidad? Primero, la lista podría ser todavía demasiado extensa. Además, los nombres pueden ocultar funcionalidad interesante que el personal involucrado realmente quiere conocer; esto es, el nivel de granularidad puede oscurecer funciones importantes. Por ejemplo, suponga que la descripción de la funcionalidad de la autorización automática de pago está contenida en el caso de uso *Procesar Venta*. Un lector de la lista de los nombres de los casos de uso no puede decir si el sistema realizará autorizaciones de pagos. Es más, uno podría desear agrupar un conjunto de casos de uso en una característica (por simplicidad), como *Administración del sistema para usuarios, seguridad, código y tablas de constantes, etcétera*.

- Algunas funcionalidades importantes, como es lógico, se expresan como sentencias cortas que no se corresponden de manera conveniente con los nombres de los casos de uso u objetivos de nivel de Procesos del Negocio Elementales (EBP). Podrían abarcar o ser ortogonales a los casos de uso. Por ejemplo, durante el primer taller de requisitos de NuevaEra, algunos podrían decir “el sistema debería ser capaz de llevar a cabo transacciones con sistemas de terceras partes de contabilidad, inventario y cálculo de impuestos”. Esta sentencia sobre la funcionalidad no representa ningún caso de uso en particular, pero es una manera cómoda y concisa para expresar, recoger y comunicar las características.
 - Una variación más drástica del último punto es la siguiente. Algunas aplicaciones requieren ante todo una descripción de la funcionalidad como características; los casos de uso no se ajustan de manera natural. Esto es común, por ejemplo, en productos *middleware*, como los servidores de aplicaciones —los casos de uso no están motivados realmente—. Suponga que el equipo está considerando la próxima versión. Durante un debate de los requisitos, la gente (como los de marketing) dirá: “La siguiente versión necesita soportar *bean* entidad EJB2.0”. Los requisitos se conciben inicialmente en términos de una lista de características, no de casos de uso.

Por tanto, un modo alternativo, complementario para expresar las funciones del sistema, es mediante las **características**, o más concretamente en este contexto, **características del sistema**, que son sentencias concisas, de alto nivel, que resumen las funciones del sistema. Más formalmente, en el UP, un **característica del sistema** es “un servicio observable externamente proporcionado por el sistema que cumple directamente una necesidad del personal involucrado” [Kruchten00].

Las características son cosas que un sistema puede *hacer*. Deberían pasar este test lingüístico:

El sistema deberá hacer <característicaX>.

Por ejemplo:

El sistema deberá hacer la autorización del pago.

Recordemos que la Visión podría utilizarse como contrato formal o informal entre los desarrolladores y la empresa. Las características del sistema constituyen un mecanismo para resumir, en este contrato, lo que el sistema *hará*. Esto es complementario a los casos de uso, ya que las características son concisas.

Las características tienen que contrastarse con varios tipos de requisitos no funcionales y restricciones, por ejemplo: “*El sistema debe ejecutarse bajo Linux, debe tener disponibilidad de 24/7 y debe tener una interfaz con pantalla táctil*”. Nótese que falla el test lingüístico.

A veces, la propiedad de que sea “un servicio observable externamente...” es difícil de resolver. Por ejemplo, ¿debería ser lo siguiente una característica del sistema?:

El sistema hará las transacciones con sistemas de terceras partes de contabilidad, inventario, recursos humanos y cálculo de impuestos.

Se trata de un tipo de comportamiento, y probablemente de interés para el personal involucrado, pero la colaboración en sí misma podría no ser visible externamente, dependiendo del margen de tiempo, lo cerca que se esté y de dónde se mire. Téngalo en cuenta —rara vez merece la pena que se preocupe por cuestiones de clasificación de poca granularidad—.

Por último, podemos observar que la mayoría de las características del sistema se expresarán de manera detallada en el texto de los casos de uso.

Notación y organización

Ante todo, son importantes las descripciones breves de alto nivel. Uno debería ser capaz de leer la lista de características del sistema rápidamente.

No es necesario incluir la frase canónica “El sistema deberá hacer...” o una frase variante, aunque es lo normal.

A continuación, presentamos un ejemplo de características de alto nivel, para un proyecto multi-sistema amplio, del cual el PDV es sólo un elemento:

Las características principales incluyen:

- *Servicios PDV.*
- *Gestión de inventario.*
- *Compras basadas en la web.*
- ...

Es habitual organizar una jerarquía de dos niveles de características del sistema. Pero en el documento de Visión, más de dos niveles nos llevaría a un detalle excesivo; el sentido de las características del sistema en la Visión es resumir la funcionalidad, no descomponerla en una larga lista de elementos de grano fino. Un ejemplo razonable en cuanto a detalle:

Las características principales incluyen:

- *Servicios PDV:*
 - *Capturar ventas.*
 - *Autorización de pago.*
 - ...
- *Gestión de inventario:*
 - *Reordenación automática.*
 - ...

Algunas veces, estas características de segundo nivel son básicamente equivalentes a los nombres de los casos de uso (u objetivos de nivel de usuario), pero no es imprescindible; las características son un modo alternativo para resumir la funcionalidad. No obstante, la mayoría de las características del sistema se expresarán de manera detallada en el texto de los casos de uso.

¿Cuántas características del sistema debería contener la Visión?

Sugerencia

Es deseable un documento de Visión con menos de 50 características. Si hay más, considere la posibilidad de agrupar y abstraer las características.

Otros requisitos en la Visión

En la Visión, las características del sistema resumen brevemente los requisitos funcionales, que se expresan en detalle en los casos de uso. De igual modo, la Visión *puede* resumir otros requisitos (por ejemplo, fiabilidad y facilidad de uso) que se detallan en la sección de *Requisitos Especiales* de los casos de uso, y en la Especificación Complementaria (SS, *Supplementary Specification*). Sin embargo, hay peligro de duplicaciones inútiles. Por ejemplo, el producto RUP proporciona plantillas para la Visión y SS que contienen secciones idénticas o similares para otros requisitos como facilidad de uso, fiabilidad, rendimiento, etcétera. Tal duplicación es inevitablemente difícil de mantener. Además, el nivel de detalle de las secciones similares (por ejemplo, rendimiento) en la Visión y SS, necesita que sea bastante parecido para que tenga sentido; es decir, las descripciones de otros requisitos “detalladas” y “esenciales”, tienden a ser prácticamente iguales.

Sugerencia

Para otros requisitos, evite su duplicación incluyéndolo de forma idéntica o casi idéntica, tanto en la Visión como en la Especificación Complementaria (SS) —y en los casos de uso—. Más bien, recójalos sólo en la SS o los casos de uso (si son específicos de los casos de uso). En la Visión dirija al lector a las secciones de éstos donde puede encontrar estos otros requisitos.

Visión, características o casos de uso—¿qué va primero?

No es útil ser estricto en el orden de algunos artefactos. Durante la colaboración para crear diferentes artefactos de requisitos, se produce una sinergia en la que el trabajo con uno influye y ayuda a clarificar los otros. Sin embargo, la secuencia recomendada es:

1. Escribir un primer borrador breve de la Visión.
2. Identificar los objetivos de usuario y los casos de uso de apoyo.
3. Escribir algunos casos de uso y comenzar la Especificación Complementaria.
4. Refinar la Visión, resumiendo la información a partir de éstos.

7.6. Ejemplo NuevaEra: un Glosario (Parcial)

Glosario

Historia de revisiones

Versión	Fecha	Descripción	Autor
Borrador de Inicio	10 Enero, 2031	Primer borrador. Para refinarse principalmente durante la elaboración.	Craig Larman

Definiciones

Término	Definición e Información	Alias
artículo	Un artículo o servicio en venta.	
autorización de pago	Validación llevada a cabo por un servicio externo de autorización de pago, que hará o garantizará el pago al vendedor.	
solicitud de autorización de pago	Un compuesto de elementos enviados electrónicamente a un servicio de autorización, normalmente como un array de caracteres. Los elementos comprenden: ID de la tienda, número de cuenta del cliente, cantidad y fecha.	
UPC	Código de 12 dígitos que identifica un artículo. Normalmente se representa mediante un código de barras en los artículos. Diríjase a http://www.uc-council.org para ver más detalles.	Código de Producto Universal
...

7.7. Comentario: Glosario (Diccionario de Datos)

En su forma más simple, el **Glosario** es una lista de los términos relevantes y sus definiciones. Es sorprendentemente habitual que un término, frecuentemente técnico o propio del dominio, se utilice de forma ligeramente distinta por diferentes personas involucradas; esto tiene que resolverse para reducir los problemas de comunicación y los requisitos ambiguos.

Sugerencia

Comience el Glosario cuanto antes. Recuerdo una experiencia trabajando con expertos en simulación, en la que descubrimos que la aparentemente inocua, pero importante, palabra “celda” tenía significados diversos y escurridizos entre los miembros del grupo.

El objetivo no es recoger todos los posibles términos, sino aquellos que no están claros, son ambiguos o que requieren algún tipo de elaboración relevante, como el formato de la información o las reglas de validación.

El Glosario como diccionario de datos

En el UP, el Glosario también juega el rol de **diccionario de datos**, un documento que recoge los datos sobre los datos, es decir, **metadatos**. Durante la fase de inicio, el glosario debe ser un documento sencillo de términos y descripciones. Durante la elaboración, podría ampliarse a un diccionario de datos.

Los atributos de los términos podrían contener:

- Alias.
- Descripción.
- Formato (tipo, longitud, unidad).
- Relaciones con otros elementos.

- Rango de valores.
- Reglas de validación.

Observe que el rango de valores y las reglas de validación del Glosario constituyen requisitos con implicaciones en el comportamiento del sistema.

Unidades

Como subraya Martin Fowler en *Analysis Patterns* [Fowler96], se deben tener en cuenta las unidades (moneda, medidas,...), especialmente en esta era de aplicaciones software internacionalizadas. Por ejemplo, en el sistema NuevaEra, que, con un poco de suerte, se venderá a muchos clientes en diferentes países, el *precio* no puede ser sólo un simple número. Debe estar en una unidad de *Dinero* o *Moneda* que permita la variación de monedas.

Términos compuestos

El Glosario no está destinado sólo a términos atómicos como el “precio del artículo”. Puede y debe incluir términos compuestos, como “venta” (que incluye otros elementos, tales como la fecha y la ubicación), y alias utilizados para describir una colección de datos que se transmiten entre los actores en los casos de uso. Por ejemplo, en el caso de uso *Procesar Venta*, considere la siguiente sentencia:

El Sistema envía una *solicitud de autorización de pago* a un Servicio externo de Autorización de Pago, y solicita la aprobación del pago.

La “solicitud de autorización del pago” es un alias para un agregado de datos, que tiene que explicarse en el Glosario.

7.8. Especificaciones fiables: ¿Un Oxímoron²?

Los requisitos escritos pueden dar la impresión de que se han entendido los requisitos reales y están bien definidos, y pueden utilizarse (pronto) para estimar y planificar el proyecto de manera fiable. Esta impresión es más intensa en los desarrolladores no programadores; los programadores conocen a partir de experiencias dolorosas lo poco fiable que es. Esto justifica en parte de la motivación de la cita de Goethe del comienzo del capítulo.

Lo que realmente importa es construir software que pase las pruebas de aceptación definidas por los usuarios y el resto de personas implicadas en el proyecto, y que satisfaga sus verdaderos objetivos (que, a menudo, no se descubren hasta que están evaluando o trabajando con el software).

² *N. del T.*: Figura en la que el adjetivo que califica un nombre contradice la esencia del nombre que modifica, como en “nieve ardiente”.

Es importante escribir como ejercicio la Visión y Especificación Complementaria para poner en claro una primera aproximación de lo que se quiere, la motivación del producto y como repositorio para las grandes ideas. Pero no son —ni ningún artefacto de requisitos— una especificación fiable. Sólo escribiendo código, probándolo, obteniendo retroalimentación, manteniendo una estrecha colaboración con los usuarios y los clientes y adaptando, se da realmente en el blanco.

Esto no es una llamada a abandonar el análisis y la reflexión e ir directamente a escribir código, sino una sugerencia para tratar los requisitos escritos de una manera ligera, y continuamente involucrar a los usuarios —de hecho, debía ser diariamente—.

7.9. Artefactos disponibles en el sitio web del proyecto

Puesto que esto es un libro, estos ejemplos y los anteriores casos de uso tienen un aspecto estático y, quizás orientado al papel. Sin embargo, deberían ser artefactos digitales recogidos sólo on-line a través del sitio web del proyecto. Y, en lugar de ser documentos estáticos planos, podrían contener hipervínculos, o guardarse en herramientas distintas a un procesador de texto u hoja de cálculo. Por ejemplo, el Glosario podría almacenarse en una tabla de una base de datos.

7.10. ¿Poco UML durante la fase de inicio?

El objetivo de la fase de inicio es recopilar sólo la información suficiente para establecer una visión común, decidir si es viable avanzar y si merece la pena una investigación seria del proyecto en la fase de elaboración. Como tal, a menudo, no son necesarios más diagramas que los simples diagramas de casos de uso UML. Durante la fase de inicio se hace hincapié en entender el alcance básico y el 10% de los requisitos, expresados textualmente. En la práctica y, por tanto, en esta presentación, la mayoría de los diagramas UML se realizarán en la siguiente fase —elaboración—.

7.11. Otros artefactos de requisitos en el UP

Como en el anterior capítulo de casos de uso, la Tabla 7.1 resume una muestra de los artefactos y su evolución temporal. Todos los artefactos de requisitos se comienzan en la fase de inicio, se trabaja sobre ellos sobre todo durante la elaboración.

Inicio

No debería ser el caso que estos artefactos de requisitos se terminen en la fase de inicio. De hecho, apenas se habrán empezado.

El personal involucrado necesita decidir si merece la pena llevar a cabo una investigación seria en el proyecto; esta investigación tiene lugar durante la elaboración, no la fase de inicio. Durante la fase de inicio, la Visión resume la idea del proyecto de manera que ayude, a los que tiene que tomar la decisión, a determinar si merece la pena continuar, y dónde comenzar.

Tabla 7.1. Muestra de los artefactos UP y evolución temporal. c – comenzar; r – refinar.

<i>Disciplina</i>	<i>Artefacto Iteración →</i>	<i>Inicio I1</i>	<i>Elab. E1...En</i>	<i>Const. C1...Cn</i>	<i>Trans. T1...T2</i>
Modelado del Negocio	Modelo del Dominio		c		
Requisitos	Modelo de Casos de Uso <i>Visión</i> <i>Especificación Complementaria</i> <i>Glosario</i>	c c c c	r r r r		
Diseño	Modelo de Diseño Documento de Arquitectura SW Modelo de Datos		c c c	r r	
Implementación	Modelo de Implementación		c	r	r
Gestión del Proyecto	Plan de Desarrollo SW	c	r	r	r
Pruebas	Modelo de Pruebas		c	r	
Entorno	Marco de Desarrollo	c	r		

Puesto que la mayoría del trabajo de requisitos ocurre durante la elaboración, la Especificación Complementaria debería desarrollarse ligeramente durante la fase de inicio, resaltando los atributos de calidad relevantes (por ejemplo, el PDV NuevaEra debe recuperarse cuando fallen los servicios externos) que muestren mayores riesgos y retos.

La información de entrada a estos artefactos podría generarse durante un taller de requisitos de la fase de inicio, tanto mediante la consideración explícita de estas cuestiones, como indirectamente, por medio del análisis de casos de uso. En el taller no se escribirá un borrador legible de los artefactos, sino que lo elaborará después el analista del sistema.

Elaboración

A través de las iteraciones de la elaboración, se refina la “visión” y la Visión, en base a la retroalimentación a partir de la construcción incremental de partes del sistema, la adaptación, y varios talleres de requisitos a lo largo de varias iteraciones de desarrollo.

Mediante la investigación de requisitos continuada y el desarrollo iterativo, los otros requisitos llegarán a verse más claros y se recogerán en la SS. Los atributos de calidad (por ejemplo, fiabilidad) identificados en la SS, serán claves para dar forma al núcleo de la arquitectura que se diseña y programa durante la elaboración. También pueden ser factores de riesgo claves que influyen en lo que se va a trabajar en las primeras iteraciones. Por ejemplo, el requisito de calidad del PDV NuevaEra de recuperación en el lado del cliente si fallan los componentes externos, se explorará durante la elaboración.

La mayoría de los términos se descubren y elaboran en el Glosario durante esta fase.

Llegando al final de la elaboración, es viable contar con casos de uso, una Especificación Complementaria y un documento de Visión que, de manera razonable, reflejan las características estabilizadas más importantes y otros requisitos que se deben completar para entregar el producto. Sin embargo, la Especificación Complementaria y la

Visión no se congelan o “se dan por concluidas” como especificaciones fijas; la adaptación —no la rigidez— es un valor central del desarrollo iterativo y el UP.

Aclaremos el comentario “dar por concluido y congelar”: es perfectamente razonable —al final de la elaboración— llegar a un acuerdo con el personal involucrado sobre lo que se hará en lo que queda del proyecto, y comprometerse (quizás contractualmente) considerando los requisitos y la planificación. En algunos puntos (el final de la elaboración, en el UP), necesitamos una idea fiable sobre “qué, cuánto y cómo”. En este sentido, es normal, y se espera, un acuerdo formal sobre los requisitos. También es necesario contar con un proceso de control de cambios (una de las mejores prácticas explícitas del UP) de forma que se tienen en cuenta y se aprueban, formalmente, los cambios en los requisitos, en lugar de cambios caóticos e incontrolados.

Más bien, el comentario “dar por concluido y congelar” implica varias ideas:

- En el desarrollo iterativo y el UP, se entiende que no importa cuanta atención medida se presta a la especificación de requisitos, algún cambio es inevitable, y se debería aceptar. Este cambio podría ser una mejora oportunista que incorpora una novedad importante en el sistema que confiere a sus propietarios ventaja competitiva, o un cambio debido a una visión mejorada.
- En el desarrollo iterativo, es un valor central contar con un compromiso continuo por parte del personal involucrado para evaluar, proporcionar retroalimentación, y dirigir el proyecto como realmente quieran. No beneficia al personal involucrado el “lavarse las manos” de un compromiso solícito, dando por concluido un conjunto de requisitos congelados y esperando el producto final, porque raramente obtendrán lo que realmente necesitan.

Construcción

En la construcción, los requisitos importantes —tanto funcionales como de otro tipo— deberían estar estabilizados —no terminados—, aunque dispuestos a cambios menores. Por tanto, no es probable que la SS y la Visión experimenten muchos cambios en esta fase.

7.12. Lecturas adicionales

Los documentos como el de Visión y Especificaciones Complementarias no son nuevos. Se utilizan en muchos proyectos y se describen en muchos libros de requisitos. Muchos de estos libros asumen de manera implícita la actitud del proceso en cascada, en el que el objetivo es obtener los requisitos detallados y correctos al principio, y llegar a un compromiso, antes de pasar al diseño y la implementación. En este sentido, la descripción tradicional que presentan no es útil aunque, por otra parte, proporcionan buenos consejos sobre posibles secciones y sus contenidos.

La mayoría de los libros sobre arquitectura software incluyen una discusión sobre el análisis de requisitos para los requisitos de calidad de la aplicación, puesto que estos requisitos de calidad tienden a influenciar fuertemente el diseño de la arquitectura. Un ejemplo es *Software Architecture in Practice* [BCK98].

Las reglas de negocio se tratan de manera exhaustiva en *The Business Rule Book* [Ross97]. El libro presenta una teoría amplia, profunda y muy bien desarrollada de las reglas del negocio, pero el método no se conecta de manera adecuada con otras técnicas de requisitos modernas, como los casos de uso o el desarrollo iterativo.

7.13. Artefactos UP y contexto del proceso

La Figura 7.1 muestra la influencia entre los artefactos, destacando los documentos de Visión, Especificación Complementaria y el Glosario.

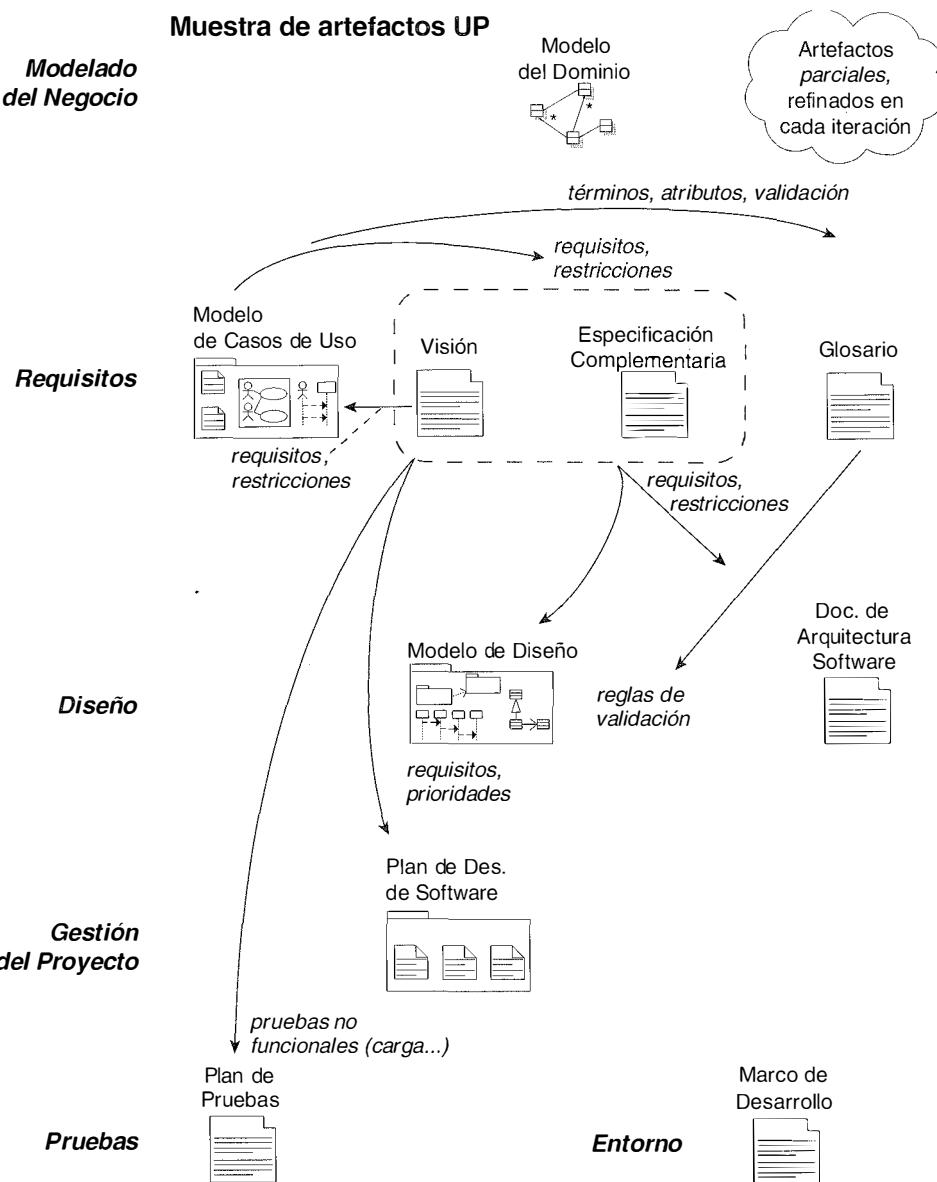


Figura 7.1. Muestra de la influencia entre los artefactos UP.

En el UP, el trabajo de Visión y Especificación Complementaria es una actividad de la disciplina de requisitos que podría iniciarse durante un taller de requisitos, junto con el análisis de casos de uso. La Figura 7.2 proporciona algunos consejos sobre el momento y el lugar para llevar a cabo este trabajo.

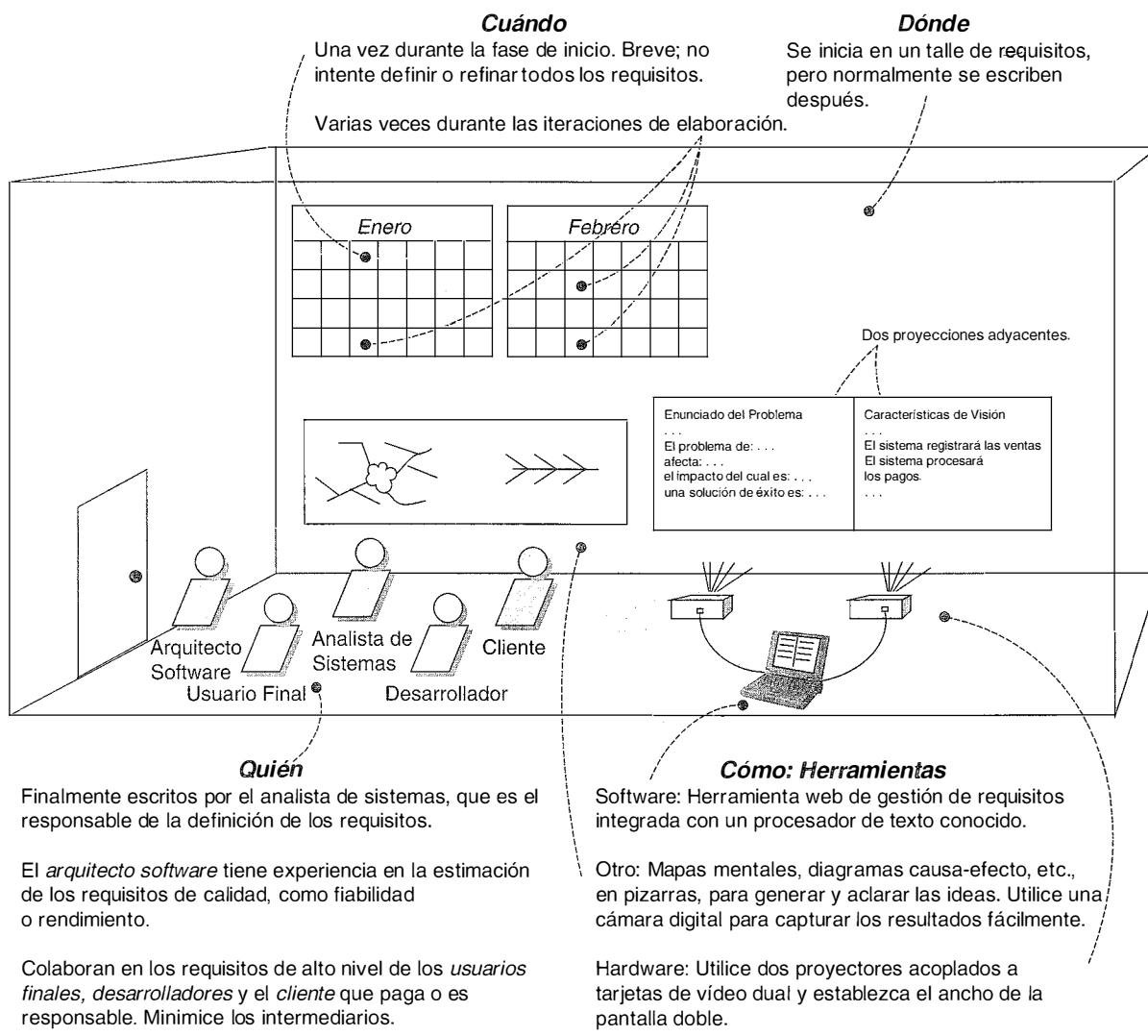


Figura 7.2. Proceso y establecimiento del contexto.

Capítulo 8

DEL INICIO A LA ELABORACIÓN

Lo duro y rígido se rompe. Lo flexible prevalece.

Tao Te Ching

Objetivos

- Definir la etapa de elaboración.
 - Motivar los siguientes capítulos de esta sección.
-

Introducción

La elaboración es la serie inicial de iteraciones durante la que:

- se descubren y estabilizan la mayoría de los requisitos
- se reducen o eliminan los riesgos importantes
- se implementan y prueban los elementos básicos de la arquitectura

Rara vez, la arquitectura no es un riesgo —por ejemplo, si se construye un sitio web como el que el equipo ya ha elaborado con éxito, con las mismas herramientas y los mismos requisitos— en cuyo caso, no tiene que considerarse en estas primeras iteraciones. En este caso, podrían implementarse características o casos de uso críticos, pero no significativos desde el punto de vista de la arquitectura.

Es en esta fase donde el libro se centra en una introducción al A/DDO, aplicando UML, patrones y arquitectura.

8.1. Punto de control: ¿qué sucedió en el inicio?

La etapa de inicio del proyecto de PDV NuevaEra puede durar sólo una semana. Los artefactos creados deberían ser breves e incompletos, la etapa rápida, y la investigación ligera.

No es la fase de requisitos del proyecto, sino una etapa breve para determinar la viabilidad, riesgo y alcance básicos, y decidir si merece la pena más investigación seria en el proyecto, que tendrá lugar en la elaboración. No se han cubierto todas las actividades que podrían ocurrir, de manera razonable, en la fase de inicio; este estudio resalta los artefactos orientados a los requisitos. Algunas de las actividades y artefactos posibles en la fase de inicio comprenden:

- Un breve taller de requisitos.
- La mayoría de los actores, objetivos y casos de usos, con los nombres.
- La mayoría de los casos de uso escritos en formato breve; del 10 al 20% de los casos de uso se escriben en detalle en formato completo para mejorar la comprensión del alcance y la complejidad.
- Identificación de la mayoría de los requisitos de calidad influyentes y de riesgo.
- Escritura de la primera versión de la Visión y Especificación Complementaria.
- Lista de riesgos.
 - Por ejemplo, el director en realidad quiere una demostración en la feria comercial POSWorld en Hamburgo, dentro de 18 meses. Pero el esfuerzo para el desarrollo de la demo no puede ser ni siquiera estimado a grandes rasgos hasta que no se haga un estudio más profundo.
- Prototipos de pruebas de conceptos técnicos y otros estudios para explorar la viabilidad técnica de los requisitos especiales (“¿Funcionan los Java Swing de manera adecuada en pantallas táctiles?”).
- Prototipos orientados a la interfaz de usuario para clarificar la visión de los requisitos funcionales.
- Recomendaciones sobre qué componentes comprar/construir/reutilizar, que se refinará en la elaboración.
 - Por ejemplo, una recomendación de compra de un paquete de cálculo de impuestos.
- Arquitectura de alto nivel *candidata* y componentes propuestos.
 - No se trata de una descripción detallada de la arquitectura, y no se pretende que sea final o correcta. Más bien, se refiere a una breve especulación que se va a utilizar como punto de partida del estudio en la elaboración. Por ejemplo, “una aplicación Java del lado del cliente, sin un servidor de aplicaciones, Oracle para la base de datos,...”. En la elaboración, podría probarse que merece la pena, o descubrir que es una mala idea y rechazarla.
- Plan para la primera iteración.
- Lista de herramientas candidatas.

8.2. En la elaboración

La elaboración es la serie inicial de iteraciones durante las que el equipo lleva a cabo un estudio serio, implementa (programas y pruebas) el núcleo central de la arquitectura, aclara la mayoría de los requisitos, y aborda las cuestiones de alto riesgo. En el UP, el “riesgo” incluye valor del negocio. Por tanto, el trabajo inicial podría incluir la implementación de los escenarios que se consideran importantes, pero que no son especialmente arriesgados desde el punto de vista técnico.

La elaboración, a menudo, consta de entre dos y cuatro iteraciones; se recomienda que cada iteración dure entre dos y seis semanas, a menos que el tamaño del equipo sea muy grande. Se fija la duración de cada iteración, entendiendo que se fija la fecha de finalización; si es probable que el equipo no cumpla con la fecha, algunos requisitos se colocan en la lista de tareas futuras, de manera que la iteración pueda concluir a tiempo, con una versión estable y que se pueda probar.

La elaboración no se corresponde con una fase de diseño o una fase en la que se desarrollan completamente los modelos preparándolos para que se implementen en la etapa de construcción —que sería un ejemplo de superposición de las ideas del desarrollo en cascada sobre el desarrollo iterativo y el UP—.

Durante esta fase, uno no está creando prototipos desecharables; sino que el código y el diseño son porciones del sistema final con calidad de producción. En algunas descripciones del UP, el término, que puede ser origen de malentendidos, “**prototipo de la arquitectura**” se utiliza para describir el sistema parcial. La intención no es que sea un prototipo en el sentido de un experimento desecharable; en el UP, significa un subconjunto de producción del sistema final. Más comúnmente se denomina **arquitectura ejecutable** o **base de la arquitectura**.

La elaboración en una frase

Construir el núcleo central de la arquitectura, resolver los elementos de alto riesgo, definir la mayoría de los requisitos, y estimar la planificación y los recursos globales.

Algunas de las ideas claves y buenas prácticas, que se pondrán de manifiesto en la elaboración, incluyen:

- Llevar a cabo iteraciones breves, de duración fija, dirigidas por el riesgo.
- Comenzar a programar pronto.
- Diseñar, implementar y probar, de manera adaptable, las partes básicas y arriesgadas de la arquitectura.
- Probar desde el principio, a menudo y de manera realista.
- Adaptar en base a la retroalimentación procedente de las pruebas, usuarios y desarrolladores.
- Escribir la mayoría de los casos de uso y otros requisitos en detalle, a través de una serie de talleres, uno por cada iteración de la elaboración.

¿Qué es significativo desde el punto de vista de la arquitectura en la elaboración?

Las iteraciones iniciales construyen y prueban el núcleo central de la arquitectura. Para el proyecto de PDV NuevaEra —y de hecho, para la mayoría— esto incluirá:

- Empleo de diseño e implementación “ancho y superficial”; o, como lo ha llamado Grady Booch, “diseño en las fronteras”.
 - Es decir, identificación de los diferentes procesos, capas, paquetes, y subsistemas, y sus responsabilidades e interfaces de alto nivel. Implementarlos parcialmente con el objeto de conectarlos y clarificar las interfaces. Los módulos contendrían, en su mayor parte, código “*stubbed*” que permite probar las conexiones e interfaces.
- Refinamiento de las interfaces locales y remotas inter-módulos (incluyendo los detalles más sutiles acerca de los parámetros y valores de retorno).
 - Por ejemplo, la interfaz del objeto que actuará de intermediario (*wrapper*) en el acceso al sistema de contabilidad de terceras partes.
 - La primera versión de una interfaz rara vez es perfecta. La preocupación inicial por insistir en la prueba, “provocar fallos” y refinamiento de las interfaces, ayuda más tarde al trabajo en paralelo de varios equipos, confiando en interfaces estables.
- Integración de los componentes existentes.
 - Por ejemplo, un calculador de impuestos.
- Implementación de escenarios simplificados de caminos de éxito y fracaso que impulsan el diseño, la implementación y las pruebas, a través de muchos componentes importantes.
 - Por ejemplo, el escenario principal de éxito de *Procesar Venta*, utilizando el escenario de extensión para el pago a crédito.

Las pruebas en la fase de elaboración son importantes, para obtener retroalimentación, adaptar y probar que el núcleo es robusto. Las pruebas iniciales del proyecto NuevaEra incluirán:

- Pruebas de usabilidad de la interfaz de usuario para *Procesar Venta*.
- Pruebas de recuperación cuando falla el servicio remoto, como la autorización de crédito.
- Pruebas de carga alta a los servicios remotos, como la carga al sistema de cálculo de impuestos remoto.

8.3. Planificación de la siguiente iteración

La planificación y gestión del proyecto son importantes pero son temas amplios. En esta sección presentamos brevemente algunas ideas clave, y se proporcionará una introducción en el Capítulo 36.

Organice los requisitos y las iteraciones según el riesgo, grado de cobertura y naturaleza crítica.

- **Riesgo:** comprende tanto la complejidad técnica como otros factores, como incertidumbre del esfuerzo o facilidad de uso.
- **Cobertura:** implica que todas las partes importantes del sistema se tratan, al menos, en las primeras iteraciones —quizás una implementación “amplia y superficial” a través de muchos componentes—.
- **Naturaleza crítica:** se refiere a las funciones de alto valor para el negocio.

Estos criterios se utilizan para priorizar el trabajo a través de las iteraciones. Los casos de uso, o los escenarios de los casos de uso, se clasifican para la implementación —las primeras iteraciones implementan los escenarios clasificados de rango alto—. Además, algunos requisitos se enuncian como características de alto nivel no relacionadas con ningún caso de uso en particular, como el servicio de registro. Éstas también se clasifican.

La clasificación se realiza antes de la Iteración 1, pero después otra vez antes de la Iteración 2, etcétera, cuando nuevos requisitos y nuevas interpretaciones influyan en el orden. Es decir, el plan es adaptable, en lugar de fijado de un modo especulativo al comienzo del proyecto.

Normalmente, surgirá una agrupación difusa de requisitos basada en algunas técnicas de clasificación de colaboración de grupos pequeños. Por ejemplo:

<i>Rango</i>	<i>Requisito (Caso de Uso o Característica)</i>	<i>Comentario</i>
Alto	Procesar Venta Registro (Logging) ...	Puntuación alta en todos los criterios de clasificación. Extendido. Difícil de añadir más tarde. ...
Medio	Mantener Usuarios ...	Afecta al subdominio de seguridad ...
Bajo

Basándonos en esta clasificación, vemos que algunos escenarios claves del caso de uso *Procesar Venta*, significativos desde el punto de vista de la arquitectura, deberían abordarse en las primeras iteraciones. Esta lista no es exhaustiva; también se abordarán otros requisitos. Además, en cada iteración se trabajará en un caso de uso, implícito o explícito, *Poner en Marcha*, para satisfacer sus necesidades de inicialización.

En términos de los artefactos del UP, unos pocos comentarios sobre esta información de planificación:

- Los requisitos elegidos para la siguiente iteración se listan brevemente en un **Plan de Iteración**. No se trata de un plan de todas las iteraciones, sólo un plan de la siguiente.
- Si la breve descripción del Plan de Iteración es insuficiente, se podría escribir con más detalle una tarea o requisitos en una **Solicitud de Cambio**, y dársela al equipo responsable.
- La clasificación de requisitos se recoge en el **Plan de Desarrollo de Software**.

8.4. Requisitos y énfasis de la iteración 1: habilidades de A/DOO fundamentales

En este caso de estudio, la Iteración 1 de la fase de elaboración hace hincapié en una gama de habilidades de A/DOO fundamentales y comunes, utilizadas en la construcción de sistemas de objetos, como la asignación de las responsabilidades a los objetos. Por supuesto, se necesitan muchas otras habilidades y etapas —como el diseño de bases de datos, ingeniería de usabilidad y diseño de interfaces de usuario— para construir software, pero quedan fuera del alcance de esta introducción al A/DOO y al UP.

Requisitos de la iteración 1

Los requisitos para la primera iteración de la aplicación PDV NuevaEra son los siguientes:

- Implementación de un escenario clave básico del caso de uso *Procesar Venta*: entrada de artículos y recepción del pago en efectivo.
- Implementación de un caso de uso *Poner en Marcha*, necesario para dar soporte a las necesidades de inicialización de la iteración.
- No se maneja nada especial o complejo, sólo un simple escenario que acaba con éxito, y el diseño e implementación que lo soporte.
- No hay colaboración con servicios externos, como el sistema de cálculo de impuestos o base de datos de artículos.
- No se aplican reglas complejas para fijar los precios.

Aunque no se cubre, también se abordaría el diseño e implementación de las interfaces de usuario de soporte.

Las siguientes iteraciones extenderán esta base.

Desarrollo incremental para el mismo caso de uso a través de las iteraciones

Nótese que no todos los requisitos del caso de uso *Procesar Venta* se han tratado en la iteración 1. Es normal continuar con el trabajo de varios escenarios o características del mismo caso de uso en varias iteraciones y gradualmente ampliar el sistema para, al final, tratar toda la funcionalidad requerida (ver Figura 8.1). Por otro lado, los casos de uso simples y cortos, podrían completarse en una iteración.

8.5. ¿Qué artefactos podrían crearse en la elaboración?

La Tabla 8.1 lista artefactos de *muestra* que podrían crearse en la elaboración, e indica qué cuestión abordan. Los capítulos siguientes presentarán algunos de ellos con mayor

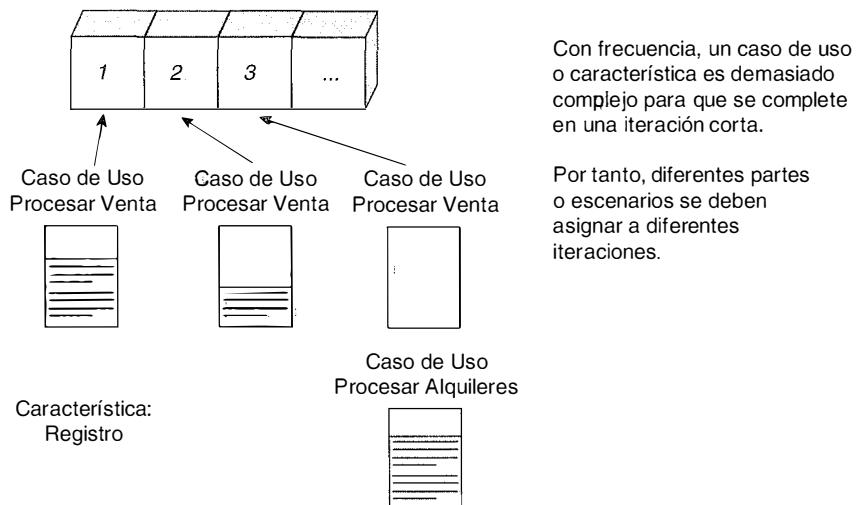


Figura 8.1. La implementación del caso de uso puede extenderse a lo largo de varias iteraciones.

Tabla 8.1. Muestra de Artefactos de elaboración, excluyendo aquellos que se crearon en la fase de inicio.

Artefacto	Comentario
Modelo del Dominio	Es una visualización de los conceptos del dominio; es similar al modelo de información estático de las entidades del dominio.
Modelo de Diseño	Es el conjunto de diagramas que describen el diseño lógico. Comprende los diagramas de clases software, diagramas de interacción, diagramas de paquetes, etcétera.
Documento de la Arquitectura Software	Una ayuda de aprendizaje que resume las cuestiones claves de la arquitectura y cómo se resuelven en el diseño. Es un resumen de las ideas destacadas del diseño y su motivación en el sistema.
Modelo de Datos	Incluye los esquemas de bases de datos, y las estrategias de transformación entre representaciones de objetos y no objetuales.
Modelo de Pruebas	Una descripción de lo que se probará y cómo.
Modelo de Implementación	Se corresponde con la implementación real —el código fuente, ejecutables, base de datos, etcétera—.
Guiones de Caso de Uso, Prototipos UI	Descripción de la interfaz de usuario, caminos de navegación, modelos de facilidad de uso, etcétera.

detalle, en especial el Modelo del Dominio y el Modelo de Diseño. Por brevedad, la tabla excluye los artefactos que pueden crearse en la fase de inicio (y se listaron en el Capítulo 4); presenta los artefactos que es más probable que se creen en la elaboración. Nótese que no se completarán en una iteración; sino que se refinarán a lo largo de una serie de iteraciones.

8.6. No se entendió la elaboración cuando...

- La duración es superior a “unos pocos” meses en la mayoría de los proyectos.
- Sólo comprende una iteración (con raras excepciones para los problemas bien entendidos).
- La mayoría de los requisitos se definieron antes de la elaboración.
- Los elementos arriesgados y el núcleo de la arquitectura no se abordan.
- El resultado no es una arquitectura *executable*; no hay programación de código de producción.
- Se considera fundamentalmente una fase de requisitos, que precede a una fase de implementación en la construcción.
- Se intenta llevar a cabo un diseño completo y cuidadoso antes de la programación.
- Existe una retroalimentación y adaptación mínima; los usuarios no se involucran continuamente en la evaluación y retroalimentación.
- No se llevan a cabo pruebas realistas en las primeras etapas.
- La arquitectura se termina de forma especulativa antes de la programación.
- Se considera una etapa para realizar la programación de pruebas de conceptos, en lugar de programar la arquitectura ejecutable básica de producción.
- No se realizan varios talleres de requisitos breves que adaptan y refinan los requisitos en base a la retroalimentación de las iteraciones anteriores y la actual.

Si un proyecto presenta estos síntomas, no se ha entendido la fase de elaboración.

Parte 3

ELABORACIÓN EN LA ITERACIÓN 1

Capítulo 9

MODELO DE CASOS DE USO: REPRESENTACIÓN DE LOS DIAGRAMAS DE SECUENCIA DEL SISTEMA

En teoría, no hay diferencia entre teoría y práctica. Pero, en la práctica, la hay.

Jan L. A. van de Snepscheut

Objetivos

- Identificar los eventos de sistema.
 - Crear diagramas de secuencia del sistema para los casos de uso.
-

Sigamos con la iteración 1

El proyecto PDV NuevaEra ha entrado en la primera iteración de desarrollo real. En la fase de inicio se ha realizado un trabajo ligero de requisitos para ayudar a decidir si merecía la pena más investigación seria en el proyecto. Se ha completado la planificación de la primera iteración, y se ha decidido abordar un escenario sencillo de éxito del caso de uso *Procesar Venta* para, únicamente, el pago en efectivo (sin colaboraciones remotas), con el objetivo de comenzar un diseño e implementación “amplio y superficial”, que incluya muchos de los elementos importantes del nuevo sistema. En la primera iteración, tienen lugar muchas tareas relacionadas con el establecimiento del entorno (herramientas, personal, procesos e instalación) que no serán consideradas.

Más bien, volvemos nuestra atención a los casos de usos y al análisis del modelado del dominio. Antes de empezar el trabajo de diseño de la iteración 1, resultará útil realizar un estudio adicional del dominio del problema. Parte de este estudio comprende la

aclaración de los eventos del sistema de entrada y salida relacionados con nuestro sistema, que puede representarse en diagramas de secuencia UML.

Introducción

Un diagrama de secuencia del sistema es un artefacto creado de manera rápida y fácil, que muestra los eventos de entrada y salida relacionados con el sistema que se está estudiando. UML incluye la notación de los diagramas de secuencia para representar los eventos que parten de los actores externos hacia el sistema.

9.1. Comportamiento del sistema

Antes de continuar con el diseño lógico de cómo funcionará la aplicación software, es conveniente estudiar y definir su comportamiento como una “caja negra”. El **comportamiento del sistema** es una descripción de *qué* hace el sistema, sin explicar cómo lo hace. Una parte de esa descripción es un diagrama de secuencia del sistema. Otras partes comprenden los casos de uso y los contratos del sistema (que se presentarán después).

9.2. Diagramas de secuencia del sistema

Los casos de uso describen cómo interactúan los actores externos con el sistema software que estamos interesados en crear. Durante esta interacción, un actor genera eventos sobre un sistema, normalmente solicitando alguna operación como respuesta. Por ejemplo, cuando un cajero inserta el ID de un artículo está solicitando al sistema PDV que registre la venta de ese artículo. Ese evento de solicitud inicia una operación sobre el sistema.

Es deseable aislar e ilustrar las operaciones que un actor externo solicita a un sistema, porque constituyen una parte importante de la comprensión del comportamiento del sistema. UML incluye los **diagramas de secuencia** como notación que puede representar las interacciones de los actores y las operaciones que inician.

Un **diagrama de secuencia del sistema** (DSS) es un dibujo que muestra, para un escenario específico de un caso de uso, los eventos que generan los actores externos, el orden y los eventos entre los sistemas. Todos los sistemas se tratan como cajas negras; los diagramas destacan los eventos que cruzan los límites del sistema desde los actores a los sistemas.

Debería hacerse un DSS para el escenario principal de éxito del caso de uso, y los escenarios alternativos complejos o frecuentes.

UML no define nada denominado diagrama de secuencia “del sistema”, sino simplemente diagrama de secuencia. La calificación se utiliza para subrayar su aplicación para representar sistemas como cajas negras. Posteriormente, se utilizarán los diagramas de secuencia en otro contexto —para ilustrar el diseño de la interacción entre objetos software para completar un trabajo.

9.3. Ejemplo de un DSS

Un DSS muestra, para un curso de eventos específico en un caso de uso, los actores externos que interactúan directamente con el sistema, el sistema (como una caja negra) y los eventos del sistema que genera el actor (ver Figura 9.1). El tiempo avanza hacia abajo, y la ordenación de los eventos debería seguir su orden en el caso de uso.

Los eventos del sistema podrían contener parámetros.

Este ejemplo muestra el escenario principal de éxito del caso de uso *Procesar Venta*. Se indica que el cajero genera los eventos del sistema *crearNuevaVenta*, *introducirArticulo*, *finalizarVenta*, y *realizarPago*.

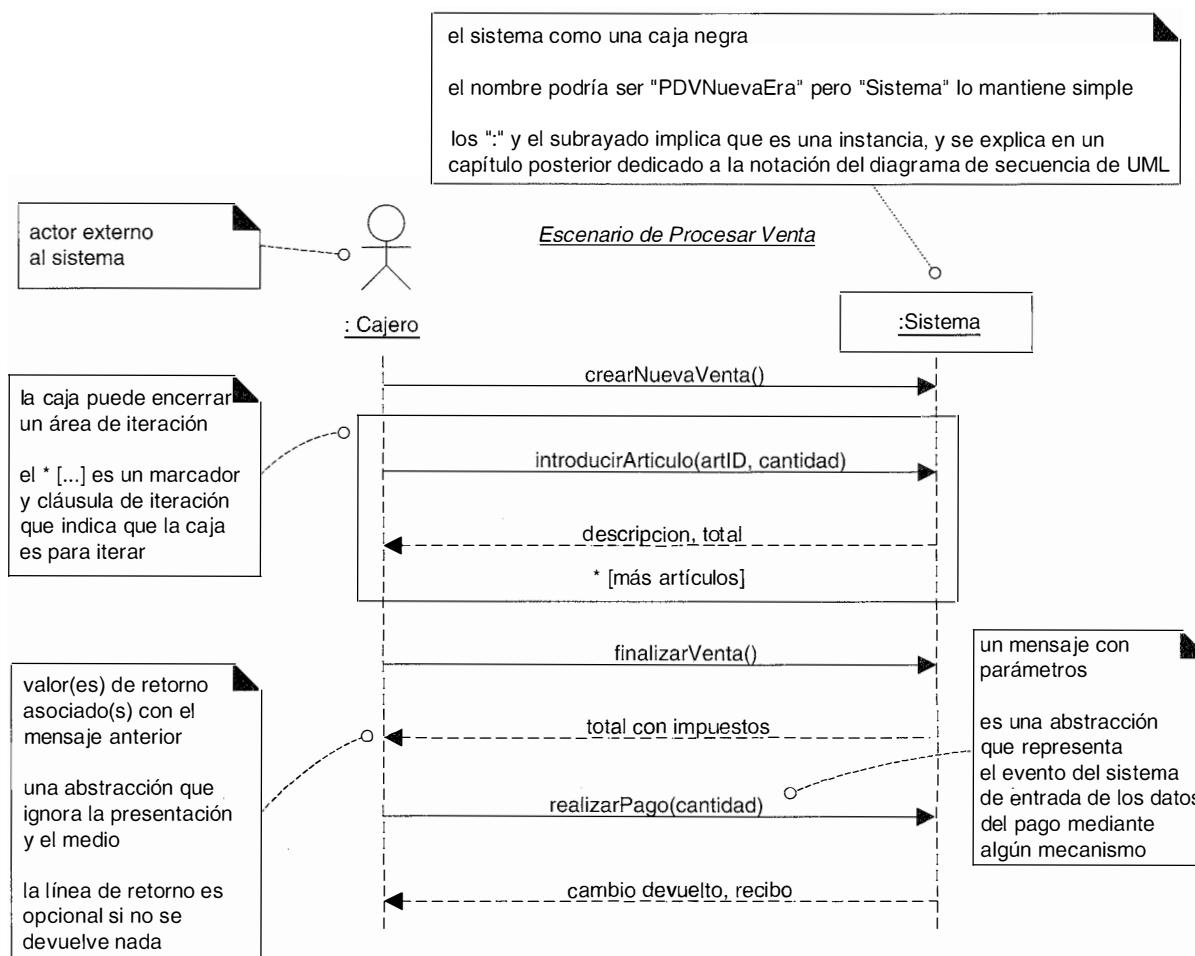


Figura 9.1. DSS para un escenario de *Procesar Venta*.

9.4. DSS entre sistemas

Los DSS también pueden utilizarse para ilustrar las colaboraciones entre sistemas, como entre el PDV NuevaEra y el sistema externo que autoriza pagos a crédito. Sin em-

bargo, se pospone hasta una iteración posterior en el caso de estudio, puesto que esta iteración no incluye las colaboraciones con sistemas remotos.

9.5. DSS y los casos de uso

Un DSS muestra los eventos del sistema para un escenario de un caso de uso, por tanto, se genera para el estudio de un caso de uso (ver Figura 9.2).

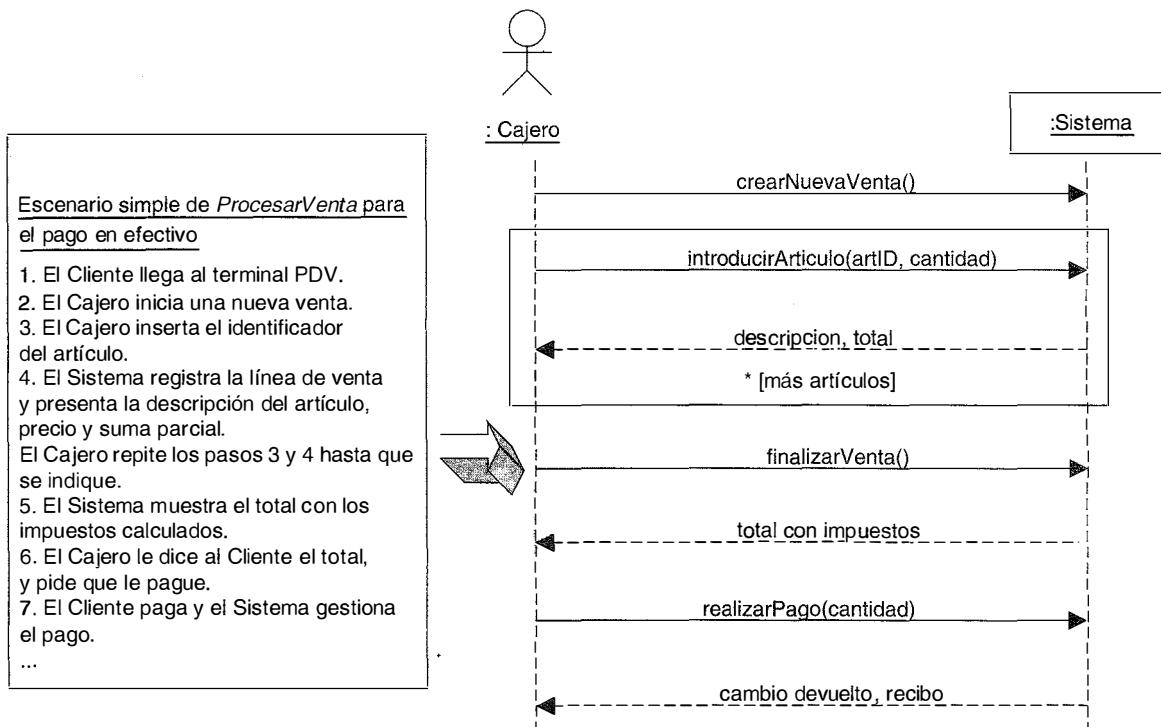


Figura 9.2. Los DSS se derivan de los casos de uso.

9.6. Eventos del sistema y los límites del sistema

Para identificar los eventos del sistema, es necesario tener claros los límites del sistema, como se presentó en el capítulo anterior sobre los casos de uso. Por lo que toca al desarrollo de software, el límite del sistema normalmente se elige para que sea el propio sistema software (y posiblemente hardware); en este contexto, un evento del sistema es un evento externo que lanza un estímulo directamente al software (ver Figura 9.3).

Consideremos el caso de uso *Procesar Venta* para identificar los eventos del sistema. Primero, debemos determinar los actores que interactúan directamente con el sistema software. El cliente interactúa con el cajero, pero para este escenario simple de pago en efectivo, no interactúa directamente con el sistema PDV —sólo lo hace el cajero—. Por tanto, el cliente no es un generador de eventos del sistema; sólo lo es el cajero.

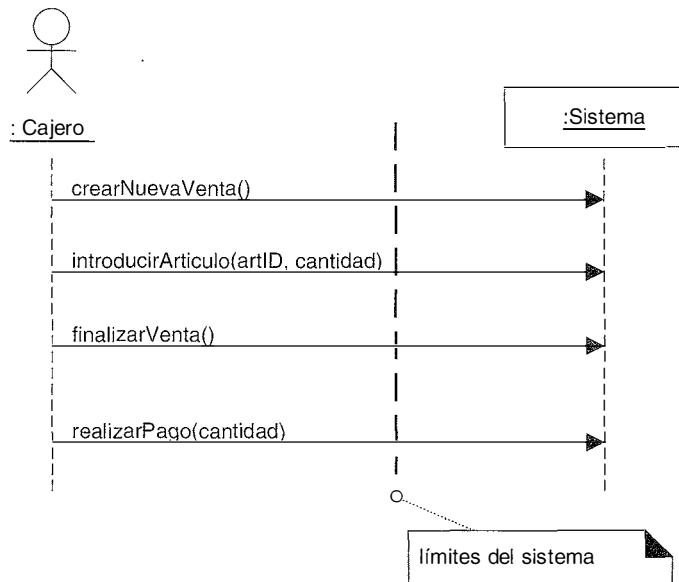


Figura 9.3. Definición de los límites del sistema.

9.7. Asignación de nombres a los eventos y operaciones

Los eventos del sistema (y sus operaciones del sistema asociadas) deberían expresarse al nivel de intenciones en lugar de en términos del medio de entrada físico o a nivel de elementos de la interfaz de usuario.

También se mejora la claridad, el comenzar el nombre de un evento del sistema con un verbo (añadir..., insertar..., finalizar..., crear...), como en la Figura 9.4, puesto que resalta la orientación de orden de estos eventos.

Así, “introducirArticulo” es mejor que “escanear” (esto es, escanear con láser) porque captura la intención de la operación, al mismo tiempo que permanece abstracta y sin compromiso respecto a las elecciones de diseño sobre qué interfaz utilizar para capturar el evento del sistema.

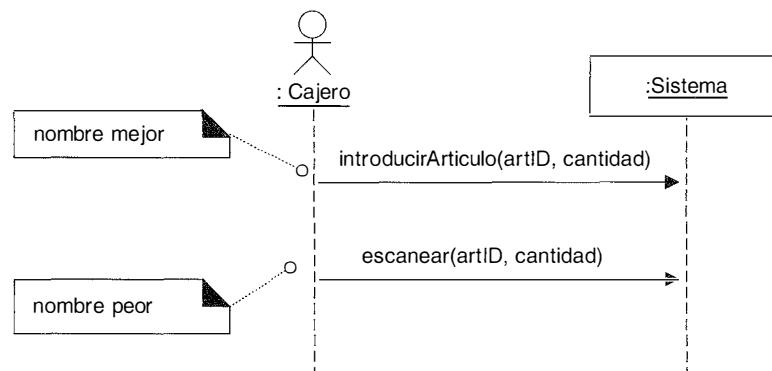


Figura 9.4. Elección de los nombres de los eventos y las operaciones a un nivel abstracto.

9.8. Mostrar el texto del caso de uso

A veces, es deseable mostrar al menos fragmentos del texto del caso de uso del escenario, con el fin de aclarar o enriquecer las dos vistas (ver Figura 9.5). El texto proporciona los detalles y el contexto; el diagrama resume visualmente la interacción.

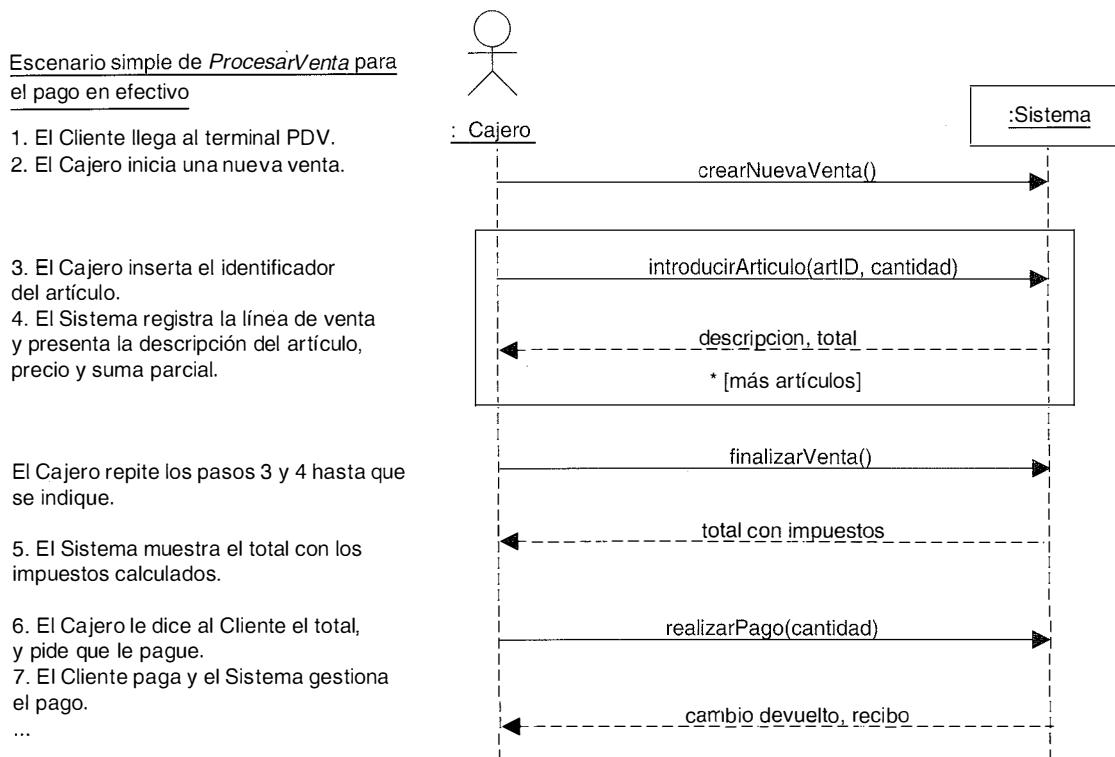


Figura 9.5. DSS con texto del caso de uso..

9.9. Los DSS y el Glosario

Los términos representados en los DSS (operaciones, parámetros, valores de retorno) son concisos. Éstos podrían necesitar una explicación más adecuada de manera que, durante el trabajo de diseño, esté claro qué es lo que entra y lo que sale. Si no se explicó en los casos de uso, podría utilizarse el Glosario.

Sin embargo, como siempre que discutimos la creación de artefactos distintos al código (lo esencial del proyecto), hay que ser desconfiado. Con los datos del Glosario debería hacerse algún uso o decisión realmente significativa, de lo contrario es un trabajo innecesario de poco valor.

9.10. DSS en el UP

Los DSSs forman parte del Modelo de Casos de Uso —una visualización de las interacciones implicadas en los casos de uso—. Los DSSs no se mencionaron explícitamente en la

descripción original del UP, aunque los creadores del UP eran conscientes y entendían la utilidad de tales diagramas. Los DSS son un ejemplo de los muchos posibles artefactos o actividades de análisis y diseño de utilidad que los documentos UP o el RUP no mencionan.

Fases

Inicio: Los DSS no se incentivan normalmente en la fase de inicio.

Elaboración: La mayoría de los DSS se crean durante la elaboración, cuando es útil identificar los detalles de los eventos del sistema para poner en claro cuáles son las operaciones que se deben diseñar que gestione el sistema, escribir los contratos de las operaciones del sistema (se presentarán en el Capítulo 13), y posiblemente, dar soporte a la estimación (por ejemplo, macroestimación con puntos de función no ajustados y COCOMO II).

Nótese que no es necesario crear DSS para todos los escenarios de todos los casos de uso —al menos no a la vez. Sino que, se crearán sólo para algunos escenarios seleccionados de la iteración actual.

Finalmente, sólo debería llevar unos pocos minutos o una media hora la creación de los DSSs.

Tabla 9.1. Muestra de los artefactos UP y evolución temporal. c – comenzar; r – refinar.

Disciplina	Artefacto Iteración →	Inicio I1	Elab. E1...En	Const. C1...Cn	Trans. T1...T2
Modelado del Negocio	Modelo del Dominio		c		
Requisitos	Modelo de Casos de Uso (DSS) Visión Especificación Complementaria Glosario	c c c c	r r r r		
Diseño	Modelo de Diseño Documento de Arquitectura SW Modelo de Datos		c c c	r r	
Implementación	Modelo de Implementación		c	r	r
Gestión del Proyecto	Plan de Desarrollo SW	c	r	r	r
Pruebas	Modelo de Pruebas		c	r	
Entorno	Marco de Desarrollo	c	r		

9.11. Lecturas adicionales

Durante décadas, se ha utilizado ampliamente variaciones de los diagramas que ilustran los eventos de entrada/salida de un sistema considerado como una caja negra; por ejemplo, en telecomunicaciones como diagramas de flujo de llamadas. Especialmente, se hicieron populares en los métodos orientados a objetos a través de su uso en el método Fusion [Coleman+94], que proporciona un ejemplo detallado de la relación de los DSSs y las operaciones del sistema con otros artefactos del análisis y diseño.

9.12. Artefactos del UP

La Figura 9.6 presenta ejemplos de relaciones de los DSSs con otros artefactos.

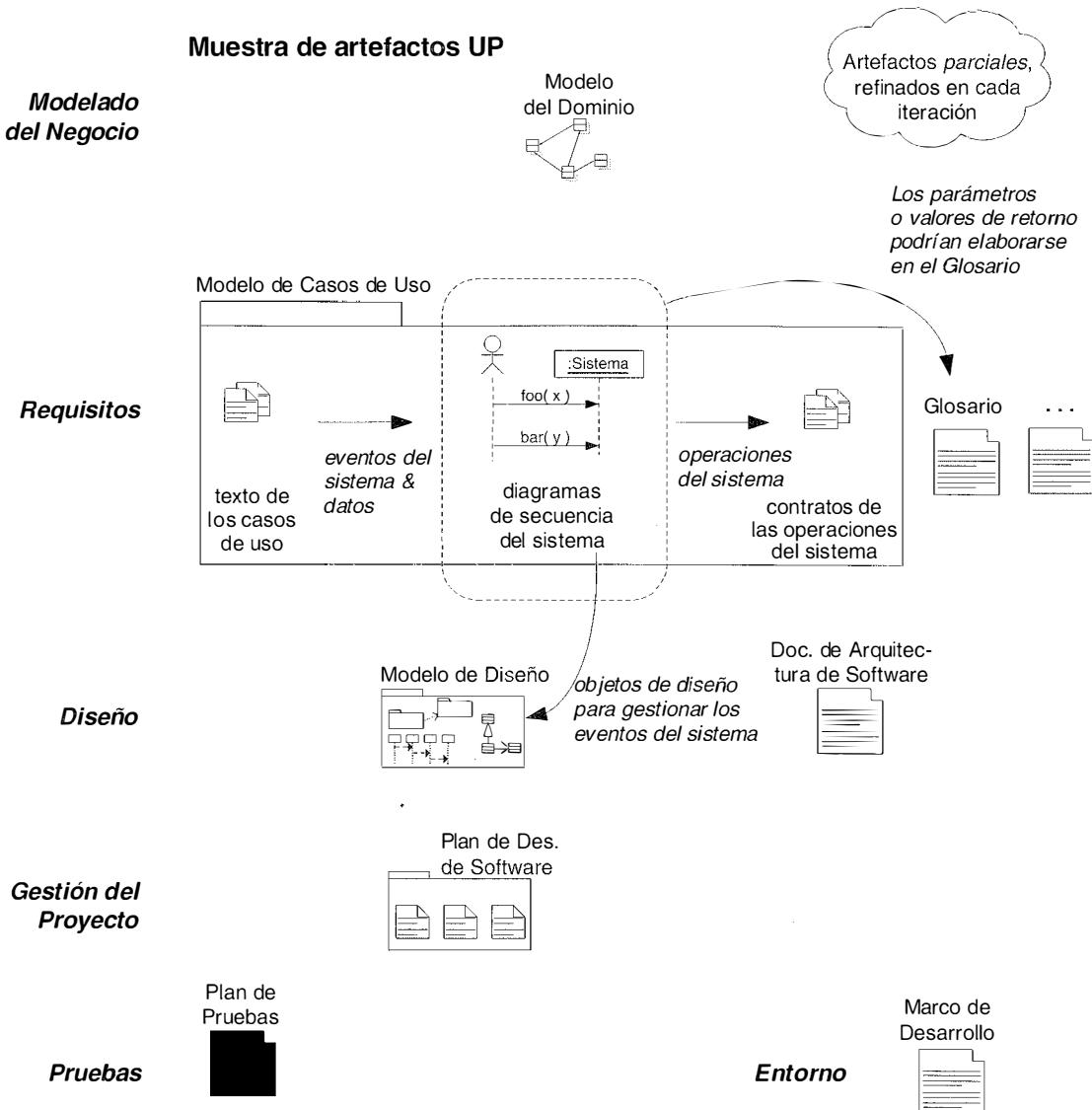


Figura 9.6. Muestra de la influencia de los artefactos UP.

Capítulo 10

MODELO DEL DOMINIO: VISUALIZACIÓN DE CONCEPTOS

Todo está muy bien en la práctica, pero nunca funcionará en la teoría.

Máxima anónima sobre la gestión

Objetivos

- Identificar las clases conceptuales relacionadas con los requisitos de la iteración actual.
 - Crear un modelo del dominio inicial.
 - Distinguir entre atributos correctos e incorrectos.
 - Añadir las clases conceptuales de *especificación*, cuando sea oportuno.
 - Comparar y contrastar las vistas conceptual y de implementación.
-

Introducción

Un modelo del dominio se utiliza con frecuencia como fuente de inspiración para el diseño de los objetos software, y será una entrada necesaria para varios de los siguientes artefactos que se presentan en este libro. Por tanto, es importante leer este capítulo si el lector no está familiarizado con el tema del modelado del dominio.

Un modelo del dominio muestra (a los modeladores) clases conceptuales significativas en un dominio del problema; es el artefacto más importante que se crea durante el análisis orientado a objetos¹. Este capítulo estudia técnicas introductorias a la creación de modelos del dominio. Los siguientes dos capítulos tratarán más extensamente las técnicas de modelado del dominio —añadiendo atributos y asociaciones—.

¹ Los casos de uso son un importante artefacto del análisis de requisitos, pero no son orientados a *objetos*. Ponen de relieve una vista de procesos del dominio.

La identificación de un conjunto rico de objetos o clases conceptuales es una parte esencial del análisis orientado a objetos, y bien merece la pena el esfuerzo en relación con los beneficios durante el trabajo de diseño e implementación.

La identificación de las clases conceptuales forma parte del estudio del dominio del problema. UML contiene notación, en forma de diagramas de clases, para representar los modelos del dominio.

Idea clave

Un modelo del dominio es una representación de las clases conceptuales del mundo real, no de componentes software. No se trata de un conjunto de diagramas que describen clases software, u objetos software con responsabilidades.

10.1. Modelos del dominio

La etapa orientada a *objetos* esencial del análisis o investigación es la descomposición de un dominio de interés en clases conceptuales individuales u objetos —las cosas de las que somos conscientes—. Un **modelo del dominio** es una representación *visual* de las clases conceptuales u objetos del mundo real en un dominio de interés [MO95, Fowler96]. También se les denomina **modelos conceptuales** (término utilizado en la primera edición de este libro), **modelo de objetos del dominio** y **modelos de objetos de análisis**².

El UP define un *Modelo de Dominio*³ como uno de los artefactos que podrían crearse en la disciplina del Modelado del Negocio.

Utilizando la notación UML, un modelo del dominio se representa con un conjunto de **diagramas de clases** en los que no se define ninguna operación. Pueden mostrar:

- Objetos del dominio o clases conceptuales.
- Asociaciones entre las clases conceptuales.
- Atributos de las clases conceptuales.

Por ejemplo, la Figura 10.1 muestra un modelo del dominio parcial. Ilustra que las clases conceptuales *Pago* y *Venta* son significativas en este dominio, que un *Pago* está relacionado con una *Venta* de un modo que es significativo hacer notar, y que una *Venta* tiene una fecha y una hora. Los detalles de la notación no son importantes en este momento.

Idea clave: Modelo del dominio—un diccionario visual de abstracciones

Por favor, reflexione un momento sobre la Figura 10.1. La figura visualiza y relaciona algunas palabras o clases conceptuales del dominio. También describe una *abstracción* de

² También están relacionados con los modelos conceptuales entidad-relación, que son capaces de mostrar vistas de los dominios puramente conceptuales, pero que han sido ampliamente re-interpretados como modelos de datos para el diseño de bases de datos. Los modelos del dominio no son modelos de datos.

³ Se utiliza *Modelo del Dominio* en mayúsculas cuando deseó resaltar que es un modelo oficial definido en el UP, frente al concepto bien conocido de “modelos de dominio”.

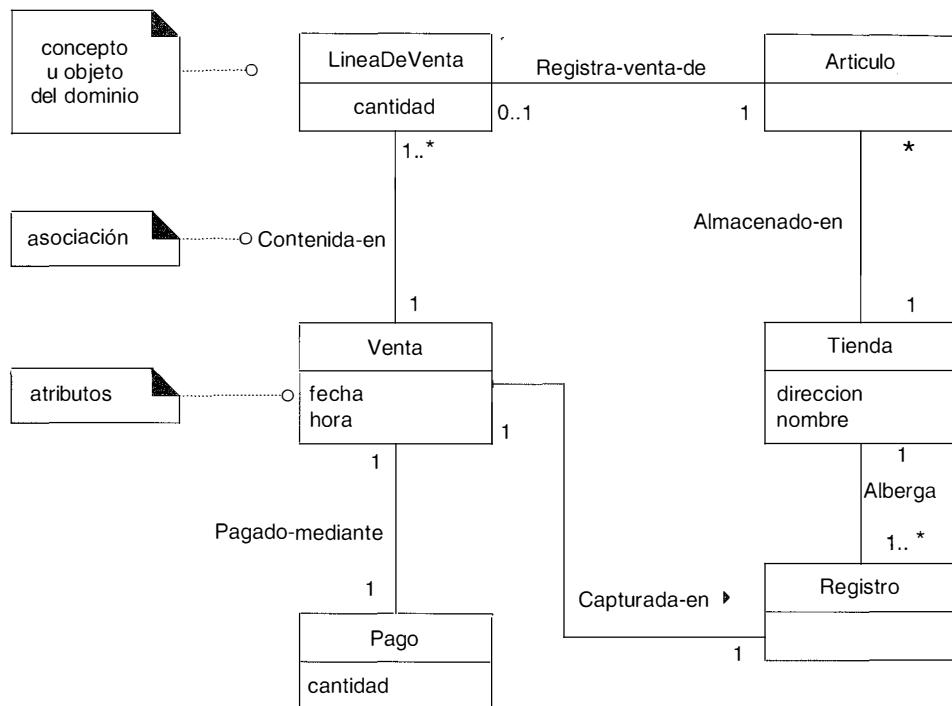


Figura 10.1. Modelo del dominio parcial —un diccionario visual—. El número de cada extremo de la línea indica la multiplicidad, que se describirá en un capítulo posterior.

las clases conceptuales, porque hay muchas cosas que uno podría comunicar sobre los registros, las ventas, etcétera. El modelo muestra una vista parcial, o abstracción, e ignora detalles sin interés (para el modelador).

La información que presenta (utilizando la notación UML) podría, de manera alternativa, haberse expresado en prosa, mediante sentencias en el Glosario o en algún otro sitio. Pero es fácil entender los distintos elementos y sus relaciones mediante este lenguaje visual, puesto que un porcentaje significativo del cerebro toma parte en el procesamiento visual —es una cualidad de los humanos—.

Por tanto, el modelo del dominio podría considerarse como un *diccionario visual* de las abstracciones relevantes, vocabulario del dominio e información del dominio.

Los modelos del dominio no son modelos de componentes software

Un modelo del dominio, como se muestra en la Figura 10.2, es una representación de las cosas del mundo real del dominio de interés, *no* de componentes software, como una clase Java o C++ (ver Figura 10.3), u objetos software con responsabilidades. Por tanto, los siguientes elementos no son adecuados en un modelo del dominio:

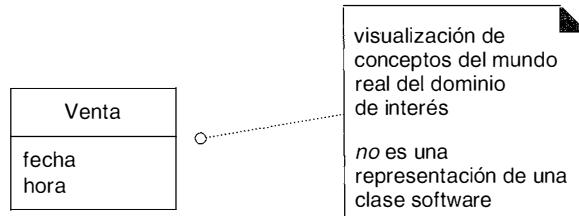


Figura 10.2. Un modelo del dominio muestra clases conceptuales del mundo real, no clases software.

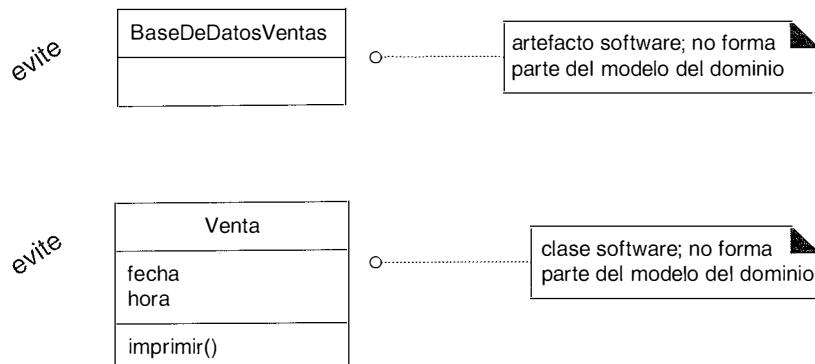


Figura 10.3. Un modelo del dominio no muestra artefactos software o clases.

- Artefactos software, como una ventana o una base de datos, a menos que el dominio que se esté modelando sea de conceptos software, como un modelo de interfaces de usuario gráficas.
- Responsabilidades o métodos⁴.

Clases conceptuales

El modelo del dominio muestra las clases conceptuales o vocabulario del dominio. Informalmente, una clase conceptual es una idea, cosa u objeto. Más formalmente, una clase conceptual podría considerarse en términos de su símbolo, intensión, y extensión [MO95] (ver Figura 10.4).

- **Símbolo:** palabras o imágenes que representan una clase conceptual.
- **Intensión:** la definición de una clase conceptual.
- **Extensión:** el conjunto de ejemplos a los que se aplica la clase conceptual.

Por ejemplo, considere la clase conceptual para el evento de una transacción de compra. Podría elegir nombrarla con el símbolo *Venta*. La intensión de una *Venta* podría es-

⁴ En el modelado de objetos, normalmente hablamos de responsabilidades relacionadas con los componentes software. Y los métodos son puramente conceptos software. Pero el modelo de dominio describe conceptos del mundo real, no componentes software. Es importante considerar las responsabilidades de los objetos durante el trabajo de *diseño*; sólo que no forma parte de este modelo. Un caso válido en el que se podrían mostrar las responsabilidades en un modelo de dominio es si incluye roles de trabajadores humanos (como el Cajero), y el modelador desea recoger las responsabilidades de estos trabajadores humanos.

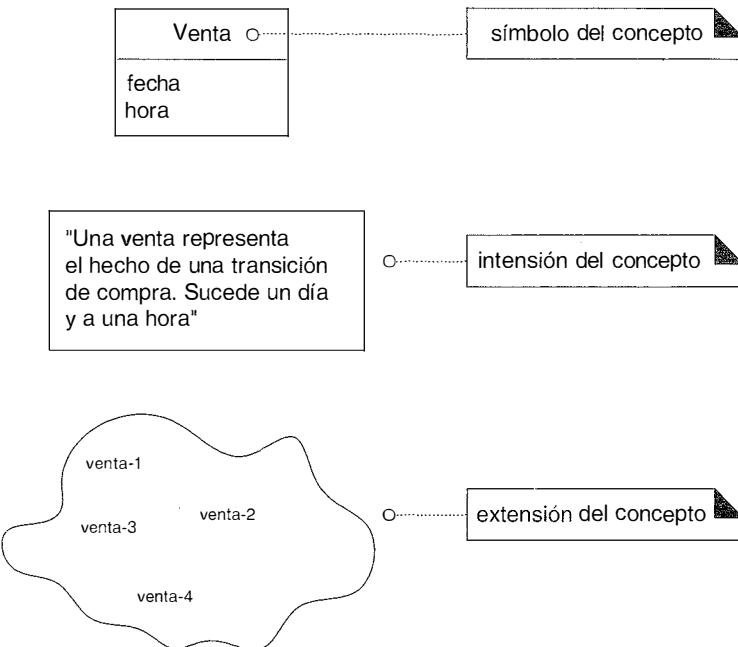


Figura 10.4. Una clase conceptual tiene un símbolo, una intención y una extensión.

tablecer que “representa el hecho de una transacción de una compra, y tiene una fecha y una hora”. La extensión de una *Venta* la forman todos los ejemplos de ventas; en otras palabras, el conjunto de todas las ventas.

Cuando creamos un modelo del dominio, normalmente, el símbolo y la vista intensional de la clase conceptual son los que tienen un mayor interés práctico.

Modelos y descomposición del dominio

Los problemas del software pueden ser complejos; la descomposición —divide y vencerás— es una estrategia común para tratar esta complejidad mediante la división del espacio del problema en unidades fáciles de comprender. En el **análisis estructurado**, la dimensión de la descomposición es por procesos o por *funciones*. Sin embargo, en el análisis orientado a objetos, la dimensión de la descomposición es fundamentalmente por cosas o entidades del dominio.

Una diferencia esencial entre el análisis orientado a objetos y el estructurado es: la división por clases conceptuales (objetos) en lugar de la división por funciones.

Por tanto, la principal tarea del análisis es identificar diferentes conceptos en el dominio del problema y documentar el resultado en un modelo del dominio.

Clases conceptuales en el dominio de ventas

Por ejemplo, en el dominio de ventas en una tienda del mundo real, existen las clases conceptuales de *Tienda*, *Registro* y *Venta*. Por tanto, nuestro modelo del dominio, mostrado en la Figura 10.5, podría incluir *Tienda*, *Registro* y *Venta*.



Figura 10.5. Modelo del dominio parcial en el dominio de la tienda.

10.2. Identificación de las clases conceptuales

Nuestro objetivo es crear un modelo del dominio de clases conceptuales interesantes o significativas del dominio de interés (ventas). En este caso, eso significa conceptos relacionados con el caso de uso *Procesar Venta*.

En el desarrollo iterativo, uno incrementalmente construye un modelo del dominio a lo largo de varias iteraciones en la fase de elaboración. En cada una, el modelo de dominio se limita a los escenarios anteriores y actual en estudio, en lugar de un modelo de “gran explosión”, que en las primeras etapas intenta capturar todas las posibles clases conceptuales y las relaciones. Por ejemplo, esta iteración está limitada al escenario de pago en efectivo de *Procesar Venta*; por tanto, se creará un modelo de dominio parcial únicamente para reflejar eso —no más—.

La tarea central es, por tanto, identificar las clases conceptuales relacionadas con el escenario que se está diseñando.

A continuación presentamos una guía útil para la identificación de las clases conceptuales:

Es mejor especificar en exceso un modelo del dominio con muchas clases conceptuales de grano fino que especificar por defecto.

No piense que un modelo de dominio es mejor si contiene pocas clases conceptuales; suele ser verdad justamente lo contrario.

Es normal obviar clases conceptuales durante la etapa de identificación inicial, y descubrirlas más tarde al considerar los atributos y asociaciones, o durante el trabajo de diseño. Cuando se encuentren, se pueden añadir al modelo del dominio.

No excluya una clase conceptual simplemente porque los requisitos no indican ninguna necesidad obvia para registrar información sobre ella (un criterio común en el modelado de datos para el diseño de bases de datos relacionales, pero no relevante en el modelado del dominio), o porque la clase conceptual no tiene atributos.

Es válido tener clases conceptuales sin atributos, o clases conceptuales con un rol puramente de comportamiento en el dominio, en lugar de un rol de información.

Estrategias para identificar clases conceptuales

En las siguientes secciones se presentan dos técnicas:

1. Utilización de una lista de categorías de clases conceptuales.
2. Identificación de frases nominales.

Otra excelente técnica para el modelado del dominio es el uso de **patrones de análisis**, que son modelos de dominios parciales existentes creados por expertos, utilizando libros publicados como *Analysis Patterns* [Fowler96] y *Data Model Patterns* [Hay96].

Utilización de una lista de categorías de clases conceptuales

Comience la creación de un modelo del dominio haciendo una lista de clases conceptuales candidatas. La Tabla 10.1 contiene muchas categorías habituales que, normalmente, merece la pena tener en cuenta, aunque no en ningún orden particular de importancia. Los ejemplos se han extraído del dominio de las tiendas y las reservas de vuelos.

Tabla 10.1. Lista de categorías de clases conceptuales.

Categoría de clase conceptual	Ejemplos
objetos tangibles o físicos	<i>RegistroAvion</i>
especificaciones, diseños o descripciones de la cosas	<i>EspecificacionDelProducto</i> <i>DescripcionDelVuelo</i>
lugares	<i>Tienda</i>
transacciones	<i>Venta, Pago</i> <i>Reserva</i>
líneas de la transacción	<i>LineaDeVenta</i>
roles de la gente	<i>Cajero</i> <i>Piloto</i>
contenedores de otras cosas	<i>Tienda, Lata</i> <i>Avion</i>
cosas en un contenedor	<i>Articulo</i> <i>Pasajero</i>
otros sistemas informáticos o electromecánicos externos al sistema	<i>SistemaAutorizacionPagoCredito</i> <i>ControlDeTraficoAereo</i>
conceptos abstractos	<i>Ansia</i> <i>Acrofobia</i>
organizaciones	<i>DepartamentoDeVentas</i> <i>CompaniaAerea</i>
hechos	<i>Venta, Pago, Reunion</i> <i>Vuelo, Colision, Aterrizaje</i>

continúa

Tabla 10.1. Lista de categorías de clases conceptuales. (Continuación)

Categoría de clase conceptual	Ejemplos
procesos (normalmente <i>no</i> se representan como conceptos, pero podría ocurrir)	<i>VentaDeUnProducto</i> <i>ReservaUnAsiento</i>
reglas y políticas	<i>PoliticaDeReintegro</i> <i>PoliticaDeCancelación</i>
catálogos	<i>CatalogoDeProductos</i> <i>CatalogoDePiezas</i>
registros de finanzas, trabajo, contratos, cuestiones legales	<i>Recibo</i> , <i>LibroMayor</i> , <i>ContratoEmpleo</i> <i>RegistroMantenimiento</i>
instrumentos y servicios financieros	<i>LíneaDeCredito</i> <i>Stock</i>
manuales, documentos, artículos de referencia, libros	<i>ListaDeCambiosDePreciosDiarios</i> <i>ManualReparaciones</i>

Descubrimiento de clases conceptuales mediante la identificación de frases nominales

Otra técnica útil (debido a su simplicidad) recomendada en [Abbot83] es el análisis lingüístico: identificar los nombres y frases nominales en las descripciones textuales de un dominio, y considerarlos como clases conceptuales o atributos candidatos.

Se debe tener cuidado con este método; no es posible realizar una correspondencia mecánica de nombres a clases, y las palabras en lenguaje natural son ambiguas.

En cualquier caso, es otra fuente de inspiración. Los casos de uso en formato completo constituyen una descripción excelente a partir de la cual extraer este análisis. Por ejemplo, se puede utilizar el escenario actual del caso de uso *Procesar Venta*.

Escenario principal de éxito (o Flujo Básico):

1. El Cliente llega a un terminal PDV con **mercancías** y/o **servicios** que comprar.
2. El Cajero comienza una nueva **venta**.
3. El Cajero introduce el **identificador del artículo**.
4. El Sistema registra la **línea de la venta** y presenta la **descripción del artículo**, **precio** y **suma** parcial. El precio se calcula a partir de un conjunto de reglas de precios. El Cajero repite los pasos 3-4 hasta que se indique.
5. El Sistema presenta el total con los **impuestos** calculados.
6. El Cajero le dice al Cliente el total y solicita el **pago**.
7. El Cliente paga y el Sistema gestiona el pago.
8. El Sistema registra la **venta** completa y envía la información de la venta y el pago al sistema de **Contabilidad** externo (para la contabilidad y las **comisiones**) y al sistema de **Inventario** (para actualizar el inventario).
9. El Sistema presenta el **recibo**.
10. El Cliente se va con el recibo y las mercancías (si es el caso).

Extensiones (o Flujos Alternativos):

...

7a. Pago en efectivo:

1. El Cajero introduce la **cantidad** de dinero ~~entregada~~ en efectivo.
2. El Sistema muestra la **cantidad** de dinero a devolver y abre el **cajón de caja**.
3. El Cajero deposita el dinero entregado y devuelve el cambio al Cliente.
4. El Sistema registra el pago en efectivo.

El modelo del dominio es una visualización de los conceptos del dominio y vocabulario relevantes. ¿Dónde se encuentran estos términos? En los casos de uso. Por tanto, constituyen una fuente rica a explorar mediante la identificación de frases nominales.

Algunas de las frases nominales son clases conceptuales candidatas, algunas podrían hacer referencia a clases conceptuales que se ignoran en esta iteración (por ejemplo, “Contabilidad” y “comisiones”), y algunas podrían ser atributos de las clases conceptuales. Por favor, diríjase a la siguiente sección y al capítulo sobre los atributos, para obtener los consejos que permiten diferenciar entre los dos.

Un punto débil de este enfoque es la imprecisión del lenguaje natural; frases nominales diferentes podrían representar la misma clase conceptual o atributo, entre otras ambigüedades. Sin embargo, se recomienda que se combine con la técnica la *Lista de Categorías de Clases Conceptuales*.

10.3. Clases conceptuales candidatas para el dominio de ventas

A partir del análisis de la Lista de Categorías de Clases Conceptuales y las frases nominales, se genera una lista de clases conceptuales candidatas del dominio. La lista está restringida a los requisitos y simplificaciones que se están estudiando actualmente —el escenario simplificado de *Procesar Venta*—.

<i>Registro</i>	<i>EspecificacionDelProducto</i>
<i>Articulo</i>	<i>LinedeVenta</i>
<i>Tienda</i>	<i>Cajero</i>
<i>Venta</i>	<i>Cliente</i>
<i>Pago</i>	<i>Encargado</i>
<i>CatalogoDeProductos</i> ⁵	

No existe una lista “correcta”. Es una colección algo arbitraria de abstracciones y vocabulario del dominio que el modelador considera relevantes. En cualquier caso, siguiendo la estrategia de identificación, diferentes modeladores producirán listas similares.

⁵ *N. del T.:* Aunque siguiendo las convenciones para nombrar las clases no se utilizaría la preposición, en este caso al igual que en *EspecificacionDelProducto* y *LinedeVenta*, se han utilizado para mejorar la legibilidad del texto.

Objetos de informes: ¿incluir el recibo en el modelo?

Un recibo es un informe de una venta y del pago, y una clase conceptual relativamente destacable del dominio, por tanto, ¿debería mostrarse en el modelo?

A continuación presentamos algunos factores a tener en cuenta:

- Un recibo es un informe de una venta. En general, no es útil mostrar un informe de otra información en un modelo del dominio puesto que toda esta información se deriva de otras fuentes; duplica información que se encuentra en otras partes. Ésta es una razón para excluirlo.
- Un recibo tiene un rol especial en término de las reglas del negocio: normalmente, confiere al portador del recibo, el derecho a devolver los artículos comprados. Ésta es una razón para mostrarlo en el modelo.

Puesto que la devolución de artículos no está siendo considerada en esta iteración, el *Recibo* será excluido. Durante la iteración que aborda el caso de uso *Gestionar Devoluciones*, estaría justificada su inclusión.

10.4. Guías para el modelado del negocio

Cómo hacer un modelo del dominio

Aplique los siguientes pasos para crear un modelo del dominio:

1. Liste las clases conceptuales candidatas, utilizando las técnicas de la Lista de Categorías de Clases Conceptuales y la identificación de frases nominales, relacionadas con los requisitos actuales en estudio.
2. Represéntelos en un modelo del dominio.
3. Añada las asociaciones necesarias para registrar las relaciones que hay que mantener en memoria (se discutirá en un capítulo siguiente).
4. Añada los atributos necesarios para satisfacer los requisitos de información (se discutirá en un capítulo siguiente).

Un método útil auxiliar es aprender y copiar patrones de análisis, que se discutirán en un capítulo posterior.

Nombrar y modelar cosas: el cartógrafo

La estrategia del cartógrafo se aplica tanto a los mapas como a los modelos del dominio.

- Haga un modelo del dominio con el espíritu del modo de trabajo de los cartógrafos:
- Utilice los nombres existentes en el territorio.
 - Excluya las características irrelevantes.
 - No añada cosas que no están ahí.

Un modelo del dominio es un tipo de mapa de conceptos o cosas de un dominio. Este espíritu destaca el rol analítico de un modelo del dominio, y sugiere lo siguiente:

- Un cartógrafo utiliza los nombres del territorio —no cambia los nombres de las ciudades en un mapa—. Para un modelo del dominio, significa que *utilice el vocabulario del dominio al nombrar los nombres de las clases conceptuales y los atributos*. Por ejemplo, si estamos desarrollando un modelo para una biblioteca, nombre al cliente como “*Prestatario*” o “*Socio*” —los términos utilizados por el personal de la biblioteca—.
- Un cartógrafo elimina las cosas de un mapa si no se consideran relevantes para el propósito del mapa; por ejemplo, la topografía o la población no es necesario que se muestren. Análogamente, un modelo de dominio podría excluir clases conceptuales del dominio del problema que no son pertinentes para los requisitos. Por ejemplo, podríamos excluir *Bolígrafo* y *BolsaPapel* de nuestro modelo del dominio (para el conjunto de requisitos actual) puesto que no tienen un rol relevante obvio.
- Un cartógrafo no muestra cosas que no estén ahí, como montañas que no existen. Igualmente, el modelo del dominio debería excluir cosas que *no* se encuentran en el dominio del problema que se está estudiando.

El principio también se conoce como la estrategia *Utilice el Vocabulario del Dominio* [Coad95].

Error típico en la identificación de las clases conceptuales

Quizás el error más típico al crear un modelo del dominio es representar algo como un atributo cuando debería haber sido un concepto. Una regla empírica para ayudar a prevenir este error es:

Si no consideramos alguna clase conceptual X que sea un número o texto en el mundo real, X es probablemente una clase conceptual, no un atributo.

Como ejemplo, ¿debería ser la *tienda* un atributo de *Venta*, o una clase conceptual separada *Tienda*?

Venta
tienda

¿o...?

Venta

Tienda
numeroTelefono

En el mundo real, una tienda no se considera un número o un texto —el término sugiere una entidad legal, una organización, y algo que ocupa espacio—. Por tanto, *Tienda* debe ser un concepto.

Vuelo
destino

¿o...?

Vuelo

Aeropuerto
nombre

Otro ejemplo, considere el dominio de las reservas de vuelos. ¿Debería ser el *destino* un atributo de *Vuelo*, o una clase conceptual aparte, *Aeropuerto*?

En el mundo real, un aeropuerto de destino no se considera un número o un texto —es una cosa grande que ocupa espacio—. Por tanto, *Aeropuerto* debería ser un concepto.

En caso de duda, considérelo un concepto separado. Los atributos deberían ser bastante raros en un modelo del dominio.

10.5. Resolución de clases conceptuales similares: Registro vs. “TPDV”

TPDV son las siglas de terminal de punto de venta. En el lenguaje informático, un terminal es cualquier dispositivo de un sistema, como un PC cliente, un PDA en red inalámbrico, etcétera. En los primeros tiempos, mucho antes a los TPDVs, una tienda mantenía un *registro* —un libro en el que se apuntaban las ventas y pagos—. Con el tiempo, esto se automatizó en “cajas registradoras” mecánicas. Hoy en día, un TPDV desempeña el rol del registro (ver Figura 10.6).

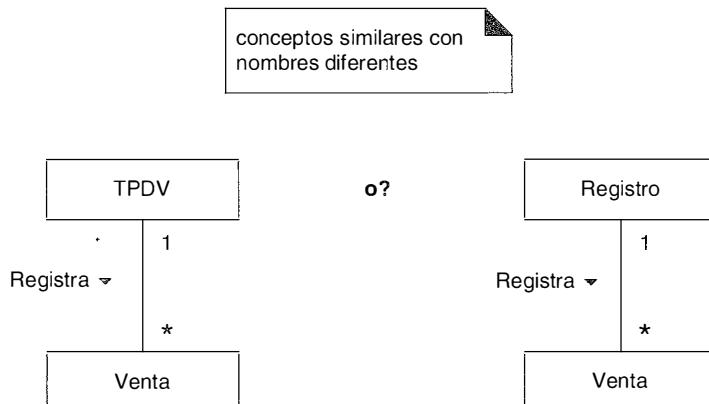


Figura 10.6. TPDV y Registro son clases conceptuales similares.

Un registro es una cosa que recoge las ventas y pagos, pero esto es un TPDV. Sin embargo, el término *registro* parece algo más abstracto y menos orientado a la implementación que un *TPDV*. Por tanto, en el modelo del dominio, ¿debería utilizarse el símbolo *Registro* en lugar de *TPDV*?

Primero, como regla empírica, un modelo del dominio no es absolutamente correcto o equivocado, sino más o menos útil; es una herramienta de comunicación.

Por el principio del cartógrafo, “*TPDV*” es un término familiar en el territorio, de manera que es un símbolo útil desde el punto de vista de la familiaridad y la comunicación. Por el objetivo de la creación de modelos que representan abstracciones y son indepen-

dientes de la implementación, *Registro* es atractivo y útil⁶. Se podría considerar *Registro* justamente, para representar tanto la clase conceptual de un sitio que registra las ventas, como una abstracción de varias clases de terminales, como un TPDV.

Ambas elecciones tienen valor; en este caso de estudio se ha elegido *Registro* algo arbitrariamente, pero el personal involucrado también habría entendido *TPDV*.

10.6. Modelado del mundo *irreal*

Algunos sistemas software son para dominios que encuentran muy poca analogía con dominios naturales o de negocios; el software para las telecomunicaciones es un ejemplo. Todavía es posible crear un modelo del dominio en estos dominios, pero requiere un alto grado de abstracción y olvidarse de diseños familiares.

Por ejemplo, algunas de las clases conceptuales candidatas relacionadas con conmutadores de telecomunicaciones son: *Mensaje*, *Conexion*, *Puerto*, *Dialogo*, *Ruta*, *Protocolo*.

10.7. Clases conceptuales de especificación o descripción

La siguiente discusión podría, al principio, parecer relacionada con un tema raro y altamente especializado. Sin embargo, prueba que la necesidad de las clases conceptuales de especificación (como se definirán) es habitual en muchos modelos del dominio. Por tanto, se destaca.

Asuma lo siguiente:

- Una instancia de un *Articulo* representa un objeto físico de una tienda; como tal, podría incluso tener un número de serie.
- Un *Articulo* tiene una descripción, precio, e identificador del artículo (articuloID), que no se recogen en ningún otro sitio.
- Todo el mundo que trabaja en la tienda tiene amnesia.
- Cada vez que se vende un artículo físico real, se elimina una instancia de *Articulo* software correspondiente, del “terreno del software”.

Con estas suposiciones, ¿qué ocurre en el siguiente escenario?

Existe una fuerte demanda de las nuevas hamburguesas vegetales que han tenido mucho éxito —ObjetoHamburguesa—. La tienda las vende todas, lo que implica que se eliminan de la memoria del ordenador todas las instancias de *Articulo* de los Objeto-Hamburguesa.

⁶ Nótese que en los primeros tiempos, un *registro* era únicamente una posible implementación del modo de registrar las ventas. El término ha adquirido un significado generalizado a lo largo del tiempo..

Ahora, aquí está lo esencial del problema: Si alguien pregunta: “¿Cuánto cuestan los ObjetosHamburguesa?”, nadie puede contestar, porque el precio se encontraba en las instancias inventariadas, que se eliminaron cuando se vendieron.

Nótese también que el modelo actual, si se implementa en el software tal y como se describe, contiene datos duplicados y es ineficiente en cuanto al espacio, puesto que la descripción, el precio y el ID, se duplican en cada instancia de *Articulo* del mismo producto.

La necesidad de especificación o descripción de las clases conceptuales

El problema anterior ilustra la necesidad de conceptos de objetos que sean especificaciones o descripciones de otras cosas. Para solucionar el problema del *Articulo*, lo que se necesita es una clase conceptual *EspecificacionDelProducto* (o *EspecificacionDelArticulo*, *DescripcionDelProducto*, ...) que recoge la información sobre los artículos. Una *EspecificacionDelProducto* no representa un *Articulo*, sino una descripción de la información sobre los artículos. Nótese que incluso si todos los elementos inventariados se venden, y sus correspondientes instancias software de *Articulo* se eliminan, permanece todavía la *EspecificacionDelProducto*.

Los objetos de descripción o especificación están fuertemente relacionados con las cosas que describen. En un modelo del dominio, es típico establecer que una *EspecificacionDeX Describe un X* (ver Figura 10.7).

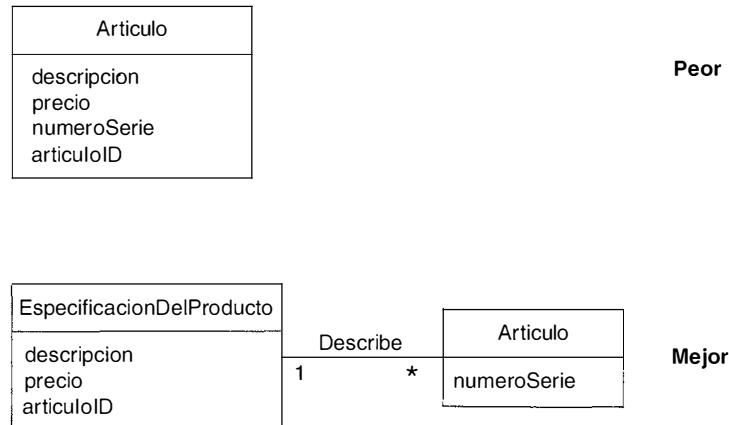


Figura 10.7. Especificaciones o descripciones sobre otras cosas. El “*” significa una multiplicidad de “muchos”. Indica que una *EspecificacionDelProducto* podría describir a muchos (*) *Articulos*.

La necesidad de clases conceptuales de especificación es habitual en los dominios de ventas y productos. También es típica en la fabricación, donde se requiere una *descripción* de la cosa que se fabrica, que es distinto de la cosa en sí misma. Se ha dedicado tiempo y espacio a motivar las clases conceptuales de especificación porque son muy frecuentes; no es un concepto de modelado raro.

¿Cuándo se requieren las clases conceptuales de especificación?

Las siguientes guías sugieren cuándo utilizar las especificaciones:

Añada una clase conceptual de especificación o descripción (por ejemplo, *EspecificacionDelProducto*) cuando:

- Se necesita la descripción de un artículo o servicio, independiente de la existencia actual de algún ejemplo de esos artículos o servicios.
- La eliminación de instancias de las cosas que describen (por ejemplo, *Articulo*) dan como resultado una pérdida de información que necesita mantenerse, debido a la asociación incorrecta de información con la cosa eliminada.
- Reduce información redundante o duplicada.

Otro ejemplo de especificación

Como otro ejemplo, considere una compañía aérea que sufre un accidente catastrófico de uno de sus aviones. Asuma que se cancelan todos los vuelos durante seis meses, pendiente de que se complete la investigación. También asuma que, cuando se cancelan los vuelos, su correspondiente objeto software *Vuelo*, se elimina de la memoria del ordenador. Por tanto, después de la colisión, se eliminan todos los objetos software *Vuelo*.

Si el único registro de los aeropuertos de destino de los vuelos se encuentra en las instancias software *Vuelo*, que representa un vuelo específico en una fecha y hora concreta, ya no habrá un registro de las rutas de los vuelos de la compañía.

Para solucionar este problema, se necesita una *DescripcionDelVuelo* (o *EspecificacionDelVuelo*) que describe un vuelo y su ruta, incluso cuando no se ha planificado ningún vuelo concreto (ver Figura 10.8).

Descripción de servicios

Nótese que el ejemplo anterior trata sobre un servicio (un vuelo) en lugar de un artículo (como una hamburguesa vegetal). Es habitual que se necesiten descripciones de servicios o planes de servicios.

Como otro ejemplo, una compañía de telefonía móvil vende paquetes con denominaciones como “bronce”, “oro”, etcétera. Es necesario tener el concepto de descripción del paquete (un tipo de plan de servicio que describe las tarifas por minuto, acceso a Internet sin cable, el coste, etcétera) separado del concepto del paquete realmente vendido (como el “paquete oro vendido a Craig Larman el 1 de enero de 2002, a \$55 al mes”). El departamento de Marketing necesita definir y registrar este plan de servicios o *DescripcionDelPaqueteDeComunicacionesMoviles* antes de que se venda alguno.

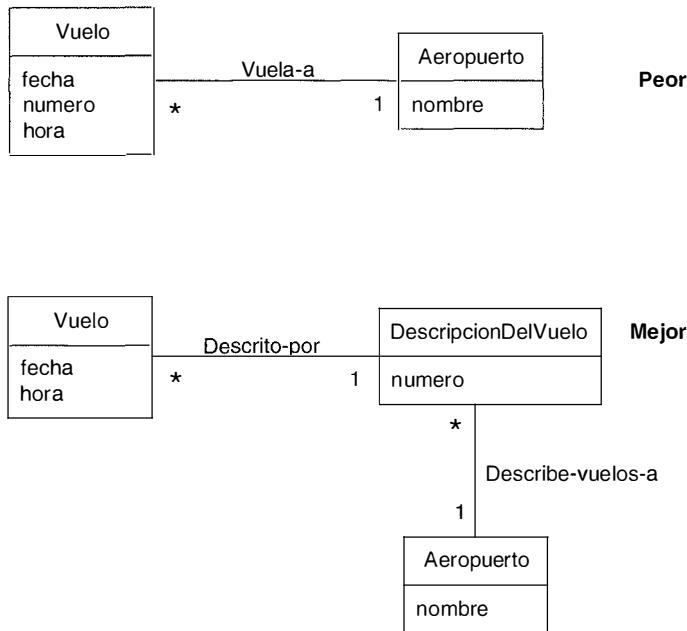


Figura 10.8. Especificaciones sobre otras cosas.

10.8. Notación UML, modelos y métodos: perspectivas múltiples

El UP define algo denominado Modelo del Dominio, que se representa con la notación UML. Sin embargo, no existe un término “Modelo del Dominio” en la documentación oficial de UML. Esto nos revela algo importante:

UML simplemente describe tipos de diagramas, como los diagramas de clases y los diagramas de secuencia. No superpone un método o perspectiva de modelado sobre ellos. Más bien, un proceso (como el UP) aplica la notación de la especificación de UML en el contexto de modelos definidos en el ámbito de una metodología.

Por ejemplo, la notación de los diagramas de clases de UML se puede utilizar para crear representaciones de las clases conceptuales del dominio (un modelo del dominio), clases software, tablas de una base de datos relacional, etcétera.

Por tanto, no debemos confundir la notación básica de los diagramas UML, con su aplicación para visualizar distintos tipos de modelos definidos por los metodólogos (ver Figura 10.9). Este punto es aplicable no sólo a los diagramas de clases UML, sino a la mayoría de la notación UML.

Como otro ejemplo de los diagramas de UML que se interpretan de manera diferente en modelos distintos, los diagramas de secuencia UML se pueden utilizar para representar el paso de mensajes entre los objetos software (como en el Modelo de Diseño del UP), o la interacción entre personas y grupos en el mundo real (como en el Modelo de Objetos del Negocio del UP).

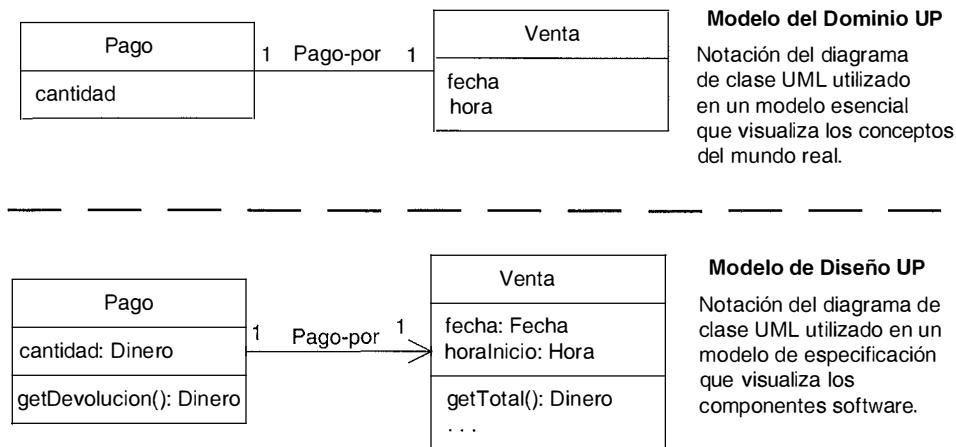


Figura 10.9. La notación de la especificación UML se puede aplicar en diferentes perspectivas y modelos definidos por un proceso o método.

Esta idea se pone de relieve en el método orientado a objetos Syntropy [CD94], y Martin Fowler la reitera en *UML Distilled* [FS00]. Es decir, la misma notación basada en diagramas se puede utilizar en tres perspectivas y tipos de modelos:

1. **Perspectiva esencial o conceptual:** se interpreta que los diagramas describen cosas del mundo real o de un dominio de interés.
2. **Perspectiva de especificación:** se interpreta que los diagramas (utilizando la misma notación de los modelos esenciales) describen abstracciones software o componentes con especificaciones e interfaces, pero no comprometidas a ninguna implementación en particular (por ejemplo, no específicamente una clase C# o Java).
3. **Perspectiva de implementación:** se interpreta que los diagramas (utilizando la misma notación de los modelos esenciales) describen implementaciones software con una tecnología y lenguaje particular (como Java).

Superposición de terminología: UML vs. métodos

En la especificación UML, las cajas rectangulares mostradas en la Figura 10.9 se denominan **clases**, pero nótese que en UML, este término abarca una variedad de fenómenos —cosas físicas, cosas del software, eventos, etcétera⁷—. Un proceso o método superpondrá una terminología alternativa sobre UML. Por ejemplo, en el UP, cuando las cajas de UML se dibujan en el Modelo del Dominio, podrían llamarse **conceptos del dominio** o **clases conceptuales**; el Modelo del Dominio ofrece una perspectiva conceptual. En el UP, cuando las cajas UML se dibujan en el Modelo de Diseño, se denominan oficialmente **clases de diseño**: el Modelo de Diseño ofrece una perspectiva de especificación o implementación, como quiere el modelador.

⁷ Una clase UML es un caso especial de un elemento de modelado UML muy general el **clasificador** —algo con características estructurales y/o comportamiento, incluyendo clases, actores, interfaces y casos de uso—.

Independientemente de la definición, la cuestión importante es que es útil distinguir entre la perspectiva de un analista que mira conceptos del mundo real tal como una venta (una perspectiva conceptual), y los diseñadores de software que especifican componentes software como la clase software *Venta* (una perspectiva de especificación o implementación).

UML se puede utilizar para ilustrar ambas perspectivas con una notación y perspectiva muy similar, así que es muy importante tener en cuenta qué perspectiva se está tomando.

Para mantener las cosas claras, este libro utilizará los términos relacionados con las clases de la siguiente forma, que es consistente con UML y el UP:

- **Clase conceptual:** concepto o cosa del mundo real. Una perspectiva conceptual o esencial. El Modelo del Dominio del UP contiene clases conceptuales.
- **Clase software:** una clase que representa una perspectiva de especificación o implementación de un componente software, independientemente del proceso o método.
- **Clase de diseño:** un miembro del Modelo de Diseño del UP. Es un sinónimo de clase software, pero por alguna razón deseo resaltar que es una clase del Modelo de Diseño. El UP permite que una clase de diseño tenga perspectiva de especificación o implementación, según desee el modelador.
- **Clase de implementación:** una clase implementada en un lenguaje orientado a objetos como Java.
- **Clase:** como en UML, el término general que representa o una cosa del mundo real (una clase conceptual) o del software (una clase software).

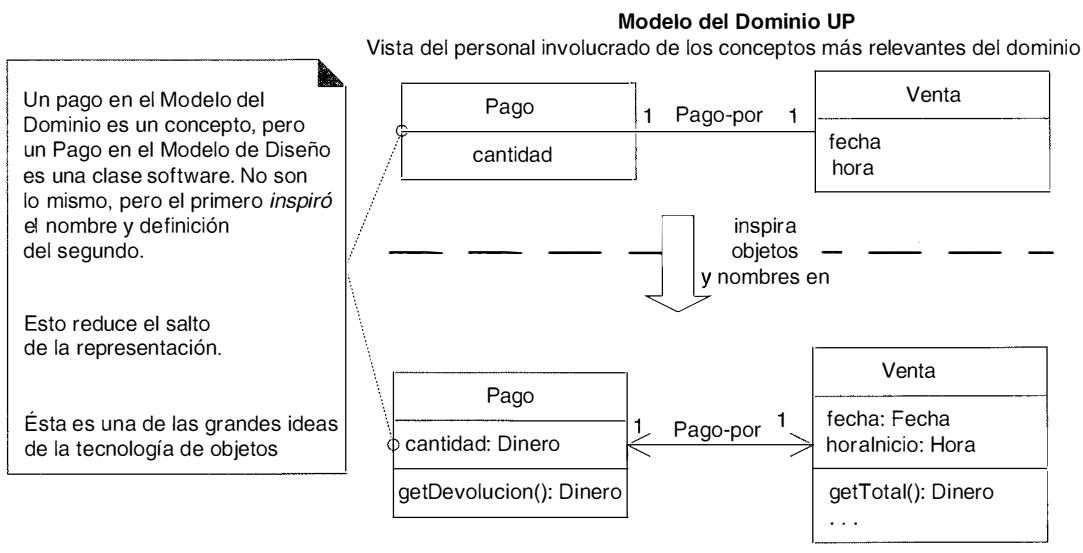
10.9. Reducción del salto en la representación

Por favor, mire atentamente la Figura 10.10. ¿Por qué los libros y educadores que presentan el diseño de objetos común sólo muestran el uso de clases software cuyos nombres reflejan el vocabulario del dominio? ¿Por qué eligen un nombre de clase software como *Venta*, y qué hace una *Venta*?

Simplemente, eligiendo nombres que reflejan el vocabulario del dominio (*Venta*) se favorece la rápida comprensión y proporciona una pista acerca de lo que se espera del trozo de código de la clase software *Venta*. Tenemos un modelo mental del dominio en cuestión (por ejemplo, una tienda que vende cosas). En el mundo real, sabemos que una venta tiene una fecha. En consecuencia, si creamos una clase Java llamada *Venta*, y le damos la responsabilidad de conocer sobre una venta real y su fecha, entonces la clase Java *Venta* se corresponde de alguna manera con nuestro modelo mental del dominio real; es decir, apela a nuestra “intuición” del dominio.

El Modelo del Dominio proporciona un diccionario visual del vocabulario y conceptos del dominio a partir de los cuales nos inspiramos para nombrar algunas cosas del diseño software.

Esto se relaciona con el tema del **salto de la representación** o salto semántico —el salto entre nuestro modelo mental del dominio y su representación en el software—.



Modelo de Diseño UP

El desarrollador orientado a objetos se ha inspirado en el dominio del mundo real al crear las clases software.

Por tanto, el salto de la representación entre el modo en el que el personal involucrado concibe el dominio, y su representación en el software, se ha reducido.

Figura 10.10. En el diseño y programación de objetos, es normal crear clases software cuyos nombres e información se inspira en el dominio del mundo real.

En un extremo, podríamos programar directamente la aplicación de PDV NuevaEra en código binario para invocar el conjunto de instrucciones del procesador. Entendemos que el salto en las representaciones es enorme, y existirá en el software un coste real —aunque difícil de cuantificar— con un salto en la representación tan grande, porque es difícil de entender y relacionar con el dominio del problema. Cercano al otro extremo del espectro, están las tecnologías de objetos que nos permiten partir el código en clases cuyos nombres reflejan el tipo de partición que percibimos en el dominio. En el mundo real percibimos una “parte” (o hecho) denominada venta, luego en el terreno del software tenemos una clase software denominada *Venta*. Esta estrecha correspondencia uno-a-uno entre el vocabulario del dominio y nuestro vocabulario del software y sus particiones reduce el salto de la representación. Esto acelera la comprensión del código existente (porque funciona de la manera que esperamos, conociendo el dominio) y sugiere formas “naturales” para extender el código que se corresponden análogamente con el dominio, o apelan a nuestras intuiciones del dominio. Diciéndolo sencillamente, el modelo software nos recuerda al modelo mental o conceptual, y funciona de manera predecible.

Los modelos software proporcionan una ventaja práctica al reducir el salto de la representación. La mayoría de los ingenieros del software saben que es verdad, incluso si es difícil de cuantificar. De hecho, una prueba de esto es que algunos de los programadores Java, de forma deliberada complican el código fuente para que sea difícil hacer ingeniería inversa a partir de los bytecodes, cambiando los nombres de las clases y los métodos Java de manera que sean ininteligibles y, por tanto, ya no evoquen nuestra visión del dominio, aunque las estructuras de control y datos no se cambian.

Por supuesto, la tecnología de objetos es valiosa también porque puede favorecer el diseño de sistemas elegantes, débilmente acoplados, que pueden crecer y ampliarse fácilmente, como veremos en lo que queda del libro. Una reducción del salto en la representación es útil, pero se puede sostener que es secundario a la ventaja que ofrecen los objetos de facilitar los cambios y extensiones, y el soporte que ofrecen para el manejo y ocultación de la complejidad.

10.10. Ejemplo: el Modelo del Dominio del PDV NuevaEra

La lista de clases conceptuales generadas para el dominio del PDV NuevaEra se podría representar gráficamente (ver Figura 10.11) para mostrar el comienzo del Modelo del Dominio.

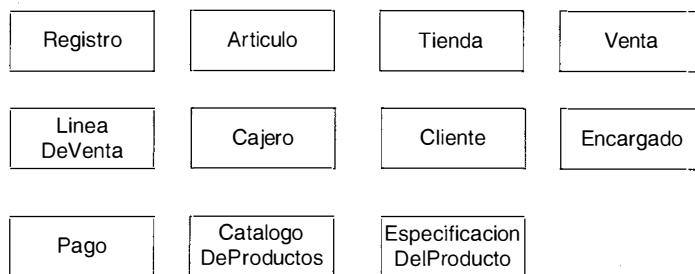


Figura 10.11. Modelo del Dominio inicial.

Las consideraciones acerca de los atributos y asociaciones del Modelo del Dominio se posponen a capítulos siguientes.

10.11. Modelos del Dominio en el UP

Como se sugiere en el ejemplo de la Tabla 10.2, un Modelo del Dominio, normalmente, se inicia y completa en la elaboración.

Inicio

Los modelos del dominio no se incentivan fuertemente en la fase de inicio, puesto que el propósito del inicio no es llevar a cabo un estudio serio, sino decidir si merece la pena un estudio más profundo en el proyecto, en una fase de elaboración.

Elaboración

El Modelo del Dominio se crea sobre todo durante las iteraciones de la elaboración, cuando la necesidad más importante es entender los conceptos relevantes y trasladar algunos a clases software durante el trabajo de diseño.

Tabla 10.2. Muestra de los artefactos UP y evolución temporal. c – comenzar; r - refinar.

Disciplina	Artefacto Iteración →	Inicio I1	Elab. E1...En	Const. C1...Cn	Trans. T1...T2
Modelado del Negocio	Modelo del Dominio		c		
Requisitos	Modelo de Casos de Uso Visión Especificación Complementaria Glosario	c c c c	r r r r		
Diseño	Modelo de Diseño Documento de Arquitectura SW Modelo de Datos		c c c	r r	
Implementación	Modelo de Implementación		c	r	r
Gestión del Proyecto	Plan de Desarrollo SW	c	r	r	r
Pruebas	Modelo de Pruebas		c	r	
Entorno	Marco de Desarrollo	c	r		

Aunque, irónicamente, se dedicarán un número significativo de páginas a explicar el modelado de los objetos del dominio, en manos experimentadas el desarrollo de un modelo del dominio (parcial, desarrollado incrementalmente) en cada iteración debería durar sólo unas pocas horas. Esto se acorta mediante el uso de patrones de análisis predefinidos.

El Modelo de Objetos del Negocio del UP vs. el Modelo del Dominio

El Modelo del Dominio del UP es una variación oficial del menos común Modelo de Objetos del Negocio del UP (BOM, *Business Object Model*). El BOM del UP —no confundir con el modo en el que otras personas o métodos podrían definir un BOM, que es un término ampliamente utilizado con significados diferentes— es un tipo de modelo de empresa utilizado para describir el negocio completo. Podría utilizarse al llevar a cabo la ingeniería o reingeniería de procesos del negocio, independiente de cualquier aplicación software (como el PDV NuevaEra). Citando textualmente:

[El BOM del UP] sirve como abstracción del modo en el que los trabajadores y las entidades del negocio necesitan relacionarse y cómo necesitan colaborar para llevar a cabo el negocio. [RUP]

El BOM se representa con varios diagramas diferentes (clase, actividad y secuencia) que muestran cómo funciona (o debería funcionar) toda la empresa. Es más útil si se realiza ingeniería de procesos de negocio por toda la empresa, pero ésta es una actividad menos común que la creación de una única aplicación software.

En consecuencia, el UP define el Modelo del Dominio como un artefacto subconjunto o una especialización del BOM, que se crea normalmente. Citando textualmente:

Puedes elegir desarrollar un modelo de objetos del negocio “incompleto”, enfocado a explicar “cosas” y productos importantes para un dominio. ... A menudo se hace referencia a éste como modelo del dominio. [RUP]

10.12. Lecturas adicionales

Object-Oriented Methods: A Foundation de Odell proporciona una sólida introducción al modelado del dominio conceptual. También resulta útil *Design Object Systems* de Cook and Daniel.

Analysis Patterns de Fowler ofrece importantes patrones de los modelos del dominio, y se recomienda sin ninguna duda. Otro libro bueno, que describe patrones de los modelos del dominio es *Data Model Patterns: Conventions of Thought* de Hay. Los consejos de los expertos en el modelado de datos, que entienden la distinción entre los modelos conceptuales puros y los modelos de los esquemas de bases de datos, pueden ser muy útiles para el modelado de los objetos del dominio.

Java Modeling in Color with UML [CDL99] contiene más consejos relevantes sobre el modelado del dominio de lo que sugiere el título. El autor identifica patrones comunes en los tipos relacionados y sus asociaciones; el aspecto de color es realmente una visualización de las categorías típicas de estos tipos, como *descripción* (azul), *roles* (amarillo), y *momento-intervalos* (rosa). El color se utiliza para ayudar a ver los patrones.

Desde el trabajo original de Abbot, el análisis lingüístico ha adquirido técnicas más sofisticadas para el análisis orientado a objetos, generalmente denominado modelado del lenguaje natural, o una variante. Véase [Moreno97] como un ejemplo.

10.13. Artefactos del UP

La influencia entre los artefactos, destacando el Modelo del Dominio, se muestra en la Figura 10.12.

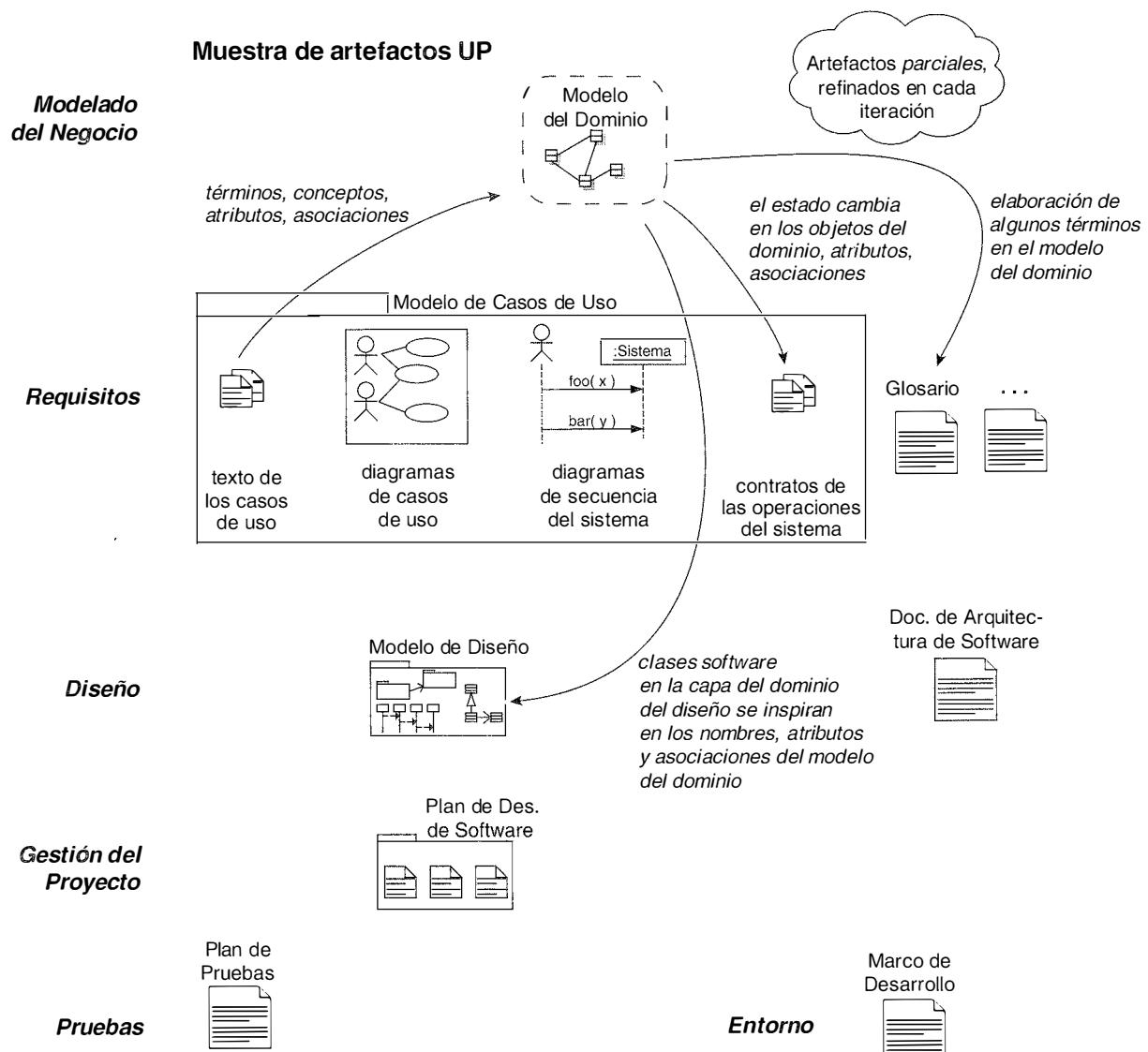


Figura 10.12. Muestra de la influencia entre los artefactos UP.

Capítulo 11

MODELO DEL DOMINIO: AÑADIR ASOCIACIONES

Objetivos

- Identificar las asociaciones del modelo del dominio.
 - Distinguir entre las asociaciones necesito-conocer y sólo-comprensión.
-

Introducción

Resulta útil identificar aquellas asociaciones entre clases conceptuales que son necesarias para satisfacer los requisitos de información de los escenarios actuales que se están desarrollando, y que ayudan a entender el modelo del dominio. Este capítulo explora la identificación de las asociaciones adecuadas, y añade las asociaciones al modelo del dominio del caso de estudio NuevaEra.

11.1. Asociaciones

Una **asociación** es una relación entre tipos (o más concretamente, instancias de estos tipos) que indica alguna conexión significativa e interesante (ver Figura 11.1).

En UML, las asociaciones se definen como “la relación semántica entre dos o más clasificadores que implica conexiones entre sus instancias”.

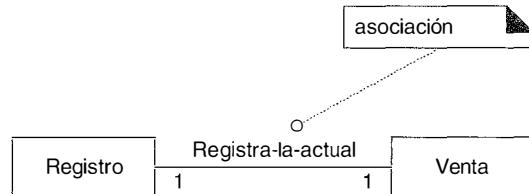


Figura 11.1. Asociaciones.

Criterio para asociaciones útiles

Las asociaciones que merece la pena registrar, normalmente, implican conocimiento de una relación que es necesario conservar durante algún tiempo —podría ser milisegundos o años, dependiendo del contexto—. En otras palabras, ¿entre qué objetos necesitamos mantener en memoria una relación? Por ejemplo, ¿necesitamos recordar qué instancias de *LíneaDeVenta* están asociadas con una instancia de *Venta*? Sin ninguna duda, de otra manera no sería posible reconstruir una venta, imprimir un recibo o calcular el total de la venta.

Considere la inclusión de las siguientes asociaciones en un modelo del dominio:

- Asociaciones de las que es necesario conservar el conocimiento de la relación durante algún tiempo (asociaciones “necesito-conocer”).
- Asociaciones derivadas de la Lista de Asociaciones Comunes.

Por el contrario, ¿necesitamos recordar una relación entre la *Venta* actual y un *Encargado*? No, los requisitos no dan a entender que se necesite ninguna relación de este tipo. No es incorrecto mostrar una relación entre una *Venta* y un *Encargado*, pero no es convincente o útil en el contexto de nuestros requisitos.

Éste es un punto importante. En un modelo del dominio con n clases del dominio diferentes, pueden existir $n(n-1)$ asociaciones entre diferentes clases conceptuales —un número potencialmente grande—. Muchas líneas en un diagrama añadirán “ruido visual” y lo hará menos comprensible. Por tanto, sea cuidadoso al añadir líneas de asociación. Utilice como criterio las guías que se sugieren en este capítulo.

11.2. Notación de las asociaciones en UML

Una asociación se representa como una línea entre clases con un nombre de asociación. La asociación es inherentemente bidireccional, lo que significa que desde las instancias de cualquiera de las dos clases, es posible el recorrido lógico hacia la otra.

Este recorrido es puramente abstracto; *no* se trata de una sentencia sobre conexiones entre entidades software.

Los extremos de la asociación podrían contener una expresión de multiplicidad que indica la relación numérica entre las instancias de las clases.

Una “flecha de dirección de lectura” opcional indica la dirección de la lectura del nombre de la asociación; **no** indica la dirección de la visibilidad o navegación.

Si no está presente, la convención es leer la asociación de izquierda a derecha o de arriba hacia abajo, aunque no es una regla de UML (ver Figura 11.2).

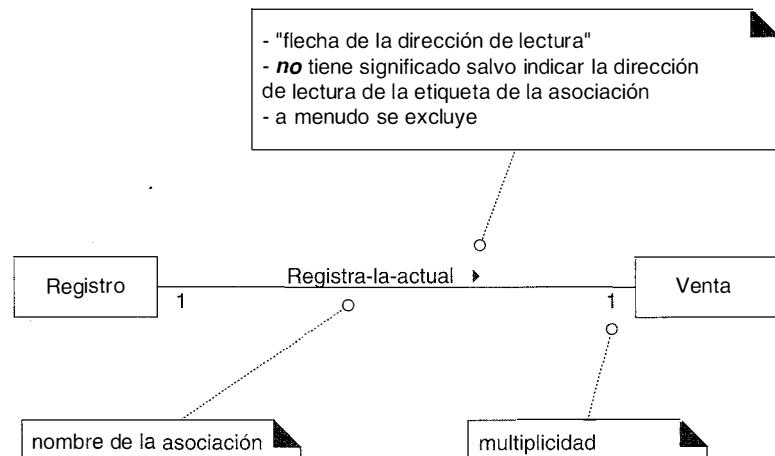


Figura 11.2. Notación UML para las asociaciones.

La flecha de dirección de lectura no tiene significado en términos del modelo; sólo es una ayuda para el lector del diagrama.

11.3. Localización de las asociaciones—lista de asociaciones comunes

Comience la inclusión de asociaciones utilizando la lista de la Tabla 11.1.

Contiene categorías comunes que, normalmente, merece la pena tener en cuenta. Los ejemplos se han extraído de los dominios de ventas y reservas de vuelos.

Tabla 11.1. Lista de asociaciones comunes.

Categoría	Ejemplos
A es una parte física de B	<i>Cajon-Registro</i> (o más concretamente, <i>TPDV</i>) <i>Ala-Avion</i>
A es una parte lógica de B	<i>LineaDeVenta-Venta</i> <i>EtapaVuelo-RutaVuelo</i>
A está contenido físicamente en B	<i>Registro-Tienda</i> , <i>Articulo-Estanteria</i> <i>Pasajero-Avion</i>
A está contenido lógicamente en B	<i>DescripcionDelArticulo-Catalogo</i> <i>Vuelo-PlanificacionVuelo</i>

continúa

Tabla 11.1. Lista de asociaciones comunes. (Continuación)

Categoría	Ejemplos
A es una descripción de B	<i>DescripcionDelArticulo-Articulo</i> <i>DescripcionDelVuelo-Vuelo</i>
A es un línea de una transacción o informe de B	<i>LíneaDeVenta-Venta</i> <i>TrabajoMantenimiento-RegistroDe-Mantenimiento</i>
A se conoce/registra/recoge/informa/captura en B	<i>Venta-Registro</i> <i>Reserva-ListaPasajeros</i>
A es miembro de B	<i>Cajero-Tienda</i> <i>Piloto-CompañíaAerea</i>
A es una subunidad organizativa de B	<i>Departamento-Tienda</i> <i>Mantenimiento-CompañíaAerea</i>
A utiliza o gestiona B	<i>Cajero-Registro</i> <i>Piloto-Avion</i>
A se comunica con B	<i>Cliente-Cajero</i> <i>AgenteDeReservas-Pasajero</i>
A está relacionado con una transacción B	<i>Cliente-Pago</i> <i>Pasajero-Billete</i>
A es una transacción relacionada con otra transacción B	<i>Pago-Venta</i> <i>Reserva-Cancelacion</i>
A está al lado de B	<i>LíneaDeVenta-LíneaDeVenta</i> <i>Ciudad-Ciudad</i>
A es propiedad de B	<i>Registro-Tienda</i> <i>Avion-CompañíaAerea</i>
A es un evento relacionado con B	<i>Venta-Cliente</i> , <i>Venta-Tienda</i> <i>Salida-Vuelo</i>

Asociaciones de prioridad alta

A continuación, presentamos algunas de las categorías de asociaciones de prioridad alta que son, invariablemente, útiles incluirlas en un modelo del dominio:

- A es una *parte lógica o física* de B.
- A está *contenida física o lógicamente* en B.
- A se *registra en* B.

11.4. Guías para las asociaciones

- Céntrese en aquellas asociaciones para las que se necesita conservar el conocimiento de la relación durante algún tiempo (asociaciones “necesito-conocer”).
 - Es más importante identificar *clases conceptuales* que identificar asociaciones.

- Demasiadas asociaciones tienden a confundir un modelo del dominio en lugar de aclararlo. Su descubrimiento puede llevar tiempo, con beneficio marginal.
- Evite mostrar asociaciones redundantes o derivadas.

11.5. Roles

Cada extremo de una asociación se denomina **rol**. Los roles pueden tener opcionalmente:

- Nombre.
- Expresión de multiplicidad.
- Navegabilidad.

La multiplicidad se presenta a continuación, y las otras dos características se discutirán en capítulos posteriores.

Multiplicidad

La **multiplicidad** define cuántas instancias de una clase A pueden asociarse con una instancia de una clase B (ver Figura 11.3).

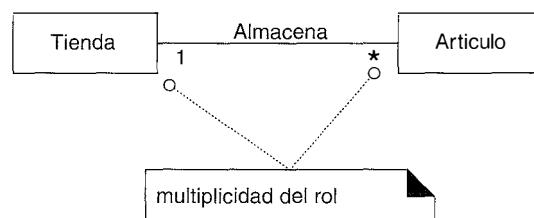


Figura 11.3. Multiplicidad de una asociación.

Por ejemplo, una instancia individual de una *Tienda* puede asociarse con “muchas” (cero o más, indicado por el “*”) instancias de *Articulo*.

En la Figura 11.4 se muestran algunos ejemplos de expresiones de multiplicidad.

El valor de la multiplicidad indica cuántas instancias se puede asociar legalmente con otra, en un momento concreto, en lugar de a lo largo de un periodo de tiempo. Por ejemplo, es posible que un coche usado pudiera a lo largo del tiempo ser vendido repetidamente a comerciantes de coches usados. Pero en un momento concreto, el coche sólo es *Abastecido-por* un comerciante. El coche no es *Abastecido-por* muchos comerciantes en un momento concreto. Análogamente, en países con leyes monógamas, una persona puede estar *Casado-con* sólo una persona en un momento dado, aunque a lo largo del tiempo, puedan casarse con muchas personas.

El valor de la multiplicidad depende de nuestros intereses como modeladores y desarrolladores de software, porque pone de manifiesto una restricción de diseño que será (o podrá ser) reflejada en el software. La Figura 11.5 muestra un ejemplo y explicación.

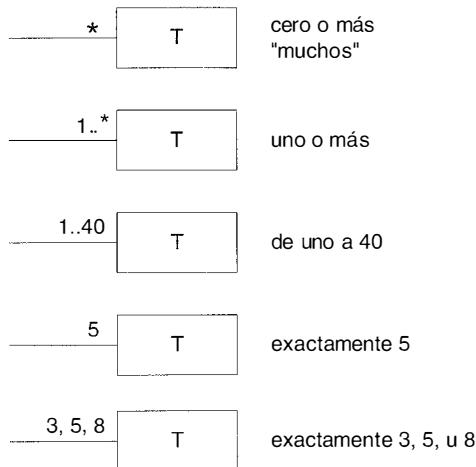


Figura 11.4. Valores de la multiplicidad.

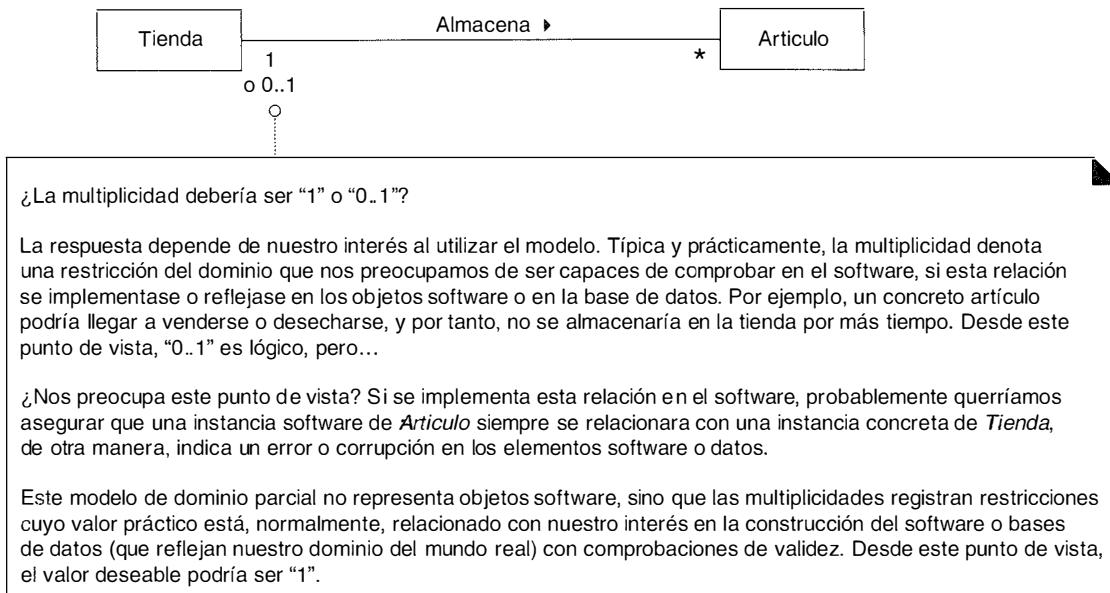


Figura 11.5. La multiplicidad es dependiente del contexto.

Rumbaugh proporciona otro ejemplo de *Persona* y *Compañía* en la asociación *Trabaja-para* [Rumbaugh91]. El que indiquemos si una instancia de *Persona* trabaja para una o varias instancias de *Compañía*, es dependiente del contexto del modelo; el departamento de impuestos está interesado en *muchas*; un sindicato probablemente sólo en *una*. Normalmente, la elección, prácticamente, depende de para quién estamos construyendo el software, y de esta manera, las multiplicidades válidas de una implementación.

11.6. ¿Cómo de detalladas deben ser las asociaciones?

Las asociaciones son importantes, pero un error típico al crear los modelos del dominio, es dedicar demasiado tiempo durante el estudio intentando descubrirlas.

Es fundamental entender lo siguiente:

Es más importante encontrar las *clases conceptuales* que las asociaciones. La mayoría del tiempo dedicado a la creación del modelo del dominio debería emplearse en la identificación de las clases conceptuales, no de las asociaciones.

11.7. Asignación de nombres a las asociaciones

Nombre una asociación en base al formato *NombreTipo-FraseVerbal-NombreTipo* donde la frase verbal crea una secuencia que es legible y tiene significado en el contexto del modelo.

Los nombres de las asociaciones deben comenzar con una letra mayúscula, puesto que una asociación representa un clasificador de enlace entre las instancias; en UML, los clasificadores deben comenzar con una letra mayúscula. Dos formatos típicos e igualmente válidos para un nombre de asociación compuesta son:

- *Pagado-mediante.*
- *PagadoMediante.*

En la Figura 11.6, la dirección por defecto para la lectura de los nombres de las asociaciones es de izquierda a derecha o de arriba abajo. No se corresponde con la dirección por defecto en UML, sino que se trata de una convención común.

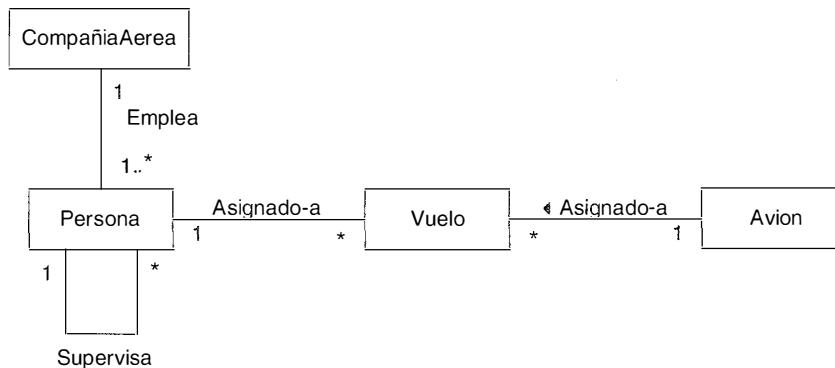
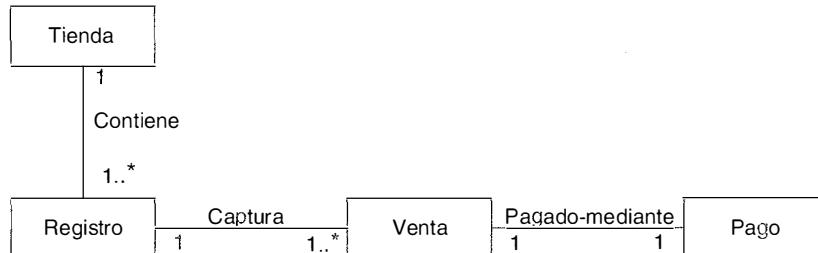


Figura 11.6. Nombres de asociaciones.

11.8. Múltiples asociaciones entre dos tipos

Dos tipos podrían tener múltiples asociaciones entre ellos; no es extraño. No existe ningún ejemplo destacado en nuestro caso de estudio del PDV, pero un ejemplo del dominio de la compañía aérea podría ser las relaciones entre un *Vuelo* (o quizás de manera más precisa, una *EtapaVuelo*) y un *Aeropuerto* (ver Figura 11.7); las asociaciones *vuela-a* y *vuela-desde* son relaciones claramente diferentes, que se deberían mostrar de manera separada.

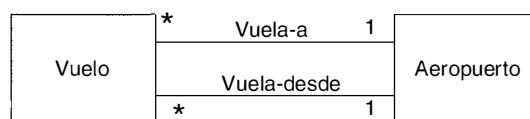


Figura 11.7. Múltiples asociaciones.

11.9. Asociaciones e implementación

Durante el modelado del dominio, una asociación *no* es una declaración sobre el flujo de datos, variables de instancia o conexiones entre objetos en una solución software; es una manifestación de que una relación es significativa en un sentido puramente conceptual —en el mundo real—. Desde un punto de vista práctico, muchas de estas relaciones se implementarán generalmente en software como caminos de navegación y visibilidad (tanto en el Modelo de Diseño como en el Modelo de Datos), pero su presencia en una vista conceptual (o esencial) de un modelo de dominio no requiere su implementación.

Al crear un modelo de dominio, podríamos definir asociaciones que no son necesarias durante la implementación. A la inversa, podríamos descubrir asociaciones que necesitan implementarse pero que se obviaron durante el modelado del dominio. En estos casos, el modelo del dominio puede actualizarse para reflejar estos descubrimientos.

Sugerencia

¿Deberían actualizarse los anteriores modelos estudiados, como el modelo del dominio, con las apreciaciones descubiertas durante el trabajo de implementación? No se moleste a menos que haya algún uso práctico del modelo en el futuro. Si es únicamente (como es el caso algunas veces) un artefacto temporal que se utiliza para inspirar las etapas siguientes, y no se utilizará de manera significativa más adelante, ¿por qué actualizarlo? Evite hacer o actualizar cualquier modelo o documentación a menos que exista una justificación concreta para su futura utilización.

Posteriormente estudiaremos formas de implementar la asociaciones en un lenguaje de programación orientado a objetos (los más habitual es utilizar un atributo que referencia una instancia de la clase asociada), pero de momento, es conveniente pensar en ellas como expresiones puramente conceptuales, *no* declaraciones sobre la solución de base de datos o software. Como siempre, al posponer las consideraciones de diseño se nos libera de informaciones y decisiones extrañas mientras hacemos un estudio de “análisis” puro y maximiza nuestras opciones de diseño más adelante.

11.10. Asociaciones del Modelo del Dominio del PDV NuevaEra

Ahora podemos añadir las asociaciones a nuestro modelo del dominio del PDV. Debemos añadir aquellas asociaciones que los requisitos (por ejemplo, los casos de uso) sugieren o implican una necesidad de recordar, o que, de otra manera, recomienda fuertemente nuestra percepción del dominio del problema. Cuando abordamos un nuevo problema, las categorías comunes de asociaciones, presentadas anteriormente, deberían revisarse y tenerse en cuenta, puesto que representan muchas de las asociaciones relevantes que normalmente necesitan recogerse.

Relaciones evidentes de la Tienda

La siguiente muestra de asociaciones es justificable en términos de necesito-conocer. Se basa en los casos de uso que se están considerando actualmente.

<i>Registro Registra Venta</i>	Para conocer la venta actual, generar el total, imprimir recibo
<i>Venta Pagada-mediante Pago</i>	Para conocer si se ha pagado la venta, relacionar la cantidad entregada con el total de la venta, e imprimir un recibo
<i>CatalogoDeProductos Registra EspecificacionDelProducto</i>	Para recuperar una <i>EspecificacionDelProducto</i> a partir de un articuloID

Aplicación de la lista de comprobación de categorías de asociaciones

Recorreremos la lista de comprobación, basada en los tipos identificados previamente, considerando los requisitos del caso de uso actual.

Categoría	Sistema
A es una parte física de B	<i>Registro—Caja</i>
A es una parte lógica de B	<i>LineaDeVenta—Venta</i>
A está contenido físicamente en B	<i>Registro—Tienda</i> <i>Articulo—Tienda</i>
A está contenido lógicamente en B	<i>EspecificacionDelProducto—CatalogoDeProductos</i> <i>CatalogoDeProductos—Tienda</i>
A es una descripción de B	<i>EspecificacionDelProducto—Articulo</i>

continúa

Categoría	Sistema
A es una línea de una transacción o informe de B	<i>LineaDeVenta—Venta</i>
A se conoce/registra/recoge/informa/captura en B	<i>(completa)Venta—Tienda</i> <i>(actual)Venta—Registro</i>
A es miembro de B	<i>Cajero—Tienda</i>
A es una subunidad organizativa de B	<i>no aplicable</i>
A utiliza o gestiona B	<i>Cajero—Registro</i> <i>Encargado—Registro</i> <i>Encargado—Cajero, aunque probablemente no es aplicable</i>
A se comunica con B	<i>Cliente—Cajero</i>
A está relacionado con una transacción B	<i>Cliente—Pago</i> <i>Cajero—Pago</i>
A es una transacción relacionada con otra transacción B	<i>Pago—Venta</i>
A está al lado de B	<i>LineaDeVenta—LineaDeVenta</i>
A es propiedad de B o informe de B	<i>Registro—Tienda</i>

11.11. Modelo del Dominio del PDV NuevaEra

El modelo del dominio de la Figura 11.8 muestra un conjunto de clases conceptuales y asociaciones que son candidatas para nuestra aplicación de PDV. Las asociaciones se derivaron sobre todo a partir de la lista de comprobación de asociaciones candidatas.

¿Conservamos sólo las asociaciones necesito-conocer?

El conjunto de asociaciones que se muestran en el modelo del dominio de la Figura 11.8 se derivaron de forma mecánica, la mayor parte, a partir de la lista de comprobación de asociaciones. Sin embargo, sería deseable que fuéramos más cuidadosos con las asociaciones que incluimos en nuestro modelo del dominio. Vista como una herramienta de comunicación, no es deseable sobrecargar el modelo del dominio con asociaciones que no se necesitan forzosamente y que no favorecen nuestra comprensión. Demasiadas asociaciones no imprescindibles oscurecen en lugar de aclarar.

Como se sugirió previamente, se recomienda el siguiente criterio para mostrar las asociaciones:

- Céntrese en aquellas asociaciones para las que el conocimiento de la relación necesita conservarse durante algún tiempo (asociaciones “necesito-conocer”).
- Evite mostrar asociaciones redundantes o derivadas.

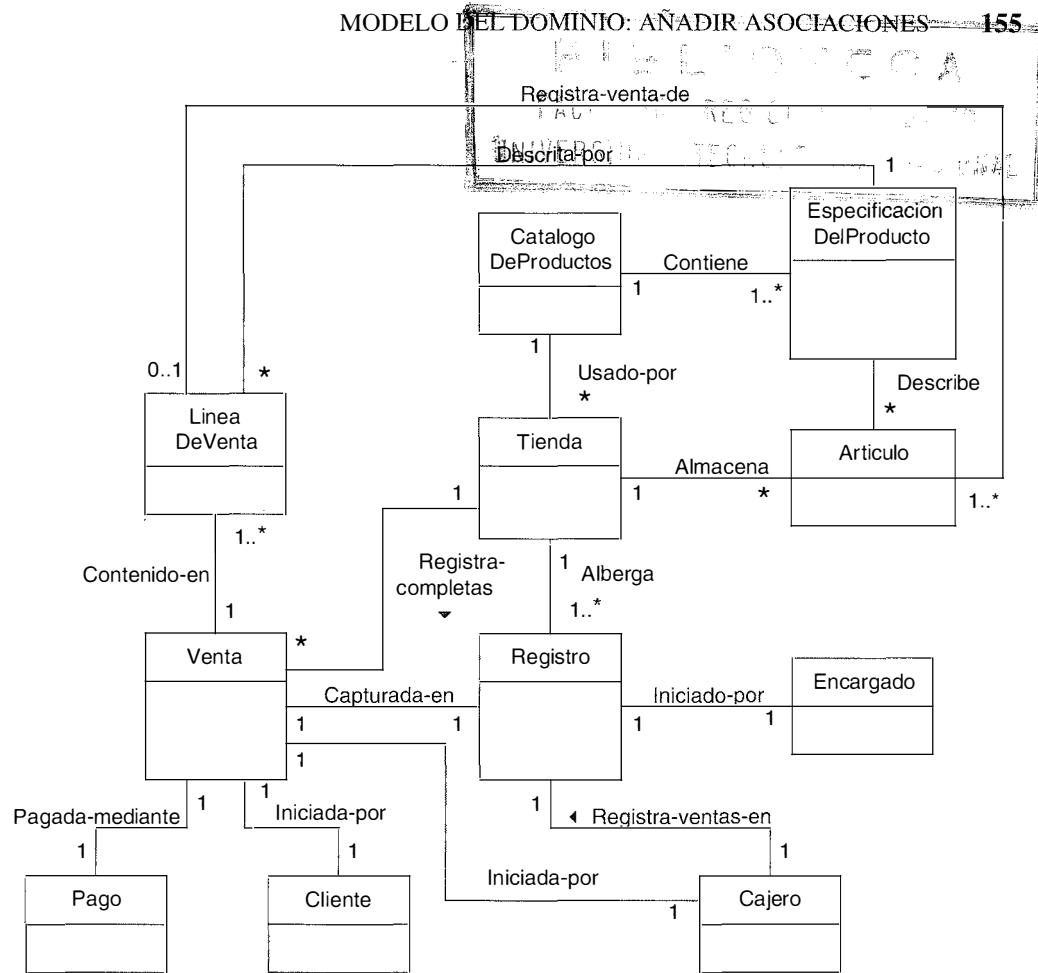


Figura 11.8. Modelo del dominio parcial.

En base a este consejo, no todas las asociaciones que se acaban de mostrar son imprescindibles. Considere lo siguiente:

Asociación	Comentario
<i>Venta Insertada-por Cajero</i>	Los requisitos no indican que necesito-conocer o registrar al cajero actual. Además, puede derivarse si está presente la asociación <i>Registro Usado-por Cajero</i> .
<i>Registro Usado-por Cajero</i>	Los requisitos no indican que necesito-conocer o registrar al cajero actual.
<i>Registro Iniciado-por Encargado</i>	Los requisitos no indican que necesito-conocer o registrar al encargado que pone en marcha un <i>Registro</i> .
<i>Venta Iniciada-por Cliente</i>	Los requisitos no indican que necesito-conocer o registrar al cliente que inicia una venta.
<i>Tienda Almacena Articulo</i>	Los requisitos no indican que necesito-conocer o mantener información del inventario.
<i>LineaDeVenta Registra-venta-de Articulo</i>	Los requisitos no indican que necesito-conocer o mantener información del inventario.

Nótese que la capacidad de justificar una asociación en función de necesito-conocer depende de los requisitos; obviamente, un cambio en éstos —como que se necesite mostrar en el recibo el ID del cajero— cambia la necesidad de recordar una relación.

Basado en el análisis anterior, *podría* justificarse la eliminación de las asociaciones en cuestión.

Asociaciones necesito-conocer vs. comprensión

Un criterio necesito-conocer estricto para el mantenimiento de las asociaciones generará un “modelo de información” mínimo de lo que se necesita para modelar el dominio del problema —limitado por los requisitos actuales que se están considerando—. Sin embargo, este enfoque podría crear un modelo que no transmite (a nosotros o a los demás) una comprensión completa del dominio.

Además de ser un modelo necesito-conocer de información sobre las cosas, el modelo del dominio es una herramienta de comunicación con la que estamos intentando entender y comunicar a otros los conceptos importantes y sus relaciones. Desde este punto de vista, eliminando algunas asociaciones que no se exigen estrictamente en una base necesito-conocer, puede crear un modelo sin interés —no comunica las ideas claves y relaciones—.

Por ejemplo, en la aplicación del PDV: aunque, tomando como base las relaciones necesito-conocer de manera estricta, podría no ser necesario registrar *Venta Iniciada-por Cliente*, su ausencia deja fuera un aspecto importante para entender el dominio —que un cliente genera las ventas—.

En cuanto a las asociaciones, un buen modelo se sitúa en alguna parte entre un modelo necesito-conocer mínimo y uno que ilustra cada relación concebible. ¿El criterio básico para juzgar su valor? —¿Satisface todos los requisitos necesito-conocer y además comunica claramente un conocimiento esencial de los conceptos importantes en el dominio del problema?—.

Céntrese en las asociaciones necesito-conocer, pero contemple las asociaciones de sólo-comprensión para enriquecer el conocimiento básico del dominio.

Capítulo 12

MODELO DEL DOMINIO: AÑADIR ATRIBUTOS

Cualquier error tardío es indistinguible de una característica.

Rich Kulawiec

Objetivos

- Identificar los atributos del modelo del dominio.
 - Distinguir entre atributos correctos e incorrectos.
-

Introducción

Resulta útil identificar aquellos atributos de las clases conceptuales que se necesitan para satisfacer los requisitos de información de los actuales escenarios en estudio. Este capítulo explora la identificación de los atributos adecuados, y añade los atributos al modelo del dominio de NuevaEra.

12.1. Atributos

Un **atributo** es un valor de datos lógico de un objeto.

Incluya los siguientes atributos en un modelo del dominio: aquellos para los que los requisitos (por ejemplo, los casos de uso) sugieren o implican una necesidad de registrar la información.

Por ejemplo, un recibo (que recoge la información de una venta) normalmente incluye una fecha y una hora, y la dirección quiere conocer las fechas y horas de las ventas por múltiples motivos. En consecuencia, la clase conceptual *Venta* necesita los atributos *fecha* y *hora*.

12.2. Notación de los atributos en UML

Los atributos se muestran en el segundo compartimento del rectángulo de la clase (ver Figura 12.1). Sus tipos podrían mostrarse opcionalmente.

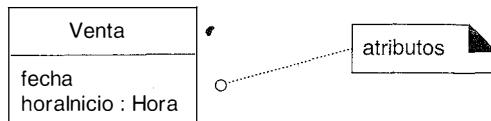


Figura 12.1. Clases y atributos.

12.3. Tipos de atributos válidos

Hay algunas cosas que no deberían representarse como atributos, sino como asociaciones. Esta sección presenta los tipos válidos.

Mantenga atributos simples

Intuitivamente, la mayoría de los atributos simples son los que, a menudo, se conocen como tipos de datos primitivos, como los números. El tipo de un atributo, normalmente, no debería ser un concepto de dominio complejo, como *Venta* o *Aeropuerto*. Por ejemplo, el siguiente atributo *registroActual* en la clase *Cajero* de la Figura 12.2, no es deseable porque su tipo tiene la intención de ser un *Registro*, que no es un tipo de atributo simple (como *Numero* o *String*). La manera más útil para expresar que un *Cajero* utiliza un *Registro* es con una asociación, no con un atributo.

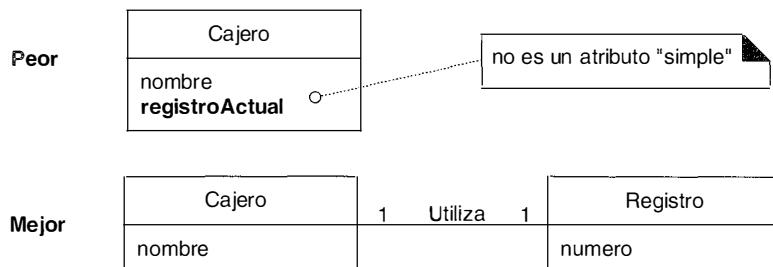


Figura 12.2. Relaciones con asociaciones, no atributos.

Los atributos en un modelo del dominio deberían ser, preferiblemente, **atributos simples o tipos de datos**.

Los tipos de datos de los atributos muy comunes incluyen: *Boolean*, *Fecha*, *Numero*, *String (Texto)*, *Hora*.

Otros tipos comunes comprenden: *Direccion, Color, Geometrico (Punto, Rectangulo), Numero de Telefono, Numero de la Seguridad Social, Codigo de Producto Universal (UPC; Universal Product Code), SKU, ZIP o códigos postales, tipos enumerados.*

Repitiendo un ejemplo previo, un error típico es modelar un concepto del dominio complejo como un atributo. Para ilustrarlo, un aeropuerto de destino no es realmente una cadena de texto; se trata de una cosa compleja que ocupa muchos kilómetros cuadrados de espacio. Por tanto, *Vuelo* debería relacionarse con *Aeropuerto* mediante una asociación, no con un atributo, como se muestra en la Figura 12.3.

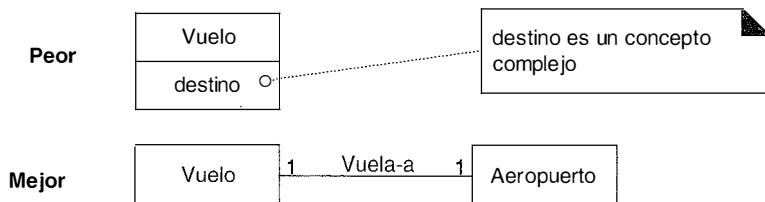


Figura 12.3. Evite la representación de conceptos del dominio complejos como atributos; utilice asociaciones.

Relacione las clases conceptuales con una asociación, no con un atributo.

Perspectiva conceptual vs. implementación: ¿qué sucede con los atributos en el código?

La restricción de que el tipo de los atributos en el modelo del dominio sea sólo un tipo de datos simple *no* implica que los atributos C++ o Java (miembros de datos, campos de una instancia) sólo deban ser de tipos de datos primitivos, o simples. El modelo del dominio se centra en declaraciones conceptuales puras sobre un dominio del problema, no en componentes software.

Posteriormente, durante el trabajo de diseño e implementación, se verá que las asociaciones entre objetos representadas en el modelo del dominio, a menudo, se implementarán como atributos que referencian a otros objetos software complejos. Sin embargo, ésta es sólo una de las posibles soluciones de diseño para implementar una asociación, y, de ahí que la decisión se deba posponer durante el modelado del dominio.

Tipos de datos

Los atributos deben ser, generalmente, **tipos de datos**. Esto es un término UML que implica un conjunto de valores para los cuales no es significativa una identidad única (en el contexto de nuestro modelo o sistema) [RJB99]. Por ejemplo, no es (normalmente) significativo distinguir entre:

- Diferentes instancias del *Numero 5*.
- Diferentes instancias del *String ‘gato’*.

- Diferentes instancias del *NumeroDeTelefono* que contiene el mismo número.
- Diferentes instancias de la *Direccion* que contiene la misma dirección.

Por el contrario, es significativo distinguir (por identidad) entre dos instancias de *Persona* cuyos nombres son, en los dos casos, “Luis García”, puesto que las dos instancias pueden representar individuos diferentes con el mismo nombre.

En cuanto al software, existen pocas situaciones donde uno compararía las direcciones de memoria de las instancias de *Numero*, *String*, *NumeroDeTelefono* o *Direccion*; sólo son relevantes las comparaciones basadas en los valores. Por el contrario, es comprensible comparar las direcciones de memoria de las instancias de *Persona*, y distinguirlas, incluso si tienen los mismos valores de los atributos, porque es importante su identidad única.

Por tanto, todos los tipos primitivos (número, string) son tipos de datos UML, pero no todos los tipos de datos son primitivos. Por ejemplo, *NumeroDeTelefono* es un tipo de dato no primitivo.

Estos valores de tipos de datos también se conocen como **objetos valor**.

La noción de tipos de datos puede ser útil. Como regla empírica, sea fiel a la prueba básica de tipos de atributos “simples”: hágalo un atributo si se considera de manera natural como un número, string, booleano, fecha u hora (etcétera); en otro caso, representelo como una clase conceptual aparte.

En caso de duda, defina algo como una clase conceptual aparte en lugar de como un atributo.

12.4. Clases de tipos de datos no primitivos

El tipo de un atributo podría representarse como una clase no primitiva por derecho propio en un modelo del dominio. Por ejemplo, en el sistema de PDV hay un identificador de artículo. Generalmente se ve simplemente como un número. Así que, ¿se debería representar como una clase no primitiva? Aplique esta guía:

Represente lo que podría considerarse, inicialmente, como un tipo de dato primitivo (como un número o string) como una clase no primitiva si:

- Está compuesto de secciones separadas.
 - Número de teléfono, nombre de persona.
- Habitualmente, hay operaciones asociadas con él, como análisis sintáctico o validación
 - Número de la seguridad social.
- Tiene otros atributos.
 - el precio de promoción podría tener una fecha (efectiva) de comienzo y fin.
- Es una cantidad con una unidad.

- La cantidad del pago tiene una unidad monetaria.
- Es una abstracción de uno o más tipos con alguna de estas cualidades.
- El identificador del artículo en el dominio de ventas es una generalización de tipos como el Código de Producto Universal (UPC) o el Número de Artículo Europeo (EAN)

Aplicando estas guías a los atributos del modelo del dominio del PDV llegamos al siguiente análisis:

- El identificador del artículo es una abstracción de varios esquemas de codificación comunes, incluyendo UPC-A, UPC-E y la familia de esquemas EAN. Estos esquemas de codificación numéricos tienen subpartes que identifican al fabricante, producto, país (para EAN), y un dígito de control para validarlos. Por tanto, debería existir una clase no primitiva *ArticuloID*, puesto que satisface muchas de las guías anteriores.
- Los atributos del *precio* y *cantidad* deberían ser clases no primitivas *Cantidad* o *Moneda* porque se trata de cantidades en una unidad monetaria.
- El atributo de la *direccion* debe ser una clase no primitiva *Direccion* porque tiene secciones diferentes.

Las clases *ArticuloID*, *Direccion* y *Cantidad* son tipos de datos (no es significativa la identidad única de las instancias) pero merece la pena considerarlas como clases independientes debido a sus cualidades.

¿Dónde representamos las clases de tipos de datos?

¿Debería mostrarse la clase *ArticuloID* como una clase conceptual independiente en el modelo del dominio? Depende de lo que quiera resaltar en el diagrama. Puesto que *ArticuloID* es un *tipo de datos* (no es importante la identidad única de las instancias), se podría mostrar en el comportamiento de los atributos del rectángulo de la clase, como se muestra en la Figura 12.4. Pero, puesto que es una clase no primitiva, con sus propios atributos y asociaciones, podría ser interesante mostrarla como una clase conceptual en su propio rectángulo. No existe una respuesta correcta; depende de cómo se esté utilizando el modelo del dominio como herramienta de comunicación, y la importancia de los conceptos en el dominio.

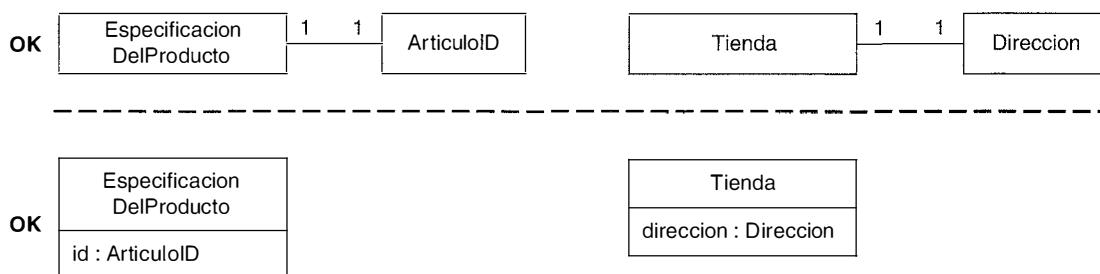


Figura 12.4. Si la clase del atributo es un tipo de datos, podría mostrarse en el rectángulo del atributo.

Un modelo del dominio es una herramienta de comunicación; las elecciones sobre lo que se muestra deben hacerse con esa consideración en mente.

12.5. Deslizarse al diseño: ningún atributo como clave ajena

No se deberían utilizar los atributos para relacionar las clases conceptuales en el modelo del dominio. La violación más típica de este principio es añadir un tipo de **atributo de clave ajena**, como se hace normalmente en el diseño de bases de datos relacionales, para asociar dos tipos. Por ejemplo, en la Figura 12.5, no es deseable el atributo *numeroRegistroActual* en la clase *Cajero* porque el propósito es relacionar el *Cajero* con un objeto *Registro*. La mejor manera de expresar que un *Cajero* utiliza un *Registro* es con una asociación, no con un atributo de clave ajena. Una vez más, relacione los tipos con una asociación, no con un atributo.

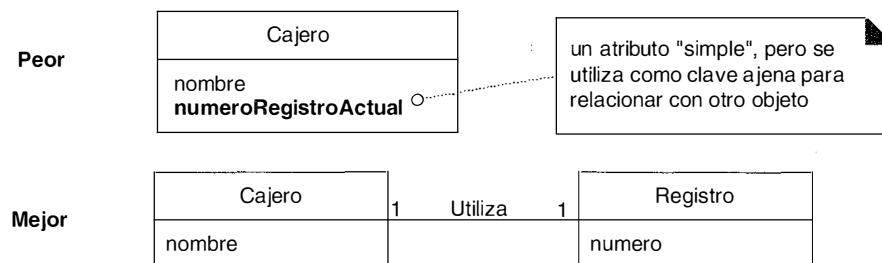


Figura 12.5. No utilice atributos como claves ajenas.

Hay muchas formas de relacionar objetos —siendo las claves ajenas una de ellas— y pospondremos el modo de implementar la relación hasta el diseño, para evitar el **deslizamiento al diseño**.

12.6. Modelado de cantidades y unidades de los atributos

La mayoría de las cantidades numéricas no deberían representarse simplemente como números. Considere el precio o la velocidad. Son cantidades con unidades asociadas, y es habitual que se necesite conocer la unidad y dar soporte a las conversiones. El software del PDV NuevaEra es para un mercado internacional y necesita soportar los precios en diferentes monedas. En el caso general, la solución consiste en representar la *Cantidad* como una clase conceptual aparte, con una *Unidad* asociada [Fowler96]. Puesto que las cantidades se consideran tipos de datos (no es importante la identidad única de las instancias), es aceptable recoger su representación en la sección de atributos del rectángulo de clase (ver Figura 12.6). También es común mostrar especializaciones de *Cantidad*. El *Dinero* es una clase de *Cantidad* cuyas unidades son monedas. El *Peso* es una cantidad cuyas unidades son kilogramos o libras.

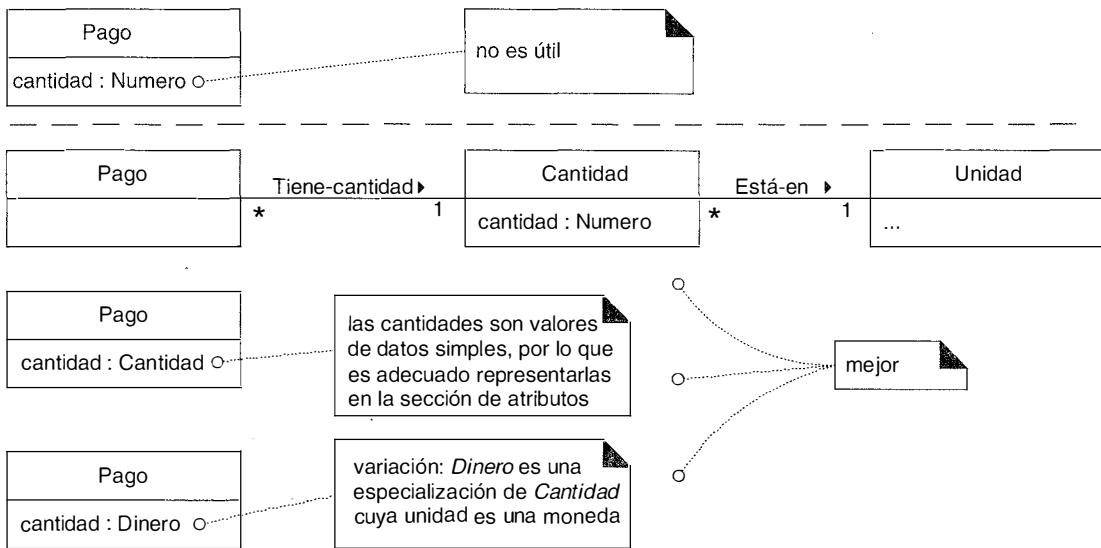


Figura 12.6. Modelado de cantidades.

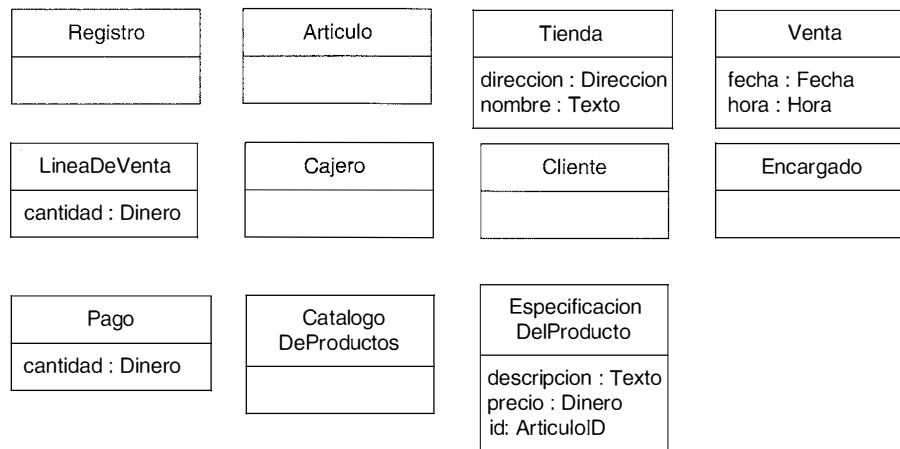
12.7. Atributos en el Modelo del Dominio de NuevaEra

Los atributos elegidos reflejan los requisitos de esta iteración —los escenarios de *Procesar Venta* de esta iteración—.

<i>Pago</i>	<i>cantidad</i> : Se debe capturar una cantidad (también conocida como “cantidad entregada”) para determinar si se proporciona el pago suficiente y calcular el cambio.
<i>EspecificacionDelProducto</i>	<i>descripcion</i> : Para mostrar la descripción en una pantalla o recibo. <i>id</i> : Para buscar una <i>EspecificacionDelProducto</i> , dado un artículoID introducido, es necesario relacionarlas con un <i>id</i> . <i>precio</i> : Para calcular el total de la venta y mostrar el precio de la línea de venta.
<i>Venta</i>	<i>fecha, hora</i> : Un recibo es un informe en papel de una venta. Normalmente muestra la fecha y la hora de la venta.
<i>LíneaDeVenta</i>	<i>cantidad</i> : Para registrar la cantidad introducida, cuando hay más de un artículo en una línea de venta (por ejemplo, <i>cinco</i> paquetes de tofu).
<i>Tienda</i>	<i>direccion, nombre</i> : El recibo requiere el nombre y la dirección de la tienda.

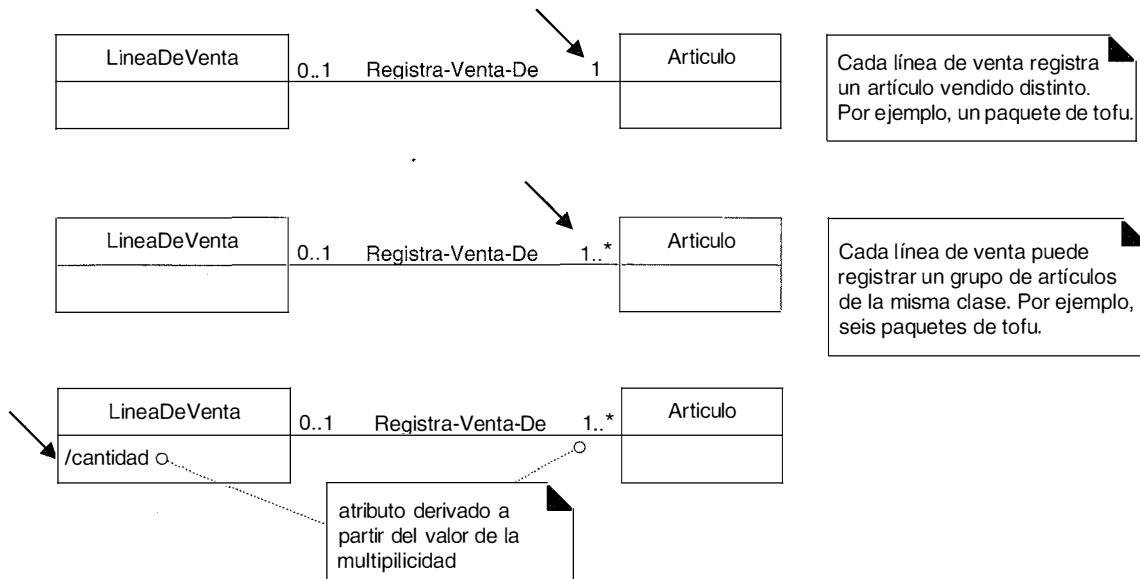
12.8. Multiplicidad de la LíneaDeVenta al Artículo

Es posible que el cajero reciba un grupo de artículos iguales (por ejemplo, seis paquetes de tofu), introduzca el *articuloID* una vez, y después introduzca una cantidad (por

**Figura 12.7.** Modelo del dominio que muestra los atributos.

ejemplo, seis). En consecuencia, una *LíneaDeVenta* individual se puede asociar con más de una instancia de un artículo.

La cantidad introducida por el cajero podría registrarse como un atributo de la *LíneaDeVenta* (Figura 12.8). Sin embargo, la cantidad se puede calcular a partir del valor de la multiplicidad actual de la relación, de ahí que pudiera caracterizarse como un **atributo derivado** —uno que puede derivarse a partir de otra información—. En UML, un atributo derivado se indica con el símbolo “/”.

**Figura 12.8.** Registro de la cantidad de artículos vendidos en una línea de venta.

12.9. Conclusión del Modelo del Dominio

La combinación de las clases conceptuales, asociaciones y atributos, descubiertos en el estudio anterior, da lugar al modelo que se presenta en la Figura 12.9.

Se ha creado un modelo del dominio, relativamente útil, para el dominio de una aplicación de PDV. No existe un único modelo correcto. Todos los modelos son aproximaciones del dominio que estamos intentando entender. Un buen modelo del dominio captura las abstracciones y la información esenciales necesarias para entender el dominio en el contexto de los requisitos actuales, y ayuda a la gente a entender el dominio —sus conceptos, terminología y relaciones—.

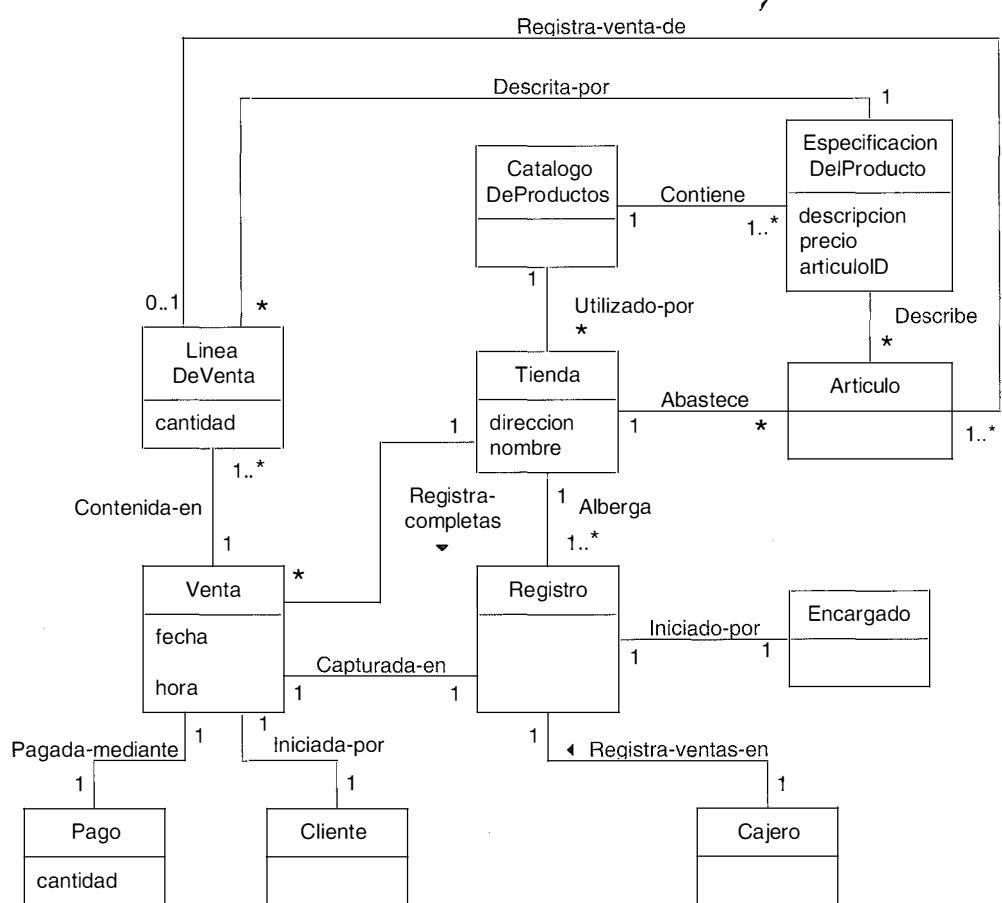


Figura 12.9. Un modelo del dominio parcial.

Capítulo 13

MODELO DE CASOS DE USO: AÑADIR DETALLES CON LOS CONTRATOS DE LAS OPERACIONES

Rápido, barato, bueno: elija dos cualesquieras.

Anónimo

Objetivos

- Crear los contratos para las operaciones del sistema.
-

Introducción

Los contratos de las operaciones pueden ayudar a definir el comportamiento del sistema; describen el resultado de la ejecución de las operaciones del sistema en función de los cambios de estado de los objetos del dominio. Este capítulo explora su uso.

13.1. Contratos

Los casos de uso son el principal mecanismo del UP para describir el comportamiento del sistema y, normalmente es suficiente. Sin embargo, algunas veces se necesita una descripción más detallada del comportamiento del sistema. Los contratos describen el comportamiento detallado del sistema en función de los cambios de estado de los objetos del Modelo del Dominio, después de la ejecución de una operación del sistema.

Operaciones del sistema y la interfaz del sistema

Se pueden definir contratos para las **operaciones del sistema** —operaciones que el sistema, como una caja negra, ofrece en su interfaz pública para manejar los eventos del sistema entrantes—. Las operaciones del sistema se pueden identificar descubriendo estos eventos del sistema, como se muestra en la Figura 13.1.

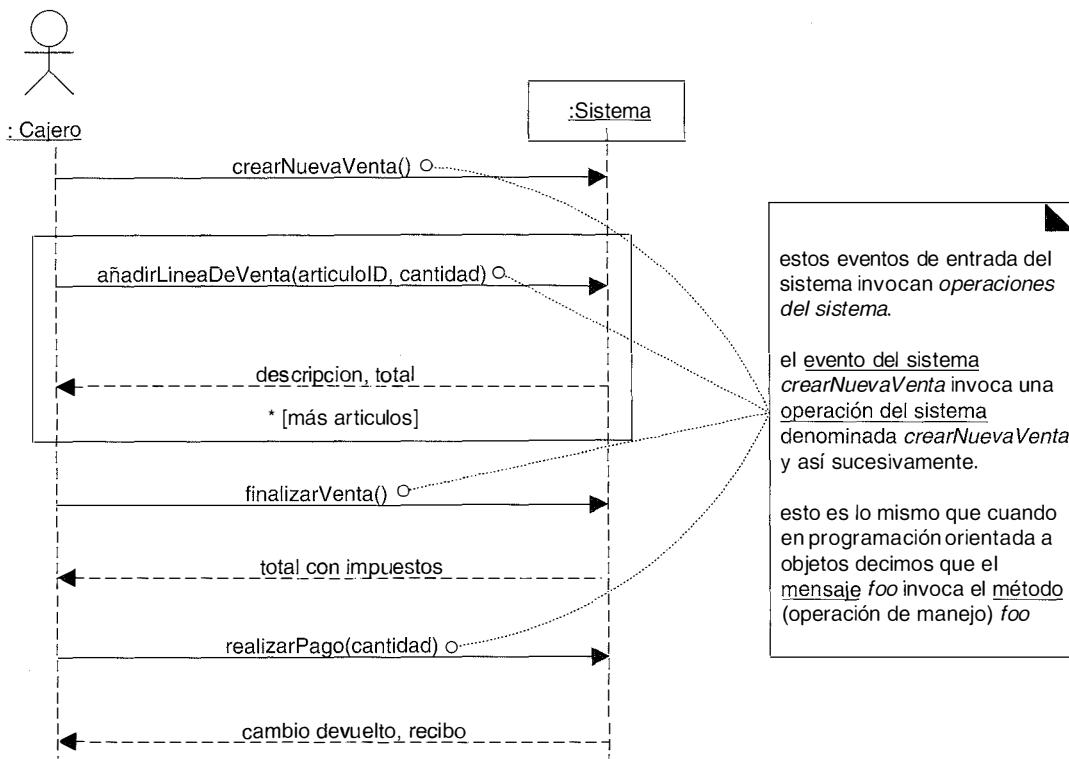


Figura 13.1. Las operaciones del sistema manejan los eventos de entrada del mismo.

El conjunto completo de operaciones del sistema, de todos los casos de uso, define la interfaz pública del sistema, viendo al sistema como un componente o clase individual. En UML, el sistema como un todo se puede representar mediante una clase.

13.2. Ejemplo de contrato: introducirArticulo

Antes de examinar las razones para la escritura de un contrato, merece la pena presentar un ejemplo. A continuación, se presenta un contrato para la operación del sistema *introducirArticulo*.

Contrato CO2: introducirArticulo

Operación:	introducirArticulo(articuloID:ArticulolID, cantidad:integer)
Referencias cruzadas:	Caso de Uso: Procesar Venta
Precondiciones:	Hay una venta en curso
Postcondiciones:	<ul style="list-style-type: none"> - Se creó una instancia de LineaDeVenta Idv (creación de instancias). - Idv se asoció con la Venta actual (formación de asociaciones). - Idv.cantidad pasó a ser cantidad (modificación de atributos). - Idv se asoció con una EspecificacionDelProducto, en base a la coincidencia del articuloID (formación de asociaciones).

13.3. Secciones del contrato

La descripción de cada una de las secciones del contrato se muestra en el siguiente esquema.

Operación:	Nombre de la operación y parámetros.
Referencias cruzadas:	(opcional) Casos de uso en los que pueden tener lugar esta operación.
Precondiciones:	<i>Suposiciones</i> relevantes sobre el estado del sistema o de los objetos del Modelo del Dominio, antes de la ejecución de la operación. No se comprobará en la lógica de esta operación, se asume que son verdad, y son suposiciones no triviales que el lector debe saber que se hicieron.
Postcondiciones:	- El estado de los objetos del Modelo del Dominio después de que se complete la operación. Se discute con detalle en la siguiente sección.

13.4. Postcondiciones

Nótese que cada una de las postcondiciones del ejemplo *introducirArticulo* incluía una categorización como *creación de instancias* o *formación de asociaciones*. He aquí un punto clave:

La postcondición describe cambios en el estado de los objetos del Modelo del Dominio. Los cambios de estado del Modelo del Dominio comprenden la creación de instancias, formación o rotura de asociaciones y cambio en los atributos.

Las postcondiciones no son acciones que se ejecutarán durante la operación; más bien, son declaraciones sobre los objetos del Modelo del Dominio que son verdad cuando la operación ha terminado —*después de que el humo se haya despejado*—.

En resumen, las postcondiciones se dividen en estas categorías:

- Creación y eliminación de instancias.
- Modificación de atributos.
- Formación y rotura de asociaciones (siendo precisos, *enlaces UML*).

Como ejemplo de postcondición que rompe una asociación, considere una operación que permite la eliminación de líneas de venta. La post-condición podría decir “Se rompió la asociación seleccionada de la *LíneaDeVenta* con la *Venta*.”. En otros dominios, cuando se cancela un préstamo o cuando alguien deja de ser socio de alguna organización, se rompen las asociaciones.

La postcondición de la eliminación de instancias es más rara, porque en el mundo real uno, normalmente, no se preocupa de forzar explícitamente la destrucción de una cosa. Sin embargo, como ejemplo: en muchos países, después de que una persona se

haya declarado en bancarrota y hayan pasado siete o diez años, se deben destruir todos los registros de su declaración de bancarrota, por ley. Nótese que esto es una perspectiva conceptual, no de implementación. Éstos no son declaraciones sobre liberar la memoria del ordenador ocupada por objetos software.

La cualidad importante es ser declarativo y enunciar con un estilo orientado al cambio en lugar de orientado a la acción, puesto que las postcondiciones son declaraciones sobre los estados o resultados, en lugar de una descripción de las acciones a ejecutar, o un diseño de una solución.

Las postcondiciones se relacionan con el Modelo del Dominio

Estas postcondiciones se expresan en el contexto de los objetos del Modelo del Dominio. ¿Qué instancias se pueden crear? —aquellas del Modelo del Dominio; ¿qué asociaciones se pueden formar? — las que se encuentran en el Modelo del Dominio; y así sucesivamente.

Una ventaja de las postcondiciones: detalle analítico

Expresados en un estilo declarativo de cambio de estado, los contratos son una herramienta excelente para el análisis de requisitos que describen los cambios de estado que requiere una operación del sistema (en función de los objetos del Modelo del Dominio) sin tener que describir *cómo* se van a llevar a cabo. En otras palabras, el diseño del software y la solución se puede diferir, y uno puede centrarse, analíticamente en *qué* debe suceder, en lugar de en *cómo* se va a realizar. Además, las postcondiciones soporan detalles de grano fino y una declaración más específica de cuál debe ser el resultado de la operación.

También es posible expresar este nivel de detalle en los casos de uso, pero normalmente no es deseable, puesto que entonces pasan a ser excesivamente elocuentes y detallados.

Considere la postcondición:

Postcondiciones:

- Se creó una instancia de *LíneaDeVenta Idv* (creación de instancias).
- *Idv* se asoció con la *Venta* actual (formación de asociaciones).
- *Idv.cantidad* pasó a ser *cantidad* (modificación de atributos).
- *Idv* se asoció con una *EspecificacionDelProducto*, en base a la coincidencia del *articuloID* (formación de asociaciones).

No se hace ningún comentario sobre el modo de crear una instancia de *LíneaDeVenta*, o cómo se asocia con una *Venta*. Esto podría ser una declaración sobre escribir en folios y que se grapen, la utilización de la tecnología Java para crear objetos software y conectarlos, o la inserción de filas en una base de datos relacional.

El espíritu de las postcondiciones: el escenario y telón

Exprese las postcondiciones en pasado, para resaltar que son declaraciones sobre un cambio de estado en el pasado. Por ejemplo:

- (mejor) Se *creó* una *LíneaDeVenta*
en lugar de

- (peor) Cree una *LíneaDeVenta*.

Piense en las postcondiciones utilizando la siguiente imagen:

- El sistema y sus objetos se presentan en el escenario de un teatro.

1. Antes de la operación, tome una fotografía del escenario.
2. Baje el telón y aplique la operación del sistema (*ruido metálico de fondo de martillos o campanas, gritos, chirridos...*).
3. Suba el telón y tome una segunda fotografía.
4. Compare las fotografías anterior y posterior, y exprese como postcondiciones el cambio en el estado del escenario (*Se creó una LíneaDeVenta...*).



Si se utilizan contratos, ¿cómo de completas deben ser las postcondiciones?

Primero, los contratos podrían no ser necesarios. Esta cuestión se discute en una posterior sección. Pero, asumiendo que se desean algunos contratos, no es probable —o incluso necesario— que se genere un conjunto completo y detallado de postcondiciones para una operación del sistema, durante el trabajo de requisitos. Trate su creación como la mejor suposición inicial, entendiendo que los contratos no serán completos. Su temprana creación —incluso si es incompleta— es, ciertamente, mejor que diferir su estudio hasta el trabajo de diseño, cuando los desarrolladores deben preocuparse por el diseño de una solución, en lugar de investigar *qué* se debe hacer.

Algunos de los detalles finos —y quizás incluso los más importantes— se descubrirán durante el trabajo de diseño. Esto no es necesariamente una cosa mala; si el esfuerzo dedicado al análisis de requisitos es demasiado grande el rendimiento decrece. Naturalmente, tiene lugar algún descubrimiento durante el trabajo de diseño, que puede entonces documentar el trabajo de requisitos de una iteración posterior. Ésta es una de las ventajas del desarrollo iterativo: los descubrimientos que se generan en una iteración anterior pueden impulsar el estudio y el trabajo de análisis de la siguiente.

13.5. Discusión: postcondiciones de introducirArticulo

La siguiente sección analiza la motivación de las postcondiciones de la operación del sistema *introducirArticulo*.

Creación y eliminación de instancias

Después de introducir el *articuloID* y la *cantidad* de un artículo, ¿qué nuevo objeto debe haberse creado? Una *LíneaDeVenta*. Por tanto:

- Se creó una instancia de *LíneaDeVenta ldv* (creación de instancias).

Obsérvese el nombre de la instancia. Este nombre simplificará las referencias a la nueva instancia en otras sentencias de la post-condición.

Modificación de atributos

Después de que el cajero haya introducido el *articuloID* y la *cantidad*, ¿qué atributos de los objetos nuevos o de los ya existentes deberían haberse modificado? La *cantidad* de la *LíneaDeVenta* debería haber pasado a ser igual que el parámetro *cantidad*. De ahí:

- *ldv.cantidad* pasó a ser *cantidad* (modificación de atributos).

Formación y rotura de asociaciones

Después de que el cajero haya introducido el *articuloID* y la *cantidad*, ¿qué asociaciones entre los objetos nuevos o los ya existentes deberían haberse formado o roto? La nueva *LíneaDeVenta* debería haberse relacionado con su *Venta*, y su *EspecificacionDelProducto*. Así:

- *ldv* se asoció con la *Venta* actual (formación de asociaciones).
- *ldv* se asoció con una *EspecificacionDelProducto*, en base a la coincidencia del *articuloID* (formación de asociaciones).

Nótese la indicación informal de que se forma una relación con una *EspecificacionDelProducto* particular —aquella cuyo *articuloID* se corresponda con el parámetro—. Son posibles otros lenguajes más sofisticados y formales, como utilizar el Lenguaje de Restricciones de Objetos (OCL, *Object Constraint Language*). Recomendación: manténgalo simple.

13.6. La escritura de los contratos da lugar a actualizaciones en el Modelo del Dominio

Es normal, durante la creación de los contratos, descubrir la necesidad de registrar nuevas clases conceptuales, atributos o asociaciones en el Modelo del Dominio. No se limite a la definición anterior del Modelo del Dominio; enriquezcalo cuando haga nuevos descubrimientos mientras piensa en los contratos de las operaciones.

13.7. ¿Cuándo son útiles los contratos? ¿Contratos vs. casos de uso?

Los casos de uso son el principal repositorio de requisitos del proyecto. Podrían proporcionar la mayoría o todos los detalles necesarios para saber qué hacer en el diseño, en cuyo caso, los contratos no son útiles. Sin embargo, hay situaciones en las que los detalles y la complejidad de los cambios de estado requeridos, son difíciles de capturar en los casos de uso.

Por ejemplo, considere un sistema de reservas de vuelos y la operación del sistema *añadirNuevaReserva*. La complejidad es muy alta considerando todos los objetos del dominio que se deben cambiar, crear y asociar. Estos detalles de grano fino se *pueden* escribir en detalle en el caso de uso asociado a esta operación, pero dará lugar a un caso de uso extremadamente detallado (por ejemplo, anotando cada atributo que se debe cambiar en todos los objetos).

Obsérvese que el formato de la postcondición del contrato ofrece y promueve un lenguaje muy preciso, analítico y exigente que soporta una detallada minuciosidad.

Si, únicamente basándose en los casos de uso y mediante continuas colaboraciones (verbales) con un experto en la materia de estudio, los desarrolladores pueden entender cómodamente qué hacer, entonces evite la escritura de los contratos.

Sin embargo, en aquellas situaciones donde la complejidad es alta y añade valor la precisión detallada, los contratos son otra herramienta de requisitos.

Muy a menudo, los contratos no estarán muy justificados de manera que si un equipo está creando contratos para todas las operaciones del sistema de cada caso de uso, es una advertencia de que, o bien los casos de uso son algo deficientes, o no hay suficiente y continua colaboración o acceso a los expertos en la materia de estudio, o el equipo está haciendo demasiada documentación innecesaria.

El caso de estudio del PDV NuevaEra muestra más contratos de los que probablemente sean necesarios, por cuestiones pedagógicas. En la práctica, la mayoría de los detalles que recogen se pueden inferir de manera obvia a partir del texto de los casos de uso. Por otro lado, “obvio” es un concepto muy escjuridizo.

13.8. Guías: contratos

Aplique el siguiente consejo para crear los contratos:

Para hacer contratos:

1. Identifique las operaciones del sistema a partir de los DSSs.
2. Construya un contrato para las operaciones del sistema complejas y quizás sutiles en sus resultados, o que no están claras en el caso de uso.
3. Para describir la postcondiciones utilice las siguientes categorías:
 - creación y eliminación de instancias
 - modificación de atributos
 - formación y rotura de asociaciones

Consejos acerca de la escritura de contratos

- Establezca las postcondiciones de forma declarativa, con una sentencia pasiva expresada en pasado para destacar que se trata de una declaración de un cambio de estado en lugar del diseño de la manera en la que se va a realizar. Por ejemplo:
 - (mejor) Se creó una *LíneaDeVenta*
 - (peor) Cree una *LíneaDeVenta*.
- Recuerde establecer una relación entre los objetos existentes o aquellos creados recientemente mediante la definición de la formación de asociaciones. Por ejemplo, no es suficiente que se cree una instancia de *LíneaDeVenta* cuando tenga lugar la operación *introducirArtículo*. Después de que se complete la operación, también debería cumplirse que la nueva instancia creada se asoció con la *Venta*; de ahí que:
 - La *LíneaDeVenta* se asoció con la *Venta* (formación de asociaciones).

El error más habitual en la creación de contratos

El problema más común es olvidarse de incluir la *formación de asociaciones*. En particular, cuando se crean nuevas instancias, es muy probable que se necesiten establecer asociaciones con varios objetos. ¡No lo olvide!

13.9. Ejemplo del PDV NuevaEra: contratos

Operaciones del sistema de Procesar Venta

Contrato CO1: crearNuevaVenta

Operación:	crearNuevaVenta()
Referencias cruzadas:	Caso de Uso: Procesar Venta
Precondiciones:	Ninguna
Postcondiciones:	<ul style="list-style-type: none"> - Se creó una instancia de <i>Venta v</i> (creación de instancias). - <i>v</i> se asoció con el Registro (formación de asociaciones). - Se inicializaron los atributos de <i>v</i>.

Obsérvese la descripción vaga de la última post-condición. Si es suficiente, está bien.

En un proyecto, todas estas postcondiciones particulares son tan obvias a partir del caso de uso que, probablemente, no se debería escribir el contrato de *crearNuevaVenta*.

Recuerde uno de los principios que guían un buen proceso y el UP: Manténgalo tan ligero como sea posible, y evite todos los artefactos a menos que realmente añadan valor.

Contrato CO2: introducirArtículo

Operación:	introducirArtículo(artículoID:ArtículoID, cantidad:integer)
Referencias cruzadas:	Caso de Uso: Procesar Venta

Precondiciones:	Hay una venta en curso
Postcondiciones:	<ul style="list-style-type: none"> - Se creó una instancia de <i>LíneaDeVenta</i> <i>ldv</i> (<i>creación de instancias</i>). - <i>ldv</i> se asoció con la <i>Venta</i> actual (<i>formación de asociaciones</i>). - <i>ldv.cantidad</i> pasó a ser <i>cantidad</i> (<i>modificación de atributos</i>). - <i>ldv</i> se asoció con una <i>EspecificaciónDelProducto</i>, en base a la <i>coincidencia del articuloID</i> (<i>formación de asociaciones</i>).

Contrato CO3: finalizarVenta

Operación:	finalizarVenta()
Referencias cruzadas:	Caso de Uso: Procesar Venta
Precondiciones:	Hay una venta en curso
Postcondiciones:	<ul style="list-style-type: none"> - <i>Venta.esCompleta</i> pasó a ser verdadero (<i>modificación de atributos</i>).

Contrato CO4: realizarPago

Operación:	realizarPago(cantidad: Dinero)
Referencias cruzadas:	Caso de Uso: Procesar Venta
Precondiciones:	Hay una venta en curso
Postcondiciones:	<ul style="list-style-type: none"> - Se creó una instancia de <i>Pago</i> <i>p</i> (<i>creación de instancias</i>). - <i>p.cantidadEntregada</i> pasó a ser <i>cantidad</i> (<i>modificación de atributos</i>). - <i>p</i> se asoció con la <i>Venta</i> actual (<i>formación de asociaciones</i>). - La <i>Venta</i> actual se asoció con la <i>Tienda</i> (<i>formación de asociaciones</i>); (para añadirlo al registro histórico de las ventas completadas).

13.10. Cambios en el Modelo del Dominio

Hay un dato sugerido por estos contratos que todavía no está representado en el modelo del dominio: la terminación de la entrada del artículo en la venta. La especificación de *finalizarVenta* lo modifica, y probablemente es una buena idea comprobarlo después, durante el trabajo de diseño, en la operación *realizarPago*, para rechazar los pagos hasta que se complete una venta.

Una manera de representar esta información es con un atributo *esCompleta* en la *Venta*, de tipo de dato booleano:

Venta
esCompleta: Boolean

fecha
hora

Existen alternativas, que se tienen en cuenta, especialmente, durante el trabajo de diseño. Una técnica se denomina el **patrón de Estado**, que se estudiará en el Capítulo 34. Otra es el uso de objetos “sesión” que siguen la pista del estado de una sesión y rechazan operaciones improcedentes; esto también se estudiará más tarde.

13.11. Contratos, operaciones y UML

Contratos en UML: especificación de operaciones

UML define las **operaciones** formalmente. Citando textualmente:

Una operación es una especificación de una transformación o consulta que se puede invocar para que la ejecute un objeto. [RJB99]

Por ejemplo, los elementos de una interfaz son operaciones, en términos de UML. Una operación es una abstracción, no una implementación. Por el contrario, un **método** (en UML) es una implementación de una operación.

Una operación UML tiene una **signatura** (nombre y parámetros), y también una **especificación de operación**, que describe los efectos producidos por la ejecución de la operación; esto es, la postcondición. El formato de la especificación de operación en UML es flexible, y no tiene que ser el formato de contrato que se muestra en este capítulo. Sin embargo, los documentos de UML proporcionan como ejemplos el estilo de contratos con pre- y postcondiciones, ya que éste es el enfoque más conocido para las especificaciones formales de las operaciones.

Resumiendo: UML define especificaciones de operaciones, que se pueden especificar con el estilo de los contratos con pre- y postcondiciones. Nótese que, como se subraya en este capítulo, una especificación de operación UML podría *no* mostrar un algoritmo o solución, sino únicamente los cambios de estado o efectos de las operaciones.

Además de utilizar los contratos para especificar las operaciones públicas del *Sistema* completo (operaciones del sistema), los contratos se pueden aplicar a las operaciones de cualquier nivel de granularidad: las operaciones públicas (o interfaz) de un subsistema, una clase abstracta, etcétera. Las operaciones presentadas en este capítulo pertenecen a la clase *Sistema*. En UML, las operaciones pertenecen a las clases. Más aún, en UML, los “subsistemas” se modelan como clases (y simultáneamente también como paquetes). En UML, el “sistema” global es el subsistema de más alto nivel, y se modela como una clase denominada *Sistema* (en realidad, cualquier nombre es legal) con operaciones y especificaciones públicas.

Contratos de las operaciones expresados con OCL

Existe un lenguaje formal asociado con UML denominado Lenguaje de Restricciones de Objetos (**OCL**, *Object Constraint Language*) [WK99], que se puede utilizar para expresar las restricciones en los modelos. OCL podría utilizarse en lugar del lenguaje natural informal que se utiliza en este capítulo; UML permite cualquier formato para una especificación de operación.

Sugerencia

A menos que exista una razón práctica apremiante que requiera que la gente aprenda y utilice OCL, mantenga las cosas simples y utilice el lenguaje natural.

OCL define un formato oficial para la especificación de las pre- y postcondiciones de las operaciones, como se muestra en este fragmento:

```
Sistema::crearNuevaVenta( )
pre: <sentencia en OCL>
post: ...
```

Detalles adicionales sobre OCL están fuera del alcance de este libro.

Contratos en el Diseño por Contrato

La forma de los contratos con pre- y postcondiciones utilizados para la especificación de las operaciones en UML, se lleva impulsando durante muchos años por Bertrand Meyer, formalizado en una técnica de diseño denominada **Diseño por Contrato** [Meyer97 (primera ed. 1989)], aunque su origen procede de un trabajo anterior en los años sesenta sobre los lenguajes de especificación formal. En el Diseño por Contrato, también se escriben los contratos para las operaciones de las clases de grano fino, no sólo para las operaciones públicas de los sistemas y subsistemas.

Además, el Diseño por Contrato fomenta la inclusión de una sección *invariante*, como es habitual en especificaciones de contrato completas. Los invariantes definen las cosas que no deben cambiar de estado antes y después de que se ejecute una operación. Los invariantes no se han utilizado en este capítulo por simplicidad.

Soporte de los lenguajes de programación para los contratos

Algunos lenguajes, como Eiffel, incluyen soporte a nivel de expresiones del lenguaje para los invariantes, las pre- y postcondiciones. Existen pre-procesadores en Java que proporcionan un soporte parecido.

13.12. Contratos de las operaciones en el UP

Un contrato con pre- y postcondiciones es un estilo bien conocido para especificar una operación en UML. En UML, las operaciones existen a muchos niveles, desde el *Sistema* hasta las clases de grano fino, como *Venta*. Los contratos de especificación de operaciones para el nivel del *Sistema* forman parte del Modelo de Casos de Uso, aunque no se pusieron de relieve en la documentación original del RUP o el UP; su inclusión en este modelo se verificó con los autores del RUP¹.

Fases

Inicio: Los contratos no se justifican durante la fase de inicio, son demasiado detallados.

Elaboración: Si es que se utilizan, los contratos se escribirán durante la elaboración, cuando se escriben la mayoría de los casos de uso. Escriba solamente los contratos para las operaciones del sistema más complejas y sutiles.

Relaciones entre los artefactos

En las Figuras 13.2 y 13.3 se muestran las relaciones entre los contratos y otros artefactos, con diferentes niveles de detalle.

¹ Comunicación privada.

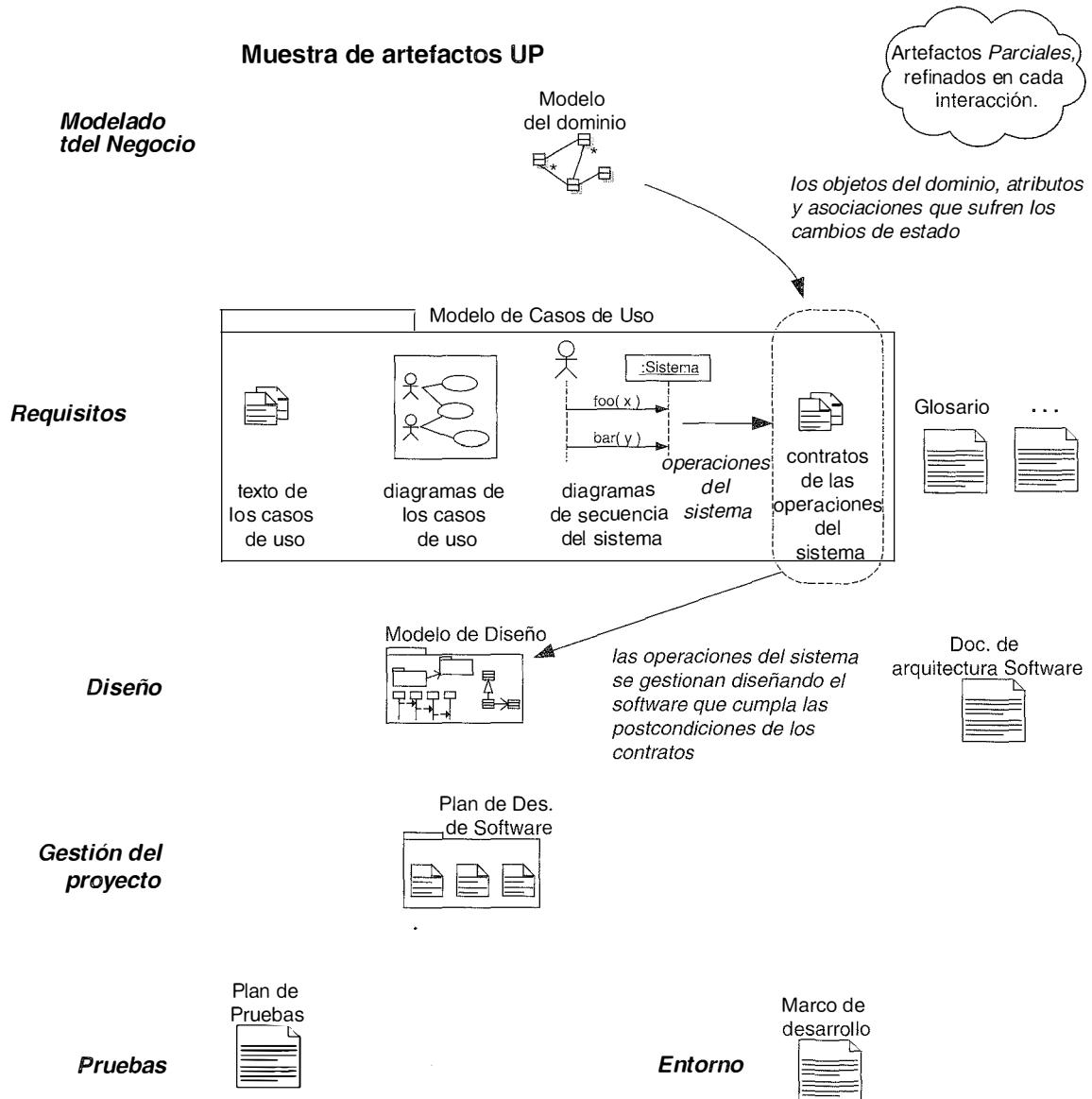


Figura 13.2. Muestra de la influencia entre los artefactos UP.

13.13. Lecturas adicionales

Los contratos de las operaciones proceden del área de las especificaciones formales, y se llevan utilizando y refinando desde los años sesenta, como en el Método de Desarrollo Viena (VDM, *Vienna Development Method*) [BJ78]; existe literatura abundante sobre VDM y otros lenguajes de especificación formal.

Bertrand Meyer contribuyó a un reconocimiento mucho más amplio de las especificaciones formales y los contratos con la inclusión de las pre- y postcondiciones en el lenguaje Eiffel; su *Construcción de Software Orientado a Objetos* proporciona los detalles. Es el creador del **Diseño por Contrato**.

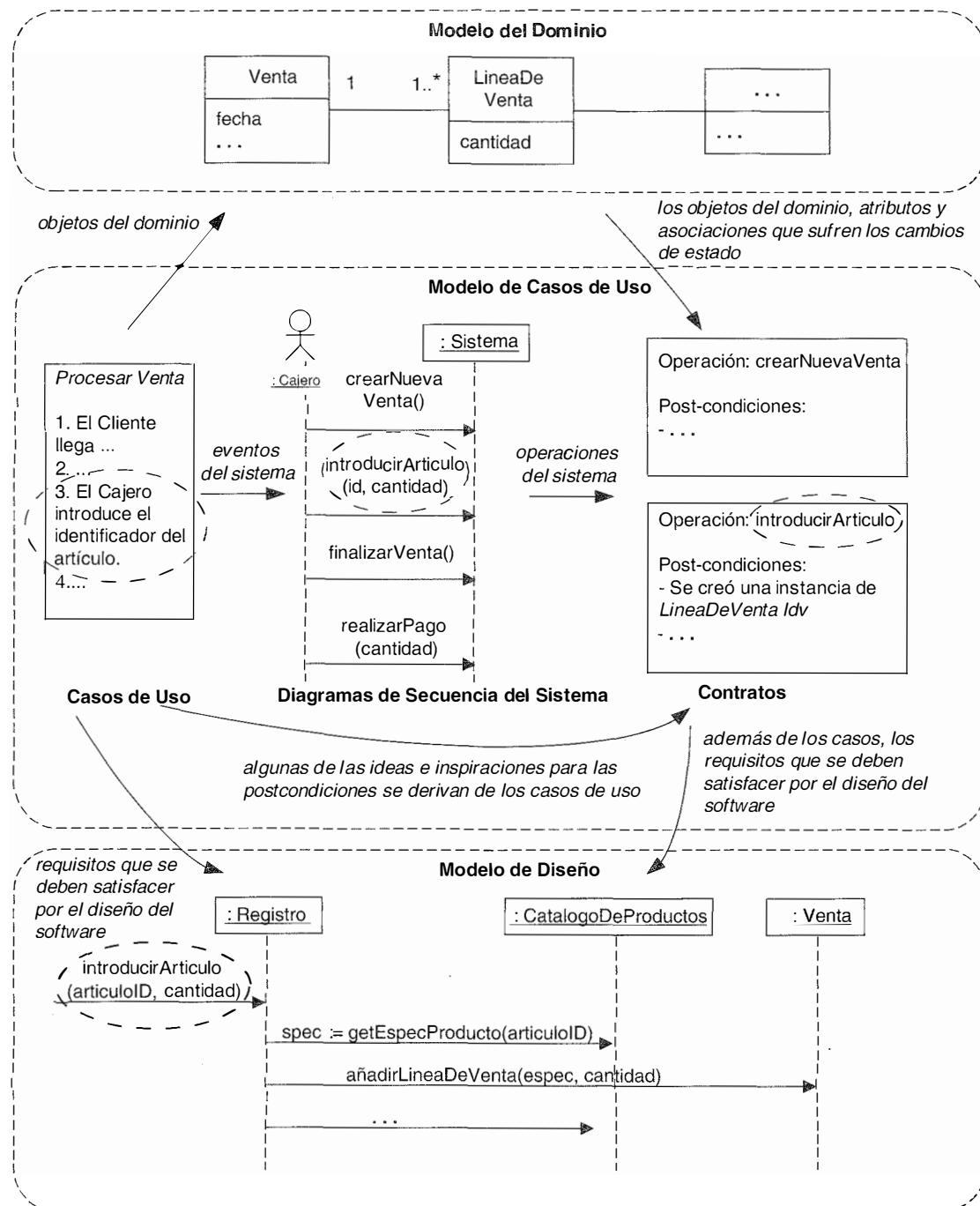


Figura 13.3. Relación del contrato con otros artefactos.

En UML, los contratos de las operaciones también pueden especificarse de manera más rigurosa en el Lenguaje de Restricciones de Objetos (OCL), para lo que se requiere leer *The Object Constraint Language: Precise Modeling with UML*, de Warmer y Kleppe.

Capítulo 14

DE LOS REQUISITOS AL DISEÑO EN ESTA ITERACIÓN

Hardware, n.: Las partes de un ordenador a las que se pueden dar patadas.

Anónimo

Objetivos

- Motivar la transición a las actividades de diseño.
 - Contrastar la importancia de las técnicas de diseño de objetos frente al conocimiento de la notación UML.
-

Introducción

Hasta el momento, el caso de estudio ha hecho hincapié en el estudio de los requisitos, conceptos y operaciones relacionadas con un sistema. Siguiendo las guías del UP, quizás se investigaron el 10% de los requisitos en la fase de inicio, y se comenzó un estudio ligeramente más profundo en esta primera iteración de la elaboración. Los capítulos siguientes cambian el énfasis hacia el diseño de una solución para esta iteración en función de objetos software que colaboran.

14.1. Iterativamente hacer lo correcto, y hacerlo correcto

Los requisitos y el análisis orientado a objetos se han centrado en aprender a *hacer lo correcto*; es decir, en entender algunos de los objetivos importantes del PDV NuevaEra, y las reglas y restricciones relacionadas. Por el contrario, el siguiente trabajo de diseño pondrá de relieve *hacerlo correcto*; esto es, diseñar con destreza una solución que satisfaga los requisitos de esta iteración.

En el desarrollo iterativo, en cada iteración tendrá lugar una transición desde un enfoque centrado en los requisitos a un enfoque centrado en el diseño y la implementación.

Además, es normal y saludable descubrir y cambiar algunos requisitos de las *primeras* iteraciones durante el trabajo de diseño e implementación. Estos descubrimientos clarificarán los objetivos del trabajo de diseño de esta iteración, y refinarán la comprensión de los requisitos para las iteraciones futuras. A lo largo de estas primeras iteraciones de elaboración el descubrimiento de los requisitos se estabilizará, de manera que al final de la elaboración quizás se definan con detalle y de manera fiable el 80% de los requisitos.

14.2. ¿No lleva eso semanas en hacerse? No, no exactamente

Después de muchos capítulos de discusión detallada, seguramente, debe parecer que el modelado anterior debe llevar semanas de esfuerzo. No es así. Cuando uno se encuentra cómodo con las técnicas de escritura de casos de uso, modelado del dominio, etcétera, el tiempo necesario para llevar a cabo el modelado real que hemos estudiado hasta el momento, de modo realista, será sólo de unos *pocos* días.

Sin embargo, eso no significa que sólo hayan pasado unos días desde el comienzo del proyecto. Muchas otras actividades, como programación de pruebas de conceptos, búsqueda de recursos (personas, software,...), planificación, acondicionamiento del entorno, etcétera, podrían consumir unas pocas semanas de preparación.

14.3. Pasar al diseño de objetos

Durante el diseño de objetos, se desarrolla una solución lógica basada en el paradigma orientado a objetos. Lo esencial de esta solución es la creación de los **diagramas de interacción**, que representan el modo en el que los objetos colaboran para satisfacer los requisitos.

Después de —o en paralelo con— la elaboración de los diagramas de interacción, se pueden representar los **diagramas de clases** (del diseño). Éstos resumen la definición de las clases software (e interfaces) que se van a implementar en el software.

Por lo que se refiere al UP, estos artefactos forman parte del **Modelo de Diseño**.

En la práctica, la creación de los diagramas de interacción y clases tiene lugar en paralelo y con sinergia entre ellos, aunque se introducen de manera lineal en este caso de estudio, por simplicidad y claridad.

La importancia de las técnicas del diseño de objetos vs. las técnicas de la notación UML

Los capítulos siguientes estudian la creación de estos artefactos, o de manera más precisa, las técnicas del diseño de objetos que subyacen a sus creaciones. Lo que es importante es saber cómo pensar y diseñar con objetos, que es muy diferente y una capa-

ciudad más importante, que conocer la notación de los diagramas de UML. Al mismo tiempo, un lenguaje visual estándar es importante y, por tanto, se presenta la notación UML necesaria para dar soporte al trabajo de diseño.

De los dos artefactos que se estudiarán, los diagramas de interacción son los más importantes —desde el punto de vista del desarrollo de un buen diseño— y requiere el mayor grado de esfuerzo creativo. La creación de los diagramas de interacción requiere la aplicación de los principios para la asignación de **responsabilidades** y el uso de los **principios y patrones de diseño**. Por tanto, el énfasis de los siguientes capítulos se pone en estos principios y patrones del diseño de objetos.

Las técnicas del diseño de objetos vs. las técnicas de la notación UML

La representación de los diagramas de interacción UML es un reflejo de la toma de decisiones sobre el diseño de objetos.

Las técnicas del diseño de objetos son lo que realmente importan, en lugar de saber cómo dibujar los diagramas UML.

El diseño de objetos básico requiere que se tenga conocimiento sobre:

- Los principios de asignación de responsabilidades.
- Patrones de diseño.

Capítulo 15

NOTACIÓN DE LOS DIAGRAMAS DE INTERACCIÓN

Los gatos son más listos que los perros. No puedes conseguir que ocho gatos empujen un trineo por la nieve.

Jeff Valdez

Objetivos

- Leer la notación de los diagramas de interacción (secuencia y colaboración) UML básicos.
-

Introducción

Los siguientes capítulos estudian el diseño de objetos. El lenguaje utilizado para ilustrar los diseños es, principalmente, los diagramas de interacción. Por tanto, es aconsejable, por lo menos, examinar superficialmente los ejemplos de este capítulo y familiarizarse con la notación antes de avanzar.

UML incluye los **diagramas de interacción** para ilustrar el modo en el que los objetos interaccionan por medio de mensajes. Este capítulo introduce la notación, mientras que los capítulos siguientes se centran en su uso en el contexto del aprendizaje y realización del diseño de objetos para el caso de estudio del PDV NuevaEra.

Lea los siguientes capítulos para las guías de diseño

Este capítulo introduce la notación. Para crear objetos bien diseñados, también se deben entender los principios de diseño. Después de familiarizarse algo con la notación de los diagramas de interacción, es importante estudiar los siguientes capítulos sobre estos principios y cómo aplicarlos mientras se dibujan los diagramas de interacción.

15.1. Diagramas de secuencia y colaboración

EL término *diagrama de interacción* es una generalización de dos tipos de diagramas UML más especializados; ambos pueden utilizarse para representar de forma similar interacciones de mensajes:

- Diagramas de colaboración.
- Diagramas de secuencia.

A lo largo del libro, se utilizarán ambos tipos, para remarcar la flexibilidad en la elección.

Los **diagramas de colaboración** ilustran las interacciones entre objetos en un formato de grafo o red, en el cual los objetos se pueden colocar en cualquier lugar del diagrama, como se muestra en la Figura 15.1.

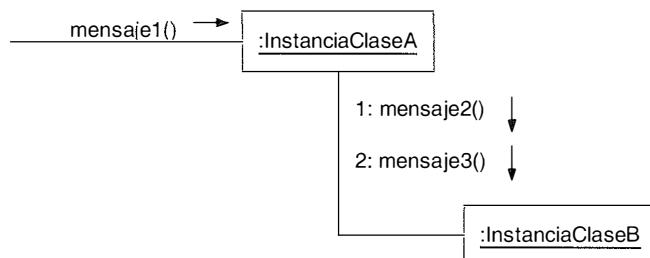


Figura 15.1. Diagrama de colaboración.

Los **diagramas de secuencia** ilustran las interacciones en un tipo de formato con el aspecto de una valla, en el que cada objeto nuevo se añade a la derecha, como se muestra en la Figura 15.2.

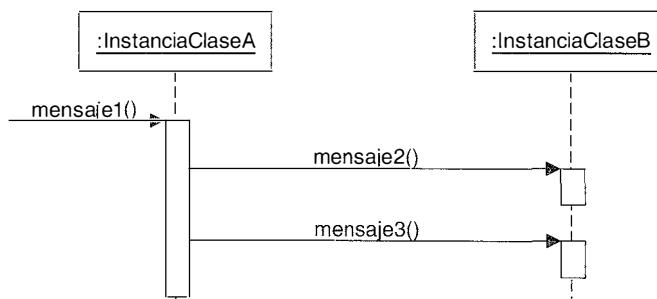


Figura 15.2. Diagrama de secuencia.

Cada tipo tiene puntos fuertes y débiles. Cuando se dibujan los diagramas para publicarlos en páginas estrechas, los diagramas de colaboración tienen la ventaja de permitir la expansión vertical para los nuevos objetos; los objetos adicionales en un diagrama de secuencia deben extenderse hacia la derecha, lo que supone una limitación. Por otro lado, los ejemplos de diagramas de colaboración dificultan que se vea fácilmente la secuencia de mensajes.

La mayoría prefieren los diagramas de secuencia cuando utilizan una herramienta CASE para hacer ingeniería inversa del código fuente a diagramas de interacción, puesto que muestran claramente la secuencia de mensajes.

Tipo	Puntos fuertes	Puntos débiles
secuencia	muestra claramente la secuencia u ordenación en el tiempo de los mensajes notación simple	fuerza a extender por la derecha cuando se añaden nuevos objetos; consume espacio horizontal
colaboración	economiza espacio, flexibilidad al añadir nuevos objetos en dos dimensiones es mejor para ilustrar bifurcaciones complejas, iteraciones y comportamiento concurrente	difícil ver la secuencia de mensajes notación más compleja

15.2. Ejemplo de diagrama de colaboración: realizarPago

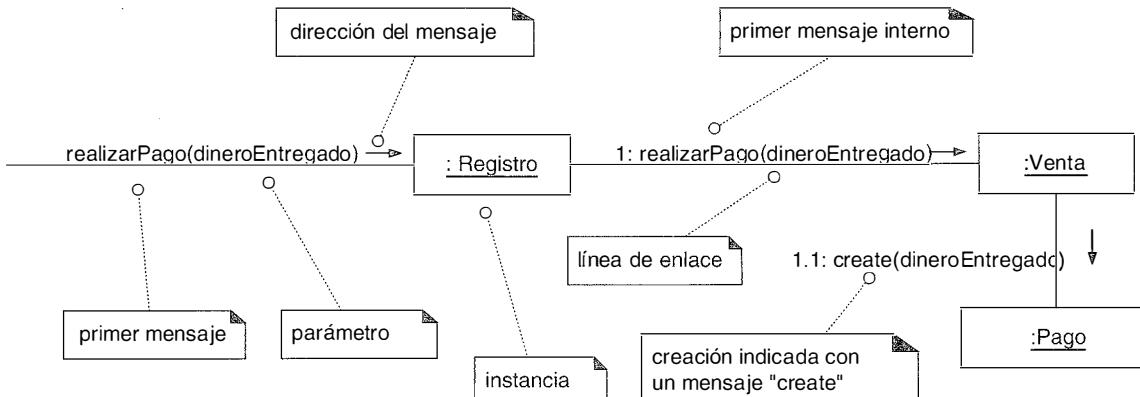


Figura 15.3. Diagrama de colaboración.

El diagrama de colaboración que se muestra en la Figura 15.3 se lee como sigue:

1. Se envía el mensaje *realizarPago* a una instancia de *Registro*. No se identifica al emisor.
2. La instancia de *Registro* envía el mensaje *realizarPago* a una instancia de *Venta*.
3. La instancia de *Venta* crea una instancia de *Pago*.

15.3. Ejemplo de diagrama de secuencia: realizarPago

El objetivo del diagrama de secuencia que se muestra en la Figura 15.4 es el mismo que el del anterior diagrama de colaboración.

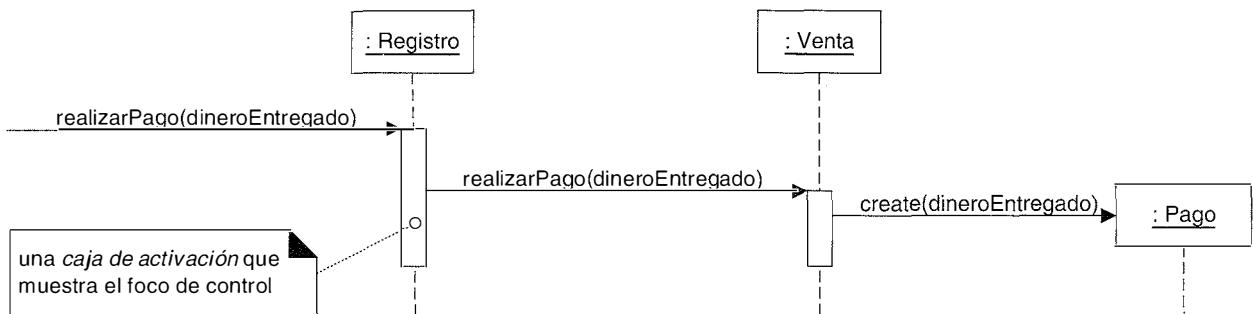


Figura 15.4. Diagrama de secuencia.

15.4. Los diagramas de interacción son importantes

Un problema típico en los proyectos de tecnología de objetos es que no aprecian el valor de llevar a cabo el diseño de objetos mediante el uso de los diagramas de interacción. Un problema relacionado es hacerlos de una manera vaga, como mostrando mensajes hacia objetos que realmente necesitan mucha elaboración adicional; por ejemplo, mostrando el mensaje *ejecutarSimulacion* a algún objeto *Simulacion*, pero sin continuar con el diseño más detallado, como pensando que en virtud de un mensaje con un buen nombre el diseño se completa mágicamente.

Se debería dedicar un tiempo y esfuerzo no trivial a la creación de diagramas de interacción, como reflejo de que se ha estudiado cuidadosamente los detalles del diseño de objetos. Por ejemplo, si la duración de la iteración es de dos semanas, quizás medio día o uno entero, cerca del comienzo de la iteración, se debería dedicar a la creación de los diagramas de interacción (y en paralelo, los diagramas de clases), antes de pasar a la programación. Sí, es cierto, el diseño que se representa en los diagramas será imperfecto y especulativo, y se modificará durante la programación, pero proporcionará un punto de partida serio, consistente y común, que sirve de inspiración durante la programación.

Sugerencia

Cree los diagramas de interacción por parejas, no solo. El diseño colaborando con otro será mejor, y los compañeros aprenderán rápidamente los unos de los otros.

Nótese que es principalmente durante esta etapa donde se requiere la aplicación de las técnicas de diseño, en términos de patrones, estilos y principios. La creación de los casos de uso, modelos del dominio, y otros artefactos, es más sencilla que la asignación de responsabilidades y la creación de diagramas de interacción bien diseñados. Esto es debido a que existen un gran número de principios de diseño sutiles y “grados de libertad” que subyacen a un diagrama de interacción bien diseñado, que a la mayoría de los otros artefactos de A/DOO.

La realización de los diagramas de interacción (en otras palabras, decidir sobre los detalles del diseño de objetos) es una etapa muy creativa del A/DOO.

Se pueden aplicar patrones codificados, principios y estilos para mejorar la calidad de sus diseños.

Los principios de diseño que se necesitan para la construcción con éxito de los diagramas de interacción *pueden* codificarse, explicarse y aplicarse de forma sistemática. Este enfoque para entender y utilizar los principios de diseño se basa en los **patrones** —guías y principios estructurados—. Por tanto, después de introducir la sintaxis de los diagramas de interacción, volveremos la atención (en los siguientes capítulos) hacia los patrones de diseño y su aplicación a los diagramas de interacción.

15.5. Notación general de los diagramas de interacción

Representación de clases e instancias

UML ha adoptado un enfoque simple y consistente para representar las **instancias** frente a los clasificadores (ver Figura 15.5):

- Para cualquier tipo de elemento UML (clase, actor,...), una instancia utiliza el mismo símbolo gráfico que el tipo, pero con la cadena de texto que lo designa subrayada.

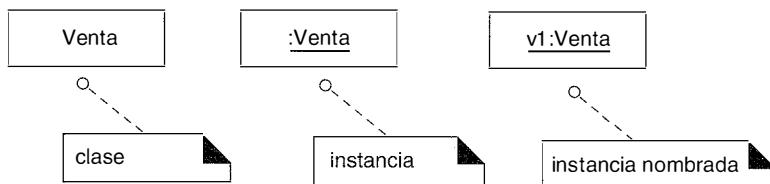


Figura 15.5. Clases e instancias.

En consecuencia, para mostrar una instancia de una clase en un diagrama de interacción, se utiliza el símbolo gráfico para una clase, esto es el rectángulo, pero con el nombre subrayado.

Se puede utilizar un nombre para identificar de manera única la instancia. Si no se utiliza, obsérvese que los “:” preceden al nombre de la clase.

Sintaxis de la expresión de mensaje básica

UML cuenta con una sintaxis básica para las expresiones de los mensajes:

return := mensaje(parametro: tipoParametro): tipoRetorno

Podría excluirse la información del tipo si es obvia o no es importante. Por ejemplo:

```

espec := getEspecProducto(id)
espec := getEspecProducto(id:ArticuloID)
espec := getEspecProducto(id:ArticuloID):EspecificacionDelProducto
  
```

15.6. Notación básica de los diagramas de colaboración

Enlaces

Un **enlace** es un camino de conexión entre dos objetos; indica que es posible alguna forma de navegación y visibilidad entre los objetos (ver Figura 15.6). De manera más formal, un enlace es una instancia de una asociación. Por ejemplo, existe un enlace —o camino de navegación— desde un *Registro* a una *Venta*, a lo largo del que pueden fluir los mensajes, como el mensaje *realizarPago*.

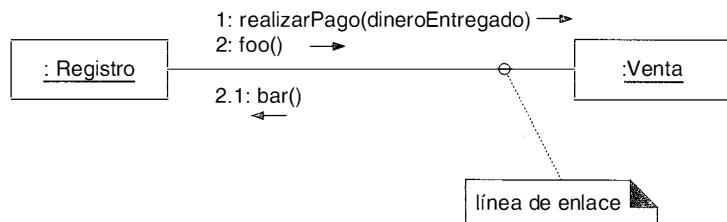


Figura 15.6. Líneas de enlaces.

Obsérvese que, a lo largo del mismo enlace, pueden fluir múltiples mensajes, y mensajes en ambas direcciones.

Mensajes

Cada mensaje entre objetos se representa con una expresión de mensaje y una pequeña flecha que indica la dirección del mensaje. Podrán fluir muchos mensajes a lo largo de este enlace (Figura 15.7). Se añade un número de secuencia para mostrar el orden secuencial de los mensajes en el hilo de control actual.

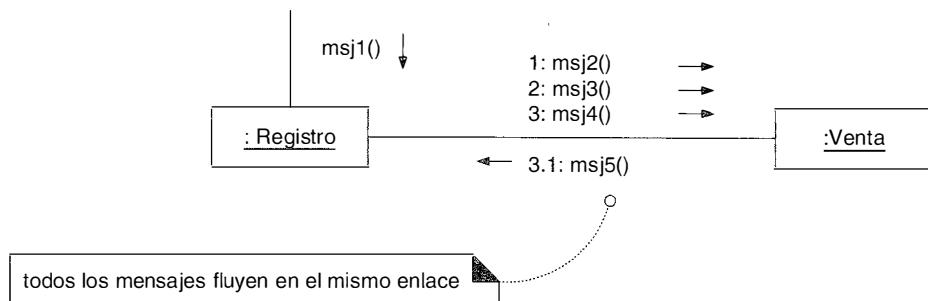
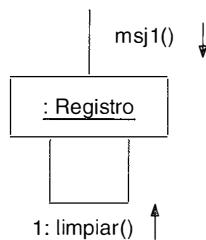


Figura 15.7. Mensajes.

Mensajes a “self” o “this”

Se puede enviar un mensaje desde un objeto a él mismo (Figura 15.8). Esto se representa mediante un enlace a él mismo, con mensajes que fluyen a lo largo del enlace.

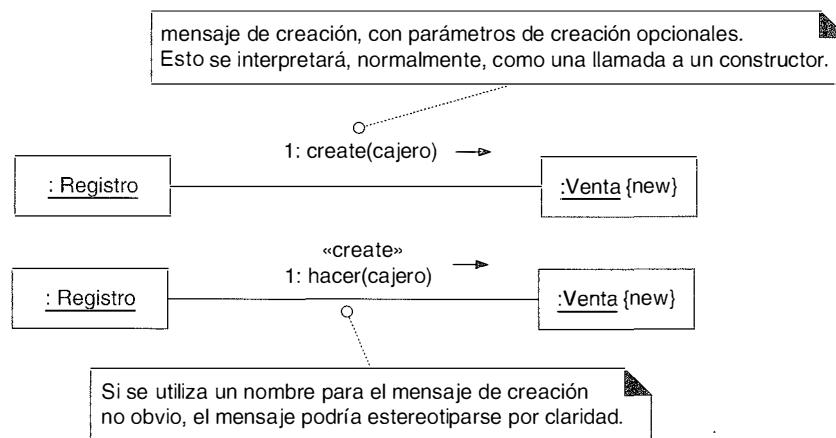
**Figura 15.8.** Mensajes a “this”.

Creación de instancias

Cualquier mensaje se puede utilizar para crear una instancia, pero en UML existe el convenio de utilizar para este fin el mensaje denominado *create*. Si se utiliza otro nombre de mensaje (quizás menos obvio), se podría añadir al mensaje una característica especial llamada estereotipo UML, como «*create*».

El mensaje *create* podría incluir parámetros, que indican el paso de valores iniciales. Esto indica, por ejemplo, la llamada a un constructor con parámetros en Java.

Además, podría añadirse opcionalmente la propiedad UML *{new}* a la caja de la instancia para resaltar la creación.

**Figura 15.9.** Creación de instancias.

Secuencia de números de mensaje

El orden de los mensajes se representa mediante **números de secuencia**, como se muestra en la Figura 15.10. El esquema de numeración es:

1. No se numera el primer mensaje. Por tanto, el *msj1()* no se numera.
2. El orden y anidamiento de los siguientes mensajes se muestran con el esquema de numeración válido en el que los mensajes anidados tienen un número adjunto. El anidamiento se denota anteponiendo el número del mensaje entrante al número del mensaje saliente.

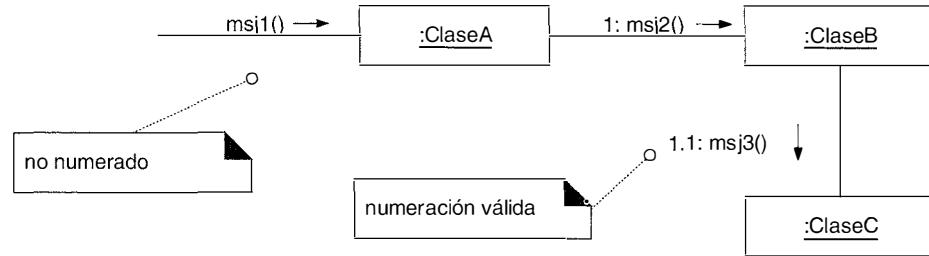


Figura 15.10. Secuencia de numeración.

En la Figura 15.11 se muestra un caso más complejo.

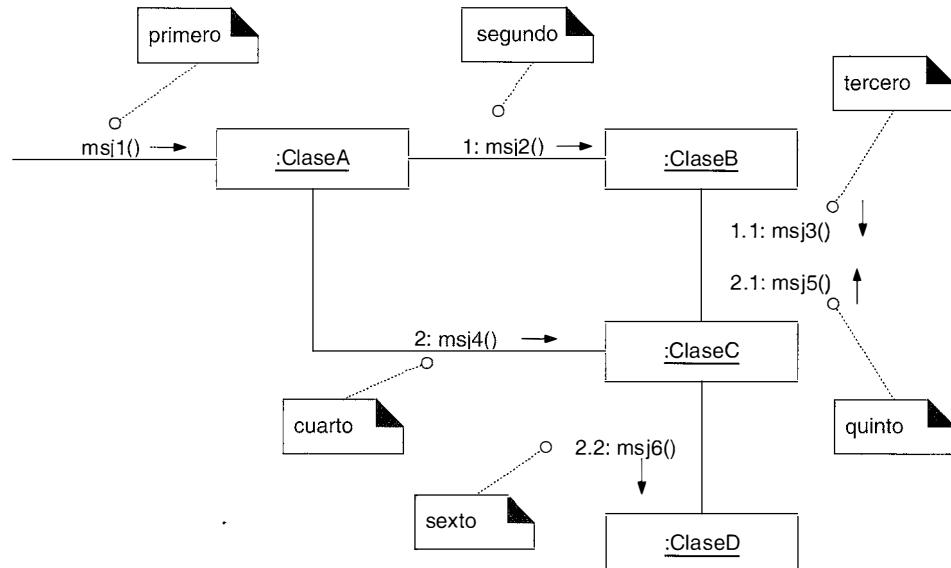


Figura 15.11. Secuencia de numeración compleja.

Mensajes condicionales

Un mensaje condicional (ver Figura 15.12) se muestra con una cláusula condicional, similar a una cláusula de iteración, entre corchetes, a continuación del número de secuencia. El mensaje sólo se envía si la evaluación de la cláusula es *verdad*.

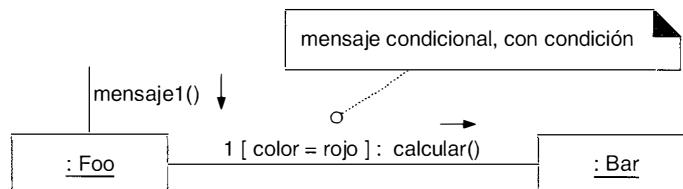


Figura 15.12. Mensaje condicional.

Caminos condicionales mutuamente exclusivos

El ejemplo de la Figura 15.13 ilustra los números de secuencia con caminos condicionales mutuamente exclusivos.

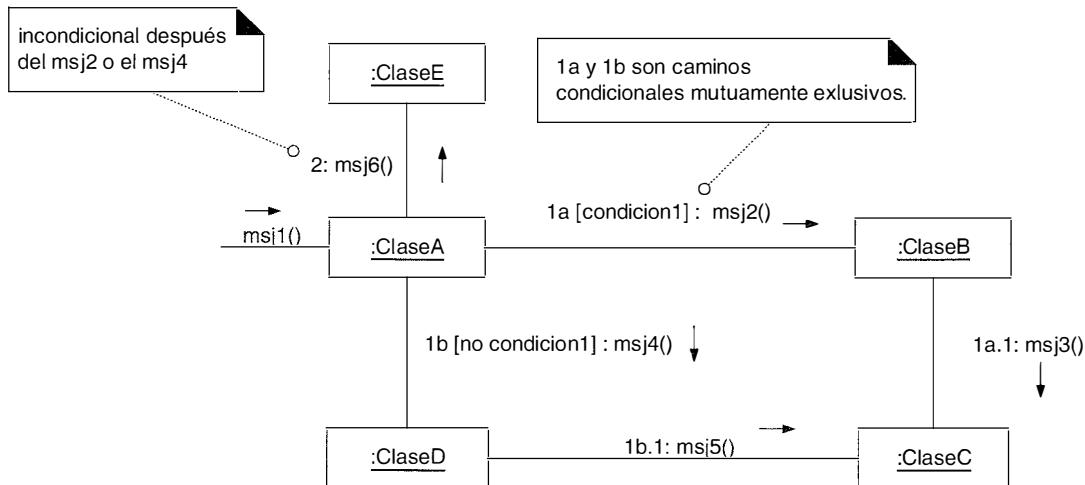


Figura 15.13. Mensajes mutuamente exclusivos.

En este caso es necesario modificar las expresiones de la secuencia con una letra de camino condicional. La primera letra que se utiliza es la **a** por convenio. La Figura 15.13 establece que se podría ejecutar o *1a* o *1b* después de *msj1()*. Ambos tienen el número de secuencia 1 puesto que cualquiera de ellos podría ser el primer mensaje interno.

Nótese que los siguientes mensajes anidados todavía se anteponen de manera consistente con su secuencia de mensaje exterior. De este modo *1b.1* es el mensaje anidado de *1b*.

Iteración o bucle

La notación de las iteraciones se muestra en la Figura 15.14. Si los detalles de la cláusula de iteración no son importantes para el modelador, se puede utilizar simplemente un “*”.

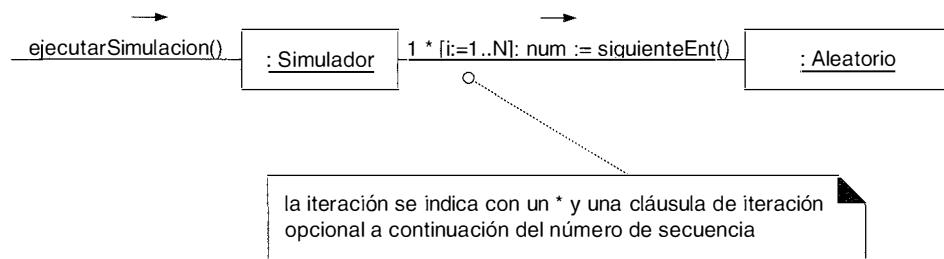


Figura 15.14. Iteración.

Iteración sobre una colección (multiobjeto)

Un algoritmo típico es iterar sobre todos los miembros de una colección (como una lista o una tabla), enviando un mensaje a cada uno de ellos. A menudo, se utiliza, finalmente, algún tipo de objeto iterador, como una implementación de `java.util.Iterator` o el iterador de la librería estándar de C++. En UML, el término **multiobjeto** se utiliza para denotar un conjunto de instancias —una colección—. En los diagramas de colaboración, se puede indicar como se muestra en la Figura 15.15.

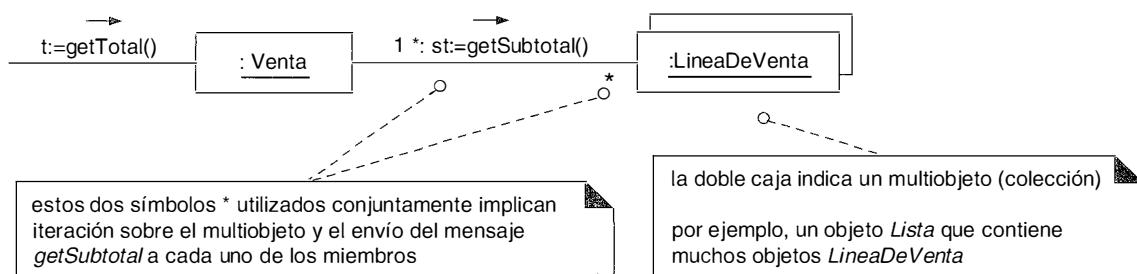


Figura 15.15. Iteración sobre un multiobjeto.

El marcador de multiplicidad “*” al final del enlace, se utiliza para indicar que se va a enviar el mensaje a cada elemento de la colección, en lugar de enviarse repetidamente al propio objeto colección.

Mensaje a un objeto clase

Los mensajes se podrían enviar a las propias clases, en lugar de una instancia, para invocar a los **métodos estáticos** o de clase. Se muestra un mensaje hacia el rectángulo de una clase cuyo nombre no está subrayado, lo que indica que el mensaje se está enviando a una clase en lugar de a una instancia (ver Figura 15.16).

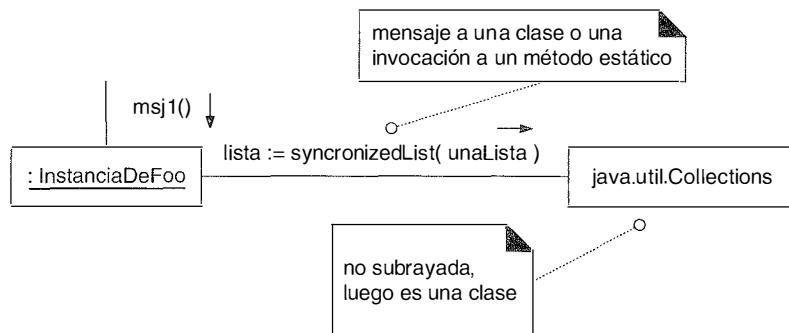


Figura 15.16. Mensaje a un objeto clase (invocación de un método estático).

En consecuencia es importante ser consistente y subrayar los nombre de las instancias cuando lo que se desea es una instancia, de otra manera, se podrían interpretar incorrectamente los mensajes a clases o instancias.

15.7. Notación básica de los diagramas de secuencia

Enlaces

A diferencia de los diagramas de colaboración, los diagramas de secuencia no muestran enlaces.

Mensajes

Cada mensaje entre objetos se representa con una expresión de mensaje sobre una línea con punta de flecha entre los objetos (ver Figura 15.17). El orden en el tiempo se organiza de arriba a abajo.

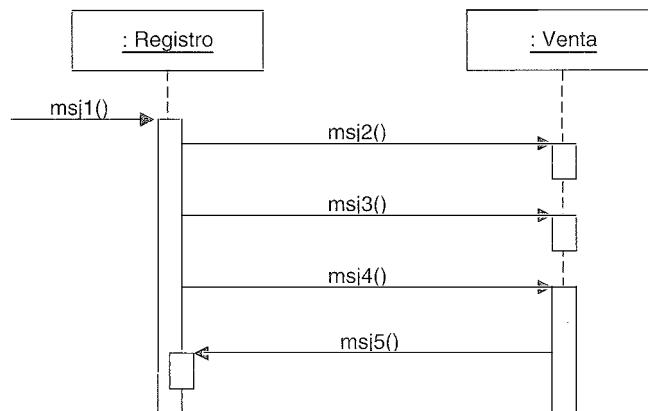


Figura 15.17. Mensajes y focos de control con cajas de activación.

Focos de control y cajas de activación

Como se ilustra en la Figura 15.17, los diagramas de secuencia podrían también mostrar los focos de control (esto es, en una llamada de rutina ordinaria, la operación se encuentra en la pila de llamadas) utilizando una **caja de activación**. La caja es opcional, pero la utilizan habitualmente los modeladores de UML.

Representación de retornos

Un diagrama de secuencia podría opcionalmente mostrar el retorno de un mensaje mediante una línea punteada con la punta de flecha abierta, al final de una caja de activación (ver Figura 15.18). Lo normal es que se excluya por quienes utilizan UML. Algunos anotan la línea de retorno para describir lo que se está devolviendo (si es el caso) a partir del mensaje.

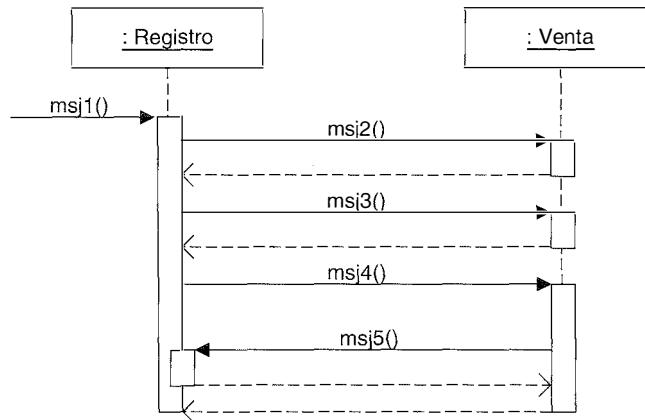


Figura 15.18. Representación de retornos.

Mensajes a “self” o “this”

Se puede representar un mensaje que se envía de un objeto a él mismo utilizando una caja de activación anidada (ver Figura 15.19).

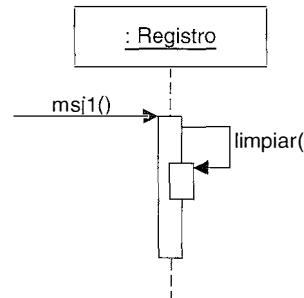


Figura 15.19. Mensajes a “this”.

Creación de instancias

La notación para la creación de instancias se muestra en la Figura 15.20.

Línea de vida del objeto y destrucción de objetos

La Figura 15.20 también ilustra las **líneas de vida de los objetos** —las líneas punteadas verticales bajo los objetos—. Éstas indican la duración de la vida de los objetos en el diagrama. En algunas circunstancias es deseable mostrar la destrucción explícita de un objeto (como en C++, que no tiene recogida de basura); la notación UML para las líneas de vida proporcionan una forma para expresar esta destrucción (ver Figura 15.21).

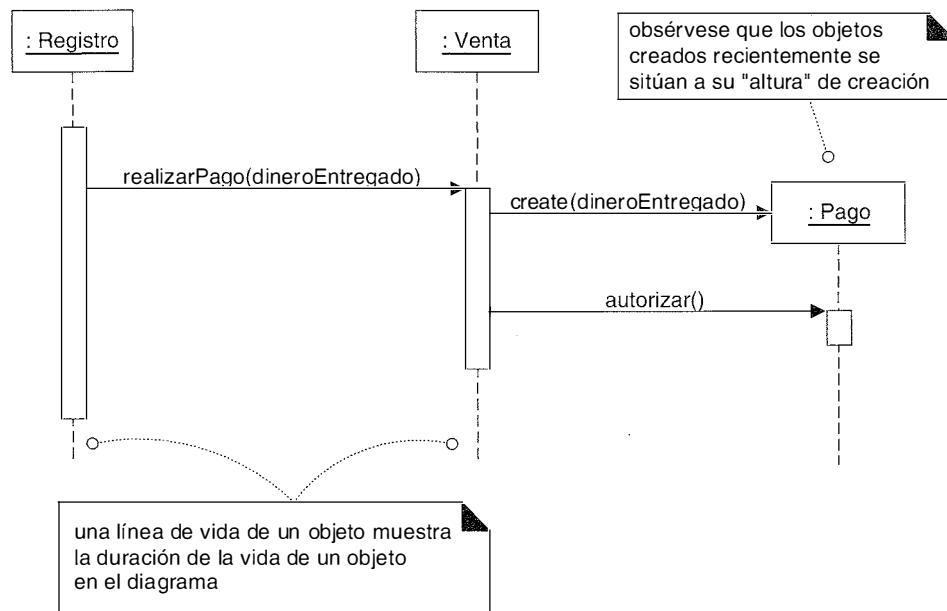


Figura 15.20. Creación de instancias y línea de vida de los objetos.

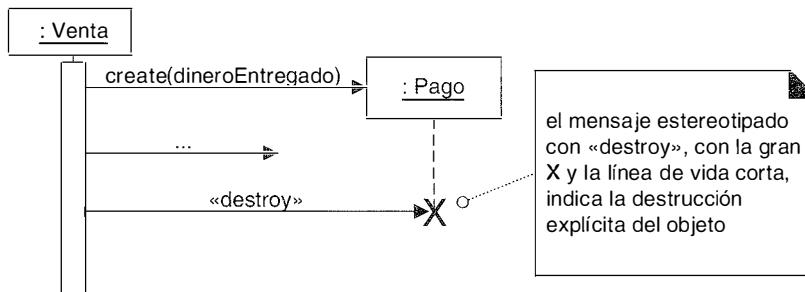


Figura 15.21. Destrucción de objetos.

Mensajes condicionales

La Figura 15.22 muestra un mensaje condicional.

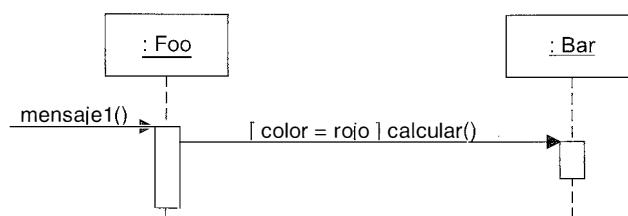


Figura 15.22. Un mensaje condicional.

Mensajes condicionales mutuamente exclusivos

La notación para este caso es un tipo de línea de mensaje con forma de ángulo que nace desde un mismo punto, como se ilustra en la Figura 15.23.

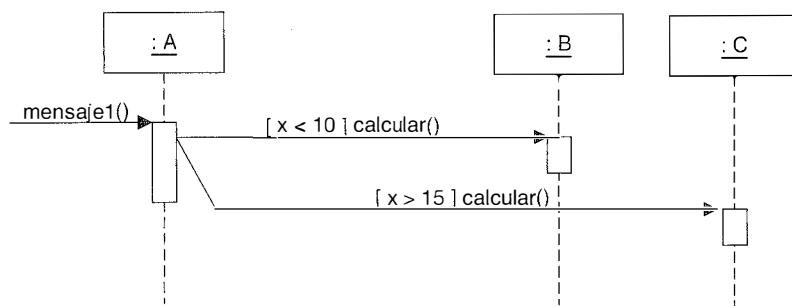


Figura 15.23. Mensajes condicionales mutuamente exclusivos.

Iteración para un único mensaje

La notación para la iteración de un mensaje se muestra en la Figura 15.24.

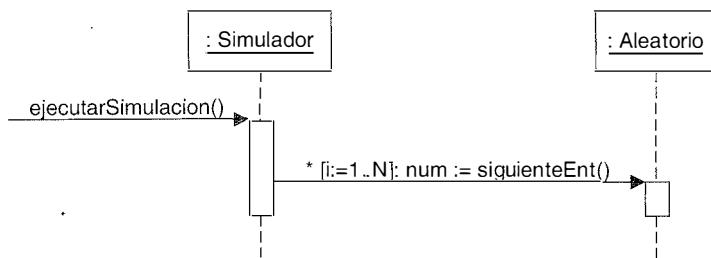


Figura 15.24. Iteración para un mensaje.

Iteración de una serie de mensajes

La Figura 15.25 muestra la notación para indicar la iteración alrededor de una serie de mensajes.

Iteración sobre una colección (multiobjeto)

En un diagrama de secuencia, la iteración sobre una colección se muestra en la Figura 15.26.

Con los diagramas de colaboración de UML, se especifica un marcador de multiplicidad '*' al final del rol (al lado del multiobjeto) para indicar el envío de un mensaje a cada elemento, en lugar de repetidamente a la propia colección. Sin embargo, UML no especifica cómo hacer esto con los diagramas de secuencia.

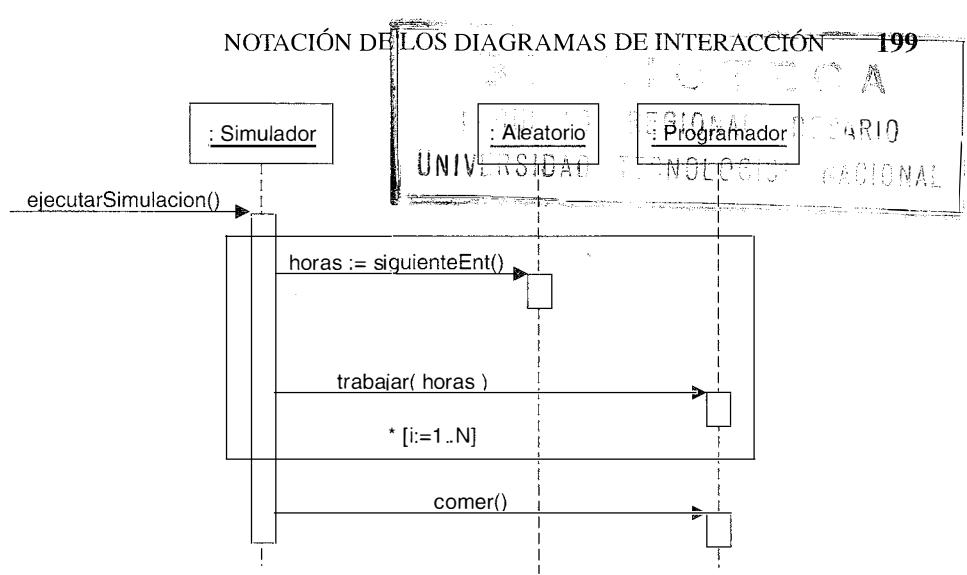


Figura 15.25. Iteración sobre una secuencia de mensajes.

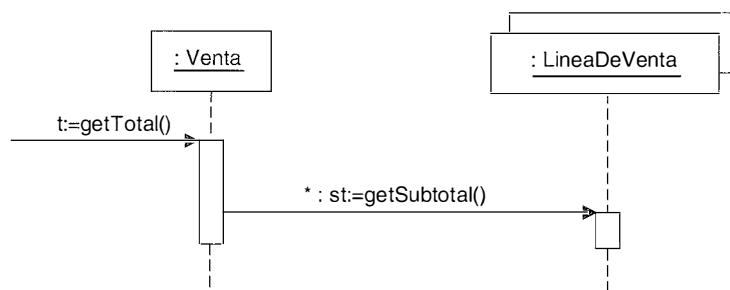


Figura 15.26. Iteración sobre un multiobjeto.

Mensajes a objetos de clase

Como en los diagramas de colaboración, las llamadas a los métodos de clase o estáticos se representan no subrayando el nombre del clasificador, lo que significa que se trata de una clase en lugar de una instancia (ver Figura 15.27).

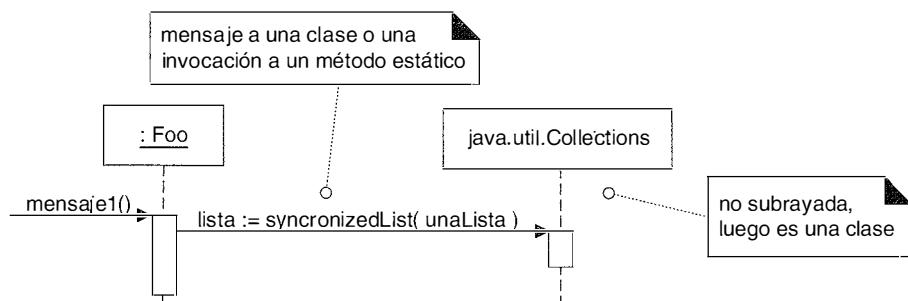


Figura 15.27. Invocación a un método de clase o estático.

GRASP: DISEÑO DE OBJETOS CON RESPONSABILIDADES

La forma más probable de que el mundo se destruya, coinciden la mayoría de los expertos, es por accidente. Aquí es donde entramos nosotros; somos profesionales informáticos. Nosotros provocamos accidentes.

Nathaniel Borenstein

Objetivos

- Definir los patrones.
 - Aprender a aplicar cinco de los patrones GRASP.
-

Introducción

El diseño de objetos se describe algunas veces como alguna variación de lo siguiente:

Después de la identificación de sus requisitos y la creación de un modelo del dominio, entonces añada métodos a las clases del software, y defina el paso de mensajes entre los objetos para satisfacer los requisitos.

Un consejo tan conciso no es especialmente útil, porque existen profundos principios y cuestiones involucrados en estas etapas. La decisión de qué métodos, dónde colocarlos y cómo deberían interactuar los objetos, es muy importante y nada trivial. Hace falta una explicación cuidadosa, aplicable mientras se realizan los diagramas y la programación.

Y esta es una etapa crítica; es la parte esencial de lo que entendemos por desarrollar un sistema orientado a objetos, no el dibujo de diagramas del modelo del dominio, diagramas de paquetes, etcétera.

GRASP como un enfoque sistemático para aprender el diseño de objetos básico

Es posible comunicar los principios detallados y el razonamiento que se requiere para entender el diseño de objetos básico, y aprender a aplicarlos en un enfoque sistemático que elimina la magia y la ambigüedad.

Los patrones GRASP constituyen un apoyo para la enseñanza que ayuda a uno a entender el diseño de objetos esencial, y aplica el razonamiento para el diseño de una forma sistemática, racional y explicable. Este enfoque para la comprensión y utilización de los principios de diseño se basa en los *patrones de asignación de responsabilidades*.

16.1. Responsabilidades y métodos

UML define una **responsabilidad** como “un contrato u obligación de un clasificador” [OMG01]. Las responsabilidades están relacionadas con las obligaciones de un objeto en cuanto a su comportamiento. Básicamente, estas responsabilidades son de los siguientes dos tipos:

- Conocer.
- Hacer.

Entre las responsabilidades de **hacer** de un objeto se encuentran:

- Hacer algo él mismo, como crear un objeto o hacer un cálculo.
- Iniciar una acción en otros objetos.
- Controlar y coordinar actividades en otros objetos.

Entre las responsabilidades de **conocer** de un objeto se encuentran:

- Conocer los datos privados encapsulados.
- Conocer los objetos relacionados.
- Conocer las cosas que puede derivar o calcular.

Las responsabilidades se asignan a las clases de los objetos durante el diseño de objetos. Por ejemplo, podría declarar que “una *Venta* es responsable de la creación de una *LíneaDeVenta*” (un hacer), o “una *Venta* es responsable de conocer su total” (un conocer). Las responsabilidades relevantes relacionadas con “conocer” a menudo se pueden inferir a partir del modelo del dominio, debido a los atributos y asociaciones que describe.

La granularidad de las responsabilidades influye en la conversión de las responsabilidades a clases y métodos. La responsabilidad de “proporcionar el acceso a bases de datos relacionales” podría implicar docenas de clases y cientos de métodos, empaquetados en un subsistema. En cambio, la responsabilidad de “crear una *Venta*” podría implicar sólo un método o unos pocos.

Una responsabilidad no es lo mismo que un método, pero los métodos se implementan para llevar a cabo responsabilidades. Las responsabilidades se implementan

utilizando métodos que o actúan solos o colaboran con otros métodos u objetos. Por ejemplo, la clase *Venta* podría definir uno o más métodos para conocer su total; digamos un método denominado *getTotal*¹. Para realizar esa responsabilidad, la *Venta* podría colaborar con otros objetos, por ejemplo, enviando el mensaje *getSubtotal* a cada objeto *LíneaDeVenta* solicitando su subtotal.

16.2. Responsabilidades y los diagramas de interacción

El objetivo de este capítulo es ayudar a aplicar sistemáticamente los principios fundamentales para asignar responsabilidades a los objetos. A menudo, se hará durante la programación. En los artefactos UML, un contexto habitual donde se tiene en cuenta estas responsabilidades (implementadas como métodos) es durante la creación de los diagramas de interacción (que forman parte del Modelo de Diseño del UP), cuya notación básica se examinó en el capítulo anterior.

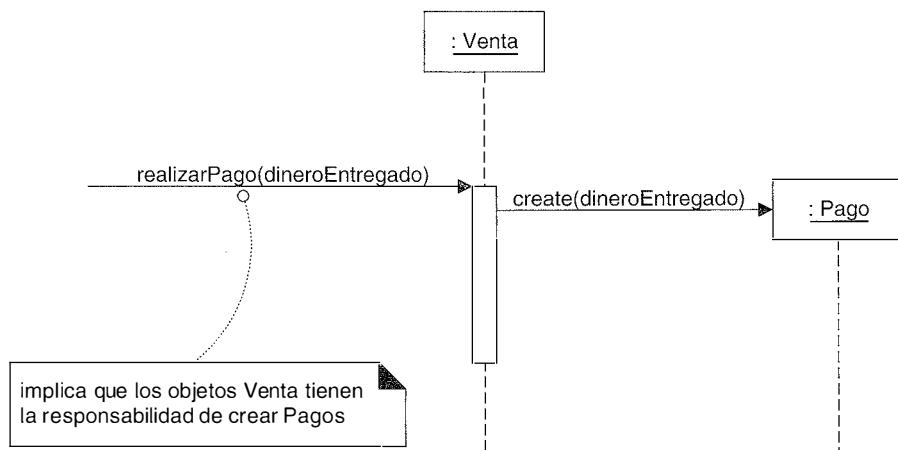


Figura 16.1. Las responsabilidades y los métodos están relacionados.

La Figura 16.1 indica que a los objetos *Venta* se les ha otorgado la responsabilidad de crear *Pagos*, que se invoca con el mensaje *realizarPago()* y se maneja con el correspondiente método *realizarPago*. Además, la realización de esta responsabilidad requiere colaboración para crear el objeto *LíneaDeVenta* e invocar a su constructor.

En resumen, los diagramas de interacción muestran elecciones en la asignación de responsabilidades a los objetos. Cuando se crean, se han tomado las decisiones acerca de la asignación de responsabilidades, lo que se refleja en los mensajes que se envían a diferentes clases de objetos. Este capítulo describe con detalle los principios fundamentales —expresados en los patrones GRASP— para guiar las elecciones sobre dónde asignar responsabilidades. Estas elecciones se reflejan en los diagramas de interacción.

¹ N. del T.: Para nombrar los métodos de una clase que retornan el valor de cierta propiedad denominada <X> de dicha clase, utilizaremos la habitual terminología *get<X>*, tal y como comenta el propio autor en el Capítulo 17.

16.3. Patrones

Los desarrolladores orientados a objetos con experiencia (y otros desarrolladores de software) acumulan un repertorio tanto de principios generales como de soluciones basadas en aplicar ciertos estilos que les guían en la creación de software. Estos principios y estilos, si se codifican con un formato estructurado que describa el problema y la solución, y se les da un nombre, podrían llamarse **patrones**. Por ejemplo, a continuación presentamos un patrón de muestra:

Nombre del patrón:	Experto en Información
Solución:	Asignar una responsabilidad a la clase que tiene la información necesaria para cumplirla.
Problema que resuelve:	¿Cuál es el principio básico mediante el cual asignaremos responsabilidades a los objetos?

En la tecnología de objetos, un **patrón** es una descripción de un problema y la solución, a la que se da un nombre, y que se puede aplicar a nuevos contextos; idealmente, proporciona consejos sobre el modo de aplicarlo en varias circunstancias, y considera los puntos fuertes y compromisos². Muchos patrones proporcionan guías sobre el modo en el que deberían asignarse las responsabilidades a los objetos, dada una categoría específica del problema.

De manera más simple, un **patrón** es un par problema/solución con nombre que se puede aplicar en nuevos contextos, con consejos acerca de cómo aplicarlo en nuevas situaciones y discusiones sobre sus compromisos.

“Un patrón de una persona es un bloque de construcción primitivo de otra” es un dicho de la tecnología de objetos que ilustra la ambigüedad de lo que puede ser un patrón [GHJV94]. Este tratamiento de los patrones pasará por alto el hecho de qué es apropiado etiquetar como un patrón, y se centrará en el valor pragmático de utilizar el estilo de los patrones como un medio para nombrar, presentar, aprender y recordar principios de ingeniería del software útiles.

Patrones repetitivos

Un *nuevo patrón* podría considerarse un oxímoron, si describe una idea nueva. El término “patrón” tiene la intención de sugerir algo repetitivo. Lo importante de los patrones no es expresar nuevas ideas de diseño. Es cierto justamente lo contrario —los patrones pretenden codificar conocimiento, estilos y principios *existentes* y que se han probado que son válidos—; cuanto más trillados y extendidos, mejor.

² La noción formal de patrones nació con los patrones arquitectónicos (de construcción) de Christofer Alexander [AIS77]. Los patrones de software se originaron en los ochenta con Kent Beck quien, llegó a ser consciente del trabajo con patrones de Alexander en arquitectura, y entonces los desarrolló junto con Ward Cunningham [BC87, Beck94].

En consecuencia, los patrones GRASP —que pronto se introducirán— no establecen nuevas ideas: son una codificación de principios básicos ampliamente utilizados. Para un experto en objetos, los patrones GRASP —por su idea, y no por su nombre— parecerán muy elementales y familiares. ¡Eso es lo importante!

Los patrones tienen nombres

Todos los patrones, idealmente, tienen nombres sugerentes. El asignar un nombre a un patrón, técnica o principio tiene las siguientes ventajas:

- Apoya la identificación e incorporación de ese concepto en nuestro conocimiento y memoria.
- Facilita la comunicación.

Asignar un nombre a una idea compleja como un patrón es un ejemplo del poder de la abstracción —reducción de una forma compleja a una simple eliminando detalles—. Por tanto, los patrones GRASP tienen nombres concisos como *Experto en Información*, *Creador*, *Variaciones Protegidas*.

La asignación de nombres a los patrones mejora la comunicación

Cuando se asigna un nombre a un patrón, podemos discutir con otros un principio complejo o una idea de diseño con un nombre sencillo. Considere la siguiente conversación entre dos diseñadores de software, que utilizan un vocabulario de patrones común (*Creador*; *Factoría*, etcétera), para tomar una decisión sobre un diseño:

Alfredo: “¿Dónde crees que deberíamos colocar la responsabilidad de crear una *LíneaDeVenta*? Creo en que una *Factoría*.”

Teresa: “Siguiendo el *Creador*, creo que la *Venta* sería adecuada.”

Alfredo: “Oh, correcto. Estoy de acuerdo.”

Identificar los estilos y principios de diseño con nombres sobreentendidos comúnmente facilita la comunicación y eleva el nivel de la investigación a un grado de abstracción más alto.

16.4. GRASP: Patrones de Principios Generales para Asignar Responsabilidades

Resumiendo la introducción anterior:

- La asignación habiliosa de responsabilidades es extremadamente importante en el diseño de objetos.
- La decisión acerca de la asignación de responsabilidades, a menudo, tiene lugar durante la creación de los diagramas de interacción y con seguridad durante la programación.

- Los patrones son pares problema/solución con un nombre que codifican buenos consejos y principios relacionados con frecuencia con la asignación de responsabilidades.

Pregunta: ¿Qué son los patrones GRASP?

Respuesta: Describen los principios fundamentales del diseño de objetos y la asignación de responsabilidades, expresados como patrones.

Es importante entender y ser capaces de aplicar estos principios durante la creación de los diagramas de interacción porque un desarrollador de software con poca experiencia en la tecnología de objetos necesita dominar estos principios tan rápido como sea posible; constituyen la base de cómo se diseñará el sistema.

GRASP es un acrónimo de **G**eneral **R**esponsibility **A**signment **S**oftware **P**atterns (patrones generales de software para asignar responsabilidades)³. El nombre se eligió para sugerir la importancia de *aprehender* (*grasping* en inglés) estos principios para diseñar con éxito el software orientado a objetos.

Cómo aplicar los patrones GRASP

Las secciones siguientes presentan los primeros cinco patrones GRASP:

- Experto en Información.
- Creador.
- Alta Cohesión.
- Bajo Acoplamiento.
- Controlador.

Hay otros, que se introducirán en un capítulo posterior, pero merece la pena dominar estos cinco primeros porque se refieren a cuestiones muy básicas, comunes y a aspectos fundamentales del diseño.

Por favor, estudie los siguientes patrones, observe cómo se utilizan en los diagramas de interacción de ejemplo, y entonces aplíquelos durante la creación de nuevos diagramas de interacción. Comience dominando *Experto en Información*, *Creador*, *Controlador*, *Alta Cohesión* y *Bajo Acoplamiento*. Después aprenda los patrones restantes.

16.5. Notación del diagrama de clases UML

Un rectángulo de clase de UML, que se utiliza para representar las clases software, normalmente muestra tres compartimentos; el tercero representa los métodos de la clase, como se muestra en la Figura 16.2.

³ Técnicamente deberíamos escribir “Patrones GRAS” en lugar de “Patrones GRASP”, pero lo segundo suena mejor.

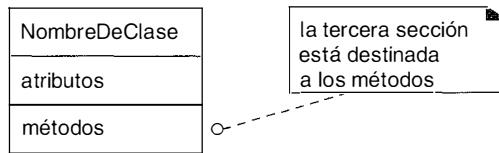


Figura 16.2. Las clases software muestran los nombres de los métodos.

Los detalles de esta notación se estudian en un capítulo siguiente. En la siguiente discusión sobre patrones, se utilizará ocasionalmente esta forma de rectángulo de clase.

16.6. Experto en Información (o Experto)

Solución Asignar una responsabilidad al experto en información —la clase que tiene la *información* necesaria para realizar la responsabilidad—.

Problema ¿Cuál es un principio general para asignar responsabilidades a los objetos?

Un Modelo de Diseño podría definir cientos o miles de clases software, y una aplicación podría requerir que se realicen cientos o miles de responsabilidades. Durante el diseño de objetos, cuando se definen las interacciones entre los objetos, tomamos decisiones sobre la asignación de responsabilidades a las clases software. Si se hace bien, los sistemas tienden a ser más fáciles de entender, mantener y ampliar, y existen más oportunidades para reutilizar componentes en futuras aplicaciones.

Ejemplo En la aplicación del PDV NuevaEra, algunas clases necesitan conocer el total de una venta.

Comience asignando responsabilidades estableciendo claramente la responsabilidad.

Siguiendo este consejo, la pregunta es:

¿Quién debería ser el responsable de conocer el total de una venta?

Siguiendo el *Experto en Información*, deberíamos buscar las clases de objetos que contienen la información necesaria para determinar el total.

Ahora llegamos a una pregunta clave: ¿miramos en el Modelo del Dominio o en Modelo de Diseño para analizar las clases que tienen la información necesaria? El Modelo del Dominio representa clases conceptuales del dominio del mundo real; el Modelo de Diseño representa clases software.

Respuesta:

1. Si hay clases relevantes en el Modelo del Diseño, mire ahí primero.
2. Si no, mire en el Modelo del Dominio, e intente utilizar (o ampliar) sus representaciones para inspirar la creación de las correspondientes clases de diseño.

Por ejemplo, asuma que acabamos de comenzar el trabajo de diseño y no hay más que un Modelo de Diseño mínimo. Por tanto, miramos en el Modelo del Dominio buscando expertos en información; quizás la *Venta* del mundo real es uno. Entonces, aña-

dimos una clase software al Modelo de Diseño denominada, igualmente, *Venta*, y le otorgamos la responsabilidad de conocer su total, representado con el método llamado *getTotal*. Este enfoque mantiene un *salto en la representación bajo* en el que el diseño de los objetos software se corresponde con nuestra concepción sobre cómo se organiza el mundo real.

Para examinar este caso en detalle, considere el Modelo del Dominio parcial de la Figura 16.3.

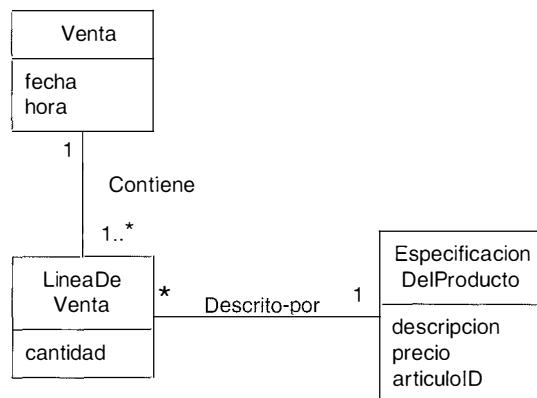


Figura 16.3. Asociaciones de Venta.

¿Qué información se necesita para determinar el total? Es necesario conocer todas las instancias de *LíneaDeVenta* de una venta y la suma de sus subtotales. Una instancia de *Venta* las contiene; por tanto, por la guía del Experto en Información, la clase de objeto *Venta* es adecuada para esta responsabilidad; es un *experto en información* para el trabajo.

Como se mencionó, es en el contexto de la creación de los diagramas de interacción cuando surgen estas cuestiones de responsabilidad. Imagine que estamos comenzando a trabajar con detalle en la elaboración de los diagramas de interacción para asignar responsabilidades a los objetos. Los diagramas de interacción y de clases parciales de la Figura 16.4 ilustran algunas decisiones.

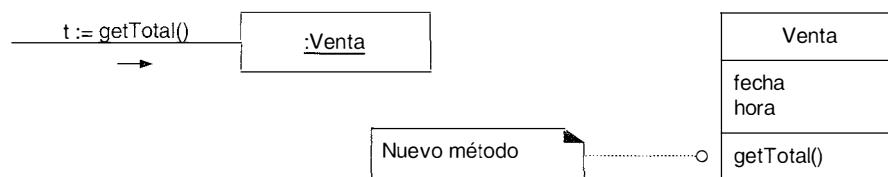


Figura 16.4. Diagramas de interacción y de clases parciales.

Todavía no hemos terminado. ¿Qué información se necesita para determinar el subtotal de la línea de venta? Se necesitan *LíneaDeVenta.cantidad* y *EspecificacionDelProducto.precio*. La *LíneaDeVenta* conoce su cantidad y la *EspecificacionDelProducto* asociada; por tanto, siguiendo el patrón Experto, la *LíneaDeVenta* debería determinar el subtotal; es el *experto en información*.

En cuanto a los diagramas de interacción, esto significa que la *Venta* necesita enviar el mensaje *getSubtotal* a cada una de las *LíneaDeVenta* y sumar el resultado; este diseño se muestra en la Figura 16.5.

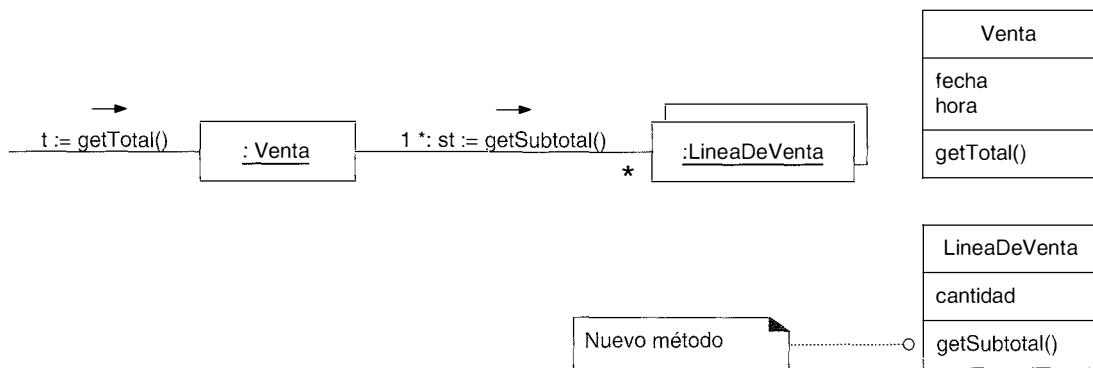


Figura 16.5. Cálculo del total de la Venta.

Una *LíneaDeVenta* para realizar la responsabilidad de conocer y proporcionar su subtotal necesita conocer el precio del artículo.

La *EspecificacionDelProducto* es un experto en información que proporciona su precio; por tanto, se le debe enviar un mensaje solicitando su precio.

El diseño se muestra en la Figura 16.6.

En conclusión, para realizar la responsabilidad de conocer y proporcionar el total de una venta, se asignaron tres responsabilidades a tres clases de diseño de objetos como sigue.

Clase de Diseño	Responsabilidad
Venta	Conocer el total de la venta.
LíneaDeVenta	Conocer el subtotal de la línea de venta.
EspecificacionDelProducto	Conocer el precio del artículo.

El contexto en el que se consideraron y optaron por estas responsabilidades fue durante la elaboración de un diagrama de interacción. La sección de métodos de un diagrama de clases puede entonces incluir los métodos.

El principio mediante el que se asignó cada responsabilidad fue el Experto en Información —colocándola en el objeto que tiene la información necesaria para realizarla—.

Discusión

El Experto en Información se utiliza con frecuencia en la asignación de responsabilidades; es un principio de guía básico que se utiliza continuamente en el diseño de objetos. El Experto no pretende ser una idea oscura o extravagante; expresa la “intuición” común de que los objetos hacen las cosas relacionadas con la información que tienen.

Nótese que el cumplimiento de la responsabilidad a menudo requiere información que se encuentra dispersa por diferentes clases de objetos. Esto implica que hay muchos expertos en información “parcial” que colaborarán en la tarea. Por ejemplo, el problema del total de las ventas al final requiere la colaboración de tres clases de objetos. Cada vez

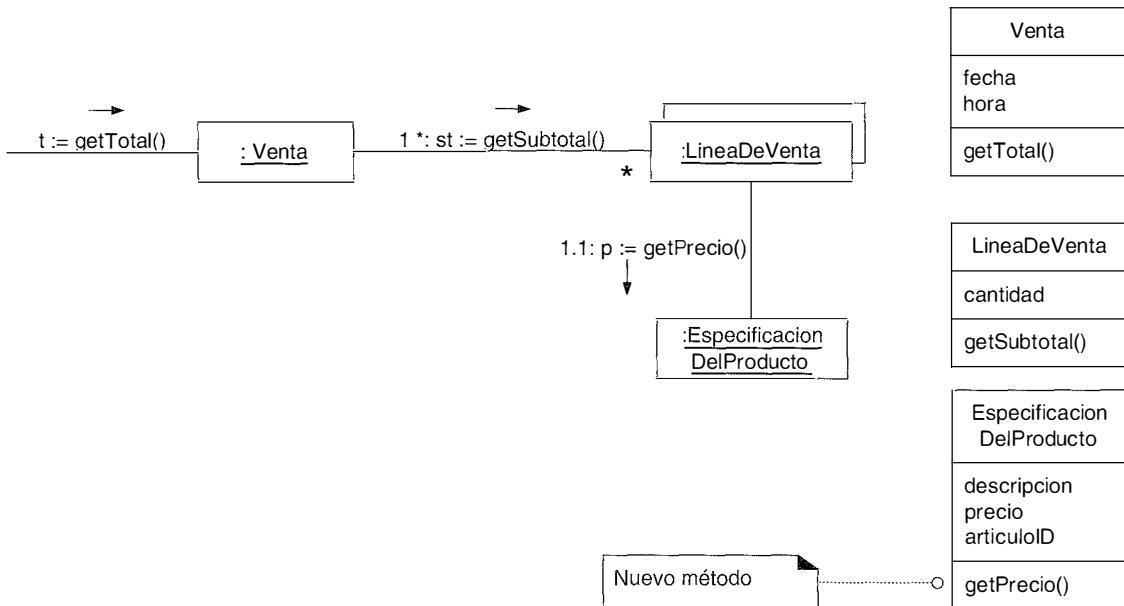


Figura 16.6. Cálculo del total de una *Venta*.

que la información se encuentre esparcida por objetos diferentes, necesitarán interactuar mediante el paso de mensajes para compartir el trabajo.

Por lo general, el Experto nos conduce a diseños donde los objetos del software realizan aquellas operaciones que normalmente se hacen a los objetos inanimados del mundo real que representan; Peter Coad denomina a esto la estrategia “Hacerlo yo mismo” [Coad95]. Por ejemplo, en el mundo real, sin el uso de ayudas mecánicas, una venta no te dice su total; es un objeto inanimado. Alguien calcula el total de la venta. Pero en el campo del software orientado a objetos, todos los objetos software están “vivos” o “animados”, y pueden tener responsabilidades y hacer cosas. Fundamentalmente, hacen cosas relacionadas con la información que conocen. Yo lo denomino el principio “de animación” en el diseño de objetos; es como estar en unos dibujos animados donde todo está vivo.

El patrón Experto en Información —como muchas cosas en la tecnología de objetos— tiene una analogía en el mundo real. Normalmente, otorgamos responsabilidades a los individuos con la información necesaria para llevar a cabo una tarea. Por ejemplo, en un negocio, ¿quién debería ser el responsable de la creación de una declaración de ganancias y pérdidas? La persona que tiene acceso a toda la información necesaria para crearla —quizás el director financiero—. Y, del mismo modo en que los objetos colaboran porque la información se encuentra dispersa, así pasa con las personas. El director financiero de la compañía podría solicitar a los contables que generen informes sobre los créditos y débitos.

Contraindicaciones

En algunas ocasiones la solución que sugiere el Experto no es deseable, normalmente debido a problemas de acoplamiento y cohesión (estos principios se discutirán más tarde en este capítulo).

Por ejemplo, ¿quién debería ser el responsable de almacenar una *Venta* en una base de datos? Ciertamente, mucha de la información que se tiene que almacenar se encuen-

tra en el objeto *Venta* y, por tanto, siguiendo el patrón Experto se podría argumentar la inclusión de la responsabilidad en la clase *Venta*. La extensión lógica de esta decisión es que cada clase contiene sus propios servicios para almacenarse ella misma en una base de datos. Pero esto nos lleva a problemas de cohesión, acoplamiento y duplicación. Por ejemplo, la clase *Venta* debe ahora contener la lógica relacionada con la gestión de la base de datos, como la relacionada con SQL y JDBC (Conexión de Base de Datos de Java, *Java Database Connectivity*). La clase ya no está centrada únicamente en la lógica de la aplicación de “ser una venta” simplemente; ahora tiene otro tipo de responsabilidades, lo que disminuye su cohesión. La clase debe acoplarse a los servicios técnicos de base de datos de otro subsistema, como los servicios JDBC, en lugar de acoplarse únicamente con otros objetos en la capa del dominio de los objetos del software, lo que eleva su acoplamiento. Y es probable que se dupliquen lógicas de bases de datos similares en muchas clases persistentes.

Todos estos problemas indican la violación de un principio arquitectural básico: diseñe separando los principales aspectos del sistema. Mantenga la lógica de la aplicación en un sitio (como en los objetos software del dominio), mantenga la lógica de la base de datos en otro sitio (como en un subsistema de servicios de persistencia separado), y así sucesivamente, en lugar de entremezclar aspectos del sistema diferentes en el mismo componente⁴.

Separando los aspectos importantes se mejora el acoplamiento y la cohesión del diseño. Por tanto, aunque siguiendo el Experto se podría justificar la asignación de la responsabilidad para el servicio de la base de datos a la clase *Venta*, por otros motivos (normalmente cohesión y acoplamiento), resulta un diseño pobre.

Beneficios

- Se mantiene el encapsulamiento de la información, puesto que los objetos utilizan su propia información para llevar a cabo las tareas. Normalmente, esto conlleva un bajo acoplamiento, lo que da lugar a sistemas más robustos y más fáciles de mantener. (Bajo Acoplamiento también es un patrón GRASP que se estudiará en una de las secciones siguientes.)
- Se distribuye el comportamiento entre las clases que contienen la información requerida, por tanto, se estimula las definiciones de clases más cohesivas y “ligeras” que son más fáciles de entender y mantener. Se soporta normalmente una alta cohesión (otro patrón que se estudiará más tarde).

Patrones o Principios Relacionados

- Bajo Acoplamiento.
- Alta Cohesión.

También conocido como; Parecido a

“Colocar las responsabilidades con los datos”, “Eso que conoces, hazlo”, “Hacerlo yo mismo”, “Colocar los Servicios con los Atributos con los que trabaja”.

16.7. Creador

Solución

Asignar a la clase B la responsabilidad de crear una instancia de clase A si se cumple uno o más de los casos siguientes:

⁴ En el Capítulo 32 se presentará una discusión sobre la separación de intereses.

- 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).

B es un *creador* de los objetos A.

Si se puede aplicar más de una opción, inclínese por una clase B que *agregue* o *contenga* la clase A.

Problema

¿Quién debería ser el responsable de la creación de una nueva instancia de alguna clase?

La creación de instancias es una de las actividades más comunes en un sistema orientado a objetos. En consecuencia, es útil contar con un principio general para la asignación de las responsabilidades de creación. Si se asignan bien, el diseño puede sopor- tar un bajo acoplamiento, mayor claridad, encapsulación y reutilización.

Ejemplo

En la aplicación del PDV, ¿quién debería ser el responsable de la creación de una instancia de *LíneaDeVenta*? Según el patrón Creador, deberíamos buscar las clases que agregan, contienen, etcétera, instancias de *LíneaDeVenta*. Considere el modelo del do- minio parcial de la Figura 16.7.

Puesto que una *Venta* contiene (de hecho, agrega) muchos objetos de *LíneaDeVenta*, el patrón Creador sugiere que *Venta* es un buen candidato para tener la responsabilidad de la creación de las instancias de *LíneaDeVenta*.

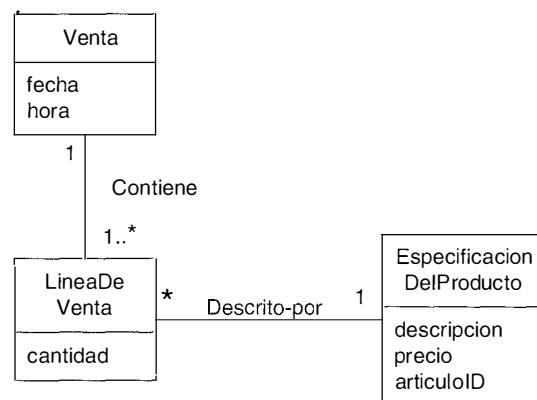


Figura 16.7. Modelo del Dominio Parcial.

Esto nos lleva al diseño de las interacciones entre los objetos que se muestran en la Figura 16.8.

Esta asignación de responsabilidades requiere que se defina en la clase *Venta* un mé- todo *crearLineaDeVenta*.

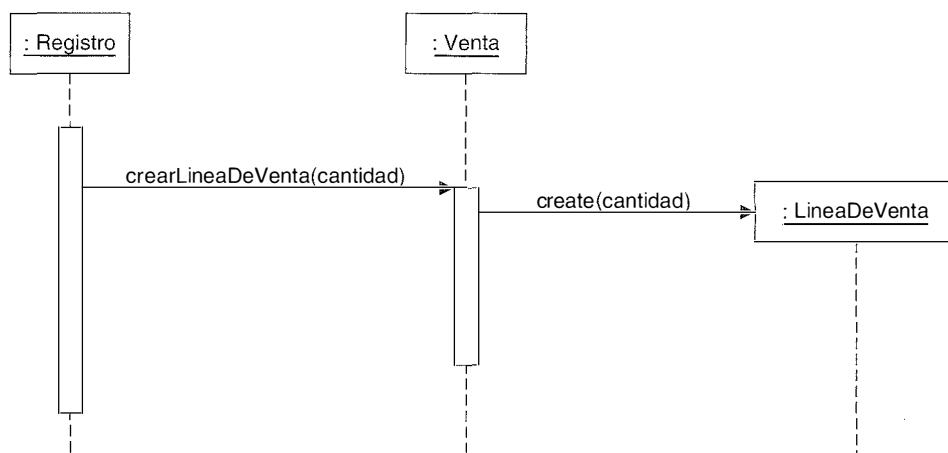


Figura 16.8. Creación de una LineaDeVenta.

Una vez más, el contexto en el que se consideraron y se optaron por estas responsabilidades fue durante la elaboración de un diagrama de interacción. La sección de los métodos de un diagrama de clases puede mostrar los resultados de la asignación de responsabilidades, materializadas concretamente como métodos.

Discusión

El patrón Creador guía la asignación de responsabilidades relacionadas con la creación de objetos, una tarea muy común. La intención básica del patrón Creador es encontrar un creador que necesite conectarse al objeto creado en alguna situación. Eliéndolo como el creador se favorece el bajo acoplamiento.

El Agregado *agrega* Partes, el Contenedor *contiene* Contenido, y el Registro *registra* Registros, son todas ellas relaciones comunes entre las clases en un diagrama de clases. El Creador sugiere que la clase contenedor o registro es una buena candidata para asignarle la responsabilidad de crear lo que contiene o registra. Por su puesto, esto es sólo una guía.

Nótese que se ha utilizado el concepto de **agregación** al considerar el patrón Creador. La agregación se examinará en el Capítulo 27; una definición breve es que una agregación involucra cosas que se encuentran en una relación Todo-Parte o Ensamblaje-Parte, como un Cuerpo agrega Pierna o un Párrafo agrega Frase.

Algunas veces se encuentra un creador buscando las clases que tienen los datos de inicialización que se pasarán durante la creación. Se trata, en realidad, de un ejemplo del patrón Experto. Los datos de inicialización se pasan durante la creación por medio de algún tipo de método de inicialización, como un constructor Java que tiene parámetros. Por ejemplo, asuma que una instancia de *Pago* necesita inicializarse, cuando se crea, con el total de la *Venta*. Puesto que la *Venta* conoce el total, la *Venta* es un creador candidato del *Pago*.

Contraindicaciones

A menudo, la creación requiere una complejidad significativa, como utilizar instancias recicladas por motivos de rendimiento, crear condicionalmente una instancia a partir de una familia de clases similares basado en el valor de alguna propiedad externa, etcétera. En estos casos, es aconsejable delegar la creación a una clase auxiliar denominada *Factoría* [GHJV95] en lugar de utilizar la clase que sugiere el *Creador*. Las factorías se estudiarán en el Capítulo 23.

Beneficios	<ul style="list-style-type: none"> • Se soporta bajo acoplamiento (descrito a continuación), lo que implica menos dependencias de mantenimiento y mayores oportunidades para reutilizar. Probablemente no se incrementa el acoplamiento porque la clase <i>creada</i> es presumible que ya sea visible a la clase <i>creadora</i>, debido a las asociaciones existentes que motivaron su elección como creador.
Patrones o Principios Relacionados	<ul style="list-style-type: none"> • Bajo Acoplamiento. • Factoría. • Todo-Parte [BMRSS96] describe un patrón para definir objetos agregados que favorece la encapsulación de componentes.

16.8. Bajo Acoplamiento

Solución	Asignar una responsabilidad de manera que el acoplamiento permanezca bajo.
Problema	¿Cómo soportar bajas dependencias, bajo impacto del cambio e incremento de la reutilización?

El **acoplamiento** es una medida de la fuerza con que un elemento está conectado a, tiene conocimiento de, confía en, otros elementos. Un elemento con bajo (o débil) acoplamiento no depende de demasiados otros elementos; “demasiados” depende del contexto, pero se estudiará. Estos elementos pueden ser clases, subsistemas, sistemas, etcétera.

Una clase con alto (o fuerte) acoplamiento confía en muchas otras clases. Tales clases podrían no ser deseables; algunas adolecen de los siguientes problemas:

- Los cambios en las clases relacionadas fuerzan cambios locales.
- Son difíciles de entender de manera aislada.
- Son difíciles de reutilizar puesto que su uso requiere la presencia adicional de las clases de las que depende.

Ejemplo Considere el siguiente diagrama de clases parcial del caso de estudio NuevaEra:



Asuma que tenemos la necesidad de crear una instancia de *Pago* y asociarla con la *Venta*. ¿Qué clase debería ser la responsable de esto? Puesto que un *Registro* “registra” un *Pago* en el dominio del mundo real, el patrón Creador sugiere el *Registro* como candidata para la creación del *Pago*. La instancia de *Registro* podría enviar entonces el mensaje *añadirPago* a la *Venta*, pasando el nuevo *Pago* como parámetro. La Figura 16.9 muestra un posible diagrama de interacción que refleja esto.

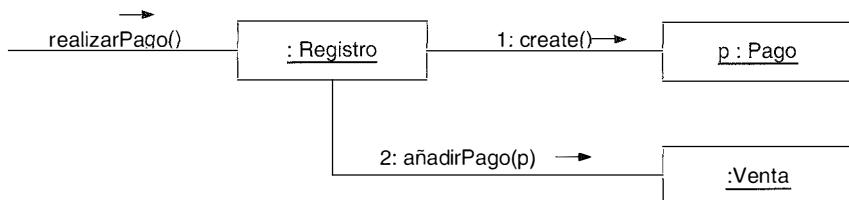


Figura 16.9. El Registro crea un Pago.

Esta asignación de responsabilidades acopla la clase *Registro* con el conocimiento de la clase *Pago*.

Notación UML: Nótese que la instancia de *Pago* se nombra explícitamente con una *p* de manera que se puede referenciar como parámetro en el mensaje 2.

La Figura 16.10 muestra una solución alternativa a crear el *Pago* y asociarlo con la *Venta*.

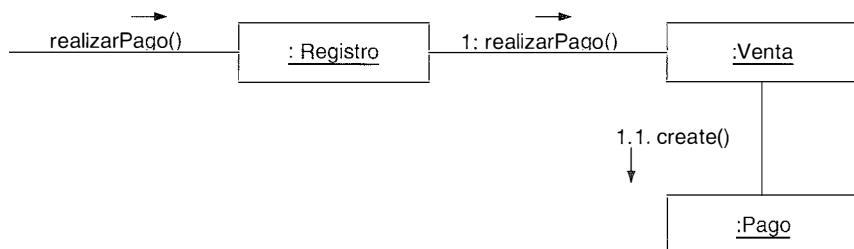


Figura 16.10. La Venta crea el Pago.

¿Qué diseño, basado en la asignación de responsabilidades, soporta Bajo Acoplamiento? En ambos casos asumiremos que la *Venta* debe finalmente acoplarse con el conocimiento del *Pago*. El Diseño 1, en el que el *Registro* crea el *Pago*, añade acoplamiento entre el *Registro* y el *Pago*, mientras que el Diseño 2, en el que la *Venta* lleva a cabo la creación del *Pago*, no incrementa el acoplamiento. Desde el punto de vista puramente del acoplamiento, es preferible el Diseño 2 porque mantiene el acoplamiento global más bajo. Éste es un ejemplo en el que dos patrones —Bajo Acoplamiento y Creador— podrían sugerir soluciones diferentes.

En la práctica, el nivel de acoplamiento no se puede considerar de manera aislada a otros principios como el Experto o Alta Cohesión. Sin embargo, es un factor a tener en cuenta para mejorar un diseño.

Discusión

El patrón de Bajo Acoplamiento es un principio a tener en mente en todas las decisiones de diseño; es un objetivo subyacente a tener en cuenta continuamente. Es un **principio evaluativo** que aplica un diseñador mientras evalúa todas las decisiones de diseño.

En los lenguajes orientados a objetos como C++, Java y C#, algunas de las formas comunes de acoplamiento entre el *TipoX* y el *TipoY* son:

- El *TipoX* tiene un atributo (miembro de datos, o variable de instancia) que hace referencia a una instancia de *TipoY*, o al propio *TipoY*.
- Un objeto de *TipoX* invoca los servicios de un objeto de *TipoY*.
- El *TipoX* tiene un método que referencia a una instancia de *TipoY*, o al propio *TipoY*, de algún modo. Esto, generalmente, comprende un parámetro o variable local de *TipoY*, o que el objeto de retorno de un mensaje sea una instancia de *TipoY*.
- El *TipoX* es una subclase, directa o indirecta, del *TipoY*.
- El *TipoY* es una interfaz y el *TipoX* implementa esa interfaz.

El patrón Bajo Acoplamiento impulsa la asignación de responsabilidades de manera que su localización no incremente el acoplamiento hasta un nivel que nos lleve a los resultados negativos que puede producir un acoplamiento alto.

El Bajo Acoplamiento soporta el diseño de clases que son más independientes, lo que reduce el impacto del cambio. No se puede considerar de manera aislada a otros patrones como el Experto o el de Alta Cohesión, sino que necesita incluirse como uno de los diferentes principios de diseño que influyen en una elección al asignar una responsabilidad.

Una subclase está fuertemente acoplada a su superclase. Se debe estudiar cuidadosamente la decisión de derivar a partir de una superclase debido a esta forma fuerte de acoplamiento. Por ejemplo, suponga que necesitamos almacenar los objetos de manera persistente en una base de datos relacional u objetual. En este caso, un diseño relativamente común consiste en crear una superclase abstracta denominada *ObjetoPersistente* a partir de la cual derivan otras clases. El inconveniente de esta creación de subclases es que acopla altamente los objetos del dominio a un servicio del nivel tecnológico particular y mezcla diferentes aspectos de la arquitectura, mientras que la ventaja es que se hereda de manera automática el comportamiento persistente.

No existe una medida absoluta de cuando el acoplamiento es demasiado alto. Lo que es importante es que un desarrollador pueda medir el grado de acoplamiento actual, y evaluar si aumentarlo le causará problemas. En general, las clases que son inherentemente muy genéricas por naturaleza, y con una probabilidad de reutilización alta, debería tener un acoplamiento especialmente bajo.

El caso extremo de Bajo Acoplamiento es cuando no existe acoplamiento entre clases. Esto no es deseable porque una metáfora central de la tecnología de objetos es un sistema de objetos conectados que se comunican mediante el paso de mensajes. Si el Bajo Acoplamiento se lleva al extremo, producirá un diseño pobre porque dará lugar a unos pocos objetos inconexos, saturados, y con actividad compleja que hacen todo el trabajo, con muchos objetos muy pasivos, sin acoplamiento que actúan como simples repositorios de datos. Es normal y necesario un grado moderado de acoplamiento entre las clases para crear un sistema orientado a objetos en el que las tareas se llevan a cabo mediante la colaboración de los objetos conectados.

Contraindicaciones

No suele ser un problema el acoplamiento alto entre objetos estables y elementos generalizados. Por ejemplo, una aplicación Java J2EE puede acoplarse con seguridad con las librerías de Java (*java.util*, etc.), porque son estables y extendidas.

Escoja sus batallas

El alto acoplamiento en sí no es un problema; es el alto acoplamiento entre elementos que no son estables en alguna dimensión, como su interfaz, implementación o su mera presencia.

Éste es un punto importante: como diseñadores, podemos añadir flexibilidad, encapsular detalles e implementaciones, y, en general, diseñar para disminuir el acoplamiento en muchas áreas del sistema. Pero, si nos esforzamos en “futuras necesidades” o en disminuir el acoplamiento en algunos puntos donde de hecho no hay motivos realistas, el tiempo no se está empleando de manera adecuada.

Los diseñadores tienen que escoger sus batallas para disminuir el acoplamiento y encapsular información. Centrándose en los puntos en los que sea realista pensar en una inestabilidad o evolución alta. Por ejemplo, en el proyecto NuevaEra, se sabe que se necesitan conectar al sistema diferentes sistemas calculadores de impuestos de terceras partes (con interfaces únicas). Por tanto, es práctico diseñar para disminuir el acoplamiento en este punto de variación.

Beneficios

- No afectan los cambios en otros componentes.
- Fácil de entender de manera aislada.
- Convenientes para reutilizar.

Antecedentes

El acoplamiento y la cohesión (descrita a continuación) son principios realmente fundamentales en el diseño y todos los desarrolladores de software deberían apreciarlos y aplicarlos por sí mismos. Larry Constantine, también un creador del diseño estructurado en los años setenta y que actualmente propugna una mayor atención a la ingeniería de usabilidad [CL99], fue el principal responsable en los sesenta de la identificación y comunicación del acoplamiento y la cohesión como principios básicos [Constantine68, CMS74].

Patrones Relacionados

- Variaciones Protegidas.

16.9. Alta Cohesión

Solución

Asignar una responsabilidad de manera que la cohesión permanezca alta.

Problema

¿Cómo mantener la complejidad manejable?

En cuanto al diseño de objetos, la **cohesión** (o de manera más específica, la cohesión funcional) es una medida de la fuerza con la que se relacionan y del grado de focalización de las responsabilidades de un elemento. Un elemento con responsabilidades altamente relacionadas, y que no hace una gran cantidad de trabajo, tiene alta cohesión. Estos elementos pueden ser clases, subsistemas, etcétera.

Una clase con baja cohesión hace muchas cosas no relacionadas, o hace demasiado trabajo. Tales clases no son convenientes; adolecen de los siguientes problemas:

- Difíciles de entender.
- Difíciles de reutilizar.
- Difíciles de mantener.
- Delicadas, constantemente afectadas por los cambios.

A menudo, las clases con baja cohesión representan un “grano grande” de abstracción, o se les han asignado responsabilidades que deberían haberse delegado en otros objetos.

Ejemplo

Podemos analizar para la Alta Cohesión el mismo ejemplo que se utilizó para el patrón de Bajo Acoplamiento.

Asumamos que necesitamos crear una instancia de *Pago* (en efectivo) y asociarla con la *Venta*. ¿Qué clase debería ser responsable de esto? Puesto que el *Registro* registra un *Pago* en el dominio del mundo real, el patrón *Creador* sugiere el *Registro* como candidato para la creación del *Pago*. La instancia de *Registro* podría entonces enviar un

mensaje *añadirPago* a la *Venta*, pasando como parámetro el nuevo *Pago*, como se muestra en la Figura 16.11.

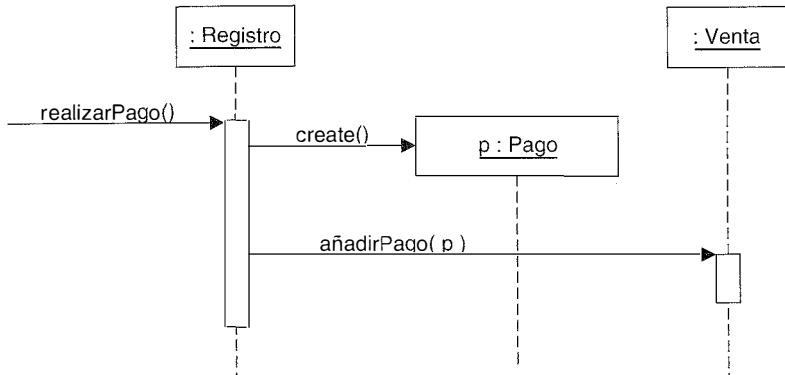


Figura 16.11. El Registro crea el Pago.

Esta asignación de responsabilidades sitúa la responsabilidad de realizar un pago en el *Registro*. El *Registro* toma parte en la responsabilidad de llevar a cabo la operación del sistema *realizarPago*.

En este ejemplo aislado, esto es aceptable; pero si continuamos haciendo responsable a la clase *Registro* de realizar parte o la mayoría del trabajo relacionado con más y más operaciones del sistema, se irá sobrecargando incrementalmente con tareas y llegará a perder la cohesión.

Imagine que hubiera cincuenta operaciones del sistema, todas recibidas por *Registro*. Si hace el trabajo relacionado con cada una, se convertirá en un objeto “saturado” y sin cohesión. El hecho no es que esta simple tarea de creación de un *Pago* en sí misma haga que el *Registro* no sea cohesivo, sino que como parte de un cuadro mayor de asignación de responsabilidades global, podría sugerir una tendencia a una baja cohesión.

Y más importante en cuanto a las habilidades de desarrollo como diseñadores de objetos, independientemente de la elección de diseño final, lo importante es que por lo menos un desarrollador sepa tener en cuenta el impacto de la cohesión.

En cambio, como se muestra en la Figura 16.12, el segundo diseño delega la responsabilidad de creación del pago a la *Venta*, que favorece una cohesión más alta en el *Registro*.

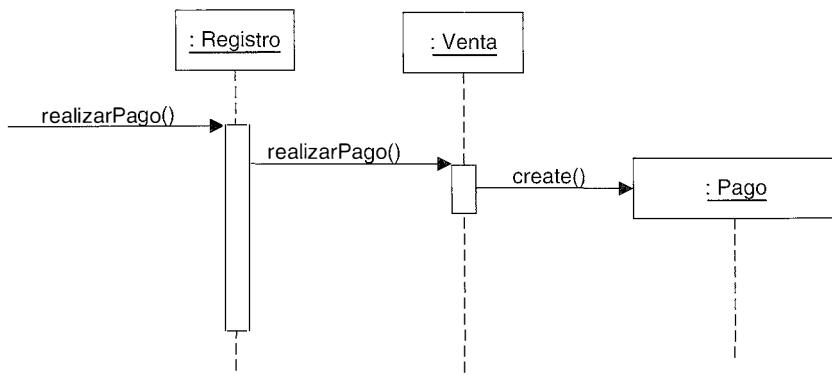


Figura 16.12. La Venta crea el Pago.

Es deseable el segundo diseño puesto que soporta tanto alta cohesión como bajo acoplamiento.

En la práctica, el nivel de cohesión no se puede considerar de manera aislada a otras responsabilidades y otros principios como los patrones Experto y Bajo Acoplamiento.

Discusión

Igual que el Bajo Acoplamiento, el patrón de Alta Cohesión es un principio a tener en mente durante todas las decisiones de diseño; es un objetivo subyacente a tener en cuenta continuamente. Es un principio evaluativo que aplica un diseñador mientras evalúa todas las decisiones de diseño.

Grady Booch establece que existe alta cohesión funcional cuando los elementos de un componente (como una clase) “trabajan todos juntos para proporcionar algún comportamiento bien delimitado” [Booch94].

A continuación presentamos algunos escenarios que ilustran diferentes grados de cohesión funcional:

1. *Muy baja cohesión.* Una única clase es responsable de muchas cosas en áreas funcionales muy diferentes.
 - Asuma que existe una clase denominada *Interfaz-BDR-RPC* que es completamente responsable de la interacción con bases de datos relacionales y la gestión de las llamadas remotas a procedimientos (*Remote Procedure Calls*). Éstas son dos áreas funcionales muy diferentes, y cada una requiere mucho código de soporte. La responsabilidad debería dividirse en una familia de clases relacionadas con el acceso BDR y una familia relacionada con el soporte de RPC.
2. *Baja cohesión.* Una única clase tiene la responsabilidad de una tarea compleja en un área funcional.
 - Asuma que existe una clase denominada *InterfazBDR* que es completamente responsable de interactuar con las bases de datos relacionales. Los métodos de la clase están todos relacionados, pero hay muchos, y una gran cantidad de código de soporte; podría haber cientos o miles de métodos. La clase debería dividirse en una familia de clases ligeras que comparten el trabajo de proporcionar el acceso BDR.
3. *Alta cohesión.* Una clase tiene una responsabilidad moderada en un área funcional y colabora con otras clases para llevar a cabo las tareas.
 - Asuma que existe una clase denominada *InterfazBDR* que es sólo parcialmente responsable de interactuar con las bases de datos relacionales. Interactúa con una docena de otras clases relacionadas con el acceso BDR para recuperar y almacenar objetos.
4. *Moderada cohesión.* Una clase tiene responsabilidades ligeras y únicas en unas pocas áreas diferentes que están lógicamente relacionadas con el concepto de la clase, pero no entre ellas.
 - Asuma que existe una clase denominada *Compañía* que es completamente responsable de (a) conocer a sus empleados y (b) conocer su información fi-

nanciera. Estas dos áreas no están fuertemente relacionadas entre ellas; aunque ambas están lógicamente relacionadas con el concepto de compañía. Además, el número total de métodos públicos es pequeño, al igual que la cantidad de código de soporte.

Como regla empírica, una clase con alta cohesión tiene un número relativamente pequeño de métodos, con funcionalidad altamente relacionada, y no realiza mucho trabajo. Colabora con otros objetos para compartir el esfuerzo si la tarea es extensa.

Una clase con alta cohesión es ventajosa porque es relativamente fácil de mantener, entender y reutilizar. El alto grado de funcionalidad relacionada, combinada con un número pequeño de operaciones, también simplifica el mantenimiento y las mejoras. El grano fino de la funcionalidad altamente relacionada también aumenta el potencial de reutilización.

El patrón de Alta Cohesión —como muchas cosas en la tecnología de objetos— tiene una analogía en el mundo real. Una observación típica es que si una persona tiene demasiadas responsabilidades no relacionadas —especialmente unas que deberían delegarse de manera adecuada en otras— entonces la persona no es efectiva. Se observa en algunos directores que no han aprendido a delegar. Estas personas padecen baja cohesión; están preparados para llegar a ser personas “despegadas”.

Otro principio clásico: diseño modular

El acoplamiento y la cohesión son viejos principios del diseño de software; diseñar con objetos no implica que se ignoren los fundamentos bien establecidos. Otro de estos principios —que está fuertemente relacionado con el acoplamiento y la cohesión— es promover el **diseño modular**. Citando textualmente:

La modularidad es la propiedad de un sistema que se ha descompuesto en un conjunto de módulos cohesivos y débilmente acoplados [Booch94].

Fomentamos el diseño modular creando métodos y clases con alta cohesión. Al nivel básico de objetos, la modularidad se alcanza diseñando cada método con un único y claro objetivo, y agrupando un conjunto de aspectos relacionados en una clase.

Cohesión y acoplamiento; el yin y el yang

Una mala cohesión causa, normalmente, un mal acoplamiento, y viceversa. Llamo a la cohesión y el acoplamiento el *yin y el yang de la ingeniería del software* debido a que influye el uno en el otro. Por ejemplo, considere una clase que implementa un elemento gráfico GUI que representa y pinta dicho elemento gráfico, almacena datos en una base de datos, e invoca el servicio de objetos remotos. No sólo es profundamente no cohesiva, sino que está acoplada a muchos (y con nada en común) elementos.

Contraindicaciones

Existen pocos casos en los que esté justificada la aceptación de baja cohesión.

Un caso es la agrupación de responsabilidades o código en una clase o componente para simplificar el mantenimiento por una persona —aunque queda advertido de que tal agrupación podría también empeorar el mantenimiento—. Pero por ejemplo, suponga

que una aplicación contiene sentencias SQL embebidas que siguiendo otros buenos principios de diseño deberían distribuirse por diez clases, tales como diez clases encargadas de la “conversión objeto-tupla de base de datos”. Ahora bien, es normal que sólo uno o dos expertos en SQL conozcan la mejor manera para definir y mantener este SQL, incluso si hay docenas de programadores orientados a objetos (OO) en el proyecto; pocos programadores OO podrían dominar SQL. Suponga que incluso el experto en SQL no se encuentra cómodo como programador OO. El arquitecto de software podría decidir agrupar todas las sentencias SQL en una clase, *OperacionesBDR*, de manera que sea fácil para el experto en SQL trabajar en SQL en un único lugar.

Otro caso de componentes con baja cohesión lo constituyen los objetos servidores distribuidos. Debido a implicaciones de costes fijos y rendimientos asociados con los objetos remotos y comunicaciones remotas, a veces, es deseable crear menos objetos servidores, de mayor tamaño y menos cohesivos que proporcionen una interfaz para muchas operaciones. Esto también está relacionado con el patrón denominado **Interfaz Remota de Grano Grueso**, en el cual, las operaciones remotas se hacen de grano más grueso con el objeto de realizar o solicitar más trabajo en las llamadas a operaciones remotas, debido a las penalizaciones de rendimiento de las llamadas remotas en la red. Como un simple ejemplo, en vez de un objeto remoto con tres operaciones de grano fino *setNombre*⁵, *setSalario*, y *setFechaAlquiler*, hay una operación remota *setDatos* que recibe un conjunto de datos. Esto da lugar a menos llamadas remotas y mejor rendimiento.

Beneficios

- Se incrementa la claridad y facilita la comprensión del diseño.
- Se simplifican el mantenimiento y las mejoras.
- Se soporta a menudo bajo acoplamiento.
- El grano fino de funcionalidad altamente relacionada incrementa la reutilización porque una clase cohesiva se puede utilizar para un propósito muy específico.

16.10. Controlador

Solución

Asignar la responsabilidad de recibir o manejar un mensaje de evento del sistema a una clase que representa una de las siguientes opciones:

- Representa el sistema global, dispositivo o subsistema (*controlador de fachada*).
- Representa un escenario de caso de uso en el que tiene lugar el evento del sistema, a menudo denominado <NombreDelCasoDeUso>Manejador, <NombreDelCasoDeUso>Coordinador o <NombreDelCasoDeUso>Sesion (*controlador de sesión o de caso de uso*).
 - Utilice la misma clase controlador para todos los eventos del sistema en el mismo escenario de caso de uso.
 - Informalmente, una sesión es una instancia de una conversación con un actor. Las sesiones pueden tener cualquier duración, pero se organizan a menudo en función de los casos de uso (sesiones de casos de uso).

⁵ *N. del T.:* Para nombrar los métodos de una clase que asignan un valor a una propiedad denominada <X>, utilizaremos la habitual terminología *set<X>*.

Corolario: Nótese que las clases “ventana”, “applet”, “widget”, “vista” y “documento” no están en esta lista. Tales clases *no* deberían abordar las tareas asociadas con los eventos del sistema, normalmente, reciben estos eventos y los delegan a un controlador.

Problema

¿Quién debe ser el responsable de gestionar un evento de entrada al sistema?

Un **evento del sistema** de entrada es un evento generado por un actor externo. Se asocian con **operaciones del sistema** —operaciones del sistema como respuesta a los eventos del sistema—, tal como se relacionan los mensajes y los métodos.

Por ejemplo, cuando un cajero utiliza un terminal PDV y presiona el botón “Finalizar Venta”, está generando un evento del sistema que indica que la “venta ha terminado”. Igualmente, cuando un escritor utiliza un procesador de texto y presiona el botón “comprobar ortografía”, está generando un evento del sistema que indica que se “ejecute una comprobación de la ortografía”.

Un **Controlador** es un objeto que no pertenece a la interfaz de usuario, responsable de recibir o manejar un evento del sistema. Un Controlador define el método para la operación del sistema.

Ejemplo

En la aplicación NuevaEra, hay varias operaciones del sistema, como se ilustra en la Figura 16.13, que muestran al propio sistema como una clase o componente (que es legal en UML).

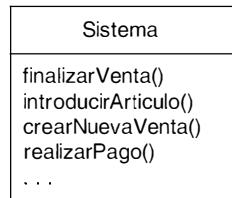


Figura 16.13. Operaciones del sistema asociadas con los eventos del sistema.

Durante el análisis, las operaciones del sistema pueden asignarse a la clase *Sistema*, para indicar que se trata de operaciones del sistema. Sin embargo, esto *no* significa que una clase software denominada *Sistema* las lleve a cabo durante el diseño. Más bien, durante el diseño, se asigna la responsabilidad de las operaciones del sistema a una clase Controlador (ver Figura 16.14).

¿Quién debería ser el controlador de los eventos del sistema como *introducirArticulo* y *finalizarVenta*?

Siguiendo el patrón Controlador, a continuación presentamos algunas de las opciones:

representa el “sistema” global, *Registro, SistemaPDV*
dispositivo o subsistema

representa un receptor o manejador
de todos los eventos del sistema
de un escenario de caso de uso *ProcesarVentaManejador*
ProcesarVentaSesion

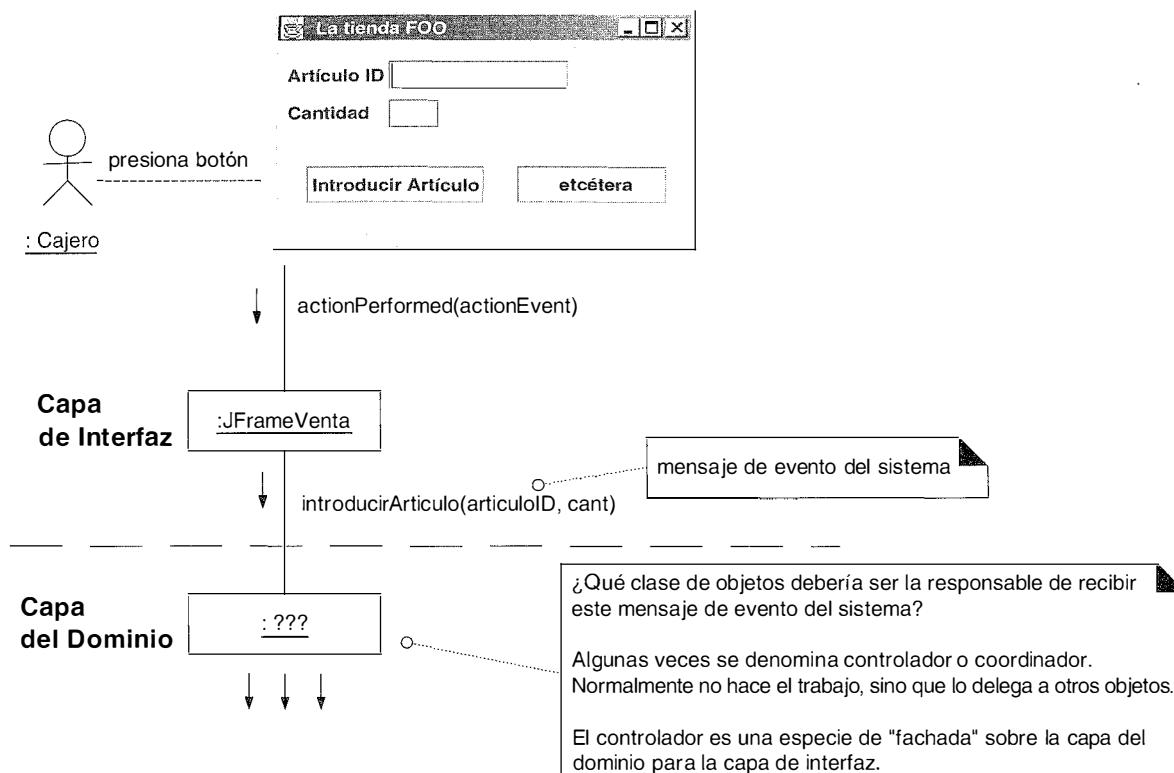


Figura 16.14. ¿Controlador para introducirArticulo?

En cuanto a los diagramas de interacción, significa que uno de los ejemplos de la Figura 16.15 podría ser útil.

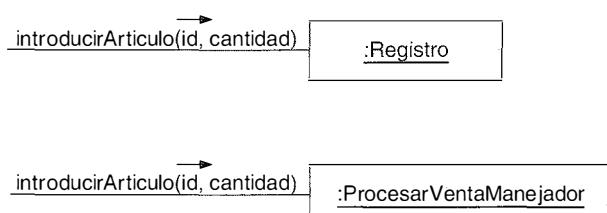


Figura 16.15. Opciones de Controlador.

En la elección de cuál de las clases es el controlador más apropiado influyen otros factores, que se estudiarán en la siguiente sección.

Durante el diseño, se asignan a una o más clases controlador las operaciones del sistema que se identificaron durante el análisis del comportamiento del sistema, como *Registro*, tal como se muestra en la Figura 16.16.

Discusión

Los sistemas reciben eventos de entrada externos, normalmente a través de una GUI manejada por una persona. Otros medios de entrada pueden ser mensajes externos, como en un comutador de telecomunicaciones para el procesamiento de llamadas, o señales desde sensores, como en sistemas de control de procesos.

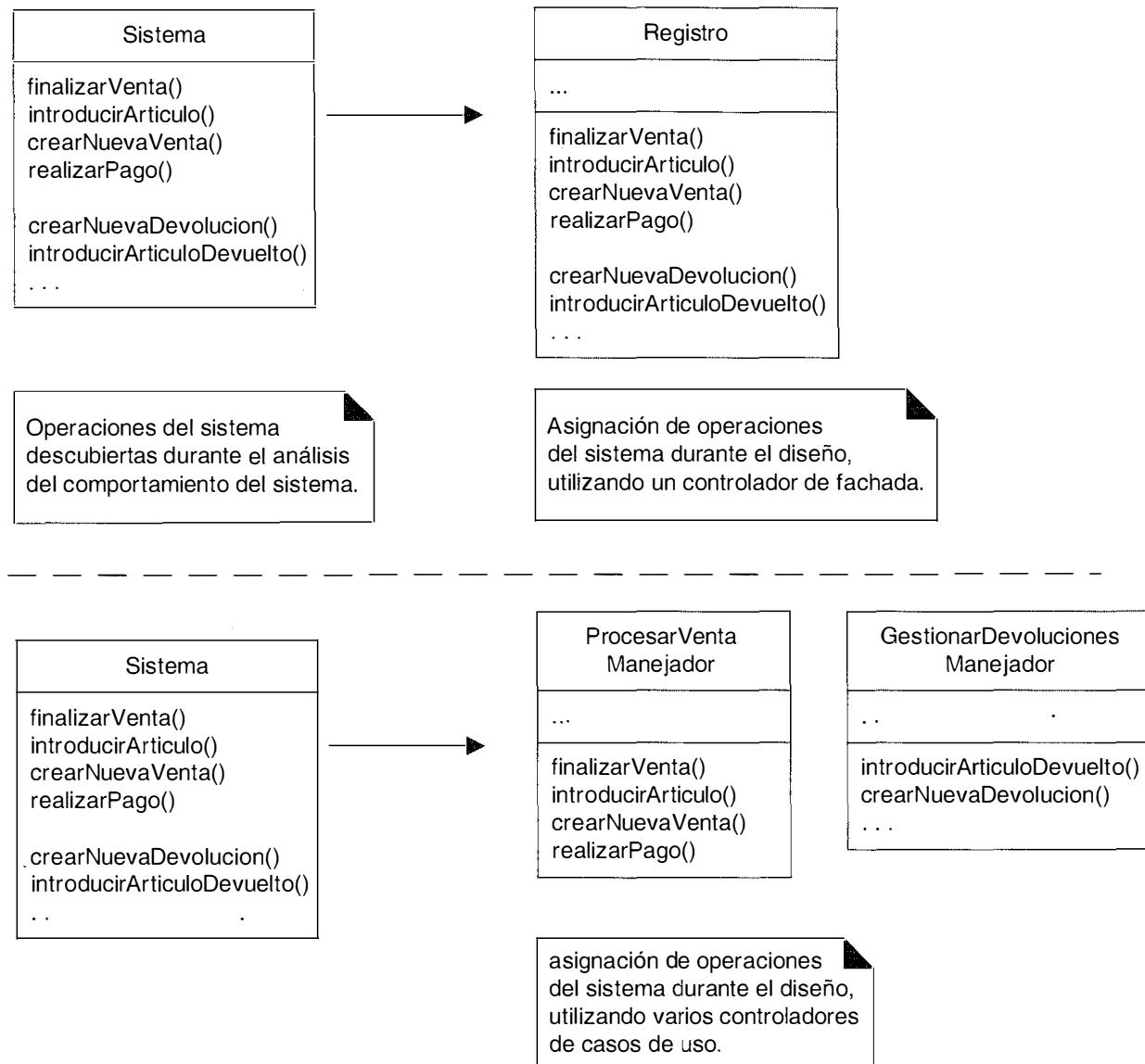


Figura 16.16. Asignación de las operaciones del sistema.

En todos los casos, si se utiliza un diseño de objetos, se debe escoger algún manejador para estos eventos. El patrón Controlador proporciona guías acerca de las opciones generalmente aceptadas y adecuadas. Como se ilustró en la Figura 16.14, el controlador es una especie de fachada en la capa del dominio para la capa de la interfaz.

A menudo, es conveniente utilizar la misma clase controlador para todos los eventos del sistema de un caso de uso de manera que es posible mantener la información acerca del estado del caso de uso en el controlador. Tal información es útil, por ejemplo, para identificar eventos del sistema que se apartan de la secuencia establecida (por ejemplo, una operación de *realizarPago* antes que la operación *finalizarVenta*). Podrían utilizarse diferentes controladores para casos de usos distintos.

Un error típico del diseño de los controladores es otorgarles demasiada responsabilidad.

Normalmente, un controlador debería *delegar* en otros objetos el trabajo que se necesita hacer; coordina o controla la actividad. No realiza mucho trabajo por sí mismo.

Por favor, diríjase a la sección “Cuestiones y Soluciones” que aparece más adelante para una discusión más elaborada.

La primera categoría de controlador es el controlador de fachada que representa al sistema global, dispositivo o subsistema. La idea es elegir algún nombre de clase que sugiera una cubierta, o fachada, sobre las otras capas de la aplicación, y que proporciona las llamadas a los servicios más importantes desde la capa de UI hacia las otras capas. Podría ser una abstracción de toda la unidad física, como un *Registro*⁶, *Conmutador-DeTelecomunicaciones*, *Teléfono* o *Robot*; una clase que represente el sistema software completo, como un *SistemaPDV*, o cualquier otro concepto que el diseñador elija que represente al sistema o subsistema global, incluso, por ejemplo, *JuegoAjedrez* si fuera un software de juegos.

Los controladores de fachada son adecuados cuando no existen “demasiados” eventos del sistema, o no es posible que la interfaz de usuario (UI) redireccione mensajes de los eventos del sistema a controladores alternativos, como un sistema de procesamiento de mensajes.

Si se elige un controlador de casos de uso, entonces hay un controlador diferente para cada caso de uso. Nótese que no es un objeto del dominio; es una construcción artificial para dar soporte al sistema (una *Fabricación Pura* en términos de los patrones GRASP). Por ejemplo, si la aplicación NuevaEra contiene casos de uso tales como *Procesar Venta* y *Gestionar Devoluciones*, entonces podría haber una clase *ProcesarVentaManejador* y así sucesivamente.

¿Cuándo se debería escoger un controlador de casos de uso? Es una alternativa a tener en cuenta cuando la asignación de las responsabilidades a un controlador de fachada nos conduce a diseños con baja cohesión o alto acoplamiento, generalmente cuando el controlador de fachada se está “inflando” con excesivas responsabilidades. Un controlador de casos de uso es una buena elección cuando hay muchos eventos del sistema repartidos en diferentes procesos; el controlador factoriza la gestión en clases separadas manejables, y también proporciona una base para conocer y razonar sobre el estado de los escenarios actuales en marcha.

En el UP y en el método más antiguo de Jacobson, Objectory [Jacobson92], existen los conceptos (opcionales) de clases frontera (*boundary*), control y entidad. Los **objetos frontera** son abstracciones de las interfaces, los **objetos entidad** son los objetos software del dominio independiente de la aplicación (y normalmente persistentes), y los **objetos control** son los manejadores de los casos de uso tal y como se describen en el patrón Controlador.

Un importante corolario del patrón Controlador es que los objetos interfaz (por ejemplo, los objetos ventana o elementos gráficos) y la capa de presentación no deberían ser responsables de llevar a cabo los eventos del sistema. En otras palabras, las operaciones del sistema se deberían manejar en la lógica de la aplicación o capas del dominio en lugar de en la capa de interfaz del sistema. Diríjase a la sección “Cuestiones y Soluciones” para ver un ejemplo.

⁶ Se utilizan varios términos para una unidad de PDV física, entre las que figuran: registro, terminal de punto de venta (TPDV), etcétera. A lo largo del tiempo, el “registro” ha llegado a encarnar la noción tanto de una unidad física, como la abstracción lógica de lo que registra ventas y pagos.

El objeto Controlador es normalmente un objeto del lado del cliente en el mismo proceso que la UI (por ejemplo, una aplicación con una GUI con las Swing de Java), entonces no es exactamente aplicable cuando el UI es un cliente Web en un navegador, y hay software del lado del servidor involucrado. En el último caso, existen varios patrones comunes para manejar los eventos del sistema que están fuertemente influenciados por el marco tecnológico escogido en el lado del servidor, como los servlets de Java. No obstante, es un estilo común crear controladores de casos de uso en el lado del servidor con o bien un servlet o un bean sesión EJB (*Enterprise JavaBeans*) para cada caso de uso. El objeto de sesión del lado del servidor representa una “sesión” de interacción con un actor externo.

Si la UI no es un cliente web (por ejemplo, es una GUI Swing o Windows), pero la aplicación invoca servicios remotos, es todavía común el uso del patrón Controlador. La UI reenvía la solicitud al Controlador local del lado del cliente, y el Controlador podría reenviar toda o parte de la gestión de la petición a los servicios remotos. Este diseño disminuye el acoplamiento entre la UI y los servicios remotos, y hace más fácil, por ejemplo, abastecer los servicios de manera local o remota, por medio de la indirección del Controlador del lado del cliente.

Resumiendo, el Controlador recibe la solicitud del servicio desde la capa de UI y coordina su realización, normalmente delegando a otros objetos.

Beneficios

- *Aumenta el potencial para reutilizar y las interfaces conectables (pluggable).* Asegura que la lógica de la aplicación *no* se maneja en la capa de interfaz. Técnicamente, las responsabilidades de un controlador podrían manejarse en un objeto interfaz, pero la implicación de tal diseño es que el código del programa y la lógica relacionada con la realización de la lógica de la aplicación estaría embebida en los objetos ventana o interfaz. Un diseño de una interfaz como controlador reduce la oportunidad de reutilizar la lógica en futuras aplicaciones, puesto que está ligada a una interfaz particular (por ejemplo, objetos ventana) que es raramente aplicable en otras aplicaciones. En cambio, delegando la responsabilidad de una operación del sistema a un controlador ayuda a la reutilización de la lógica en futuras aplicaciones. Y puesto que la lógica no está ligada a la capa de interfaz, puede sustituirse por una interfaz nueva.
- *Razonamiento sobre el estado de los casos de uso.* A veces es necesario asegurar que las operaciones del sistema tienen lugar en una secuencia válida, o ser capaces de razonar sobre el estado actual de la actividad y operaciones del caso de uso que está en marcha. Por ejemplo, podría ser necesario garantizar que la operación *realizarPago* no puede ocurrir hasta que haya tenido lugar la operación *finalizarVenta*. Si es así, es necesario capturar en algún sitio esta información de estado; el controlador es una opción razonable, especialmente si el mismo controlador se utiliza en todo el caso de uso (lo que se recomienda).

Cuestiones y Soluciones

Controladores saturados

Una clase controlador pobremente diseñada tendrá baja cohesión —no centrada en algo concreto y que gestiona demasiadas áreas de responsabilidad—; este controlador se denomina **controlador saturado**. Signos de que existe un controlador saturado son:

- Existe una *única* clase controlador que recibe *todos* los eventos del sistema en el sistema, y hay muchos. Esto ocurre a veces si se elige un controlador de fachada.
- El propio controlador realiza muchas de las tareas necesarias para llevar a cabo los eventos del sistema, sin delegar trabajo. Normalmente esto conlleva una violación de los patrones Experto en Información y Alta Cohesión.
- Un controlador tiene muchos atributos y mantiene información significativa sobre el sistema o el dominio, que debería haberse distribuido a otros objetos, o duplica información que se encuentra en otros sitios.

Hay varios remedios para un controlador saturado entre los que se encuentran:

1. Añadir más controladores: un sistema no tiene que tener sólo uno. En lugar de un controlador de fachada, utilice controladores de casos de uso. Por ejemplo, considere una aplicación con muchos eventos del sistema, como un sistema de reservas de vuelos.

Podría contener los siguientes controladores:

<i>Controladores de casos de uso</i>
RealizarReservaManejador
GestionarHorariosManejador
GestionarTarifasManejador

2. Diseñe el controlador de manera que, ante todo, delegue el cumplimiento de cada responsabilidad de una operación del sistema a otros objetos.

La capa de interfaz no maneja eventos del sistema

Reiterando: un corolario importante del patrón Controlador es que los objetos interfaz (por ejemplo, los objetos ventana) y la capa de interfaz, no deberían ser responsables de manejar los eventos del sistema. Como ejemplo, considere un diseño en Java que utiliza un *JFrame* para mostrar la información.

Asuma que la aplicación NuevaEra tiene una ventana que muestra la información de la venta y captura las operaciones del cajero. Utilizando el patrón Controlador, la Figura 16.17 ilustra una relación aceptable entre el *JFrame*, el Controlador y otros objetos en una parte del sistema de PDV (simplificado).

Nótese que la clase *JFrameVenta* —perteneciente a la capa de interfaz— pasa el mensaje *introducirArticulo* al objeto *Registro*. No se involucró en el procesamiento de la operación o en decidir cómo manejarla; la ventana sólo la delega a otra capa.

Asignando la responsabilidad de las operaciones del sistema a objetos en la capa de aplicación o del dominio —utilizando el patrón Controlador— en lugar de en la capa de interfaz incrementa el potencial para reutilizar. Si un objeto de la capa de interfaz (como el *JFrameVenta*) maneja una operación del sistema —que representa parte de un proceso de negocio— entonces la lógica del proceso del negocio estaría contenida en un

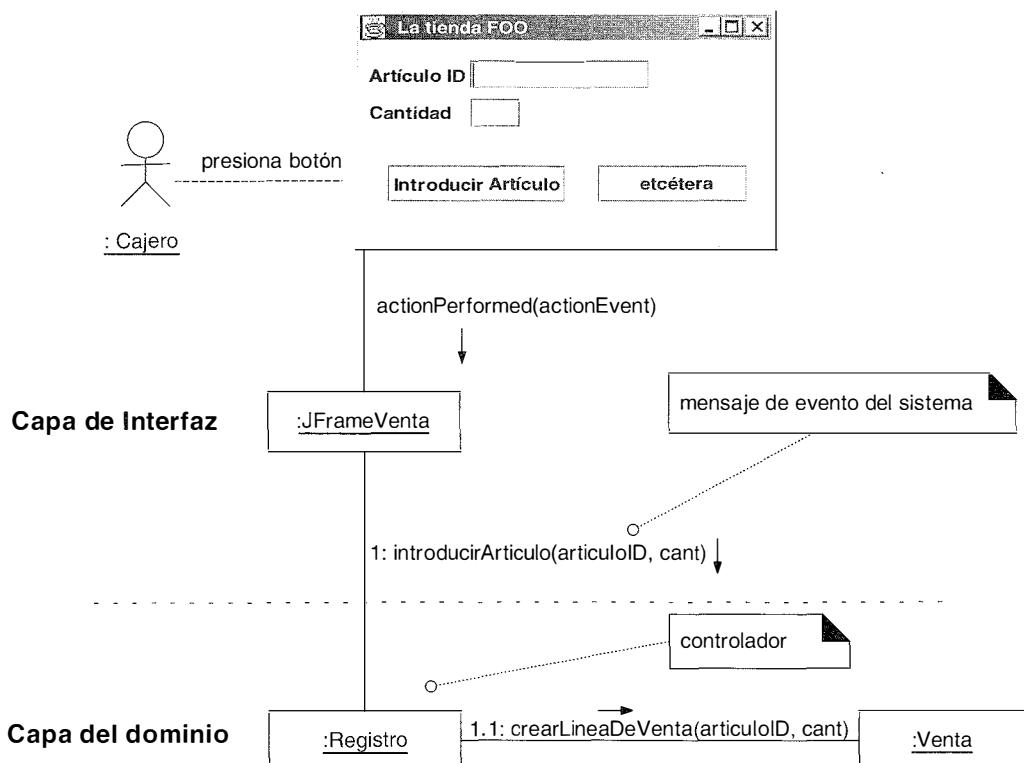


Figura 16.17. Acoplamiento deseable entre la capa de interfaz y la del dominio.

objeto interfaz (por ejemplo, del tipo ventana), por lo que la oportunidad para reutilizar es baja debido a su acoplamiento con una interfaz particular y con la aplicación.

En consecuencia, no es conveniente el diseño de la Figura 16.18.

La localización de la responsabilidad de las operaciones del sistema en un controlador que es un objeto del dominio facilita la reutilización de la lógica del programa lo que permite soportar el proceso de negocio asociado en futuras aplicaciones. También facilita la desconexión de la capa de interfaz y la utilización de un framework o tecnología de interfaz diferente o ejecutar el sistema en un modo “por lotes” sin conexión.

Sistemas de manejo de mensajes y el patrón *Command*

Algunas aplicaciones son sistemas de manejo de mensajes o servidores que reciben peticiones de otros procesos. Un comutador de telecomunicaciones es un ejemplo típico. En tales sistemas, el diseño de la interfaz y el controlador es algo diferente. Los detalles se estudiarán en un capítulo posterior, pero en esencia, una solución típica es utilizar el patrón *Command* [GHJV95] y el patrón *Command Processor* [BMRSS96], que se presentarán en el Capítulo 34.

Patrones Relacionados

- **Command:** En un sistema de manejo de mensajes, cada mensaje podría representarse y manejarse mediante un objeto *Command* separado [GHJV95].
- **Fachada:** Un controlador de fachada es un tipo de Fachada (*Facade*) [GHJV95].

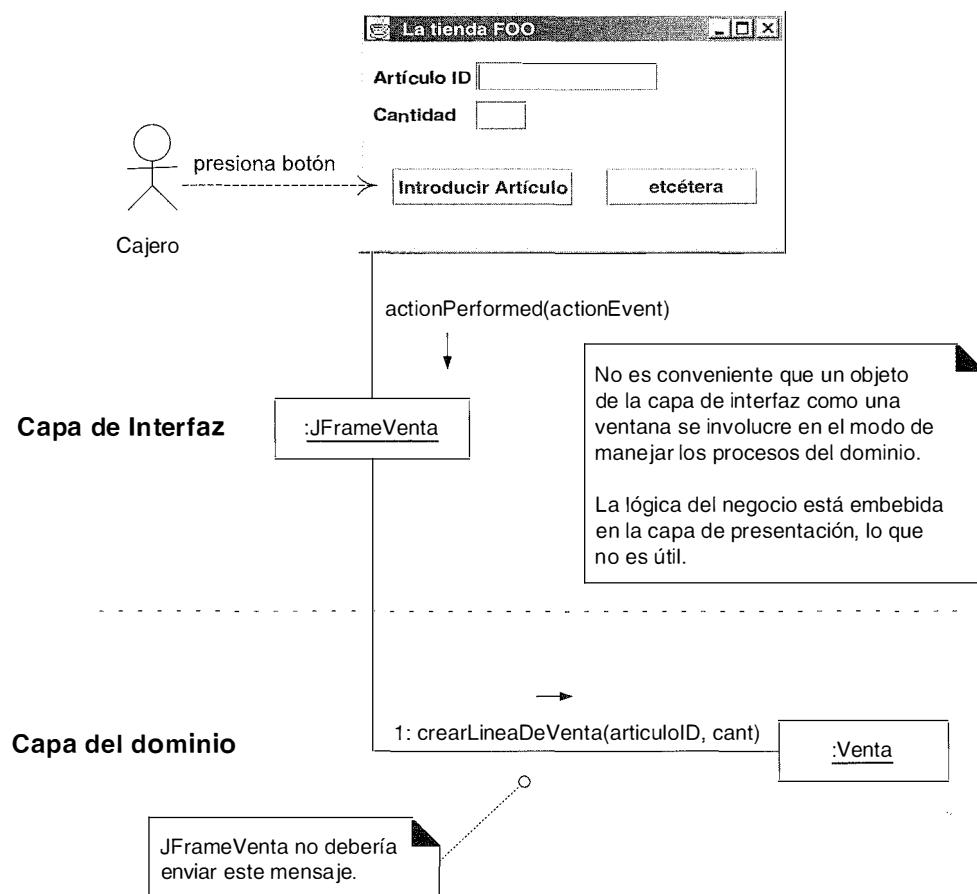


Figura 16.18. Acoplamiento menos conveniente entre la capa de interfaz y la del dominio.

- **Capas:** Es un patrón POSA [BMRSS96]. La ubicación de la lógica del dominio en la capa del dominio en lugar de en la capa de presentación forma parte del patrón de Capas (Layers).
- **Fabricación Pura:** Es otro patrón GRASP. Una Fabricación Pura es una creación arbitraria del diseñador, no una clase software cuyo nombre se inspira en el Modelo del Dominio. Un controlador de caso de uso es un tipo de Fabricación Pura.

16.11. Diseño de objetos y tarjetas CRC

Aunque formalmente no forma parte de UML, otro mecanismo que se utiliza algunas veces para ayudar a asignar responsabilidades e indicar las colaboraciones con otros objetos son las **tarjetas CRC** (tarjetas Clase-Responsabilidad-Colaborador) [BC89]. Kent Beck y Ward Cunningham fueron quienes promovieron en el uso de estas tarjetas y son los principales responsables de estimular a los diseñadores de software a pensar de manera más abstracta en términos de asignación de responsabilidades y colaboraciones, y también del uso de los patrones.

Las tarjetas CRC son fichas, una por cada clase, en las que se escriben brevemente las responsabilidades de la clase, y una lista de los objetos con los que colabora para llevar a cabo esas responsabilidades. Se desarrollan normalmente en una sesión de trabajo en grupo pequeño. Los patrones GRASP se podrían aplicar cuando se tiene en cuenta el diseño mientras se utilizan las tarjetas CRC.

Las tarjetas CRC son una técnica para registrar los resultados de la asignación de responsabilidades y asignaciones. La información recopilada se puede enriquecer utilizando diagramas de clases y de interacción. Lo importante no son las tarjetas o los diagramas, sino tener presente la asignación de responsabilidades.

16.12. Lecturas adicionales

La metáfora de los objetos que colaboran con responsabilidades, o el **Diseño Dirigido por Responsabilidades**, surgió especialmente a partir del trabajo sobre objetos en Smalltalk en Tektronix, Portland, de Kent Beck, Ward Cunningham, Rebecca Wirfs-Brock, y otros, que ha tenido gran influencia. El libro de texto que sirve de referencia es *Designing Object-Oriented Software* [WWW90], tan relevante hoy como cuando fue escrito.

Otros dos textos recomendados que destacan los principios fundamentales del diseño de objetos son *Object-Oriented Design Heuristics* de Riel y *Object Models* de Coad.

Capítulo 17

MODELO DE DISEÑO: REALIZACIÓN DE CASOS DE USO CON LOS PATRONES GRASP

Para inventar, necesitas una buena imaginación y un montón de trastos viejos.

Thomas Edison

Objetivos

- Diseñar las realizaciones de los casos de uso.
 - Aplicar los patrones GRASP para asignar responsabilidades a las clases.
 - Utilizar la notación de los diagramas de interacción UML para ilustrar el diseño de objetos.
-

Introducción

Este capítulo presenta cómo crear un diseño de objetos que colaboran con responsabilidades. Se presta especial atención a la aplicación de los patrones GRASP para desarrollar una solución bien diseñada. Por favor, obsérvese que los patrones GRASP como tales o por el nombre no son lo importante; son simplemente un apoyo al aprendizaje para ayudar a discutir y realizar de manera metódica el diseño de objetos fundamental.

Este capítulo expone los principios, utilizando el ejemplo del PDV NuevaEra, según los cuales un diseñador orientado a objetos asigna las responsabilidades y establece las interacciones entre los objetos —una técnica fundamental en el desarrollo orientado a objetos—.

Nota:

La asignación de responsabilidades y el diseño de colaboraciones son etapas muy importantes y creativas durante el diseño, mientras se elaboran los diagramas o mientras se programa.

El material es intencionalmente detallado; pretende ilustrar de manera exhaustiva que no existen decisiones “mágicas” o injustificadas en el diseño de objetos —la asignación de responsabilidades y la elección de las interacciones de objetos pueden explicarse y aprenderse de una forma racional—.

17.1. Realizaciones de casos de uso

Citando textualmente, “Una realización de caso de uso describe cómo se realiza un caso de uso particular en el modelo de diseño, en función de los objetos que colaboran” [RUP]. De manera más precisa, un diseñador puede describir el diseño de uno o más *escenarios* de un caso de uso; cada uno de estos se denomina una realización del caso de uso. La realización del caso de uso es un término o concepto del UP que se utiliza para recordarnos la conexión entre los requisitos expresados como casos de uso, y el diseño de objetos que satisface los requisitos.

Los diagramas de interacción UML son un lenguaje común para ilustrar las realizaciones de los casos de uso. Y como se estudió en el capítulo anterior, existen principios o patrones del diseño de objetos, como el Experto en Información y Bajo Acoplamiento, que se pueden aplicar durante este trabajo de diseño.

Como repaso, la Figura 17.20 (casi al final de este capítulo) expone las relaciones entre algunos artefactos del UP:

- El caso de uso sugiere los eventos del sistema que se muestran explícitamente en los diagramas de secuencia del sistema.
- Opcionalmente, podrían describirse los detalles de los efectos de los eventos del sistema en términos de los cambios de los objetos del dominio en los contratos de las operaciones del sistema.
- Los eventos del sistema representan los mensajes que inician los diagramas de interacción, los cuales representan el modo en el que los objetos interactúan para llevar a cabo las tareas requeridas —la realización del caso de uso—.
- Los diagramas de interacción comprenden la interacción de mensajes entre objetos software cuyos nombres se inspiran algunas veces en los nombres de las clases conceptuales del Modelo del Dominio, además de otras clases de objetos.

17.2. Comentarios sobre los artefactos

Diagramas de interacción y realizaciones de casos de uso

En la iteración actual estamos teniendo en cuenta varios escenarios y eventos del sistema tales como:

- *Procesar Venta: crearNuevaVenta, introducirArticulo, finalizarVenta, realizarPago.*

Si se utilizan los diagramas de interacción para representar las realizaciones de los casos de uso, se necesitará un diagrama de colaboración diferente para mostrar el manejo de cada mensaje de evento del sistema. Por ejemplo (Figura 17.1):

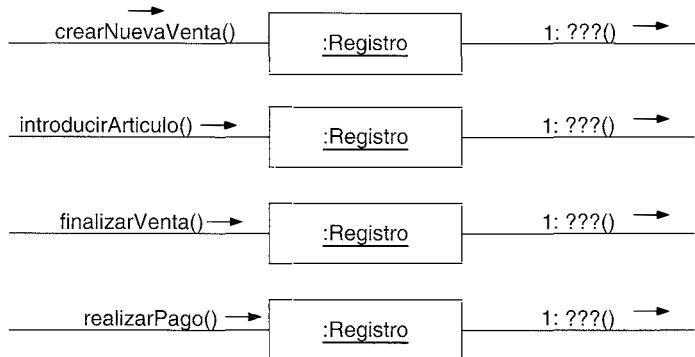


Figura 17.1. Diagramas de colaboración y manejo de los mensajes de eventos del sistema.

Por otro lado, si se utilizan los diagramas de secuencia, *podría* ser posible encajar todos los mensajes de eventos del sistema en el mismo diagrama, como en la Figura 17.2.

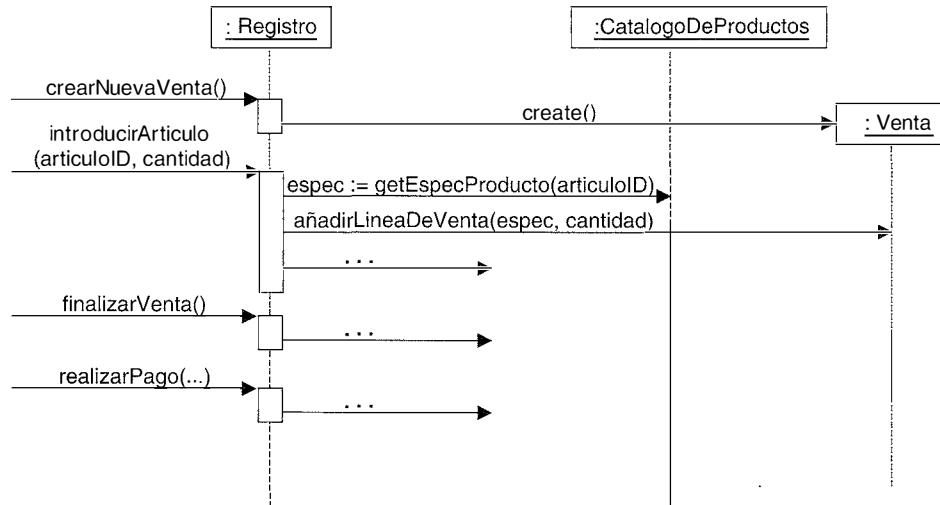


Figura 17.2. Un diagrama de secuencia y el manejo de los mensajes de eventos del sistema.

Sin embargo, ocurre a menudo que el diagrama de secuencia es entonces demasiado complejo o largo. Es legal, como con los diagramas de interacción, utilizar un diagrama de secuencia para cada mensaje de evento del sistema, como en la Figura 17.3.

Contratos y las realizaciones de casos de uso

Reiterando, podría ser posible diseñar las realizaciones de los casos de uso directamente a partir del texto de los casos de uso. Además, para algunas operaciones del sistema, se podrían haber escrito los contratos que añaden más detalles o son más específicos. Por ejemplo:

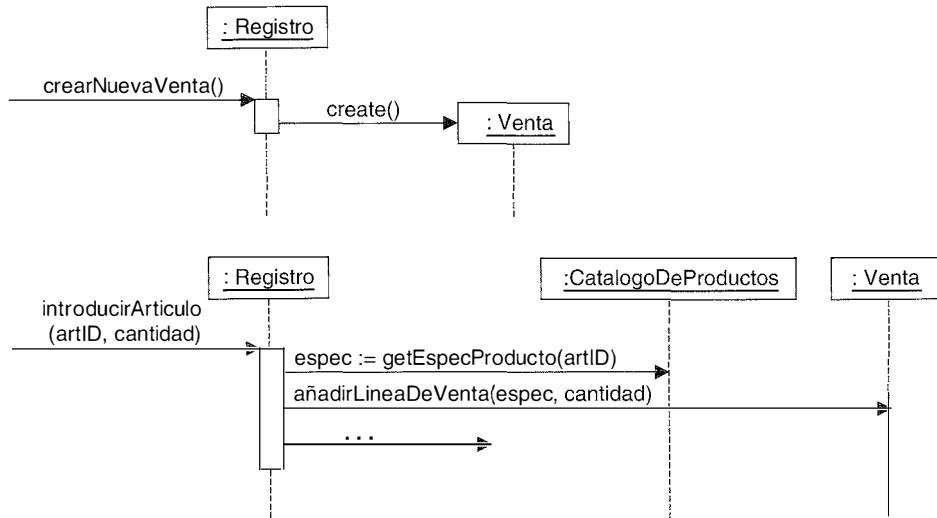


Figura 17.3. Múltiples diagramas de secuencia y manejo de los mensajes de eventos del sistema.

Contrato CO2: introducirArticulo

Operación:	introducirArticulo(articuloID:ArticuloID, cantidad:integer)
Referencias cruzadas:	Caso de Uso: Procesar Venta
Precondiciones:	Hay una venta en curso
Postcondiciones:	<ul style="list-style-type: none"> - Se creó una instancia de LineaDeVenta Idv (creación de instancias). - ...

Al mismo tiempo que tenemos en cuenta el texto de los casos de uso, para cada contrato, trabajamos cuidadosamente en expresar en el cambio de estado en las postcondiciones y diseñamos las interacciones para satisfacer los requisitos. Por ejemplo, dada esta operación parcial del sistema *introducirArticulo*, la Figura 17.4 muestra un diagrama de interacción parcial que satisface el cambio de estado de la creación de la instancia de *LíneaDeVenta*.

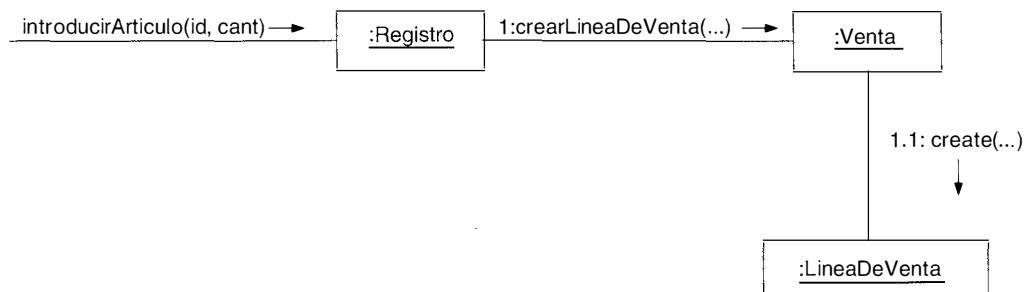


Figura 17.4. Diagrama de interacción parcial.

Advertencia: Los requisitos no son perfectos

Es útil tener presente que los casos de uso que se han escrito previamente y los contratos dan sólo una idea aproximada de lo que se tiene que conseguir. La historia del desarrollo del software nos va descubriendo invariablemente que los requisitos no son perfectos, o han cambiado. Esto no es una excusa para no tratar de hacer un buen trabajo de requisitos, sino el reconocimiento de que se necesita involucrar continuamente a los clientes y a los expertos en la materia en estudio para revisar y proporcionar retroalimentación sobre el comportamiento del sistema que se está desarrollando.

Una ventaja del desarrollo iterativo es que favorece de manera natural el descubrimiento de nuevos resultados del análisis y diseño durante el trabajo de diseño e implementación. El espíritu del desarrollo iterativo es capturar un grado “razonable” de información durante el análisis de requisitos, completando los detalles durante el diseño y la implementación.

El Modelo del Dominio y las realizaciones de los casos de uso

Algunos de los objetos software que interactúan mediante el paso de mensajes en los diagramas de interacción se inspiran en el Modelo del Dominio, como la clase conceptual *Venta* y la clase del diseño *Venta*. La elección de la asignación adecuada de la responsabilidad utilizando los patrones GRASP depende, en parte, de la información del Modelo del Dominio. Como ya se ha dicho, el Modelo del Dominio existente es probable que no sea perfecto; se espera que haya errores y omisiones. Se descubrirán nuevos conceptos que se olvidaron previamente, se ignorarán conceptos que se identificaron anteriormente, y lo mismo ocurrirá con las asociaciones y los atributos.

Clases conceptuales vs. clases del diseño

Recordemos que el Modelo del Dominio del UP no representa clases software, pero podría utilizarse para inspirar la presencia y los nombres de algunas clases software en el Modelo de Diseño. Durante la elaboración de los diagramas de interacción o la programación, los desarrolladores podrían mirar en el Modelo del Dominio para asignar los nombres a algunas clases del diseño; por tanto, se crea un diseño con un salto en la representación más bajo entre el diseño del software y nuestra percepción del dominio del mundo real con el que el software está relacionado (ver Figura 17.5).

¿Se deben limitar las clases del Modelo de Diseño a clases con nombres inspirados a partir del Modelo del Dominio? En absoluto, durante este trabajo de diseño es conveniente descubrir nuevas clases conceptuales que se obviaron durante el análisis de dominio inicial, y también crear clases software cuyos nombres y objetivos no estén relacionados en absoluto con el Modelo del Dominio.

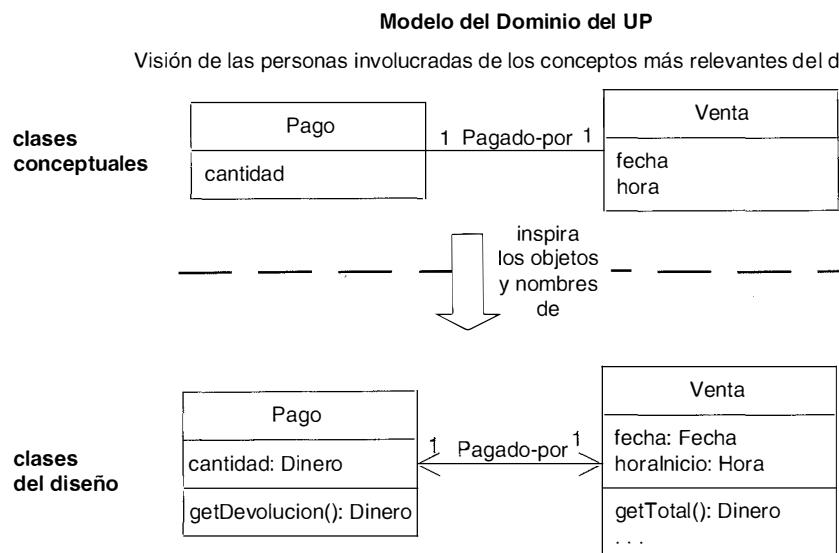


Figura 17.5. Disminución del salto en la representación nombrando las clases del diseño a partir de las clases conceptuales.

17.3. Realizaciones de casos de uso para la iteración de NuevaEra

Las siguientes secciones exploran las elecciones y decisiones tomadas durante el diseño de una realización de caso de uso con objetos basado en los patrones GRASP. Las explicaciones son intencionalmente detalladas, en un intento de ilustrar que la creación de los diagramas de interacción bien diseñados no se trata de un “rollo” sin fundamento, sino que su construcción se basa en principios justificables.

En cuanto a la notación, el diseño de objetos para cada mensaje de eventos del sistema se representará en un diagrama independiente, para centrarse en las cuestiones de diseño de cada uno. Sin embargo, podrían haberse agrupado juntos en un único diagrama de secuencia.

17.4. Diseño de objetos: crearNuevaVenta

La operación del sistema *crearNuevaVenta* tiene lugar cuando un cajero solicita comenzar una nueva venta, después de que haya llegado un cliente con cosas que comprar. El caso de uso podría haber sido suficiente para decidir lo que era necesario, pero para este caso de estudio escribimos los contratos para todos los eventos del sistema, para que la explicación sea lo más completa posible.

Contrato CO1: crearNuevaVenta

Operación:	crearNuevaVenta()
Referencias cruzadas:	Caso de Uso: Procesar Venta
Precondiciones:	ninguna
Postcondiciones:	<ul style="list-style-type: none"> - Se creó una instancia de Venta v (creación de instancias). - v se asoció con el Registro (formación de asociaciones). - Se inicializaron los atributos de v.

Elección de la clase controlador

Nuestra primera elección de diseño comprende la elección del controlador para el mensaje de operación del sistema *crearNuevaVenta*. De acuerdo con el patrón Controlador, algunas de las opciones podrían ser:

representa el “sistema” global, *Registro, SistemaPDV*
dispositivo o subsistema

representa un receptor o manejador *ProcesarVentaManejador;*
de todos los eventos del sistema *ProcesarVentaSesion*
de un escenario de caso de uso

Es aceptable elegir un controlador de fachada como el *Registro* si sólo hay unas pocas operaciones del sistema y el controlador no está asumiendo demasiadas responsabilidades (en otras palabras, si va a perder la cohesión). Es adecuado elegir un controlador de caso de uso cuando hay muchas operaciones del sistema y deseamos distribuir las responsabilidades con el fin de mantener cada clase controlador ligera y centrada (en otras palabras, cohesiva). En este caso, *Registro* será suficiente, puesto que sólo hay unas pocas operaciones del sistema.

Este *Registro* es un objeto software del Modelo de Diseño. No es un registro físico real sino una abstracción del software cuyo nombre se eligió para disminuir el salto en la representación entre nuestra concepción del dominio y el software.

Por tanto, el diagrama de interacción que se muestra en la Figura 17.6, comienza enviando el mensaje *crearNuevaVenta* al objeto software *Registro*.

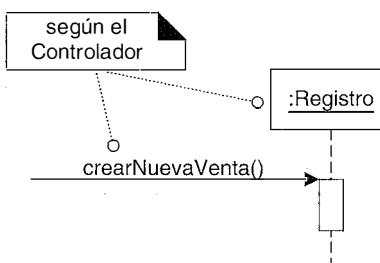


Figura 17.6. Aplicación del patrón GRASP Controlador.

Creación de una nueva Venta

Se debe crear un nuevo objeto software *Venta*, y el patrón GRASP Creador sugiere la asignación de la responsabilidad de creación a la clase que agrega, contiene o registra el objeto que se va a crear.

El análisis del Modelo del Dominio revela que se puede considerar que el *Registro* registra una *Venta*; de hecho, la palabra “registro” durante muchos años ha significado la cosa que graba (o registra) las transacciones contables, como las ventas.

Por tanto, el *Registro* es un candidato razonable para crear una *Venta*. Y consintiendo que el *Registro* cree la *Venta*, se puede asociar fácilmente a ella a lo largo del tiempo, de manera que durante futuras operaciones en la sesión, el *Registro* tendrá una referencia a la instancia de *Venta* actual.

Además de lo anterior, cuando se crea la *Venta*, se debe crear una colección vacía (contenedor, como una *List* de Java) para guardar todas las futuras instancias de *LíneaDeVenta* que se añadirán. La instancia de *Venta* contendrá y mantendrá esta colección, lo que implica, según el Creador, que la *Venta* es una buena candidata para crearla.

Por tanto, el *Registro* crea la *Venta*, y la *Venta* crea una colección vacía, representada mediante un multiobjeto en el diagrama de interacción.

Por lo tanto, el diagrama de interacción de la Figura 17.7 ilustra el diseño.

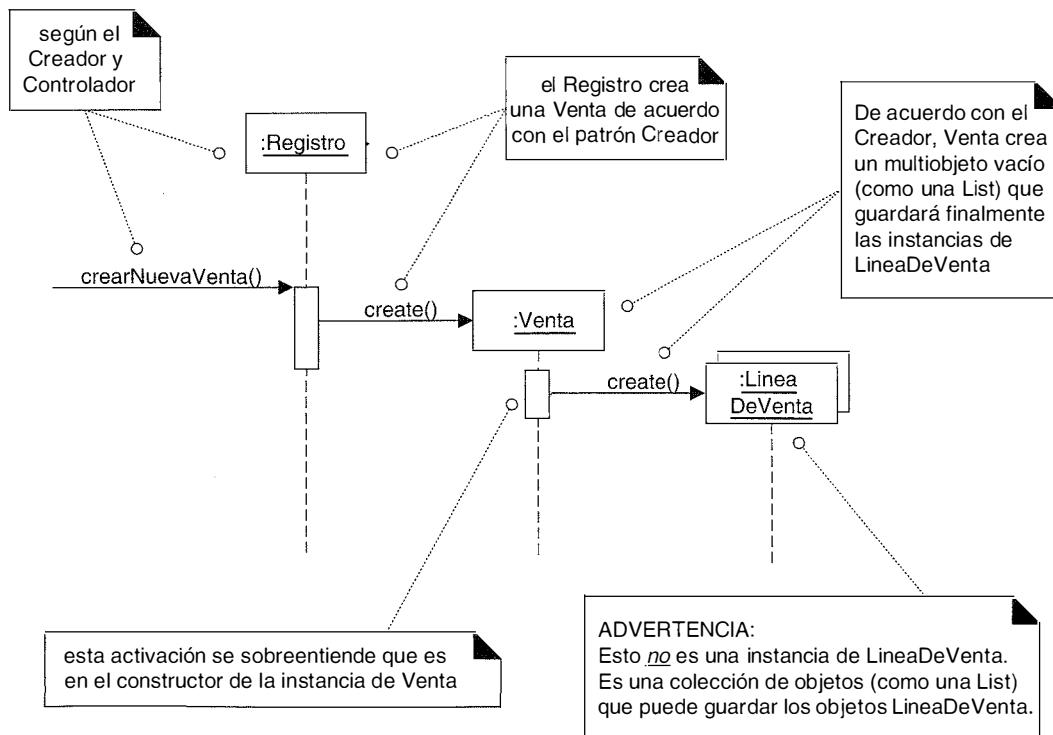


Figura 17.7. Creación de Venta y multiobjeto.

Conclusión

El diseño no fue difícil, pero lo importante es haberlo explicado de manera cuidadosa en términos del Controlador y el Creador para ilustrar que se pueden decidir y explicar, de manera racional y sistemática, los detalles de un diseño en base a principios y patrones, como los GRASP.

17.5. Diseño de objetos: introducirArticulo

La operación del sistema *introducirArticulo* tiene lugar cuando un cajero introduce el *articuloID* y (opcionalmente) la cantidad de ese artículo que se va a comprar. A continuación presentamos el contrato completo:

Contrato CO2: introducirArticulo

Operación:	introducirArticulo(articuloID:ArticuloID, cantidad:integer)
Referencias cruzadas:	Caso de Uso: Procesar Venta
Precondiciones:	Hay una venta en curso
Postcondiciones:	<ul style="list-style-type: none"> - Se creó una instancia de LineaDeVenta Idv (creación de instancias). - Idv se asoció con la Venta actual (formación de asociaciones). - Idv.cantidad pasó a ser cantidad (modificación de atributos). - Idv se asoció con una EspecificacionDelProducto, en base a la coincidencia del articuloID (formación de asociaciones).

Se construirá un diagrama de interacción que satisfaga la postcondición de *introducirArticulo*, utilizando los patrones GRASP para ayudar a tomar las decisiones de diseño.

Elección de la clase controlador

Nuestra primera elección tiene que ver con el manejo de la responsabilidad para el mensaje de operación del sistema *introducirArticulo*. Basado en el patrón Controlador, como para *crearNuevaVenta*, continuaremos utilizando el *Registro* como controlador.

¿Mostrar por pantalla la descripción y precio del artículo?

Debido a un principio de diseño denominado **Separación Modelo-Vista**, los objetos que no pertenecen a la GUI (como un *Registro* o *Venta*) no son responsables de involucrarse en las tareas de salida. Por tanto, aunque el caso de uso establezca que se muestran por pantalla la descripción y el precio después de la operación, el diseño lo ignorará en este momento.

Todo lo que se requiere con respecto a las responsabilidades de mostrar la información es que la información se conozca, lo que sucede en este caso.

Creación de una nueva LineaDeVenta

La postcondición del contrato *introducirArticulo* indica la creación, inicialización y asociación de una *LineaDeVenta*. El análisis del Modelo del Dominio revela que una *Venta* contiene objetos *LineaDeVenta*. Inspirándonos en el dominio, una *Venta* software podría contener igualmente objetos software *LineaDeVenta*. De aquí que, según el Creador, una *Venta* software sea una candidata adecuada para crear una *LineaDeVenta*.

La *Venta* puede asociarse con la *LineaDeVenta* recién creada almacenando la nueva instancia en su colección de líneas de venta. La postcondición indica que la nueva *LineaDeVenta*, cuando se crea, necesita una cantidad; por tanto, el *Registro* debe pasar esta cantidad a la *Venta*, quien debe pasársela como parámetro en el mensaje *create* (en Java, eso se implementaría como una llamada al constructor con un parámetro).

Por tanto, de acuerdo con el Creador, se envía a la *Venta* el mensaje *crearLineaDeVenta* para que cree una *LineaDeVenta*. La *Venta* crea una *LineaDeVenta*, y después almacena la nueva instancia en su colección permanente.

Los parámetros del mensaje *crearLineaDeVenta* incluyen la *cantidad*, de manera que la *LineaDeVenta* pueda registrarla, y del mismo modo la *EspecificacionDelProducto* que se corresponde con el *articuloID*.

Localización de una EspecificacionDelProducto

La *LineaDeVenta* necesita asociarse con la *EspecificacionDelProducto* que se corresponde con el *articuloID* de entrada. Esto implica que es necesario recuperar una *EspecificacionDelProducto*, en base a la coincidencia de un *articuloID*.

Antes de considerar *cómo* realizar la búsqueda, es conveniente considerar *quién* debe ser el responsable de ella. Por tanto, el primer paso es:

Comience asignando responsabilidades estableciendo claramente la responsabilidad.

Volviendo a plantear el problema:

¿Quién debe ser el responsable de conocer una *EspecificacionDelProducto*, basada en la coincidencia de un *articuloID*?

Esto no es ni un problema de creación ni uno sobre la elección de un controlador para un evento del sistema. Vamos a ver la primera aplicación del Experto en Información en el diseño.

En muchos casos, el patrón Experto es el principal patrón que se aplica. El Experto en Información sugiere que el objeto que tiene la información que se requiere para abordar la responsabilidad debería llevarla a cabo. ¿Quién conoce todos los objetos *EspecificacionDelProducto*?

El análisis del Modelo del Dominio revela que el *CatalogoDeProductos* lógicamente contiene todas las instancias *EspecificacionDelProducto*. Una vez más, inspirándonos en el dominio, diseñamos las clases software con una organización parecida: un *CatalogoDeProductos* software contendrá objetos software *EspecificacionDelProducto*.

Con eso decidido, entonces según el Experto en Información el *CatalogoDeProductos* es un buen candidato para esta responsabilidad de búsqueda puesto que conoce todos los objetos *EspecificacionDelProducto*.

Esto podría implementarse, por ejemplo, con un método denominado *getEspecificacion*¹.

Visibilidad del CatalogoDeProductos

¿Quién debería enviar el mensaje *getEspecificacion* al *CatalogoDeProductos* para solicitar una *EspecificacionDelProducto*?

Es razonable asumir que se crearon un *Registro* y un *CatalogoDeProductos* durante el caso de uso inicial *Poner en Marcha*, y que hay una conexión permanente desde el objeto *Registro* al objeto *CatalogoDeProductos*. Con este supuesto (que podríamos anotar en una lista de tareas de cosas que debemos asegurar en el diseño cuando vayamos a diseñar la inicialización), entonces es posible que el *Registro* envíe el mensaje *getEspecificacion* al *CatalogoDeProductos*.

Esto implica otro concepto en el diseño de objetos: la visibilidad. La **visibilidad** es la capacidad de un objeto de “ver” o tener una referencia a otro objeto.

Para que un objeto envíe un mensaje a otro objeto éste tiene que ser visible a aquél.

Puesto que asumiremos que el *Registro* tiene una conexión permanente —o referencia— al *CatalogoDeProductos*, éste es visible al *Registro* y, por tanto, puede enviarle un mensaje como el de *getEspecificacion*.

El siguiente capítulo abordará el tema de la visibilidad con más detenimiento.

Recuperación de objetos EspecificacionDelProducto de una base de datos

En la versión final de la aplicación del PDV NuevaEra, es improbable que todas las instancias de *EspecificacionDelProducto* se encuentren realmente en memoria. Será más probable que se almacenen en una base de datos relacional u objetual y se recuperarán bajo demanda; algunas podrían almacenarse en el proceso del cliente por motivos de rendimiento y tolerancia a fallos. Sin embargo, las cuestiones relacionadas con la recuperación de una base de datos se pospondrán por ahora en aras de la simplicidad. Se asumirá que todos los objetos *EspecificacionDelProducto* se encuentran en memoria.

El Capítulo 34 presentará el tema del acceso a base de datos de los objetos persistentes, que es un tema amplio en el que influye normalmente la elección de la tecnología, como J2EE, .NET, etcétera.

¹ La asignación de los nombres a los métodos de acceso es, por supuesto, una cuestión de estilo de cada lenguaje. Java siempre utiliza la forma *objeto.getFoo()*, C++ tiende a utilizar *objeto.foo()*, y C# utiliza *objeto.Foo*, lo que oculta (como Eiffel y Ada) si es una llamada a un método o un acceso directo a un atributo público. En los ejemplos se utiliza el estilo de Java.

El diseño de objetos de introducirArticulo

Dada la discusión anterior, el diagrama de interacción de la Figura 17.8 refleja las decisiones en cuanto a la asignación de responsabilidades y el modo en el que deberían interactuar los objetos. Obsérvese la considerable reflexión que se hizo para llegar a este diseño, basada en los patrones GRASP; el diseño de las interacciones entre objetos y la asignación de responsabilidades requiere alguna deliberación.

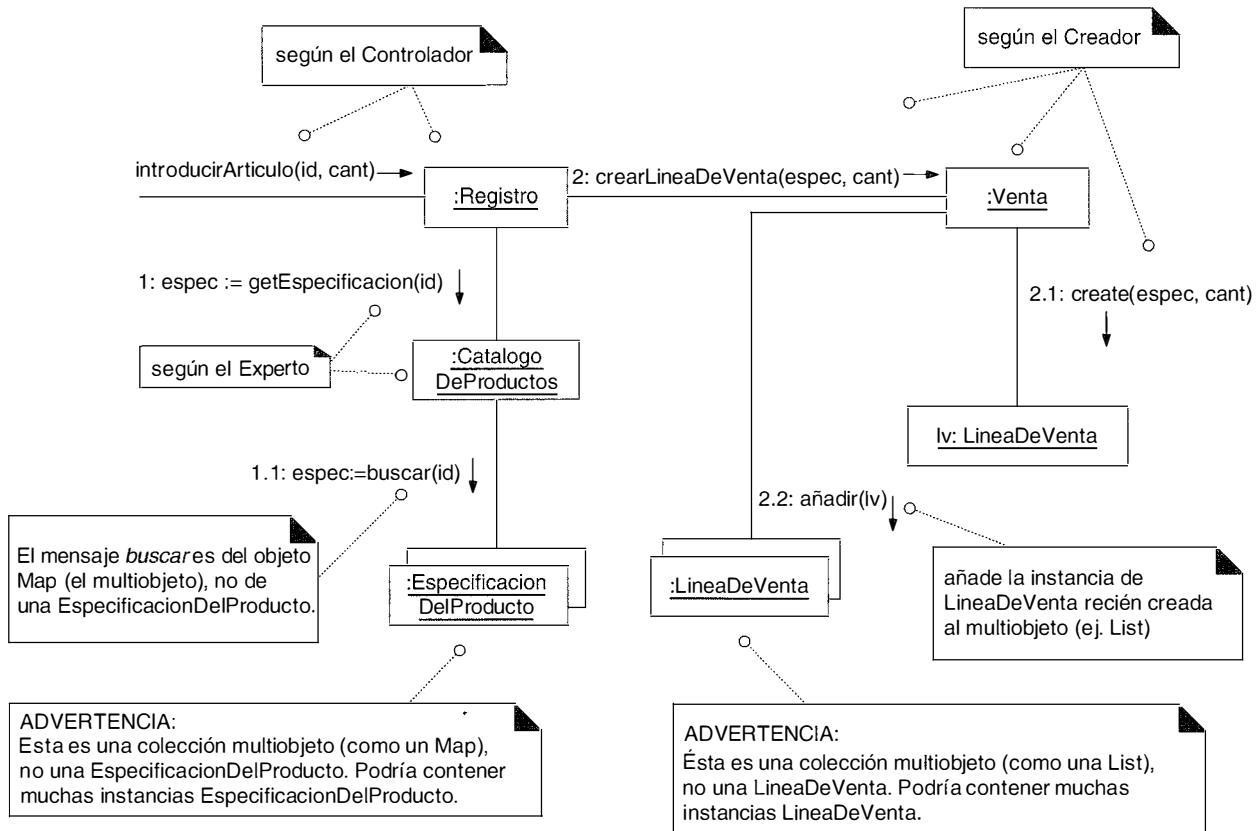


Figura 17.8. El diagrama de interacción para introducirArticulo.

Mensajes a los multiobjetos

Nótese que la interpretación en UML del envío de un mensaje a un multiobjeto es que se trata de un mensaje a la propia colección de objetos, en vez de una transmisión implícita a los miembros de la misma. Esto es especialmente obvio para las operaciones de colección genéricas como *buscar* y *añadir*.

Por ejemplo, en el diagrama de interacción de *introducirArticulo*:

- El mensaje *buscar* enviado al multiobjeto *EspecificacionDelProducto* es un mensaje que se envía una vez a la estructura de datos de la colección representada por el multiobjeto (como un *Map* de Java).
 - El mensaje genérico e independiente del lenguaje *buscar* se traducirá, durante la programación, para un lenguaje específico y librería. Quizás será finalmente

Map.get en Java. Podría haberse utilizado el mensaje *get* en los diagramas; se utilizó *buscar* para hacer entender que los diagramas de diseño podrían requerir alguna traducción a diferentes lenguajes y librerías.

- El mensaje *añadir* enviado al multiobjeto *LíneaDeVenta* es para añadir un elemento a la estructura de datos de la colección representada por un multiobjeto (como una *List* de Java).

17.6. Diseño de objetos: finalizarVenta

La operación del sistema *finalizarVenta* tiene lugar cuando un cajero presiona un botón indicando el final de la venta. A continuación presentamos el contrato:

Contrato CO3: finalizarVenta

Operación:	finalizarVenta()
Referencias cruzadas:	Caso de Uso: Procesar Venta
Precondiciones:	Hay una venta en curso
Postcondiciones:	- <i>Venta.esCompleta</i> pasó a ser verdad (modificación de atributos).

Elección de la clase controlador

Nuestra primera elección tiene que ver con el manejo de la responsabilidad para el mensaje de operación del sistema *finalizarVenta*. Basado en el patrón GRASP Controlador, como para *introducirArtículo*, continuaremos utilizando el *Registro* como controlador.

Valor del atributo *Venta.esCompleta*

La postcondición del contrato establece que:

- *Venta.esCompleta* pasó a ser verdad (modificación de atributos).

Como siempre, el primer patrón a tener en cuenta debería ser el Experto, a menos que sea un problema de creación o del controlador (que no lo es).

¿Quién debería ser el responsable de poner el valor del atributo *esCompleto* de la *Venta* a verdad?

Según el Experto, debería ser la propia *Venta*, puesto que conoce y mantiene el atributo *esCompleta*. Por tanto, el *Registro* enviará un mensaje *seHaCompletado* a la *Venta* para asignarle el valor *true*.

Notación UML para mostrar las restricciones, notas y algoritmos

La Figura 17.9 muestra el mensaje *seHaCompletado*, pero no pone de manifiesto lo que ocurre en el método *seHaCompletado* (aunque en este caso se reconoce que es trivial). Algunas veces en UML deseamos utilizar texto para describir el algoritmo de un método, o especificar alguna restricción.

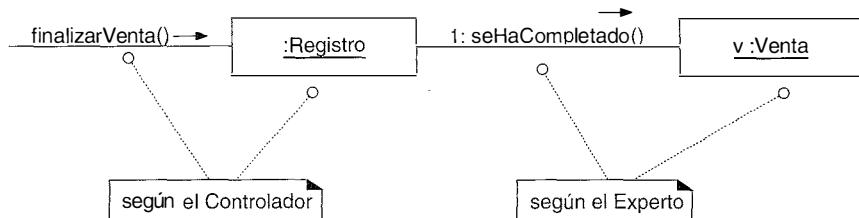


Figura 17.9. Finalización de la entrada de un artículo.

Para estas necesidades, UML proporciona tanto **restricciones** como **notas**. Una restricción UML es una información semánticamente significativa que se anexa a un elemento del modelo. Las restricciones en UML son texto encerrado entre llaves {}; por ejemplo, { $x > 20$ }. Se puede utilizar cualquier lenguaje formal o informal para las restricciones, y UML incluye especialmente **OCCL** (lenguaje de restricciones de objetos) [WK99] si uno desea utilizarlo.

Una nota de UML es un comentario que no tiene impacto semántico, como la fecha de la creación o el autor.

Una nota siempre se muestra en un **cuadro de nota** (un cuadro de texto con la esquina doblada).

Una restricción podría mostrarse como un simple texto entre llaves, que es adecuado para declaraciones cortas. Sin embargo, las restricciones largas también podrían colocarse en un “cuadro de nota”, en cuyo caso el presunto cuadro de nota realmente contiene una restricción en lugar de una nota. El texto del cuadro va entre llaves para indicar que es una restricción.

En la Figura 17.10 se utilizan ambos estilos. Nótese que el estilo de restricción simple (entre llaves pero no en un cuadro) solamente muestra una sentencia que debe ser verdad (el significado clásico de una restricción en lógica). Por otro lado, la “restricción” en el cuadro de nota muestra la implementación del método Java de la restricción. Ambos estilos son legales para representar una restricción en UML.

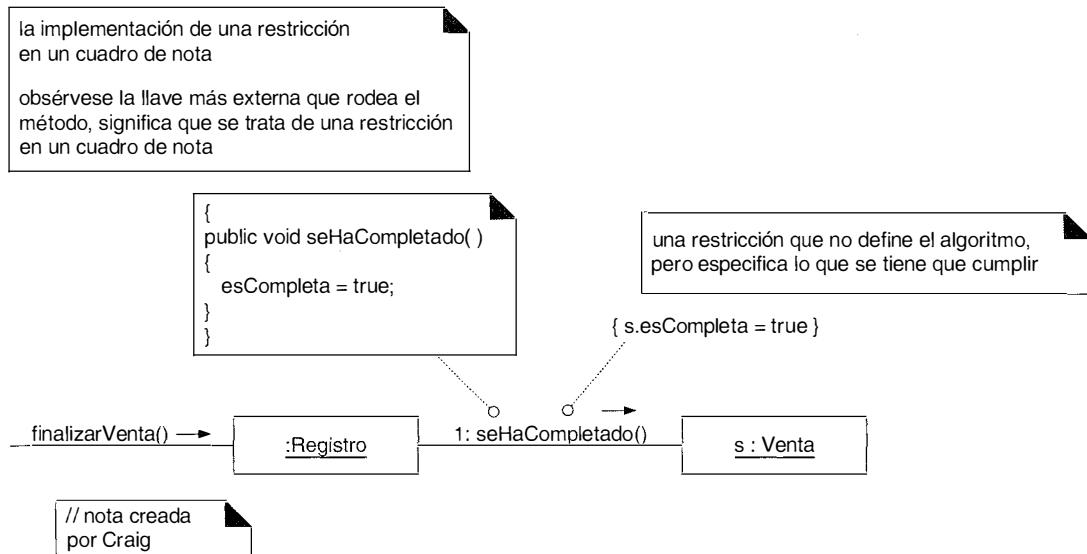


Figura 17.10. Restricciones y notas.

Cálculo del total de la Venta

Consideré el siguiente fragmento del caso de uso *Procesar Venta*:

Escenario principal de éxito (o Flujo Básico):

1. El Cliente llega...
2. El Cajero le dice al Sistema que cree una nueva venta
3. El Cajero introduce el identificador del artículo
4. El Sistema registra la línea de la venta y ...
El Cajero repite los pasos 3-4 hasta que se indique
5. El Sistema presenta el total con los impuestos calculados.

En el paso 5, se presenta (o se muestra por pantalla) un total. Debido al principio de Separación Modelo-Vista, no deberíamos preocuparnos por el diseño de cómo se mostrará el total de la venta, pero es necesario asegurar que se conoce el total. Nótese que ninguna clase de diseño actualmente conoce el total de la venta, de manera que necesitamos crear un diseño de interacciones de objetos que satisfaga este requisito.

Como siempre, el Experto en Información debería ser un patrón a tener en cuenta a no ser que se trate de un problema de controlador o de creación (que no lo es).

Probablemente es obvio que la propia *Venta* deba ser la responsable de conocer su total, pero considere el siguiente análisis sólo con el fin de hacer el proceso de razonamiento para encontrar un Experto transparente como el cristal —con un ejemplo sencillo—.

1. Establezca la responsabilidad:
 - ¿Quién debe ser el responsable de conocer el total de la venta?
2. Reúna la información necesaria:
 - El total de la venta es la suma de los subtotales de todas las líneas de venta.
 - El subtotal de la línea de venta := cantidad de la línea de venta * precio de la descripción del producto.
3. Liste la información requerida para abordar esta responsabilidad y las clases que conocen esta información.

Información requerida para el total de la Venta	Experto en Información
<i>EspecificacionDelProducto.precio</i>	<i>EspecificacionDelProducto</i>
<i>LineaDeVenta.cantidad</i>	<i>LineaDeVenta</i>
Todas las <i>LineasDeVenta</i> de la Venta actual	<i>Venta</i>

A continuación presentamos un análisis detallado:

- ¿Quién deber ser responsable del cálculo del total de la *Venta*? Segundo el Experto, debería ser la propia *Venta*, puesto que conoce todas las instancias de *LineaDeVenta* cuyos subtotales se deben sumar para calcular el total de la venta. Por tanto, la *Venta* tendrá la responsabilidad de conocer su total, implementada como un método *getTotal*.

- Para que una *Venta* calcule su total, necesita el subtotal de cada *LíneaDeVenta*. ¿Quién debe ser responsable de crear el subtotal de la *LíneaDeVenta*? Según el Experto, debería ser la propia *LíneaDeVenta*, puesto que conoce la cantidad y la *EspecificacionDelProducto* con la que está asociada. Por tanto, la *LíneaDeVenta* tendrá la responsabilidad de conocer su subtotal, implementada como un método *getSubtotal*.
- Para que una *LíneaDeVenta* calcule su subtotal, necesita el precio de la *EspecificacionDelProducto*. ¿Quién debería ser responsable de proporcionar el precio de la *EspecificacionDelProducto*? Según el Experto, debería ser la propia *EspecificacionDelProducto*, puesto que encapsula el precio como un atributo. Por tanto, la *EspecificacionDelProducto* tendrá la responsabilidad de conocer su precio, implementada como una operación *getPrecio*.

Aunque el análisis anterior es trivial en este caso, y el atormentador grado de elaboración presentado está fuera de lugar para el diseño que nos ocupa, la misma estrategia de razonamiento para encontrar un Experto puede y debe aplicarse en situaciones más difíciles. Descubrirá que una vez que aprenda estos principios puede hacer rápidamente esta clase de razonamiento mentalmente.

El diseño de Venta--*getTotal*

Dada la anterior discusión, es conveniente ahora construir un diagrama de interacción que ilustre lo que ocurre cuando se envía el mensaje *getTotal* a la *Venta*. El primer mensaje de este diagrama es *getTotal*, pero observe que el mensaje *getTotal* no es un evento del sistema.

Eso nos lleva a la siguiente observación:

No todos los diagramas de interacción comienzan con un mensaje de evento del sistema; pueden comenzar con cualquier mensaje para el que el diseñador desee mostrar las interacciones.

El diagrama de interacción se muestra en la Figura 17.11. Primero se envía el mensaje *getTotal* a la instancia de *Venta*. La *Venta* entonces enviará un mensaje *getSubtotal* a cada instancia de *LíneaDeVenta* relacionada. La *LíneaDeVenta* enviará sucesivamente un mensaje *getPrecio* a las instancias *EspecificacionDelProducto* asociadas.

Puesto que la aritmética (usualmente) no se ilustra mediante mensajes, los detalles de los cálculos se pueden ilustrar adjuntando al diagrama algoritmos o restricciones que definan los cálculos.

¿Quién enviará el mensaje *getTotal* a la *Venta*? Lo más probable es que sea un objeto de la capa de UI, como un *JFrame* de Java.

Obsérvese en la Figura 17.12 el uso de las notas de algoritmos y restricciones, para exponer los detalles de *getTotal* y *getSubtotal*.

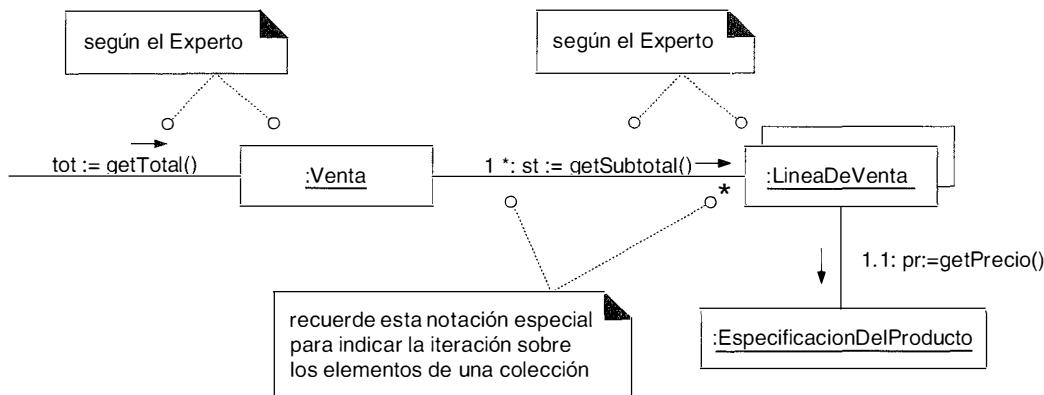


Figura 17.11. Diagrama de interacción Venta--getTotal.

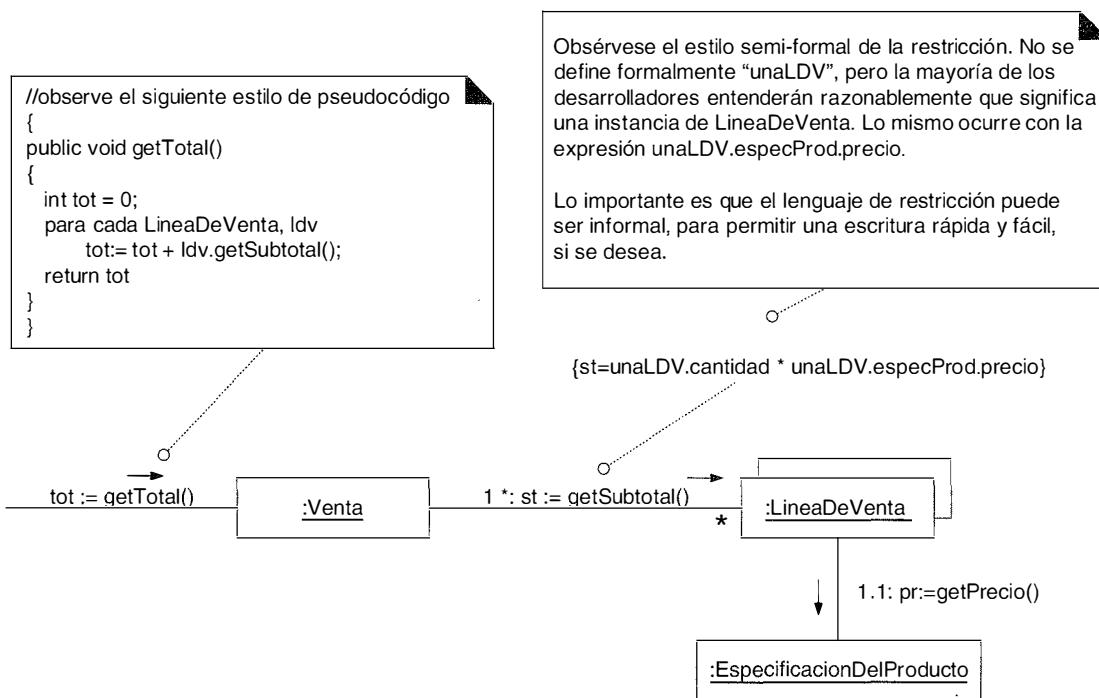


Figura 17.12. Notas de algoritmos y restricciones.

17.7. Diseño de objetos: realizarPago

La operación del sistema *realizarPago* tiene lugar cuando un cajero introduce la cantidad de dinero entregada para el pago. A continuación presentamos el contrato completo:

Contrato CO4: realizarPago

Operación:	realizarPago(cantidad: Dinero)
Referencias cruzadas:	Caso de Uso: Procesar Venta

Precondiciones:	Hay una venta en curso
Postcondiciones:	<ul style="list-style-type: none"> - Se creó una instancia de Pago <i>p</i> (creación de instancias). - <i>p.cantidadEntregada</i> pasó a ser cantidad (modificación de atributos). - <i>p</i> se asoció con la Venta actual (formación de asociaciones). - La Venta actual se asoció con la Tienda (formación de asociaciones); (para añadirlo al registro histórico de las ventas completadas).

Se construirá un diseño que satisfaga la postcondición de *realizarPago*.

Creación del Pago

Una de las postcondiciones del contrato establece:

- Se creó una instancia de *Pago p* (creación de instancias).

Ésta es una responsabilidad de creación, de manera que se debería aplicar el patrón GRASP Creador.

¿Quién registra, agrega, utiliza más estrechamente, o contiene un *Pago*? Existe un cierto atractivo en establecer que un *Registro* lógicamente registra un *Pago*, porque en el dominio real un “registro” recopila la información contable, luego es un candidato según el objetivo de reducir el salto en la representación en el diseño del software. Adicionalmente, es razonable esperar que una *Venta* software utilizará estrechamente un *Pago*; por tanto, podría ser una candidata.

Otra forma de encontrar un creador es utilizar el patrón Experto en función de quién es el Experto en Información con respecto a los datos de inicialización —la cantidad entregada en este caso—. El *Registro* es el controlador que recibe el mensaje de la operación del sistema *realizarPago*, luego, inicialmente tendrá la cantidad entregada. En consecuencia, el *Registro* es de nuevo un candidato.

Resumiendo, hay dos candidatos:

- *Registro*.
- *Venta*.

Ahora, esto nos lleva a una idea de diseño clave:

Cuando hay elecciones de diseño alternativas, mire detenidamente las implicaciones en cuanto a la cohesión y el acoplamiento de las alternativas, y posiblemente también considere las futuras presiones de evolución de las alternativas. Elija una alternativa con buena cohesión, acoplamiento y estabilidad ante posibles cambios futuros.

Considere algunas de las implicaciones de estas elecciones en función de los patrones GRASP Alta Cohesión y Bajo Acoplamiento. Si se elige la *Venta* para crear el *Pago*, es más ligero el trabajo (o responsabilidades) del *Registro* —dando lugar a una defini-

ción de *Registro* más simple—. También, el *Registro* no necesita conocer la existencia de una instancia de *Pago* porque se puede registrar indirectamente por medio de la *Venta* —lo que produce una disminución del acoplamiento en el *Registro*—. Esto nos lleva al diseño que se muestra en la Figura 17.13.

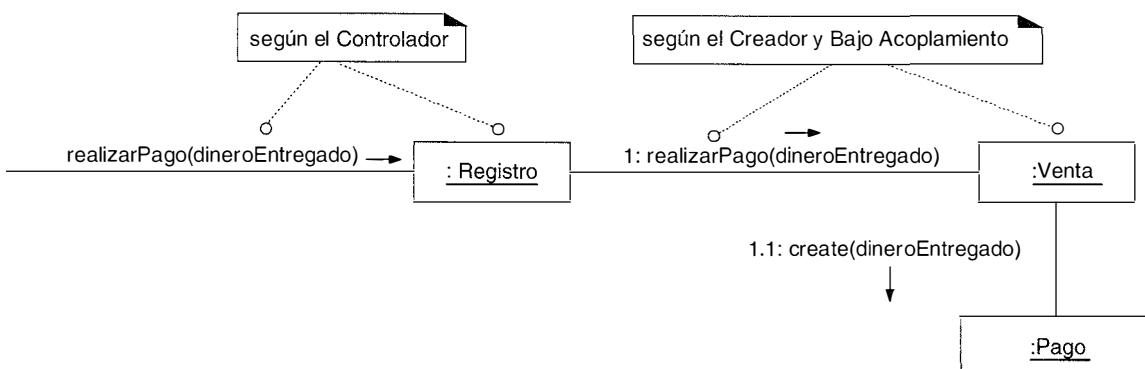


Figura 17.13. Diagrama de interacción Registro--realizarPago.

Este diagrama de interacción satisface la postcondición del contrato: se ha creado el *Pago*, asociado con la *Venta*, y se asigna el valor a su *cantidadEntregada*.

Registro de una Venta

Una vez completada, los requisitos establecen que la venta se debería colocar en un registro de históricos. Como siempre, el Experto en Información debería ser uno de los primeros patrones a tener en cuenta a menos que sea un problema de controlador o de creación (que no lo es), y se debería establecer la responsabilidad:

¿Quién es el responsable de conocer todas las ventas registradas y hacer el apunte en el registro?

Según el objetivo de salto en la representación bajo en el diseño del software (en relación con nuestros conceptos del dominio) es razonable que una *Tienda* conozca todas las ventas registradas, puesto que están fuertemente relacionadas con sus asuntos financieros. Otras alternativas comprenden conceptos clásicos de contabilidad, como *LibroMayorDeVentas*. Tiene sentido utilizar un objeto *LibroMayorDeVentas* cuando el diseño crece y la tienda pierde la cohesión (ver Figura 17.14).

Nótese también que la postcondición del contrato indica que se relacione la *Venta* con la *Tienda*. Éste es un ejemplo donde las postcondiciones podría no ser lo que realmente queremos conseguir en el diseño. Quizás no pensamos en el *LibroMayorDeVentas* antes, pero ahora que lo tenemos, elegimos usarlo en lugar de una *Tienda*. Si éste fuera el caso, idealmente se añadiría también el *LibroMayorDeVentas* al Modelo del Dominio, ya que es un nombre de un concepto en el dominio del mundo real. Son de esperar este tipo de descubrimientos y cambios durante el trabajo de diseño.

En este caso, nos mantendremos fieles al plan original de utilizar la *Tienda* (ver Figura 17.15).

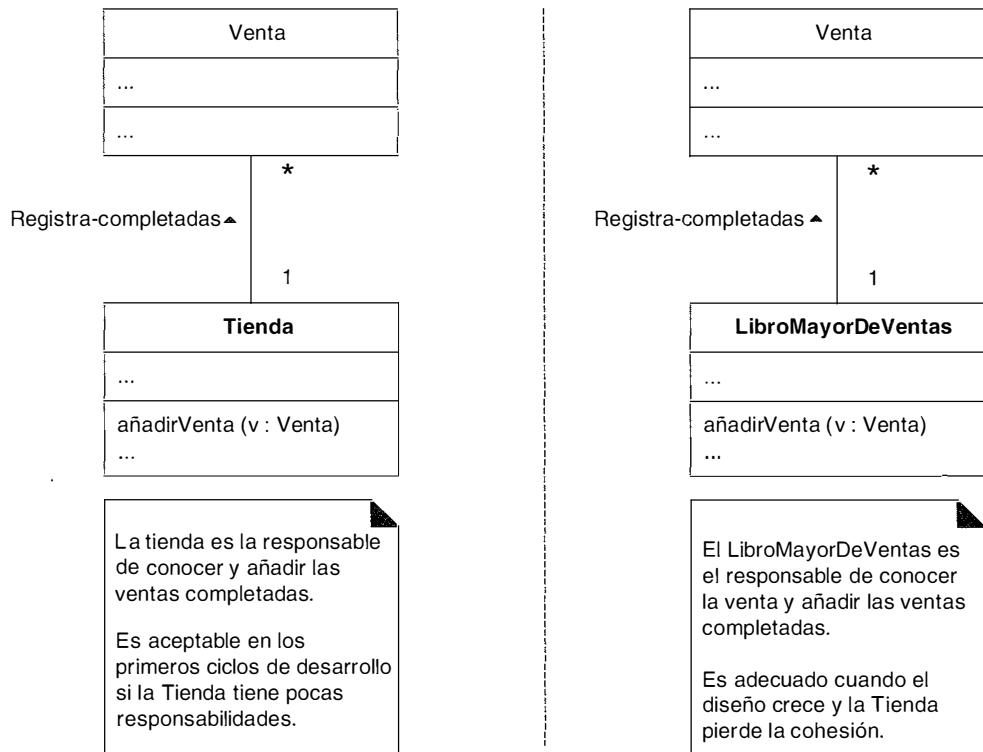


Figura 17.14. ¿Quién debería ser responsable de conocer la venta completada?

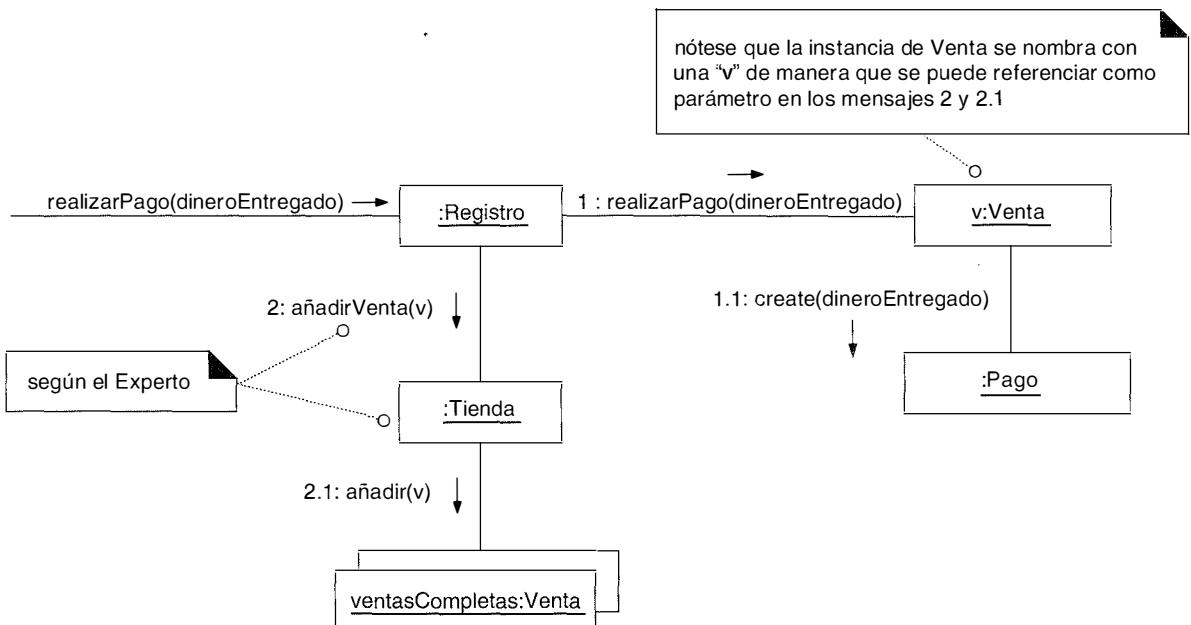


Figura 17.15. Registro de una venta completada.

Cálculo de la devolución

El caso de uso *Procesar Venta* implica que se imprima el saldo deudor a partir de un pago en un recibo y se muestre por pantalla de algún modo.

Debido al principio de Separación Modelo-Vista, no deberíamos preocuparnos por el modo en el que se visualizará o imprimirá el dinero que hay que devolver, pero es necesario asegurar que se conoce. Nótese que ninguna clase conoce actualmente la devolución, de manera que necesitamos crear un diseño de interacciones de objetos que satisfaga estos requisitos.

Como siempre, se debería tener en cuenta el Experto en Información a menos que sea un problema de controlador o creación (que no lo es), y se debería establecer la responsabilidad:

¿Quién es el responsable de conocer la devolución?

Para calcular la devolución, se requiere el total de la venta y el dinero en efectivo entregado. Por tanto, la *Venta* y el *Pago* son Expertos parciales en la solución del problema.

Si el *Pago* es ante todo responsable de conocer la devolución, necesitará tener visibilidad de la *Venta*, para pedirle su total. Puesto que actualmente no sabe nada de la *Venta*, este enfoque incrementa el acoplamiento global en el diseño —no soportaría el patrón de Bajo Acoplamiento—.

En cambio, si la *Venta* es principalmente responsable de conocer la devolución, necesita tener visibilidad del *Pago*, para solicitarle el dinero en efectivo entregado. Puesto que la *Venta* ya tiene visibilidad del *Pago* —como su creador— este enfoque no incrementa el acoplamiento global y, por tanto, es preferible este diseño.

En consecuencia, el diagrama de interacción de la Figura 17.16 proporciona una solución para conocer la cantidad a devolver.

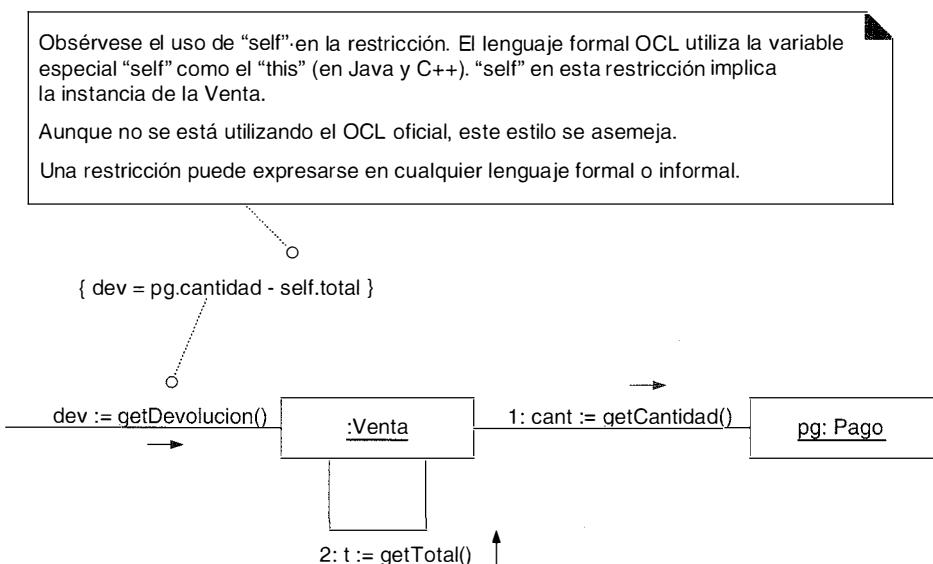


Figura 17.16. Diagrama de interacción *Venta--getDevolucion*.

17.8. Diseño de objetos: ponerEnMarcha

¿Cuándo crear el diseño de ponerEnMarcha?

La mayoría, si no todos, los sistemas tienen el caso de uso *ponerEnMarcha*, y alguna operación del sistema inicial relacionada con el comienzo de la aplicación. Aunque esta operación del sistema *ponerEnMarcha* es la primera que se va a ejecutar, posponga el desarrollo de un diagrama de interacción para ella hasta que se hayan tenido en cuenta todas las otras operaciones del sistema. Esto asegura que se haya descubierto la información relativa a las actividades de inicialización que se requieren para soportar los diagramas de interacción de operaciones del sistema posteriores.

Haga el diseño de la inicialización en último lugar.

Cómo comienzan las aplicaciones

La operación *ponerEnMarcha* representa de manera abstracta la fase de inicialización de la ejecución cuando se lanza una aplicación. Para entender cómo diseñar un diagrama de interacción para esta operación, es útil entender el contexto en el que puede ocurrir la inicialización. El modo en el que una aplicación comienza y se inicializa depende del lenguaje de programación y del sistema operativo.

En todos los casos, un estilo de diseño común es crear en último término un **objeto del dominio inicial**, que es el primer objeto software del “dominio” que se crea.

Una nota sobre la terminología: como se verá, las aplicaciones se organizan en capas lógicas que separan los aspectos más importantes de la aplicación. Esto comprende la capa de UI (para las cuestiones de UI) y una capa del “dominio” (para cuestiones de la lógica del dominio). La capa del dominio del Modelo de Diseño está formada por las clases software cuyos nombres están inspirados en el vocabulario del dominio, y que contienen la lógica de la aplicación. Prácticamente todos los objetos del diseño que hemos considerado, como *Venta* y *Registro*, son objetos del dominio en la capa del dominio del Modelo de Diseño.

El objeto de dominio inicial, una vez creado, es el responsable de la creación de los objetos del dominio que son sus “hijos”. Por ejemplo, si se elige una *Tienda* como el objeto del dominio inicial, podría ser el responsable de la creación de un objeto *Registro*.

El lugar donde se crea este objeto del dominio inicial depende de la tecnología de objetos escogida. Por ejemplo, en una aplicación Java, podría crearlo el método *main*, o delegar el trabajo al objeto *fabrica* que lo crea.

```
public class Main
{
    public static void main(String [] args)
    {
        //La Tienda es el objeto del dominio inicial.
        //La Tienda crea algún otro objeto del dominio.
```

```

Tienda tienda = new Tienda();
Registro registro = tienda.getRegistro();
JFrameProcesarVenta frame = new JFrameProcesarVenta(registro);
...
}
}

```

Interpretación de la operación del sistema ponerEnMarcha

La discusión anterior ilustra que la operación del sistema *ponerEnMarcha* es una abstracción independiente del lenguaje. Durante el diseño, existen variaciones en cuanto al lugar de creación del objeto inicial, y si controla o no el proceso. El objeto del dominio inicial no suele tomar el control si se trata de una GUI; en otro caso, lo hace con frecuencia.

Los diagramas de operación para la operación *ponerEnMarcha* representan lo que ocurre cuando se crea el objeto inicial del dominio del problema, y opcionalmente lo que sucede si toma el control. No incluyen ninguna actividad anterior o siguiente de los objetos en la capa de GUI, si existe alguno.

Por tanto, la operación *ponerEnMarcha* puede reinterpretarse como:

1. En un diagrama de interacción, envíe un mensaje *create()* para crear el objeto de dominio inicial.
2. (opcional) Si el objeto inicial toma el control del proceso, en un segundo diagrama de interacción, envíe el mensaje *ejecutar* (o algo equivalente) al objeto inicial.

La operación PonerEnMarcha de la aplicación del PDV

La operación del sistema *ponerEnMarcha* tiene lugar cuando un responsable de la tienda enciende el sistema de PDV y se carga el software. Asuma que el objeto del dominio inicial *no* es responsable de controlar el proceso; el control permanecerá en la capa de UI (como un *JFrame* de Java) después de que se cree el objeto del dominio inicial. Por tanto, el diagrama de interacción para la operación *ponerEnMarcha* podría reinterpretarse únicamente como el envío del mensaje *create()* para crear el objeto inicial.

Elección del objeto del dominio inicial

¿Cuál debería ser la clase del objeto del dominio inicial?

Elija como objeto del dominio inicial la clase de la raíz de la jerarquía de agregación o contención, o cercana a ella. Esto podría ser un controlador de fachada, como un *Registro*, o algún otro objeto que se considera que contiene todos o la mayoría de los objetos, como una *Tienda*.

Las consideraciones de Alta Cohesión y Bajo Acoplamiento podrían influir en la elección entre las alternativas. En esta aplicación, se elige la *Tienda* como el objeto inicial.

Objetos persistentes: EspecificacionDelProducto

Las instancias de *EspecificacionDelProducto* residirán en un medio de almacenamiento persistente, como una base de datos relacional u objetual. Durante la operación *ponerEnMarcha*, si sólo hay unos pocos de estos objetos, se podrían cargar todos en la memoria principal del ordenador. Sin embargo, si hay muchos, cargarlos todos consumiría demasiada memoria o tiempo. Alternativamente —y más probable— se cargarán en memoria bajo demanda las instancias individuales cuando se requieran.

El diseño de la manera de cargar dinámicamente bajo demanda los objetos desde una base de datos a memoria es sencilla si se utiliza una base de datos objetual, pero difícil para una base de datos relacional. Este problema se pospone por ahora y se asume que todas las instancias de *EspecificacionDelProducto* pueden ser creadas en memoria “mágicamente” por el objeto *CatalogoDeProductos*.

El Capítulo 34 estudia el problema de los objetos persistentes y el modo de cargarlos en memoria.

Diseño de Tienda--create()

Las tareas de creación e inicialización se derivan a partir de las necesidades del trabajo de diseño anterior, como el diseño de la gestión de *introducirArticulo*, etcétera. Reflexionando sobre los diseños de interacciones previos, se puede identificar el siguiente trabajo de inicialización:

- Se necesita crear una *Tienda*, *Registro*, *CatalogoDeProductos* y objetos *EspecificacionDelProducto*.
- Se necesitan asociar los objetos *EspecificacionDelProducto* con el *CatalogoDeProductos*.
- Se necesita asociar la *Tienda* con el *CatalogoDeProductos*.
- Se necesita asociar la *Tienda* con el *Registro*.
- Se necesita asociar el *Registro* con el *CatalogoDeProductos*.

La Figura 17.17 muestra un diseño. Se escogió la *Tienda* para crear el *CatalogoDeProductos* y el *Registro* según el patrón Creador. Del mismo modo, se eligió el *CatalogoDeProductos* para crear los objetos *EspecificacionDelProducto*. Recuerde que este enfoque de creación de las especificaciones es temporal. En el diseño final, se materializará desde una base de datos, cuando sea necesario.

Notación UML: Obsérvese que la creación de todas las instancias de *EspecificacionDelProducto* y su inclusión en un contenedor tienen lugar en una sección de repetición, que se indica con el “*” a continuación de los números de secuencia.

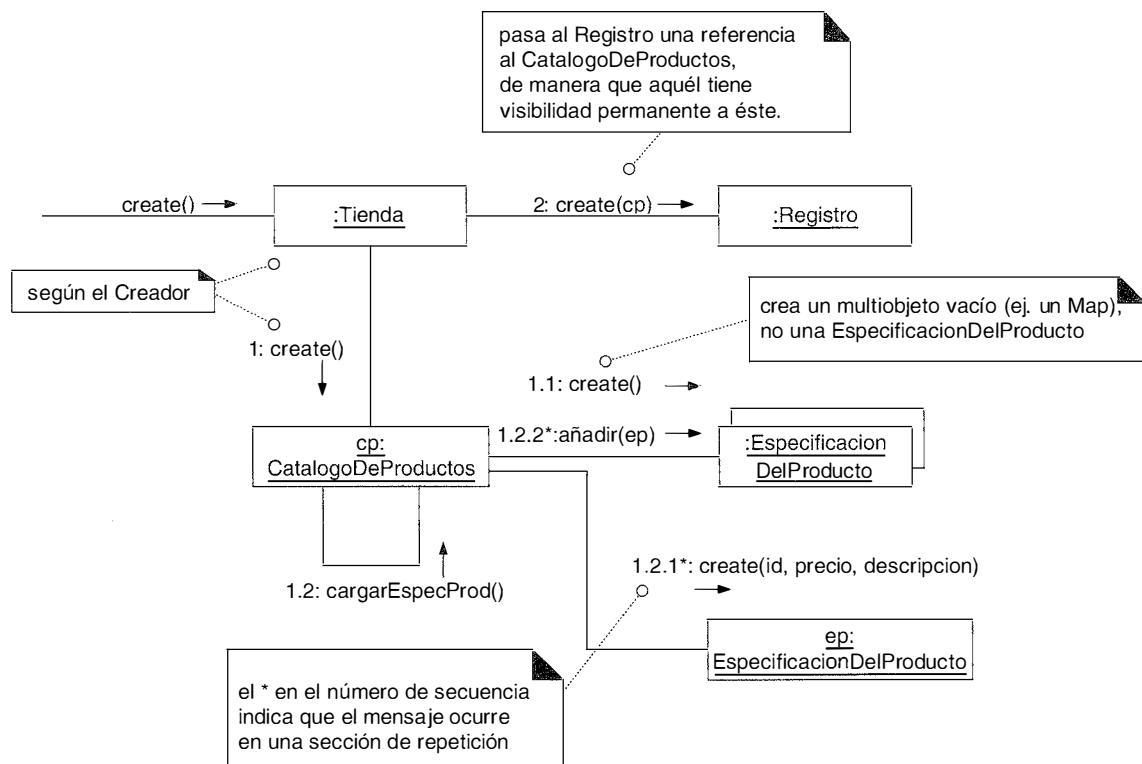


Figura 17.17. Creación del objeto del dominio inicial y los objetos siguientes.

Una desviación interesante entre el modelado del mundo real y el diseño se ilustra en el hecho de que el objeto software *Tienda* sólo crea *un* objeto *Registro*. Una tienda real podría albergar *muchos* registros o terminales de PDV reales. Sin embargo, estamos considerando un diseño software, no la vida real. En nuestros requisitos actuales, nuestra *Tienda* software sólo necesita crear una única instancia de un *Registro* software.

La multiplicidad entre las clases de objetos del Modelo del Dominio y el Modelo del Diseño podría no ser la misma.

17.9. Conexión de la capa de UI con la capa del dominio

Como se ha discutido brevemente, las aplicaciones se organizan en capas lógicas que separan los aspectos más importantes de la aplicación, como la capa de UI (para las cuestiones de la UI) y una capa de “dominio” (para las cuestiones de la lógica del dominio).

Entre los diseños típicos según los cuales los objetos de la capa del dominio son visibles a los objetos de la capa de la UI encontramos los siguientes:

- Una rutina de inicialización (por ejemplo, un método *main* en Java) crea tanto un objeto de la UI como un objeto del dominio, y pasa el objeto del dominio a la UI.

- Un objeto de la UI recupera el objeto del dominio de una fuente bien conocida, como un objeto factoría que es responsable de la creación de los objetos del dominio.

La muestra de código que se presentó anteriormente es un ejemplo del primer enfoque:

```
public class Main
{
    public static void main (String[] args)
    {
        Tienda tienda = new Tienda();
        Registro registro = tienda.getRegistro();
        JFrameProcesarVenta frame = new JFrameProcesarVenta(registro);
        ...
    }
}
```

Una vez que el objeto de la UI está conectado a la instancia del *Registro* (el controlador de fachada en este diseño), puede reenviarle mensajes de eventos del sistema, como los mensajes *introducirArticulo* y *finalizarVenta* (ver Figura 17.18).

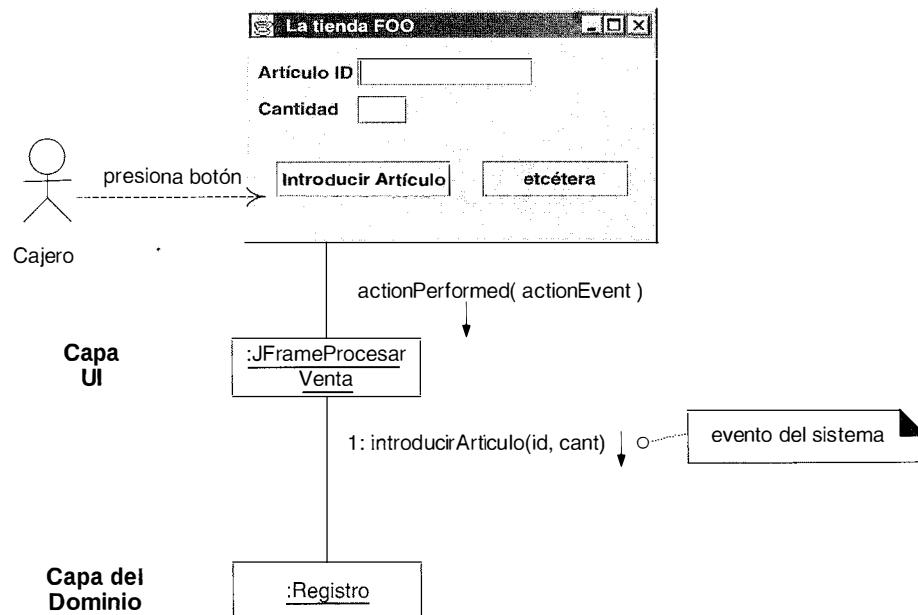


Figura 17.18. Conexión de las capas de UI y del dominio.

En el caso del mensaje *introducirArticulo*, la ventana necesita mostrar la suma parcial después de cada entrada. Hay varias soluciones de diseño:

- Añadir el método *getTotal* al *Registro*. La UI envía el mensaje *getTotal* al *Registro*, quien lo reenvía a la *Venta*. Esto tiene la posible ventaja de que mantiene bajo acoplamiento entre la UI y el modelo del dominio —la UI sólo conoce al objeto *Re-*

gistro—. Pero comienza a expandir el interfaz del objeto *Registro*, haciéndolo menos cohesivo.

- Una UI solicita una referencia al objeto *Venta* actual, y entonces cuando necesita el total (o cualquier otra información relacionada con la venta), envía el mensaje directamente a la *Venta*. Este diseño incrementa el acoplamiento entre la UI y el modelo del dominio. Sin embargo, como se estudió en la discusión del patrón GRASP Bajo Acoplamiento, un mayor acoplamiento por sí mismo no es un problema; más bien, en especial constituye un problema el acoplamiento entre cosas inestables. Asuma que decidimos que la *Venta* es un objeto estable que será una parte integral del diseño —lo que es muy razonable—. Entonces, el acoplamiento con la *Venta* no es un problema.

Como se ilustra en la Figura 17.19, este diseño sigue el segundo enfoque.

Nótese que en estos diagramas la ventana Java (*JFrameProcesarVenta*), que forma parte de la capa de UI, no es responsable de manejar la lógica de la aplicación. La ventana remite las solicitudes de trabajo (las operaciones del sistema) a la capa del dominio, por medio del *Registro*. Esto nos lleva al siguiente principio de diseño:

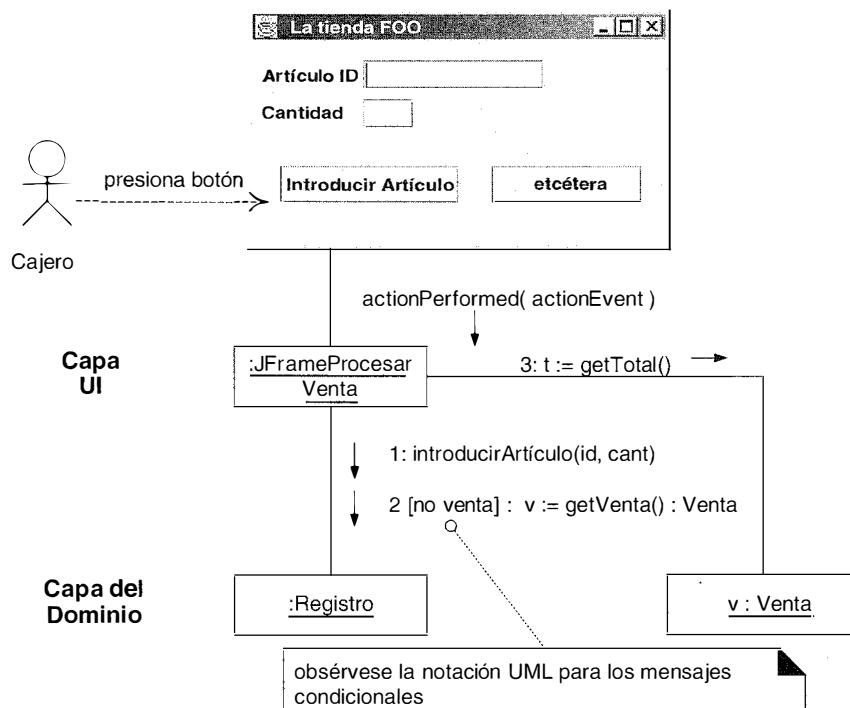


Figura 17.19. Conexión de las capas de la UI y del dominio.

Responsabilidades de la capa del dominio y de interfaz

La capa de UI no debería tener ninguna responsabilidad de la lógica del dominio. Sólo debería ser responsable de las tareas de la interfaz de usuario, como actualizar los elementos gráficos.

La capa de UI debería remitir las solicitudes de las tareas orientadas al dominio a la capa del dominio, que es la responsable de manejarlas.

17.10. Realizaciones de casos de uso en el UP

Las realizaciones de los casos de uso forman parte del Modelo de Diseño del UP. Este capítulo ha resaltado la elaboración de los diagramas de interacción, pero es común y se recomienda que se elaboren los diagramas de clases en paralelo. Los diagramas de clases se estudiarán en el Capítulo 19.

Tabla 17.1. Muestra de los artefactos UP y evolución temporal. c – comenzar; r – refiniar.

Disciplina	Artefacto Iteración →	Inicio I1	Elab. E1...En	Const. C1...Cn	Trans. T1...T2
Modelado del Negocio	Modelo del Dominio		c		
Requisitos	Modelo de Casos de Uso (DSS) Visión Especificación complementaria Glosario	c c c c	r r r r		
Diseño	Modelo de Diseño Documento de Arquitectura SW Modelo de Datos		c c c	r r	
Implementación	Modelo de Implementación		c	r	r
Gestión del Proyecto	Plan de Desarrollo SW	c	r	r	r
Pruebas	Modelo de Pruebas		c	r	
Entorno	Marco de Desarrollo	c	r		

Fases

Inicio: El Modelo de Diseño y las realizaciones de los casos de uso normalmente no comenzarán hasta la elaboración porque comprende decisiones de diseño detalladas que son prematuras durante la fase de inicio.

Elaboración: Durante esta fase, podrían crearse las realizaciones de los casos de uso para los escenarios más significativos desde el punto de vista de la arquitectura o de más riesgo del diseño. Sin embargo, no se harán los diagramas de UML para cada escenario, y no necesariamente con todo detalle o de grano fino. La idea es realizar los diagramas de interacción para las realizaciones de los casos de uso claves que se benefician de algún estudio anticipado y exploración de alternativas, centrándose en las decisiones de diseño más importantes.

Construcción: Se crean las realizaciones de los casos de uso para el resto de problemas de diseño.

En el UP, el trabajo de la realización de los casos de uso es una actividad de diseño. La Figura 17.21 ofrece sugerencias sobre el momento y el lugar para llevar a cabo este trabajo.

17.11. Resumen

Lo esencial de un diseño de objetos lo constituyen el diseño de las interacciones de objetos y la asignación de responsabilidades. Las decisiones que se tomen pueden influir

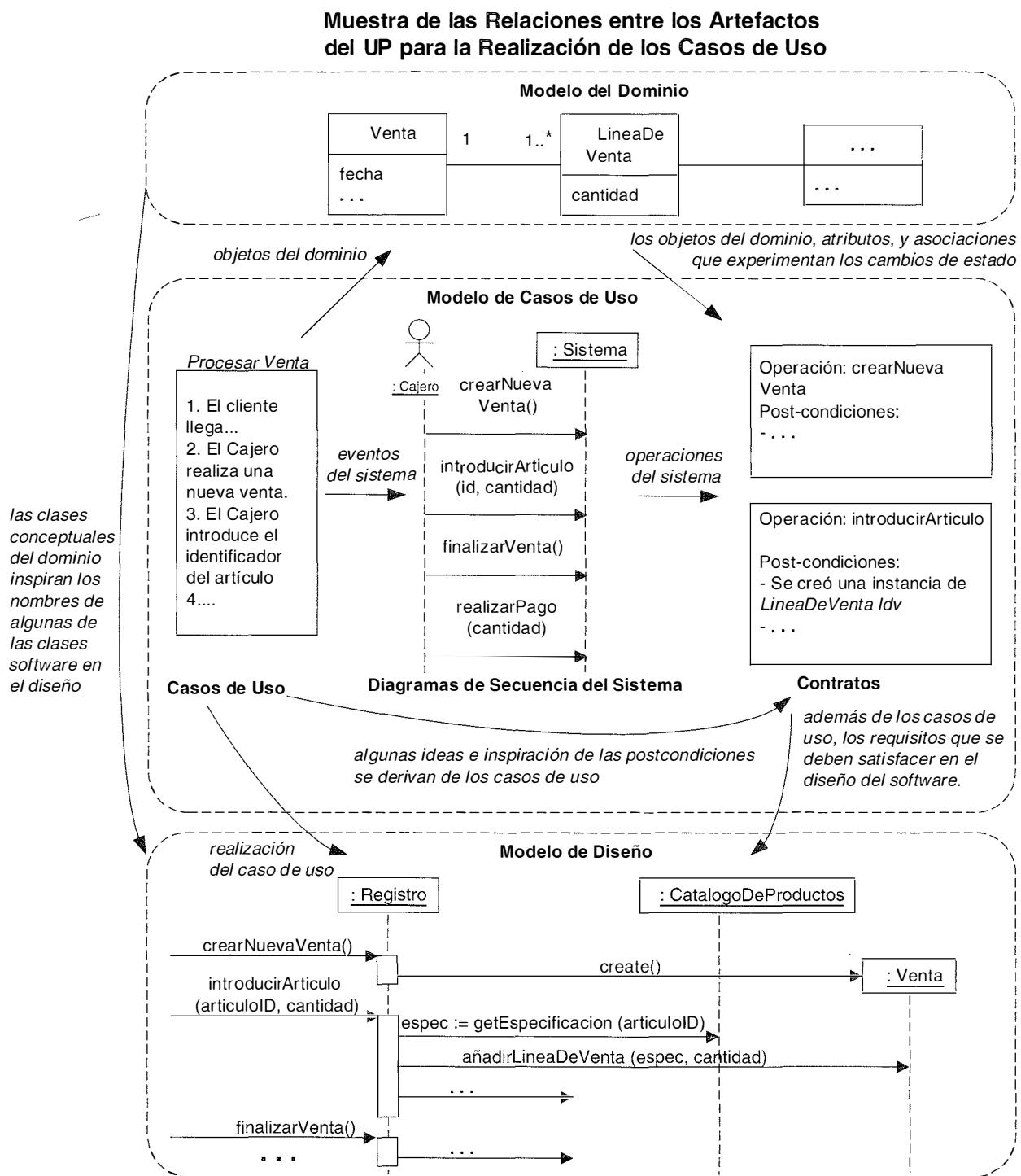


Figura 17.20. Muestra de la influencia entre los artefactos UP.

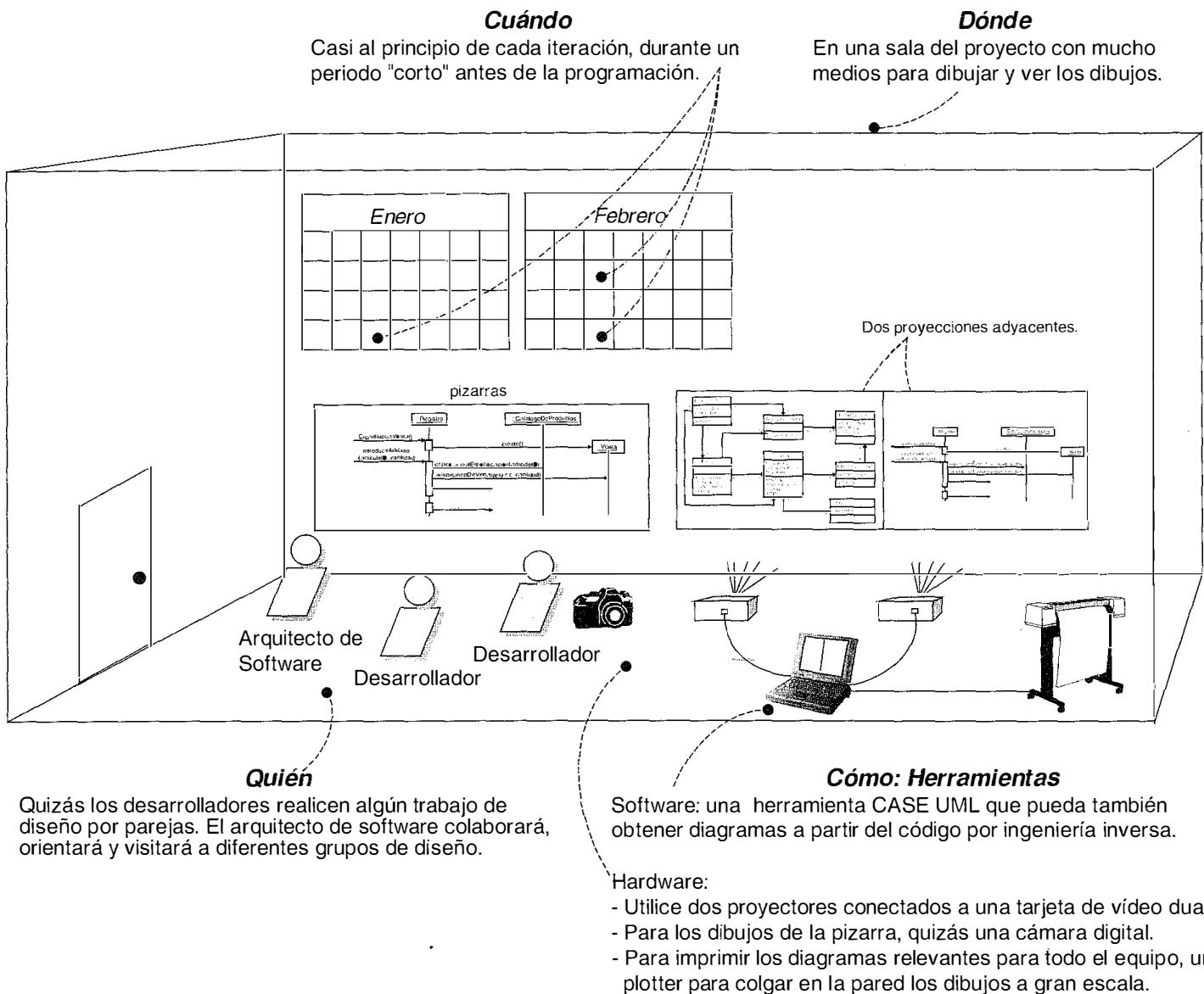


Figura 17.21. Proceso y establecimiento del contexto.

profundamente en la extensibilidad, claridad y mantenimiento del sistema software de objetos, además de en el grado y calidad de los componentes reutilizables. Existen principios que se pueden seguir para tomar las decisiones en cuanto a la asignación de responsabilidades; los patrones GRASP resumen algunos que los diseñadores orientados a objetos utilizan con frecuencia y están generalizados.

Capítulo 18

MODELO DE DISEÑO: DETERMINACIÓN DE LA VISIBILIDAD

Un matemático es una máquina que transforma café en teoremas.

Paul Erdős

Objetivos

- Identificar cuatro tipos de visibilidad.
 - Diseñar para establecer la visibilidad.
 - Ilustrar los tipos de visibilidad en la notación UML.
-

Introducción

La visibilidad es la capacidad de un objeto de ver o tener una referencia a otro. Este capítulo estudia las cuestiones de diseño relacionadas con la visibilidad.

18.1. Visibilidad entre objetos

Los diseños creados para los eventos del sistema (*introducirArticulo*, etcétera) ilustran mensajes entre objetos. Para que un objeto emisor envíe un mensaje a un objeto receptor, el receptor debe ser visible al emisor —éste debe tener algún tipo de referencia o apuntador al objeto receptor—.

Por ejemplo, el mensaje *getEspecificacion* enviado desde un *Registro* a un *CatalogoDeProductos* implica que la instancia de *CatalogoDeProductos* es visible a la instancia de *Registro*, como se muestra en la Figura 18.1.

Cuando creamos un diseño de objetos que interaccionan, es necesario asegurar que se presenta la visibilidad adecuada para soportar la interacción de mensajes.

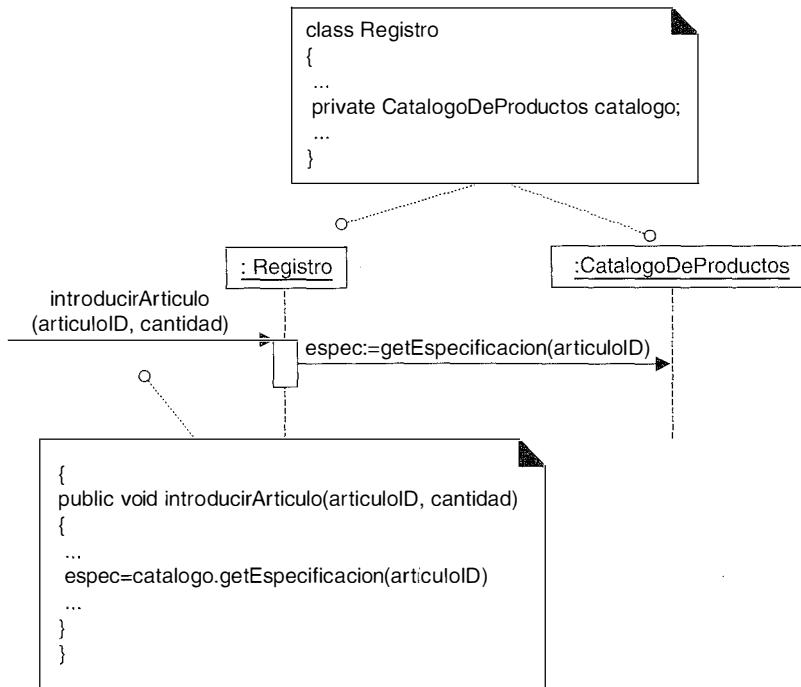


Figura 18.1. Se requiere visibilidad desde el Registro al CatalogoDeProductos¹.

UML tiene una notación especial para representar la visibilidad; este capítulo presenta varios tipos de visibilidad y su descripción.

18.2. Visibilidad

En su uso habitual, por **visibilidad** se entiende la capacidad de un objeto de “ver” o tener una referencia a otro objeto. De manera más general, está relacionado con el tema del alcance: ¿se encuentra un recurso (tal como una instancia) al alcance de otro? Hay cuatro formas comunes de alcanzar la visibilidad desde un objeto A a un objeto B:

- **Visibilidad de atributo:** B es un atributo de A.
- **Visibilidad de parámetro:** B es un parámetro de un método de A.
- **Visibilidad local:** B es un objeto local (no un parámetro) en un método de A.
- **Visibilidad global:** B es de algún modo visible globalmente.

El motivo para tener en cuenta la visibilidad es la siguiente:

Para que un objeto A envíe un mensaje a un objeto B, B debe ser visible a A.

Por ejemplo, para crear un diagrama de interacción en el que se envía un mensaje desde una instancia de `Registro` a una instancia de `CatalogoDeProductos`, el `Registro`

¹ En éste y en los siguientes ejemplos de código, podrían hacerse simplificaciones del lenguaje en aras de la brevedad y claridad.

debe tener visibilidad del *CatalogoDeProductos*. Una solución típica para la visibilidad es mantener una referencia a una instancia del *CatalogoDeProductos* como atributo del *Registro*.

Visibilidad de atributo

La **visibilidad de atributo** desde A a B existe cuando B es un atributo de A. Es una visibilidad relativamente permanente porque persiste mientras existan A y B. Ésta es una forma de visibilidad muy común en los sistemas orientados a objetos.

Para ilustrarlo, sea una definición de clase en Java para el *Registro*, una instancia de *Registro* tendría visibilidad de atributo a un *CatalogoDeProducto*, puesto que es un atributo (variable de instancia Java) del *Registro*.

```
public class Registro
{
...
private CatalogoDeProductos catalogo;
...
}
```

Se requiere esta visibilidad porque en el diagrama de *introducirArticulo* que se muestra en la Figura 18.2, un *Registro* necesita enviar el mensaje *getEspecificacion* a un *CatalogoDeProductos*:

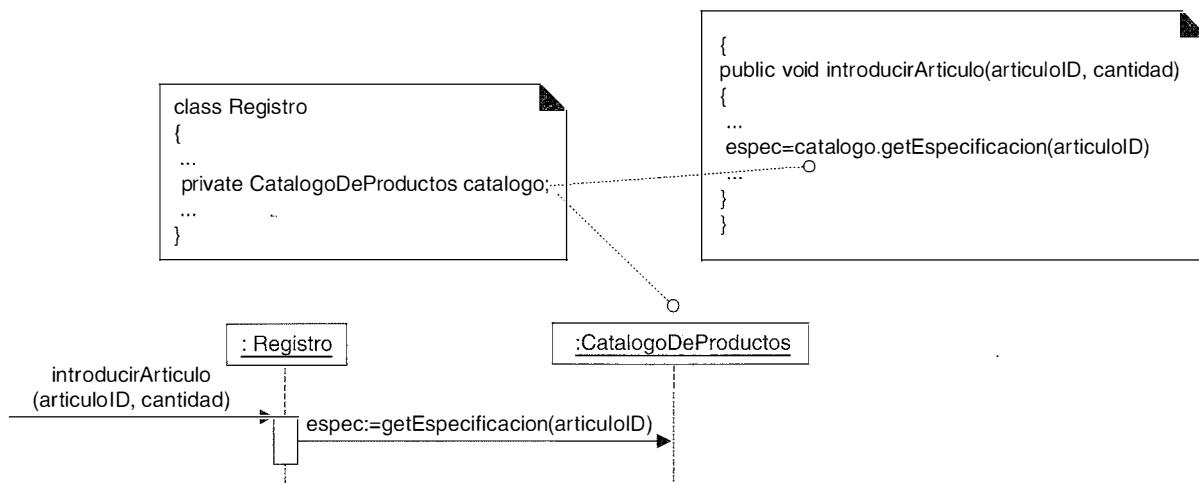


Figura 18.2. Visibilidad de atributo.

Visibilidad de parámetro

La **visibilidad de parámetro** desde A a B existe cuando B se pasa como parámetro a un método de A. Es una visibilidad relativamente temporal porque persiste sólo en el alcance del método. Después de la visibilidad de atributo, es la segunda forma más común de visibilidad en los sistemas orientados a objetos.

Para ilustrarlo, cuando se envía el mensaje *crearLineaDeVenta* a la instancia de *Venta*, se pasa como parámetro una instancia de *EspecificacionDelProducto*. En el alcance del método *crearLineaDeVenta*, la *Venta* tiene visibilidad de parámetro de una *EspecificacionDelProducto* (ver Figura 18.3).

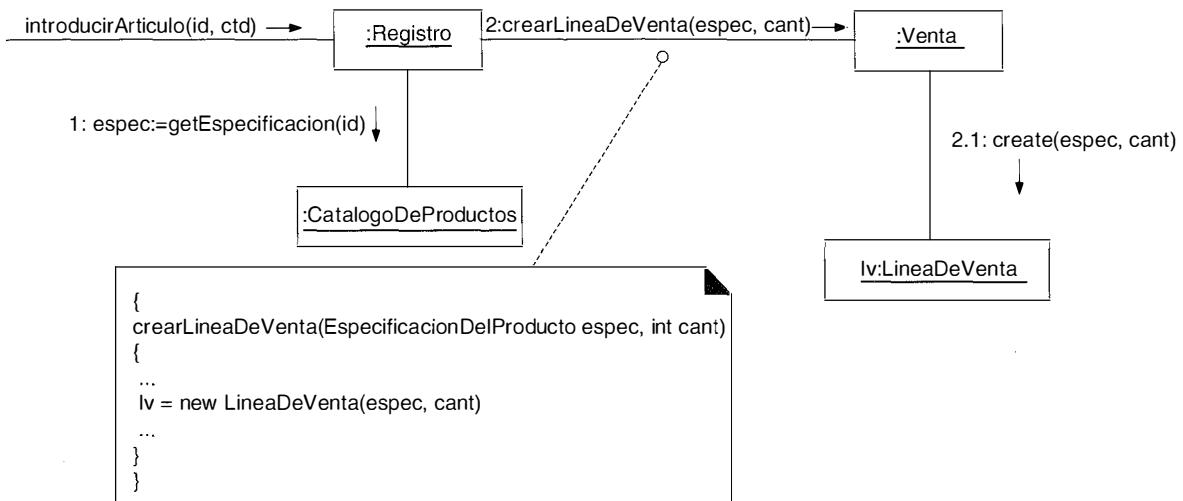


Figura 18.3. Visibilidad de parámetro.

Es habitual transformar la visibilidad de parámetro en visibilidad de atributo. Por ejemplo, cuando la *Venta* crea una nueva *LineaDeVenta*, le pasa en el método de inicialización una *EspecificacionDelProducto* (sería su **constructor** en Java y C++). En el método de inicialización, se asigna el parámetro a un atributo; por tanto, se establece visibilidad de atributo (ver Figura 18.4).

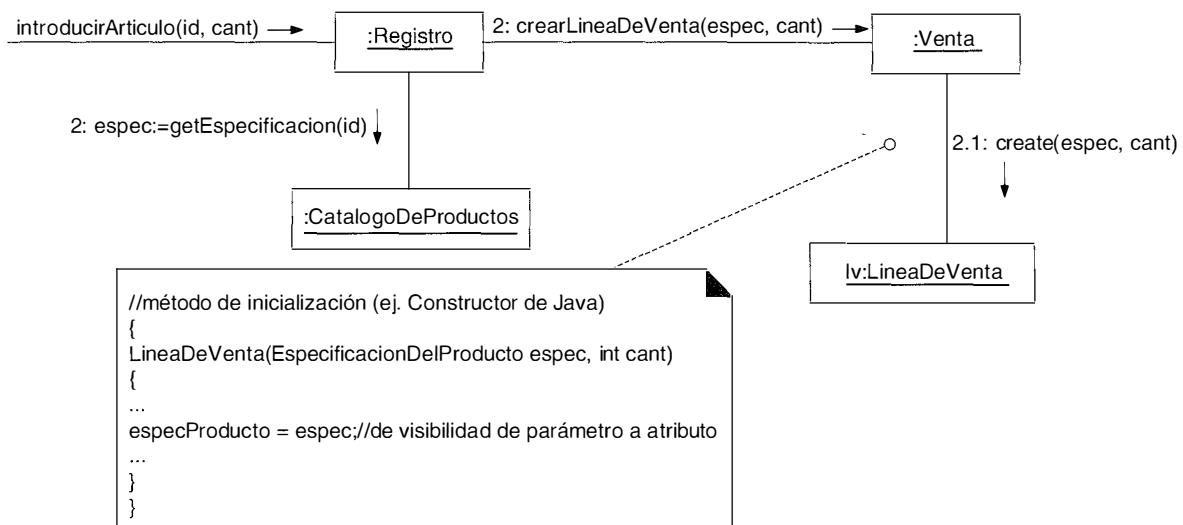


Figura 18.4. De visibilidad de parámetro a visibilidad de atributo.

Visibilidad local

La **visibilidad local** desde A a B existe cuando B se declara como un objeto local en un método de A. Es una visibilidad relativamente temporal porque sólo persiste en el alcance del método. Después de la visibilidad de parámetro, es la tercera forma de visibilidad más común en los sistemas orientados a objetos.

Dos medios comunes de alcanzar la visibilidad local son:

- Crear una nueva instancia local y asignarla a una variable local.
- Asignar a una variable local el objeto de retorno de la invocación a un método.

Como con la visibilidad de parámetro, es habitual transformar la visibilidad declarada localmente en visibilidad de atributo.

Un ejemplo de la segunda variación (asignación a una variable local del objeto de retorno) se puede encontrar en el método *introducirArticulo* de la clase *Registro* (Figura 18.5).

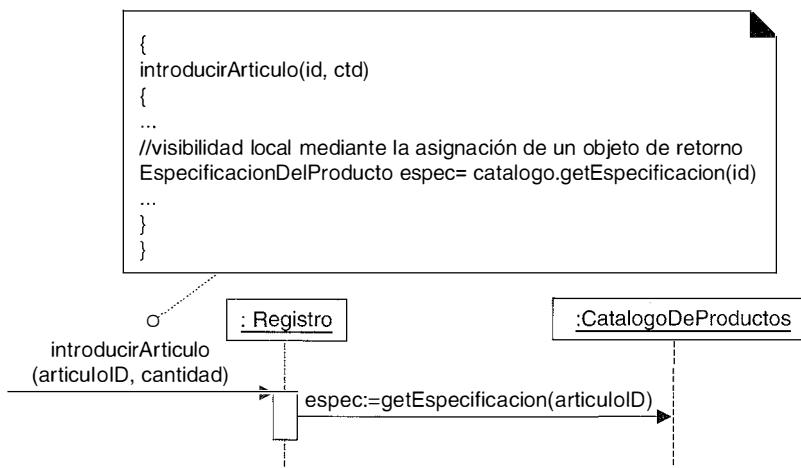


Figura 18.5. Visibilidad local.

Una versión sutil de la segunda variación es cuando el método no declara explícitamente una variable, sino que implícitamente existe una como resultado de un objeto de retorno de una invocación a un método. Por ejemplo:

```
//hay una visibilidad local implícita al objeto foo
//resultado de la llamada a getFoo
unObjeto.getFoo().hacerBar();
```

Visibilidad global

La **visibilidad global** de A a B existe cuando B es global a A. Es una visibilidad relativamente permanente porque persiste mientras existan A y B. Es la forma menos común de visibilidad en los sistemas orientados a objetos.

Un medio de conseguir visibilidad global es asignar una instancia a una variable global, lo que es posible en algunos lenguajes, como C++, pero no en otros, como Java.

El método que se prefiere para conseguir la visibilidad global es utilizar el patrón **Singleton** [GHJV95], que se presentará en un capítulo posterior.

18.3. Representación de la visibilidad en UML

UML incluye notación para representar el tipo de visibilidad en un diagrama de colaboración (ver Figura 18.6). Estos adornos son opcionales y normalmente no se exigen; son útiles cuando se necesita alguna aclaración.

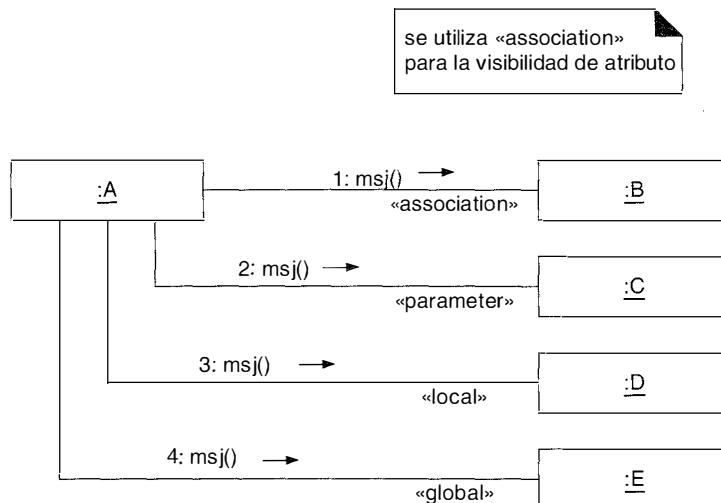


Figura 18.6. Implementación de los estereotipos para la visibilidad.

Capítulo 19

MODELO DE DISEÑO: CREACIÓN DE LOS DIAGRAMAS DE CLASES DE DISEÑO

Iterar es humano, ser recursico, divino.

Anónimo

Objetivos

- Crear los diagramas de clases de diseño (DCD).
 - Identificar las clases, métodos y asociaciones que se van a mostrar en un DCD.
-

Introducción

Una vez terminados los diagramas de interacción para las realizaciones de los casos de uso de la iteración actual de la aplicación de PDV NuevaEra, es posible identificar la especificación de las clases software (e interfaces) que participan en la solución software, y añadirles detalles de diseño, como los métodos.

UML proporciona la notación para representar los detalles de diseño en los diagramas de clase; en este capítulo, vamos a estudiarla y a crear los DCD.

19.1. Cuándo crear los DCD

Aunque esta presentación de los DCD viene después de la creación de los diagramas de interacción, en la práctica normalmente se crean en paralelo. Al comienzo del diseño se podrían esbozar muchas clases, nombres de métodos y relaciones aplicando los patrones para asignar responsabilidades, antes de la elaboración de los diagramas de interacción. Es posible y deseable elaborar algo de los diagramas de interacción, actualizar entonces los DCD, después extender los diagramas de interacción algo más, y así sucesivamente.

Estos diagramas de clases podrían utilizarse como una notación más gráfica alternativa a las tarjetas CRC para recoger las responsabilidades y colaboraciones.

19.2. Ejemplo de DCD

El DCD de la Figura 19.1 ilustra una definición software parcial de las clases *Registro* y *Venta*.

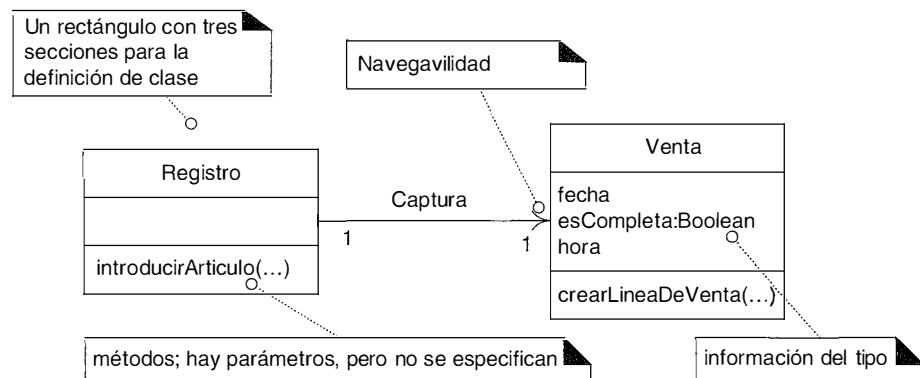


Figura 19.1. Ejemplo de diagrama de clases de diseño.

Además de las asociaciones y atributos básicos, el diagrama se amplía para representar, por ejemplo, los métodos de cada clase, información del tipo de los atributos, visibilidad de los atributos y navegación entre los objetos.

19.3. Terminología del DCD y el UP

Un **diagrama de clases de diseño** (DCD) representa las especificaciones de las clases e interfaces software (por ejemplo, las interfaces de Java) en una aplicación. Entre la información general encontramos:

- Clases, asociaciones y atributos.
- Interfaces, con sus operaciones y constantes.
- Métodos.
- Información acerca del tipo de los atributos.
- Navegabilidad.
- Dependencias.

A diferencia de las clases conceptuales del Modelo del Dominio, las clases de diseño de los DCD muestran las definiciones de las clases software en lugar de los conceptos del mundo real.

El UP no define de manera específica ningún artefacto denominado “diagrama de clases de diseño”. El UP define el Modelo de Diseño, que contiene varios tipos de diagramas, que incluye los diagramas de interacción, de paquetes, y los de clases. Los diagramas de clases del Modelo de Diseño del UP contienen “clases de diseño” en términos del UP. De ahí que sea común hablar de “diagramas de clases de diseño”, que es más corto que, e implica, “diagramas de clases en el Modelo de Diseño”.

19.4. Clases del Modelo de Dominio vs. clases del Modelo de Diseño

Seamos reiterativos, en el Modelo de Dominio del UP, una *Venta* no representa una definición software, sino que se trata de una abstracción de un concepto del mundo real acerca del cual estamos interesados en realizar una declaración. En cambio, los DCD expresan —para la aplicación software— la definición de las clases como componentes software. En estos diagramas, una *Venta* representa una clase software (ver Figura 19.2).

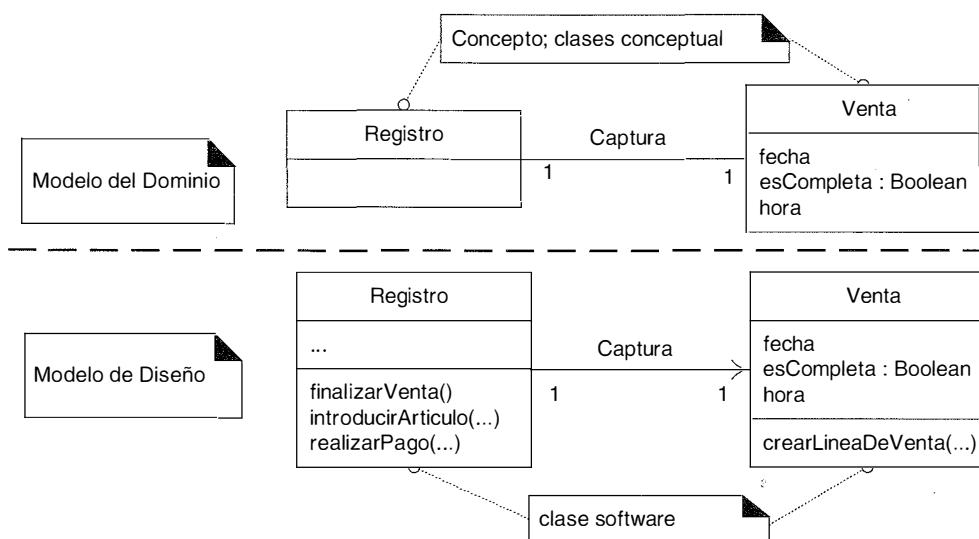


Figura 19.2. Clases del Modelo del Dominio vs. clases del Modelo de Diseño.

19.5. Creación de un DCD del PDV NuevaEra

Identificación y representación de las clases software

El primer paso en la creación de los DCD como parte del modelo de la solución es identificar aquellas clases que participan en la solución software. Se pueden encontrar examinando todos los diagramas de interacción y listando las clases que se mencionan.

Para la aplicación del PDV son:

Registro

Venta

CatalogoDeProductos

EspecificacionDelProducto

Tienda

LineaDeVenta

Pago

El siguiente paso es dibujar un diagrama de clases para estas clases e incluir los atributos que se identificaron previamente en el Modelo del Dominio que también se utilizan en el diseño (ver Figura 19.3).

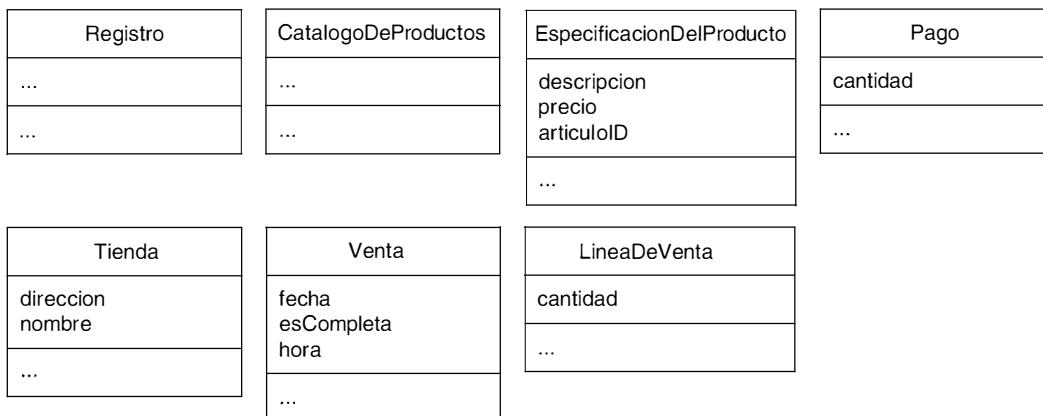


Figura 19.3. Clases software de la aplicación.

Nótese que algunos de los conceptos del Modelo del Dominio, como *Cajero*, no están presentes en el diseño. No es necesario —para la iteración actual— representarlos en el software. Sin embargo, en iteraciones posteriores, cuando se aborden nuevos requisitos y casos de uso, podrían formar parte del diseño. Por ejemplo, cuando se implementen los requisitos de seguridad y de inicio de sesión, es probable que sea relevante una clase software denominada *Cajero*.

Añadir los nombres de los métodos

Se pueden identificar los nombres de los métodos analizando los diagramas de interacción. Por ejemplo, si se envía el mensaje *crearLineaDeVenta* a una instancia de la clase *Venta*, entonces la clase *Venta* debe definir un método *crearLineaDeVenta* (ver Figura 19.4).

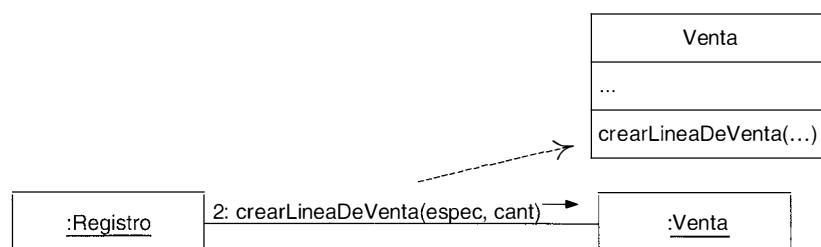


Figura 19.4. Nombres de los métodos a partir de los diagramas de interacción.

En general, el conjunto de todos los mensajes enviados a una clase X a lo largo de todos los diagramas de interacción indican la mayoría de los métodos que debe definir la clase X.

La inspección de todos los diagramas de interacción de la aplicación del PDV da lugar a la asignación de métodos que se muestra en la Figura 19.5.

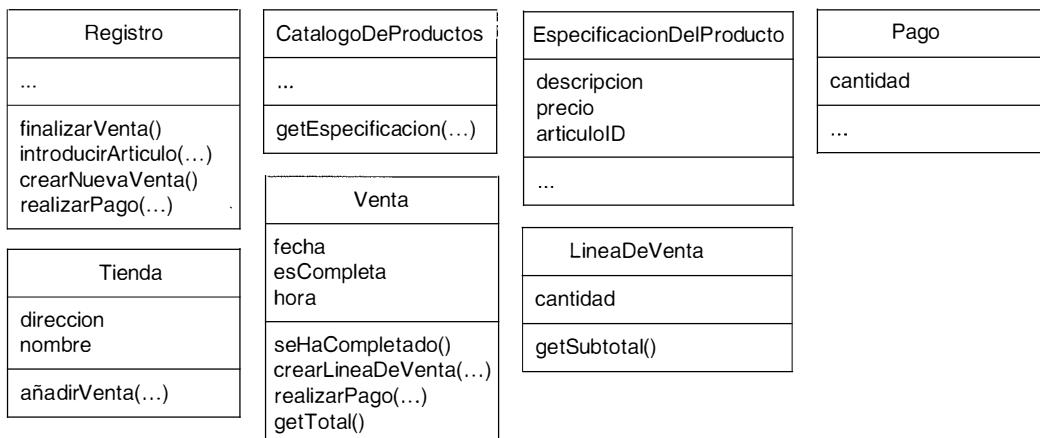


Figura 19.5. Métodos de la aplicación.

Cuestiones acerca de los nombres de los métodos

Se deben tener en cuenta las siguientes cuestiones especiales en relación con los nombres de los métodos:

- Interpretación del mensaje *create*.
- Descripción de los métodos de acceso.
- Interpretación de los mensajes a los multiobjetos.
- Sintaxis dependiente del lenguaje.

Nombres de los métodos—*create*

El mensaje *create* es una forma independiente del lenguaje posible en UML para indicar instanciación e inicialización. Al traducir el diseño a un lenguaje de programación orientado a objetos, se debe expresar en función de sus estilos para la instanciación e inicialización. No existe un método *create* ni en C++, ni en Java, ni en Smalltalk. Por ejemplo, en C++, implica la asignación automática, o asignación de almacenamiento libre con el operador *new*, seguido de una llamada al constructor. En Java, implica la invocación del operador *new*, seguido de una llamada al constructor.

Debido a las múltiples interpretaciones, y también debido a que la inicialización es una actividad muy común, es habitual omitir en el DCD los métodos relacionados con la creación y los constructores.

Nombres de los métodos—métodos de acceso

Los **métodos de acceso** recuperan (método de obtención, *accessor*) o establecen (método de cambio, *mutador*) el valor de los atributos. En algunos lenguajes (como Java) es un estilo común tener un accessor y un mutator para cada atributo, y declarar todos los

atributos como privados (para imponer la encapsulación de datos). Normalmente, se excluye la descripción de estos métodos en el diagrama de clases debido a que generan mucho ruido; para n atributos, hay $2n$ métodos sin interés. Por ejemplo, no se muestra, aunque está presente, el método *getPrecio* (o *precio*) de la *EspecificacionDelProducto*, porque se trata simplemente de un método de acceso.

Nombres de los métodos—multiobjetos

Un mensaje a un multiobjeto se interpreta como un mensaje al propio objeto contenedor/colección. Por ejemplo, el siguiente mensaje al multiobjeto *buscar* se debe interpretar como un mensaje al objeto contenedor/colección, como a un *Map* en Java, un *map* en C++, o un *Dictionary* en Smalltalk (ver Figura 19.6).

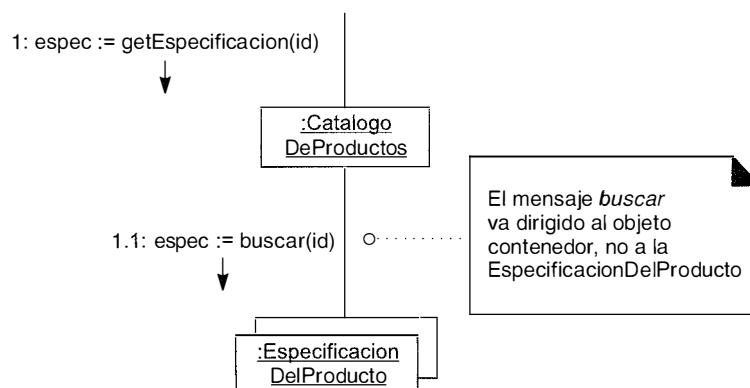


Figura 19.6. Mensaje a un multiobjeto.

Por tanto, el método *buscar* no forma parte de la clase *EspecificacionDelProducto* sino de la interfaz del multiobjeto. En consecuencia, no es correcto añadir el método *buscar* a la clase *EspecificacionDelProducto*.

Normalmente, estas interfaces o clases de contenedores/colecciones (como la interfaz *java.util.Map*) son elementos de las librerías predefinidas, y no es útil mostrar explícitamente estas clases en el DCD, porque añaden ruido, pero poca información nueva.

Nombres de los métodos—sintaxis dependiente del lenguaje

Algunos lenguajes, como Smalltalk, tienen una sintaxis que es muy diferente del formato UML básico de *nombreMetodo(listaParametros)*. Se recomienda que se utilice el formato UML básico, incluso si el lenguaje de implementación que se planea utilizar tiene una sintaxis diferente. Idealmente, la traducción debería tener lugar en el momento de la generación de código, en lugar de durante la creación de los diagramas de clase. Sin embargo, UML permite otra sintaxis para la especificación de los métodos.

Añadir más información sobre los tipos

Opcionalmente, todos los tipos de los atributos, parámetros de los métodos y los valores de retorno se podrían mostrar. La cuestión sobre si se muestra o no esta información se debe considerar en el siguiente contexto:

Se debería crear un DCD teniendo en cuenta los destinatarios.

- Si se está creando en una herramienta CASE con generación automática de código, son necesarios todos los detalles y de modo exhaustivo.
- Si se está creando para que lo lean los desarrolladores de software, los detalles exhaustivos de bajo nivel podrían afectar negativamente por el nivel de ruido.

Por ejemplo, ¿es necesario mostrar todos los parámetros y la información de sus tipos? Depende de lo obvia que sea la información para la audiencia a la que está destinada.

El diagrama de clases de diseño de la Figura 19.7 muestra más información sobre los tipos.

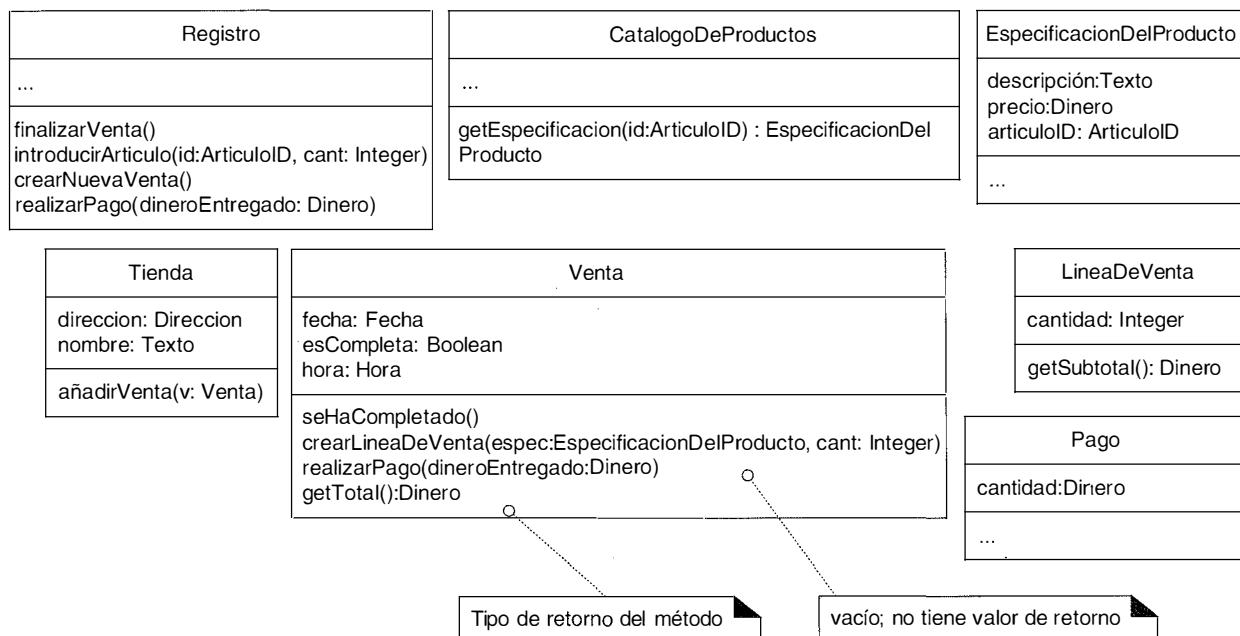


Figura 19.7. Añadir información acerca de los tipos.

Añadir asociaciones y navegabilidad

Cada extremo de asociación se denomina rol, y en los DCD el rol podría decorarse con una flecha de navegabilidad. La **navegabilidad** es una propiedad del rol que indica que es posible navegar unidireccionalmente a través de la asociación desde los objetos de la clase origen a la clase destino. La navegabilidad implica visibilidad —normalmente visibilidad de atributo (ver Figura 19.8)—.

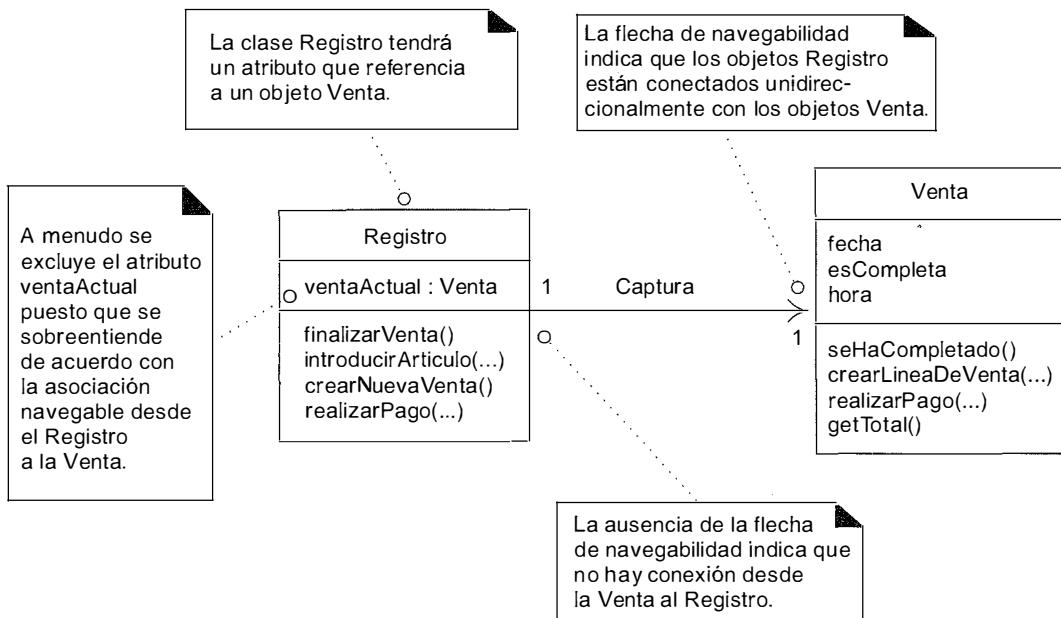


Figura 19.8. Representación de la navegabilidad o visibilidad de atributo.

La interpretación habitual de una asociación con una flecha de visibilidad es la visibilidad de atributo desde la clase origen hasta la clase destino. Durante la implementación en un lenguaje orientado a objetos por lo general se transforma en un atributo en la clase origen que hace referencia a una instancia de la clase destino. Por ejemplo, la clase *Registro* definirá un atributo que referencia a una instancia de *Venta*.

La mayoría, por no decir todas, las asociaciones en los DCD deberían adornarse con las flechas de navegabilidad necesarias.

En un DCD, se eligen las asociaciones de acuerdo a un criterio necesito-conocer orientado al software estricto —¿qué asociaciones se requieren para satisfacer la visibilidad y las necesidades de memoria actuales que se indican en los diagramas de interacción?—. Esto contrasta con las asociaciones en el Modelo del Dominio, que se podrían justificar por la intención de mejorar la comprensión del dominio del problema. Una vez más, vemos que existe una diferencia entre los objetivos del Modelo del Diseño y del Modelo del Dominio: uno es analítico, el otro una descripción de los componentes software.

La visibilidad y las asociaciones requeridas entre las clases se dan a conocer mediante los diagramas de interacción. A continuación, presentamos algunas situaciones comunes que sugieren la necesidad de definir una asociación con un adorno de visibilidad de A a B:

- A envía un mensaje a B.
- A crea una instancia de B.
- A necesita mantener una conexión a B.

Por ejemplo, a partir del diagrama de interacción de la Figura 19.9 que comienza con el mensaje *create* a la *Tienda*, y a partir del contexto más amplio de los otros diagramas

de interacción, se puede distinguir que probablemente la *Tienda* debería tener una conexión permanente con las instancias de *Registro* y el *CatalogoDeProductos* que crea. También es razonable que el *CatalogoDeProductos* necesite una conexión permanente con la colección de objetos *EspecificacionDelProducto* que crea. De hecho, muy a menudo el creador de otro objeto requiere una conexión permanente con él. Por tanto, las conexiones implícitas se presentarán como asociaciones en el diagrama de clases.

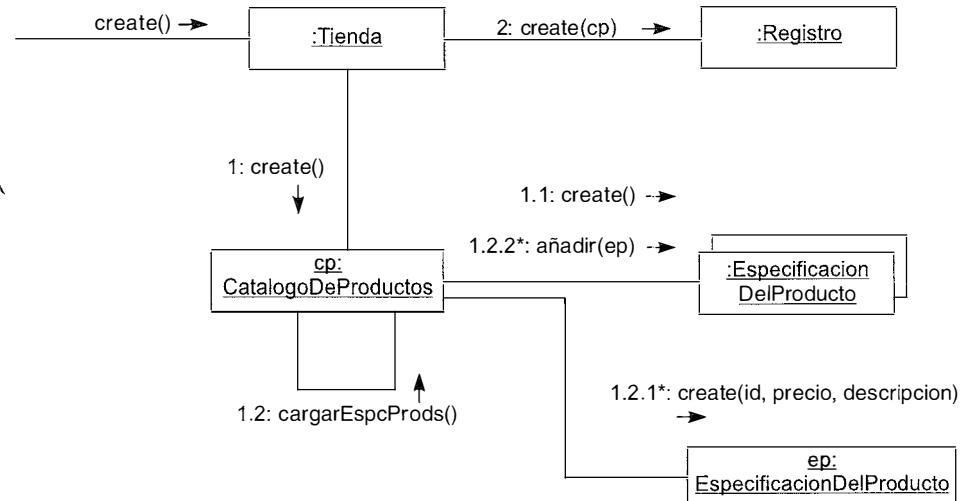


Figura 19.9. La navegabilidad se identifica a partir de los diagramas de interacción.

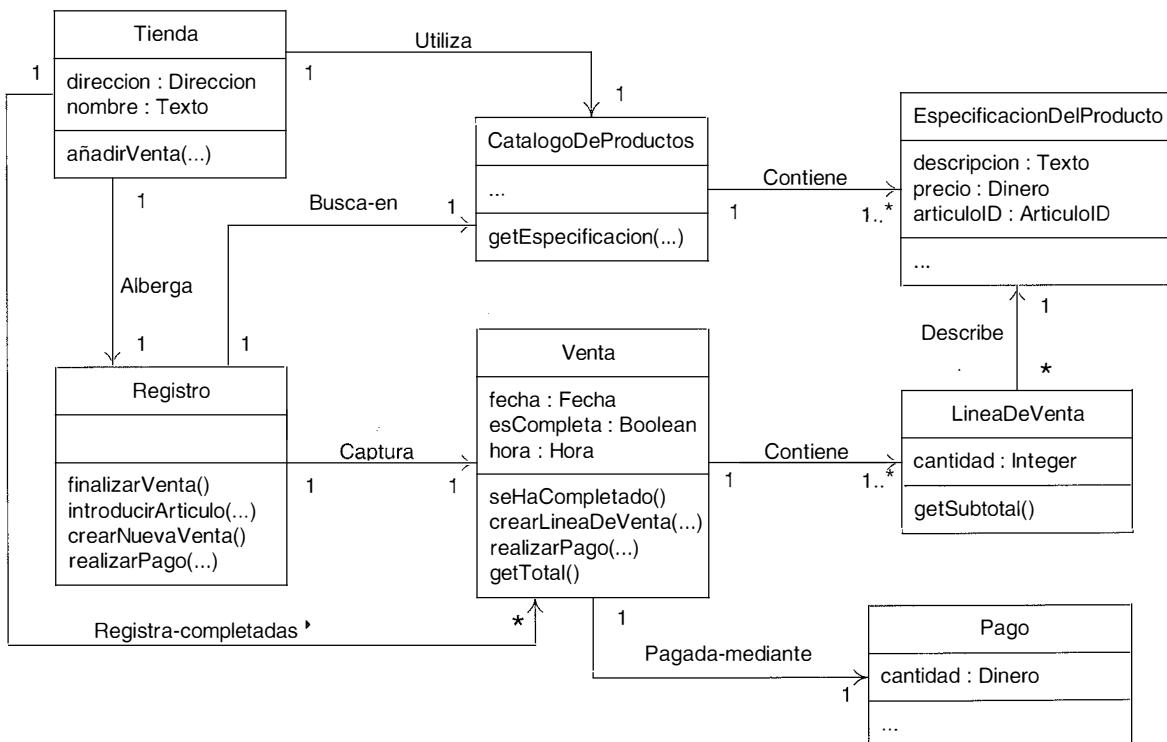


Figura 19.10. Asociaciones con adornos de navegabilidad.

Basado en el criterio anterior para las asociaciones y la navegabilidad, el análisis de todos los diagramas de interacción generados para la aplicación del PDV NuevaEra dará lugar a un diagrama de clases (ver Figura 19.10) con las siguientes asociaciones (se oculta la información exhaustiva sobre los tipos por claridad).

Nótese que éste no es exactamente el mismo conjunto de asociaciones que se generó para el diagrama de clases del Modelo del Dominio. Por ejemplo, en el modelo del dominio no existía la asociación *Busca-en* entre el *Registro* y el *CatalogoDeProductos* —no se pensó que fuera una relación importante y permanente en ese momento—. Pero durante la creación de los diagramas de interacción, se decidió que el objeto software *Registro* debería tener una conexión permanente con el *CatalogoDeProductos* software para buscar los objetos *EspecificacionDelProducto*.

Añadir las relaciones de dependencia

UML incluye una **relación de dependencia** general, que indica que un elemento (de cualquier tipo, como clases, casos de uso, etc.) tiene conocimiento de otro elemento. Se representa mediante una línea de flecha punteada. En los diagramas de clases la relación de dependencia es útil para describir la visibilidad entre clases que no es de atributo; en otras palabras, declaración de visibilidad de parámetro, local o global. En cambio, la vi-

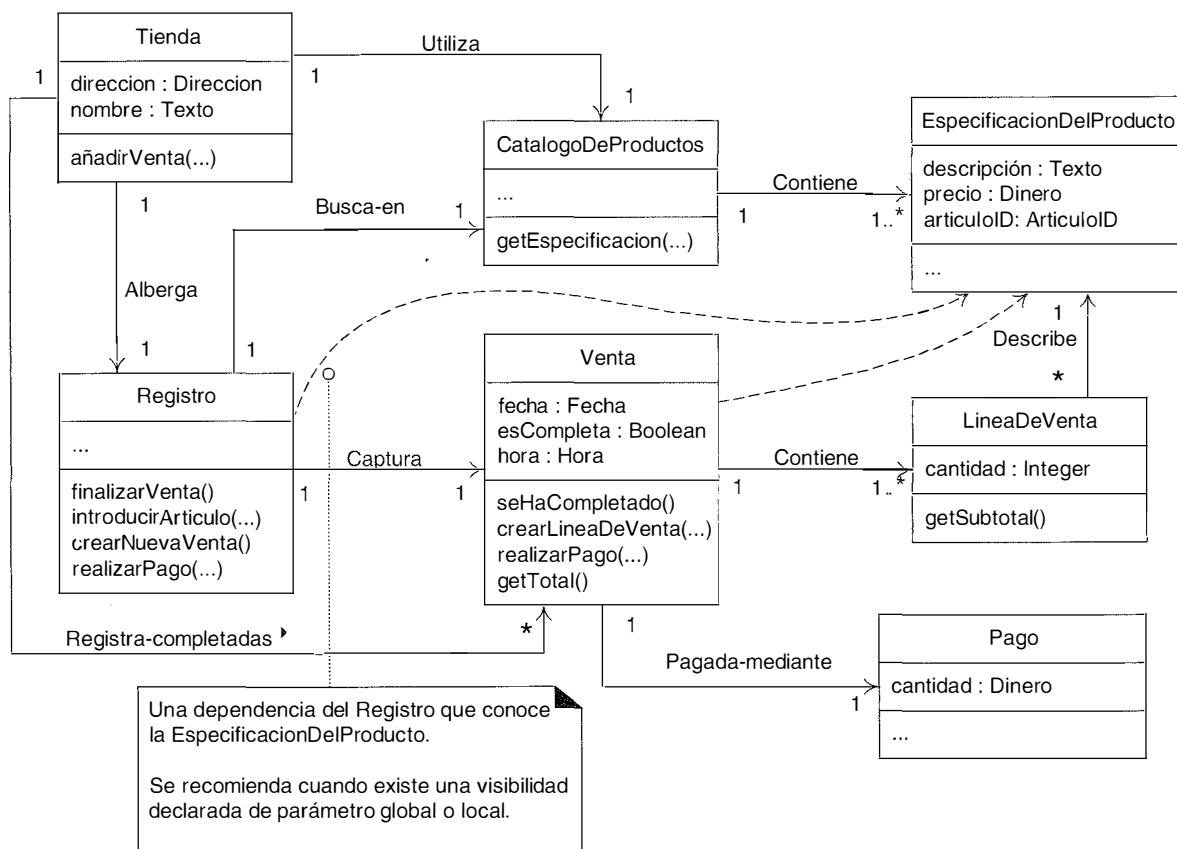


Figura 19.11. Relaciones de dependencia que indican una visibilidad que no es de atributo.

sibilidad de atributo simple se muestra mediante una línea de asociación ordinaria y una flecha de navegabilidad. Por ejemplo, el objeto software *Registro* recibe un objeto de retorno de tipo *EspecificacionDelProducto* a partir del mensaje de especificación que envía al *CatalogoDeProductos*. Por tanto, el *Registro* tiene una visibilidad a corto plazo, declarada localmente, de la *EspecificacionDelProducto*. Y una *Venta* recibe una *EspecificacionDelProducto* como parámetro en el mensaje *crearLineaDeVenta*; tiene visibilidad de parámetro de una especificación.

Estas visibilidades que no son de atributo podrían representarse con la línea de flecha punteada que indica una relación de dependencia (ver Figura 19.11). La curvatura de las líneas de dependencias no tiene ningún significado; se acomoda a la representación gráfica.

19.6. Notación para los detalles de los miembros

UML proporciona una notación variada para describir las características de los miembros de las clases e interfaces, como la visibilidad, valores iniciales, etcétera. En la Figura 19.12 se muestra un ejemplo.

ClaseEjemplo	java.awt.Font	«interface» Runnable
atributoDeClase + atributoPublico - atributoPrivado atributoConVisibilidadSinEspecificiar atributo1 : tipo hamburguesas : Lista de HamburguesasVegetales atributo2 : tipo = valor inicial atributoConstanteFinal : int = 5 { frozen } /atributoDerivado	plain : Integer = 0 { frozen } bold : Integer = 1 { frozen } name : String style : Integer = 0 ... + getFont(name : String) : Font + getName() : String ...	run()
metodoDeClase() + «constructor» ClaseEjemplo(int) metodoConVisibilidadSinEspecificiar() metodoDevuelveAlgo() : Foo metodoAbstracto() metodoAbstracto2() { abstract } // alternativo + metodoPublico() - metodoPrivado() # metodoProtegido() ~ metodoConVisibilidadDePaquete() metodoFinal() { leaf } metodoSinEfectosLaterales() { query } metodoSincronizado() { guarded } metodo1ConParametros(in parm1:String, inout parm2:int) metodo2ConParametros(parm1:String, parm2:float) metodo3ConParametros(parm1, parm2) metodo4ConParametros(String, int) metodoConParametrosYValorRetorno(parm1: String) : Foo metodoConParametrosSinEspecificiar(...) : Foo metodoConParametrosYValorRetornoAmbosSinEspecificiar()	java.awt.Toolkit o java.awt.Toolkit { abstract } ... // hay atributos, pero no se muestran. # createButton(target : Button) : ButtonPeer + getColorModel() : ColorModel ...	RelojDeAlarma run() ...
	FinalClass { leaf }	un compartimiento vacío sin los tres puntos significa que sin ninguna duda no existe ningún miembro (en este caso, no hay atributos)
	... // hay métodos, pero no se muestran	

Figura 19.12. Detalles de la notación UML para algunos miembros de los diagramas de clases.

Visibilidad por defecto en UML

Si no se muestra explícitamente ningún marcador de visibilidad para un atributo o método, ¿cuál es el valor por defecto? Respuesta: no hay un valor por defecto. Si no se muestra nada, en UML significa “sin especificar”. Sin embargo, la convención común es asumir que los atributos son privados y los métodos públicos, a menos que se indique otra cosa.

La iteración actual del diagrama de clases de diseño del PDV NuevaEra (ver Figura 19.13) no tiene muchos detalles interesantes de los miembros; todos los atributos son privados y todos los métodos públicos.

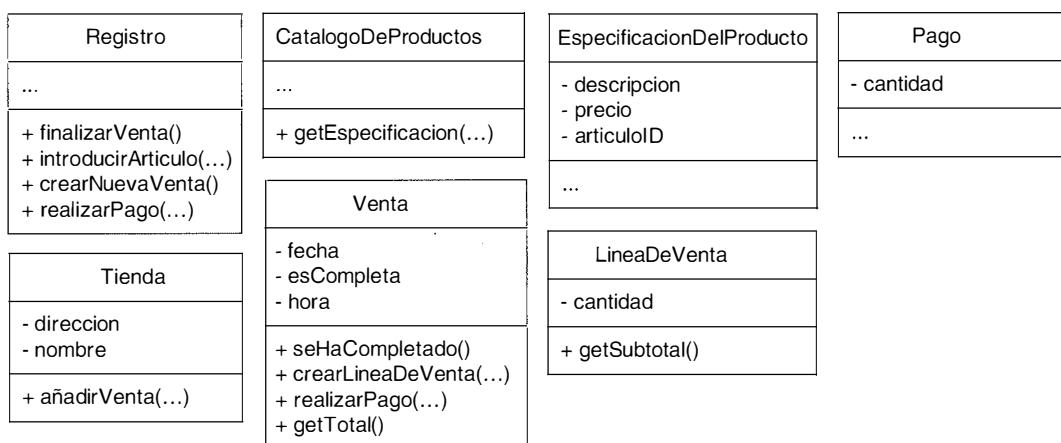


Figura 19.13. Detalles de los miembros del diagrama de clases del PDV.

Notación para el cuerpo de los métodos en los DCD (y los diagramas de interacción)

El cuerpo de un método se puede representar como se ilustra en la Figura 19.14 tanto en el DCD como en el diagrama de interacción.

19.7. DCD, dibujo y herramientas CASE

Las herramientas CASE pueden hacer ingeniería inversa (generar) de los DCD a partir del código fuente. En el Capítulo 35 sobre el dibujo y las herramientas CASE, se presentará una breve discusión sobre el contexto del proceso y la práctica de dibujar los DCD.

19.8. DCD en el UP

Los DCD forman parte de la realización de los casos de uso y, por tanto, miembros del Modelo de Diseño del UP.

Fases

Inicio: El Modelo de Diseño y los DCD normalmente no comenzarán hasta la elaboración porque comprenden decisiones de diseño, que son prematuras durante la fase de inicio.

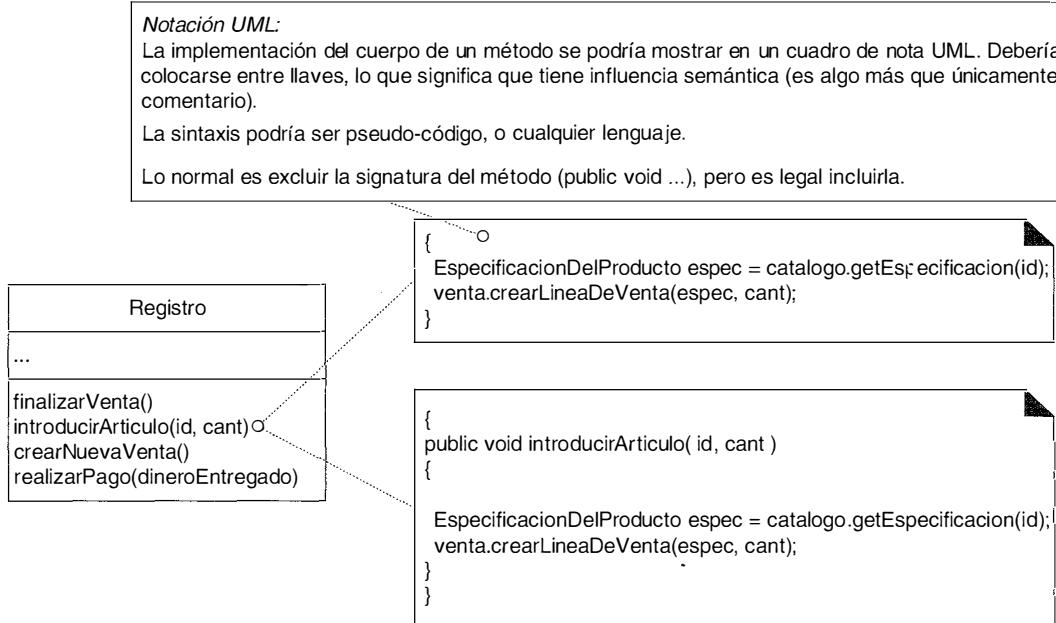


Figura 19.14. Notación para el cuerpo de los métodos.

Tabla 19.1. Muestra de los artefactos UP y evolución temporal. c-comenzar; r-refinar

Disciplina	Artefacto Iteración →	Inicio I1	Elab. E1...En	Const. C1...Cn	Trans. T1...T2
Modelado del Negocio	Modelo del Dominio		c		
Requisitos	Modelo de Casos de Uso (DSS) Visión Especificación Complementaria Glosario	c c c c	r r r r		
Diseño	Modelo de Diseño Documento de Arquitectura SW Modelo de Datos		c c c	r r	
Implementación	Modelo de Implementación		c	r	r
Gestión del Proyecto	Plan de Desarrollo SW	c	r	r	r
Pruebas	Modelo de Pruebas		c	r	
Entorno	Marco de Desarrollo	c	r		

Elaboración: Durante esta fase, los DCD acompañarán a los diagramas de interacción de las realizaciones de los casos de uso; podrían crearse para las clases del diseño más significativas desde el punto de vista de la arquitectura.

Nótese que las herramientas CASE pueden hacer ingeniería inversa (generar) de los DCD a partir del código fuente. Es recomendable generar los DCD de manera regular a partir del código fuente, para visualizar la estructura estática del sistema.

Construcción: Los DCD se continuarán generando a partir del código fuente como apoyo a la visualización de la estructura estática del sistema.

19.9. Artefactos del UP

La Figura 19.15 muestra la influencia entre los artefactos destacando los DCD.

Muestra de las relaciones entre los artefactos del UP para Diagramas de Clases de Diseño

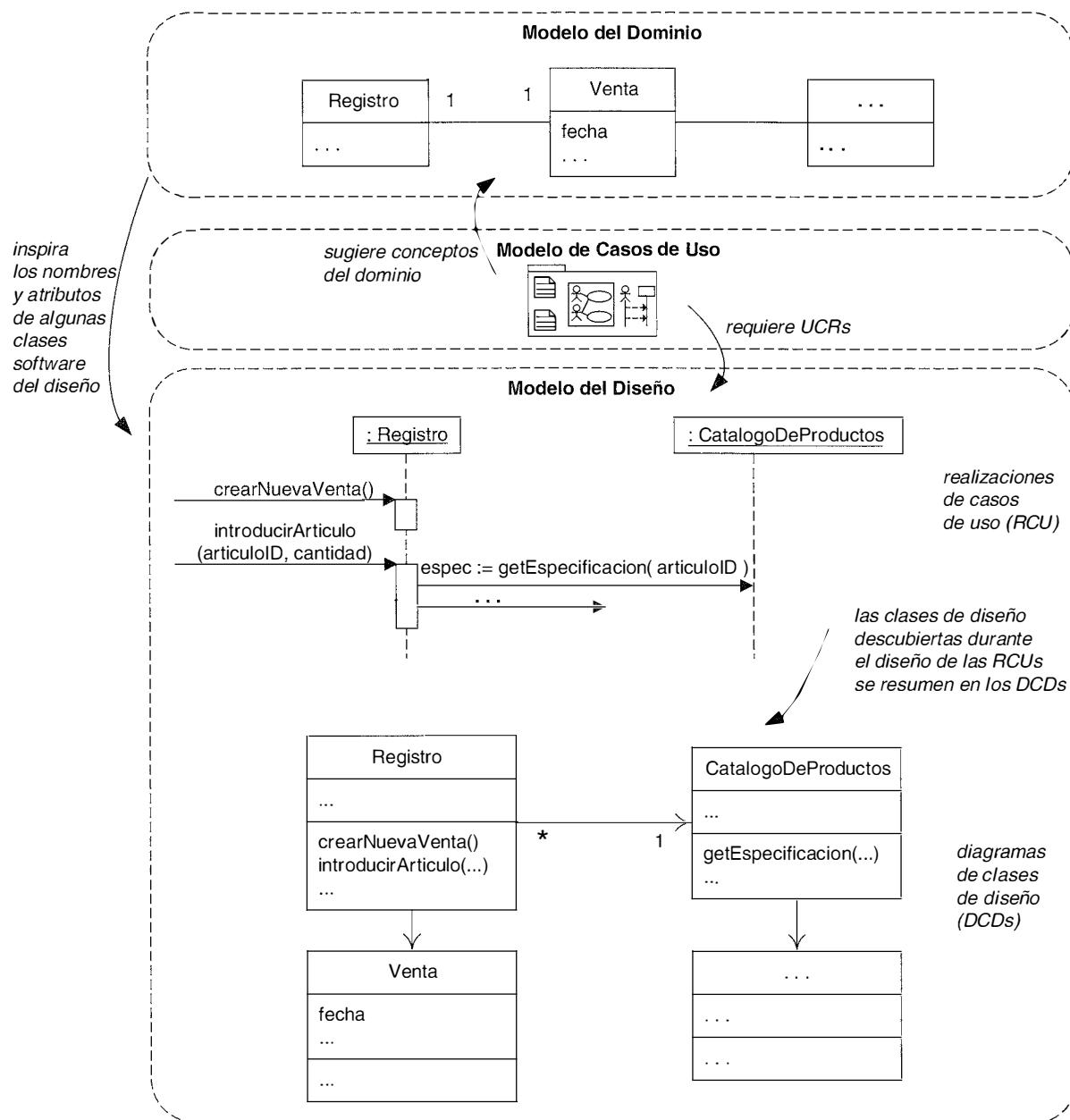


Figura 19.15. Muestra de la influencia entre los artefactos del UP.

Capítulo 20

MODELO DE IMPLEMENTACIÓN: TRANSFORMACIÓN DE LOS DISEÑOS EN CÓDIGO

Esté atento a los errores del programa anterior; sólo he demostrado que es correcto, pero nunca lo he usado.

Donald Knuth

Objetivos

- Transformar los artefactos del diseño en código en un lenguaje orientado a objetos.
-

Introducción

Después de completar los diagramas de interacción y los DCDs para la iteración actual de la aplicación NuevaEra, disponemos de suficientes detalles para generar el código de la capa del dominio de los objetos.

Los artefactos UML creados durante el trabajo de diseño —los diagramas de interacción y los DCDs— se utilizarán como entradas en el proceso de generación de código.

El UP define el Modelo de Implementación. Éste contiene los artefactos de implementación como el código fuente, las definiciones de bases de datos, las páginas JSP/XML/HTML, etcétera. Por tanto, el código que se va a crear en este capítulo forma parte del Modelo de Implementación.

Elección del lenguaje

Se utiliza Java para los ejemplos debido a su uso extendido y familiaridad. Sin embargo, la intención no es dar a entender que se recomienda Java en especial; C#, Visual Basic, C++, Smalltalk, Python y muchos otros lenguajes son susceptibles de aplicar los prin-

cipios de diseño de objetos y la transformación a código que se presenta en este caso de estudio.

20.1. Programación y el proceso de desarrollo

El trabajo de diseño anterior no debería implicar que no exista prototipado o diseño durante la programación; las herramientas de desarrollo modernas proporcionan un entorno excelente para explorar rápidamente enfoques alternativos, y normalmente merece la pena elaborar algo (o incluso mucho) de diseño mientras se programa.

Sin embargo, algunos desarrolladores encuentran útil anticipar un poco mediante el modelado visual antes de programar, especialmente aquellos que se encuentran cómodos con el razonamiento visual o los lenguajes basados en diagramas.

Sugerencia

Para una iteración de dos semanas, considere dedicar por lo menos medio día próximo al comienzo de la iteración para llevar a cabo algo de trabajo de diseño mediante el modelado visual, antes de pasar a la programación. Utilice “herramientas” sencillas que soporten la realización rápida y creativa de diagramas, como una pizarra y una cámara digital. Si encuentra una herramienta para ingeniería del software asistida por ordenador (CASE, *computer-aid software engineering*) para UML que sea igualmente rápida, sencilla, y adecuada, excelente.

La creación de código en un lenguaje de programación orientado a objetos —como Java o C#— no forma parte del A/DOO; es un objetivo final. Los artefactos creados en el Modelo de Diseño del UP proporcionan parte de la información necesaria para generar el código.

Una ventaja del A/DOO y de la programación OO —cuando se utiliza con el UP— es que proporciona una guía de principio a fin desde los requisitos hasta el código. Los distintos artefactos suministran información a los artefactos posteriores de manera útil y siguiendo una traza, culminando finalmente en una aplicación en ejecución. Esto no sugiere que el camino será fácil, o que se pueda seguir de manera mecánica —existen demasiadas variables—. Pero contar con una guía proporciona un punto de partida para experimentar y discutir.

Creatividad y cambio durante la implementación

Durante el trabajo de diseño se tomaron algunas decisiones y se llevó a cabo un trabajo creativo. Veremos a lo largo de la siguiente presentación que la generación de código —en este ejemplo— es un proceso de traducción relativamente mecánico.

Sin embargo, en general, el trabajo de programación no es una etapa de generación de código trivial, más bien lo contrario. En realidad, los resultados generados durante el diseño son un primer paso incompleto; durante la programación y pruebas, se realizarán innumerables cambios y se descubrirán y resolverán problemas complicados.

Si se hace bien, los artefactos del diseño proporcionarán un núcleo elástico que podrá extenderse con elegancia y robustez para satisfacer los nuevos problemas encon-

trados durante la programación. En consecuencia, espere y planifique para cambios y desviaciones del diseño durante la programación.

Cambios del código y el proceso iterativo

Una ventaja del proceso de desarrollo iterativo e incremental es que los resultados de la iteración anterior pueden proporcionar los datos para el comienzo de la siguiente iteración (ver Figura 20.1). Por tanto, los resultados del análisis y diseño actual se están refinando y enriqueciendo continuamente a partir del trabajo de implementación anterior. Por ejemplo, cuando el código de la iteración N se desvía del diseño de la iteración N (que lo hará inevitablemente), el diseño final que es la base de la implementación puede ser la entrada para los modelos de análisis y diseño de la iteración N+1.

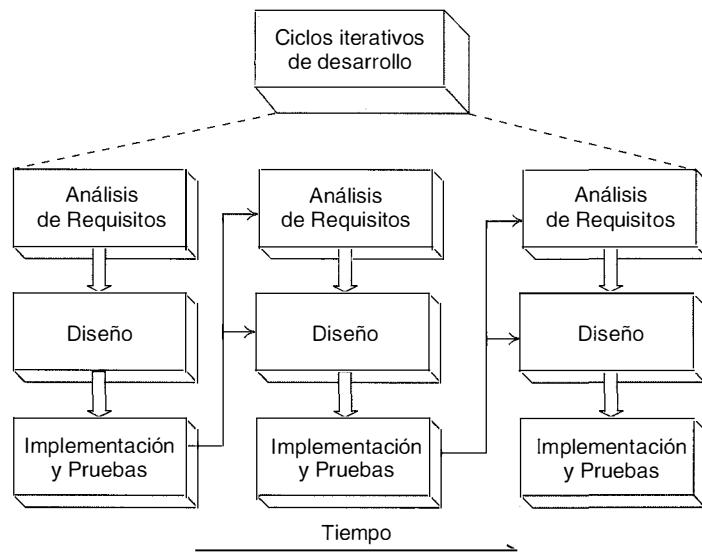


Figura 20.1. La implementación en una iteración influye en el diseño posterior.

Una de las primeras actividades en una iteración es sincronizar los diagramas de diseño; los primeros diagramas de la iteración N no corresponderán con el código final de esa misma iteración, y necesitan sincronizarse antes de que se extiendan con nuevos resultados del diseño.

Cambios en el código, herramientas CASE, e ingeniería inversa

Es deseable que los diagramas generados durante el diseño se actualicen de manera semi-automática para reflejar los cambios en el trabajo de codificación siguiente. Generalmente esto debería hacerse con una herramienta CASE que puede leer el código fuente y generar automáticamente, por ejemplo, diagramas de paquetes, clases y secuencia. Este es un aspecto de la **ingeniería inversa** —la actividad de generar diagramas a partir del código fuente (o a veces, ejecutable)—.

20.2. Transformación de los diseños en código

La implementación en un lenguaje orientado a objetos requiere la escritura de código fuente para:

- Las definiciones de las clases e interfaces.
- Las definiciones de los métodos.

Las secciones siguientes presentarán la generación en Java (como un caso típico).

20.3. Creación de las definiciones de las clases a partir de los DCDs

Como mínimo, los DCDs describen los nombres de las clases o interfaces, las superclases, signatura de los métodos y los atributos simples de una clase. Esto es suficiente para crear una definición de clase básica en un lenguaje de programación orientado a objetos. Una discusión posterior explorará la incorporación de la información acerca de las interfaces y ámbito de nombres (o paquetes), entre otros detalles.

Definición de una clase con métodos y atributos simples

A partir del DCD, la transformación de las definiciones de los atributos básicas (campos de instancia simples de Java) y las signaturas de los métodos de la *LíneaDeVenta* a la definición Java es directa, como se muestra en la Figura 20.2.

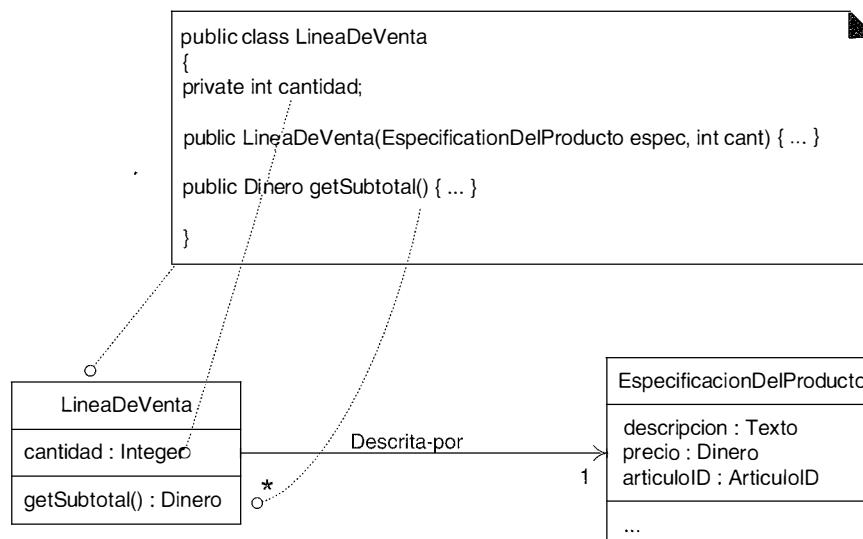


Figura 20.2. LineaDeVenta en Java.

Obsérvese que se ha incorporado en el código fuente el constructor Java *LineaDeVenta(...)*. Se deriva del envío del mensaje *create(espec, ctd)* a la *LineaDeVenta* en el diagrama de interacción *introducirArticulo*. Esto indica, en Java, que se requiere un constructor que soporte estos parámetros. A menudo se excluye de los diagramas de clases el método *create* puesto que es común que aparezca y tiene múltiples interpretaciones, dependiendo del lenguaje que se vaya a utilizar.

Añadir atributos de referencia

Un **atributo de referencia** es un atributo que referencia a otro objeto complejo, no a un tipo primitivo como una cadena de texto, un número, etcétera.

Los atributos de referencia de una clase se deducen de las asociaciones y la navegabilidad de un diagrama de clases.

Por ejemplo, una *LíneaDeVenta* tiene una asociación con una *EspecificacionDelProducto*, con navegabilidad hacia ella. Es habitual que esto se interprete como un atributo de referencia en la clase *LíneaDeVenta* que hace referencia a la instancia de *EspecificacionDelProducto* (ver Figura 20.3).

En Java, esto significa que se recomienda un campo de instancia que referencia a una instancia de *EspecificacionDelProducto*.

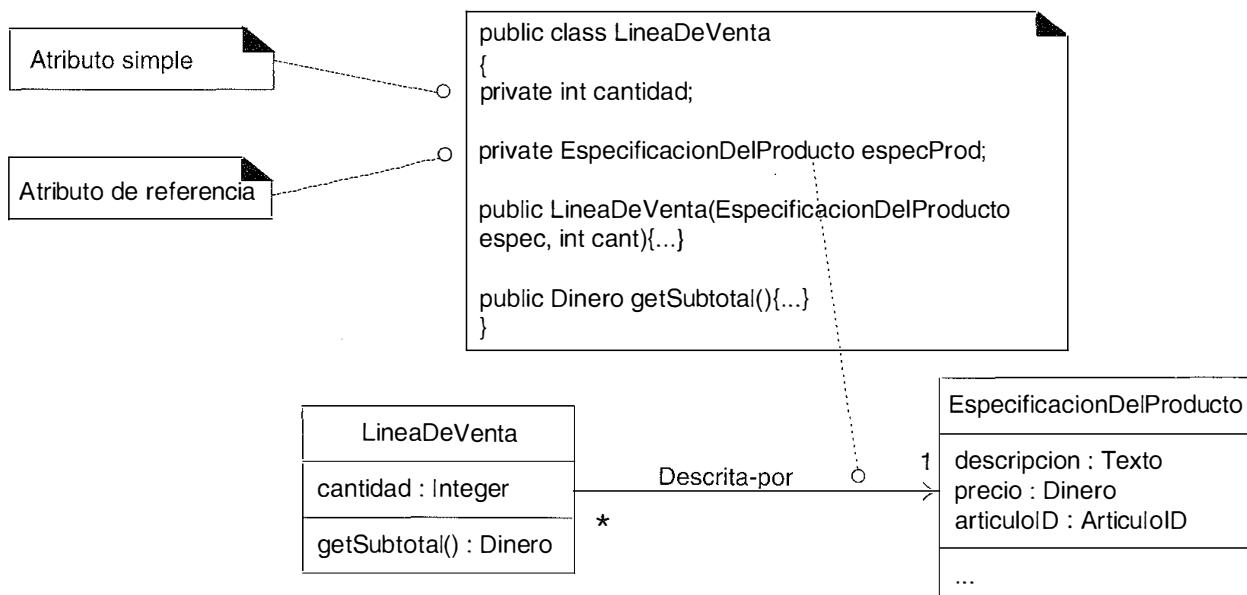


Figura 20.3. Inclusión de atributos de referencia.

Obsérvese que los atributos de referencia de una clase a menudo están implícitos, en lugar de explícitos, en un DCD.

Por ejemplo, aunque hemos añadido un campo de instancia en la definición Java de *LíneaDeVenta* para que apunte a una *EspecificacionDelProducto*, no se declara explícitamente como un atributo en la sección de atributos del rectángulo de la clase. Existe una visibilidad de atributo *recomendada* —indicada por la asociación y navegabilidad— que se define explícitamente mediante un atributo durante la fase de generación de código.

Atributos de referencia y los nombres de los roles

La siguiente iteración estudiará el concepto de los nombres de los roles en los diagramas de estructura estáticos. Cada extremo de asociación se denomina rol. Brevemente, un **nombre de rol** es un nombre que identifica al rol y, a menudo, proporciona algo del contexto semántico acerca de la naturaleza del rol.

Si el nombre de un rol está presente en un diagrama de clases, utilícelo como base para el nombre del atributo de referencia durante la generación de código, como se muestra en la Figura 20.4.

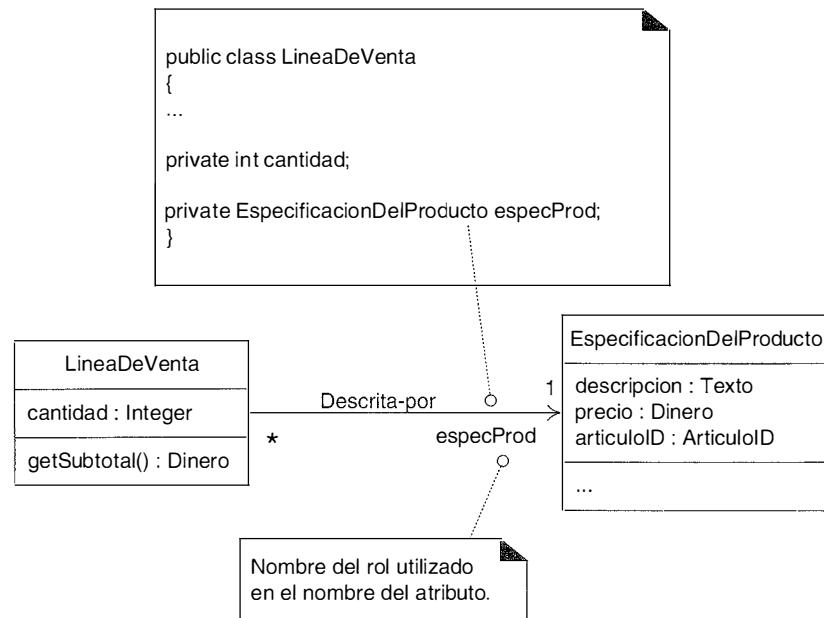


Figura 20.4. Los nombres de los roles podrían utilizarse para generar los nombres de las variables de instancia.

Transformación de los atributos

La clase `Venta` ilustra que en algunos casos uno debe considerar la transformación de los atributos desde el diseño al código en lenguajes diferentes. La Figura 20.5 muestra el problema y su solución.

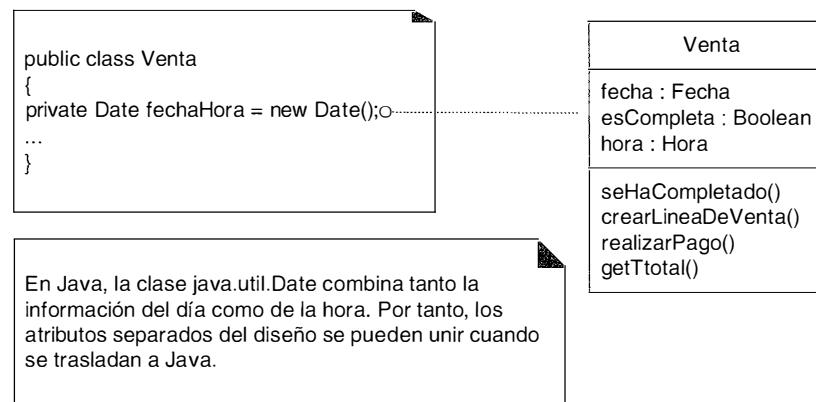


Figura 20.5. Transformación de la fecha y la hora en Java.

20.4. Creación de métodos a partir de los diagramas de interacción

Un diagrama de interacción muestra los mensajes que se envían como respuesta a la invocación de un método. La secuencia de estos mensajes se traduce en una serie de sentencias en la definición del método. El diagrama de interacción *introducirArticulo* de la Figura 20.6 ilustra la definición Java del método *introducirArticulo*.

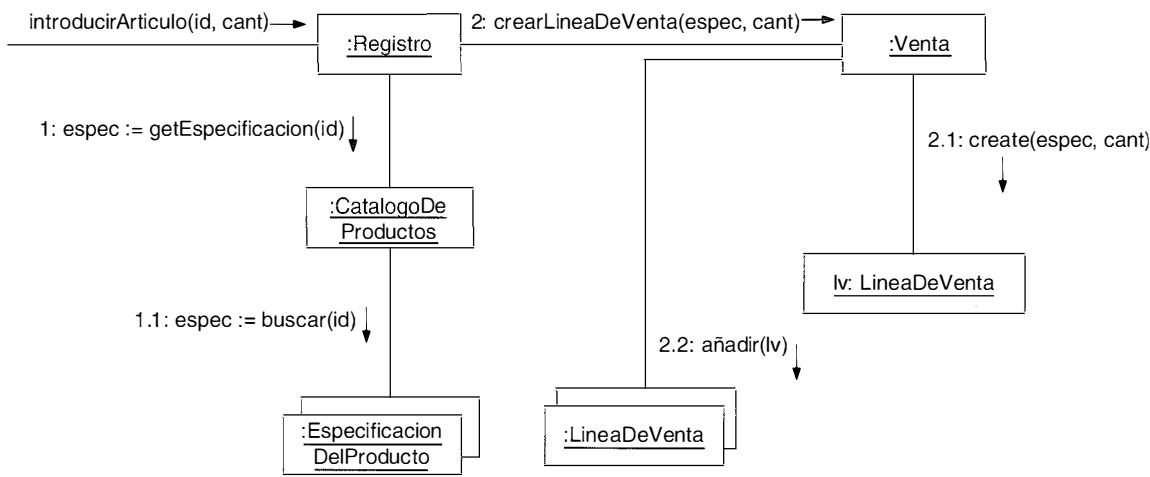


Figura 20.6. El diagrama de interacción de introducirArticulo.

En este ejemplo, se utilizará la clase *Registro*. La Figura 20.7 muestra una definición en Java.

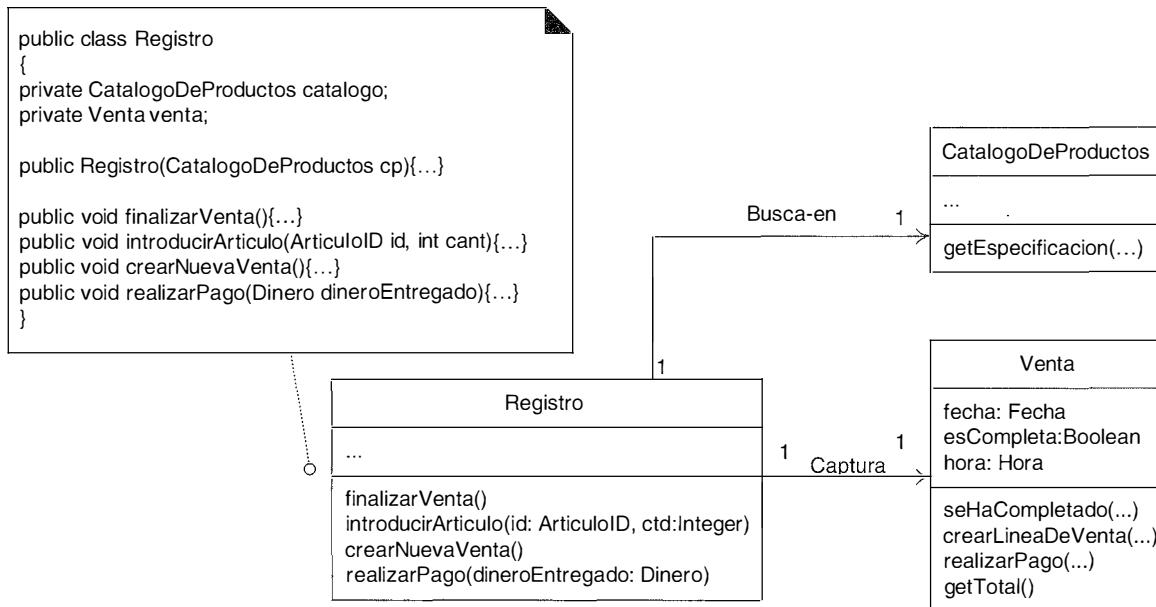


Figura 20.7. La clase Registro.

El método Registro--introducirArticulo

Se envía el mensaje *introducirArticulo* a una instancia de *Registro*; por tanto, se define el método *introducirArticulo* en la clase *Registro*.

```
public void introducirArticulo(ArticuloID articuloID, int ctd)
```

Mensaje 1: Se envía el mensaje *getEspecificacion* al *CatalogoDeProductos* para recuperar una *EspecificacionDelProducto*.

```
EspecificacionDelProducto espec =
catalogo.getEspecificacion(articuloID);
```

Mensaje 2: Se envía el mensaje *crearNuevaVenta* a la *Venta*.

```
venta.crearLineaDeVenta(espec, ctd);
```

Resumiendo, cada mensaje de la secuencia del método, que se muestra en el diagrama de interacción, se transforma en una sentencia del método Java.

La Figura 20.8 muestra el método completo *introducirArticulo* y su relación con el diagrama de interacción.

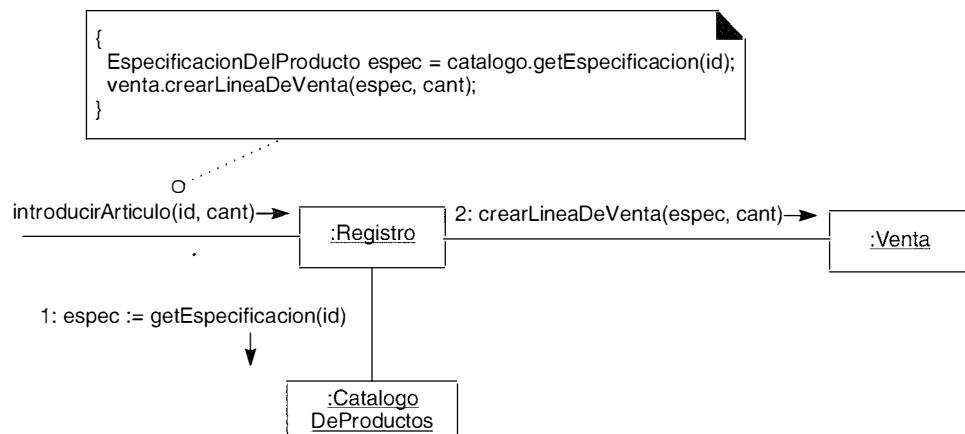


Figura 20.8. El método introducirArticulo.

20.5. Clases contenedoras/colecciones en el código

A menudo es necesario que un objeto mantenga la visibilidad a un grupo de otros objetos; normalmente esta necesidad es evidente a partir del valor de la multiplicidad en el diagrama de clases —podría ser mayor que uno—. Por ejemplo, una *Venta* debe mantener la visibilidad a un grupo de instancias de *LineaDeVenta*, como se muestra en la Figura 20.9.

En los lenguajes de programación OO, con frecuencia se implementan estas relaciones introduciendo un contenedor o colección intermedia. La clase del lado del uno de-

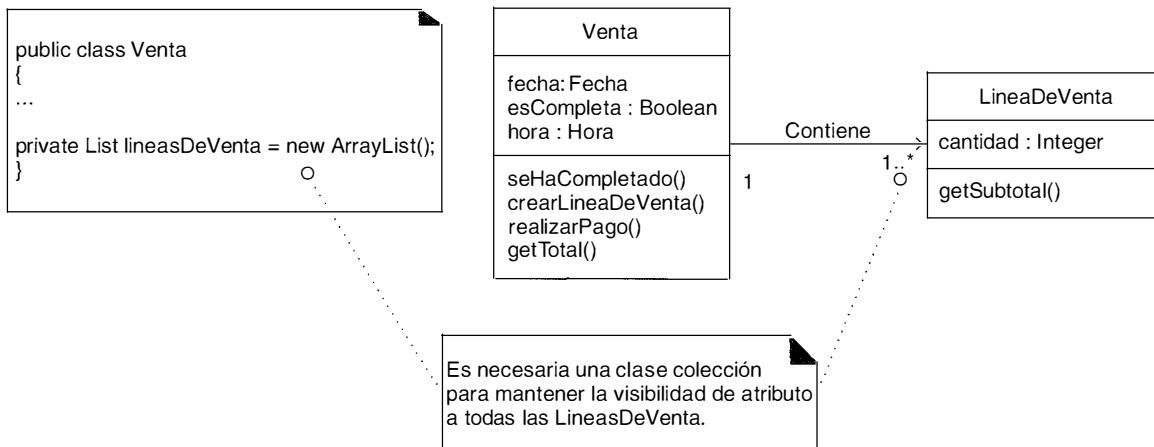


Figura 20.9. Incorporación de una colección.

fine un atributo de referencia que apunta a la instancia del contenedor/colección, que contiene instancias de la clase del lado de muchos.

Por ejemplo, las librerías de Java contienen las clases de colección como *ArrayList* y *HashMap*, que implementan las interfaces de *List* y *Map*, respectivamente. Utilizando *ArrayList*, la clase *Venta* puede definir un atributo que mantenga una lista ordenada de las instancias de *LineaDeVenta*.

Por supuesto, los requisitos influyen en la elección de la clase colección; las búsquedas por claves requieren el uso de un *Map*, una lista ordenada creciente requiere una *List*, etcétera.

20.6. Manejo de excepciones y de errores

Hasta el momento se ha ignorado el manejo de las excepciones en el desarrollo de una solución. Esto se hizo a propósito para centrarnos en las cuestiones básicas de la asignación de responsabilidades y el diseño de objetos. Sin embargo, en el desarrollo de la aplicación, es aconsejable tener en cuenta el manejo de las excepciones durante el trabajo de diseño y, por supuesto, durante la implementación.

Brevemente, en UML, las excepciones se representan como mensajes asíncronos en los diagramas de interacción. Esto se estudiará en el Capítulo 33.

20.7. Definición del método Venta-- crearLineaDeVenta

Como último ejemplo, también se puede escribir el método *crearLineaDeVenta* de la clase *Venta* inspeccionando el diagrama de colaboración *introducirArticulo*. La Figura 20.10 muestra una versión abreviada del diagrama de interacción, con el método Java acompañante.

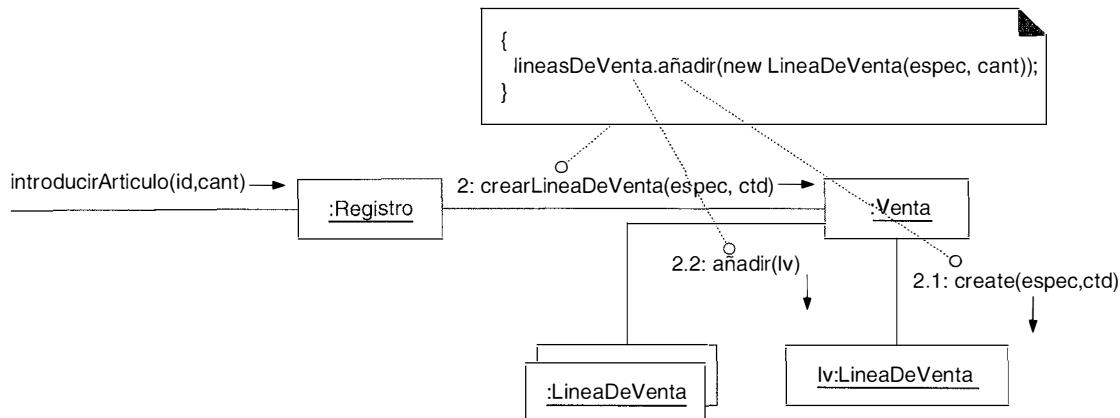


Figura 20.10. Método Venta--crearLineaDeVenta.

20.8. Orden de implementación

Es necesario implementar las clases (e idealmente, hacer las pruebas de unidad completamente) desde la menos a la más acoplada (ver Figura 20.11). Por ejemplo, las posibles primeras clases que podríamos implementar son *Pago* o *EspecificacionDelProducto*; a continuación las clases que sólo dependen de la implementación anterior —*CatalogoDeProductos* o *LineaDeVenta*—.

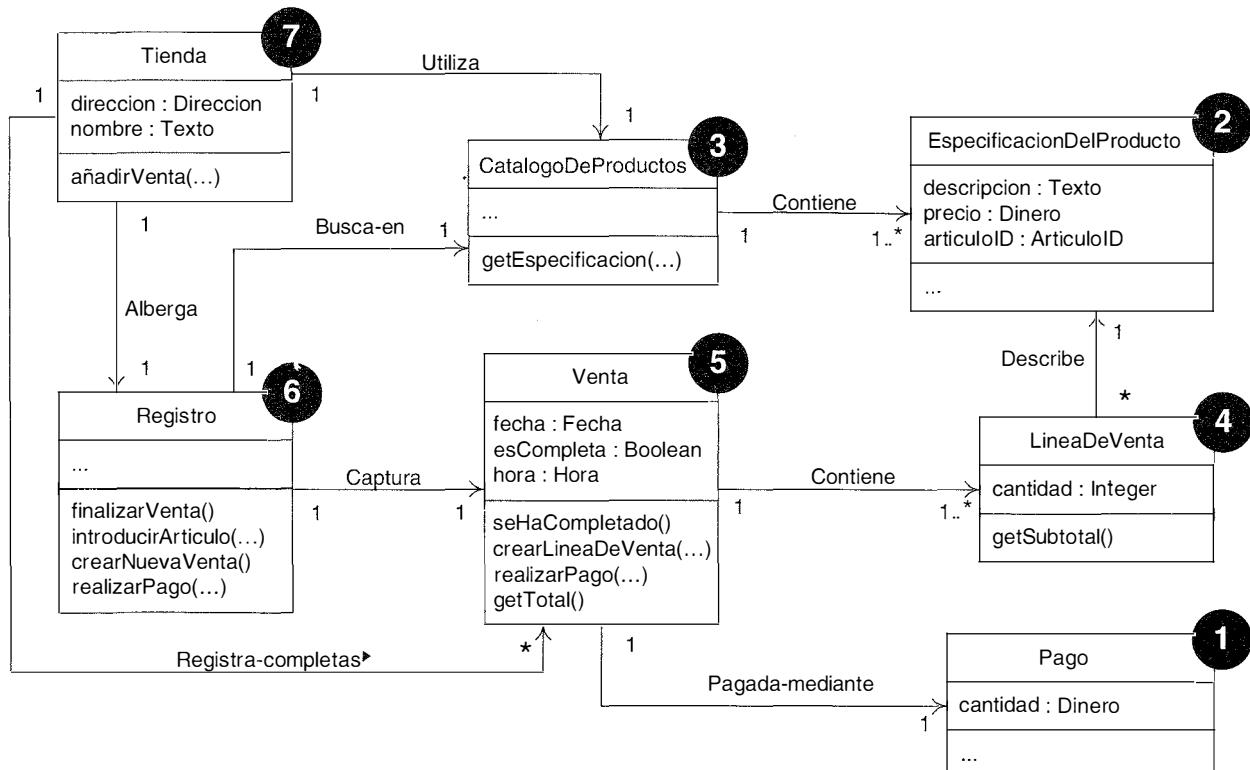


Figura 20.11. Posible orden para la implementación y prueba de las clases.

20.9. Programar probando primero

Una práctica excelente promovida por el método de Programación Extrema (XP, *eXtreme Programming*) [Beck00], y aplicable al UP (como son la mayoría de las prácticas de XP), es **programar probando primero**. En esta práctica, se escribe el código de las pruebas de unidad *antes* del código que se va a probar, y los desarrolladores escriben el código de las pruebas de unidad para *todo* el código de producción. La secuencia básica es escribir un poco del código de prueba, luego escribir un poco del código de producción, hacer que pase las pruebas, y entonces escribir más código de prueba, y así sucesivamente.

Entre las ventajas encontramos:

- **Se escriben realmente las pruebas de unidad:** La naturaleza humana (o al menos la de los programadores) es de tal manera que es muy común evitar la escritura de las pruebas de unidad, si se dejó como una idea adicional.
- **Satisfacción de los programadores:** Si un desarrollador escribe el código de producción, depura de manera informal, y entonces como ocurrencia tardía añade pruebas de unidad, no se siente muy satisfecho. Sin embargo, si se escriben primero las pruebas, y entonces se crea y refina el código de producción para que pase las pruebas, existe un sentimiento de logro —de pasar una prueba—. No se puede ignorar el aspecto psicológico del desarrollo —la programación es un esfuerzo humano—.
- **Aclaración de las interfaces y el comportamiento:** A menudo, la interfaz pública exacta y el comportamiento de una clase no están perfectamente claros hasta que se implementan. Escribiendo las pruebas de unidad primero, uno aclara el diseño de la clase.
- **Verificación demostrable:** Obviamente, el contar con cientos o miles de pruebas de unidad proporciona una verificación significativa de la corrección.
- **La confianza en cambiar cosas:** Si programamos probando primero, dispondremos de cientos o miles de pruebas de unidad, y una clase de pruebas de unidad por cada clase de producción. Cuando un desarrollador necesita cambiar el código existente —escrito por ellos mismos o por otros— existe un conjunto de pruebas de unidad que pueden ejecutarse, proporcionando una retroalimentación inmediata sobre si el cambio causó un error.

Como ejemplo, JUnit (www.junit.org) es un marco de pruebas de unidad para Java gratuito, simple y popular. Suponga que estamos utilizando JUnit y seguimos la estrategia de programar probando primero, para crear la clase *Venta*. Antes de programar la clase *Venta*, escribimos un método de prueba de unidad en la clase *PruebaVenta* que hace lo siguiente:

- Crea una nueva venta.
- Le añade algunas líneas de venta.
- Solicita el total y comprueba si tiene el valor esperado.

Por ejemplo,

```

public class PruebaVenta extends TestCase
{
    /**
     * ...
     */
    public void pruebaTotal()
    {
        //inicializa la prueba
        Dinero total = new Dinero(7.5);
        Dinero precio = new Dinero(2.5);
        ArticuloID id = new ArticuloID(1);
        EspecificacionDelProducto espec;
        espec = new EspecificacionDelProducto(id, precio, "producto 1");
        Venta venta = new Venta();

        //añade los artículos
        venta.crearLineaDeVenta(espec, 1);
        venta.crearLineaDeVenta(espec, 2);

        //Comprobar que el total es 7.5
        assertEquals(venta.getTotal(), total);
    }
}

```

Sólo después de crear esta clase *PruebaVenta* será cuando escribamos la clase *Venta* que pase esta prueba. Sin embargo, no es necesario escribir todos los métodos de prueba de antemano. Un desarrollador escribe un método de prueba, después el código de producción que la satisface, a continuación otro método de prueba, y así sucesivamente.

20.10. Resumen de la transformación de los diseños en código

El proceso de traducción de los DCD a las definiciones de las clases, y de los diagramas de interacción en los métodos, es relativamente directo. Durante el trabajo de programación todavía hay cabida para tomar decisiones, realizar cambios en el diseño y explorar, pero algunas de las grandes ideas del diseño se tuvieron en cuenta antes de la programación.

20.11. Introducción a la solución del programa

Esta sección presenta una muestra de la solución del programa de la capa de los objetos del dominio en Java para esta iteración. La generación del código se deriva casi por completo de los diagramas de clases del diseño y los diagramas de interacción que se definieron en el trabajo de diseño, y se basa en los principios de transformación de los diseños a código que se han estudiado previamente.

Lo más importante de este listado es mostrar que se traducen los artefactos de diseño a una base de código. Este código define un caso sencillo; la intención no es ilustrar un programa Java robusto y totalmente desarrollado con sincronización, manejo de excepciones, etcétera.

Clase Pago

```
public class Pago
{
    private Dinero cantidad;
    public Pago(Dinero dineroEntregado){ cantidad = dineroEntregado; }
    public Dinero getCantidad() { return cantidad; }
}
```

Clase CatalogoDeProductos

```
public class CatalogoDeProductos
{
    private Map especificacionesDeProductos = new HashMap();

    public CatalogoDeProductos()
    {
        // datos de ejemplo
        ArticuloID id1 = new ArticuloID( 100 );
        ArticuloID id2 = new ArticuloID( 200 );
        Dinero precio = new Dinero( 3 );

        EspecificacionDelProducto ep;
        ep = new EspecificacionDelProducto( id1, precio, "producto 1");
        especificacionesDeProductos.put( id1, ep );
        ep = new EspecificacionDelProducto( id2, precio, "producto 2");
        especificacionesDeProductos.put( id2, ep );
    }

    public EspecificacionDelProducto getEspecificacion (ArticuloID id)
    {
        return (EspecificacionDelProducto)especificacionesDeProductos.get(id);
    }
}
```

Clase Registro

```
public class Registro{
{
    private CatalogoDeProductos catalogo;
    private Venta venta;

    public Registro(CatalogoDeProductos catalogo)
    {
        this.catalogo = catalogo;
    }

    public void finalizarVenta()
    {
        venta.seHaCompletado();
    }

    public void introducirArticulo(ArticuloID id, int cantidad)
    {
        EspecificacionDelProducto espec = catalogo.getEspecificacion(id);
    }
}
```

```

        venta.crearLineaDeVenta( espec, cantidad);
    }

    public void crearNuevaVenta()
    {
        venta = new Venta();
    }

    public void realizarPago(Dinero dineroEntregado)
    {
        venta.realizarPago(dineroEntregado);
    }

}

```

Clase EspecificacionDelProducto

```

public class EspecificacionDelProducto
{
    private ArticuloID id;
    private Dinero precio;
    private String descripcion;

    public EspecificacionDelProducto
        (ArticuloID id, Dinero precio, String descripcion)
    {
        this.id = id;
        this.precio = precio;
        this.descripcion = descripcion;
    }

    public ArticuloID getArticuloID() { return id; }
    public Dinero getPrecio() { return precio; }
    public String getDescripcion() { return descripcion; }
}

```

Clase Venta

```

public class Venta
{
    private List lineasDeVenta = new ArrayList();
    private Date fecha = new Date();
    private boolean esCompleta() = false;
    private Pago pago;

    public Dinero getDevolucion()
    {
        return pago.getCantidad().minus( getTotal() );
    }

    public void seHaCompletado(){ esCompleta = true; }

    public boolean esCompleta(){ return esCompleta; }
}

```

```

public void crearLineaDeVenta
    (EspecificacionDelProducto espec, int cantidad)
{
    lineasDeVenta.add( new LineaDeVenta(espec, cantidad) );
}

public Dinero getTotal()
{
    Dinero total = new Dinero();
    Iterator i = lineasDeVenta.iterator();
    while (i.hasNext())
    {
        LineaDeVenta ldv = (LineaDeVenta)i.next();
        total.add(ldv.getSubtotal());
    }
    return total;
}

public void realizarPago(Dinero dineroEntregado)
{
    pago = new Pago( dineroEntregado );
}
}

```

Clase LineaDeVenta

```

public class LineaDeVenta
{
    private int cantidad;
    private EspecificacionDelProducto especProducto;

    public LineaDeVenta (EspecificacionDelProducto espec, int cantidad)
    {
        this.especProducto = espec;
        this.cantidad = cantidad;
    }

    public Dinero getSubtotal()
    {
        return especProducto.getPrecio().times( cantidad );
    }
}

```

Clase Tienda

```

public class Tienda
{
    private CatalogoDeProductos catalogo = new CatalogoDeProductos();
    private Registro registro = new Registro(catalogo);

    public Registro getRegistro() { return registro; }
}

```


Parte 4

ELABORACIÓN EN LA ITERACIÓN 2

LA ITERACIÓN 2 Y SUS REQUISITOS

21.1. Énfasis de la Iteración 2: diseño de objetos y patrones

Los capítulos de la fase de concepción y los de la Iteración 1 de la fase de elaboración han hecho hincapié en un conjunto de técnicas fundamentales del análisis y diseño de objetos, con el objeto de compartir información sobre una amplia gama de pasos comunes en la construcción de sistemas de objetos.

En esta iteración, el caso de estudio se centrará en:

- El diseño de objetos esencial.
- El uso de patrones para crear un diseño sólido.
- La aplicación de UML para visualizar los modelos.

Estos son objetivos básicos del libro y habilidades necesarias.

Hay una discusión mínima acerca del análisis de requisitos o el modelado del dominio, y la explicación del diseño es más concisa, ahora que (en la Iteración 1) ya se ha presentado una explicación detallada sobre las bases de cómo pensar en objetos.

Por supuesto, en esta iteración tendrían lugar muchas otras actividades de análisis, diseño e implementación, pero se les presta menos atención para centrarnos en proporcionar información sobre cómo hacer el diseño de objetos.

21.2. De la Iteración 1 a la 2

Cuando termina la Iteración 1, se debería cumplir lo siguiente:

- Se ha dedicado bastante esfuerzo a probar todo el software: unidad, aceptación, carga, facilidad de uso, etcétera. La idea en el UP es llevar a cabo verificaciones de calidad y corrección tempranas, realistas y continuas, de manera que la pronta retroalimentación guíe a los desarrolladores a adaptar y mejorar el sistema, encontrando su “camino verdadero”.
- Los clientes se han involucrado regularmente en la evaluación del sistema parcial, para obtener retroalimentación con el objeto de adaptar y clarificar los requisitos. Y los clientes consiguen ver pronto un progreso visible del sistema.
- El sistema, a través de todos los subsistemas, se ha integrado completamente y estabilizado como versión base interna.

En aras de la brevedad, se han omitido muchas actividades del final de la Iteración 1 e inicio de la Iteración 2, puesto que esta presentación concede mucha importancia a introducir el A/DDO. A continuación presentamos comentarios sobre unas pocas de las muchas actividades que se han saltado:

- Al comienzo de la nueva iteración, utilice una herramienta CASE para llevar a cabo ingeniería inversa y generar los diagramas UML a partir del código fuente de la última iteración (los resultados forman parte del Modelo de Diseño del UP). Éstos se pueden imprimir a gran tamaño con un plotter y colgarse en las paredes de la sala del proyecto, lo que nos ayudaría a comunicar la representación del punto de partida del diseño lógico de la siguiente iteración.
- Se ha comenzado el análisis e ingeniería de usabilidad para la UI. Ésta es una técnica y una actividad extraordinariamente importantes para el éxito de muchos sistemas. Sin embargo, se trata de un tema extenso y no trivial, y queda fuera del alcance de este libro.
- Se ha comenzado el modelado e implementación de la base de datos.
- Se eligen los requisitos de la siguiente iteración cerca del final de la iteración anterior.
- Tiene lugar otro taller de requisitos de dos días (por ejemplo), en el que se escriben más casos de uso en formato completo. Durante la elaboración, mientras quizás se han diseñado e implementado el 10% de los requisitos de más riesgo, hay una actividad *paralela* para estudiar y definir en profundidad quizás el 80% de los casos de uso del sistema, aunque la mayoría de los requisitos no se implementarán hasta la construcción.
 - Entre los participantes se incluirán unos pocos desarrolladores (como el arquitecto de software) de la primera iteración, de manera que, durante este taller, la investigación y las preguntas se documentan a partir de la comprensión (y confusión) adquirida con la construcción rápida y real de parte del software. No hay nada como construir algo del software para descubrir qué es lo que realmente no

conocemos acerca de los requisitos —ésta es una idea clave en el UP y en el desarrollo iterativo—.

Simplificaciones en el caso de estudio

En un proyecto UP bien hecho, los requisitos que se eligen para las primeras iteraciones se organizan de acuerdo con el riesgo y el alto valor para el negocio, de manera que se identifican y se resuelven primero las cuestiones de más riesgo. Sin embargo, si este caso de estudio siguiera exactamente esta estrategia, no ayudaría a explicar las ideas y principios fundamentales del A/DDO en las primeras iteraciones. Por tanto, se han tomado algunas licencias con respecto a la prioridad de los requisitos, prefiriendo aquellos que apoyan los objetivos pedagógicos, en lugar de los objetivos de riesgo del proyecto.

21.3. Requisitos de la Iteración 2

La Iteración 2 de la aplicación del PDV NuevaEra maneja varios requisitos interesantes:

1. Soporte a las variaciones de servicios externos de terceras partes. Por ejemplo, se deben conectar al sistema diferentes calculadores de impuestos, cada uno con su propia interfaz. Del mismo modo con diferentes sistemas de contabilidad, etcétera. Cada uno ofrecerá un API y protocolo diferentes para un núcleo de las funciones comunes.
2. Reglas de fijación de precios complejas.
3. Reglas de negocio conectables.
4. Un diseño para actualizar una ventana del GUI cuando cambia el total de la venta.

Sólo se tendrán en cuenta estos requisitos (en esta iteración) en el contexto de los escenarios del caso de uso *Procesar Venta*.

Nótese que no se han descubierto requisitos nuevos; se identificaron durante la fase de inicio. Por ejemplo, el caso de uso *Procesar Venta* original señala el problema de la fijación de precios:

Escenario principal de éxito:

1. El Cliente llega a un terminal PDV con mercancías y/o servicios que comprar.
 2. El Cajero le dice al sistema que cree una nueva venta.
 3. El Cajero introduce el identificador del artículo.
 4. El Sistema registra la línea de la venta y presenta la descripción del artículo, precio y suma parcial. ***El precio se calcula a partir de un conjunto de reglas de precios.***
- ...

Además, las secciones en la Especificación Complementaria registran los detalles de las reglas del dominio para los precios, y señalan la necesidad de soportar varios sistemas externos:

Especificación Complementaria

...

Interfaces

Interfaces software

Para la mayoría de los sistemas externos de colaboración (calculadora de impuestos, contabilidad, inventario,...) necesitamos ser capaces de conectar diversos sistemas y, por tanto, diversas interfaces.

...

Reglas de Dominio (Negocio)

ID	Regla	Grado de variación	Fuente
REGLA4	Reglas de descuento al comprador. Ejemplos: Empleados—20% de descuento Clientes preferentes—10% de descuento Antiguos—15% de descuento	Alto. Cada tienda utiliza reglas diferentes.	Política de la tienda.
...

Información en dominios de interés

Fijación de precios

Además de las reglas de fijación de precios que se describen en la sección de reglas del dominio, nótese que los artículos tienen un *precio original*, y opcionalmente, un *precio rebajado*. El precio de los artículos (antes de los descuentos adicionales) es el precio rebajado, si existe. Las organizaciones mantienen el precio original incluso si hay un precio rebajado, por razones de contabilidad e impuestos.

...

Desarrollo incremental para el mismo caso de uso a lo largo de la iteraciones

Debido a estos requisitos, en la Iteración 2 estamos volviendo sobre el caso de uso *Procesar Venta*, pero implementando más escenarios, de manera que el sistema crece incrementalmente. Es habitual trabajar en varios escenarios o características del mismo caso de uso a lo largo de varias iteraciones y extender el sistema gradualmente para finalmente manejar toda la funcionalidad requerida. Por otro lado, los casos de uso breves y simples, podrían implementarse completamente en una iteración.

La Iteración 1 hizo simplificaciones de manera que el problema y la solución no fueran excesivamente complejas de abordar. Una vez más —por la misma razón— se considera una cantidad relativamente pequeña de funcionalidad adicional.

En un proyecto de desarrollo real los requisitos que se han escogido en el libro para esta iteración no serían la elección indiscutible —otra posibilidad es actualizar el inventario—, gestionar los pagos a crédito, o un caso de uso completamente diferente. Sin embargo, ésta es una elección de suficiente riqueza con oportunidades de aprendizaje valiosas.

21.4. Refinamiento de los artefactos orientados al análisis en esta iteración

Modelo de casos de uso: casos de uso

Como resultado de los requisitos escogidos para esta iteración no se requiere ningún refinamiento de los casos de uso, aunque podrían cambiar como resultado de otras percepciones.

Sin embargo, además del diseño de objetos y la programación, en esta iteración tendrá lugar una actividad paralela de un breve taller de requisitos, con el que se investigarán y escribirán en detalle más casos de uso. Se revisarán los anteriores casos de uso en formato completo (por ejemplo, *Procesar Venta*) y probablemente se refinarán en base al conocimiento adquirido a partir de la Iteración 1. Algunas de estas actualizaciones de los casos de uso se podrían tener en cuenta en la siguiente iteración de la fase de elaboración, pero muchas se pospondrán hasta la construcción (porque no son significativas o arriesgadas para la arquitectura).

Modelo de casos de uso: DSS

Esta iteración abarca la inclusión del soporte para sistemas externos de terceras partes con varias interfaces, como un calculador de impuestos. El sistema del PDV NuevaEra se comunicará de manera remota con los sistemas externos. En consecuencia, deberían actualizarse los DSS para reflejar al menos algunas de las colaboraciones inter-sistemas, para clarificar cuáles son los nuevos eventos de nivel del sistema.

La Figura 21.1 ilustra un DSS para un escenario de pago a crédito, que requiere la colaboración con varios sistemas externos. Aunque en esta iteración no se maneja el pago a crédito, el diseñador (yo) ha dibujado un DSS basado en él (y probablemente varios otros también), para entender mejor la colaboración entre los sistemas y, por tanto, el soporte que se requiere para distintas interfaces en los sistemas externos.

Modelo del Dominio

Después de haber adquirido un poco de experiencia en el modelado del dominio, un modelador puede estimar si un conjunto de requisitos nuevos tendrán un mayor o menor impacto en el Modelo del Dominio, en términos de muchos nuevos conceptos, asociaciones y atributos. A diferencia de la iteración anterior, los requisitos que se van a abordar esta vez no comprenden muchos conceptos del dominio nuevos. Un breve estudio de los nuevos requisitos sugiere algo como *ReglaDePrecio* como concepto del dominio, pero probablemente no hay docenas de cosas nuevas.

En esta situación, es bastante razonable saltarse el refinamiento del Modelo del Dominio, y pasar rápidamente al trabajo de diseño, y dejar que tenga lugar el descubrimiento de nuevos conceptos del dominio durante el diseño de objetos, cuando los diseñadores están considerando detalladamente una solución. Una señal de madurez en el proceso con el UP es entender cuándo la creación de un nuevo artefacto añadirá un

valor significativo, o es una especie de paso mecánico “hacer trabajo” y es mejor saltarlo.

Esta flexibilidad es un arma de doble filo. Con mucha frecuencia, la flexibilidad de saltarse las actividades anteriores a la programación tiene lugar dentro de una creencia excesivamente optimista de que el problema se puede solucionar simplemente precipitándose hacia el código. Si verdaderamente puede, estupendo, porque la programación es el trabajo que realmente importa, no dibujar modelos del dominio. Por otro lado, la mayoría de los desarrolladores tienen historias donde un poco de reflexión, investigación y previsión antes de programar habrían reducido dolor y sufrimiento.

Modelo de caso de uso: contratos de las operaciones del sistema

En esta iteración no se van a considerar nuevas operaciones del sistema y, por tanto, no se necesitan los contratos. En cualquier caso, los contratos son únicamente una opción a tener en cuenta cuando la precisión detallada que ofrecen mejora las descripciones de los casos de uso.

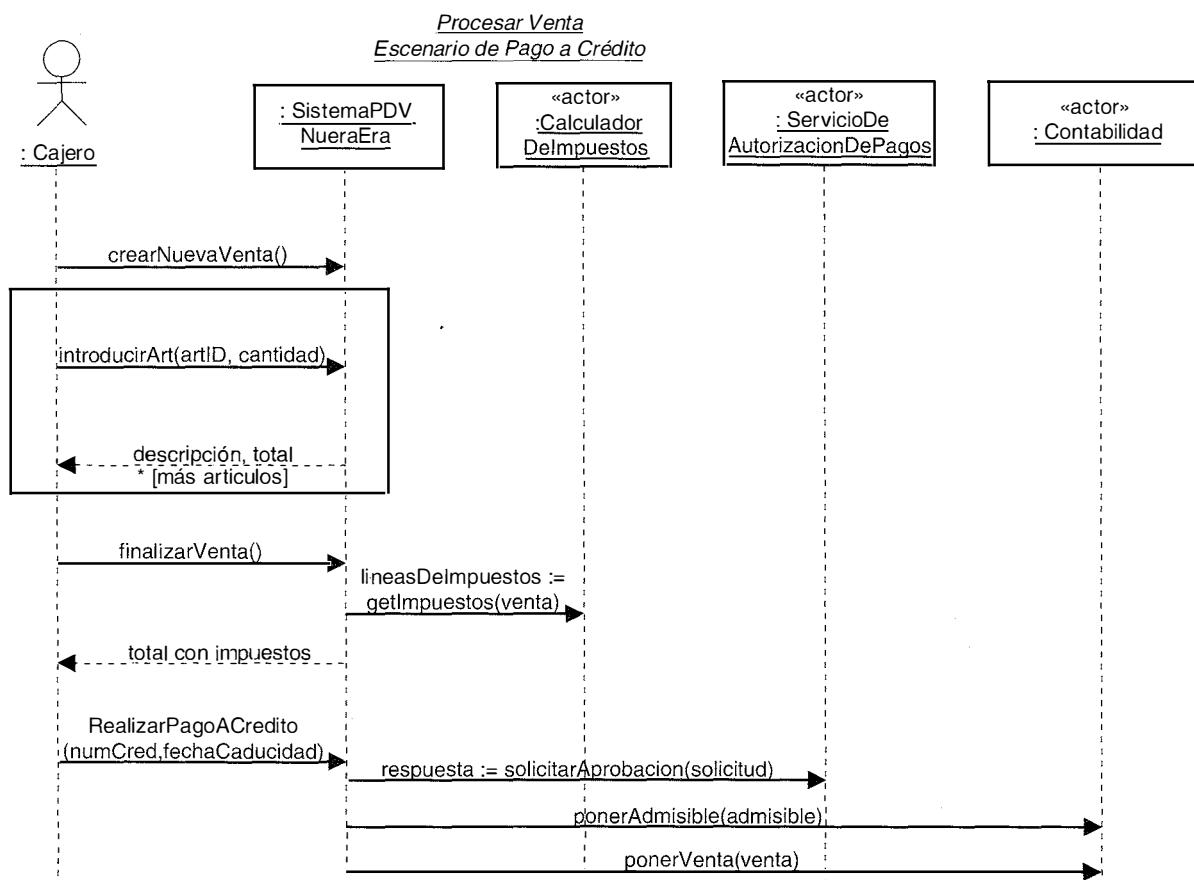


Figura 21.1. Un escenario DSS que ilustra algunos eventos externos.

Capítulo 22

GRASP: MÁS PATRONES PARA ASIGNAR RESPONSABILIDADES

La suerte es el residuo del diseño.

Branch Rickey

Objetivos

- Aprender a aplicar el resto de los patrones GRASP.
-

Introducción

Anteriormente, estudiamos la aplicación de los cinco primeros patrones GRASP:

- Experto en Información, Creador, Alta Cohesión, Bajo Acoplamiento y Controlador.

Los cuatro últimos patrones GRASP son:

- Polimorfismo.
- Indirección.
- Fabricación Pura.
- Variaciones Protegidas.

Una vez que se hayan explicado, tendremos un vocabulario rico y de uso común con el que discutir los diseños. Y este vocabulario crecerá puesto que también se presentan (en capítulos siguientes) algunos de los patrones de diseño de la “pandilla de los cuatro” (GoF, *Gang-of-Four*) (como Estrategia y Factoría). Una frase corta como, “Sugiero una Estrategia generada a partir de una Factoría para soportar Variaciones Protegidas y Bajo Acoplamiento con respecto a <X>” revela mucha información sobre el diseño, puesto que el nombre de los patrones transmite de manera concisa un concepto de diseño complejo.

Este capítulo presenta el resto de los patrones GRASP, una ayuda para aprender los principios fundamentales según los cuales se asignan las responsabilidades a los objetos y se diseñan éstos.

Los capítulos siguientes introducen otros patrones útiles y los aplican al desarrollo de la segunda iteración de la aplicación del PDV NuevaEra.

22.1. Polimorfismo

Solución Cuando las alternativas o comportamientos relacionados varían según el tipo (clase), asigne la responsabilidad para el comportamiento —utilizando operaciones polimórficas— a los tipos para los que varía el comportamiento¹.

Corolario: No haga comprobaciones acerca del tipo de un objeto y no utilice la lógica condicional para llevar a cabo alternativas diferentes basadas en el tipo.

Problema ¿Cómo manejar las alternativas basadas en el tipo? ¿Cómo crear componentes software conectables (*pluggable*)?

Alternativas basadas en el tipo: La variación condicional es un tema fundamental en los programas. Si se diseña un programa utilizando sentencias de lógica condicional *if-then-else* o *case*, entonces, si aparece una nueva variación, requiere la modificación de la lógica de casos. Este enfoque dificulta que el programa se extienda con facilidad con nuevas variaciones porque se tiende a necesitar los cambios en varios sitios —en donde quiera que exista la lógica condicional.

Componentes software conectables: Viendo los componentes en las relaciones cliente-servidor, ¿cómo podemos sustituir un componente del servidor por otro, sin afectar al cliente?

Ejemplo En la aplicación del PDV NuevaEra, se deben soportar diferentes sistemas externos de cálculo de impuestos de terceras partes (como Master-En-Impuestos e Impuestos-Pro); el sistema necesita ser capaz de integrarse con distintos de ellos. Cada calculador de impuestos tiene un interfaz diferente, con lo que hay un comportamiento parecido pero que varía para adaptarse a cada uno de estos interfaces externos o APIs. Un producto podría soportar un simple protocolo de sockets TCP, otro podría ofrecer un interfaz SOAP, y un tercero podría ofrecer un interfaz RMI de Java.

¿Qué objetos deberían ser los responsables de manejar estas interfaces diferentes de los calculadores de impuestos externos?

Puesto que el comportamiento para la adaptación del calculador varía según el tipo de calculador, según el Polimorfismo deberíamos asignar la responsabilidad de la adaptación a los diferentes objetos calculadores (o adaptadores de calculador), implementada con una operación polimórfica *getImpuestos* (ver Figura 22.1).

¹ El **polimorfismo** tiene varios significados relacionados. En este contexto, significa “asignar el mismo nombre a servicios en diferentes objetos” [Coad95] cuando los servicios son parecidos o están relacionados. Los diferentes tipos de objetos normalmente implementan un interfaz común o están relacionados en una jerarquía de implementación con una superclase común, pero esto depende del lenguaje; por ejemplo, los lenguajes con ligadura dinámica como Smalltalk no requieren esto.

Estos objetos adaptadores de los calculadores no son los calculadores externos, sino objetos software locales que representan los calculadores externos, o el adaptador para el calculador. Enviando un mensaje al objeto local, finalmente se hará una llamada al calculador externo a través de su API.

Cada método *getImpuestos* toma el objeto *Venta* como parámetro, de manera que el calculador pueda analizar la venta. La implementación de cada método *getImpuestos* será diferente: el *AdaptadorMasterEnImpuestos* adaptará la solicitud al API del Master-En-Impuestos, y así sucesivamente.

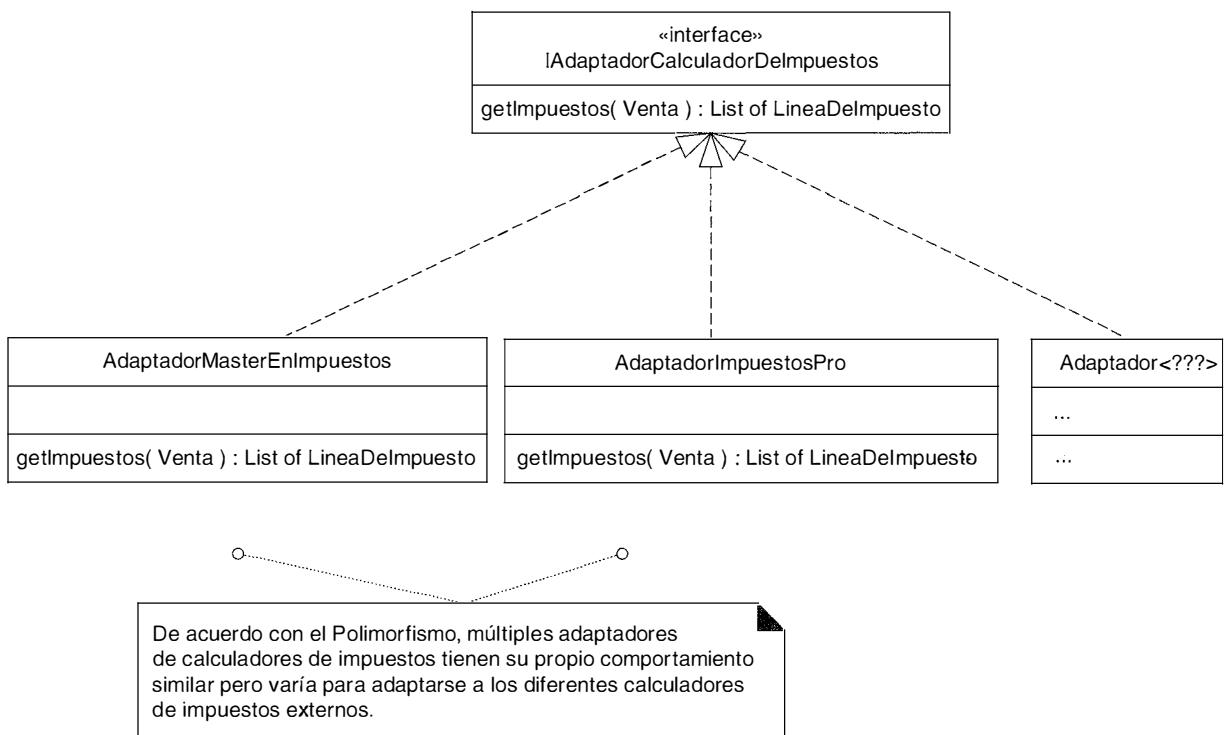


Figura 22.1. Polimorfismo en la adaptación a diferentes calculadores de impuestos externos.

Notación UML: La Figura 22.1 presenta alguna notación nueva para especificar las **interfaces** (un descriptor de operaciones sin implementación), la implementación de las interfaces, y los valores de retorno de tipo “colección”; la Figura 22.2 lo explica en detalle. Se utiliza un **estereotipo** UML para designar una interfaz; un estereotipo es un mecanismo para crear una nueva categoría de elementos. El nombre de un estereotipo se pone entre comillas francesas, como en «interface». Las comillas francesas son un signo ortográfico de *un único* carácter conocido sobre todo por su uso en la tipografía francesa para indicar una cita; pero citando a Rumbaugh, “quienes tengan problemas tipográficos pueden utilizar dobles corchetes angulares (<>>)” [RJB99].

Discusión

El polimorfismo es un principio fundamental para diseñar cómo se organiza el sistema para gestionar variaciones similares. Según el Polimorfismo, un diseño basado en la asignación de responsabilidades puede extenderse fácilmente para manejar nuevas variaciones. Por ejemplo, la inclusión de una nueva clase adaptador de calculadora con su propio método polimórfico *getImpuestos* tendrá un impacto menor en el diseño existente.

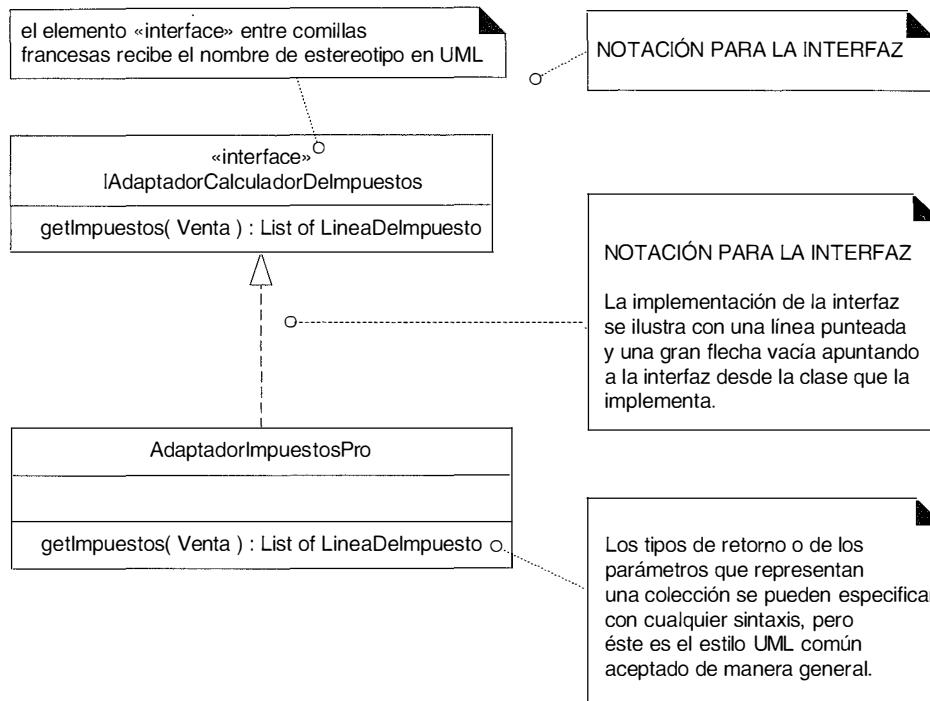


Figura 22.2. Notación UML para las interfaces y los tipos de retorno.

Contraindicaciones

Algunas veces, los desarrolladores diseñan sistemas con interfaces y polimorfismo para “futuras necesidades” especulativas frente a posibles variaciones desconocidas. Si el punto de variación está motivado sin ninguna duda por una variabilidad inmediata o muy probable, entonces, por supuesto, es razonable el esfuerzo de añadir la flexibilidad por medio del polimorfismo. Pero es conveniente una evaluación crítica, puesto que no es raro que se dedique un esfuerzo innecesario a preparar para el futuro un diseño con polimorfismo en puntos de variación que en realidad son improbables y nunca aparecerán realmente. Sea realista sobre la verdadera probabilidad de la posible variación antes de invertir en incrementar la flexibilidad.

Beneficios

- Se añaden fácilmente las extensiones necesarias para nuevas variaciones.
- Las nuevas implementaciones se pueden introducir sin afectar a los clientes.

Patrones Relacionados

- Variaciones Protegidas.
- Varios de los populares patrones de diseño GoF [GHJV95], que se estudiarán en este libro dependen del polimorfismo, entre los que se encuentran los patrones Adaptador, Command, Composite, Proxy, Estado y Estrategia.

También conocido como; parecido a

Elección del Mensaje, No preguntes “¿Qué clase?”.

22.2. Fabricación Pura

Solución

Asigne un conjunto de responsabilidades altamente cohesivo a una clase artificial o de conveniencia que no representa un concepto del dominio del problema —algo inventado para soportar alta cohesión, bajo acoplamiento y reutilización—.

Tal clase es una *fabricación* de la imaginación. Idealmente, las responsabilidades asignadas a esta fabricación soportan alta cohesión y bajo acoplamiento, de manera que el diseño de la fabricación es muy limpio, o *puro* —de ahí que sea una fabricación pura—.

Finalmente, una fabricación pura implica construir algo, ¡lo que hacemos cuando estamos desesperados!

Problema

¿Qué objetos deberían tener la responsabilidad cuando no quiere violar los objetivos de Alta Cohesión y Bajo Acoplamiento, u otros, pero las soluciones que ofrece el Experto (por ejemplo) no son adecuadas?

Los diseños orientados a objetos algunas veces se caracterizan por implementar como clases software las representaciones de los conceptos del dominio del mundo real para reducir el salto en la representación; por ejemplo, una clase *Venta* y *Cliente*. Sin embargo, hay muchas situaciones en las que la asignación de las responsabilidades sólo a las clases software de la capa del dominio da lugar a problemas en cuanto a cohesión y acoplamiento pobres, o que el potencial para reutilizar sea bajo.

Ejemplo

Por ejemplo, suponga que se necesita soporte para almacenar las instancias de *Venta* en una base de datos relacional. Según el Experto en Información, se puede justificar la asignación de esta responsabilidad a la propia clase *Venta*, puesto que la venta tiene los datos que se necesitan almacenar. Pero considere las siguientes implicaciones:

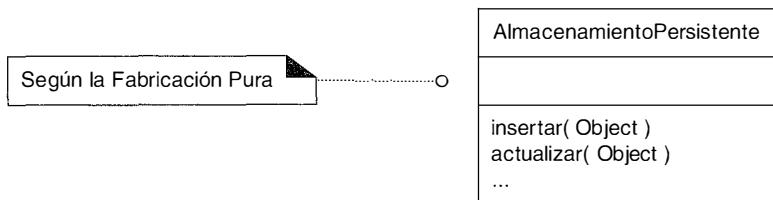
- La tarea requiere un número relativamente amplio de operaciones orientadas al soporte de la base de datos, ninguna relacionada con el concepto de ventas, luego la clase *Venta* deja de tener cohesión.
- La clase *Venta* tiene que acoplarse a la interfaz de la base de datos relacional (como JDBC en las tecnologías Java), luego el acoplamiento aumenta. E incluso el acoplamiento no se produce con otro objeto del dominio, sino con un tipo particular de interfaz de base de datos.
- Almacenar los objetos en una base de datos relacional es una tarea muy general que muchas clases necesitan soportar. La colocación de esta responsabilidad en la clase *Venta* da a entender que la reutilización va a ser pobre o que va a haber mucha duplicación en otras clases que hagan lo mismo.

Por tanto, aunque la *Venta* es una candidata lógica en virtud del Experto en Información para almacenarse a sí misma en una base de datos, nos lleva a un diseño con baja cohesión, alto acoplamiento y bajo potencial para reutilizar —exactamente el tipo de situación desesperada que pide que se cree algo—.

Una solución razonable es crear una nueva clase que es responsable únicamente de almacenar los objetos en algún tipo de medio de almacenamiento persistente, como una base de datos relacional; llamémosla *AlmacenamientoPersistente*². Esta clase es una Fabricación Pura —una invención de la imaginación—.

Obsérvese el nombre: *AlmacenamientoPersistente*. Es un concepto que se entiende, pero el nombre o concepto de “almacenamiento persistente” no es algo que uno encon-

² En un *framework* de persistencia real, se necesita finalmente más de una única clase de fabricación pura para crear un diseño razonable. Este objeto será una fachada *front-end* sobre un gran número de objetos de ayuda *back-end*.



traría en el Modelo del Dominio. Y si un diseñador le pregunta a una persona del negocio en una tienda, “¿Trabaja con objetos en almacenamiento persistente?” no le entendería. Ellos entienden conceptos como “venta” y “pago”. El *AlmacenamientoPersistente* no es un concepto del dominio, sino algo creado o fabricado para facilitar las cosas al desarrollador de software.

Esta Fabricación Pura soluciona los siguientes problemas de diseño:

- La *Venta* permanece bien diseñada, con alta cohesión y bajo acoplamiento.
- La clase *AlmacenamientoPersistente* es relativamente cohesiva, teniendo como único objetivo el almacenamiento o inserción de los objetos en un medio de almacenamiento persistente.
- La clase *AlmacenamientoPersistente* es un objeto muy genérico y reutilizable.

La creación de una fabricación pura en este ejemplo es exactamente la situación en la que se requiere su uso —sustituir un mal diseño basado en el Experto, con cohesión y acoplamiento pobres, por un buen diseño en el que el potencial para reutilizar es mayor—.

Observe, como con todos los patrones GRASP, se concede mucha importancia al lugar donde se debería colocar la responsabilidad. En este ejemplo las responsabilidades se cambian de la clase *Venta* (motivada según el Experto) a una Fabricación Pura.

Discusión

El diseño de los objetos se puede dividir en general en dos grupos:

1. Los escogidos de acuerdo a una **descomposición de la representación**.
2. Los escogidos según una **descomposición del comportamiento**.

Por ejemplo, se crean las clases software como la *Venta* de acuerdo a una descomposición de la representación; la clase software está relacionada o representa una cosa en un dominio. La descomposición de la representación es una estrategia común en el diseño de objetos y favorece el objetivo de salto en la representación reducido. Pero algunas veces, deseamos asignar responsabilidades agrupando comportamientos o algoritmos, sin que interese crear una clase con un nombre u objetivo que esté relacionado con un concepto del dominio del mundo real.

Un buen ejemplo es un objeto “algoritmo” como un *GeneradorDeTablaDeContenidos*, cuyo objetivo es (sorpresa) generar una tabla de contenidos y lo creó un desarrollador como clase de ayuda o de conveniencia, sin preocuparse por elegir un nombre del vocabulario del dominio de libros y documentos. Existe como clase de conveniencia concebida por el desarrollador para agrupar algún comportamiento o métodos relacionados y, por tanto, está motivada según la *descomposición de comportamiento*.

En cambio, una clase software denominada *TablaDeContenidos* está inspirada siguiendo una *descomposición de la representación*, y debería contener información consistente con nuestra concepción del dominio real (como los nombres de los capítulos).

No es crítico identificar una clase como Fabricación Pura. Es un objetivo educativo transmitir la idea general de que algunas clases software se inspiran de acuerdo con las representaciones del dominio, y otras simplemente se “crean” por conveniencia del diseñador de objetos. Estas clases de conveniencia normalmente se crean para agrupar algún comportamiento común y, por tanto, se inspiran según una descomposición de comportamiento en lugar de una descomposición de la representación.

Dicho de otro modo, una Fabricación Pura normalmente se organiza en base a funcionalidad relacionada, y de este modo es un objeto centrado en la función o de comportamiento.

Muchos de los patrones de diseño orientados a objetos existentes son ejemplo de Fabricación Pura: Adaptador, Estrategia, *Command*, etcétera [GHJV95].

Como comentario final merece la pena reiterar: Algunas veces la solución que ofrece el Experto en Información no es conveniente. Aunque el objeto sea un candidato porque tiene mucha de la información relacionada con la responsabilidad, en otros aspectos, su elección nos lleva a un diseño pobre, normalmente debido a problemas en la cohesión o el acoplamiento.

Beneficios

- Se soporta Alta Cohesión puesto que las responsabilidades se factorizan en una clase de grano fino que sólo se centra en un conjunto muy específico de tareas relacionadas.
- El potencial para reutilizar podría aumentar debido a la presencia de clases de Fabricación Pura de grano fino cuyas responsabilidades tienen aplicación en otras aplicaciones.

Contraindicaciones

Algunas veces, los inexpertos en el diseño de objetos y que conocen bien la descomposición u organización del software en base a sus funciones, abusan de la descomposición de comportamiento en los objetos de Fabricación Pura. Exagerando, sucede que las funciones se convierten en objetos. No existe nada inherentemente incorrecto en la creación de objetos “función” o “algoritmo”, pero necesita equilibrarse con la capacidad de diseñar con descomposición de la representación, como la capacidad de aplicar el Experto en Información de manera que una clase de la representación como la *Venta* también tenga responsabilidades. El Experto en Información enuncia el objetivo de colocar las responsabilidades con los objetos que conocen la información necesaria para esas responsabilidades, lo cual tiende a favorecer la disminución del acoplamiento. Si se abusa, la Fabricación Pura podría dar lugar a demasiados objetos de comportamiento con responsabilidades que *no* se colocaron con la información necesaria para su realización, lo que puede afectar negativamente al acoplamiento. El síntoma habitual es que la mayoría de los datos contenidos en los objetos se pasan a otros objetos para trabajar con ellos.

Patrones y Principios Relacionados

- Bajo Acoplamiento.
- Alta Cohesión.
- Usualmente una Fabricación Pura asume responsabilidades de las clases del dominio a las que se les asignaría esas responsabilidades en base al patrón Experto.
- Todos los patrones de diseño GoF [GHJV95], como el Adaptador, *Command*, Estrategia, etcétera, son Fabricaciones Puras.
- Prácticamente el resto de patrones de diseño son Fabricaciones Puras.

22.3. Indirección

Solución

Asigne la responsabilidad a un objeto intermedio que medie entre otros componentes o servicios de manera que no se acoplen directamente.

El intermediario crea una *indirección* entre los otros componentes.

Problema

¿Dónde asignar una responsabilidad, para evitar el acoplamiento directo entre dos (o más) cosas? ¿Cómo desacoplar los objetos de manera que se soporte el bajo acoplamiento y el potencial para reutilizar permanezca más alto?

Ejemplos

AdaptadorCalculadorDeImpuestos

Estos objetos actúan como intermediarios con los calculadores de impuestos externos. Mediante el polimorfismo, proporcionan una interfaz consistente para los objetos internos y ocultan las variaciones en las APIs externas. Añadiendo un nivel de indirección y el polimorfismo, los objetos adaptador protegen el diseño interno frente a las variaciones de las interfaces externas (ver Figura 22.3).

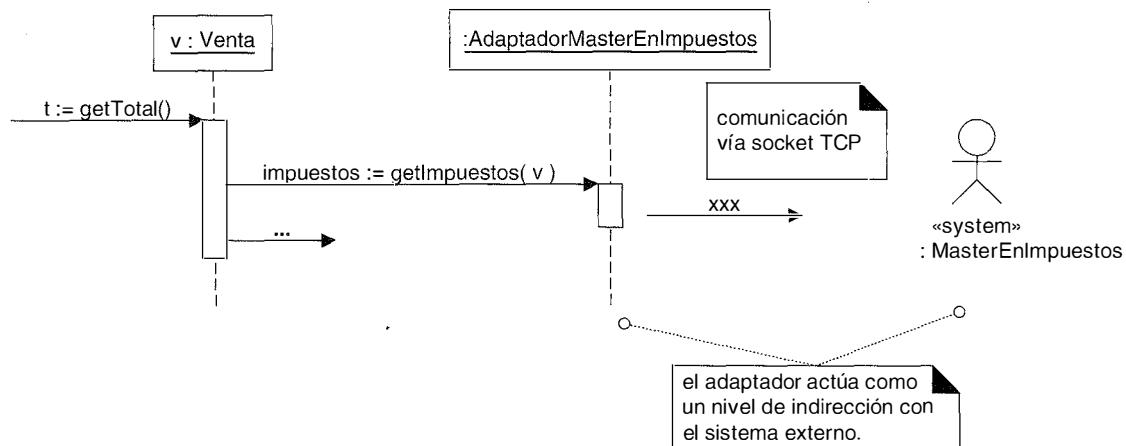


Figura 22.3. Indirección por medio del adaptador.

AlmacenamientoPersistente

El ejemplo de Fabricación Pura de desacoplar la *Venta* de los servicios de base de datos relacional mediante la incorporación de una clase *AlmacenamientoPersistente*, también es un ejemplo de asignación de responsabilidades para soportar la Indirección. El *AlmacenamientoPersistente* actúa como intermediario entre la *Venta* y la base de datos.

Discusión

“La mayoría de los problemas en informática se pueden resolver mediante otro nivel de indirección” es un viejo dicho de particular relevancia en los diseños orientados a objetos³.

³ ¡Si algún dicho es viejo en informática! He olvidado la fuente (¿Parnas?). Nótese que también existe el dicho contrario: “¡La mayoría de los problemas en ejecución se pueden resolver eliminando alguna capa de indirección!”.

Del mismo modo que muchos de los patrones de diseño existentes son especializaciones de Fabricación Pura, muchos son también especializaciones de Indirección. Algunos ejemplos son Adaptador, Fachada y Observador. Además, muchas Fabricaciones Puras se generan debido a la Indirección. El motivo para la Indirección normalmente es el Bajo Acoplamiento; se añade un intermediario para desacoplar otros componentes o servicios.

- | | |
|---|--|
| Beneficios | <ul style="list-style-type: none"> • Disminuir el acoplamiento entre los componentes. |
| Patrones y Principios Relacionados | <ul style="list-style-type: none"> • Variaciones Protegidas. • Bajo Acoplamiento. • Muchos patrones GoF como Adaptador, Puente, Fachada, Observador y Mediador [GHJV95]. • Muchos intermediarios de Indirección son Fabricaciones Puras. |

22.4. Variaciones Protegidas

- | | |
|------------------|--|
| Solución | Identifique los puntos de variaciones previstas o de inestabilidad; asigne responsabilidades para crear una interfaz estable alrededor de ellos. |
| | Nota: El término “interfaz” se utiliza en el sentido más general de una vista de acceso; no sólo significa literalmente algo como una interfaz de Java o COM. |
| Problema | ¿Cómo diseñar objetos, subsistemas y sistemas de manera que las variaciones o inestabilidades en estos elementos no tengan un impacto no deseable en otros elementos? |
| Ejemplo | Por ejemplo, el problema anterior del calculador de impuestos externo y su solución con el Polimorfismo ilustra las Variaciones Protegidas (Figura 22.1). El punto de inestabilidad o variación lo forman las diferentes interfaces o APIs de los calculadores de impuestos externos. El sistema de PDV necesita ser capaz de integrarse con muchos sistemas de cálculo de impuestos existentes, y también con futuros calculadores de terceras partes que no existen todavía. |
| | Añadiendo un nivel de indirección, una interfaz, y utilizando el polimorfismo con varias implementaciones de <i>IAdaptadorCalculadorDeImpuestos</i> , se consigue proteger al sistema de las variaciones en las APIs externas. Los objetos internos colaboran con una interfaz estable; las distintas implementaciones del adaptador ocultan las variaciones de los sistemas externos. |
| Discusión | Cockburn publicó por primera vez las Variaciones Protegidas (VP) como patrón en [VCK96], aunque este principio de diseño tan fundamental ha estado por ahí durante décadas bajo distintos términos. |

Mecanismos motivados por VP

VP es un principio fundamental que motiva la mayoría de los mecanismos y patrones en la programación y el diseño destinados a proporcionar flexibilidad y protección frente a las variaciones.

A un nivel, se puede ver la madurez de un desarrollador o arquitecto en su conocimiento creciente de mecanismos de amplia aplicación para conseguir VP, elegir las batallas VP apropiada que merecen la pena pelear, y su capacidad para escoger una solución de VP adecuada. En las primeras etapas, uno aprende acerca de la encapsulación de datos, interfaces y polimorfismo —todos ellos mecanismos básicos para alcanzar VP—. Más tarde, uno aprende técnicas como los lenguajes basados en reglas, intérpretes de reglas, diseños basados en reflexión y metadatos, máquinas virtuales, etcétera —todas ellas se pueden aplicar para proteger contra alguna variación—.

Por ejemplo:

Mecanismos básicos de Variaciones Protegidas

VP motiva la encapsulación de datos, interfaces, polimorfismo, indirección y estándares. Nótese que los componentes como gestores (*brokers*) y máquinas virtuales son ejemplos complejos de indirección para conseguir VP.

Diseños dirigidos por los datos

Los diseños dirigidos por los datos cubren una amplia familia de técnicas entre las que se encuentran la lectura de códigos, valores, rutas de ficheros de clase, nombres de clase, etcétera, procedentes de una fuente externa para cambiar el comportamiento, o “parametrizar” un sistema de alguna manera en tiempo de ejecución. Otras variantes pueden ser hojas de estilo, metadatos para la correspondencia objeto-relacional, ficheros de propiedades, lectura de distribución de los elementos de una ventana y muchos más. Se protege al sistema del impacto de variaciones de los datos, metadatos o declarativas registrando externamente la variante, leyéndola, y razonando sobre ella.

Búsqueda de servicios

La búsqueda de servicios incluye técnicas como la utilización de servicios de nombres (por ejemplo, JNDI de Java) o intermediarios (*traders*) para obtener un servicio (por ejemplo, Jini de Java, o UDDI para los servicios Web). Se protegen a los clientes de las variaciones en la ubicación de los servicios, utilizando la interfaz estable de la búsqueda del servicio. Es un caso especial de diseño dirigido por los datos.

Diseños dirigidos por un intérprete

Los diseños dirigidos por un intérprete comprenden los intérpretes de reglas que ejecutan las reglas leídas de una fuente externa, intérpretes de script o lenguajes que leen y ejecutan programas, máquinas virtuales, motores de redes neuronales que ejecutan redes, motores lógicos de restricciones que leen y razonan con conjuntos de restricciones, etcétera. Este enfoque permite cambiar o parametrizar el comportamiento de un sistema por medio de expresiones lógicas externas. Se protege al sistema del impacto de las variaciones lógicas registrando externamente la lógica, leyéndola, y utilizando un intérprete.

Diseños reflexivos o de meta-nivel

Un ejemplo de este enfoque es utilizar el *java.beans.Introspector* para obtener un objeto *BeanInfo*, solicitar el objeto *Method* para el método de acceso (*getter*) para la propiedad del *bean X*, e invocar *Method.invoke*. Se protege al sistema del impacto de la lógica o variaciones externas del código mediante algoritmos reflexivos que utilizan introspección y servicios de metalenguaje. Podría considerarse un caso especial de diseño dirigido por los datos.

Acceso uniforme

Algunos lenguajes como Ada, Eiffel y C# soportan un constructor sintáctico de manera que se expresan de la misma manera el acceso a un método y a un campo. Por ejemplo, *unCírculo.radio* podría invocar al método *radio():float* o referirse directamente al campo público, dependiendo de la definición de la clase. Podemos cambiar de campos públicos a métodos de acceso, sin cambiar el código del cliente.

Principio de Sustitución de Liskov (PSL)

PSL [Liskov88] formaliza el principio de protección contra las variaciones en implementaciones diferentes de una interfaz, o una subclase que extiende una superclase.

Citando textualmente:

Lo que se quiere aquí es algo como la siguiente propiedad de sustitución: si para cada objeto *o1* de tipo *S* hay un objeto *o2* de tipo *T* tal que para todos los programas *P* definidos en términos de *T*, el comportamiento de *P* no varía cuando se sustituye *o1* por *o2* entonces *S* es un subtipo de *T* [Liskov88].

Informalmente, el software (métodos, clases,...) que hace referencia a un tipo *T* (alguna interfaz o superclase abstracta) debería trabajar correctamente o como se espera con cualquier implementación o subclase de *T* que la sustituya —llamémosla *S*—. Por ejemplo:

```
public void añadirImpuestos
    (IAdaptadorCalculadorDeImpuestos calculador, Venta venta)
{
    List líneasDeImpuestos = calculador.getImpuestos( venta );
    ...
}
```

Para este método *añadirImpuestos*, no importa la implementación de *IAdaptadorCalculadorDeImpuestos* que se pase como argumento real, el método debería continuar funcionando “como se espera”. PSL es una idea simple, intuitiva para la mayoría de los desarrolladores de software, que formaliza esta intuición.

Diseños de ocultación de la estructura

En la primera edición de este libro, se enunció un importante principio de diseño de objetos denominado **No Hable con Extraños** o la **Ley de Demeter** [Lieberherr88] como uno de los nueve patrones GRASP. Brevemente, significa evitar crear diseños que recorren largos caminos de la estructura de objetos y envía mensajes (o habla) con objetos distantes, indirectos (extraños). Tales diseños son frágiles con respecto a los cambios en las estructuras de los objetos —un punto frecuente de inestabilidad—. Pero en la segunda edición el principio más general VP sustituye a No Hable con Extraños, porque el segundo es un caso especial del primero. Es decir, un mecanismo para lograr proteger de los cambios en la estructura es aplicar las reglas de No Hable con Extraños.

No Hable con Extraños restringe los objetos a los que deberías enviar los mensajes dentro de un método. Establece que en un método, sólo deberían enviarse mensajes a los siguientes objetos:

1. El objeto *this* (o *self*).
2. Un parámetro del método.
3. Un atributo de *this*.
4. Un elemento de una colección que es un atributo de *this*.
5. Un objeto creado en el método.

La intención es evitar el acoplamiento de un cliente con conocimiento de objetos indirectos y las conexiones entre objetos.

Los objetos directos son “conocidos” del cliente, los objetos indirectos son “extraños”. Un cliente debería hablar con conocidos y evitar hablar con extraños.

A continuación presentamos un ejemplo que viola (ligeramente) el principio No Hables con Extraños. Los comentarios explican la violación.

```
class Registro
{
    private Venta venta;

    public void metodoAlgoFragil()
    {
        //venta.getPago() envía un mensaje a un "conocido" (según #3)
        //pero en venta.getPago().getCantidadEntregada()
        //el mensaje getCantidadEntregada() se aplica a un Pago "extraño"

        Dinero cantidad = venta.getPago().getCantidadEntregada();

        //...
    }
    //...
}
```

Este código recorre conexiones estructurales a partir de un objeto conocido (la *Venta*) a un objeto extraño (el *Pago*), y entonces le envía el mensaje. Es ligeramente frágil, ya que depende del hecho de que los objetos *Venta* se conecten a los objetos *Pago*. En realidad, no es probable que esto sea un problema.

Pero, considere el siguiente fragmento, que llega más lejos a lo largo del camino de la estructura:

```
public void metodoMasFragil()
{
    TitularCuenta titular =
        venta.getPago().getCuenta().getTitularCuenta();

    //...
}
```

El ejemplo es artificial, pero puede ilustrar el patrón: llegar más lejos a lo largo de un camino de conexiones de objetos para enviar un mensaje a un objeto distante e indirecto —hablando con un extraño alejado—. El diseño está acoplado a una estructura particular de cómo están conectados los objetos. Cuanto más lejos a lo largo de un camino recorre el programa, más frágil es.

Karl Lieberherr y sus colegas han investigado sobre buenos principios de diseño de objetos, en el marco del proyecto Demeter. Esta Ley de Demeter (No Hable con Extraños) se identificó debido a que vieron que con frecuencia se producían cambios e inestabilidad en la estructura de los objetos, y como resultado frecuentes roturas en el código que estaba acoplado con el conocimiento de las conexiones de los objetos.

Pero, como examinaremos en la siguiente sección “VP especulativo y escoja sus batallas”, no siempre es necesario proteger contra esto; depende de la inestabilidad de la estructura de los objetos. En las librerías estándar (como las librerías de Java) las conexiones estructurales entre las clases de los objetos son relativamente estables. En sistemas maduros, la estructura es más estable. En sistemas nuevos en las primeras iteraciones, no es estable.

En general, cuanto más lejos se recorre a lo largo de un camino, más frágil es y, por tanto, es más útil ser conforme con el principio No Hable con Extraños.

Si obedecemos estrictamente esta ley —protección contra las variaciones estructurales— es necesario que se añadan nuevas operaciones públicas a los “conocidos” de un objeto; estas operaciones proporcionan en último término la información deseada, y ocultan cómo se obtuvo. Por ejemplo, para soportar No Hable con Extraños en los dos casos anteriores:

```
//caso 1
Dinero cantidad = venta.getCantidadEntregadaEnPago();

//caso 2
TitularCuenta titular = venta.getTitularCuentaDelPago();
```

Contraindicaciones**Advertencia: VP especulativo y escoja sus batallas**

En primer lugar, merece la pena definir dos puntos de cambio:

- **Punto de variación:** variaciones en el sistema actual, existente o en los requisitos, como las múltiples interfaces de los calculadores de impuestos que se deben soportar.
- **Puntos de evolución:** puntos especulativos de variación que podrían aparecer en el futuro, pero que no están presentes en los requisitos actuales⁴.

VP se aplica tanto a los puntos de variación como de evolución

Una advertencia: algunas veces el coste de “futuras necesidades” especulativas en los puntos de evolución pesa más que el coste en el que se incurre por un diseño simple, más “frágil” que se revisa cuando sea necesario en respuesta a las verdaderas presiones de cambio. Es decir, el coste de diseñar la protección en los puntos de evolución puede ser más alto que rehacer un diseño simple.

Por ejemplo, recuerdo un sistema de manejo de mensajes del busca en el que el arquitecto incluyó un lenguaje de script y un intérprete para dar soporte a la flexibilidad y variaciones protegidas en un punto de evolución. Sin embargo, durante la revisión en una versión incremental, se eliminó el complejo (e ineficiente) script —simplemente no era necesario—. Y cuando empecé con la programación orientada a objetos (al principio de los ochenta) padecí la enfermedad de “generalizitis” en la que solía pasar muchas horas creando superclases de las clases que realmente necesitaba escribir. Yo haría todo muy general y flexible (y protegido contra las variaciones), para esas situaciones futuras en las que realmente sería beneficioso —lo que no sucedía nunca—. Era un pobre juez de cuando merecía la pena el esfuerzo.

La idea no es abogar por diseños frágiles que hay que rehacer. Si la necesidad de flexibilidad y protección de cambios es realista, entonces está motivada la aplicación de VP. Pero si es para futuras necesidades o supuesta “reutilización” con probabilidades muy inciertas, entonces se requiere un modo de ver las cosas restringido y crítico.

Los desarrolladores sin experiencia tienden a realizar diseños frágiles, los desarrolladores de nivel intermedio tienden a unos diseños excesivamente flexibles y generalizados (de unas formas que nunca se utilizan). Los diseñadores expertos eligen con perspicacia; quizás un diseño simple y frágil en el que existe un equilibrio entre el coste del cambio y su probabilidad.

Beneficios

- Se añaden fácilmente las extensiones que se necesitan para nuevas variaciones.
- Se pueden introducir nuevas implementaciones sin afectar a los clientes.
- Se reduce el acoplamiento.
- Puede disminuirse el impacto o coste de los cambios.

⁴ En el UP, los puntos de evolución se pueden documentar formalmente en **Casos de Cambio**; cada uno describe aspectos relevantes de un punto de evolución a beneficio de un futuro arquitecto.

Patrones y Principios Relacionados

- La mayoría de los patrones y principios de diseño son mecanismos para variaciones protegidas, entre los que se encuentran el polimorfismo, las interfaces, indicación, encapsulación de datos, la mayoría de los patrones de diseño GoF, etcétera.
- En [Pree95] los puntos de variación y evolución reciben el nombre de “puntos cálidos”.

También conocido como; Parecido a

VP es esencialmente lo mismo que los principios de ocultación de la información y de abierto-cerrado, que son términos más antiguos. Como patrón “oficial” en la comunidad de los patrones, fue Cockburn en [VCK96] quien lo denominó “Variaciones Protegidas” en 1996.

Ocultación de la Información

El famoso artículo de David Parnas *On the Criteria To Be Used in Decomposing Systems Into Modules* (Criterios a Utilizar para Descomponer los Sistemas en Módulos) [Parnas72] es un ejemplo de los clásicos que siempre se citan pero que rara vez se leen. En él, Parnas introduce el concepto de **ocultación de información**. Quizás debido a que el término suena como la idea de encapsulación de datos, se ha mal interpretado como eso, y algunos libros definen erróneamente los conceptos como sinónimos. Más bien, lo que Parnas quiso decir con la ocultación de información era ocultar información sobre los diseños a otros módulos, en los puntos de cambio difíciles o probables. Citando literalmente su explicación sobre la ocultación de información como principio de diseño conductor:

Proponemos en lugar de eso que uno comience con una lista de decisiones de diseño difíciles o decisiones de diseño que es probable que cambien. Cada módulo entonces se diseña para ocultar tal decisión a los otros.

Esto es, el principio de Parnas de ocultación de información es el mismo que se enuncia en VP, y no es simplemente encapsulación de datos —que no es más que una de las muchas técnicas para ocultar información acerca del diseño—. Sin embargo, el término ha sido tan ampliamente reinterpretado como sinónimo de encapsulación de datos que ya no es posible utilizarlo en el sentido original sin malentendidos.

Principio Abierto-Cerrado

El **Principio Abierto-Cerrado** (PAC), descrito por Bertrand Meyer en [Meyer88] es esencialmente equivalente al patrón VP y a la ocultación de información. Una definición del PAC es:

Los módulos deberían ser tanto abiertos (para extenderse; adaptables) como cerrados (el módulo está cerrado a las modificaciones que afecten a los clientes).

El PAC y las VP son esencialmente dos expresiones del mismo principio, que resaltan diferentes puntos: protección en los puntos de variación y evolución. En el PAC, los “módulos” comprenden todos los elementos de software discretos, como los métodos, clases, subsistemas, aplicaciones, etcétera.

En el contexto del PAC, la frase “cerrado con respecto a X” significa que los clientes no se ven afectados si X cambia. Por ejemplo, “la clase está cerrada con respecto a las definiciones de los campos de instancia” mediante el mecanismo de encapsulación de datos con campos privados y métodos de acceso públicos. Al mismo tiempo, están abiertas a la modificación de las definiciones de los datos privados, porque los clientes externos no están acoplados directamente a los datos privados.

Otro ejemplo podría ser, “los adaptadores de los calculadores de impuestos están cerrados con respecto a su interfaz pública” mediante la implementación de la interfaz estable *IAdaptadorCalculadorDeImpuestos*. Sin embargo, los adaptadores están abiertos a la extensión siendo modificados de manera privada en respuesta a los cambios en las APIs de los calculadores de impuestos externos, de manera que no afectan a sus clientes.

Capítulo 23

DISEÑO DE LAS REALIZACIONES DE CASOS DE USO CON LOS PATRONES DE DISEÑO GoF

Cualquier cosa que tú puedas hacer, yo puedo hacerla meta.

Daniel Dennett

Objetivos

- Aplicar los patrones de diseño GRASP y GoF al diseño del caso de estudio NuevaEra.
-

Introducción

Este capítulo estudia el diseño de objetos para las realizaciones de los casos de usos de la siguiente iteración del caso de estudio NuevaEra, que aborda el soporte para el acceso a los servicios externos de terceras partes cuyas interfaces podrían variar, de reglas más complejas para establecer el precio de los productos y de reglas del negocio conectables.

En el contexto de los problemas de diseño que se discutirán, también se introducirá nueva notación UML muy utilizada.

Lo importante es mostrar cómo aplicar los patrones GoF y los GRASP más básicos. Se pretende ilustrar que el diseño de objetos y la asignación de responsabilidades se puede explicar y aprender en base a la aplicación de los patrones —un vocabulario de principios y estilos que se pueden combinar para diseñar los objetos—.

Los patrones de la “pandilla de los cuatro” (*Gang-of-Four*)

Los patrones adicionales que se presentan en este capítulo proceden de *Design Patterns* [GHJV95], un libro básico y muy popular que presenta 23 patrones que son útiles durante el diseño de objetos. Puesto que el libro fue escrito por cuatro autores, estos patrones se conocen como los patrones de la “pandilla de los cuatro” o patrones “GoF”¹.

Este capítulo proporciona una introducción a algunos de los patrones GoF más utilizados; capítulos posteriores presentarán más². Se recomienda un estudio completo y cuidadoso del libro *Design Patterns* para iniciarse como diseñador de objetos, aunque el libro asume que el lector ya es un diseñador con alguna experiencia; este libro ofrece una introducción.

Un vocabulario compartido

Además del vocabulario visual de la notación UML, al final de este capítulo tendremos un vocabulario de diseño compartido más rico, en términos de nombres de patrones. Por tanto, será posible comunicar progresivamente ideas de diseño del software sobre todo mediante diagramas UML, con algunas notas conectadas que indican los patrones (Indirección, Estrategia,...) que se van a aplicar.

23.1. Adaptador (GoF)

El problema que se presentó en el capítulo anterior para motivar el patrón Polimorfismo, y su solución, es más concretamente un ejemplo del patrón GoF **Adaptador**.

Adaptador (<i>Adapter</i>)
<i>Contexto/Problema</i>
¿Cómo resolver interfaces incompatibles, o proporcionar una interfaz estable para componentes parecidos con diferentes interfaces?
<i>Solución</i>
Convierta la interfaz original de un componente en otra interfaz, mediante un objeto adaptador intermedio.

Recordemos que: el sistema de PDV NuevaEra necesita soportar diferentes tipos de servicios externos de terceras partes, entre los que se encuentran los calculadores de impuestos, servicios de autorización de pagos, sistemas de inventario, y sistemas de contabilidad, entre otros. Cada uno tiene un API diferente, que no puede ser cambiada.

Una solución es añadir un nivel de indirección con objetos que adapten las distintas interfaces externas a una interfaz consistente que es el que se utiliza en la aplicación. La solución se ilustra en la Figura 23.1.

¹ Con una referencia tangencial a la política China.

² En la práctica, quizás unos 15 de estos 23 patrones son los que se utilizan con frecuencia.

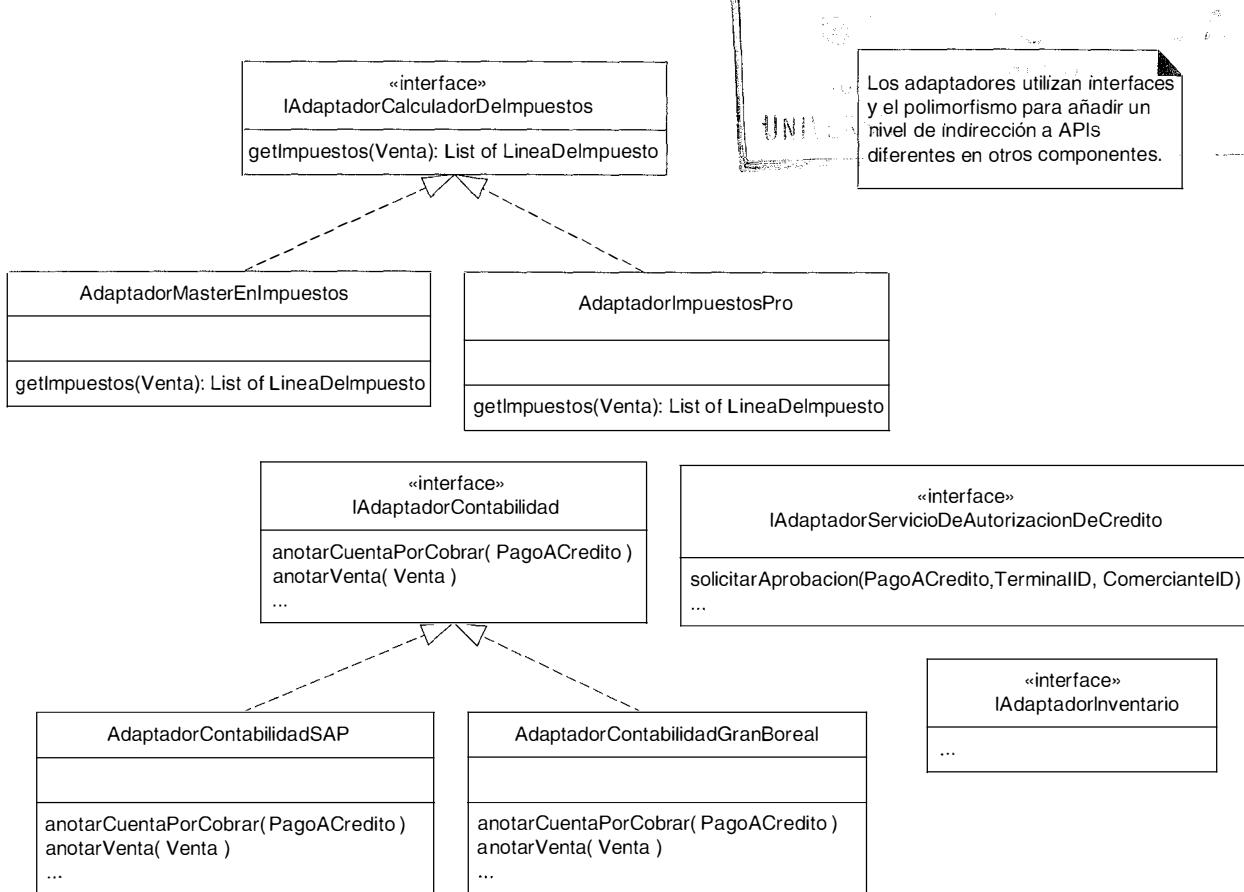


Figura 23.1. El patrón Adaptador.

Como se ilustra en la Figura 23.2, una instancia de un adaptador particular se instanciará para el servicio externo elegido³, como SAP para contabilidad, y adaptará la solicitud *anotarVenta* a la interfaz externa, tal como una interfaz SOAP XML sobre HTTPS para un servicio Web de intranet ofrecido por SAP.

Notación UML: Obsérvese en la Figura 23.2 el uso de un “piruli” de interfaz para indicar que la instancia *AdaptadorContabilidadSAP* implementa un interfaz relevante.

Polimorfismo, Indirección y Variaciones Protegidas (GRASP)

La aplicación anterior del patrón Adaptador es una especialización de los componentes básicos GRASP. Ofrece Variaciones Protegidas de los cambios en las interfaces externas o paquetes de terceras partes mediante el uso de un objeto de Indirección que aplica interfaces y Polimorfismo.

³ En la Arquitectura de Conexión de J2EE, estos adaptadores a los servicios externos se denominan de manera más específica **adaptadores de recursos**.

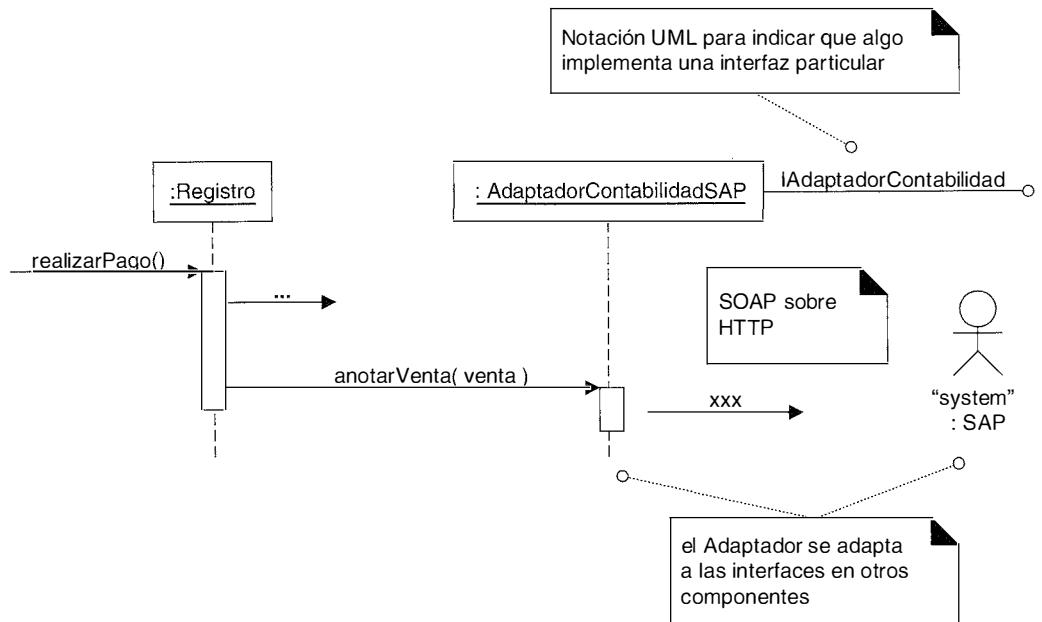


Figura 23.2. Utilización de un adaptador.

Nótese que se pueden analizar patrones mucho más complejos y especializados en función de la familia GRASP básica. Existen cientos de patrones de diseño publicados, y aunque es útil estudiarlos para acelerar el aprendizaje, entender sus conceptos básicos subyacentes (Variaciones Protegidas, Bajo Acoplamiento, Polimorfismo, Indirección...) nos ayudan a abrirnos camino a través de los innumerables detalles y ver el “alfabeto” esencial de las técnicas de diseño que se van a aplicar.

Convención de nombres: ¿incluimos el nombre del patrón en el nombre del tipo?

Nótese que los nombres de los tipos incluyen el nombre del patrón “Adaptador”. Éste es un estilo relativamente común y tiene la ventaja de poder comunicar fácilmente a los que leen el código o los diagramas el patrón de diseño que se va a utilizar.

23.2. Descubrimientos del “análisis” durante el diseño: Modelo del Dominio

Observe que en el diseño del Adaptador en la Figura 23.1, la operación `getImpuestos` devuelve una lista de `LíneaDeImpuesto`. Es decir, al reflexionar e investigar en profundidad el modo de gestionar los impuestos y cómo funcionan los calculadores de impuestos, el diseñador (yo) se dio cuenta de que se asocia una lista de líneas de impuestos con una venta, como impuesto del gobierno central, impuesto del gobierno regional, etcétera (¡siempre que exista la oportunidad los gobiernos inventarán nuevos impuestos!).

Además de ser una clase software recién creada en el Modelo de Diseño, se trata de un concepto del dominio. Es normal y común descubrir conceptos del dominio relevantes y refinar el conocimiento de los requisitos durante el diseño o la programación —el desarrollo iterativo apoya esta clase de descubrimiento incremental—.

¿Debería reflejarse este descubrimiento en el Modelo del Dominio (o Glosario)? Si el Modelo del Dominio se utilizará en el futuro como fuente de inspiración para el trabajo de diseño posterior, o como ayuda visual para comunicar los conceptos claves del dominio, entonces podría ser útil añadirlo. La Figura 23.3 presenta un Modelo del Dominio actualizado.

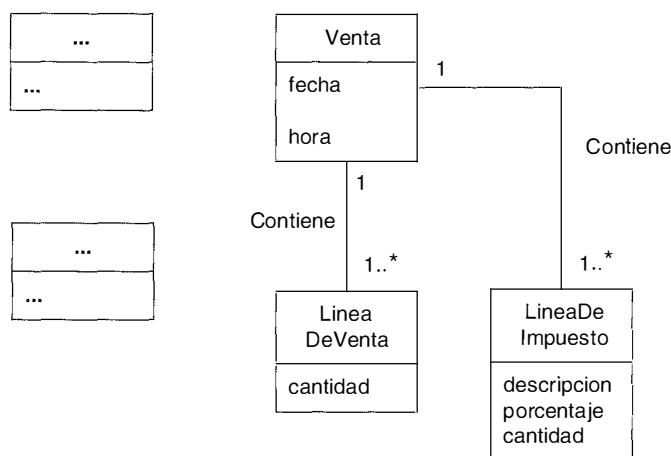


Figura 23.3. Modelo del Dominio parcial actualizado.

¿Mantenemos el Modelo del Dominio?

Puntualizando el comentario anterior sobre la actualización del Modelo del Dominio: observe que la arquitectura del Modelo de Diseño normalmente se organizará en capas (lo que se discutirá con más detalle en un capítulo posterior). Una de estas capas de clases de diseño se denominará **capa del dominio**; esta capa contendrá las clases software cuyos nombres y estructuras se inspiraron en el vocabulario y los conceptos del dominio (*Venta*, *LineaDeImpuesto*, etcétera).

Sugerencia

Después de algunas iteraciones, el Modelo del Dominio —como primera fuente de inspiración para las clases del diseño en la capa del dominio del Modelo de Diseño— podría dejar de tener utilidad. Si la actualización del Modelo del Dominio para reflejar los cambios en el Modelo de Diseño deja de tener un valor práctico, piense en eliminarlo.

Más bien, únicamente lleve a cabo un trabajo de ingeniería inversa (con una herramienta CASE de UML) para generar un diagrama de clases de la capa del dominio a partir de las clases de diseño del Modelo de Diseño. Aunque son clases software en lugar de clases conceptuales del dominio puras, reflejan el vocabulario del dominio más relevante que ha surgido en el diseño del software y, por tanto, un diagrama de clases UML de las clases del diseño en la capa del dominio del Modelo de Diseño puede ser una “aproximación” útil al verdadero Modelo del Dominio.

Por favor, no lo mal interprete: no sugerimos que se elimine definitivamente un Modelo del Dominio, sino más bien que considere si merece la pena mantenerlo, o es simplemente documentación para generar trabajo, y que conozca qué alternativas pueden ser útiles.

Patrones Relacionados

Un adaptador de recursos que oculta un sistema externo podría también ser considerado un objeto Fachada (otro patrón GoF que se expondrá en este capítulo), puesto que envuelve el acceso al subsistema o sistema con un único objeto (que es la esencia de la Fachada). Sin embargo, el nombre de adaptador está motivado especialmente cuando el objeto que envuelve facilita la adaptación a diversas interfaces externas.

23.3. Factoría (GoF)

El adaptador da lugar a un nuevo problema en el diseño: en la solución del patrón Adaptador anterior para servicios externos con diversas interfaces, ¿quién crea el adaptador? ¿Y cómo determinar qué clase de adaptador crear, como *AdaptadorMasterENImpuestos* o *AdaptadorImpuestosPro*?

Si los crea algún objeto del dominio, las responsabilidades de los objetos del dominio exceden la lógica pura de la aplicación (como el cálculo del total de la venta) y entran en otras cuestiones relacionadas con la conexión con componentes software externos.

Este punto subraya otro principio de diseño fundamental (normalmente considerado un principio de diseño de arquitectura): diseñe para mantener una **separación de intereses** (*separation of concerns*). Es decir, divida en módulos o separe intereses distintos en áreas diferentes, de manera que cada una tenga un propósito cohesivo. Por ejemplo, la capa del dominio de los objetos software destaca las responsabilidades centradas relativamente en la lógica de la aplicación, mientras que un grupo diferente de objetos es responsable de las cuestiones de conectividad con sistemas externos.

Por tanto, la elección de un objeto del dominio (como el *Registro*) para crear los adaptadores no está de acuerdo con el objetivo de separación de intereses, y disminuye su cohesión.

Notación UML: Observe el estilo del diagrama UML de la Figura 23.4 que incluye una nota que muestra el pseudocódigo detallado de *getAdaptadorCalculadorDeImpuestos*. Este estilo nos permite incluir detalles dinámicos sobre algoritmos en los diagramas de clases estáticos de manera que podría disminuir la necesidad de utilizar diagramas de interacción.

Una alternativa típica en este caso es aplicar el patrón **Factoría** (o **Factoría Concreta**), en el que se define un objeto Fabricación Pura “factoría” para crear los objetos.

Los objetos factoría tienen varias ventajas:

- Separan la responsabilidad de la creación compleja en objetos de apoyo (*helper*) cohesivos.
- Ocultan la lógica de creación potencialmente compleja.
- Permiten introducir estrategias para mejorar el rendimiento de la gestión de la memoria, como objetos caché o de reciclaje.

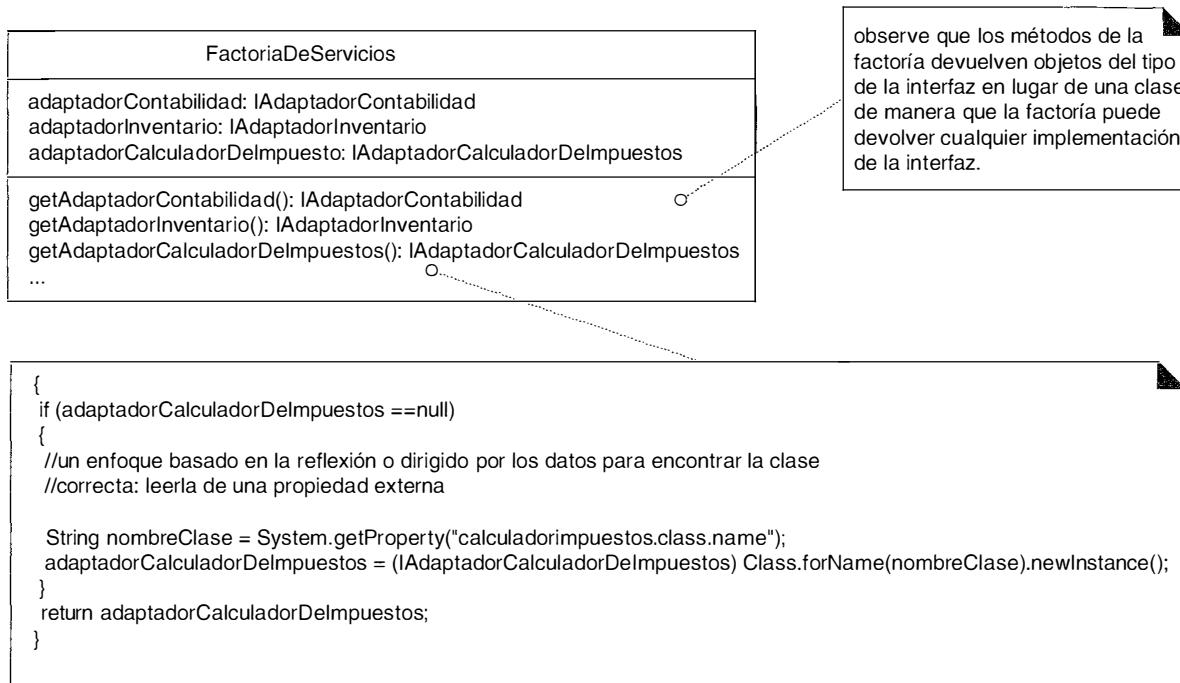
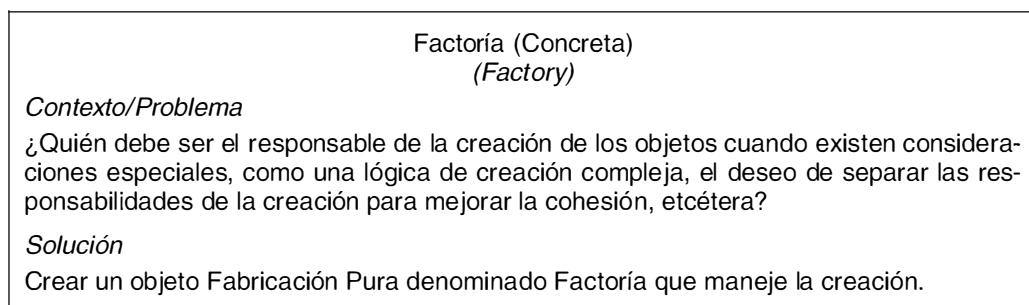


Figura 23.4. El patrón Factoría.



La solución de la Factoría se ilustra en la Figura 23.4.

Nótese que en la *FactoriaDeServicios*, la lógica para decidir qué clase se crea se resuelve leyendo el nombre de la clase desde una fuente externa (por ejemplo, por medio de una propiedad del sistema, si se utiliza Java) y después se carga la clase dinámicamente. Éste es un ejemplo de **diseño dirigido por los datos** parcial. El diseño consigue Variaciones Protegidas con respecto a los cambios en la clase de implementación del adaptador. Sin cambiar el código fuente en la clase factoría, podemos crear instancias de nuevas clases adaptador cambiando el valor de la propiedad y asegurando que la nueva clase es visible en la ruta de clases de Java para que pueda cargarse.

Patrones Relacionados

A menudo se accede a las factorías con el patrón Singleton⁴.

⁴ N. del T.: Al igual que para otros patrones, no hemos traducido el nombre de este patrón ya que a diferencia de otros (como Factoría o Adaptador) normalmente se utiliza el nombre original en inglés tanto en la expresión oral como escrita en castellano.

23.4. Singleton (GoF)

Con la *FactoriaDeServicios* surge un nuevo problema de diseño: ¿quién crea la propia factoría y cómo se accede?

En primer lugar, observe que sólo se necesita una instancia de la factoría en el proceso. Segundo, una rápida reflexión sugiere que los métodos de esta factoría podrían necesitar que se invoquen desde varios sitios del código, puesto que en diferentes lugares se necesita acceder a los adaptadores para solicitar los servicios externos. Por tanto, existe un problema de visibilidad: ¿cómo conseguir la visibilidad a esta única instancia de *FactoriaDeServicios*?

Una solución es pasar la instancia de *FactoriaDeServicios* como parámetro donde quiera que se descubra que se necesita que sea visible, o inicializar los objetos que necesitan que la factoría sea visible, con una referencia permanente. Esto es posible pero no es conveniente; una alternativa es el patrón **Singleton**.

Ocasionalmente, es conveniente mantener visibilidad global o un único punto de acceso a una única instancia de una clase, en lugar de cualquier otra forma de visibilidad. Esto se cumple para la instancia de *FactoriaDeServicios*.

Singleton

Contexto/Problema

Se admite exactamente una instancia de una clase —es un “singleton”—. Los objetos necesitan un único punto de acceso global.

Solución

Defina un método estático de la clase que devuelva el singleton.

Por ejemplo, la Figura 23.5 muestra una implementación del patrón Singleton.

Por tanto, la idea clave es que la clase X defina un método estático *getInstancia* que él mismo proporciona una única instancia de X.

Con este enfoque, un desarrollador tiene visibilidad global a esta instancia única, por medio del método estático de la clase *getInstancia*, como vemos en este ejemplo:

```
public class Registro
{
    public void inicializar()
    {
        ... hace algún trabajo ...
    }

    // acceso a la Factoría Singleton mediante la llamada a
    getInstancia adaptadorContabilidad =
        FactoriaDeServicios.getInstancia().getAdaptadorContabilidad();
```

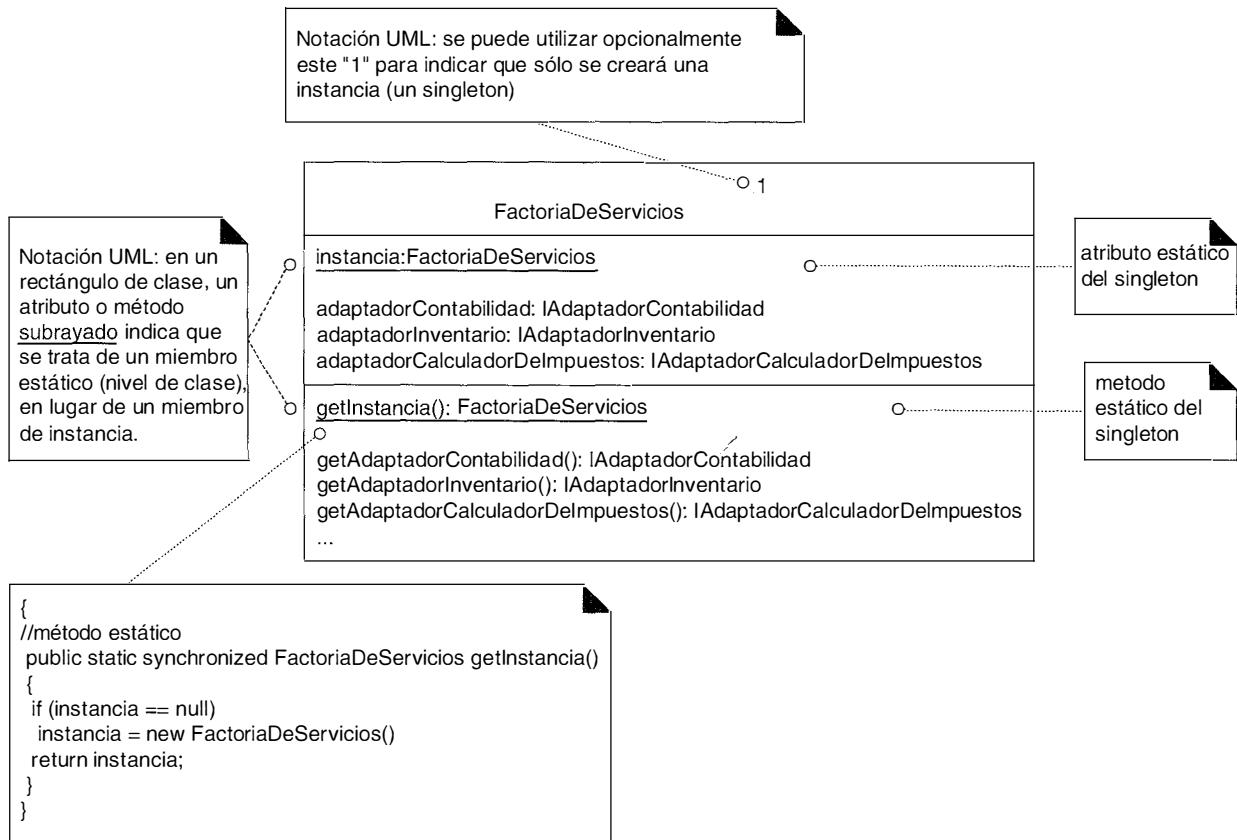


Figura 23.5. El patrón Singleton en la clase *FactoriaDeServicios*.

```
... hace algún trabajo ...
}

// otros métodos...
}
```

Puesto que la visibilidad de las clases públicas tiene un alcance global (en la mayoría de los lenguajes), en cualquier punto del código, en cualquier método de cualquier clase, uno puede escribir `ClaseSingleton.getInstancia()` para obtener la visibilidad de la instancia del singleton, y entonces enviarle un mensaje, como `ClaseSingleton.getInstancia().hacerFoo()`. Es difícil superar la emoción de ser capaces de *hacerFoo* globalmente.

Abreviatura UML para el acceso a un Singleton en los diagramas de interacción

La notación UML que implica —pero no muestra explícitamente— el mensaje `getInstancia` en un diagrama de interacción es añadir el estereotipo «`singleton`» a la instancia, como en la Figura 23.6. Este enfoque evita tener que mostrar explícitamente el mensaje (sin interés) `getInstancia` a la clase antes de enviar el mensaje a la instancia singleton.

Cuestiones de diseño e implementación

A menudo, se invoca con frecuencia al método *getInstancia* del Singleton. En aplicaciones con varios hilos de ejecución, la etapa de creación de la lógica de **inicialización perezosa** (*lazy*) es una sección crítica que requiere el control de concurrencia del hilo. Por tanto, asumiendo que la instancia se inicializa de manera perezosa, es habitual envolver el método con control de concurrencia. En Java, por ejemplo:

```
public static synchronized FactoriaDeServicios getInstance()
{
    if ( instancia == null )
    {
        //sección crítica si es una aplicación con varios hilos
        instancia = new FactoriaDeServicios();
    }
    return instancia;
}
```

A propósito de la inicialización perezosa, ¿no es preferible una **inicialización impaciente** (*eager*), como en este ejemplo?

```
public class FactoriaDeServicios
{
    //inicialización impaciente
    private static FactoriaDeServicios instancia =
        new FactoriaDeServicios();

    public static FactoriaDeServicios getInstance()
    {
        return instancia;
    }

    //otros métodos...
}
```

Es preferible el primer enfoque de inicialización perezosa al menos por estas razones:

- Se evita el trabajo de creación (y quizás retener recursos “caros”) si nunca se accede a la instancia.
- La inicialización perezosa de *getInstance* algunas veces contiene lógica de creación compleja y condicional.

Otra pregunta típica de la implementación del Singleton es: ¿por qué no hacer que todos los métodos de los servicios sean métodos *estáticos* de la propia clase, en lugar de utilizar un objeto instancia con métodos de instancia? Por ejemplo, qué pasa si añadimos un método *estático* denominado *getAdaptadorContabilidad* a la *FactoriaDeServicios*. Pero normalmente es preferible utilizar una instancia y los métodos de instancia por estos motivos:

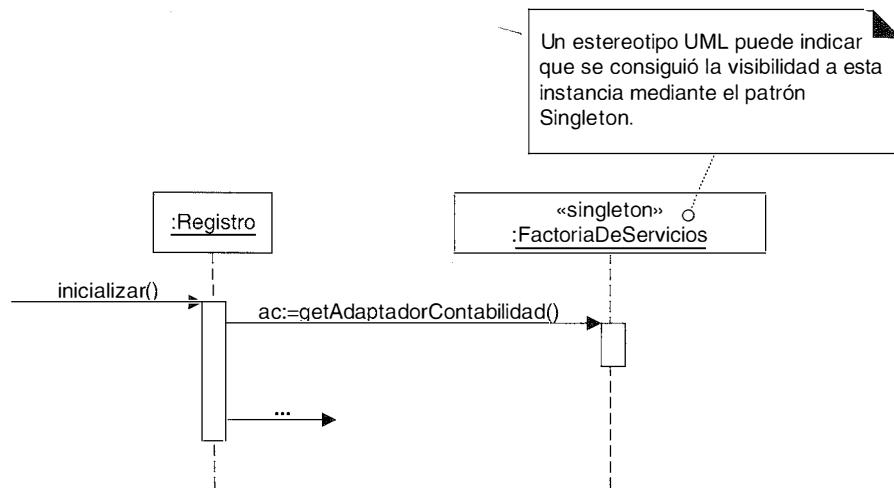


Figura 23.6. Mensaje implícito del patrón Singleton *getInstance* que se indica en UML con un estereotipo.

- Los métodos de instancia permiten que se definan subclases y se refinen la clase singleton; los métodos estáticos no son polimórficos (virtual) y en la mayoría de los lenguajes (excepto Smalltalk) no se pueden redefinir en las subclases.
- La mayoría de los mecanismos de comunicación remota orientada a objetos (por ejemplo el RMI de Java) sólo soportan la activación remota de los métodos de instancia, no de los métodos estáticos. Se podría activar de manera remota una instancia singleton, aunque es verdad que raramente se hace.
- Una clase no siempre es un singleton en todos los contextos de aplicación. En la aplicación X, podría ser un singleton, pero podría ser una “multi-tonelada” en la aplicación Y. También es habitual comenzar un diseño pensando que será un singleton y luego descubrir que se necesitan múltiples instancias en el mismo proceso. Por tanto, la solución basada en la instancia es más flexible.

Patrones Relacionados

El patrón Singleton se utiliza a menudo para los objetos Factoría y Fachada —otro patrón GoF que se presentará—.

23.5. Conclusiones del problema de los servicios externos con diversas interfaces

Se ha utilizado una combinación de los patrones Adaptador, Factoría y Singleton para proporcionar Variaciones Protegidas contra las diversas interfaces de los calculadores de impuestos externos, sistemas de contabilidad, etcétera. La Figura 23.7 ilustra un contexto más amplio de la utilización de los patrones en la realización del caso de uso.

Este diseño podría no ser ideal, y siempre se puede mejorar. Pero uno de los objetivos que intenta conseguir este caso de estudio es demostrar que se puede construir por lo menos un diseño a partir de un conjunto de principios o patrones que son los “componentes básicos”, y que existe un enfoque metódico para llevar a cabo y explicar un di-

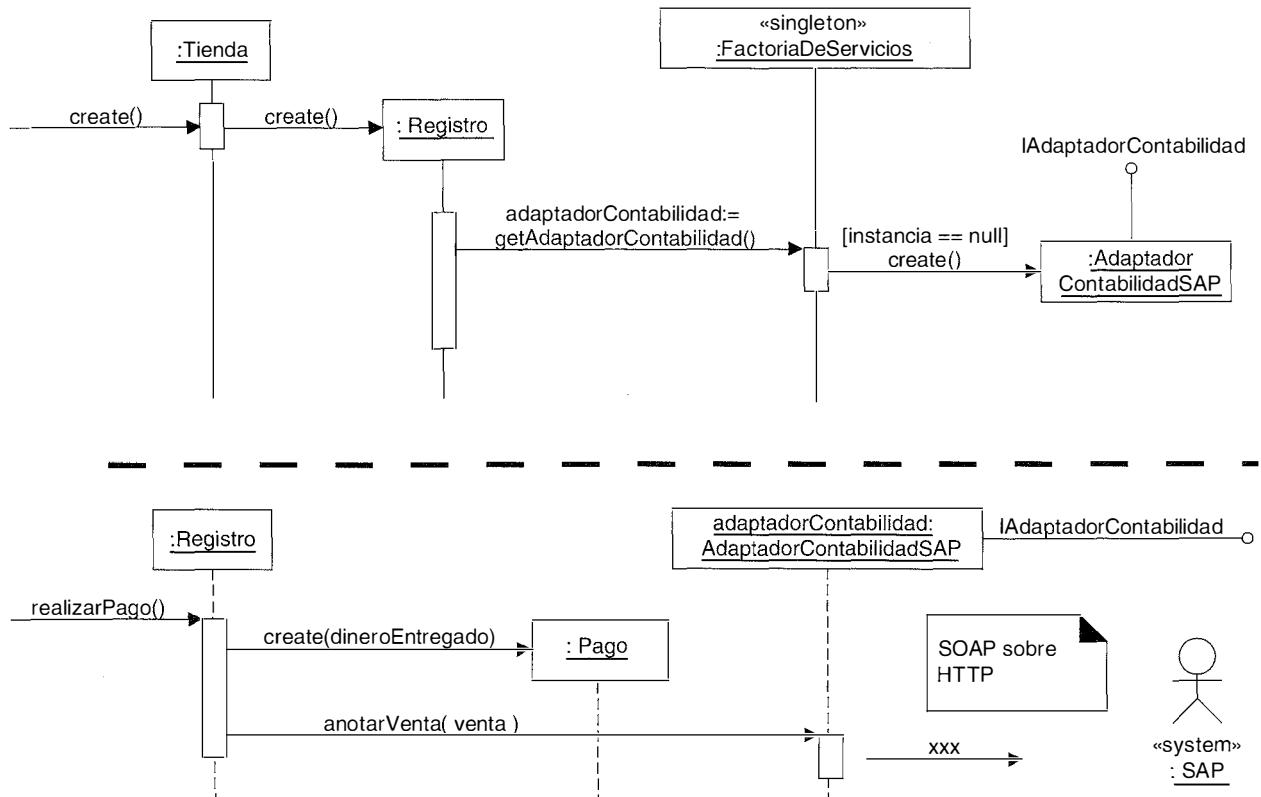


Figura 23.7. Patrones Adaptador, Factoría y Singleton aplicados al diseño.

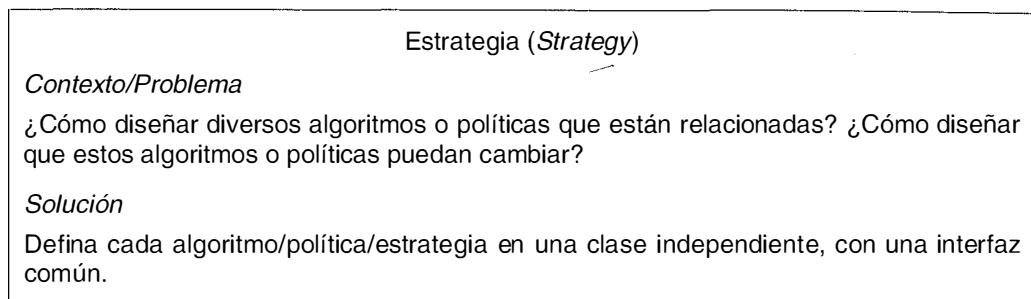
seño. Espero sinceramente que se pueda ver cómo surge el diseño de la Figura 23.7 razonando en base al Controlador, Creador, Variaciones Protegidas, Bajo Acoplamiento, Alta Cohesión, Indirección, Polimorfismo, Adaptador, Factoría y Singleton.

Nótese lo conciso que puede ser un diseñador en una conversación o en la documentación cuando hay un conocimiento común de los patrones. Puedo decir, “Para resolver el problema de tener diversas interfaces para los servicios externos, podemos utilizar Adaptadores generados desde una Factoría Singleton”. En realidad, los diseñadores de objetos tienen conversaciones de este tipo; utilizando los patrones y los nombres de los patrones se eleva el nivel de abstracción en la comunicación del diseño.

23.6. Estrategia (GoF)

El siguiente problema de diseño que se va a resolver es proporcionar una lógica más compleja para fijar los precios, como descuentos para un día en toda la tienda, descuentos para las personas mayores, etcétera.

La estrategia de fijación de precios (que podría llamarse también regla, política o algoritmo) de una venta puede variar. Durante un periodo podría ser el 10% de descuento en todas las ventas, después podría ser un descuento de 10 € si el total de la venta es superior a 200 €, y muchas otras variaciones. ¿Cómo diseñamos los diversos algoritmos de fijación de precios?



Puesto que el comportamiento de la fijación de precios varía según la estrategia (o algoritmo), creamos múltiples clases *EstrategiaFijarPreciosVenta*, cada una con un método polimórfico *getTotal* (ver Figura 23.8). A cada método *getTotal* se le pasa como parámetro el objeto *Venta*, de manera que el objeto de la estrategia de fijación de precios puede encontrar el precio anterior al descuento de la *Venta*, para aplicar después la regla de descuento. La implementación de cada *getTotal* será diferente: la *EstrategiaFijarPreciosPorcentajeDescuento*, aplicará el descuento de acuerdo a un porcentaje, y así sucesivamente.

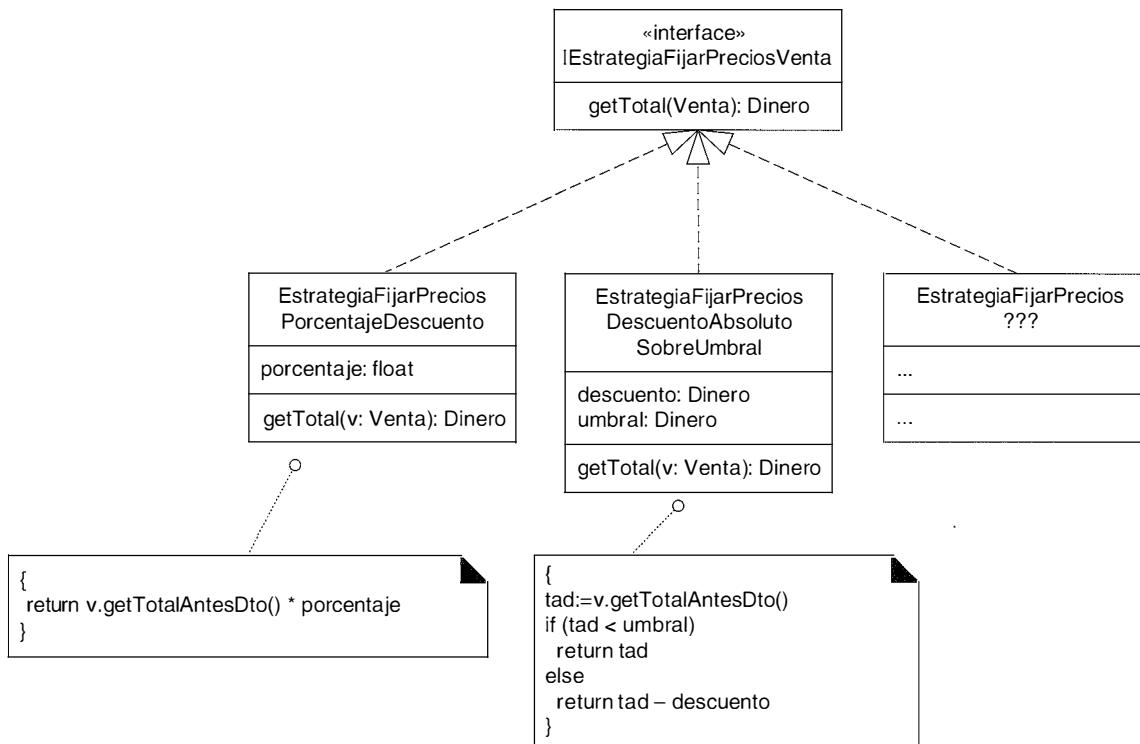


Figura 23.8. Clases para la Estrategia de fijación de precios.

Un objeto estrategia se conecta a un **objeto de contexto** —el objeto al que se aplica el algoritmo—. En este ejemplo, el objeto de contexto es una *Venta*. Cuando se envía el mensaje *getTotal* a la *Venta*, delega parte del trabajo a su objeto estrategia, como se ilustra en la Figura 23.9. No es necesario que el mensaje que se envía al objeto de contexto

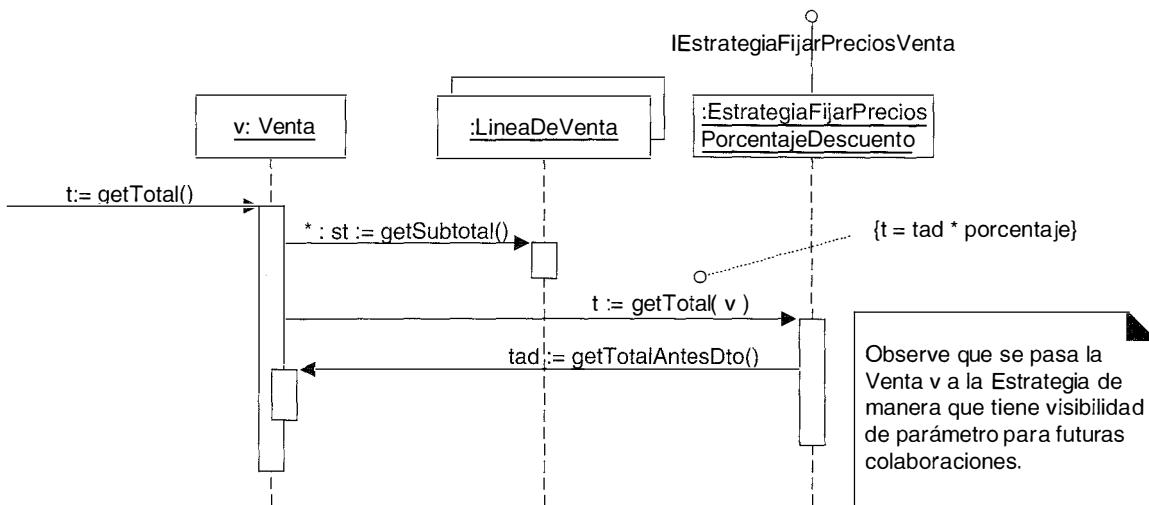


Figura 23.9. Estrategia en colaboración.

y al objeto estrategia tengan el mismo nombre, como en este ejemplo (por ejemplo, *getTotal* y *getTotal*), aunque es lo normal. Sin embargo, es habitual —de hecho, normalmente necesario— que el objeto de contexto pase una referencia a él mismo (*this*) al objeto estrategia, de manera que la estrategia tenga visibilidad de parámetro del objeto contexto, para futuras colaboraciones.

Obsérvese que el objeto de contexto (*Venta*) necesita tener visibilidad de atributo de su estrategia. Esto se refleja en el DCD de la Figura 23.10.

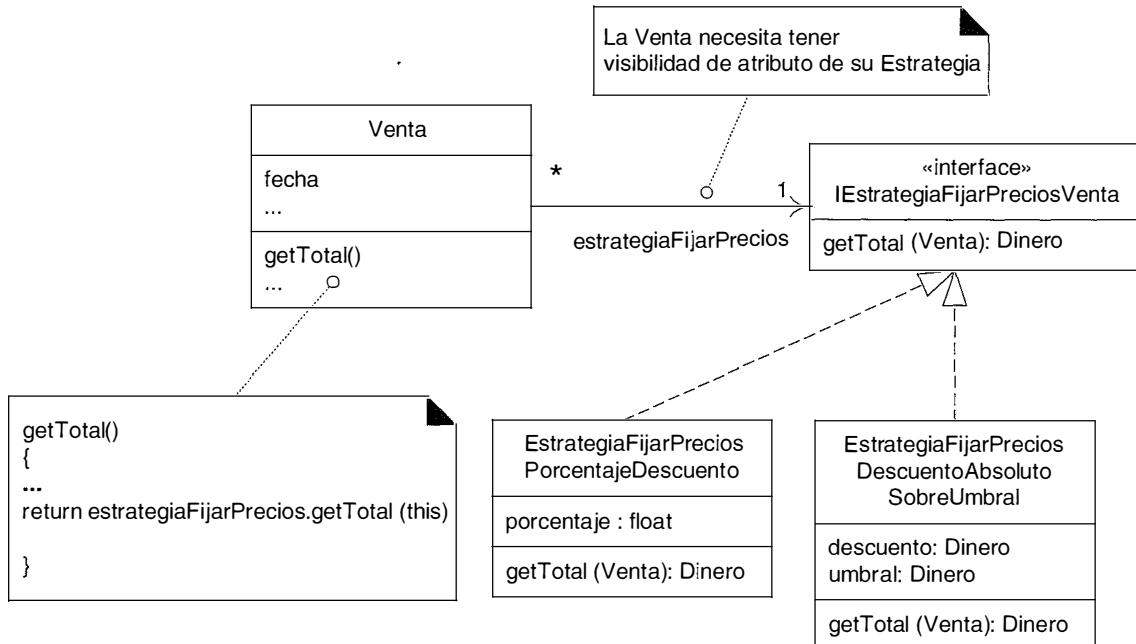


Figura 23.10. Los objetos de contexto necesitan tener visibilidad de atributo de su estrategia.

Creación de una Estrategia con una Factoría

Existen diferentes algoritmos o estrategias de fijación de precios, y cambian con el tiempo. ¿Quién debería crear la estrategia? Un enfoque directo es aplicar de nuevo el patrón Factoría: una *FactoriaDeEstrategiasFijarPrecios* puede ser la responsable de crear todas las estrategias (todos los algoritmos o las políticas conectables o cambiantes) que se necesitan en la aplicación. Como con la *FactoriaDeServicios*, se puede leer el nombre de la clase de implementación de la estrategia de fijación de precios como una propiedad del sistema (o alguna fuente de datos externa), y después crear una instancia. Con este *diseño dirigido por los datos* parcial (o diseño reflexivo) uno puede cambiar dinámicamente en cualquier momento —mientras se está ejecutando la aplicación del PDV NuevaEra— la política de fijación de precios, especificando la creación de una clase de Estrategia diferente.

Obsérvese que se utiliza una nueva factoría para las estrategias; es decir, diferente a la *FactoriaDeServicios*. Esto se ajusta al objetivo de Alta Cohesión —cada factoría está centrada de manera cohesiva en la creación de una familia de objetos relacionados—.

Notación UML: Observe que en la Figura 23.10 se referencia mediante una asociación directa a la interfaz *IESTRATEGIAFIJARPRECIOSVENTA*, no a una clase concreta. Eso indica que el atributo de referencia en la *Venta* se declarará en términos de la interfaz, no de una clase, de manera que el atributo se puede ligar con cualquier implementación de la interfaz.

Nótese que debido a que la política de fijación de precios cambia con frecuencia (podría ser cada hora), *no* es conveniente almacenar la instancia de la estrategia creada en un campo de la *FactoriaDeEstrategiasFijarPrecios*, sino crear una cada vez, leyendo la propiedad externa para obtener el nombre de la clase e instanciando luego la estrategia.

Y como con la mayoría de las factorías, la *FactoriaDeEstrategiasFijarPrecios* será un singleton (una instancia) y se accederá mediante el patrón Singleton (ver Figura 23.11).

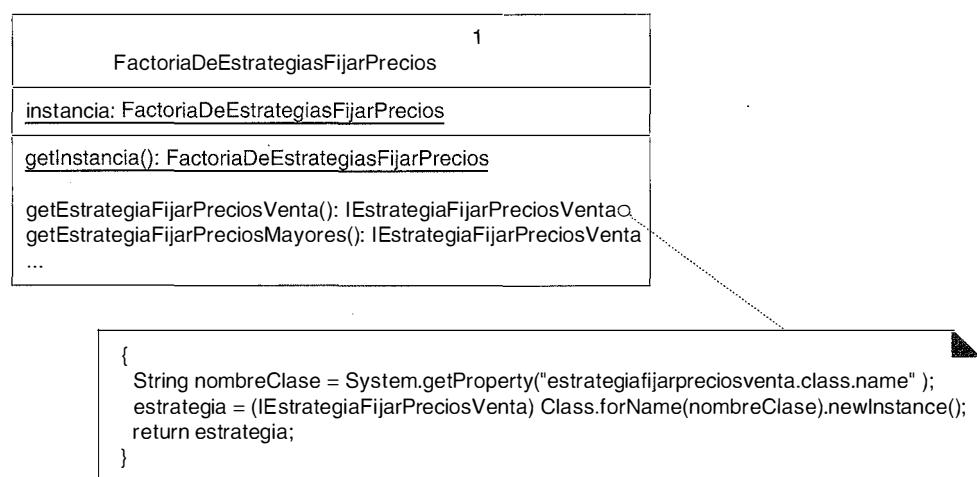


Figura 23.11. Factoría de estrategias.

Cuando se crea una instancia de la *Venta*, puede pedir a la factoría su estrategia de fijación de precios, como se muestra en la Figura 23.12.

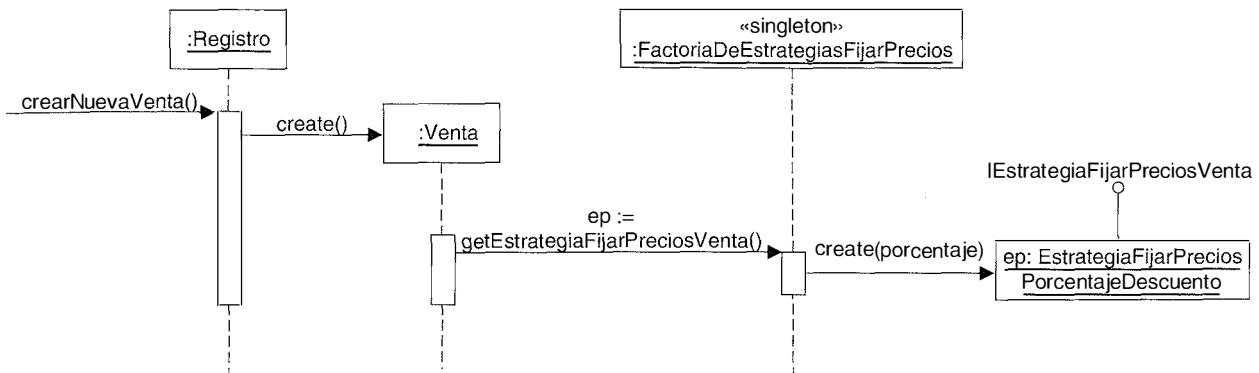


Figura 23.12. Creación de una estrategia.

Lectura e inicialización del valor del porcentaje

Finalmente, un problema de diseño que se ha ignorado hasta ahora es cómo encontrar los diferentes valores de los porcentajes o descuentos absolutos. Por ejemplo, el valor del porcentaje de la *EstrategiaFijarPreciosPorcentajeDescuento* podría ser 10% el lunes, pero 20% el martes.

Nótese también que el porcentaje de descuento podría estar relacionado con el tipo del comprador, como una persona mayor, en lugar de con un periodo de tiempo.

Estos números se almacenarán en algún almacén de datos externo, como una base de datos relacional, por lo que se pueden cambiar fácilmente. Luego, ¿qué objeto los leerá y asegurará que se asignan a la estrategia? Una opción razonable es la propia *FactoriaDeEstrategias*, puesto que crea la estrategia para fijar precios, y puede saber cuál es el porcentaje que hay que leer del almacén de datos (“el descuento actual del almacén”, “el descuento de las personas mayores”, etcétera).

Los diseños para la lectura de estos números procedentes de los almacenamientos de datos externos, varían desde los más simples a los más complejos, como una simple llamada SQL JDBC (por ejemplo, si se utilizan las tecnologías Java) o la colaboración con objetos que añaden niveles de indirección para ocultar la localización concreta, el lenguaje de consulta de los datos, o el tipo de almacén de datos. El análisis de los puntos de variación y evolución con respecto al almacén de datos revelará si es necesario que se proteja contra las variaciones. Por ejemplo, podríamos preguntar, “¿Estamos todos cómodos con el compromiso a largo plazo de utilizar una base de datos relacional que entienda SQL?”. Si es así, podría ser suficiente una simple llamada JDBC desde la *FactoriaDeEstrategias*.

Resumen

Con los patrones Estrategia y Factoría se ha conseguido Variaciones Protegidas con respecto a las políticas para fijar precios que varían dinámicamente. La Estrategia se fundamenta en el Polimorfismo y las interfaces para permitir algoritmos conectables en un diseño de objetos.

Patrones Relacionados

El patrón Estrategia se basa en el Polimorfismo, y proporciona Variaciones Protegidas con respecto a los algoritmos que cambian. Las Estrategias normalmente se crean mediante una Factoría.

23.7. Composite⁵ (GoF) y otros principios de diseño

Todavía surge otro interesante problema en los requisitos y el diseño: ¿cómo gestionamos el caso de varias políticas contradictorias de fijación de precios? Por ejemplo, suponga que una tienda tiene hoy (lunes) en vigor las siguientes políticas:

- Política de descuento del 20% a las personas mayores.
- Descuento del 15% en compras superiores a los 400 € para clientes preferentes.
- Los lunes, hay un 50% de descuento en las compras superiores a 500 €.
- Comprando una caja de té Darjeeling, obtiene el 15% de descuento en todo.

Suponga que una persona mayor que también es cliente preferente compra una caja de té Darjeeling, y 600 € de hamburguesas vegetales (claramente un vegetariano entusiasta al que le encanta el chai). ¿Qué política de fijación de precios se debería aplicar?

Explicándolo más claramente: ahora las estrategias de fijación de precios que se conectan a la venta dependen de tres factores:

- Periodo de tiempo (lunes).
- Tipo de cliente (persona mayor).
- Un producto concreto (té Darjeeling).

Otro punto que hay que aclarar: tres de las cuatro políticas del ejemplo son en realidad simplemente estrategias de “porcentaje de descuento”, lo que simplifica nuestra perspectiva del problema.

Parte de la respuesta a este problema requiere que se defina la **estrategia de resolución de conflictos** de la tienda. Normalmente, una tienda aplica “lo mejor para el cliente” (el precio más bajo) como estrategia de resolución de conflictos, pero no es obligatorio y podría cambiar. Por ejemplo, durante un periodo con dificultades financieras, la tienda podría verse obligada a utilizar la estrategia de resolución de conflictos “el precio más alto”.

El primer punto a tener en cuenta es que pueden existir múltiples estrategias coexistiendo al mismo tiempo, es decir, una venta podría tener asociadas varias estrategias para fijar el precio. Otro punto a destacar es que una estrategia de fijación de precios puede estar relacionada con el tipo de cliente (por ejemplo, una persona mayor). Esto afecta al diseño de la creación: la *FactoriaDeEstrategias* debe conocer el tipo del cliente en el momento de la creación de una estrategia de fijación de precios para el cliente.

Análogamente, una estrategia de fijación de precios puede estar relacionada con el tipo de producto que se compra (por ejemplo, té Darjeeling). Esto, del mismo modo, tiene implicaciones en el diseño de la creación: la *FactoriaDeEstrategias* debe conocer la *EspecificacionDelProducto* en el momento de la creación de una estrategia de fijación de precios influenciada por el producto.

⁵ N. del T.: No se ha traducido por el mismo motivo que en el caso del patrón Singleton.

¿Hay alguna forma de cambiar el diseño de manera que el objeto *Venta* no conozca si está tratando con una o más estrategias, y ofrecer también un diseño para la resolución de conflictos? Sí, con el patrón Composite.

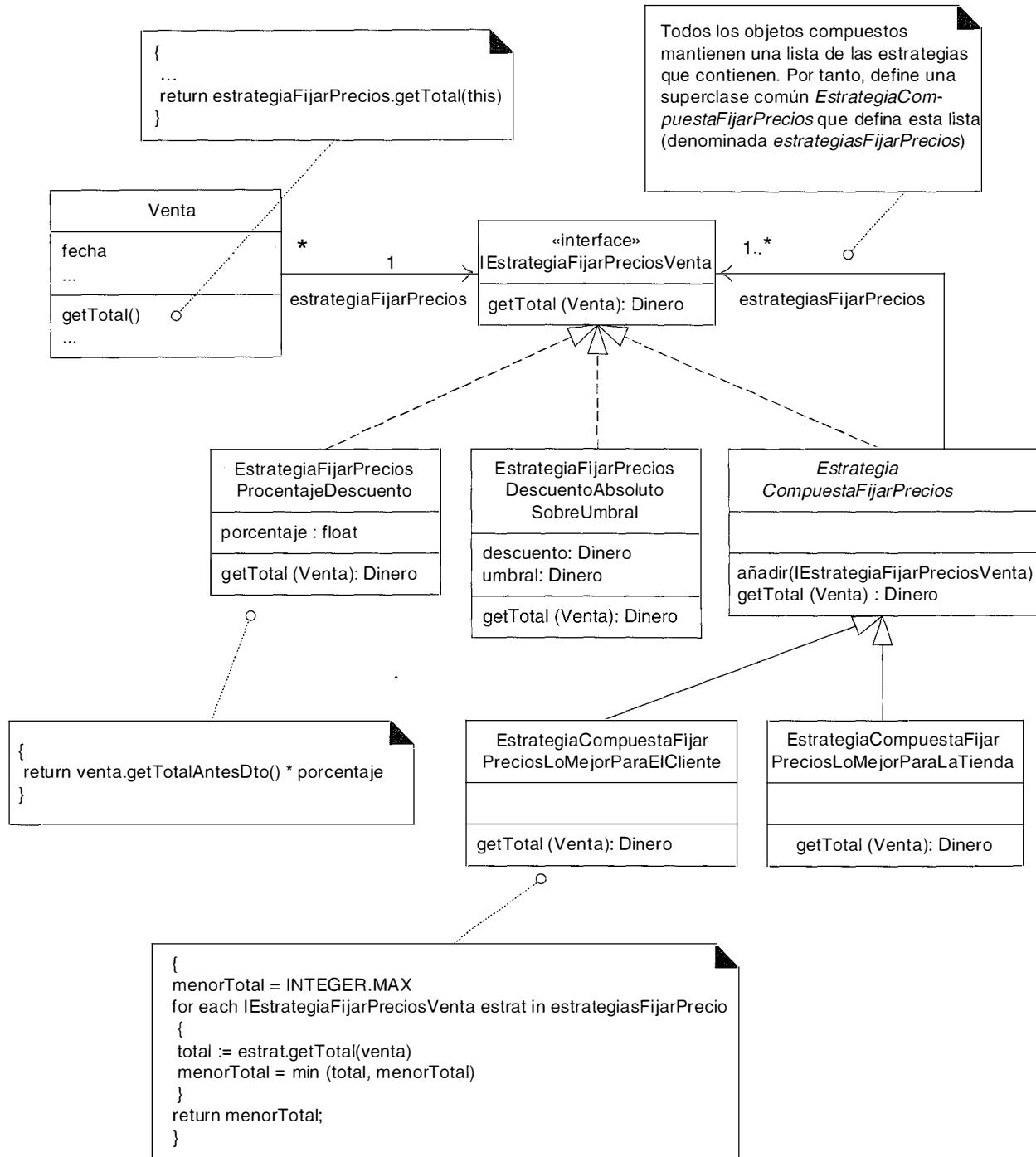
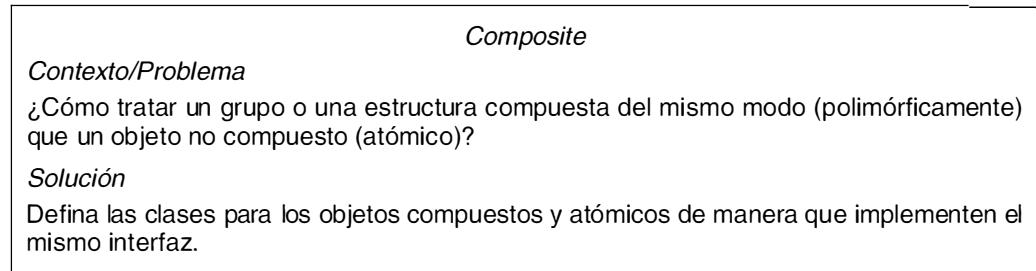


Figura 23.13. El patrón Composite.



Por ejemplo, una nueva clase denominada *EstrategiaCompuestaFijarPreciosLoMejorParaElCliente* (bueno, por lo menos es descriptivo) puede implementar la *IESTRATEGIAFIJARPRECIOSVENTA* y ella misma contiene otros objetos *IESTRATEGIAFIJARPRECIOSVENTA*. La Figura 23.13 explica en detalle la idea de diseño.

Observe que en este diseño, la clase compuesta como *EstrategiaCompuesta-FijarPreciosLoMejorParaElCliente* hereda el atributo *estrategiasFijarPrecios* que contiene una lista de más objetos *IESTRATEGIAFIJARPRECIOSVENTA*. Ésta es una característica distintiva de un objeto compuesto: el objeto compuesto externo contiene una lista de los objetos internos, y tanto los objetos externos como los internos implementan la misma interfaz. Es decir, la propia clase compuesta implementa la interfaz *IESTRATEGIAFIJARPRECIOSVENTA*.

Por tanto, podemos conectar al objeto *Venta* tanto a un objeto *EstrategiaCompuestaFijarPreciosLoMejorParaElCliente* (que contiene otras estrategias dentro de él) o a un objeto atómico *EstrategiaFijarPreciosPorcentajeDescuento*, y la *Venta* no conoce o no se preocupa si su estrategia de fijación de precios es atómica o compuesta —parecen iguales para el objeto *Venta*—. Es simplemente otro objeto que implementa la interfaz *IESTRATEGIAFIJARPRECIOSVENTA* y entiende el mensaje *getTotal* (Figura 23.14).

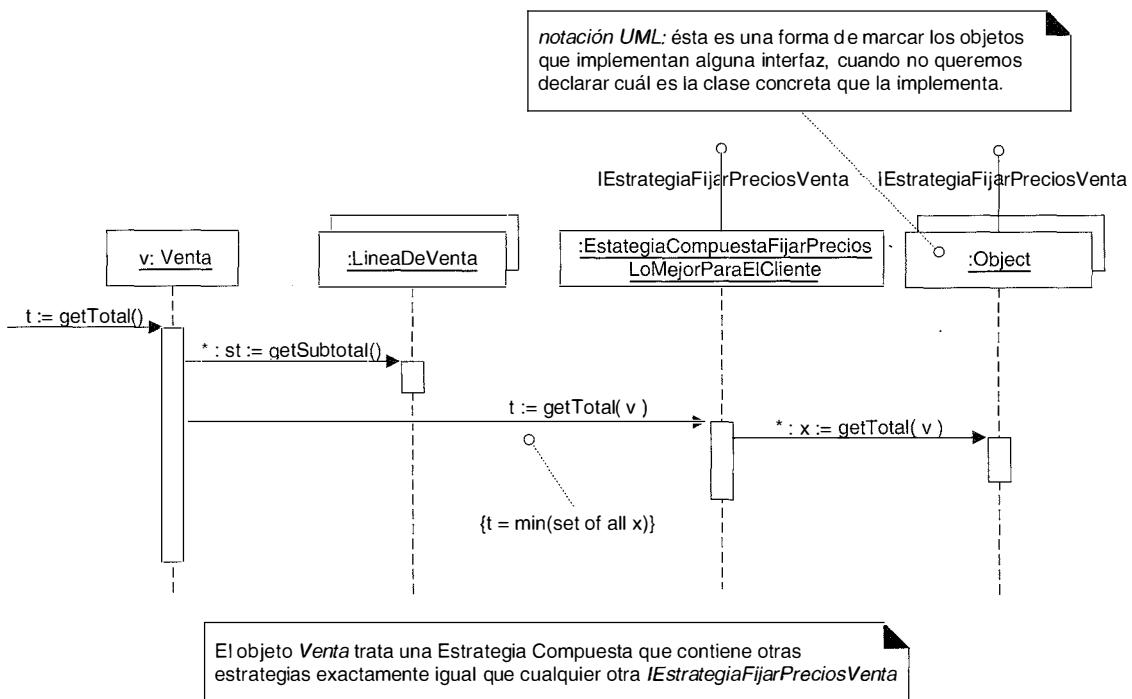


Figura 23.14. Colaboración con un Composite.

Notación UML: En la Figura 23.14, nótese, por favor, el modo de indicar los objetos que implementan una interfaz, cuando no nos preocupa especificar la clase de implementación exacta. Simplemente especificando la clase de implementación como *Object* queremos decir “sin comentarios” acerca de la clase específica. Ésta es una necesidad habitual en la elaboración de diagramas.

A continuación definimos la *EstrategiaCompuestaFijarPrecios* y una de sus subclases para aclarar la explicación con una muestra de código en Java:

```
//superclase luego todas las subclases heredan una List de estrategias

public abstract class EstrategiaCompuestaFijarPrecios
    implements IEstrategiaFijarPreciosVenta
{
    protected List estrategiasFijarPrecios = new ArrayList();

    public añadir(IEstrategiaFijarPreciosVenta e)
    {
        estrategiasFijarPrecios.add(e);
    }

    public abstract Dinero getTotal( Venta v );
}

//final de la clase

//Una Estrategia Compuesta que devuelve el total más pequeño
//de sus instancias EstrategiaFijarPreciosVenta internas

public class EstrategiaCompuestaFijarPreciosLoMejorParaElCliente
    extends EstrategiaCompuestaFijarPrecios
{
    public Dinero getTotal(Venta venta)
    {
        Dinero menorTotal = new Dinero (Integer.MAX_VALUE);

        //iteramos sobre todas las estrategias internas

        for( Iterator i = estrategiasFijarPrecios.iterator(); i.hasNext(); )
        {
            IEstrategiaFijarPreciosVenta estrategia =
                (IEstrategiaFijarPreciosVenta)i.next();
            Dinero total = estrategia.getTotal(venta);
            menorTotal = total.min(menorTotal);
        }
        return menorTotal();
    }

} //final de la clase
```

Notación UML: la Figura 23.13 introduce nueva notación UML para representar las jerarquías de clases y la herencia, que se explica en la Figura 23.15.

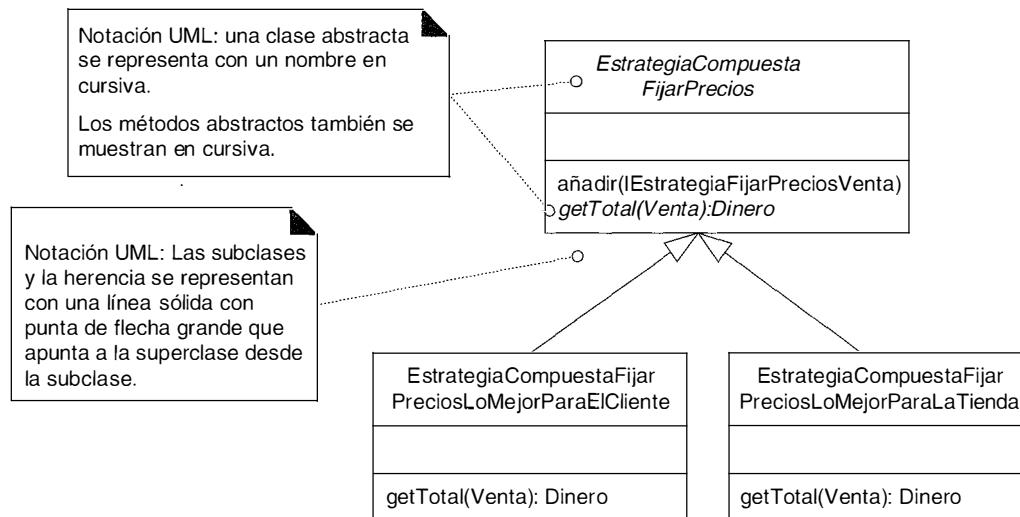


Figura 23.15. Superclases abstractas, métodos abstractos y herencia en UML.

Creación de múltiples instancias *EstrategiaFijarPreciosVenta*

Con el patrón Composite, hemos creado un grupo de diferentes (y contradictorias) estrategias de fijación de precios, que para *Venta* aparecen como una única estrategia. El objeto compuesto que contiene el grupo también implementa la interfaz *IEstrategiaFijarPreciosVenta*. La parte más desafiante (e interesante) de este problema de diseño es: ¿cuándo creamos estas estrategias?

Un diseño deseable comenzará creando un Composite que contenga las políticas de descuento de la tienda en el momento actual (que podría ser 0% de descuento si no hay ninguna activa), como alguna *EstrategiaFijarPreciosPorcentajeDescuento*. Entonces, si en un paso posterior del escenario, se descubre que se debe aplicar otra estrategia para fijar precios (como el descuento a las personas mayores), se añadirá fácilmente al objeto compuesto utilizando el método heredado *EstrategiaCompuestaFijarPrecios.add*.

Existen tres puntos en el escenario donde se podrían agregar al objeto compuesto las estrategias de fijación de precios:

1. Descuento actual a nivel de tienda, se añade cuando se crea la venta.
2. Descuento por el tipo de cliente, se añade cuando se informa al PDV del tipo de cliente.
3. Descuento según el tipo de producto (si compra el té Darjeeling obtendrá un descuento del 15% sobre el total de la venta), se añade cuando se introduce el producto en la venta.

El diseño del primer caso se muestra en la Figura 23.16. Como en el diseño original que se discutió anteriormente, el nombre de la clase estrategia que se va a instanciar se podría leer como una propiedad del sistema, y el valor del porcentaje se podría leer desde un almacenamiento de datos externo.

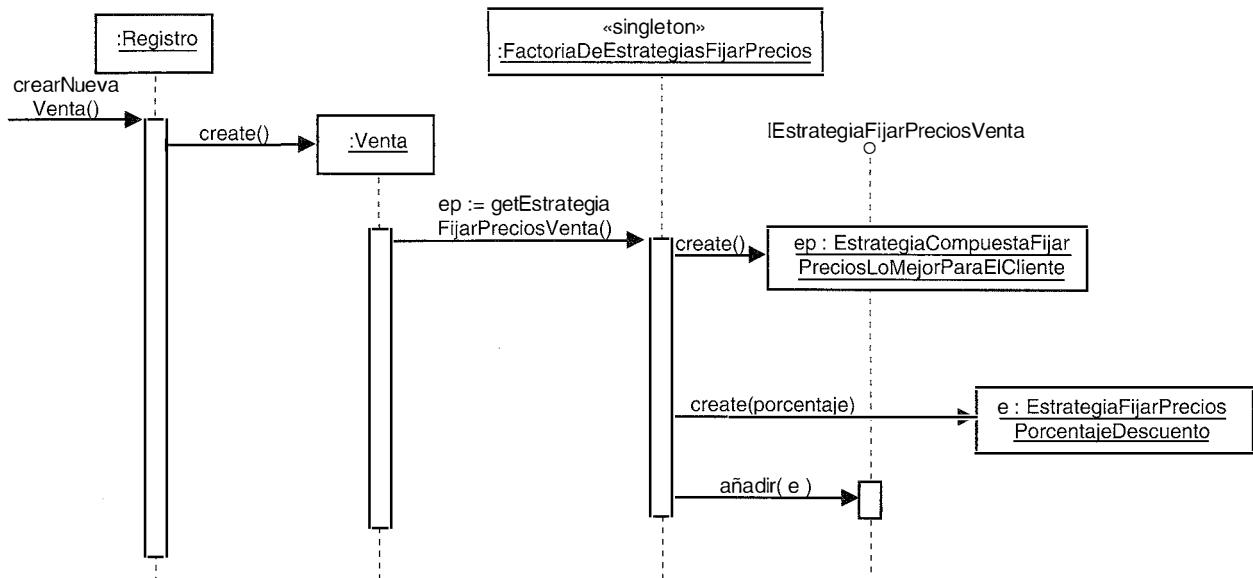


Figura 23.16. Creación de una estrategia compuesta.

Para el segundo caso de descuento según el tipo de cliente, en primer lugar recordemos la extensión del caso de uso que identificó anteriormente este requisito:

Caso de uso UC1: Procesar Venta

Extensiones (o Flujos Alternativos):

- 5b. El Cliente dice que le son aplicables descuentos (ej. empleado, cliente preferente):
1. El Cajero señala la petición de descuento.
 2. El Cajero introduce la identificación del Cliente.
 3. El Sistema presenta el descuento total, basado en las reglas de descuento.

Esto indica una nueva operación del sistema en el sistema de PDV, además de *crearNuevaVenta*, *introducirArticulo*, *finalizarVenta* y *realizarPago*. Llamaremos a esta quinta operación del sistema *introducirClienteParaDescuento*; opcionalmente podría tener lugar después de la operación *finalizarVenta*. Esto implica que se deberá introducir algún tipo de identificación del cliente a través de la interfaz de usuario, el *clienteID*. Quizás podría capturarse mediante un lector de tarjetas o a través del teclado.

El diseño de este segundo caso se muestra en las Figuras 23.17 y 23.18. No sorprende que el objeto factoría sea el responsable de la creación de la estrategia de fijación de precios adicional. Podría crear otra *EstrategiaFijarPreciosPorcentajeDescuento* que represente, por ejemplo, un descuento para las personas mayores. Pero como con el di-

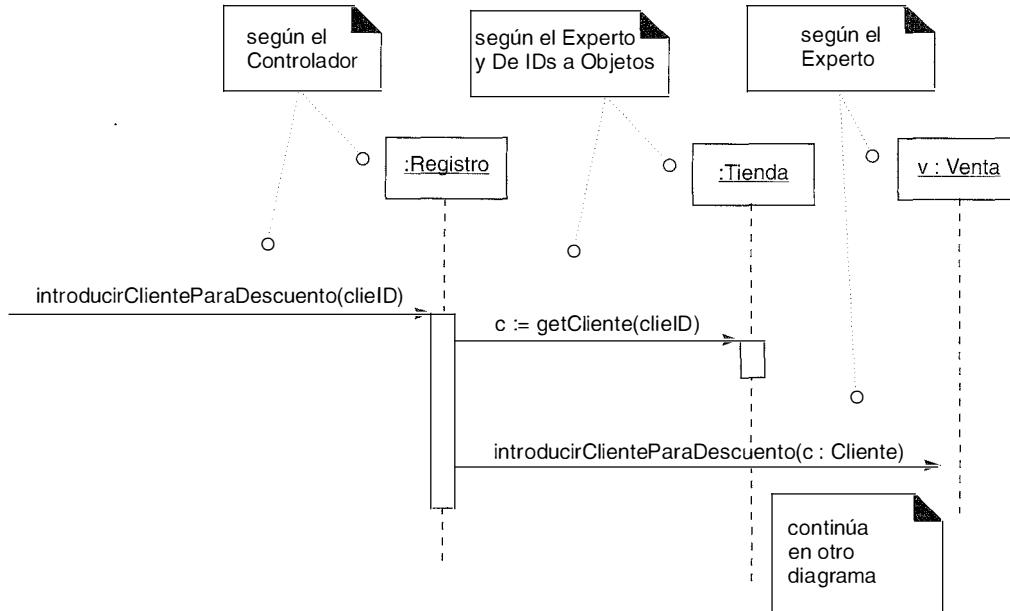


Figura 23.17. Creación de la estrategia de fijación de precios para el descuento de los clientes, parte 1.

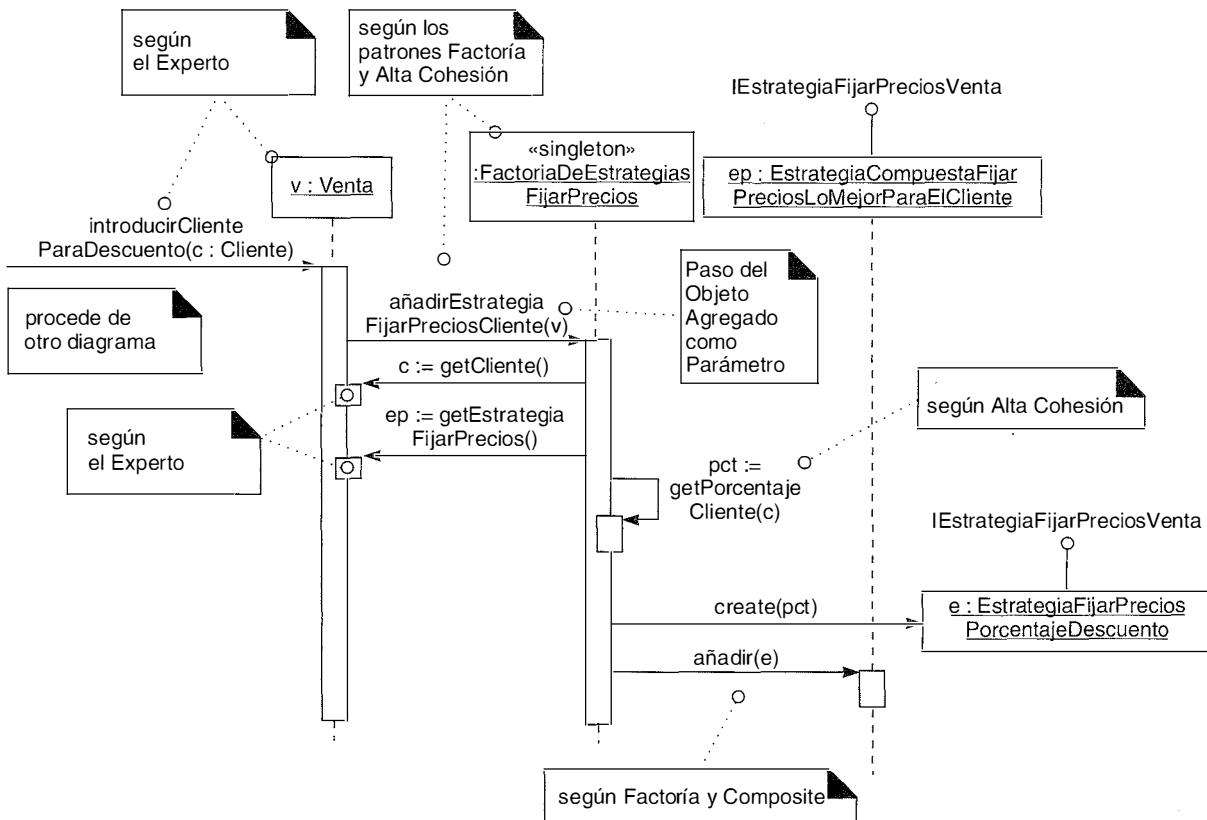


Figura 23.18. Creación de la estrategia de fijación de precios para el descuento de los clientes, parte 2.

seño de la creación original, la elección de la clase se leerá como una propiedad del sistema, al igual que el porcentaje específico para el tipo de cliente, para proporcionar Variaciones Protegidas con respecto a los cambios de las clases o los valores. Nótese que en virtud del patrón Composite, la *Venta* podría tener conectadas dos o tres estrategias contradictorias de fijación de precios, pero sigue pareciendo una única estrategia para el objeto *Venta*.

Notación UML: Las Figuras 23.17 y 23.18 muestran una importante idea UML sobre los diagramas de interacción: la división de un diagrama en dos, para que cada uno de ellos sea más legible.

Consideración de los patrones GRASP y otros principios en el diseño

Revisemos el razonamiento en función de algunos de los patrones GRASP básicos: para este segundo caso, ¿por qué no envía el *Registro* un mensaje a la *FactoriaDeEstrategiasFijarPrecios*, para crear esta nueva estrategia de fijación de precios y pasársela entonces a la *Venta*? Uno de los motivos es para mantener Bajo Acoplamiento. La *Venta* ya está acoplada a la factoría; si hacemos que el *Registro* también colabore con ella, incrementaríamos el acoplamiento en el diseño. Además, la *Venta* es el Experto en Información que conoce su estrategia actual de fijación de precios (que se va a modificar); luego, según el Experto, está justificado que se delegue en la *Venta*.

Obsérvese en el diseño que el *clienteID* se transforma en un objeto *Cliente*, preguntando el *Registro* a la *Tienda* por un *Cliente*, dado un ID. En primer lugar, se puede justificar que se otorgue a la *Tienda* la responsabilidad *getCliente*; de acuerdo con el Experto en Información y el objetivo de salto en la representación reducido, la *Tienda* puede conocer a todos los *Clientes*. Y el *Registro* pregunta a la *Tienda*, porque el *Registro* ya tiene visibilidad de atributo de la *Tienda* (a partir del trabajo de diseño previo); si la *Venta* tuviera que preguntar a la *Tienda*, la *Venta* necesitaría una referencia a la *Tienda*, lo que incrementaría el acoplamiento más allá de los niveles actuales y, por tanto, no mantendría Bajo Acoplamiento.

Transformación de IDs en objetos

Segundo, ¿por qué transformar *clienteID* (un “ID” —quizás un número—) en un objeto *Cliente*? Esta es una práctica habitual del diseño de objetos —transformar claves e identificadores (IDs) de las cosas en verdaderos objetos—. A menudo esta transformación tiene lugar poco después de que se introduzca el ID o la clave en la capa del dominio del Modelo de Diseño desde la capa de UI. No tiene nombre de patrón, pero podría ser un candidato puesto que es un estilo habitual entre los diseñadores de objetos con experiencia —quizás *De IDs a Objetos*—. ¿Por qué preocuparse? Porque tener un auténtico objeto *Cliente* que encapsula un conjunto de información sobre el cliente, y que puede tener comportamiento (relacionado con el Experto en Información, por ejemplo), con frecuencia se vuelve beneficioso y flexible cuando crece el diseño, incluso si el diseñador no creyó que fuera necesario un verdadero objeto y pensó en cambio que un simple número sería suficiente. Nótese que en el diseño inicial, la transformación del *articuloID* en un objeto *EspecificacionDelProducto* es otro ejemplo del patrón *De IDs a Objetos*.

Paso de objetos agregados como parámetros

Finalmente, observe que en el mensaje *añadirEstrategiaFijarPrecioCliente(v:Venta)* pasamos una *Venta* a la factoría, y entonces la factoría se vuelve y pregunta a la *Venta* por el *Cliente* y su *EstrategiaFijarPrecios*.

¿Por qué no extraer exactamente estos dos objetos de la *Venta*, y, en lugar de lo anterior, pasar a la factoría el *Cliente* y la *EstrategiaFijarPrecios*? La respuesta es otro estilo del diseño de objetos típico: evitar extraer los objetos hijos de un objeto padre o agregado, y entonces pasar los objetos hijos. Mejor, pase el objeto agregado que contiene los hijos.

Si se sigue este principio se incrementa la flexibilidad, porque entonces la factoría puede colaborar con la *Venta* completa de algunas maneras que no habíamos anticipado que fueran necesarias previamente (que es muy normal), y como corolario, reduce la necesidad de anticipar lo que necesita el objeto factoría; el diseñador pasa exactamente la *Venta* completa, sin conocer los objetos más específicos que podría necesitar la factoría. Aunque este estilo no tiene nombre, está relacionado con el Bajo Acoplamiento y Variaciones Protegidas. Quizás podría llamarse patrón *Paso de Objetos Agregados como Parámetro*.

Resumen

De este problema de diseño se han extraído muchos consejos de diseño de objetos. Un diseñador de objetos experto ha incorporado a su memoria muchos de estos patrones estudiando las explicaciones que se han publicado, y ha asimilado los principios fundamentales, como los que se describen en la familia GRASP.

Por favor, obsérvese que aunque esta explicación del Composite se ha realizado para una familia de Estrategias, el patrón Composite se puede aplicar a otros tipos de objetos, no sólo estrategias. Por ejemplo, es habitual que se creen “macro commands” —commands que contienen otros commands— mediante el uso del Composite. El patrón Command se describirá en un capítulo posterior.

Ejercicios Propuestos

Ejercicio 1

La compra de un producto específico da lugar a un nuevo descuento en la venta completa. Por ejemplo, si compró té Darjeeling, el 15% de descuento en el total de la venta.

Ejercicio 2

Todas las políticas de fijación de precios que se han considerado hasta el momento se aplican sobre el total de la venta, se denominan algunas veces descuentos a nivel de transacción. Pero el reto de diseño más interesante es gestionar los descuentos de las líneas de venta. Por ejemplo:

- Comprando dos trajes, obtenga uno gratis.
- Comprando tres ordenadores X, obtenga un descuento del 50% en la impresora Y.

¿Hay una forma elegante de diseñar esto con los objetos Estrategia?

Patrones Relacionados

El patrón Composite se utiliza normalmente junto con los patrones Estrategia y *Command*. El Composite se basa en el Polimorfismo y proporciona Variaciones Protegidas a los clientes de manera que no les afecta si el objeto con el que se relacionan es atómico o compuesto.

23.8. Fachada (GoF)

Otro requisito que se ha elegido para esta iteración es dar soporte a *reglas de negocio conectables*. Es decir, en puntos predecibles de los escenarios, como cuando tienen lugar *crearNuevaVenta* e *introducirArticulo* en el caso de uso *Procesar Venta*, o cuando un cajero comienza a vender, a distintos clientes que deseen comprar el PDV NuevaEra les gustaría adaptar ligeramente su comportamiento.

Siendo más precisos, asuma que se desea que las reglas puedan invalidar una acción. Por ejemplo:

- Suponga que cuando se crea una nueva venta, es posible identificar que se pagará mediante un vale-regalo (lo que es posible y habitual). Entonces, una tienda podría tener una regla para permitir que sólo se compre un único artículo si se utiliza un vale-regalo. En consecuencia, deberían invalidarse las operaciones *introducirArticulo* que sigan a la primera.
- Si se paga la venta mediante un vale-regalo, se invalidan todos los tipos de devoluciones de dinero al cliente excepto otro vale-regalo. Por ejemplo, si el cajero solicita el cambio en dinero en efectivo o como crédito para la cuenta del cliente en la tienda, se deben invalidar estas peticiones.
- Suponga que cuando se crea una nueva venta es posible identificar que es una donación benéfica (de la tienda a una ONG). La tienda podría tener una regla que sólo permitiera la entrada de artículos de valor menor a 250 € cada uno, y también añadir sólo artículos a la venta si el “cajero” que inició la sesión es un encargado.

En cuanto al análisis de requisitos, se deben identificar los puntos concretos del escenario en todos los casos de uso (*introducirArticulo*, *elegirCambioEnEfectivo*,...). Para este estudio, sólo se considerará el punto *introducirArticulo*, pero se puede aplicar la misma solución a todos los puntos.

Suponga que el arquitecto software quiere un diseño que afecte poco a los componentes software que ya existen. Es decir, quiere diseñar separando los intereses, y factorizar esta regla en un interés separado. Es más, suponga que el arquitecto no está seguro de cuál es la mejor implementación para gestionar esta regla conectable, y podría querer experimentar con diferentes soluciones para representar, cargar y evaluar las reglas. Por ejemplo, las reglas se pueden implementar siguiendo el patrón Estrategia, o con intérpretes de reglas de libre distribución que leen e interpretan un conjunto de reglas IF-THEN, o con intérpretes de reglas comerciales, que hay que comprar, entre otras soluciones.

Para solucionar este problema de diseño, se puede utilizar el patrón Fachada.

Fachada (*Facade*)

Contexto/Problema

Se requiere una interfaz común, unificada para un conjunto de implementaciones o interfaces dispares —como en un subsistema—. Podría no ser conveniente acoplarla con muchas cosas del subsistema, o la implementación del subsistema podría cambiar. ¿Qué hacemos?

Solución

Defina un único punto de conexión con el subsistema —un objeto fachada que envuelve al subsistema—. Este objeto fachada presenta una única interfaz unificada y es responsable de colaborar con los componentes del subsistema.

Una Fachada es un objeto “front-end” que es el único punto de entrada para los servicios de un subsistema⁶; la implementación y otros componentes del subsistema son privados y no pueden verlos los componentes externos. La Fachada proporciona Variaciones Protegidas frente a los cambios en las implementaciones de un subsistema.

Por ejemplo, definiremos un subsistema “motor de reglas”, cuya implementación específica no se conoce todavía. Será responsable de evaluar un conjunto de reglas contra una operación (mediante alguna implementación oculta), e indicar entonces si alguna de las reglas invalida la operación.

El objeto fachada de este subsistema se llamará *FachadaMotorReglasPDV*. El diseñador decide colocar las llamadas a esta fachada cerca del comienzo de los métodos que se han definido como los puntos para las reglas conectables, como en este ejemplo:

```
public class Venta
{
    public void crearLineaDeVenta( EspecificacionDelProducto espec, int cantidad)
    {
        LineaDeVenta ldv = new LineaDeVenta(espec, cantidad);

        //llamada a la fachada
        if (FachadaMotorReglasPDV.getInstancia().esInvalido (ldv, this))
            return;

        lineasDeVenta.add(ldv);
    }

    //...
}

//final de la clase
```

Véase el uso del patrón Singleton. Normalmente se accede a la Fachada por medio del Singleton.

Con este diseño, la complejidad y la implementación del modo en el que se representarán y evaluarán las reglas se oculta en el subsistema del “motor de reglas”, al que se accede por medio de la fachada *FachadaMotorReglasPDV*. Obsérvese que el subsistema que oculta el objeto fachada podría contener docenas o cientos de clases de objetos, o incluso una solución no orientada a objetos, únicamente como cliente del subsistema sólo vemos su único punto de acceso público.

Y se ha conseguido una separación de intereses en cierta medida — se han delegado en otro subsistema todas las cuestiones de manejo de reglas—.

Resumen

El patrón Fachada es sencillo y se utiliza ampliamente. Oculta un subsistema detrás de un objeto.

⁶ El término “subsistema” se está utilizando en un sentido informal para designar a un grupo de componentes relacionados, no exactamente como se define en UML.

Ejercicios Propuestos**Ejercicio 1**

Diseñe la gestión de las reglas con el patrón Estrategia, cuyos nombres de clase se obtienen dinámicamente leyendo de una fuente externa.

Ejercicio 2

Si se implementa en Java, diseñe la gestión de las reglas con Jess, un intérprete de reglas de libre distribución para uso con fines académicos disponible en <http://herzberg.ca.sandia.gov/jess/>

Patrones Relacionados

Normalmente se accede a las fachadas por medio del patrón Singleton. Los dos proporcionan Variaciones Protegidas de la implementación de un subsistema, añadiendo un objeto de Indirección que ayuda a mantener Bajo Acoplamiento. Los objetos externos se acoplan a un punto del subsistema: el objeto fachada.

Como se describe en el patrón Adaptador, un objeto adaptador puede utilizarse para envolver el acceso a sistemas externos que tienen interfaces diferentes. Esto es una clase de fachada pero el énfasis está en proporcionar adaptación a interfaces diferentes, y por ello se llama más específicamente un adaptador.

Notación UML: UML proporciona una notación para agrupaciones de propósito general denominadas **paquetes**, cuyo ícono es un tipo de carpeta etiquetada. Los paquetes se pueden utilizar para mostrar agrupaciones lógicas de objetos; se podría corresponder con algo como los paquetes de Java o los *namespaces* de C++, o con otros componentes agregados o subsistemas lógicamente distintos. Nótese que en la Figura 23.19 sólo la *FachadaMotorReglasPDV* es pública con respecto a su paquete.

Existe notación UML más compleja para representar subsistemas, pero la notación de la Figura 23.19 será suficiente por ahora. El diseño con paquetes se estudiará con más detalle en la siguiente iteración.

23.9. Observador/Publicar-Suscribir/Modelo de Delegación de Eventos (GoF)

Otro requisito de la iteración es añadir la capacidad de que una ventana GUI actualice la información que muestra sobre el total de la venta cuando éste cambia (ver Figura 23.20). La idea es solucionar el problema para este caso particular, y después en iteraciones posteriores, extender la solución para actualizar la información de la GUI también para los cambios de otros datos.

¿Por qué no hacer lo siguiente como solución? Cuando la *Venta* cambia su total, el objeto *Venta* envía un mensaje a la ventana, pidiéndole que actualice la información que muestra.

Recordemos que el principio de Separación Modelo-Vista disuade de tales soluciones. Establece que los objetos del “modelo” (objetos que no pertenecen al UI como la *Venta*) no deberían conocer los objetos de la vista o presentación tales como una ventana. Dicho principio fomenta Bajo Acoplamiento entre los objetos de otras capas y los de la capa de presentación (UI).

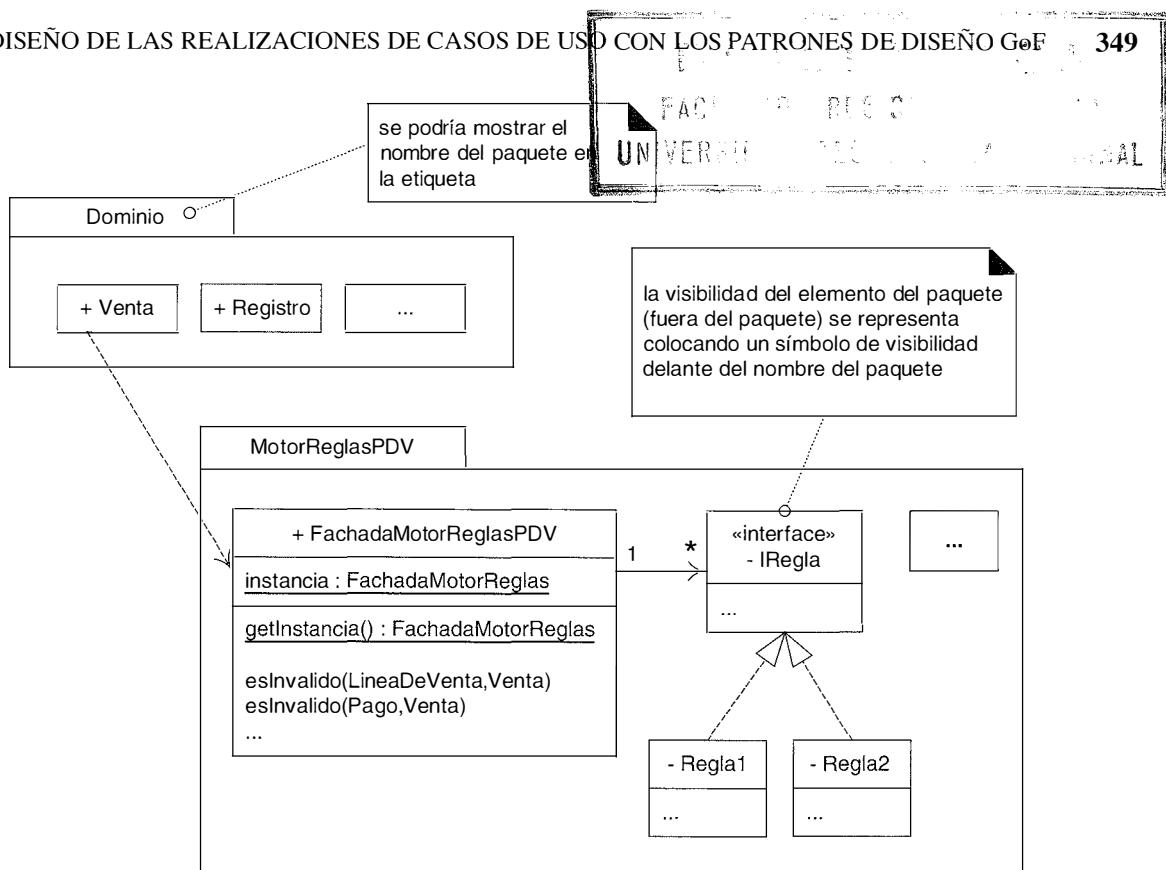


Figura 23.19. Notación de paquetes UML.

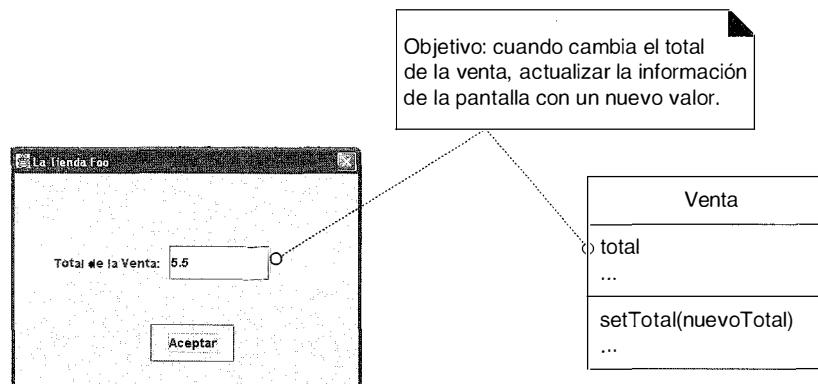
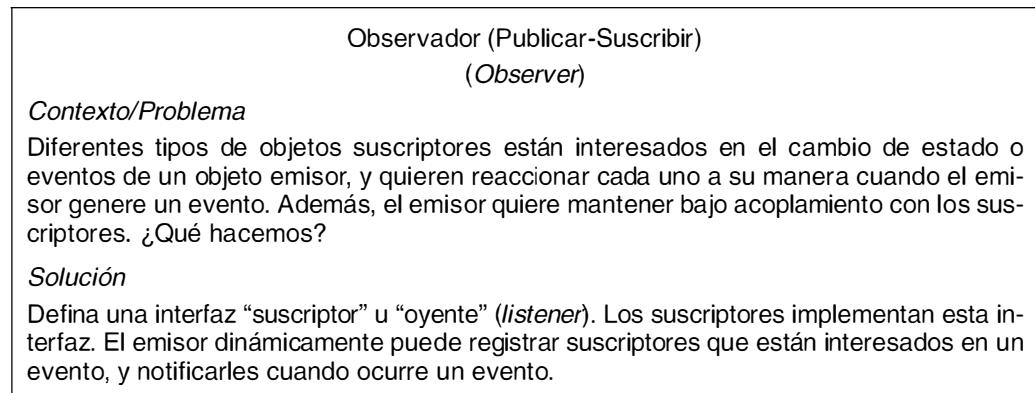


Figura 23.20. Actualización de la interfaz cuando cambia el total de la venta.

Una consecuencia de mantener este bajo acoplamiento es que permite reemplazar la vista o capa de presentación por una nueva, o una ventana concreta por otra nueva, sin afectar a los objetos que no pertenecen a la UI. Si los objetos del modelo no conocen los objetos Swing de Java (por ejemplo), entonces es posible desconectar una interfaz Swing, o desconectar una ventana concreta, y conectar algo más.

Por tanto, la Separación Modelo-Vista mantiene Variaciones Protegidas con respecto a los cambios en la interfaz de usuario.

Para solucionar este problema de diseño se puede utilizar el patrón Observador.



Una solución de ejemplo se describe en detalle en la Figura 23.21.

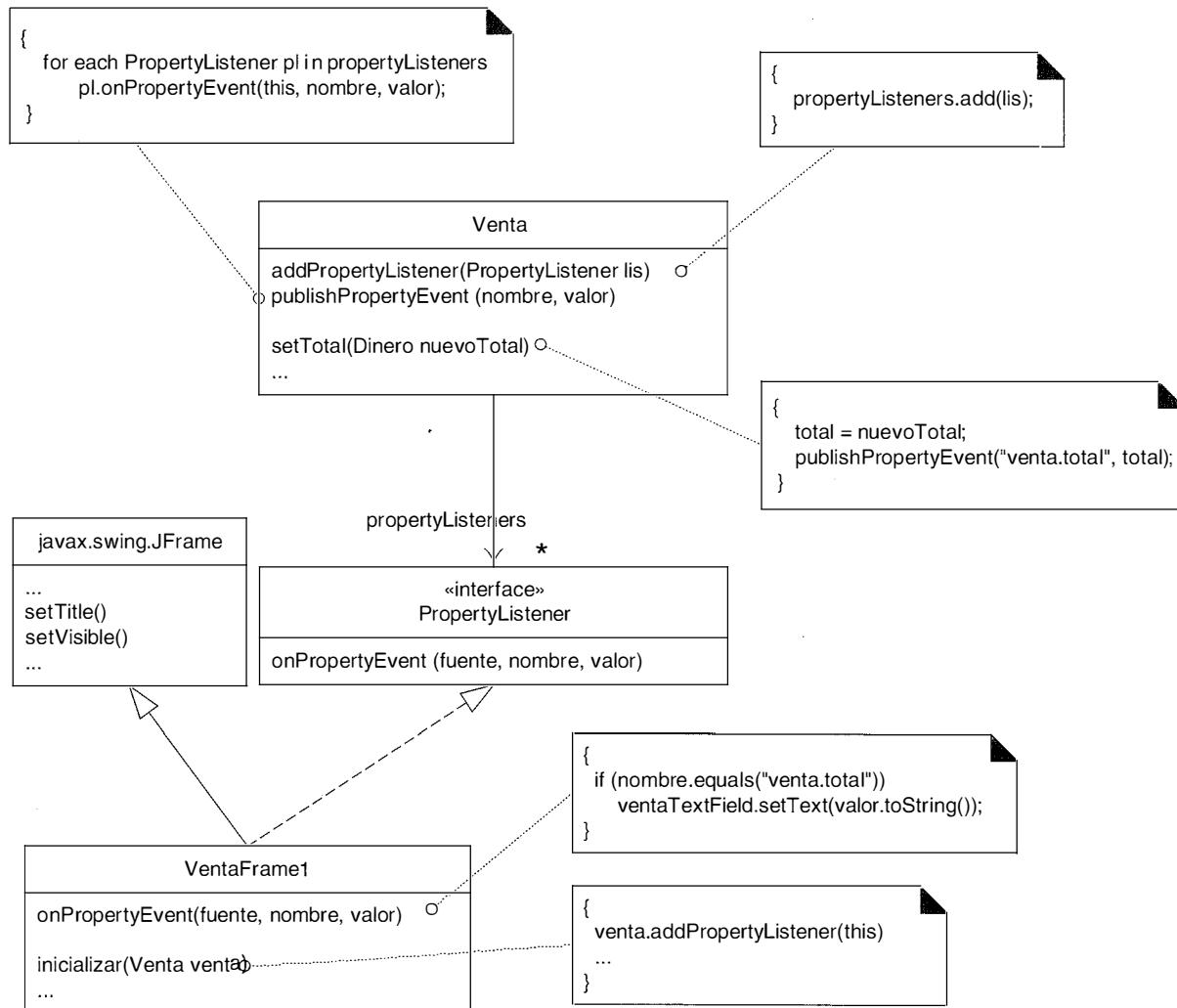


Figura 23.21. El patrón Observador.

Las ideas y pasos fundamentales de este ejemplo son:

1. Se define una interfaz; en este caso *PropertyListener*⁷ con la operación *onPropertyEvent*.
2. Se define la ventana que implementa la interfaz.
 - o *VentaFrame1* implementará el método *onPropertyEvent*.
3. Cuando se inicializa la ventana *VentaFrame1*, se le pasa la instancia de *Venta* de la que está mostrando el total.
4. La ventana *VentaFrame1* se registra o suscribe a la instancia de *Venta* para que le notifique acerca de los “eventos sobre la propiedad”, por medio del mensaje *addPropertyListener*. Es decir, cuando una propiedad (como el total) cambia, la ventana quiere que se le notifique.
5. Observe que la *Venta* no conoce los objetos *VentaFrame1*; más bien sólo conoce los objetos que implementan la interfaz *PropertyListener*. Esto disminuye el acoplamiento entre la *Venta* y la ventana —se acopla sólo con una interfaz, no con la clase de la GUI—.
6. Por tanto, la instancia de *Venta* es un *emisor* de “eventos sobre la propiedad”. Cuando cambia el total, itera sobre todos los objetos *PropertyListener* que están suscritos, y se lo notifica a cada uno.

Notación UML: Obsérvese en la Figura 23.21 que a los métodos interesantes se les añaden notas de comentario que muestran la implementación. Estas notas añaden información sobre el comportamiento dinámico a un diagrama de tipos estáticos. En algunos casos, un diagrama de clases con estas notas puede sustituir la necesidad de diagramas de interacción adicionales. Esto no significa que estemos aconsejando que se eviten los diagramas de interacción, sino que indica enfoques de notación alternativos.

El objeto *VentaFrame1* es el observador/suscriptor/oyente. En la Figura 23.22, *suscribe* su interés en los eventos sobre las propiedades de la *Venta*, que es un *emisor* de

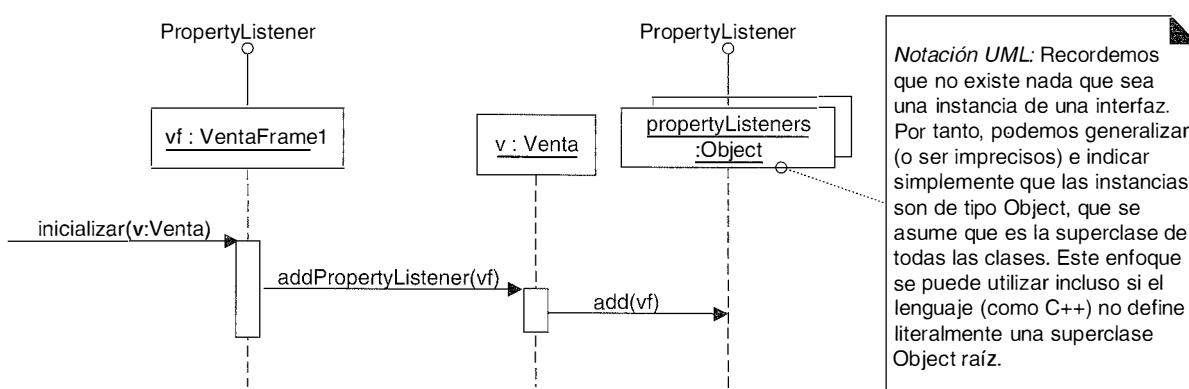


Figura 23.22. El observador *VentaFrame1* se suscribe al emisor *Venta*.

⁷ *N. del T.:* No se han traducido los nombres de los métodos en los ejemplos de esta sección ya que corresponden a la implementación en Java del patrón.

eventos sobre la propiedad. La *Venta* añade el objeto a su lista de suscriptores de tipo *PropertyListener*. Nótese que la *Venta* no conoce a *VentaFrame1* como un objeto *VentaFrame1* sino como un objeto *PropertyListener*; esto disminuye el acoplamiento entre la capa del modelo y la vista.

Como se ilustra en la Figura 23.23, cuando cambia el total de la *Venta*, itera sobre todos los suscriptores que se han registrado, y “publica un evento” enviando a cada uno el mensaje *onPropertyEvent*.

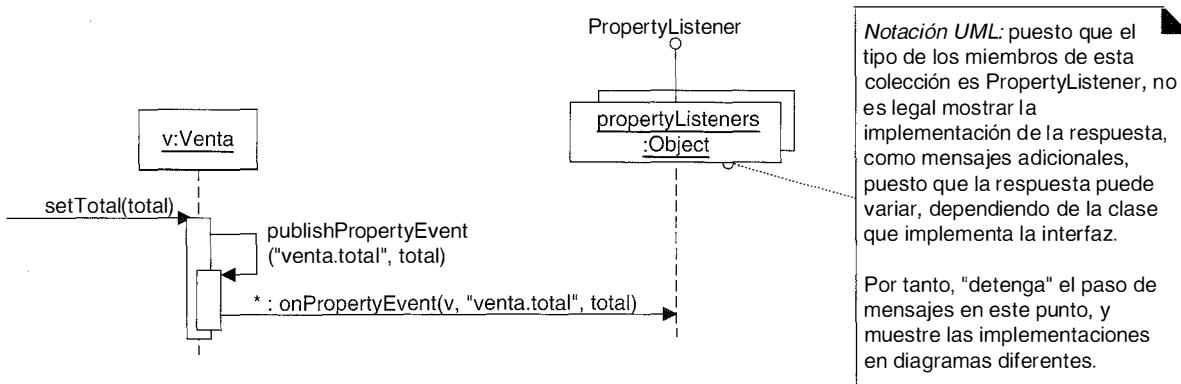


Figura 23.23. La *Venta* publica un evento sobre la propiedad a todos sus suscriptores.

Notación UML: Observe el enfoque para gestionar los mensajes polimórficos en un diagrama de interacción, en la Figura 23.23.

VentaFrame1, que implementa la interfaz *PropertyListener*, por tanto implementa el método *onPropertyEvent*. Cuando el *VentaFrame1* recibe el mensaje, envía un mensaje a su objeto *JTextField* que es un elemento gráfico de la GUI para actualizar el nuevo total de la venta. Ver Figura 23.24.

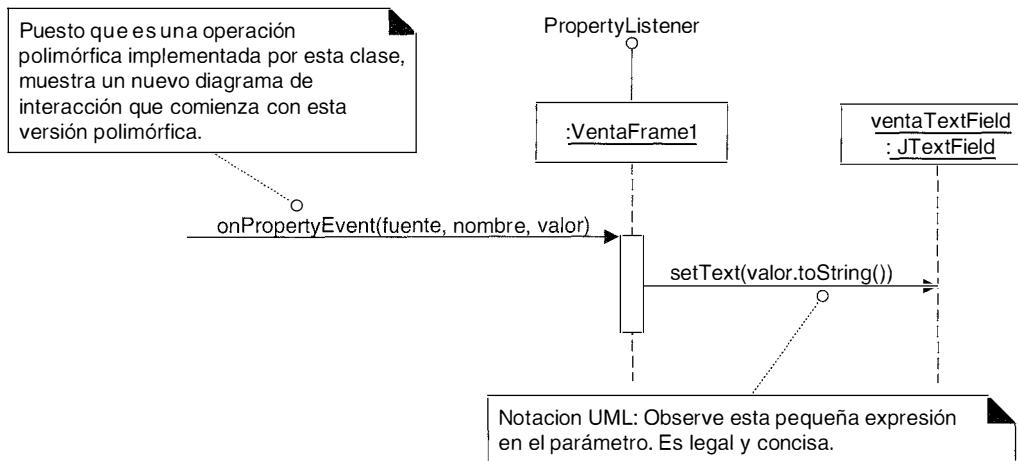


Figura 23.24. El suscriptor *VentaFrame1* recibe la notificación de un evento publicado.

En este patrón, existe todavía algo de acoplamiento entre el objeto del modelo (la *Venta*) y el objeto de la vista (el *VentaFrame1*). Pero se trata de un acoplamiento débil con

una interfaz independiente de la capa de presentación —la interfaz *PropertyListener*—. Y el diseño realmente no necesita que se registre ningún objeto suscriptor en el emisor (ningún objeto tiene que estar escuchando). Es decir, la lista de objetos *PropertyListener* que se registran en la *Venta* puede estar vacía. Resumiendo, el acoplamiento con una interfaz genérica de objetos que no necesita estar presente y a la que se le pueden añadir (y eliminar) elementos dinámicamente, mantiene bajo acoplamiento. Por tanto, se ha conseguido Variaciones Protegidas con respecto a los cambios en la interfaz de usuario, mediante el uso de una interfaz y el polimorfismo.

¿Por qué se le denomina Observador, Publicar-Suscribir, o Modelo de Delegación de Eventos?

Originalmente, este estilo se llamó publicar-suscribir, y todavía se le conoce ampliamente por este nombre. Un objeto “publica eventos”, como la *Venta* que publica el “evento sobre la propiedad” cuando cambia el total. Podría pasar que ningún objeto estuviera interesado en este evento, en cuyo caso, la *Venta* no tendría ningún suscriptor registrado. Pero los objetos que están interesados, “suscriben” o registran su interés en un evento pidiéndole al emisor que le notifique. Esto se hizo con el mensaje *Venta--addPropertyListener*. Cuando tiene lugar el evento, se notifica a los suscriptores que están registrados mediante un mensaje.

Se le ha llamado Observador porque el oyente o suscriptor está observando el evento; ese término se hizo popular en Smalltalk a principios de los ochenta.

También se le ha denominado Modelo de Delegación de Eventos (en Java) porque el emisor delega la gestión de los eventos a los “oyentes” (suscriptores; ver Figura 23.25).

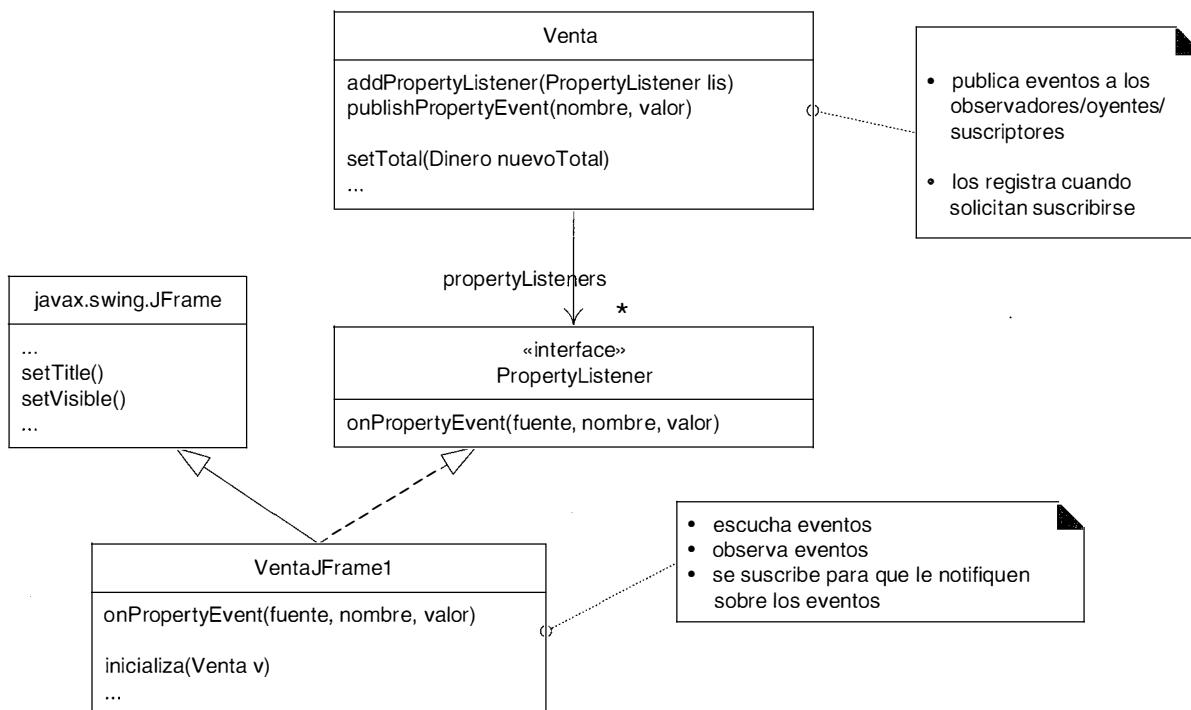


Figura 23.25. ¿Quién es el observador, oyente, suscriptor y emisor?

El Observador no sólo es para conectar UIs y objetos del modelo

El ejemplo anterior ilustraba la conexión con el Observador entre los objetos que no pertenecen a la UI y los objetos de la UI. Sin embargo, existen otros usos habituales.

El uso más común de ese patrón es para el manejo de los eventos de los elementos gráficos de la GUI, tanto en la tecnología Java (AWT y Swing) como en la .NET de Microsoft. Cada elemento gráfico es un emisor de eventos relacionados con la GUI, y otros objetos pueden suscribir su interés en ellos. Por ejemplo, un *JButton* de las Swing publica un “evento de acción” cuando se presiona. Otro objeto se registrará con el botón de manera que cuando se presione, se envía un mensaje al objeto y puede realizar alguna acción.

Otro ejemplo se muestra en la Figura 23.26 donde se ilustra un *RelojAlarma*, que es un emisor de eventos de alarma, y varios suscriptores. Este ejemplo es ilustrativo ya que pone de relieve que la interfaz *AlarmaListener* puede ser implementada por muchas clases, se pueden registrar simultáneamente muchos objetos como oyentes, y cada uno reaccionará ante el “evento de alarma” de manera diferente.

Un emisor puede tener muchos suscriptores a un evento

Como se desprende de la Figura 23.26, una instancia de emisor podría tener de uno a muchos suscriptores registrados. Por ejemplo, una instancia de *RelojAlarma* podría tener registrados tres objetos *VentanaDeAlarma*, cuatro objetos *Busca* y un *GuardianDeFidabilidad*. Cuando ocurre un evento de alarma, se le notifica a los ocho objetos *AlarmaListener* por medio del evento *onAlarmaEvent*.

Implementación

Eventos

Tanto en las implementaciones del Observador en Java como en C# .NET, se comunica un “evento” por medio de un mensaje ordinario, como *onPropertyEvent*. Además, en ambos casos, el evento se define más formalmente como una clase, y se rellena con los datos del evento apropiados. Entonces el evento se pasa como parámetro en el mensaje de evento.

Por ejemplo:

```
class PropertyEvent extends Event
{
    private Object fuenteDelEvento;
    private String nombrePropiedad;
    private Object valorAntiguo;
    private Object valorNuevo;
    //...
}
//...
```

```

class Venta
{
    private void publishPropertyEvent(
        String nombre, Object antiguo, Object nuevo)
    {
        PropertyEvent evt =
            new PropertyEvent (this, "venta.total", antiguo, nuevo);

        for each AlarmaListener al in alarmaListeners
            al.onPropertyEvent(evt);
    }

    //...
}

```

Java

Cuando se lanzó el JDK 1.0 en enero de 1996, contenía una implementación simple de publicar-suscribir basada en una clase y una interfaz denominadas *Observable* y

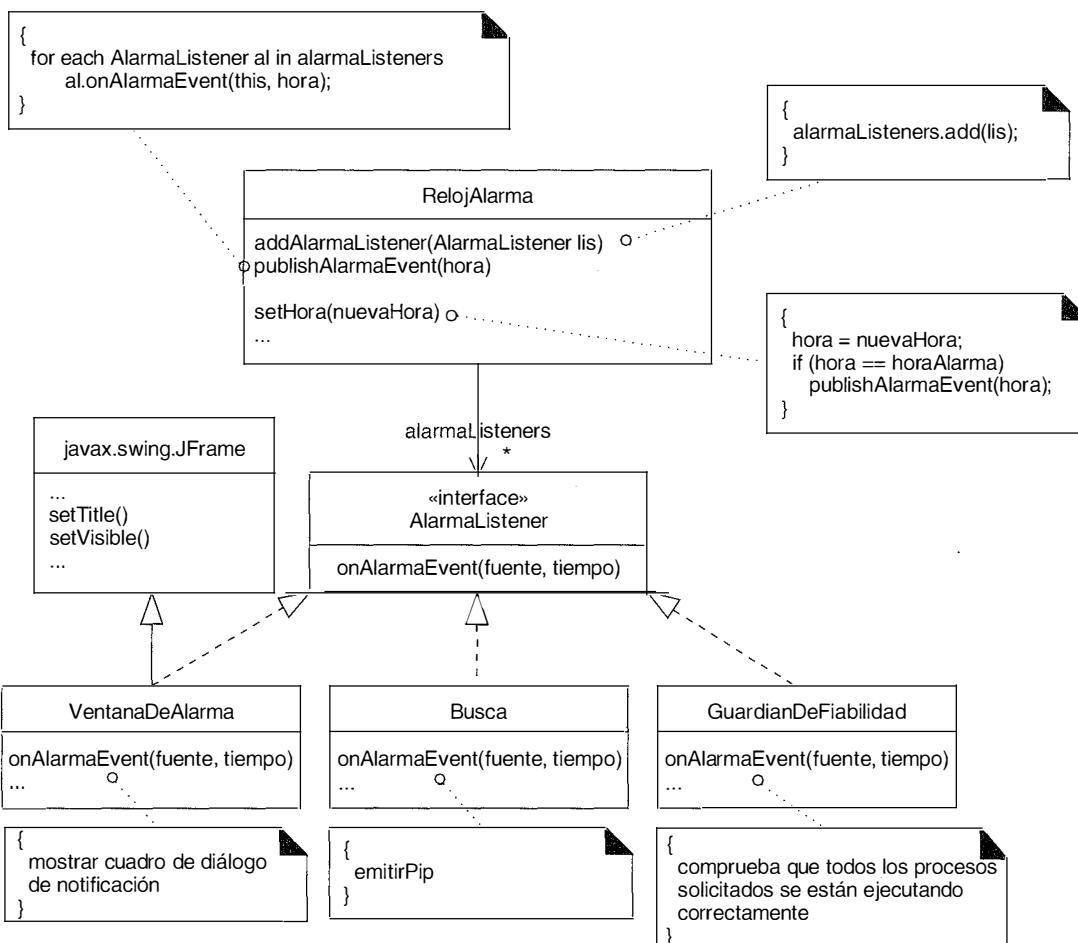


Figura 23.26. El Observador aplicado a los eventos de la alarma, con diferentes suscriptores.

Observer, respectivamente. Esto se copió básicamente, sin ninguna mejora, del enfoque de primeros de los ochenta de la implementación de publicar-suscribir en Smalltalk.

Por tanto, a finales de 1996, dentro de la versión JDK 1.1, se sustituyó el diseño Observable-Observer por el Modelo de Delegación de Eventos de Java (DEM, *Delegation Event Model*), una versión más robusta de publicar-suscribir, aunque se mantuvo el diseño original por compatibilidad con las versiones anteriores (pero en general debe evitarse).

Los diseños descritos en este capítulo son consistentes con el DEM, pero se han simplificado ligeramente para resaltar las ideas fundamentales.

Resumen

El Observador proporciona un modo de acoplar débilmente los objetos en términos de la comunicación. Los emisores conocen a los suscriptores sólo a través de una interfaz, y los suscriptores pueden registrarse (o darse de baja) de los emisores dinámicamente.

Patrones Relacionados

El Observador se basa en el Polimorfismo, y proporciona Variaciones Protegidas en cuanto a que protege al emisor del conocimiento de la clase específica de objetos, y número de objetos, con los que se comunica cuando genera un evento.

23.10. Conclusión

La lección principal que se puede extraer de esta exposición es que se pueden diseñar los objetos y asignar las responsabilidades con la ayuda de los patrones. Esto proporciona un conjunto explicable de estilos mediante los cuales se pueden construir sistemas orientados a objetos bien diseñados.

23.11. Lecturas adicionales

El libro de texto básico es *Design Patterns* de Gamma, Helm, Johnson y Vlissides, y es fundamental que lo lean todos los diseñadores de objetos.

Cada año se celebra un congreso “Pattern Languages of Programs” (PLOP, Lenguajes de Patrones de Programas) a partir del cual se publica una recopilación de patrones, en la serie *Pattern Languages of Program Design*, volumen 1, 2, etcétera. Se recomienda toda la serie.

Pattern-Oriented Software Architecture, volumen 1 y 2, promueve la discusión sobre los patrones en cuestiones de arquitectura a gran escala. El volumen 1 presenta una taxonomía de patrones.

Hay publicados cientos de patrones. Rising resume un porcentaje respetable de ellos en *The Pattern Almanac*.

PARTE 5

**ELABORACIÓN
EN LA ITERACIÓN 3**

Capítulo 24

LA ITERACIÓN 3 Y SUS REQUISITOS

24.1. Requisitos de la Iteración 3

- Cuando no se puede acceder a los servicios remotos proporcionar el mantenimiento de los servicios ante los fallos mediante servicios locales. Por ejemplo, si no se puede acceder al producto remoto de la base de datos, utilice una versión local de datos almacenados en la caché.
- Proporcionar el soporte para el manejo de los dispositivos de PDV, como el cajón de caja y el dispensador de monedas.
- Manejar las autorizaciones de pago a crédito.
- Soporte para objetos persistentes.

24.2. Énfasis de la Iteración 3

Durante la fase de inicio y la Iteración 1 se exploraron una variedad de cuestiones fundamentales en el análisis de requisitos y el A/DOO. La Iteración 2 se ocupó en detalle del diseño de objetos. Esta tercera iteración considera de nuevo una perspectiva más amplia, explorando una extensa variedad de temas relacionados con el análisis y diseño, entre los que se encuentran:

- Relaciones entre casos de uso.
- Generalización y especialización.
- Modelado de estados.
- Arquitecturas en capas.
- El diseño de paquetes.
- Análisis arquitectural.
- Más patrones de diseño GoF.
- El diseño de *frameworks*, en particular un framework de persistencia.

Capítulo 25

RELACIONES ENTRE CASOS DE USO

¿Por qué los programadores confunden Halloween y las navidades? Porque OCT(31)=DEC(25).

Objetivos

- Relacionar los casos de uso mediante las asociaciones *include* y *extend*.
-

Introducción

Los casos de uso se pueden relacionar entre ellos. Por ejemplo, un caso de uso de subfunción como *Gestionar Pagos a Crédito* podría formar parte de varios casos de uso ordinarios, como *Procesar Venta* o *Procesar Alquiler*. La organización de los casos de uso mediante relaciones no influye en el comportamiento o los requisitos del sistema. Más bien, es simplemente una forma de organizar para (idealmente) mejorar la comunicación y la comprensión de los casos de uso, reducir la duplicación de texto y mejorar la gestión de los documentos de casos de uso.

Una advertencia

En algunas organizaciones que trabajan con casos de uso, se dedica mucho tiempo improductivo a debatir sobre cómo relacionar los casos de uso en los diagramas de casos de uso, en lugar de dedicarlo al trabajo importante de los casos de uso: escribir texto. En consecuencia, aunque este capítulo presenta las relaciones entre los casos de uso, el tema y la dedicación que merece deberían ponerse en perspectiva: tiene algo de valor, pero el trabajo importante es escribir el texto de los casos de uso. La especificación de los requisitos se hace escribiendo, no organizando los casos de uso, que es un paso opcional, posiblemente para ayudar a comprenderlos o reducir duplicaciones. Si un equipo co-

mienza el modelado de los casos de uso dedicando horas (o peor, días) discutiendo un diagrama de caso de uso y las relaciones entre los casos de uso (“¿Ésa debería ser una relación *include* o *extend*? ¿Deberíamos especializar este caso de uso?”), en lugar de centrarse en escribir rápidamente el texto clave del caso de uso, se está haciendo un esfuerzo significativo en el sitio equivocado.

Es más, la organización de los casos de uso utilizando las relaciones puede evolucionar iterativamente en pequeñas etapas a lo largo de la fase de elaboración; no es útil intentar aplicar un esfuerzo del estilo del modelo en cascada en el que se define y refina totalmente un diagrama completo de caso de uso y el conjunto de relaciones en una etapa al comienzo del proyecto.

25.1. La relación de inclusión (*include*)

Ésta es la relación más común e importante.

Es habitual tener algún comportamiento parcial común a varios casos de uso. Por ejemplo, la descripción del pago a crédito tiene lugar en varios casos de uso, entre los que se encuentran *Procesar Venta*, *Procesar Alquiler*, *Contribuir a Plan de Ahorro*, etcétera. En lugar de duplicar este texto, es conveniente separarlo en su propio caso de uso de subfunción, e indicar su inclusión. Esto es sencillamente factorizar y enlazar texto para evitar duplicaciones¹.

Por ejemplo:

UC1: Procesar Venta

...

Escenario principal de éxito:

1. El Cliente llega a un terminal PDV con mercancías y/o servicios para comprar.
- ...
7. El Cliente paga y el Sistema gestiona el pago.
- ...

Extensiones:

- 7b. Pago a crédito: Incluye Gestionar Pago a Crédito.
- 7c. Pago con cheque: Incluye Gestionar Pago con Cheque.
- ...

UC7: Procesar Alquiler

...

Extensiones:

- 6b. Pago a crédito: Incluye Gestionar Pago a Crédito.
- ...

¹ Sirve de ayuda que los enlaces se implementen también con hipervínculos navegables.

UC12: Gestionar Pago a Crédito

...

Nivel: Subfunción.

Escenario principal de éxito:

1. El Cliente introduce la información acerca de su cuenta de crédito.
2. El Sistema envía la solicitud de autorización del pago a un Sistema para el Servicio de Autorización de Pago, y solicita la aprobación del pago.
3. El Sistema recibe la aprobación del pago y la notifica al Cajero.
4. ...

Extensiones:

- 2a. El sistema detecta algún fallo al colaborar con el sistema externo:
 1. El Sistema informa del error al Cajero.
 2. El Cajero le pide al Cliente un modo de pago alternativo.

...

Ésta es la relación **de inclusión (include)**.

Una notación algo más breve (y, por tanto, quizás se prefiera) para indicar el caso de uso que se incluye es simplemente subrayarlo o destacarlo de alguna manera. Por ejemplo:

UC1: Procesar Venta

...

Extensiones:

- 7b. Pago a crédito: Gestionar Pago a Crédito.
- 7c. Pago con cheque: Gestionar Pago con Cheque.

Nótese que el caso de uso de subfunción *Gestionar Pago a Crédito* se encontraba originalmente en la sección de *Extensiones* del caso de uso *Procesar Venta*, pero se factorizó aparte para evitar duplicaciones. También observe que en el caso de uso de subfunción se utilizan las mismas estructuras *Escenario principal de éxito* y *Extensiones* que en los casos de uso de procesos de negocio ordinarios como *Procesar Venta*.

Fowler ofrece una guía sencilla y práctica sobre cuándo utilizar la relación de inclusión [FS00]:

Utilice *include* cuando se está repitiendo en dos o más casos de usos separados y quiere evitar repeticiones.

Otro motivo es simplemente descomponer un caso de uso abrumadoramente largo en subunidades para mejorar la comprensión.

Utilización de include con el manejo de eventos asíncronos

La relación de inclusión también se utiliza para describir el manejo de un evento asíncrono, como cuando un usuario es capaz de seleccionar o bifurcar, en cualquier momento, a una ventana, función o página web específica, o en un rango de pasos.

De hecho, la notación de los casos de uso para representar esta bifurcación asíncrona ya se estudió en la introducción a los casos de uso en el Capítulo 6, pero en ese momento no se presentó la incorporación de una invocación a un subcaso de uso que se incluye.

La notación básica es utilizar etiquetas siguiendo el estilo *a*, b*...* en la sección de Extensiones. Recordemos que esto implica una extensión o evento que puede ocurrir en cualquier momento. Una variación menor es utilizar una etiqueta con un rango, como *3-9*, cuando el evento asíncrono puede ocurrir en un rango relativamente amplio de los pasos del caso de uso, pero no todos.

UC1: Procesar Algo

...

Escenario principal de éxito:

1. ...

Extensiones:

a*. En cualquier momento, el Cliente selecciona la edición de la información personal: *Editar Información Personal*

b*. En cualquier momento, el Cliente selecciona imprimir la ayuda: *Presentar Ayuda Impresa*

2-11. El Cliente cancela: *Cancelar Confirmación de Transacción*

...

Resumen

La relación de inclusión se puede utilizar para la mayoría de problemas de relaciones de casos de uso. En resumen:

Factorice casos de uso de subfunción separados y utilice la relación *include* cuando:

- Están duplicados en otros casos de uso.
- Un caso de uso es *muy* complejo y largo, y separarlos en subunidades facilita la comprensión.

Como explicaremos, existen otras relaciones: extensión y generalización. Pero Cockburn, un modelador de casos de uso experto, aconseja escoger la relación de inclusión por encima de las relaciones de extensión y generalización:

Como primera regla empírica, utilice siempre la relación *include* entre los casos de uso. La gente que sigue esta regla declara que existen menos malentendidos entre ellos y sus lectores sobre lo que escriben que la gente que mezcla *include* con *extend* y *generalizes* [Cockburn01].

25.2. Terminología: casos de uso concretos, abstractos, base y adicional

Un caso de uso **concreto** es iniciado por un actor y lleva a cabo todo el comportamiento que desea el actor [RUP]. Éstos son los casos de uso de los procesos del negocio ele-

mentales. Por ejemplo, *Procesar Venta* es un caso de uso concreto. En cambio, un **caso de uso abstracto** nunca se instancia por él mismo; es un caso de uso de subfunción que forma parte de otro caso de uso. *Gestionar Pago a Crédito* es abstracto; no se mantiene independiente, sino que siempre forma parte de otra historia, como *Procesar Venta*.

Un caso de uso que incluye otro caso de uso, o que es extendido o especializado por otro caso de uso se denomina **caso de uso base**. *Procesar Venta* es un caso de uso base con respecto al caso de uso de subfunción *Gestionar Pago a Crédito* que incluye. Por otro lado, el caso de uso que es una inclusión, extensión o especialización se denomina **caso de uso adicional**. *Gestionar Pago a Crédito* es el caso de uso adicional en la relación de inclusión de *Procesar Venta*. Los casos de uso adicionales normalmente son abstractos. Los casos de uso base generalmente son concretos.

25.3. La relación de extensión (*extend*)

Suponga que el texto de un caso de uso no debiera modificarse (al menos no de manera significativa) por alguna razón. Quizás modificar continuamente el caso de uso con innumerables nuevas extensiones y pasos condicionales es un dolor de cabeza de mantenimiento, o se ha establecido el caso de uso como un artefacto estable, y no se puede tocar. ¿Cómo añadir al caso de uso sin modificar su texto original?

La relación **de extensión (*extend*)** proporciona una respuesta. La idea es crear un caso de uso que extiende o añade, y con él, describe dónde y bajo qué condiciones extiende el comportamiento de algún caso de uso base. Por ejemplo:

UC1: Procesar Venta (el caso de uso base)

...

Puntos de Extensión: Cliente VIP, paso 1. Pago, paso 7.

Escenario principal de éxito:

1. El Cliente llega a un terminal PDV con mercancías y/o servicios para comprar.
- ...
7. El Cliente paga y el Sistema gestiona el pago.
- ...

UC15: Gestionar Pago con Vale-Regalo (el caso de uso que extiende)

...

Activa: El Cliente quiere pagar con vale-regalo

Puntos de Extensión: Pago en Procesar Venta

Nivel: Subfunción.

Escenario Principal de Éxito:

1. El Cliente le da el vale-regalo al Cajero.
2. El Cajero introduce el identificador del vale-regalo.
- ...

Éste es un ejemplo de una relación **de extensión**. Observe el uso de un **punto de extensión**, y que el caso de uso que extiende se activa por alguna condición. Los puntos de extensión son etiquetas en el caso de uso base que extiende referencia como puntos de extensión, de manera que los números de los pasos del caso de uso base pueden cambiar sin afectar al caso de uso que extiende —indirección una vez más—.

Algunas veces, el punto de extensión es simplemente “En cualquier punto del caso de uso X”. Esto es especialmente habitual en sistemas con muchos eventos asíncronos, como un procesador de texto (“Hacer ahora la corrección ortográfica”, “Hacer ahora una búsqueda en el tesauro”), o sistemas de control reactivo. Observe sin embargo, como se describe en la sección anterior de la relación de inclusión, que también se puede utilizar la relación de inclusión para describir el manejo de eventos asíncronos. La alternativa *extend* es una opción cuando el caso de uso base está cerrado a las modificaciones.

Observe que una señal de calidad de la relación de extensión es que el caso de uso base (*Procesar Venta*) no referencia al caso de uso que lo extiende (*Gestionar Pago con Vale-Regalo*) y, por tanto, no define o controla las condiciones bajo las cuales se activan las extensiones. El caso de uso *Procesar Venta* es completo y forma un todo por él mismo, sin conocer los casos de uso que lo extiende.

Fíjese que el caso de uso adicional *Gestionar Pago con Vale-Regalo* de manera alternativa se podría haber referenciado en *Procesar Venta* mediante una relación de inclusión, como con *Gestionar Pago a Crédito*. Con frecuencia es adecuado. Pero este ejemplo estaba motivado por la restricción de que el caso de uso *Procesar Venta* no se iba a modificar, que es la situación en la que se utiliza la extensión en lugar de la inclusión.

Además, nótese que este escenario del vale-regalo se podría haber registrado simplemente añadiéndolo como una extensión en la sección de *Extensiones de Procesar Venta*. Este enfoque evita tanto la relación de inclusión como la de extensión, y la creación de un caso de uso de subfunción separado.

De hecho, normalmente es preferible actualizar simplemente la sección de *Extensiones*, en lugar de crear complejas relaciones de casos de uso.

Algunas guías de casos de uso recomiendan la utilización de casos de uso que extienden y la relación de extensión para modelar el comportamiento condicional u opcional del caso de uso base. Esto no es incorrecto, pero no comprende que el comportamiento condicional u opcional se puede registrar simplemente como texto en la sección de *Extensiones* del caso de uso base. La dificultad de utilizar la relación de extensión y más casos de uso no está motivada solamente por el comportamiento opcional.

Lo que motiva esencialmente el uso de la técnica de extensión es cuando no es conveniente por alguna razón modificar el caso de uso base.

25.4. La relación de generalización (*generalize*)

La discusión acerca de la relación de generalización queda fuera del alcance de este libro. Sin embargo, observe que los expertos en casos de uso han estado realizando con éxito el trabajo sobre casos de uso sin esta relación opcional, que añade otro nivel de complejidad a los casos de uso, y todavía no existe un acuerdo entre los que la utilizan

sobre una guía de buenas prácticas de cómo sacar provecho de esta idea. Los consultores sobre casos de uso observan que habitualmente da lugar a complicaciones y que se dedica mucho tiempo improductivo a la inclusión de muchas relaciones de casos de uso.

25.5. Diagramas de casos de uso

La Figura 25.1 ilustra la notación UML para la relación de inclusión, que es la única que se va a utilizar en el caso de estudio, siguiendo el consejo de los expertos en casos de uso de mantener las cosas sencillas y preferir la relación de inclusión.

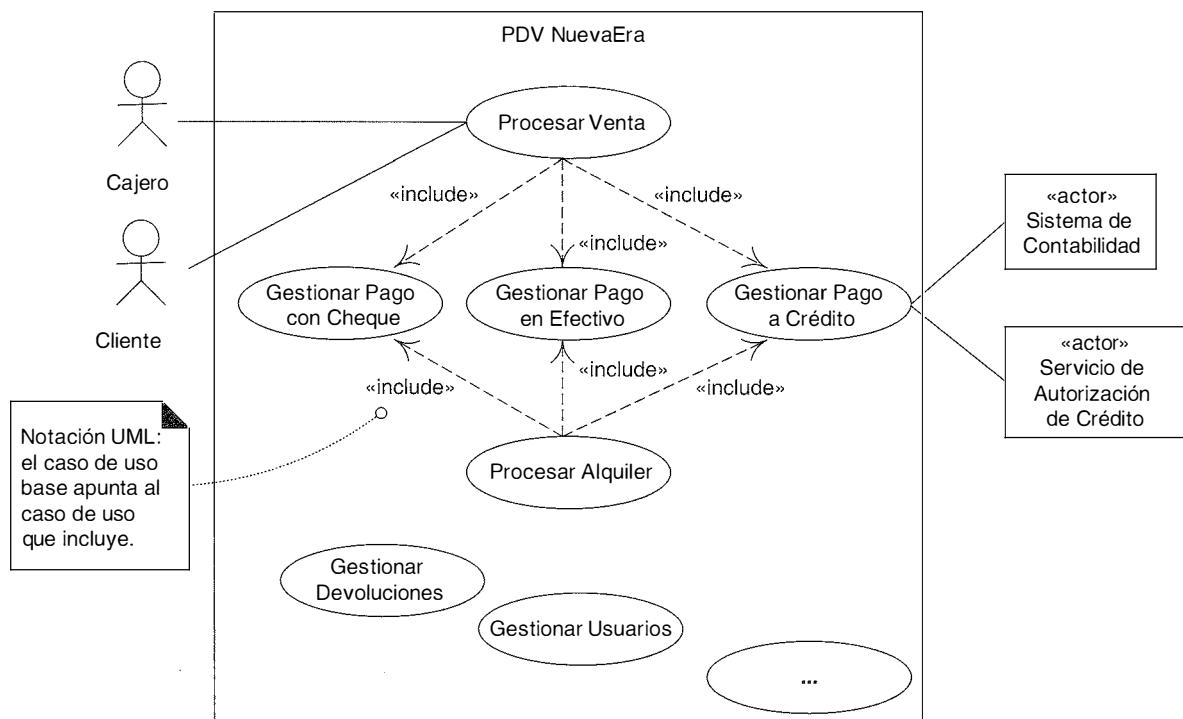


Figura 25.1. Relación de inclusión de los casos de uso en el Modelo de Casos de Uso.

La notación de la relación de extensión se ilustra en la Figura 25.2.

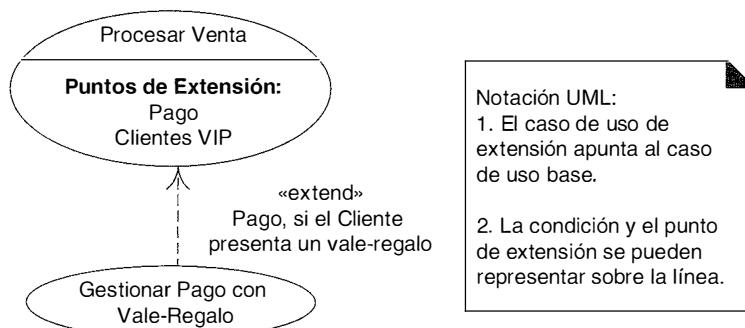


Figura 25.2. La relación de extensión.



Capítulo 26

MODELADO DE LA GENERALIZACIÓN

Las clasificaciones simplonas y las generalizaciones falsas son las maldiciones de la vida organizada.

Una generalización de H. G. Wells

Objetivos

- Crear jerarquías de generalización-especialización.
 - Identificar cuándo merece la pena mostrar una subclase.
 - Aplicar las pruebas del “100%” y “Es-un” para validar las subclases.
-

Introducción

La generalización y la especialización son conceptos fundamentales en el modelado del dominio que favorece una economía de palabras; más aún, las jerarquías de clases conceptuales a menudo constituyen la fuente de inspiración para las jerarquías de clases software que se aprovechan de la herencia y reducen la duplicación de código.

26.1. Nuevos conceptos para el Modelo del Dominio

Como en la iteración 1, el Modelo del Dominio del UP podría desarrollarse incrementalmente considerando los conceptos en los requisitos para esta iteración. Nos servirán de ayuda técnicas como la *Lista de Categorías de Conceptos* y la identificación de frases nominales. Un enfoque efectivo para desarrollar un modelo del dominio robusto y rico es estudiar el trabajo de otros autores sobre este tema, como [Fowler96]. Muchas de las cuestiones sutiles de modelado que ellos exploran quedan fuera del alcance de este libro.

Lista de Categorías de Conceptos

La Tabla 26.1 muestra algunos conceptos relevantes que se van a tener en cuenta en esta iteración.

Tabla 26.1. Lista de Categorías de Conceptos.

<i>Categoría</i>	<i>Ejemplos</i>
objetos físicos o tangibles	<i>TarjetaDeCredito, Cheque</i>
especificaciones, diseños o descripciones de cosas	
lugares	
transacciones	<i>PagoEnEfectivo, PagoACredito, PagoConCheque</i>
líneas de la transacción	
roles de la gente	
contenedores de otras cosas	
cosas en un contenedor	
otros sistemas informáticos o electro-mecánicos externos a nuestro sistema	<i>ServicioAutorizacionCredito ServicioAutorizacionCheque</i>
conceptos abstractos	
organizaciones	<i>ServicioAutorizacionCredito ServicioAutorizacionCheque</i>
eventos	
reglas y políticas	
catálogos	
registros de cuentas, trabajo, contratos, cuestiones legales	<i>CuentasPorCobrar</i>
instrumentos financieros y servicios	
manuales, libros	

Identificación de frases nominales

Recordemos que la identificación de frases nominales no se puede aplicar mecánicamente para identificar los conceptos relevantes que se van a incluir en el modelo del dominio. Se debe aplicar el sentido común y desarrollar las abstracciones adecuadas, puesto que el lenguaje natural es ambiguo y los conceptos relevantes no siempre se encuentran de manera explícita o clara en el texto existente. Sin embargo, es una técnica práctica en el modelado del dominio puesto que es directa.

Esta iteración maneja los escenarios del caso de uso *Procesar Venta* para los pagos a crédito y con cheque. A continuación se muestran algunas de las frases nominales identificadas a partir de estas extensiones:

Caso de Uso UC1: Procesar Venta

...

Extensiones:

7b. Pago a crédito:

1. El Cliente introduce la **información de su cuenta de crédito**.
2. El Sistema envía la **peticIÓN de autorización del pago** al Sistema de **Servicio de Autorización de Pago** externo, y solicita la **aprobación del pago**.
- 2a. El Sistema detecta un fallo en la colaboración con el sistema externo:
 1. El Sistema señala el error al Cajero.
 2. El Cajero le pide al Cliente un modo de pago alternativo.
3. El Sistema recibe la **aprobación del pago** y lo notifica al Cajero.
- 3a. El Sistema recibe la **denegación del pago**:
 1. El Sistema señala la denegación al Cajero.
 2. El Cajero le pide al Cliente un modo de pago alternativo.
4. El Sistema registra el **pago a crédito**, que incluye la aprobación del pago.
5. El Sistema presenta el mecanismo de entrada para la firma del pago a crédito.
6. El Cajero le pide al Cliente que firme el pago a crédito. El Cliente introduce la firma.

7c. Pago con cheque:

1. El Cliente escribe un **cheque**, y lo da junto con su **carnet de conducir** al Cajero.
2. El Cajero escribe el número del carnet de conducir en el cheque, lo introduce, y solicita la **autorización del pago con cheque**.
3. Genera una **solicitud del pago con cheque** y lo envía al **Servicio de Autorización de Cheques**.
4. Recibe la aprobación del pago con cheque y notifica la aprobación al Cajero.
5. El sistema registra el **pago con cheque**, que incluye la aprobación del pago.

...

Transacciones del servicio de autorización

La identificación de frases nominales revela conceptos como *SolicitudPagoACredito* y *RespuestaAprobacionCredito*. Éstos en realidad podrían verse como transacciones de los servicios externos, y en general, resulta útil identificar tales transacciones porque las actividades y los procesos tienden a girar alrededor de ellos.

Estas transacciones no tienen que representar registros informáticos o bits que viajan por una línea. Representan la abstracción de la transacción independientemente del modo en el que se ejecute. Por ejemplo, la solicitud de un pago a crédito podría ejecutarse mediante una llamada telefónica o el envío de registros o mensajes entre dos ordenadores, etcétera.

26.2. Generalización

Los conceptos de *PagoEnEfectivo*, *PagoACredito*, y *PagoConCheque* son todos muy parecidos. En esta situación, es posible (y útil¹) organizarlos (como en la Figura 26.1) en una **jerarquía de clases de generalización-especialización** (o simplemente **jerarquía**

¹ Más tarde, en este capítulo, estudiaremos los motivos para definir jerarquías de clases.

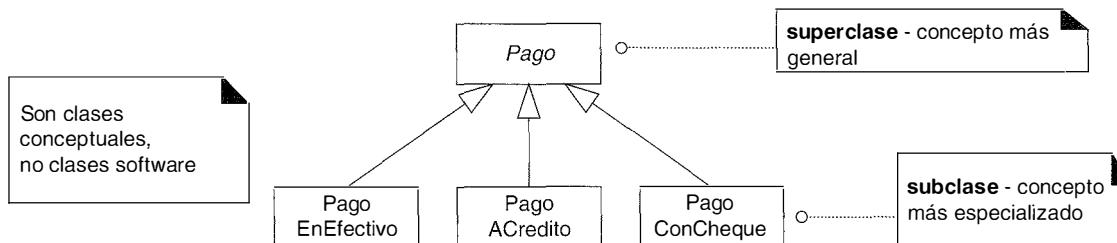


Figura 26.1. Jerarquía de generalización-especialización.

de clases) en la cual, la **superclase** *Pago* representa un concepto más general, y las **subclases** conceptos más especializados.

Nótese que la presentación de las clases en este capítulo se refiere a las clases *conceptuales* y no a las clases software.

La **generalización** es la actividad de identificar elementos comunes entre los conceptos y definir las relaciones de superclase (concepto general) y subclase (concepto especializado). Es una forma de construir clasificaciones taxonómicas entre los conceptos que entonces se representan en jerarquías de clases.

La identificación de una superclase y las subclases es útil en un modelo del dominio porque su presencia nos permite entender los conceptos en términos más generales, refinados y abstractos. Esto nos lleva a una economía de expresión, a mejorar la comprensión y a reducir la información repetida. Y aunque ahora nos estamos centrando en el Modelo del Dominio del UP y no en el Modelo de Diseño del software, el diseño e implementación posterior de la superclase y subclases, como clases software que utilizan la herencia, producirán un mejor software.

Por tanto:

Identifique las superclases y subclases del dominio relevantes para el estudio actual, y represéntelas en el Modelo del Dominio.

Notación UML: Revisemos la notación de la generalización que se introdujo en un capítulo anterior, en UML, la relación de generalización entre elementos se representa con una línea con un triángulo hueco grande en el extremo que apunta a la clase más general desde las más especializadas (ver Figura 26.2). Se puede utilizar tanto un estilo con líneas separadas o unidas.

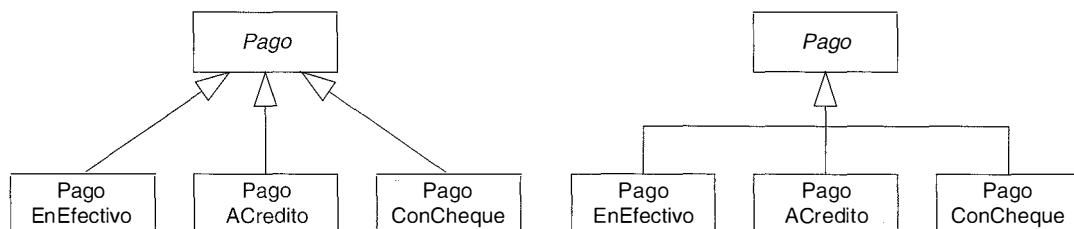


Figura 26.2. Jerarquía de clases con notaciones de líneas separadas o unidas.

26.3. Definición de superclases y subclases conceptuales

Puesto que es útil identificar las super- y subclases conceptuales, es conveniente entender con claridad y precisión la generalización, superclases y subclases en términos de la definición de clase y conjuntos de clases². De esto se ocupan las siguientes secciones.

Generalización y definición de clase conceptual

¿Cuál es la relación de una superclase conceptual con una subclase?

La definición de una superclase conceptual es más general y abarca más que la definición de una subclase.

Por ejemplo, considere la superclase *Pago* y sus subclases (*PagoEnEfectivo*, etcétera). Asuma que la definición del *Pago* representa la transacción de transferir dinero (no necesariamente en efectivo) para una compra desde una parte a otra, y que todos los pagos tienen una cantidad de dinero que es la que transfieren. El modelo que se corresponde a esta definición es el que se muestra en la Figura 26.3.

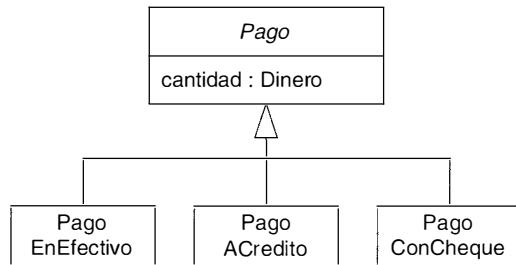


Figura 26.3. Jerarquía de clases del pago.

Un *PagoACredito* es una transferencia de dinero por medio de una institución de crédito que necesita que se autorice. Mi definición de *Pago* abarca más y es más general que mi definición de *PagoACredito*.

Generalización y conjuntos de clases

Las superclases y las subclases conceptuales están relacionadas en términos de pertenencia a un conjunto.

Todos los miembros del conjunto de una subclase conceptual son miembros del conjunto de su superclase.

² Es decir, la intención y extensión de la clase. Esta discusión está inspirada en [MO95].

Por ejemplo, en términos de la pertenencia a un conjunto, todas las instancias del conjunto de *PagoACredito* también son miembros del conjunto de *Pago*. En un diagrama de Venn, se representa como en la Figura 26.4.

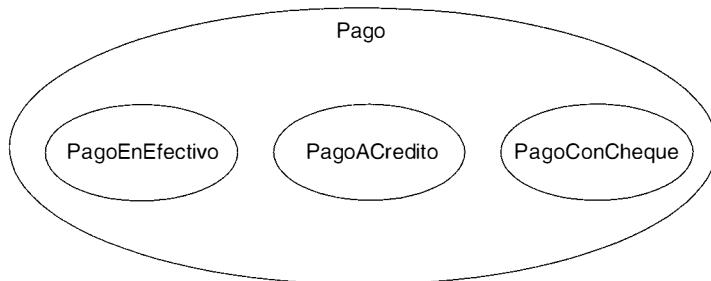


Figura 26.4. Diagrama de Venn de las relaciones de conjuntos.

Conformidad de la definición de la subclase conceptual

Cuando se crea una jerarquía de clases, se hacen declaraciones sobre las superclases que se aplican a las subclases. Por ejemplo, la Figura 26.5 establece que todos los *Pagos* tienen una *cantidad* y que se asocian con una *Venta*.

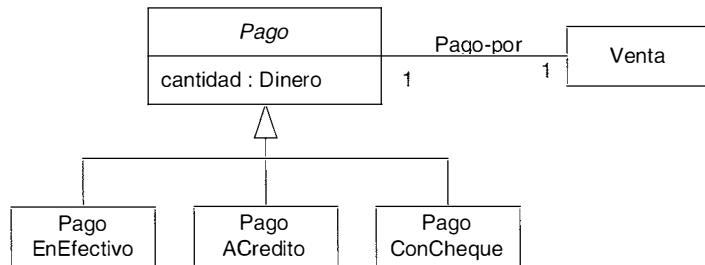


Figura 26.5. Conformidad de las subclases.

Todas las subclases de *Pago* deben ajustarse a tener una cantidad y ser el pago de una *Venta*. En general, esta regla de conformidad con la definición de una superclase es la *Regla del 100%*:

Regla del 100%

El 100% de la definición de la superclase conceptual se debe poder aplicar a la subclase. La subclase debe ajustarse al 100% de los:

- Atributos.
- Asociaciones de la superclase.

Conformidad del conjunto de la subclase conceptual

Una subclase conceptual debería ser un miembro del conjunto de la superclase. Por tanto, un *PagoACredito* debería ser un miembro del conjunto de los *Pagos*.

Informalmente, esto expresa la noción de que la subclase conceptual *es un tipo de* superclase. Un *PagoACredito* *es un tipo de Pago*. De manera más concisa, *es-un-tipo-de* se denomina *es-un*.

Este tipo de conformidad es la *Regla Es-un*:

Regla Es-un:

Todos los miembros del conjunto de una subclase deben ser miembros del conjunto de su superclase.

En lenguaje natural, esto puede comprobarse informalmente formando la sentencia: *Subclase es una Superclase*.

Por ejemplo, la sentencia *PagoACredito es un Pago* tiene sentido, y transmite la noción de conformidad con la pertenencia a un conjunto.

¿Qué es una subclase conceptual correcta?

A partir de la exposición anterior, aplique las siguientes pruebas³ para definir una subclase correcta cuando construya un modelo del dominio:

Una subclase potencial debería estar de acuerdo con:

- La Regla del 100% (conformidad en la definición).
- La Regla Es-un (conformidad con la pertenencia al conjunto).

26.4 Cuándo definir una clase conceptual

Se han presentado las reglas que aseguran que una subclase es correcta (las reglas del 100% y Es-un). Sin embargo, ¿cuándo deberíamos preocuparnos de definir una subclase? En primer lugar, una definición: Una **partición de clases conceptuales** es una división de las clases conceptuales en subclases disjuntas (o **tipos** según la terminología de Odell) [MO95].

La pregunta se podría volver a enunciarse como:

“¿Cuándo es útil representar una partición de clases conceptuales?”

Por ejemplo, en el dominio del PDV, el *Cliente* podría particionarse correctamente en (o dividirse en las subclases) *ClienteHombre* y *ClienteMujer*. Pero, ¿es relevante o útil mostrar esto en nuestro modelo (ver Figura 26.6)?

Esta partición no es útil para nuestro dominio; la siguiente sección explica por qué.

³ Se han elegido estos nombres de las clases por su ayuda nemotécnica en lugar de por precisión.

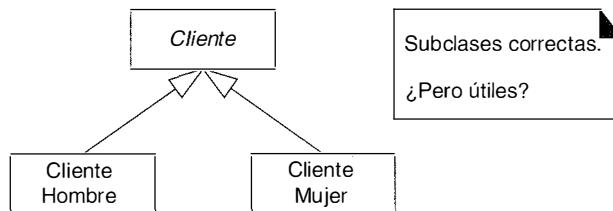


Figura 26.6. Partición de clases conceptuales legales, ¿pero es útil en nuestro dominio?

Razones para particionar una clase conceptual en subclases

A continuación presentamos razones de peso para particionar una clase en subclases:

Cree una subclase conceptual de una superclase cuando:

1. La subclase tiene atributos adicionales de interés.
2. La subclase tiene asociaciones adicionales de interés.
3. El concepto de la subclase funciona, se maneja, reacciona, o se manipula de manera diferente a la superclase o a otras subclases, de alguna manera que es interesante.
4. El concepto de la subclase representa una cosa animada (por ejemplo, animal o robot) que se comporta de manera diferente a la superclase o a otras subclases, de alguna manera que es interesante.

En base al criterio anterior, la partición del *Cliente* en las subclases *ClienteHombre* y *ClienteMujer* no está justificada porque no tienen atributos adicionales o asociaciones, y no se opera sobre ellos (se les trata) de manera diferente, y no se comportan de manera diferente de alguna manera que sea de interés⁴.

La Tabla 26.2 muestra algunos ejemplos de particiones de clases en el dominio de los pagos y otras áreas, utilizando este criterio.

Tabla 26.2. Ejemplo de particiones de subclases.

Motivación de la subclase conceptual	Ejemplos
La subclase tiene atributos adicionales de interés	Pagos: no aplicable Biblioteca: <i>Libro</i> , subclase de <i>RecursoPrestable</i> , tiene un atributo <i>ISBN</i> .
La subclase tiene asociaciones adicionales de interés	Pagos: <i>PagoACredito</i> , subclase de <i>Pago</i> , asociada con una <i>TarjetaDeCredito</i> . Biblioteca: <i>Vídeo</i> , subclase de <i>RecursoPrestable</i> , asociada con un <i>Encargado</i> .

continúa

⁴ Los hombres y las mujeres tienen diferentes hábitos a la hora de comprar. Sin embargo, éstos no son relevantes para los requisitos del caso de uso actual —el criterio que delimita nuestro estudio—.

Tabla 26.2. Ejemplo de particiones de subclases. (Continuación)

<i>Motivación de la subclase conceptual</i>	<i>Ejemplos</i>
El concepto de la subclase funciona, se gestiona, reacciona, o se manipula de manera diferente a la superclase o a otras subclases, de alguna manera que es interesante.	Pagos: <i>PagoACredito</i> , subclase de <i>Pago</i> , se gestiona de manera diferente a otros tipos de pagos en el modo de autorizarlo. Biblioteca: <i>Software</i> , subclase de <i>RecursoPrestable</i> , requiere un depósito antes de que pueda prestarse.
El concepto de la subclase representa una cosa animada (por ejemplo, animal o robot) que se comporta de manera diferente a la superclase o a otras subclases, de alguna manera que es interesante.	Pagos: no aplicable. Biblioteca: no aplicable. Investigación de Mercado: <i>Hombre</i> , subclase de <i>Persona</i> , se comporta de manera diferente a la <i>Mujer</i> con respecto a los hábitos de compras.

26.5. Cuándo definir una superclase conceptual

Normalmente, se aconseja generalizar en una superclase común cuando se identifican elementos comunes entre subclases potenciales. A continuación presentamos los motivos para generalizar y definir una superclase:

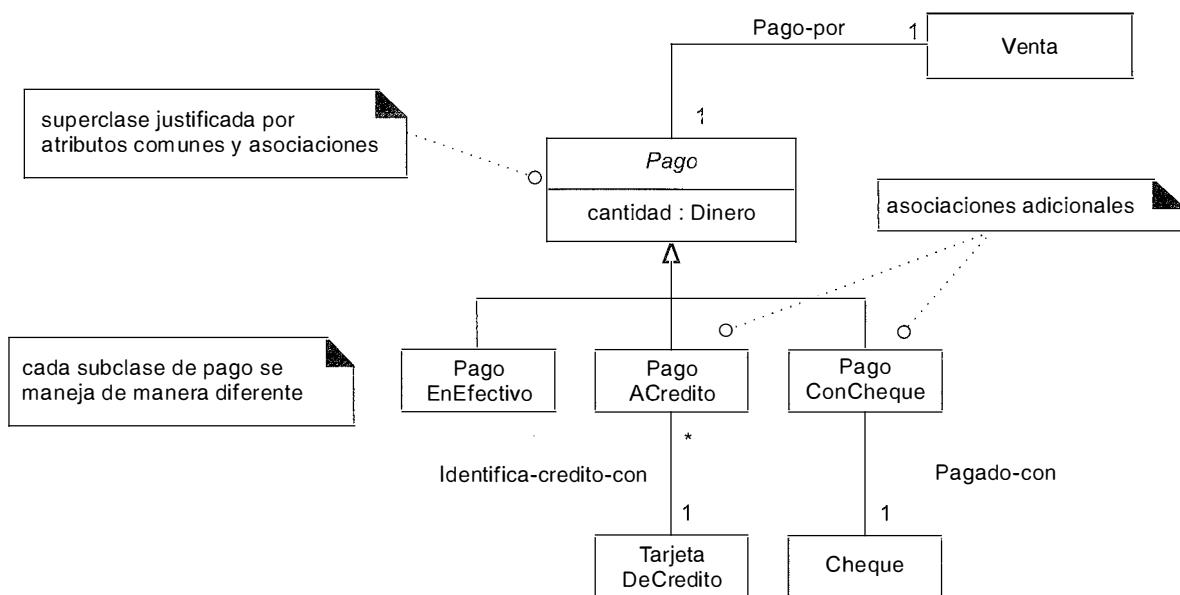
- Cree una superclase conceptual en una relación de generalización de subclases cuando:
- Cuando las subclases potenciales representan variaciones de un concepto similar.
 - Las subclases se ajustarán a las reglas del 100% y Es-un.
 - Todas las subclases tienen el mismo atributo que se puede factorizar y expresar en la superclase.
 - Todas las subclases tienen la misma asociación que se puede factorizar y relacionar con la superclase.

Las siguientes secciones ilustran estos puntos.

26.6. Jerarquías de clases conceptuales del PDV NuevaEra

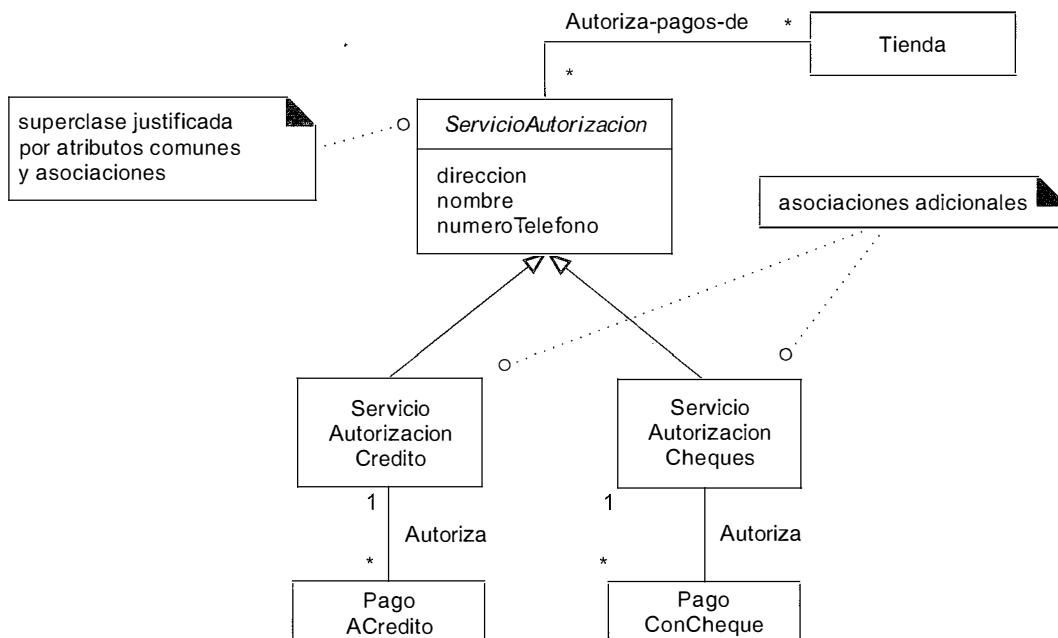
Clases de Pago

Si nos basamos en el criterio anterior para particionar la clase *Pago*, es útil crear una jerarquía de clases de varios tipos de pagos. La justificación de las clases y subclases se muestra en la Figura 26.7.

Figura 26.7. Justificación de las subclases de *Pago*.

Clases de Servicio de Autorización

Los servicios de autorización de crédito y cheques son variaciones de un concepto similar, y tienen atributos comunes de interés. Esto nos lleva a la jerarquía de clases de la Figura 26.8.

Figura 26.8. Justificación de la jerarquía de *ServicioAutorización*.

Clases de Transacción de Autorización

El modelado de distintos tipos de transacciones del servicio de autorización (solicitudes y respuestas) presenta un caso interesante. En general, es útil mostrar las transacciones de los servicios externos en un modelo del dominio porque las actividades y los procesos suelen girar alrededor de ellas. Son conceptos importantes.

¿Debería el modelador representar *todas* las variaciones de una transacción de un servicio externo? Depende. Como se mencionó, los modelos del dominio no son necesariamente correctos o incorrectos, sino más bien, son más o menos útiles. Son útiles, porque cada clase de transacción está relacionada con diferentes conceptos, procesos y reglas del negocio⁵.

Una segunda pregunta interesante es el grado de generalización que es conveniente mostrar en un modelo. Por motivos de la explicación, asumamos que cada clase transacción tiene una fecha y hora. Estos atributos comunes, junto con el deseo de crear una generalización final para esta familia de conceptos relacionados, justifica la creación de *TransaccionAutorizacionPago*.

¿Pero es útil generalizar una respuesta en *RespuestaAutorizacionPagoACredito* y *RespuestaAutorizacionPagoConCheque*, como se muestra en la Figura 26.9, o es suficiente menos generalización, como se muestra en la Figura 26.10?

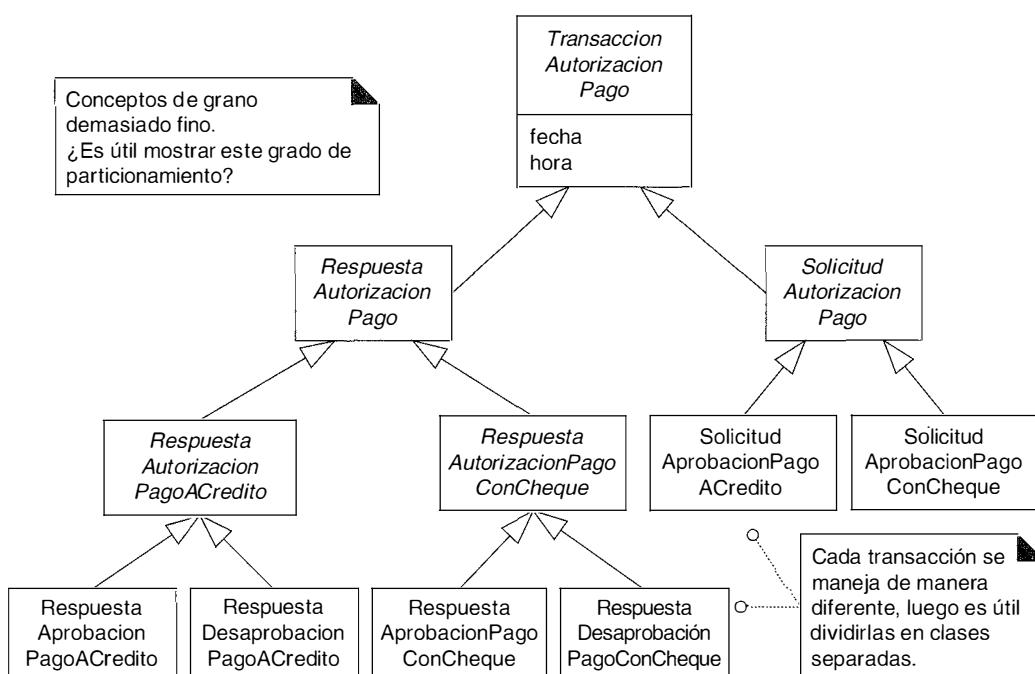


Figura 26.9. Una posible jerarquía de clases para las transacciones con los servicios externos.

⁵ En los modelos del dominio de las telecomunicaciones, es igualmente útil identificar cada tipo de mensaje de intercambio o cambio.

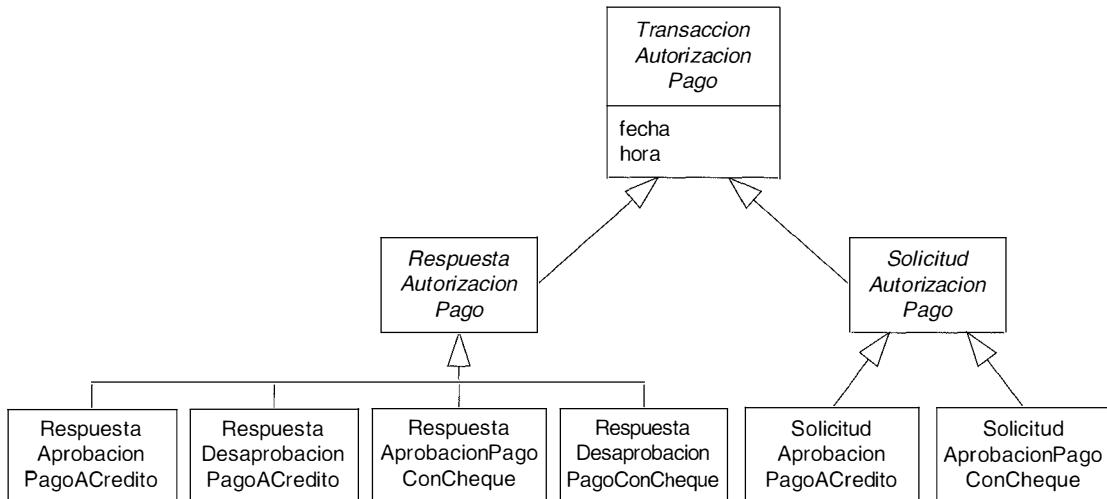


Figura 26.10. Una jerarquía de clases de transacción alternativa.

La jerarquía de clases que se muestra en la Figura 26.10 es suficientemente útil en cuanto a la generalización, porque las generalizaciones adicionales no añaden ningún valor obvio. La jerarquía de la Figura 26.9 expresa una granularidad de generalización más fina que no enriquece de manera significativa nuestra comprensión de los conceptos y reglas del negocio, sino que hace el modelo más complejo —y no es conveniente añadir complejidad a menos que proporcione otros beneficios.

26.7. Clases conceptuales abstractas

Es útil identificar las clases abstractas en el modelo del dominio porque restringe las clases que pueden tener instancias concretas; por tanto, clarifica las reglas del dominio del problema.

Si cada miembro de una clase C debe ser también un miembro de una subclase, entonces la clase C se denomina **clase conceptual abstracta**.

Por ejemplo, asuma que cada instancia de *Pago* debe ser más concretamente una instancia de la subclase *PagoACredito*, *PagoEnEfectivo* o *PagoConCheque*. Esto se ilustra en el diagrama de Venn de la Figura 26.11 (b). Puesto que cada miembro de *Pago* también es un miembro de una de las subclases, el *Pago* es una clase conceptual por definición.

En cambio, si puede haber instancias de *Pago* que no son miembros de una subclase, no es una clase abstracta, como se ilustra en la Figura 26.11 (a).

En el dominio del PDV, cada *Pago* es realmente un miembro de una subclase. La Figura 26.11 (b) es la descripción correcta de los pagos, el *Pago* es una clase conceptual abstracta.

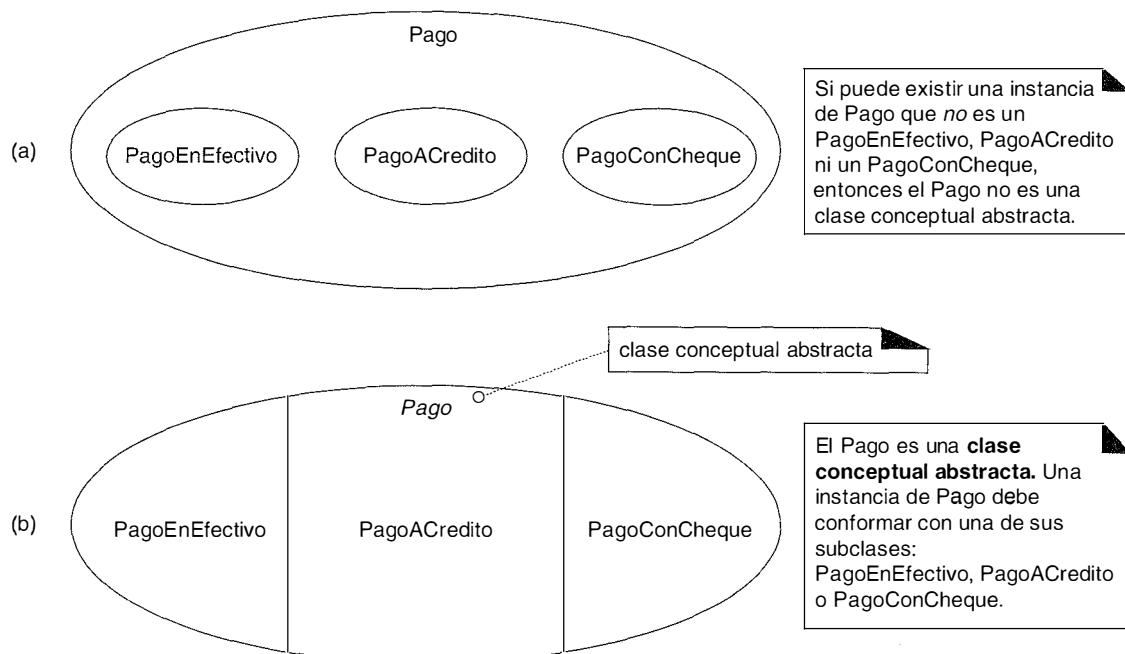


Figura 26.11. Clases conceptuales abstractas.

Notación para las clases abstractas en UML

Recordemos que UML proporciona una notación para representar las clases abstractas —el nombre de la clase en cursiva (ver Figura 26.12)—.

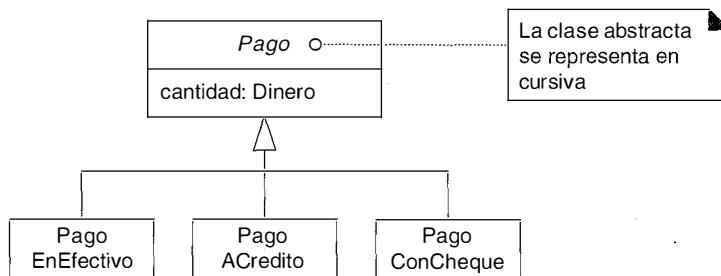


Figura 26.12. Notación para las clases abstractas.

Identifique las clases abstractas y represéntelas con un nombre en cursiva en el Modelo del Dominio.

26.8. Modelado de los cambios de estado

Asuma que un pago puede estar o en estado autorizado o no autorizado, y es significativo mostrarlo en el modelo del dominio (podría no serlo realmente, pero asúmalo para la exposición). Como se muestra en la Figura 26.13, un enfoque de modelado es definir

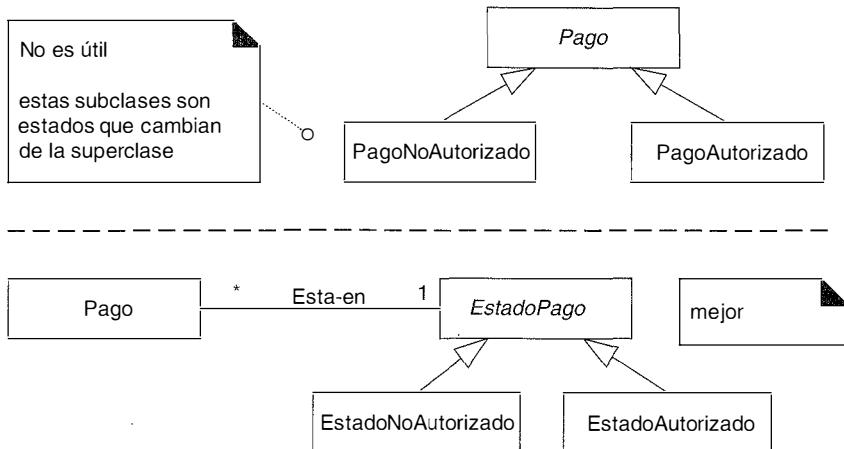


Figura 26.13. Modelado de los cambios de estado.

subclases de *Pago*: *PagoNoAutorizado* y *PagoAutorizado*. Sin embargo, observe que un pago no permanece en uno de estos estados; normalmente cambia de no autorizado a autorizado. Esto nos lleva a la siguiente guía:

No modele el estado de un concepto X como subclases de X. Más bien, o:

- Defina una jerarquía de estados y asocie los estados con X, o
- Ignore la representación de los estados de un concepto en el modelo del dominio; en lugar de eso represente los estados en diagramas de estados.

26.9. Jerarquías de clases y herencia en el software

La explicación de las jerarquías de clases conceptuales no ha mencionado la *herencia* porque la discusión se centra en un modelo del dominio de cosas del mundo, no en artefactos software. En un lenguaje de programación orientado a objetos, una subclase software **hereda** la definición de los atributos y operaciones de sus superclases mediante la creación de **jerarquías de clases software**. La **herencia** es un mecanismo software para hacer que las cosas de la superclase se apliquen a las subclases. Permite factorizar código de las subclases y subirlo arriba de la jerarquía de clases. Por tanto, la herencia no tiene ningún papel real que desempeñar en la discusión del modelo del dominio, aunque lo tiene cuando pasamos al punto de vista del diseño o la implementación.

Las jerarquías de clases conceptuales generadas aquí podrían reflejarse, o no, en el Modelo de Diseño. Por ejemplo, la jerarquía de clases de transacciones de los servicios externos podría reunirse o expandirse en jerarquías de clases alternativas, dependiendo de las características del lenguaje y otros factores. Por ejemplo, las clases plantilla (*template*) de C++ algunas veces pueden reducir el número de clases.

Capítulo 27

REFINAMIENTO DEL MODELO DEL DOMINIO

PRESENTE, n. Aquella parte de la eternidad que divide el dominio de la desilusión del reino de la esperanza.

Ambrose Bierce.

Objetivos

- Añadir clases asociación al Modelo del Dominio.
 - Añadir relaciones de agregación.
 - Modelar los intervalos de tiempo de la información donde es aplicable.
 - Elegir cómo modelar roles.
 - Organizar el Modelo del Dominio en paquetes.
-

Introducción

Este capítulo presenta ideas útiles y notación adicional disponible para el modelado del dominio, y las aplica para refinar aspectos del Modelo del Dominio del PDV NuevaEra.

27.1 Clases asociación

Los siguientes requisitos del dominio disponen el escenario para las clases asociación:

- Los servicios de autorización asignan un ID de comerciante a cada tienda para que se identifique en las comunicaciones.
- Una solicitud de autorización de pago desde la tienda a un servicio de autorización necesita el ID de comerciante que identifica la tienda al servicio.

- Además, una tienda tiene un ID de comerciante distinto para cada servicio.

¿Dónde debería residir el ID de comerciante en el Modelo del Dominio del UP?

Es incorrecto colocar el *comercianteID* en la *Tienda* porque una *Tienda* puede tener más de un valor para el *comercianteID*. Lo mismo ocurre si lo colocamos en el *ServicioAutorizacion* (ver Figura 27.1).

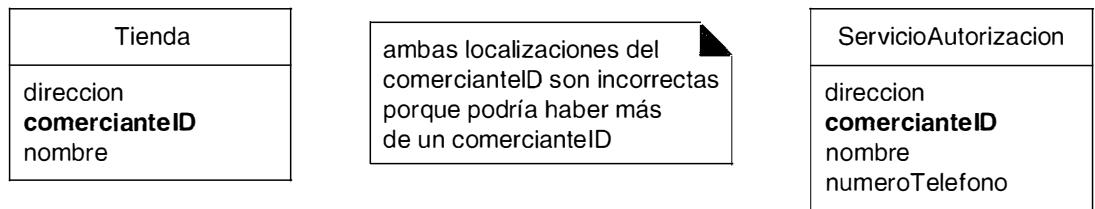


Figura 27.1. Uso inapropiado de un atributo.

Esto nos lleva al siguiente principio de modelado:

En un modelo del dominio, si una clase C puede tener simultáneamente muchos valores para el mismo tipo de atributo A, no coloque el atributo A en C. Coloque el atributo A en otra clase que esté asociada con C.

Por ejemplo:

- Una *Persona* podría tener muchos números de teléfono. Coloque el número de teléfono en otra clase, como *NumeroTelefono* o *InformacionDeContacto*, y asocie muchas de éstas a *Persona*.

El principio anterior sugiere que es más apropiado un modelo como el de la Figura 27.2. En el mundo de los negocios, ¿qué concepto recoge formalmente la información relacionada con los servicios que un servicio proporciona a un cliente? —un *Contrato* o *Cuenta*—.

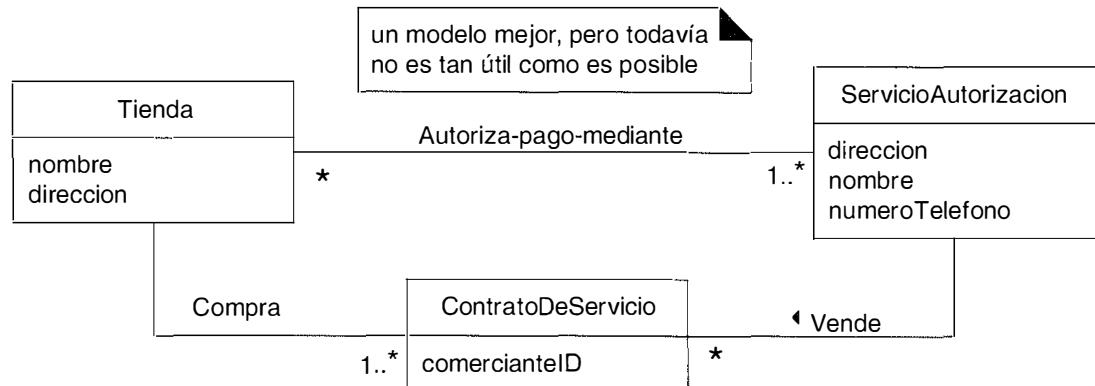


Figura 27.2. Primer intento de modelado del problema del comercianteID.

El hecho de que tanto *Tienda* como *ServicioAutorizacion* estén relacionados con *ContratoDeServicio* es un síntoma de que depende de la relación entre los dos. Se podría pensar en el *comercianteID* como un atributo relacionado con la asociación entre la *Tienda* y el *ServicioAutorizacion*.

Esto nos lleva a la noción de **clase asociación**, en la que podemos añadir características a la propia asociación. Se podría modelar *ContratoDeServicio* como una clase asociación relacionada con la asociación entre *Tienda* y *ServicioAutorizacion*.

En UML, esto se representa con una línea punteada desde la asociación a la clase asociación. La Figura 27.3 transmite visualmente la idea de que un *ContratoDeServicio* y sus atributos están relacionados con la asociación entre una *Tienda* y un *ServicioAutorizacion*, y que el tiempo de vida del *ContratoDeServicio* depende de la relación.

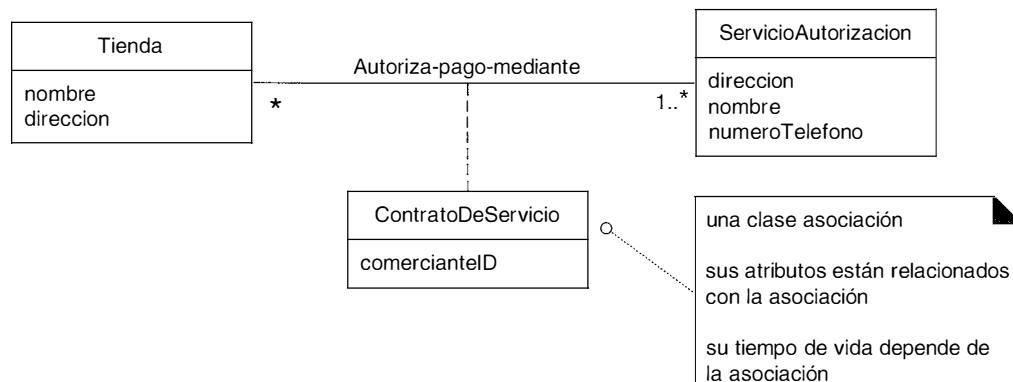


Figura 27.3. Una clase asociación.

Guías

Entre las guías para incluir clases asociaciones se encuentran las siguientes:

Indicios de que podría ser útil una clase asociación en un modelo del dominio:

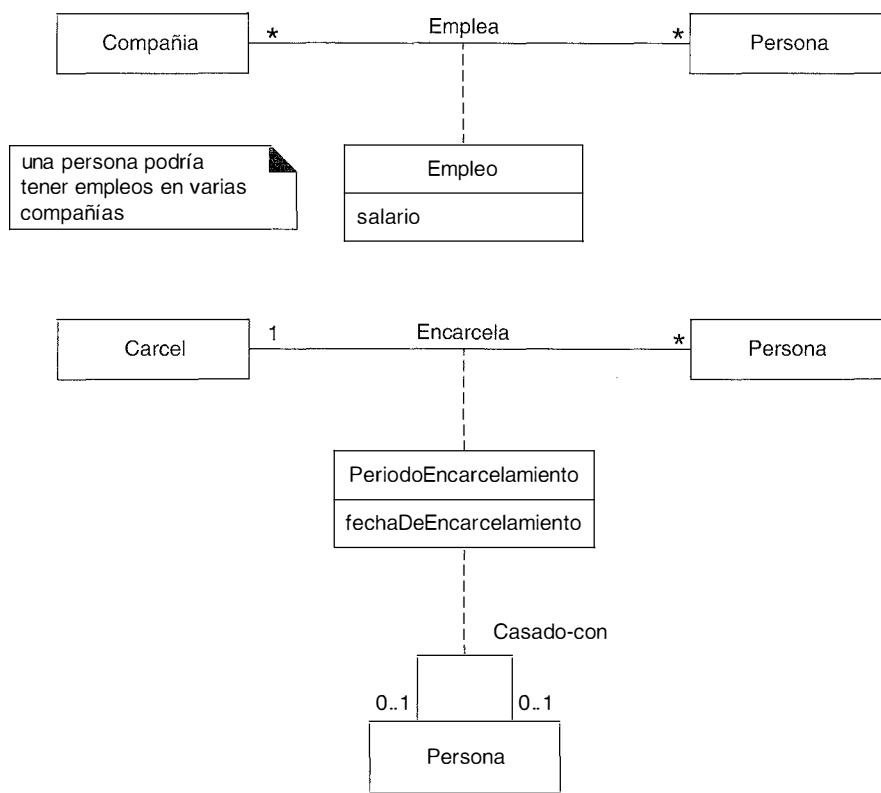
- Un atributo está relacionado con una asociación.
- El tiempo de vida de las instancias de la clase asociación depende de la asociación.
- Existe una asociación muchos-a-muchos entre dos conceptos, e información asociada con la propia asociación.

La presencia de una asociación muchos-a-muchos es un signo típico de que una clase asociación útil está escondida en segundo plano en algún sitio; cuando vea una, tenga en cuenta una clase asociación.

La Figura 27.4 ilustra algunos otros ejemplos de clases asociación.

27.2. Agregación y composición

La **agregación** es un tipo de asociación que se utiliza para modelar las relaciones todo-partido entre las cosas. El todo se denomina **compuesto**.

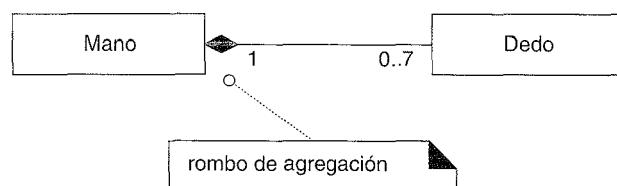
**Figura 27.4.** Clases asociación.

Por ejemplo, los ensamblajes físicos se organizan mediante relaciones de agregación, del mismo modo que una *Mano* agrega *Dedos*.

Agregación en UML

La agregación se representa en UML mediante un rombo hueco o relleno en el extremo del compuesto de una asociación todo-parte (ver Figura 27.5).

La agregación es una propiedad de un rol de asociación¹.

**Figura 27.5.** Notación de la agregación.

¹ Recordemos que cada extremo de asociación es un rol, y que un rol en UML tiene varias propiedades, como *multiplicidad*, *nombre*, *navegabilidad* y *esAgregado*.

El nombre de la asociación se excluye a menudo de las relaciones de agregación puesto que se piensa habitualmente como *Tiene-partes*. Sin embargo, uno puede acostumbrarse a proporcionar más detalles semánticos.

Agregación de composición: rombo relleno

La **agregación de composición**, o **composición**, significa que la parte es un miembro de un único objeto compuesto y que existe una dependencia de existencia y disposición de la parte sobre el compuesto. Por ejemplo, existe una relación de composición entre la mano y un dedo.

En el Modelo de Diseño, la composición y su implicación de dependencia de existencia indica que los objetos software compuestos crean (o provocan la creación de) los objetos software de la parte (por ejemplo, la *Venta* crea los objetos *LíneaDeVenta*).

Pero en el Modelo del Dominio, puesto que no representa objetos software, rara vez es relevante la noción del todo que crea las partes (una venta real no crea una línea de venta real). Sin embargo, existe todavía una analogía. Por ejemplo, en el modelo del dominio de un “cuerpo humano”, uno piensa en la mano que incluye los dedos, luego si uno dice, “Se ha formado una mano”, entendemos que también significa que se han formado los dedos igualmente.

La composición se denota con un rombo relleno. Esto implica que solamente el compuesto posee la parte, y que se encuentran en una jerarquía de partes en forma de árbol; es la forma de agregación más común que se presenta al modelar.

Por ejemplo, un dedo es una parte de una única mano (¡esperamos!), por tanto el rombo de agregación está lleno para indicar agregación de composición (ver Figura 27.6).

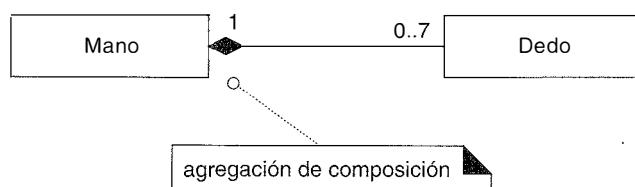


Figura 27.6. Agregación de composición.

Si la multiplicidad en el extremo del compuesto es exactamente uno, la parte *no* podría existir separada de algún compuesto. Por ejemplo, si se quita un dedo de una mano, debe conectarse inmediatamente a otro objeto compuesto (otra mano, pie,...); al menos, eso es lo que está declarando el modelo, ¡independientemente de los méritos médicos de esta idea!

Si la multiplicidad en el extremo del compuesto es *0..1*, entonces podría eliminarse la parte del compuesto, y existir todavía sin pertenecer a ningún otro compuesto. Luego, si quiere que los dedos floten en el aire por sí mismos, utilice *0..1*.

Agregación compartida: rombo hueco

La **agregación compartida** significa que la multiplicidad en el extremo del compuesto podría ser más de uno, y se representa mediante un rombo hueco. Implica que la parte podría estar simultáneamente en muchas instancias del compuesto. La agregación compartida rara vez (si existe alguna) existe en agregaciones físicas, sino más bien en conceptos no físicos.

Por ejemplo, se podría considerar que un paquete UML agrega sus elementos. Pero un elemento podría ser referenciado en más de un paquete (pertenece a un paquete, y es referenciado en otros), lo cual es un ejemplo de agregación compartida (ver Figura 27.7).

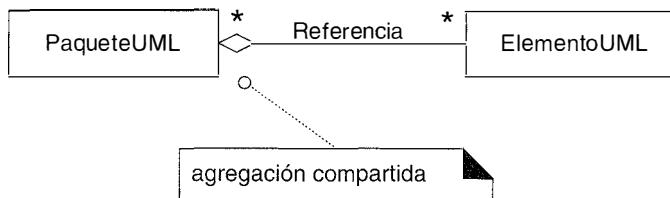


Figura 27.7. Agregación compartida.

Cómo identificar la agregación

En algunos casos, la presencia de la agregación es obvia —normalmente en ensamblajes físicos—. Pero algunas veces, no está claro.

En la agregación: si hay duda, descártela.

A continuación presentamos algunas guías que sugieren cuando mostrar una agregación:

Considere mostrar una agregación cuando:

- El tiempo de vida de la parte está ligado al tiempo de vida del compuesto —existe una dependencia de creación-eliminación de la parte en el todo—.
- Existe un ensamblaje obvio todo-parte físico o lógico.
- Alguna propiedad del compuesto se propaga a las partes, como la ubicación.
- Las operaciones que se aplican sobre el compuesto se propagan a las partes, como la destrucción, movimiento o grabación.

A parte de lo que es un ensamblaje de partes obvio, la siguiente pista más útil es la presencia de una dependencia de creación-eliminación de la parte en el todo.

Un beneficio de la representación de la agregación

Identificar y representar la agregación *no* es excesivamente importante; es bastante posible que se excluya del modelo del dominio. La mayoría —si no todos— los modeladores del dominio con experiencia han visto que se malgasta tiempo improductivo debatiendo los puntos más sutiles de estas asociaciones.

Descubra y muestre las agregaciones si eso le proporciona los siguientes beneficios, la mayoría de ellos se relacionan con el diseño en lugar de con el análisis, que es por lo que no es muy significativa su exclusión del modelo del dominio.

- Aclara las restricciones del dominio en cuanto a la existencia que se desea de la parte independiente del todo. En la agregación de composición, la parte no podría existir fuera del tiempo de vida del todo.
- Durante el trabajo de diseño, esto influye en las dependencias de creación-eliminación entre las clases software del todo y la parte y los elementos de la base de datos (en cuanto a la integridad referencial y los caminos de eliminación en cascada).
- Ayuda en la identificación de un creador (el compuesto) utilizando el patrón GRASP Creador.
- Las operaciones —como la copia y la eliminación— que se aplican al todo a menudo se propagan a las partes.

Agregación en el Modelo del Dominio del PDV

En el dominio del PDV, se podría considerar las instancias *LíneaDeVenta* como parte de una *Venta* compuesta; en general, las líneas de una transacción se ven como parte de una transacción agregada (ver Figura 27.8). Además de ajustarse a ese patrón, existe una dependencia de creación-eliminación de las líneas de venta con la *Venta* —sus tiempos de vida están ligados al tiempo de vida de la *Venta*—.

Por una justificación parecida, el *CatalogoDeProductos* es un agregado de objetos *EspecificacionDelProducto*.

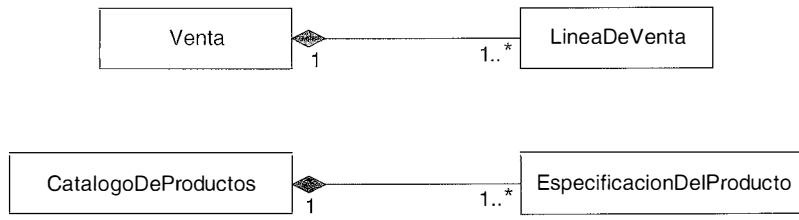


Figura 27.8. Agregación en la aplicación de punto de venta.

Ninguna otra relación es una combinación convincente que sugiera la semántica todo-parte, la dependencia creación-eliminación, y “En caso de duda, descártela”.

27.3. Intervalos de tiempo y precios de los productos: arreglar un “error” de la Iteración 1

En la primera iteración, las instancias de *LíneaDeVenta* se asociaron con las instancias de *EspecificacionDelProducto*, que recogen el precio de un artículo. Esto era una simplificación razonable para las primeras iteraciones, pero necesita que se corrija. Surge la cuestión interesante —y ampliamente aplicable— de los **intervalos de tiempo** asociados con información, contratos y otras cosas por el estilo.

Si una *LíneaDeVenta* siempre recuperaba el precio actual registrado en una *EspecificacionDelProducto*, entonces cuando se cambió el precio en el objeto, las antiguas ventas referenciarían a los nuevos precios, lo cual es incorrecto. Lo que se necesita es distinguir entre el precio histórico de cuando se creó la venta, y el precio actual.

Dependiendo de los requisitos de información, hay al menos dos formas de modelar esto. Una es simplemente copiar el precio del producto en la *LíneaDeVenta* y mantener el precio actual en la *EspecificacionDelProducto*.

El otro enfoque, más robusto, es asociar una colección de objetos *PrecioProducto* con una *EspecificacionDelProducto*, cada uno con un intervalo de tiempo aplicable asociado. Por tanto, la organización puede registrar todos los precios anteriores (para resolver el problema de los precios de venta y para análisis de tendencias) y también registrar los futuros precios previstos (ver Figura 27.9). Diríjase a [CLD99] para una discusión más amplia sobre los intervalos de tiempo, bajo la categoría de arquetipos **Momento-Intervalo**.

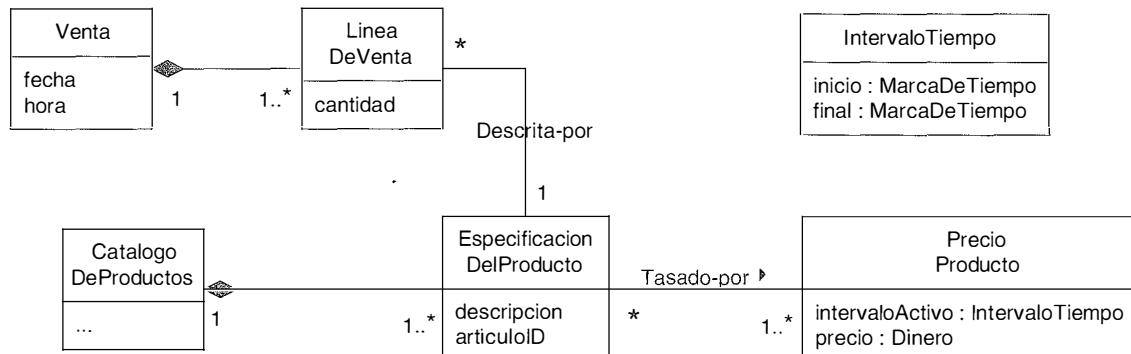


Figura 27.9. PrecioProducto e intervalos de tiempo.

Es habitual que se necesite mantener una colección de información relacionada con intervalos de tiempo, en lugar de un valor simple. Medidas físicas, médicas y científicas, y muchos artefactos de contabilidad y jurídicos, tienen este requisito.

27.4 Nombres de los roles de asociación

Cada extremo de asociación es un rol, que tiene varias propiedades como:

- nombre
- multiplicidad

Un nombre de rol identifica un extremo de una asociación e idealmente describe el papel que juegan los objetos en la asociación. La Figura 27.10 muestra ejemplos de nombres de roles.

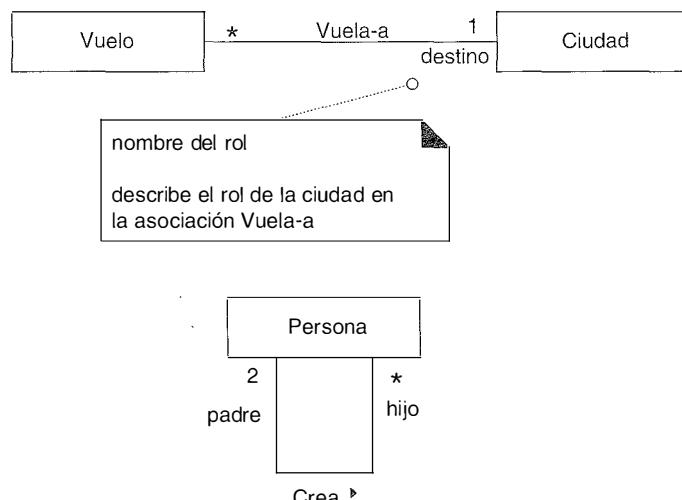


Figura 27.10. Nombres de roles.

No se requiere un nombre de rol explícito —es útil cuando no está claro el rol del objeto—. Normalmente comienza con una letra en minúscula. Si no se muestra explícitamente, asuma que, por defecto, el nombre del rol es igual al nombre de la clase con la que se relaciona, aunque empezando con minúscula.

Como se presentó previamente durante la explicación del paso del diseño al código, los roles utilizados en los DCDs podrían interpretarse como base para los nombres de los atributos durante la generación de código.

27.5. Roles como conceptos vs. roles en asociaciones

En un modelo del dominio, un rol del mundo real —especialmente un rol humano— podría modelarse de varias formas, como un concepto separado, o representado como un rol en una asociación². Por ejemplo, el rol del cajero y el encargado se pueden expresar al menos de las dos formas que se muestran en la Figura 27.11.

El primer enfoque podría llamarse “roles en asociaciones”; el segundo “roles como conceptos”. Ambos enfoques tienen ventajas.

Los roles en asociaciones son atractivos porque son una forma relativamente precisa de expresar que la misma instancia de una persona asume múltiples (y cambiantes dinámicamente) roles en varias asociaciones. Yo, una persona, simultáneamente o sucesivamente, podría asumir el rol de escritor, diseñador de software, padre, etcétera.

² Por simplicidad, se han ignorado otras excelentes soluciones como las que se discuten en [Fowler96].

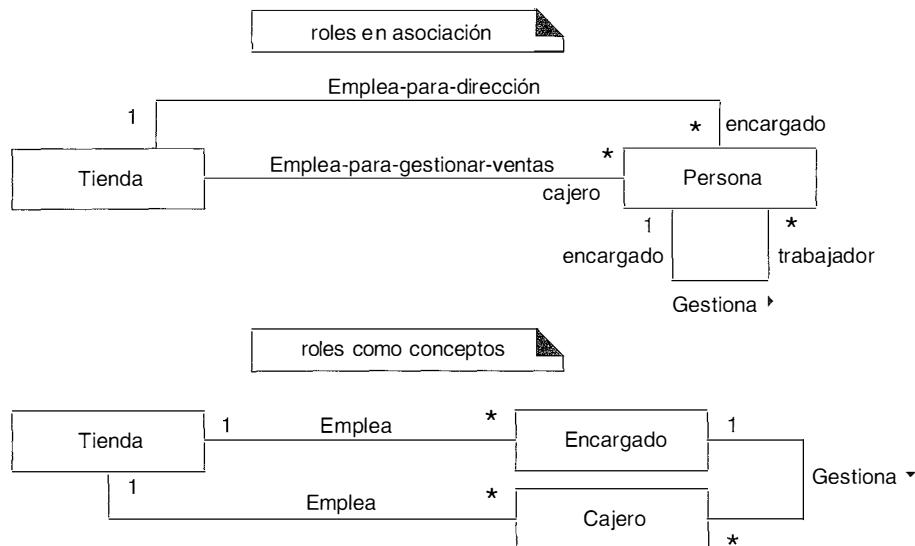


Figura 27.11. Dos formas de modelar los roles humanos.

Por otro lado, los roles como conceptos facilitan y hacen más flexible la inclusión de atributos únicos, asociaciones y semántica adicional. Además, la implementación de los roles como clases separadas es más sencillo debido a las limitaciones de los lenguajes de programación orientados a objetos comerciales actuales —no es conveniente cambiar dinámicamente una instancia de una clase a otra, o añadir dinámicamente comportamiento y atributos cuando cambia el rol de una persona—.

27.6. Elementos derivados

Un elemento derivado puede ser determinado a partir de otros. Los atributos y las asociaciones son los elementos derivados más comunes. ¿Cuándo se deben mostrar los elementos derivados?

Evite mostrar los elementos derivados en un diagrama, puesto que añaden complejidad sin información nueva. Sin embargo, añada un elemento derivado cuando sea un elemento destacado en la terminología, y si se excluye se perjudica la comprensión.

Por ejemplo, el *total* de una *Venta* se puede derivar a partir de la información de los objetos *LíneaDeVenta* y *EspecificaciónDelProducto* (ver Figura 27.12). En UML, se representa anteponiendo una “*T*” al nombre del elemento.

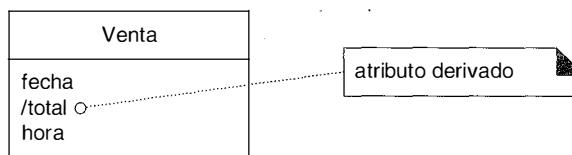


Figura 27.12. Atributo derivado.

Otro ejemplo, la *cantidad* de una *LíneaDeVenta* realmente se puede derivar a partir del número de instancias de *Articulo* asociadas con la línea de venta (ver Figura 27.13).

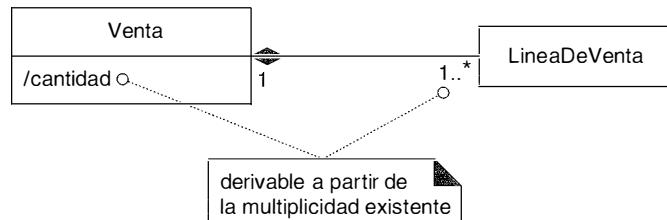


Figura 27.13. Atributo derivado relacionado con la multiplicidad.

27.7. Asociaciones calificadas

En una asociación podría utilizarse un **calificador**; que permite distinguir dentro del conjunto de objetos en el otro extremo de la asociación en base al valor del calificador. Una asociación con un calificador es una **asociación calificada**.

Por ejemplo, podrían distinguirse los objetos *EspecificacionDelProducto* en un *CatalogoDeProductos* según su *articuloID*, como se ilustra en la Figura 27.14 (b). Comparando las figuras (a) y (b) en la Figura 27.14, la calificación reduce la multiplicidad en el extremo más alejado de la asociación, normalmente la disminuye de muchos a uno. La inclusión de un calificador en un modelo del dominio comunica cómo, en el dominio, se distinguen las instancias de una clase en relación con otra clase. No debería utilizarse, en el modelo del dominio, para expresar decisiones de diseño sobre claves de búsqueda, aunque es conveniente en otros diagramas que ilustren decisiones de diseño.

Los calificadores normalmente no añaden nueva información útil convincente, y podemos caer en la trampa de “pensar-diseño”. Sin embargo, utilizados juiciosamente, pueden mejorar el conocimiento sobre el dominio. La asociación calificada entre *CatalogoDeProductos* y *EspecificacionDelProducto* proporciona un ejemplo razonable de un calificador con valor añadido.

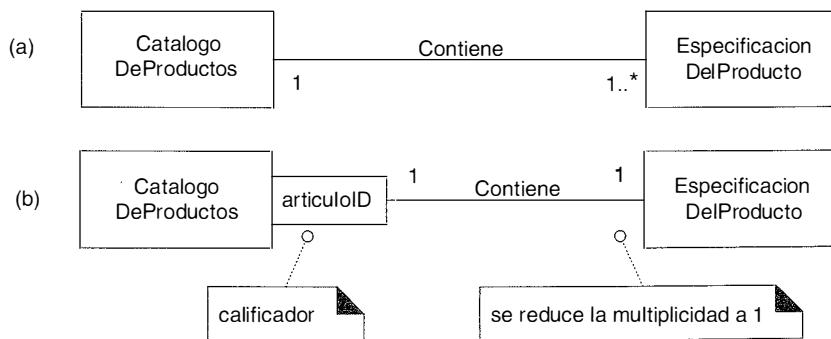


Figura 27.14. Asociación calificada.

27.8. Asociaciones reflexivas

Un concepto podría estar asociado con él mismo; esto se conoce como **asociación reflexiva**³ (ver Figura 27.15).

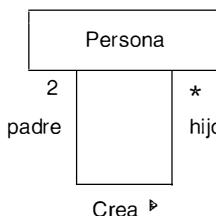


Figura 27.15. Asociación reflexiva.

27.9. Elementos ordenados

Si los objetos asociados están ordenados, esto se puede representar como en la Figura 27.16. Por ejemplo, las instancias de una *LíneaDeVenta* se deben mantener en el orden de entrada.

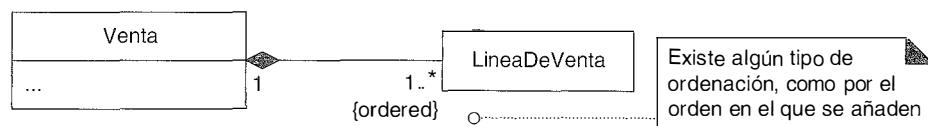


Figura 27.16. Elementos ordenados.

27.10. Utilización de paquetes para organizar el Modelo del Dominio

Un modelo del dominio puede crecer fácilmente y llegar a ser lo suficientemente amplio para que sea conveniente dividirlo en paquetes que incluyen conceptos fuertemente relacionados, esto sirve de ayuda para mejorar la comprensión y para abordar trabajo de análisis en paralelo, en el que diferentes personas realizan el análisis del dominio en diferentes subdominios. Las siguientes secciones ilustran una estructura de paquetes para el Modelo del Dominio del UP.

Notación de paquetes en UML

Recordemos que un paquete en UML se representa mediante una carpeta (ver Figura 27.17). Podrían mostrarse dentro de un paquete otros paquetes subordinados. Si el pa-

³ [MO95] restringe más aún la definición de las asociaciones reflexivas.

que describe sus elementos, el nombre del paquete se coloca en la etiqueta; en otro caso, se centra en la propia carpeta.

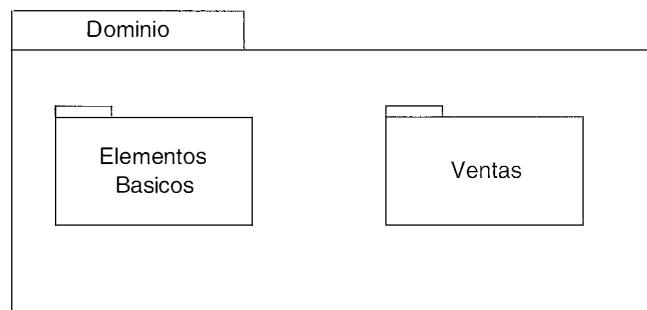


Figura 27.17. Un paquete UML.

Pertenencia y referencias

Un elemento *p pertenece* al paquete donde está definido, pero podría ser referenciado en otros paquetes. En ese caso, el nombre del elemento se califica con el nombre del paquete utilizando el formato del nombre de camino *NombrePaquete::NombreElemento* (ver Figura 27.18). Una clase que se muestra en un paquete que no es al que pertenece se podría modificar con nuevas asociaciones, pero por lo demás permanece sin alterar.

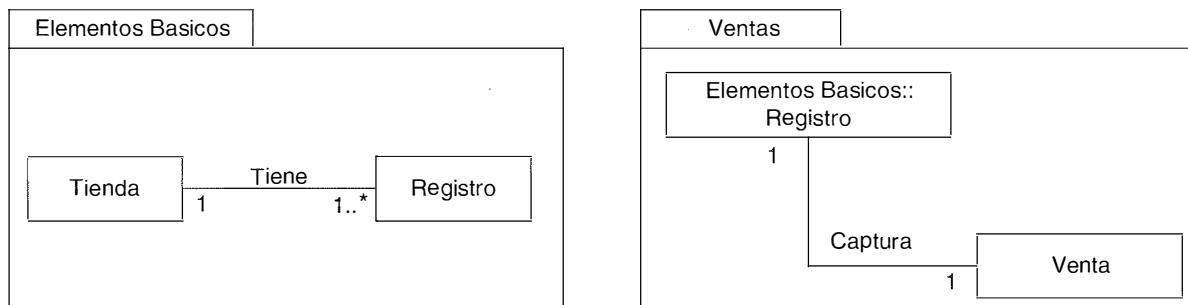


Figura 27.18. Una clase referenciada en un paquete.

Dependencias entre paquetes

Si un elemento del modelo depende de algún modo de otro, se podría representar la dependencia con una relación de dependencia, descrita por una línea con punta de flecha. Una dependencia entre paquetes indica que los elementos del paquete dependiente conocen o están acoplados de algún modo con los elementos del paquete destino.

Por ejemplo, si un paquete referencia a un elemento que pertenece a otro, existe una dependencia. Por tanto, el paquete de *Ventas* tiene una dependencia con el paquete *Elementos_Basicos* (ver Figura 27.19).

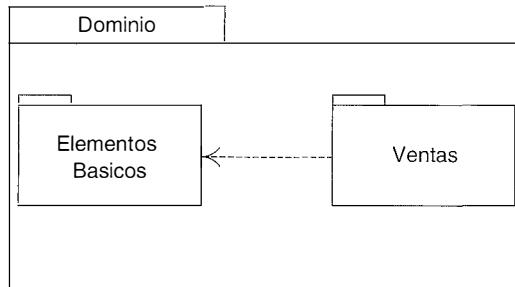


Figura 27.19. Dependencia entre paquetes.

Indicación del paquete sin el diagrama de paquetes

A veces, no es conveniente dibujar un diagrama de paquetes, pero no obstante es deseable indicar el paquete al que pertenecen los elementos.

En esta situación, incluya una nota (un rectángulo con la esquina doblada), como se ilustra en la Figura 27.20.

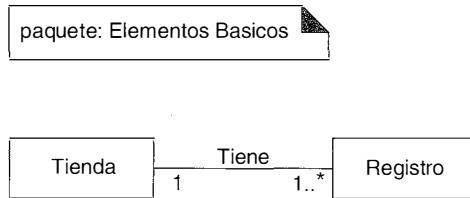


Figura 27.20. Representación de la pertenencia a un paquete con una nota.

Cómo se partitiona el Modelo del Dominio

¿Cómo deberían organizarse en paquetes las clases del modelo del dominio? Aplique las siguientes guías generales:

Para partitionar el modelo del dominio en paquetes, ponga juntos los elementos que:

- se encuentran en el mismo área de interés —estrechamente relacionados por conceptos u objetivos—
- están juntos en una jerarquía de clases
- participan en los mismos casos de uso
- están fuertemente asociados

Resulta útil que todos los elementos relacionados con el modelo del dominio tengan como raíz un paquete denominado *Dominio*, y todos los conceptos básicos, comunes, compartidos, se definen en un paquete que se puede llamar algo así como *Elementos Basicos* o *Conceptos Comunes*, en ausencia de cualquier otro paquete significativo en el que colocarlos.

Paquetes del Modelo del Dominio del PDV

En base al criterio anterior, la organización de paquetes para el Modelo del Dominio del PDV se muestra en la Figura 27.21.

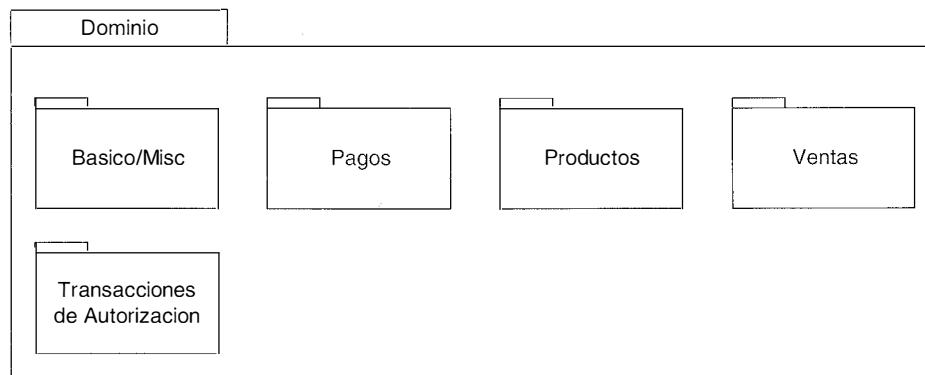


Figura 27.21. Paquetes de conceptos del dominio.

Paquete Basico/Misc

Un paquete Basico/Misc (ver Figura 27.22) es conveniente que contenga conceptos ampliamente compartidos o aquéllos sin una ubicación obvia. En referencias posteriores, se abreviará el nombre del paquete a *Basico*.

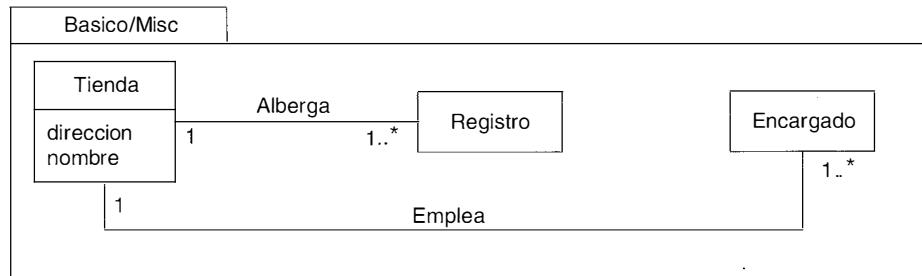


Figura 27.22. Paquete básico.

No existen nuevos conceptos ni asociaciones específicas de esta iteración en este paquete.

Pagos

Como en la iteración 1, sobre todo la consideración del criterio necesito-conocer motiva la aparición de nuevas asociaciones. Por ejemplo, se necesita registrar la relación entre el *PagoACredito* y la *TarjetaDeCredito*. En cambio, se añaden algunas asociaciones más para mejorar la comprensión, como *CarnetConducir Identifica Cliente* (ver Figura 27.23).

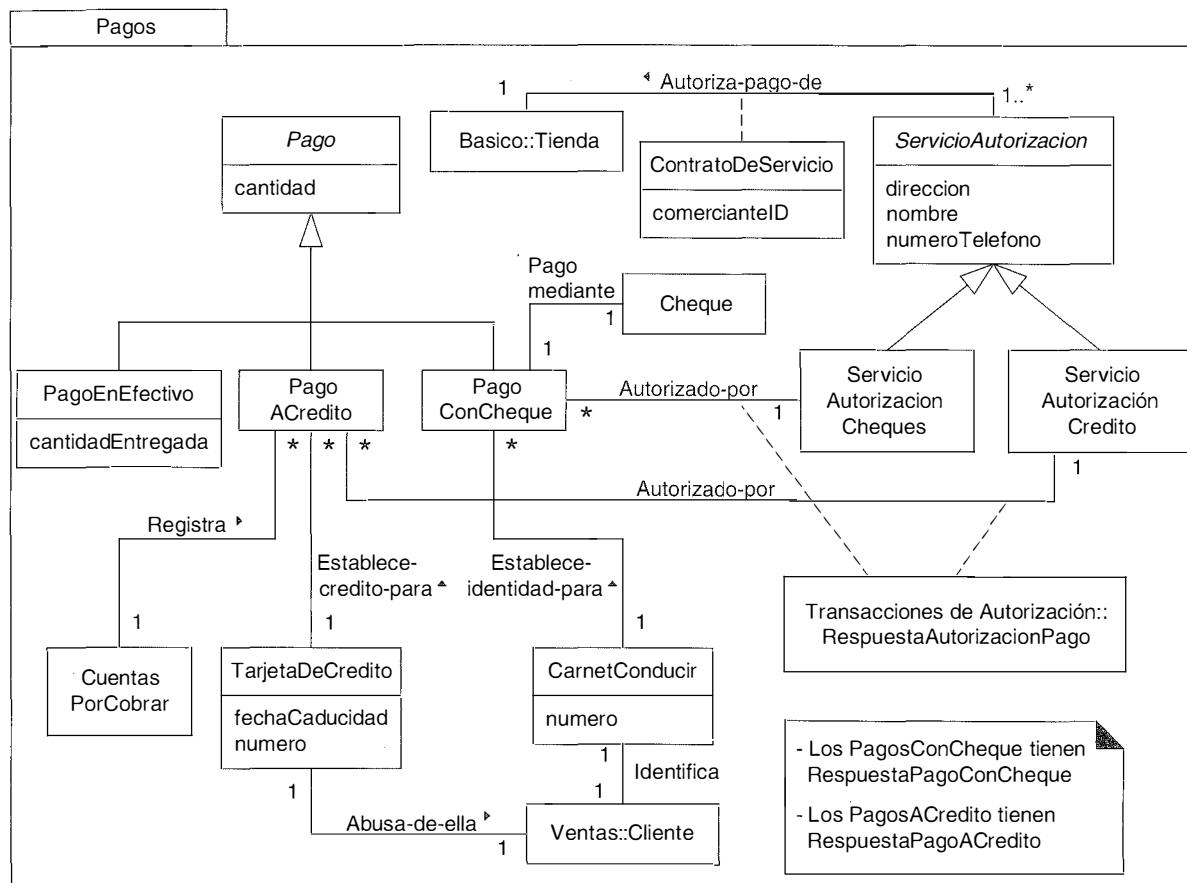


Figura 27.23. Paquete de pagos.

Nótese que la *RespuestaAutorizacionPago* se representa como una clase asociación. Una respuesta surge de la asociación entre un pago y su servicio de autorización.

Productos

Con la excepción de la agregación de composición, no existen nuevos conceptos o asociaciones específicas de esta iteración (ver Figura 27.24).

Ventas

Con la excepción de la agregación de composición y los atributos derivados, no existen nuevos conceptos o asociaciones específicas de esta iteración (ver Figura 27.25).

Transacciones de Autorización

Aunque se recomienda proporcionar nombres significativos a las asociaciones, en algunos casos podría no ser indispensable, especialmente si se considera que el propósito

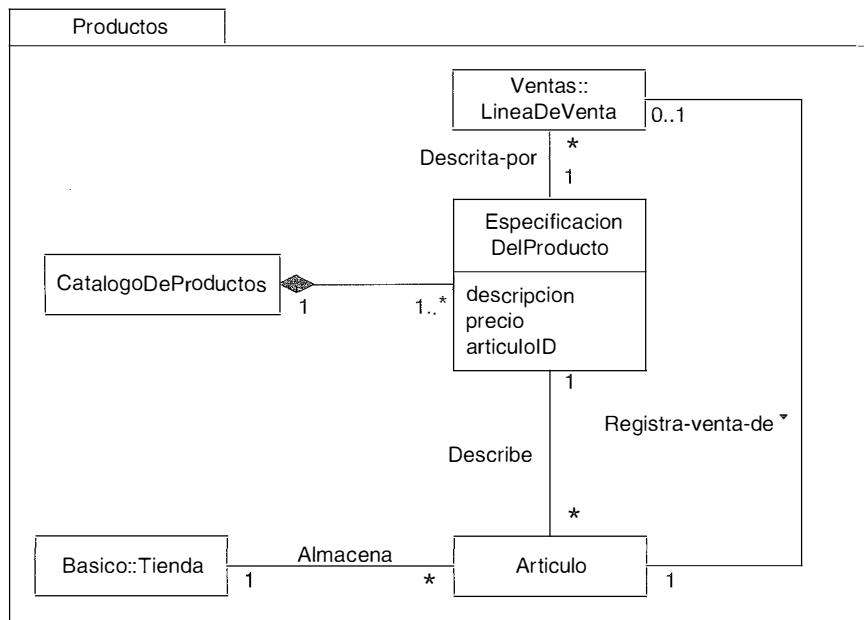


Figura 27.24. Paquete de productos

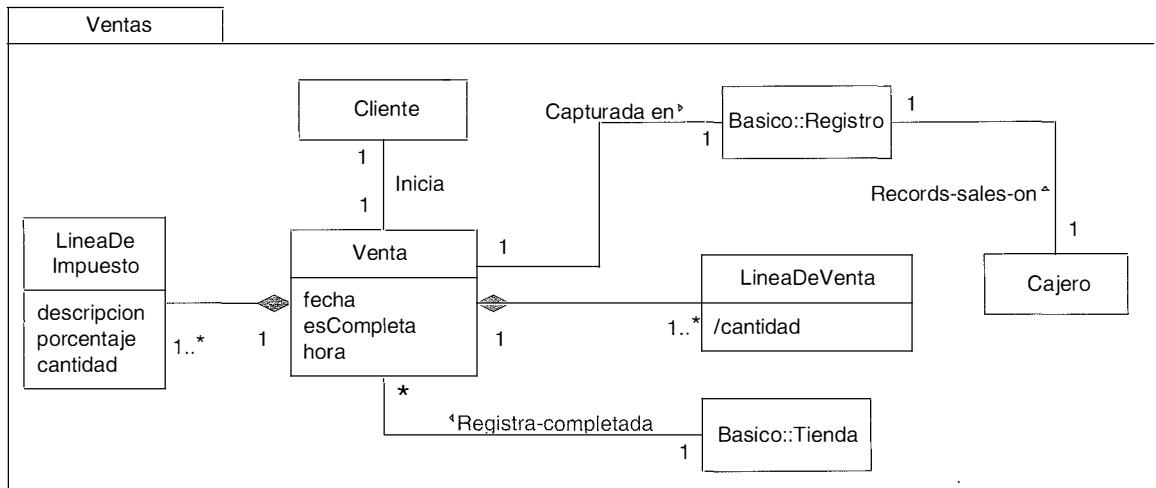


Figura 27.25. Paquete de ventas.

de la asociación es obvio para los destinatarios. Un ejemplo de este caso es la asociación entre los pagos y sus transacciones. Los nombres se han dejado sin especificar porque podemos asumir que las personas que van a leer el diagrama de clases de la Figura 27.26 entenderán que las transacciones son para los pagos; añadir los nombres simplemente enmaraña el diagrama.

¿Es este diagrama demasiado detallado y muestra demasiadas especializaciones? Depende. El verdadero criterio es la utilidad. Aunque no es incorrecto, ¿añade algún valor a mejorar la comprensión del dominio? La respuesta debería influir en el número de especializaciones que se representan en un modelo del dominio.

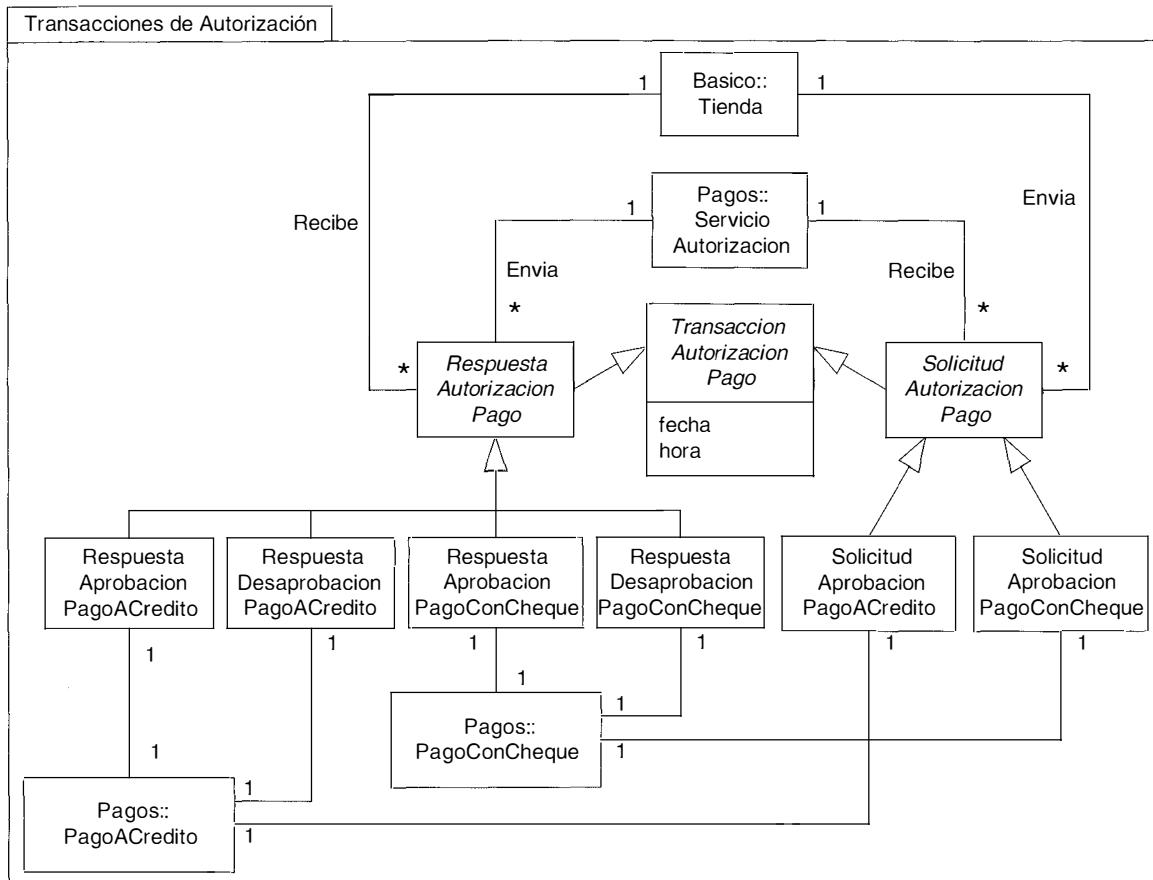


Figura 27.26. Paquete de transacciones de autorización.

Capítulo 28

AÑADIR NUEVOS DSSs Y CONTRATOS

La virtud es una tentación insuficiente.

George Bernard Shaw

Objetivos

- Definir los DSSs y los contratos de las operaciones del sistema para la iteración actual.
-

28.1. Nuevos diagramas de secuencia del sistema

En la iteración actual, los nuevos requisitos para la gestión de los pagos implican nuevas colaboraciones con sistemas externos. Recordemos que los DSSs utilizan la notación de los diagramas de secuencia para representar las colaboraciones entre sistemas, tratando cada sistema como una caja negra. Es conveniente ilustrar los nuevos eventos del sistema mediante DSSs para aclarar:

- Las nuevas operaciones del sistema a las que el sistema de PDV NuevaEra necesitará dar soporte.
- Las solicitudes a otros sistemas, y las respuestas esperadas de estas solicitudes.

Inicio común del escenario Procesar Venta

El DSS para la parte del inicio del escenario básico incluye los eventos del sistema *crearNuevaVenta*, *introducirArticulo*, y *finalizarVenta*; esta parte es común, independientemente del método de pago (ver Figura 28.1).

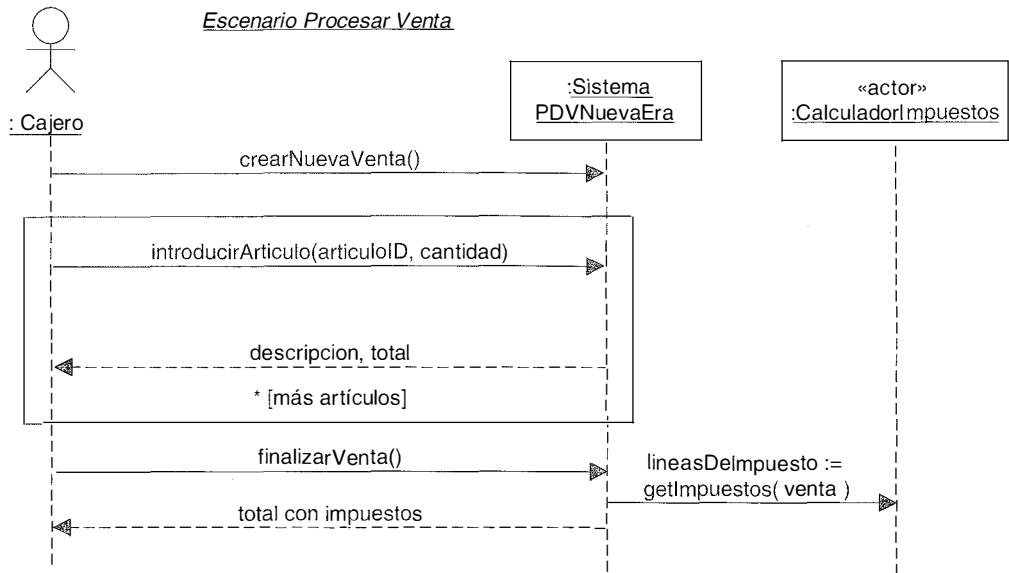


Figura 28.1. DSS del inicio común.

Pago a crédito

Este DSS del escenario de pago a crédito comienza después del inicio común (ver Figura 28.2).

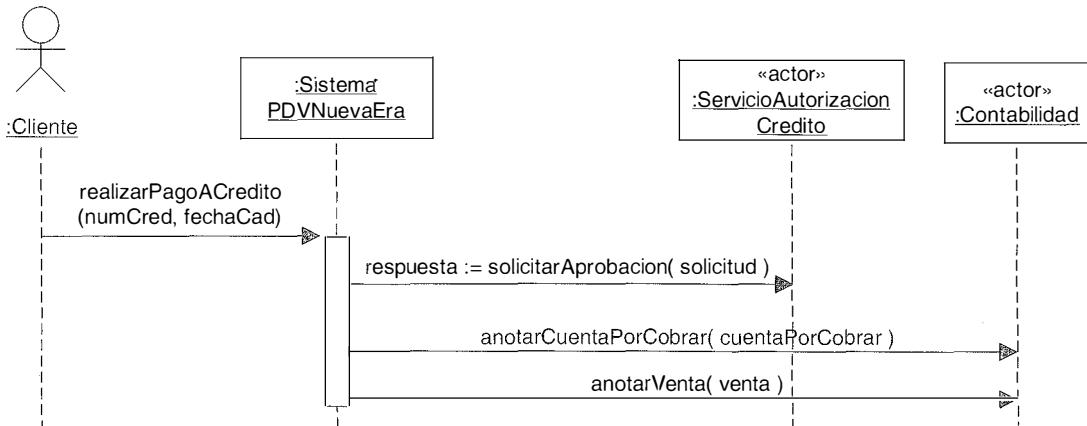


Figura 28.2. DSS del pago a crédito.

Tanto en el pago a crédito como en el pago con cheque, se asume por simplicidad (para esta iteración) que el pago es exactamente igual al total de la venta y, por tanto, no hay que pasar como parámetro una cantidad “entregada” diferente.

Nótese que la solicitud al *ServicioAutorizacionCredito* externo se modela como un mensaje síncrono ordinario con un valor de retorno. Esto es una abstracción; podría im-

plementarse con una solicitud SOAP sobre HTTPS seguros, o con cualquier mecanismo de comunicación remota. Los adaptadores de recursos que se definieron en la iteración anterior ocultarán el protocolo específico.

La operación del sistema *realizarPagoACredito* —y el caso de uso— asume que la información sobre el crédito del cliente procede de una tarjeta de crédito y, por tanto, se introduce en el sistema un número de cuenta de crédito y una fecha de caducidad (probablemente mediante un lector de tarjetas). Aunque admitimos que en el futuro surgirán mecanismos alternativos para comunicar la información acerca del crédito, la suposición del uso de tarjetas de crédito es muy estable.

Recordemos que cuando un servicio de autorización de crédito aprueba un pago a crédito, está en deuda con la tienda por el pago; por tanto, es necesario que se añada en el sistema de contabilidad una entrada de cuenta por cobrar.

Pago con cheque

El DSS para el escenario del pago con cheque se muestra en la Figura 28.3.

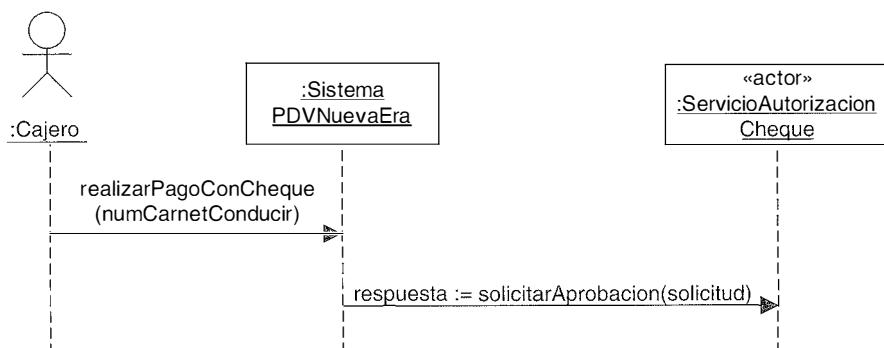


Figura 28.3. DSS del pago con cheque.

De acuerdo con el caso de uso, el cajero debe introducir el número del carnet de conducir para validararlo.

28.2. Nuevas operaciones del sistema

En esta iteración, las nuevas operaciones del sistema que debe gestionar el sistema son:

- *realizarPagoACredito*
- *realizarPagoConCheque*

En la primera iteración, el evento del sistema y la operación para el pago en efectivo era simplemente *realizarPago*. Ahora que los pagos son de diferentes tipos, se renombra como *realizarPagoEnEfectivo*.

28.3. Nuevos contratos de operaciones del sistema

Recordemos que los contratos de las operaciones del sistema son un artefacto de requisitos opcional (parte del Modelo de Casos de Uso) que añade detalles sutiles relativos a una operación del sistema. Algunas veces, el propio texto del caso de uso es suficiente, y no son necesarios estos contratos. Pero en ocasiones, resultan útiles por ser una manera precisa y detallada para identificar lo que ocurre cuando se invoca una operación compleja en el sistema, en términos de cambios de estado de los objetos definidos en el Modelo del Dominio.

A continuación presentamos los contratos de las nuevas operaciones del sistema:

Contrato CO5: realizarPagoACredito

Operación:	realizarPagoACredito(numCtaCredito, fechaCaducidad)
Referencias Cruzadas:	Casos de Uso: Procesar Venta
Precondiciones:	Existe una venta en curso y se han introducido todos los artículos.
Postcondiciones:	<ul style="list-style-type: none"> • Se creó un PagoACredito pg • pg se asoció con la Venta actual vta • se creó una TarjetaDeCredito tc; tc.numero = numCtaCredito, tc.fechaCaducidad = fechaCaducidad • tc se asoció con pg • se creó una SolicitudPagoACredito spc • pg se asoció con spc • se creó una EntradaCuentaPorCobrar ec • ec se asoció con el sistema externo de ContabilidadEntradasPorCobrar • vta se asoció con la Tienda como una venta completa

Obsérvese la postcondición que indica la asociación de una nueva entrada por cobrar en la contabilidad de cuentas por cobrar. Aunque esta responsabilidad está fuera de los límites del sistema NuevaEra, el sistema de contabilidad de cuentas por cobrar se encuentra dentro del control del negocio, por lo que se ha añadido la sentencia como comprobación de corrección.

Por ejemplo, durante las pruebas, está claro a partir de esta postcondición que se debería comprobar que el sistema de contabilidad de cuentas por cobrar contiene una nueva entrada a cobrar.

Contrato CO6: realizarPagoConCheque

Operación:	realizarPagoConCheque(numCarnetConducir)
Referencias Cruzadas:	Casos de Uso: Procesar Venta
Precondiciones:	Existe una venta en curso y se han introducido todos los artículos.
Postcondiciones:	<ul style="list-style-type: none"> • Se creó un PagoConCheque pg • pg se asoció con la Venta actual vta • se creó un CarnetDeConducir cc; cc.numero = numCarnetConducir

Postcondiciones:
(continuación)

- cc se asocia con pg
- se creó una SolicitudPagoConCheque spc
- pg se asocia con spc
- vta se asocia con la Tienda como una venta completa



Capítulo 29

MODELADO DEL COMPORTAMIENTO CON DIAGRAMAS DE ESTADOS

La utilidad es como el oxígeno —nunca lo notas hasta que lo pierdes—.

Anónimo

Objetivos

- Crear diagramas de estados para las clases y los casos de uso.
-

Introducción

UML incluye la notación de los diagramas de estados para representar los eventos y los estados de las cosas —transacciones, casos de uso, personas, etcétera—. En esta introducción se presentan las características más importantes de la notación, pero hay otras que no se cubren.

Se destaca el uso de los diagramas de estados para mostrar los eventos del sistema en los casos de uso, pero podrían aplicarse adicionalmente a cualquier clase.

29.1. Eventos, estados y transiciones

Un **evento** es una ocurrencia significativa o relevante. Por ejemplo:

- Descolgar el teléfono.

Un **estado** es la condición de un objeto en un instante del tiempo —el tiempo entre eventos—. Por ejemplo:

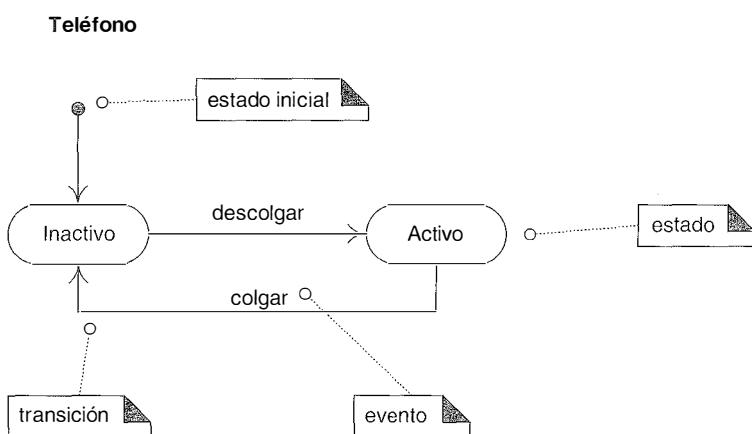
- Un teléfono está en estado “inactivo” después de colgarlo y antes de descolgarlo.

Una **transición** es una relación entre dos estados que indica que cuando tiene lugar un evento, el objeto pasa del estado anterior al estado siguiente. Por ejemplo:

- Cuando tiene lugar el evento de “descolgar”, hay una transición del estado del teléfono de “inactivo” a “activo”.

29.2. Diagramas de estados

Un diagrama de estados UML, como se muestra en la Figura 29.1, representa los eventos y estados interesantes de un objeto, y el comportamiento de un objeto como reacción a un evento. Las transiciones se representan con flechas, etiquetadas con sus eventos. Los estados se representan en rectángulos de esquinas redondeadas. Es habitual incluir un pseudo-estado inicial, que pasa automáticamente a otro estado cuando se crea la instancia.



. Figura 29.1. Diagrama de estados para un teléfono.

Un diagrama de estados muestra el ciclo de vida de un objeto: qué eventos experimenta, sus transiciones y los estados en los que se encuentra entre estos eventos. No es necesario ilustrar todos los posibles eventos; si surge un evento que no está representado en el diagrama, se ignora el evento por lo que al diagrama de estados se refiere. Por tanto, podemos crear un diagrama de estados que describa el ciclo de vida de un objeto a un nivel de detalle arbitrariamente simple o complejo, dependiendo de nuestras necesidades.

Aplicaciones de los diagramas de estados

Un diagrama de estados podría aplicarse a una variedad de elementos UML, entre los que se encuentran:

- Las clases (conceptuales o de software).
- Los casos de uso.

Puesto que un “sistema” completo se podría representar mediante una clase, también podría tener su propio diagrama de estados.

29.3. ¿Diagramas de estados en el UP?

No existe en el UP ningún modelo que se llame “modelo de estados”. Más bien, cualquier elemento de cualquier modelo (Modelo de Diseño, Modelo del Dominio, etcétera) podría tener una máquina de estados para entenderlo mejor o para comunicar su comportamiento dinámico como respuesta a los eventos. Por ejemplo, una máquina de estados asociada a la clase de diseño *Venta* del Modelo de Diseño también forma parte del Modelo de Diseño.

29.4. Diagramas de estados de casos de uso

Una aplicación útil de los diagramas de estados es la descripción de la secuencia legal de eventos del sistema externo que reconoce y maneja un sistema en el contexto de un caso de uso. Por ejemplo:

- Durante el caso de uso *Procesar Venta* en la aplicación del PDV NuevaEra, no es legal llevar a cabo la operación *realizarPagoACredito* hasta que haya tenido lugar el evento *finalizarVenta*.
- Durante el caso de uso de *Procesar Documento* en un procesador de texto, no es legal ejecutar la operación *Guardar-Fichero* hasta que haya tenido lugar el evento *Nuevo-Fichero* o *Abrir-Fichero*.

Un diagrama de estados que describe los eventos del sistema global y sus secuencias en un caso de uso es una especie de **diagrama de estados de casos de uso**. El diagrama de estados de casos de uso de la Figura 29.2 muestra una versión simplificada de los eventos del sistema para el caso de uso *Procesar Venta* en la aplicación del PDV. Ilustra que no es legal generar un evento *realizarPago* si previamente el evento *finalizarVenta* no ha causado la transición del sistema al estado *EsperandoPago*.

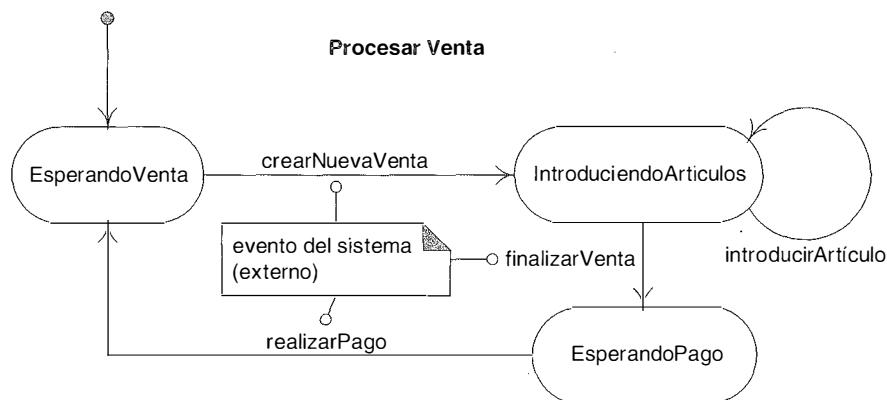


Figura 29.2. Diagrama de estados de caso de uso para *Procesar Venta*.

Utilidad de los diagramas de estados de casos de uso

El número de eventos del sistema y su orden legal para el caso de uso *Procesar Venta* son (hasta el momento) relativamente triviales; por tanto, podría no ser necesario el uso de un

diagrama de estados para mostrar la secuencia válida. Pero para un caso de uso complejo, con innumerables eventos del sistema —como cuando se utiliza un procesador de texto— resulta útil utilizar un diagrama de estados que ilustre el orden válido de los eventos externos.

Veamos cómo: durante el trabajo de diseño e implementación, es necesario crear e implementar un diseño que asegure que no ocurran eventos fuera de la secuencia establecida, de otra manera podría producirse una condición de error. Por ejemplo, no se debería permitir al sistema que reciba un pago a menos que se complete una venta; se debe escribir el código que garantice eso.

Proporcionando un conjunto de diagramas de estados, un diseñador puede desarrollar metódicamente un diseño que asegure el orden correcto de los eventos del sistema. Entre las posibles soluciones de diseño se encuentran:

- estructura condicional en el código para comprobar que los eventos ocurren en el orden correcto
- utilizar el patrón *Estado* (que se presentará en un capítulo posterior)
- deshabilitar los elementos gráficos de las ventanas activas para rechazar los eventos no válidos (un enfoque deseable)
- un intérprete de máquinas de estado que ejecuta una tabla de estados que representa un diagrama de estados de casos de uso.

En un dominio con muchos eventos del sistema, la concisión y minuciosidad de los diagramas de estados de casos de uso ayudan al diseñador a asegurar que no se ha omitido nada.

29.5. Diagramas de estados de casos de uso para la aplicación del PDV

Procesar Venta

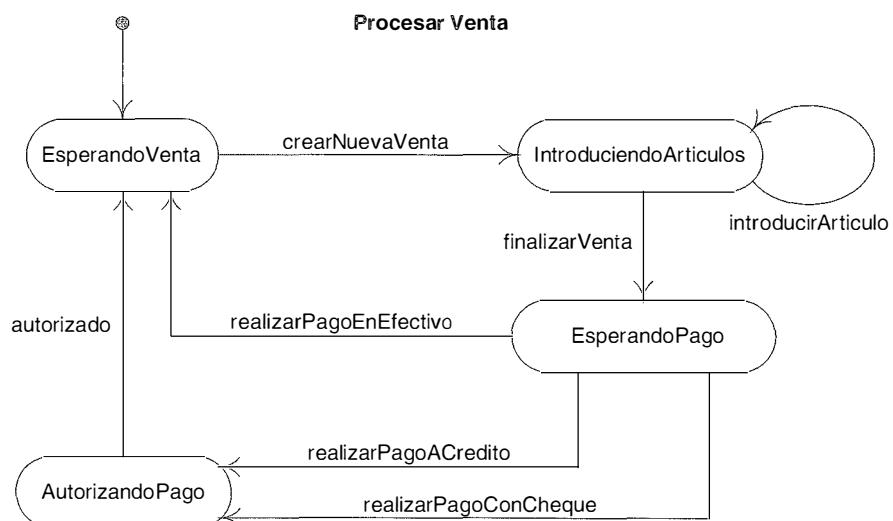


Figura 29.3. Un diagrama de estados de muestra.

29.6. Clases que se benefician de los diagramas de estados

Además de crearse los diagramas de estados para los casos de uso o el sistema global, podrían crearse prácticamente para cualquier tipo o clase.

Objetos dependientes e independientes del estado

Si un objeto siempre responde de la misma manera a un evento, entonces se considera **independiente-del-estado** (o sin modo, *modeless*) con respecto al evento. Por ejemplo, si un objeto recibe un mensaje, y el método que responde siempre hace lo mismo —el método no tendrá normalmente ninguna lógica condicional. El objeto es independiente del estado con respecto al mensaje. Si, para todos los eventos de interés, un objeto siempre reacciona de la misma manera, es un **objeto independiente-del-estado**. En cambio, los **objetos dependientes-del-estado** reaccionan de manera diferente a los eventos dependiendo de su estado.

Cree máquinas de estado para los objetos dependientes del estado con un comportamiento complejo.

En general, en los sistemas de información de gestión las clases realmente dependientes del estado son una minoría. En cambio, los dominios de control de procesos y telecomunicaciones con frecuencia tienen muchos objetos dependientes del estado.

Clases dependientes del estado comunes

A continuación presentamos una lista de objetos comunes que normalmente son dependientes del estado, y para los que podría ser útil crear un diagrama de estados:

- **Casos de uso**
 - Visto como una clase, el caso de uso *Procesar Venta* reacciona de manera diferente al evento *finalizarVenta* dependiendo de si la venta está en curso o no.
- **Sesiones con estado:** son objetos software del lado del servidor que representan sesiones en marcha o conversaciones con un cliente; por ejemplo, los objetos sesión con estado (*stateful*) EJB.
 - Otro ejemplo muy común es la gestión del lado del servidor de aplicaciones web del cliente y la lógica del flujo de la presentación; por ejemplo, un servlet de Java *helper* o “controlador” que recuerda el estado de la sesión con un cliente Web, y controla las transiciones a nuevas páginas web, o las modificaciones en la información que muestra la pagina web actual, basado en el estado de la sesión o la conversación.
 - Una sesión con estado normalmente se suele ver como una clase software que representa un caso de uso. Recordemos que uno de las variantes del patrón GRASP Controlador es un controlador de casos de uso; que es un objeto de sesión con estado de caso de uso.

- **Sistemas:** Es una clase que representa la aplicación o sistema global.
 - El “*sistema de PDV*.”
- **Ventanas**
 - La acción Editar-Pegar sólo es válida si hay algo en el “portapapeles” para pegar.
- **Controladores:** Son objetos controlador GRASP.
 - La clase *Registro*, que maneja los eventos del sistema *introducirArticulo* y *finalizarVenta*.
- **Transacciones:** Las formas en las que reacciona una transacción (una venta, pedido, pago) a un evento a menudo dependen del estado actual en el que se encuentra dentro del ciclo de vida global.
 - Si una *Venta* recibe el mensaje *crearLineaDeVenta* después del evento *finalizarVenta*, debería dar lugar a una condición de error o ignorarlo.
- **Dispositivos**
 - La TV, el microondas: reaccionan de manera diferente a un evento particular dependiendo de su estado actual.
- **Cambiador de Rol:** Son clases que cambian de rol.
 - Una *Persona* que cambia de rol pasa de ser un civil a ser un militar.

29.7. Representación de eventos externos e internos

Tipos de eventos

Es útil clasificar los eventos como sigue:

- **Evento externo:** También conocido como evento del sistema, lo origina algo fuera de los límites del sistema (por ejemplo, un actor). Los DSSs muestran los eventos externos. Los eventos externos relevantes originan la invocación de las operaciones del sistema para responder a ellos.
 - Cuando un cajero presiona el botón “introducir artículo” en un terminal de PDV, ha ocurrido un evento externo.
- **Evento interno:** Causado por algo de dentro de los límites de nuestro sistema. Por lo que se refiere al software, un evento interno surge cuando se invoca a un método por medio de un mensaje o señal que fue enviada desde otro objeto interno. Los mensajes de los diagramas de interacción sugieren eventos internos.
 - Cuando una *Venta* recibe un mensaje *crearLineaDeVenta*, ha ocurrido un evento interno.
- **Evento de tiempo:** Causado por la ocurrencia de una fecha y hora específicas o el paso del tiempo. En cuanto al software, un evento temporal lo dirige un reloj de tiempo real o de tiempo simulado.
 - Suponga que después de que tenga lugar la operación *finalizarVenta*, debe ocurrir la operación *realizarPago* en cinco minutos, en otro caso la venta actual se elimina automáticamente.

Diagramas de estados para eventos internos

Un diagrama de estados puede mostrar eventos *internos* que representan generalmente los mensajes que recibe desde otros objetos. Puesto que los diagramas de interacción también muestran los mensajes y sus reacciones (en función de otros mensajes), ¿por qué utilizar un diagrama de estados para ilustrar eventos internos y el diseño de objetos? El paradigma del diseño de objetos es aquel en el que los objetos colaboran mediante mensajes para llevar a cabo las tareas; los diagramas de interacción UML ilustran de una forma evidente ese paradigma. Es algo incoherente utilizar diagramas de estados para mostrar un diseño del paso de mensajes e interacciones entre objetos¹.

En consecuencia, tengo mis reservas sobre recomendar el uso de diagramas de estados para mostrar los eventos internos con el fin de obtener un diseño de objetos creativo². Sin embargo, podrían ser útiles para resumir los resultados de un diseño, después de que se complete.

En cambio, como explicó la discusión anterior sobre los diagramas de estados de casos de uso, un diagrama de estados podría ser una herramienta útil y concisa para los eventos *externos*.

Es preferible utilizar los diagramas de estados para ilustrar los eventos externos y de tiempo, y las reacciones a ellos, en lugar de utilizarlas para diseñar el comportamiento de los objetos basado en los eventos internos.

29.8. Notación adicional de los diagramas de estados

La notación UML para los diagramas de estados contiene un conjunto rico de características que no se han utilizado en esta introducción. Tres características significativas son:

- Acciones de la transición
- Condiciones de guarda de la transición
- Estados anidados

Acciones y condiciones de guarda de la transición

Una transición puede provocar que se dispare una acción. En una implementación software, esto podría representar la invocación de un método de la clase del diagrama de estados.

¹ Un lector de literatura sobre el A/DOO encontrará ejemplos en publicaciones y libros de texto de diagramas de estados complejos que se dedican a los eventos *internos* y las reacciones de los objetos a ellos. Esencialmente, sus creadores han reemplazado el paradigma de interacción y colaboración de objetos mediante mensajes por el paradigma de objetos como máquinas de estados, y han utilizado diagramas de estados para diseñar el comportamiento de los objetos, en lugar de utilizar diagramas de colaboración. De manera abstracta, las dos visiones son equivalentes.

² Es razonable utilizar los diagramas de estados para mostrar el diseño de objetos basado en los eventos internos, cuando se va a obtener el código con un generador de código que está dirigido por los diagramas de estados, o cuando se utiliza un intérprete de máquinas de estados para ejecutar el sistema software.

Una transición podría tener también una condición de guarda —o condición booleana—. Sólo ocurre la transición si se cumple la condición.

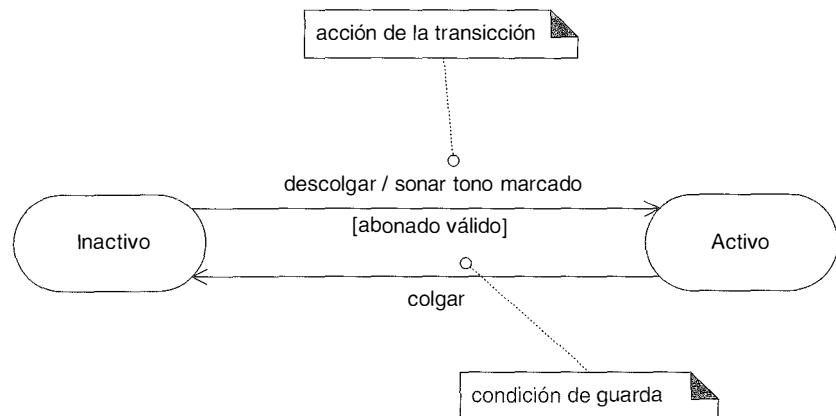


Figura 29.4. Notación para una acción y condición de guarda de una transición.

Estados anidados

Un estado permite el anidamiento para contener subestados; un subestado hereda la transición de su superestado (el estado que lo incluye). Ésta es una contribución clave de la notación de los diagramas de estados de Harel en la que se basa UML, lo que nos lleva a diagramas de estados concisos. Los subestados se podrían mostrar gráficamente anidándolos en una caja que representa el superestado.

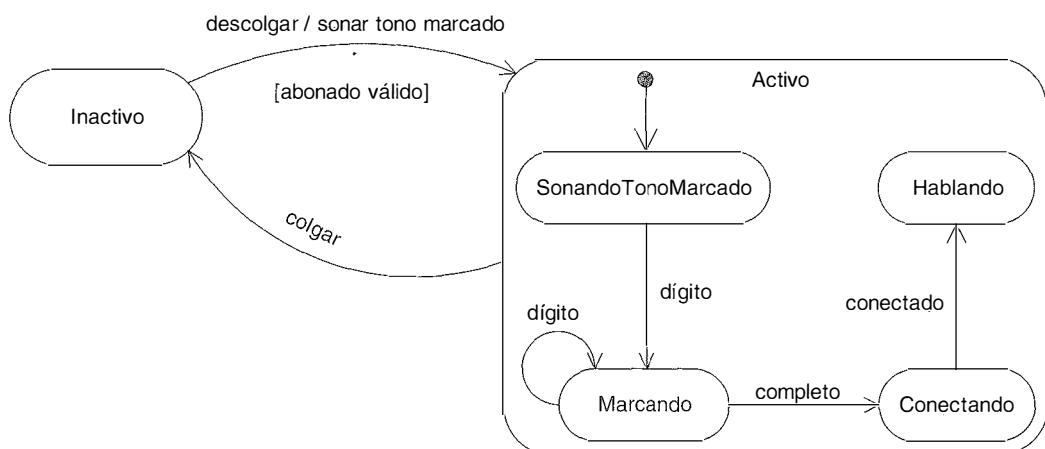


Figura 29.5. Estados anidados.

Por ejemplo, cuando ocurre una transición al estado *Activo*, tiene lugar la creación y transición al subestado *SonandoTonoMarcado*. No importa el subestado en el que se encuentre el objeto, si tiene lugar el evento *colgar* relacionado con el superestado *Activo*, tiene lugar una transición al estado *Inactivo*.

29.9. Lecturas adicionales

La aplicación de los modelos de estados en el A/DOO se cubre bien en *Designing Object Systems* de Cook y Daniels. También *Doing Hard Time* de Douglas proporciona una discusión excelente acerca del modelado de estados; el contenido se centra en los sistemas en tiempo real, pero se puede aplicar en general.

Capítulo 30

DISEÑO DE LA ARQUITECTURA LÓGICA CON PATRONES

0x2B \ ~0x2B

Hamlet

Objetivos

- Diseñar una arquitectura lógica en función de capas y particiones con el patrón Capas (*Layers*).
 - Ilustrar la arquitectura lógica utilizando los diagramas de paquetes de UML.
 - Aplicar los patrones Fachada, Observador y Controlador.
-

Introducción

Lo primero de todo, para fijar las expectativas, decir que este capítulo es una *introducción* al tema de la arquitectura lógica, un tema realmente extenso.

Las iteraciones anteriores se centraron en un grupo de objetos software del “dominio” estrechamente relacionados, en el Modelo del Diseño (como *Venta* y *Pago*). No se prestó atención al interfaz de usuario o al acceso a recursos como una base de datos. El motivo era mantener las cosas simples y centrarse en las técnicas fundamentales del diseño de objetos.

Sin embargo, un sistema típico está compuesto de muchos paquetes lógicos, como un paquete de interfaz de usuario, un paquete de acceso a bases de datos, etcétera. Cada paquete agrupa un conjunto cohesivo de responsabilidades (ej. el acceso a bases de datos). Esta es la práctica básica de aplicar la modularidad para dar soporte a la separación de intereses.

Este capítulo presenta brevemente las arquitecturas lógicas, y la comunicación y acoplamiento entre los paquetes.

30.1. Arquitectura del software

Una definición de arquitectura del software es:

Una arquitectura es el conjunto de decisiones significativas sobre la organización del sistema software, la selección de los elementos estructurales y sus interfaces, con los que se compone el sistema, junto con su comportamiento tal como se especifica en las colaboraciones entre esos elementos, la composición de esos elementos estructurales y de comportamiento en subsistemas progresivamente más amplios, y el estilo de arquitectura que guía esta organización —estos elementos y sus interfaces, sus colaboraciones, y su composición—. [BRJ99]

Independientemente de la definición (y hay muchas) el tema común en todas las definiciones de arquitectura del software es que tiene que ver con la gran escala —las Grandes Ideas en las influencias, organización, estilos, patrones, responsabilidades, colaboraciones, conexiones y motivaciones de un sistema (o un sistema de sistemas), y los subsistemas importantes—.

En el desarrollo de software, arquitectura se considera tanto un nombre como un verbo.

Como nombre, la arquitectura comprende —como indica la definición anterior— la organización y estructura de los elementos importantes del sistema. Más allá de esta definición estática, incluye el comportamiento del sistema, especialmente en función de responsabilidades de gran escala de los sistemas y subsistemas, y sus colaboraciones. En cuanto a una descripción, la arquitectura comprende las *motivaciones* o fundamentos de por qué el sistema está diseñado de la forma que está.

Como verbo, la arquitectura es parte investigación y parte trabajo de diseño; por claridad, el término es mejor que se califique como en investigación arquitectural o diseño arquitectural.

La investigación arquitectural implica la identificación de aquellos requisitos funcionales y no-funcionales que influyen (o deberían influir) de manera significativa en el diseño del sistema, como tendencias del mercado, rendimiento, coste, mantenimiento, y puntos de evolución. Ampliamente, se trata del análisis de requisitos centrado en aquellos requisitos que tienen una influencia especial en las decisiones de diseño del sistema más importantes.

El diseño arquitectural es la resolución de estas influencias y requisitos en el diseño del software, el hardware y las redes, operaciones, políticas, etcétera.

En el UP, el diseño y la investigación de la arquitectura se llaman conjuntamente **análisis arquitectural**, cuyo proceso se introducirán brevemente en el Capítulo 32.

Dimensiones y vistas de la arquitectura en el Proceso Unificado

La arquitectura de un sistema abarca varias dimensiones. Por ejemplo:

- La arquitectura lógica, que describe el sistema en términos de su organización conceptual en capas, paquetes, frameworks importantes, clases, interfaces y subsistemas.

- El despliegue de la arquitectura, que describe el sistema en términos de la asignación de los procesos a unidades de proceso, y la configuración de la red.

El Proceso Unificado sugiere seis vistas de la arquitectura (lógica, despliegue, etcétera), todas ellas se definirán en el Capítulo 32.

Este capítulo se centra en la vista lógica de la arquitectura.

Patrones de arquitectura y categorías de patrones

Existen buenas prácticas bien conocidas en el diseño arquitectural, especialmente en cuanto a la arquitectura lógica a gran escala, y estas prácticas se han escrito en forma de patrones, como el patrón Capas (*Layers*). El primer libro que se dedicó al tema de los patrones de arquitectura fue *Pattern-Oriented Software Architecture* (POSA) [BMRSS96].

El libro POSA también ofrece una clasificación de los patrones sencilla y útil, a diferentes niveles:

1. **Patrones de arquitectura:** Relacionados con el diseño a gran escala y de grano grueso, y que se aplican típicamente durante las primeras iteraciones (la fase de elaboración) cuando se establecen las estructuras y conexiones más importantes.
 - Los patrones Capas, que estructuran el sistema en las principales capas.
2. **Patrones de diseño:** Relacionados con el diseño de los objetos y frameworks de pequeña y mediana escala. Aplicables al diseño de una solución para conectar los elementos de gran escala que se definen mediante los patrones de arquitectura, y durante el trabajo de diseño detallado para cualquier aspecto del diseño local. También se conocen como patrones de micro-arquitectura.
 - El patrón Fachada, que se puede utilizar para proporcionar la interfaz de una capa a la siguiente.
 - El patrón Estrategia, para permitir algoritmos conectables.
3. **Estilos:** Soluciones de diseño de bajo nivel orientadas a la implementación o al lenguaje.
 - El patrón Singleton, para asegurar el acceso global a una única instancia de una clase.

Este capítulo se centra en los patrones de arquitectura y la aplicación de los patrones de diseño para realizar las conexiones entre las estructuras a gran escala.

Existen otras categorías de patrones. Las categorías del POSA forman una nítida tríada, y son útiles para muchos patrones, pero no cubre la gama completa de patrones

que se han publicado. Aun con el riesgo de simplificar demasiado, un patrón es la repetición de las mejores prácticas de lo que funciona —en cualquier dominio—. Otras categorías de patrones publicadas comprenden:

- Patrones del proceso del desarrollo de software y organizacionales.
- Patrones de interfaz de usuario.
- Patrones de pruebas.

30.2. Patrón de arquitectura: Capas (*Layers*)

Solución Las ideas esenciales del patrón Capas [BMRSS96] son simples:

- Organizar la estructura lógica de gran escala de un sistema en capas separadas de responsabilidades distintas y relacionadas, con una separación clara y cohesiva de intereses como que las capas “más bajas” son servicios generales de bajo nivel, y las capas más altas son más específicas de la aplicación.
- La colaboración y el acoplamiento es desde las capas más altas hacia las más bajas; se evita el acoplamiento de las capas más bajas a las más altas.

Una capa es un elemento de gran escala, a menudo compuesto de varios paquetes o subsistemas.

El patrón Capas se relaciona con la arquitectura lógica; es decir, describe la organización conceptual de los elementos del diseño en grupos, independiente de su empaquetamiento o despliegue físico.

Las Capas definen un modelo general de N-niveles para la arquitectura lógica; produce una **arquitectura en capas**. Se lleva tanto tiempo aplicando y escribiendo sobre esto como un patrón que en *Pattern Almanac 2000* [Rising00] se listan alrededor de 100 patrones que son variantes o están relacionados con el patrón Capas.

- Problemas**
- Los cambios del código fuente se propagan a lo largo de todo el sistema —muchas partes del sistema están altamente acopladas—.
 - La lógica de la aplicación se entrelaza con la interfaz de usuario, entonces no se puede reutilizar con una interfaz diferente, ni distribuirse a otro nodo de proceso.
 - La lógica más específica de la aplicación se entrelaza con los servicios técnicos o la lógica del negocio potencialmente generales, entonces no se puede reutilizar, distribuir a otro nodo o reemplazar fácilmente con una implementación diferente.
 - Existe un alto acoplamiento entre diferentes áreas de interés. Esto es por lo que es difícil dividir el trabajo para diferentes desarrolladores mediante límites claros.
 - Debido al alto acoplamiento y la mezcla de intereses, es difícil que la funcionalidad evolucione, que el sistema crezca de forma proporcionada o que se actualice para utilizar nuevas tecnologías.

Ejemplo El objetivo y número de las capas varía de una aplicación a otra y entre dominios de aplicación (sistemas de información, sistemas operativos, etcétera). Aplicado a los sistemas de información, las capas típicas se ilustran y explican en la Figura 30.1.

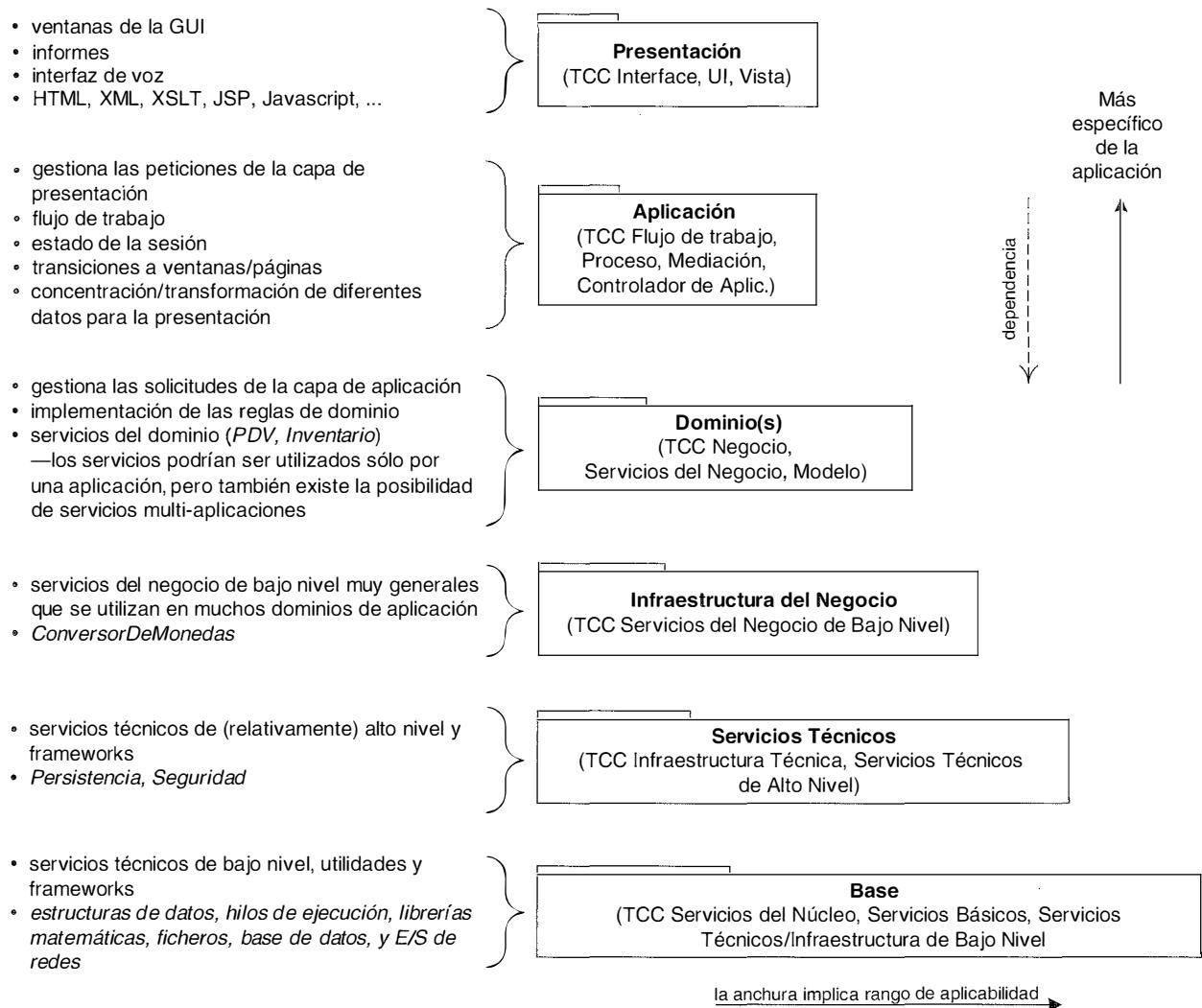


Figura 30.1. Capas comunes en una arquitectura lógica de un sistema de información¹.

Basado en estos arquetipos, la Figura 30.2 ilustra una arquitectura lógica en capas parcial de la aplicación NuevaEra.

Notación UML: Los diagramas de paquetes se utilizan para representar las capas. En UML una capa es simplemente un paquete.

Obsérvese que para esta iteración del diseño no existe una capa de Aplicación; como se discutirá después, no siempre es necesario.

Puesto que esto es un desarrollo iterativo, es normal crear un diseño de capas que comience siendo simple y evolucione a lo largo de las iteraciones de la fase de elaboración.

¹ La anchura del paquete se utiliza para comunicar el rango de aplicabilidad en este diagrama, pero no es una práctica general en UML. TCC significa “también conocido como”.

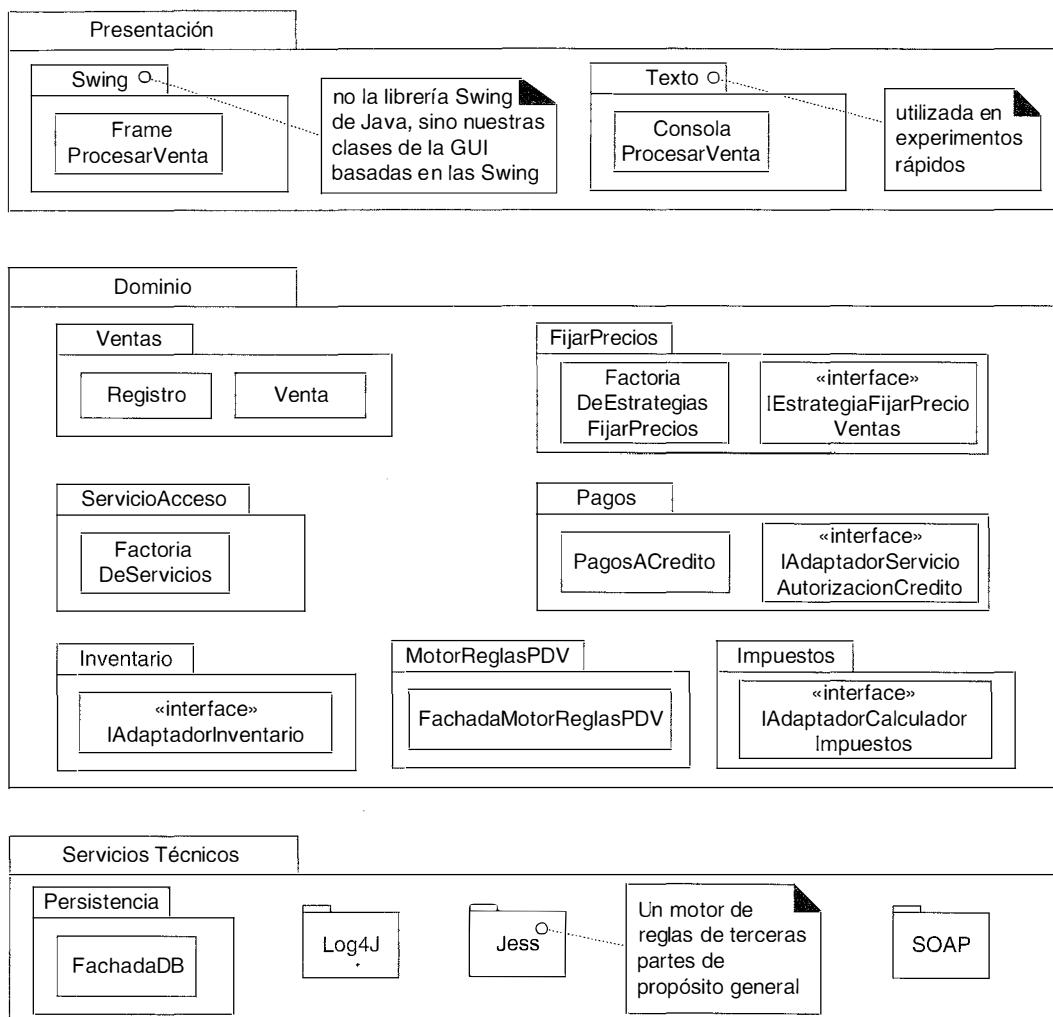


Figura 30.2. Vista lógica parcial de las capas en la aplicación NuevaEra.

ración. Uno de los objetivos de esta fase es establecer la arquitectura básica (diseñarla e implementarla) al final de las iteraciones de la elaboración, pero esto no significa realizar un diseño especulativo detallado de la arquitectura por adelantado, antes de empezar a programar. Más bien, se diseña una arquitectura lógica tentativa en las primeras iteraciones, que evolucionará incrementalmente a lo largo de la fase de elaboración.

Obsérvese que en este diagrama de paquetes sólo se presentan unos pocos tipos de elementos como muestra; esto no sólo está motivado por las limitaciones de espacio al dar formato al libro, sino que es una señal de calidad de un diagrama de la **vista de la arquitectura** —sólo muestra unos pocos elementos relevantes para transmitir de manera concisa las ideas importantes de los aspectos más significativos de la arquitectura—. La idea de un documento de la vista de la arquitectura del UP es decirle al lector, “He elegido este pequeño conjunto de elementos instructivos para transmitir las grandes ideas”.

Comentarios acerca del diagrama:

- Existen otros tipos en estos paquetes; sólo se muestran unos pocos para indicar los aspectos relevantes.
- No se mostró en esta vista la capa Base; el arquitecto (yo) decidió que no añadía información interesante, aunque el equipo de desarrollo, con seguridad, añadirá algunas clases Base, como utilidades avanzadas para la manipulación de *Strings*.
- Hasta el momento, no se utiliza una capa de Aplicación separada. Las responsabilidades de control o los objetos de sesión de la capa de aplicación las maneja el objeto *Registro*. El arquitecto añadirá una capa de Aplicación en una iteración posterior cuando crezca la complejidad del comportamiento y se introduzcan interfaces alternativas para los clientes (como un navegador web y un PDA portátil de red sin cable).

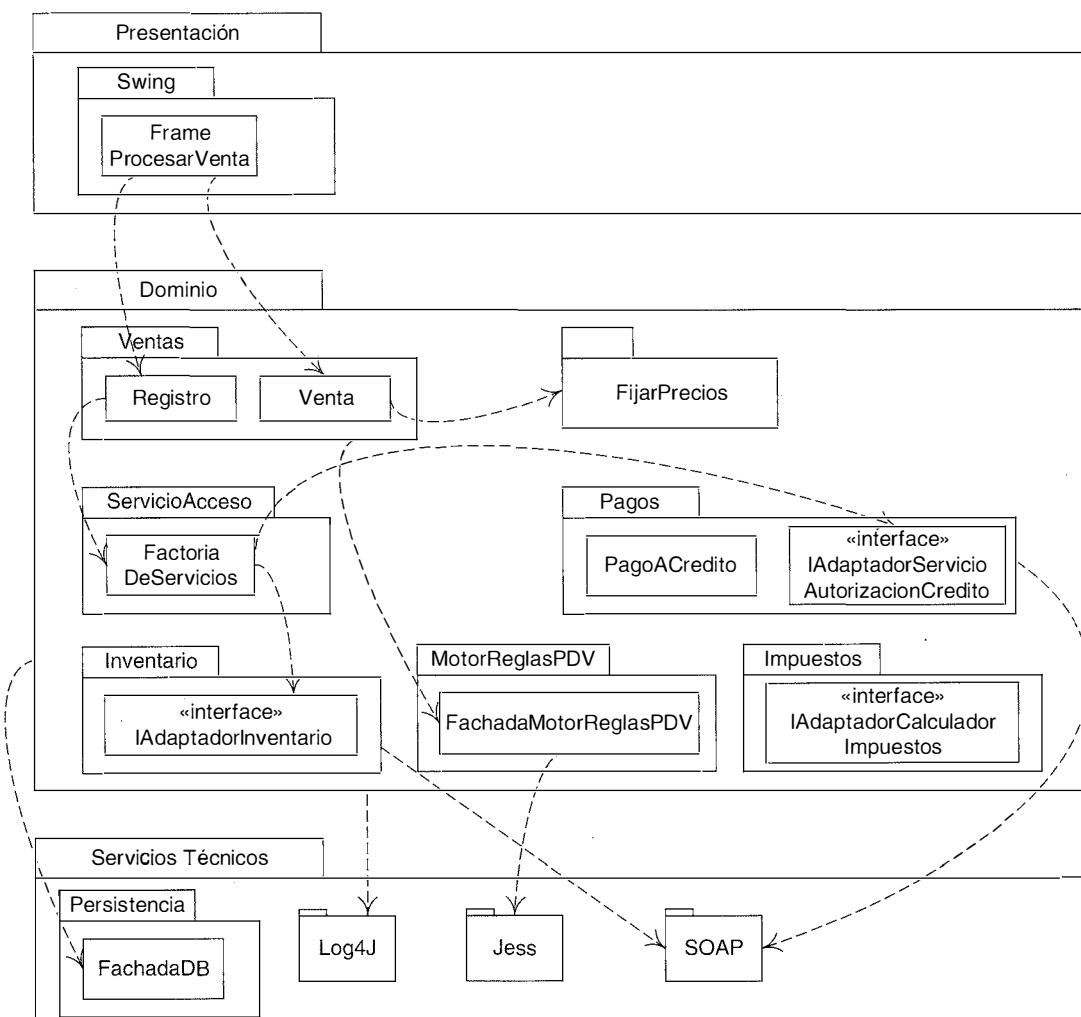


Figura 30.3. Acoplamiento parcial entre paquetes.

Acoplamiento entre capas y entre paquetes

También proporciona información la inclusión de un diagrama en la vista lógica que muestre el acoplamiento relevante entre las capas y los paquetes. La Figura 30.3 ilustra un ejemplo parcial.

Notación UML:

- Obsérvese que se pueden utilizar líneas de dependencia para mostrar el acoplamiento entre los paquetes o los tipos de los paquetes. Es conveniente utilizar simples líneas de dependencia cuando al comunicador no le preocupa especificar la dependencia exacta (visibilidad de atributo, subclase,...), sino simplemente quiere resaltar las dependencias.
- Nótese también el uso de una línea de dependencia que sale de un paquete en lugar de desde un tipo específico, como desde el paquete *Ventas* a la clase *FachadaMotorReglasPDV*, y del paquete del *Dominio* al paquete *Log4J*. Esto es útil cuando o bien no es interesante el tipo concreto del que depende, o bien el comunicador quiere dar a entender que podrían compartir la dependencia muchos elementos del paquete.

Otro uso común del diagrama de paquetes es ocultar los tipos específicos, y centrarse en ilustrar el acoplamiento paquete-paquete, como en el diagrama parcial de la Figura 30.4.

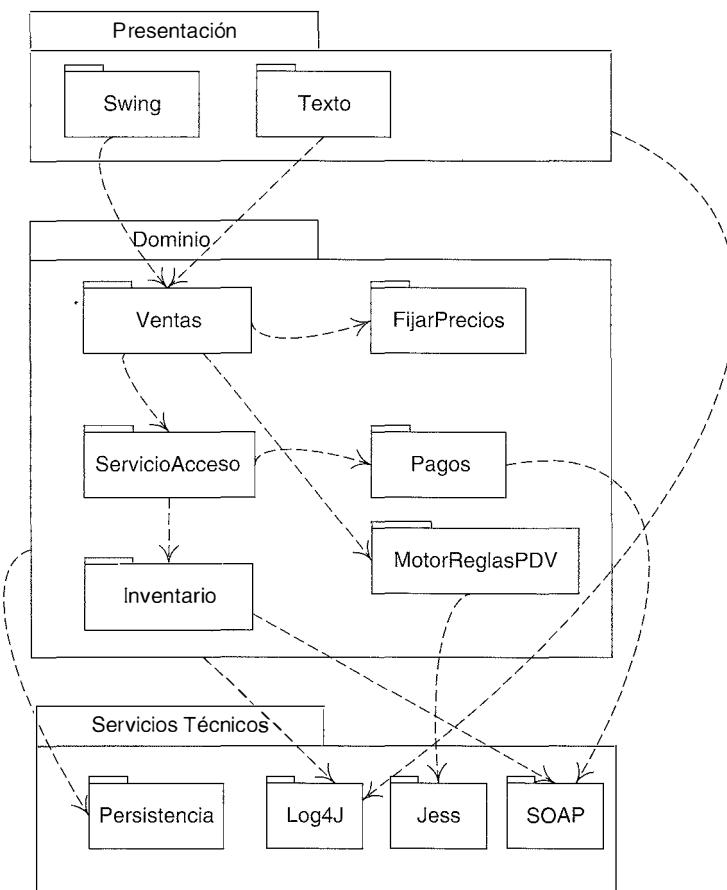


Figura 30.4. Acoplamiento parcial entre los paquetes.

De hecho, la Figura 30.4 ilustra probablemente el estilo más común del diagrama de la arquitectura lógica en UML —un diagrama de paquetes que normalmente muestra de 5 a 20 paquetes importantes, y sus dependencias—.

Escenarios de interacción entre capas y entre paquetes

Los diagramas de paquetes muestran información estática. Un diagrama de interacción proporciona la información para entender la dinámica del modo en el que se conectan y comunican los objetos entre las capas. En el espíritu de una “vista de la arquitectura” que oculta los detalles irrelevantes, y destaca lo que la arquitectura quiere transmitir, un diagrama de interacción en la vista lógica de la arquitectura se centra en las colaboraciones que cruzan los límites de las capas y los paquetes. Por tanto, es útil contar con un conjunto de diagramas de interacción que ilustren los **escenarios más significativos desde el punto de vista de la arquitectura** (en el sentido de que ilustran muchos aspectos de gran escala o grandes ideas del diseño).

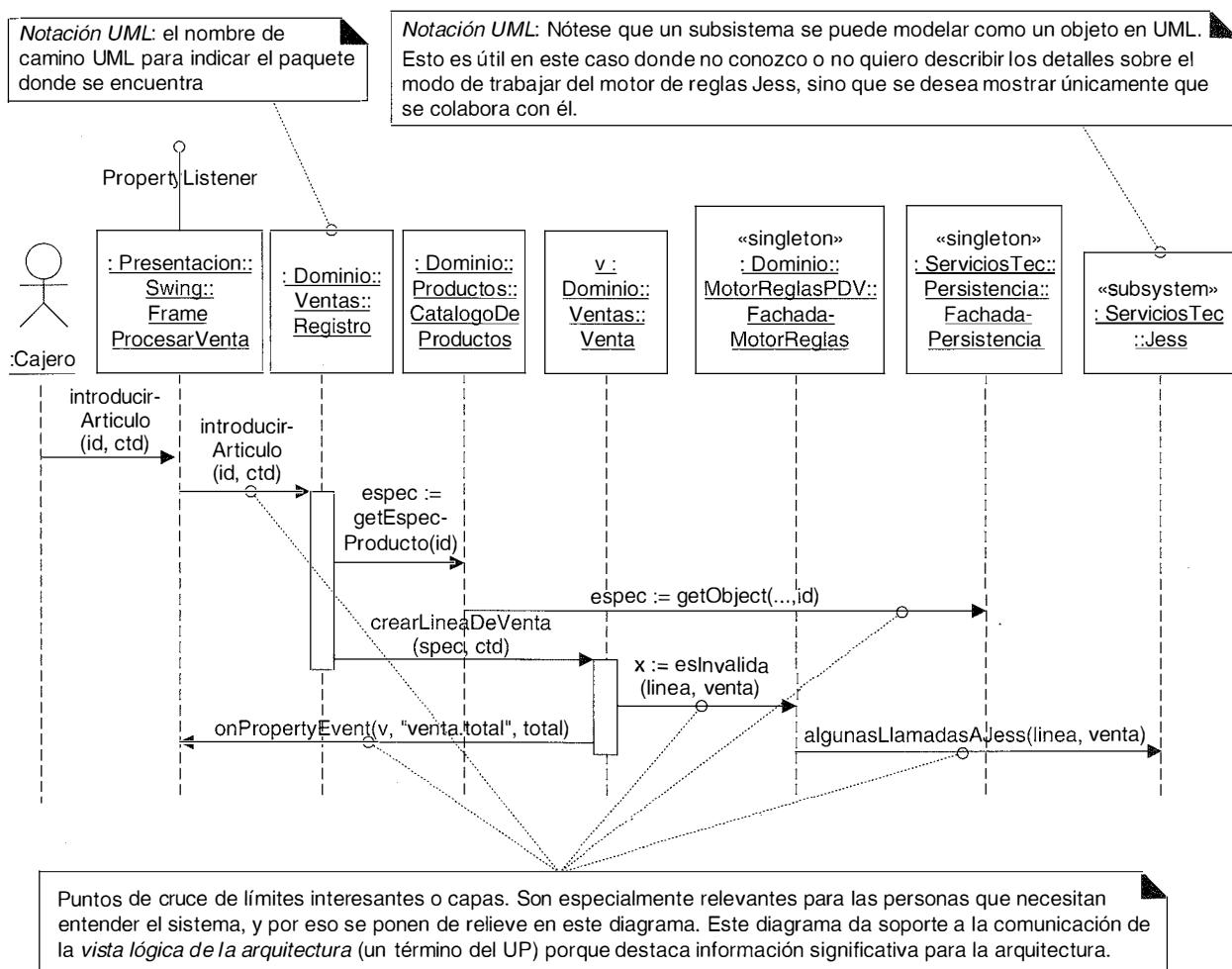


Figura 30.5. Un diagrama de interacción significativo desde el punto de vista de la arquitectura que pone de relieve las conexiones que cruzan los límites.

Por ejemplo, la Figura 30.5 ilustra parte del escenario *Procesar Venta* que pone de relieve los puntos de conexión entre las capas y los paquetes.

Notación UML:

- El paquete al que pertenece un tipo se puede mostrar opcionalmente calificando el tipo con la expresión del **nombre de camino** de UML <NombrePaquete>::<NombreTipo>. Por ejemplo, *Dominio*::*Ventas*::*Registro*. Esto se puede aprovechar para destacarle al lector las conexiones entre los paquetes y entre las capas en el diagrama de interacción.
- Nótese también el uso del estereotipo «*subsystem*». En UML, un subsistema es una entidad discreta que tiene comportamiento e interfaces. Se puede modelar un subsistema como un tipo especial de paquete, o —como se muestra aquí— como un objeto, que es útil cuando uno quiere mostrar las colaboraciones entre subsistemas (o sistemas). En UML, el sistema entero es también un “subsistema” (la raíz) y, por tanto, también se puede mostrar como un objeto en un diagrama de interacción (como un DSS).

Obsérvese que el diagrama no muestra algunos mensajes, como ciertas colaboraciones de la *Venta*, para poner de relieve las interacciones significativas para la arquitectura.

Colaboraciones Dos decisiones de diseño al nivel de la arquitectura son:

1. ¿Cuáles son las partes importantes?
2. ¿Cómo se conectan?

Mientras el patrón de arquitectura Capas guía en la definición de las partes importantes, los patrones de micro-arquitectura como Fachada, Controlador y Observador se utilizan comúnmente para el diseño de las conexiones entre las capas y los paquetes. Esta sección estudia los patrones para la conexión y comunicación entre las capas y los paquetes.

Paquetes simples vs. subsistemas

Algunos paquetes o capas no son simplemente grupos de cosas conceptuales, sino que son verdaderos subsistemas con comportamiento e interfaces. Para contrastar:

- El paquete *FijarPrecios* no es un subsistema; agrupa simplemente la factoría y las estrategias que se utilizan para fijar los precios. De igual modo que con los paquetes Base como *java.util*.
- Por otro lado, los paquetes *Persistencia*, *MotorReglasPDV* y *Jess* son subsistemas. Son motores discretos con responsabilidades cohesivas que realizan un trabajo.

En UML, un subsistema se puede identificar con un estereotipo como en la Figura 30.6.

Fachada

Para los paquetes que representan subsistemas, el patrón más común de acceso es el Fachada, un patrón de diseño GoF. Esto es, un objeto fachada público define los ser-

vicios para el subsistema, y los clientes colaboran con la fachada, no con componentes internos del subsistema. Esto es cierto en lo que se refiere a *FachadaMotorReglasPDV* y *FachadaPersistencia* para acceder a los subsistemas de motor de reglas y de persistencia.

Normalmente, la fachada no debería incluir muchas operaciones de bajo nivel. Más bien, es deseable que la fachada incluyera un pequeño número de operaciones de alto nivel —los servicios de grano grueso—. Cuando una fachada da a conocer muchas operaciones de bajo nivel, tiende a perder la cohesión. Además, si la fachada será, o podría llegar a ser, un objeto distribuido o remoto (como un *bean* de sesión EJB, o un objeto servidor RMI), los servicios de grano fino dan lugar a problemas de rendimiento en las comunicaciones —muchas llamadas remotas pequeñas constituyen un cuello de botella en cuanto al rendimiento en los sistemas distribuidos—.

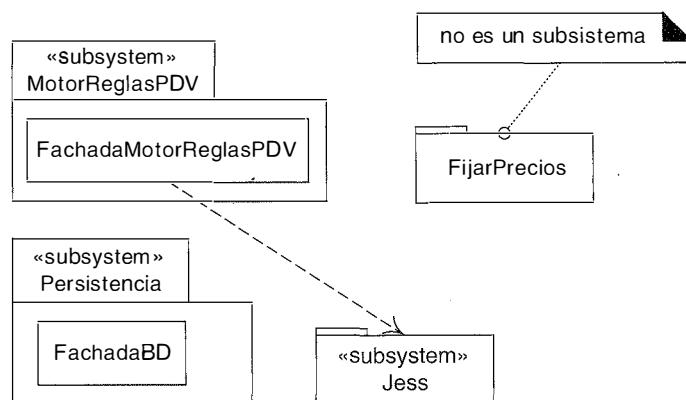


Figura 30.6. Estereotipos de subsistemas.

Por otro lado, normalmente una fachada no realiza su propio trabajo. Más bien, actúa para concentrar como mediador con los objetos del subsistema subyacente, que son los que hacen el trabajo.

Por ejemplo, la *FachadaMotorReglasPDV* es el envoltorio (*wrapper*) y único punto de acceso al motor de reglas para la aplicación PDV. Los otros paquetes no ven la implementación de este subsistema, puesto que está oculta detrás de la fachada. Suponga (ésta es sólo una de las muchas implementaciones) que el subsistema del motor de reglas del PDV se implementa colaborando con el motor de reglas Jess. Jess es un subsistema que expone en su interfaz muchas operaciones de grano fino (esto es común en los subsistemas de terceras partes muy generales). Pero la *FachadaMotorReglasPDV* no pone al descubierto en su interfaz las operaciones de bajo nivel de Jess, sino que proporciona sólo unas pocas operaciones de alto nivel como *esInvalida(linea, venta)*.

Si la aplicación tiene sólo un número “pequeño” de operaciones del sistema, entonces es habitual que la capa del Dominio o de la Aplicación sólo exponga un objeto a una capa superior. Por otro lado, la capa de Servicios Técnicos, que contiene varios subsistemas, expone al menos una fachada (o varios objetos públicos, si no se utilizan las fachadas) a las clases superiores para cada subsistema. Véase la Figura 30.7.

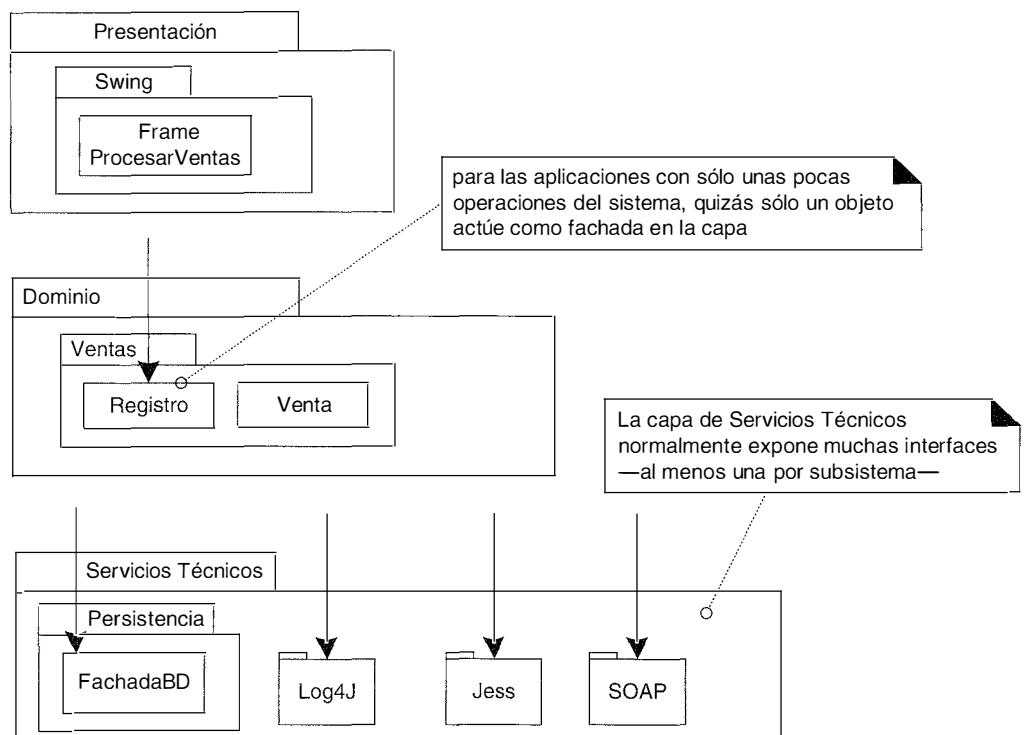


Figura 30.7. Número de interfaces que se muestran a las capas superiores.

Fachadas de Sesión y la capa de Aplicación

A diferencia de la Figura 30.7, cuando una aplicación tiene muchas operaciones del sistema y da soporte a muchos casos de uso, es común que haya más de un objeto mediando entre la capa de Presentación y del Dominio.

En la versión actual del sistema NuevaEra, existe un diseño simple de un único objeto *Registro* que actúa como fachada en la capa del Dominio (en virtud del patrón GRASP Controlador).

Sin embargo, cuando el sistema crece para manejar muchos casos de uso y operaciones del sistema, no es raro que se introduzca una capa de Aplicación de objetos que mantienen el estado de la sesión para las operaciones de un caso de uso, donde cada instancia de sesión representa una sesión con un cliente. Estos objetos se conocen como Fachadas de Sesión, y su uso es otra recomendación del patrón GRASP Controlador, como en el controlador fachada de sesión de caso de uso que es una variante del patrón. En la Figura 30.8 se presenta un ejemplo del modo en el que podría evolucionar la arquitectura del NuevaEra con una capa de Aplicación y fachadas de sesión.

Controlador

El patrón GRASP Controlador describe alternativas comunes en el manejo del lado del cliente (o controladores, como se les ha denominado) de las peticiones de las operaciones del sistema que se emiten desde la capa de Presentación. La Figura 30.9 lo ilustra.

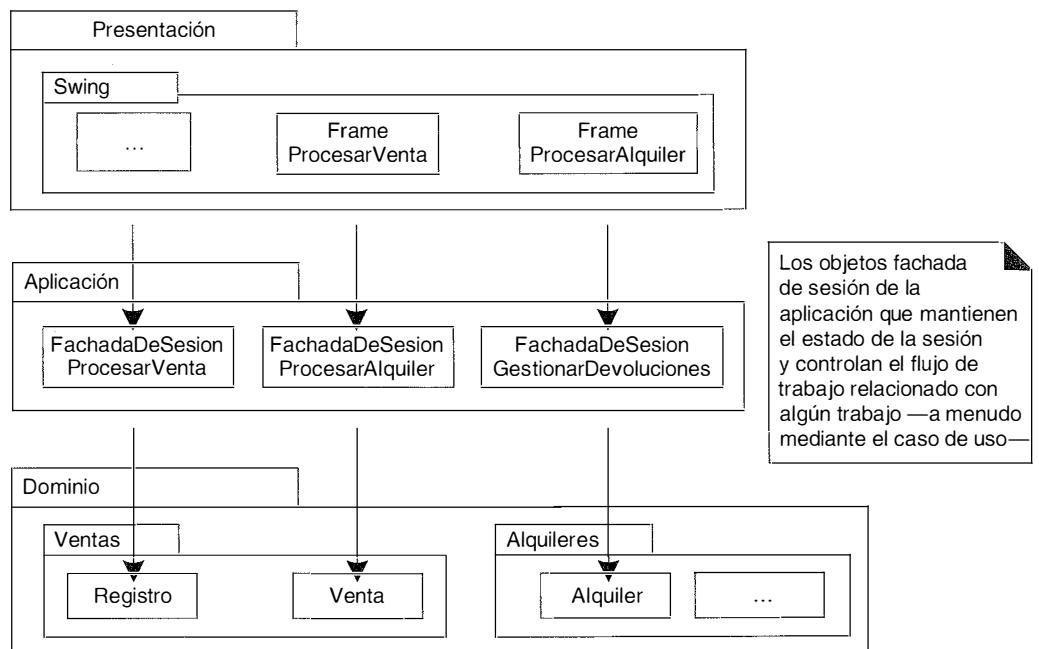


Figura 30.8. Fachadas de sesión y una Capa de Aplicación.

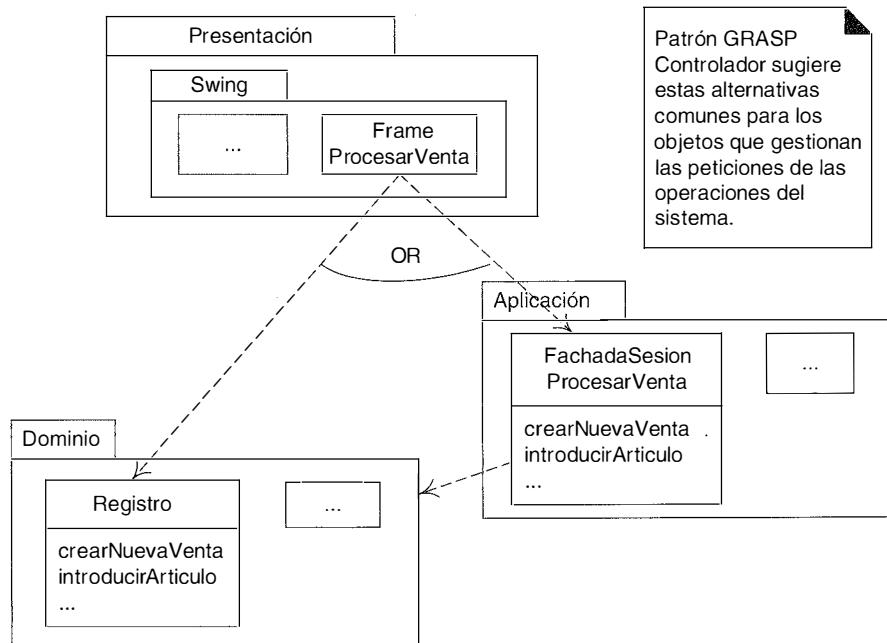


Figura 30.9. Alternativas del Controlador.

Operaciones del sistema y las capas

Los DSS ilustran las operaciones del sistema, ocultando los objetos de presentación del diagrama. Las operaciones del sistema que se invocan sobre el sistema en la Figura 30.10

son peticiones que genera un actor por medio de la capa de Presentación, en la capa de Aplicación o del Dominio.

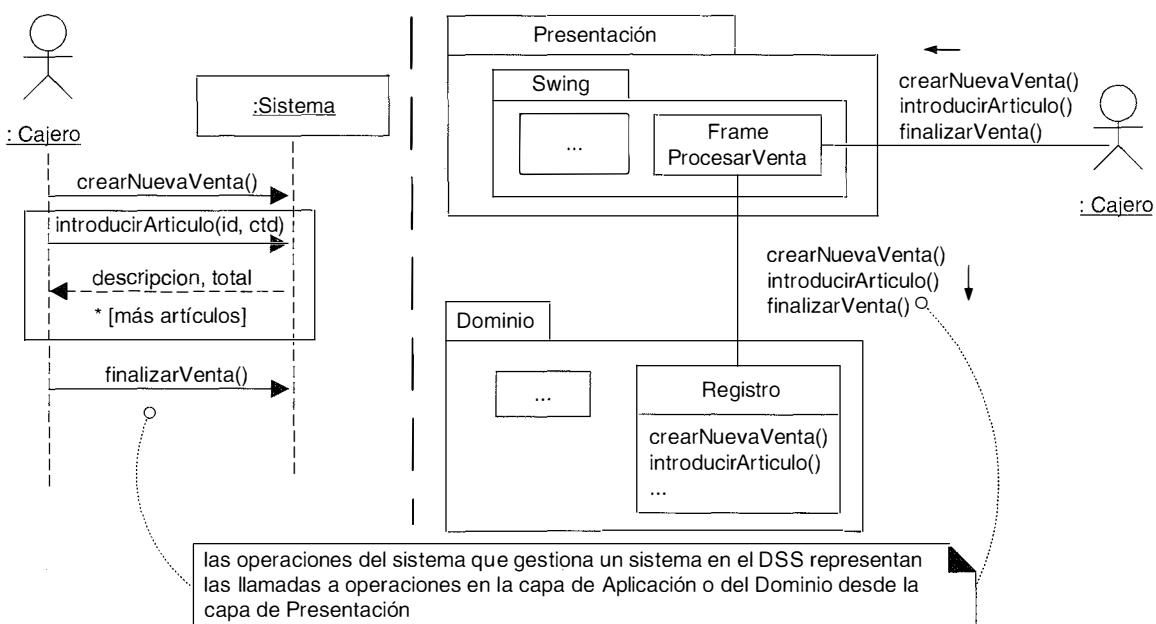


Figura 30.10. Operaciones del sistema en los DSSs y en función de las capas.

Colaboraciones ascendentes con el Observador

El patrón Fachada se utiliza normalmente para la colaboración “descendente” desde una capa más alta a una más baja, o para el acceso a los servicios de otros subsistemas de la misma capa. Cuando las capas más bajas Aplicación o Dominio necesitan comunicarse hacia arriba con la capa de Presentación, por lo general se hace por medio del patrón Observador. Es decir, los objetos UI en la capa de Presentación más alta implementan una interfaz como *PropertyListener* o *AlarmaListener*, y se suscriben o escuchan los eventos (como los eventos sobre la propiedad o de la alarma) que proceden de los objetos de las capas más bajas. Los objetos de las capas más bajas envían mensajes directamente a los objetos superiores de la capa de UI, pero sólo se acopla con los objetos vistos como cosas que implementan una interfaz como *PropertyListener*, no vistos como una ventana concreta de la GUI.

Esto se estudió cuando se presentó el patrón Observador. La Figura 30.11 resume la idea en relación con las capas.

Acoplamiento relajado entre las capas

Las capas en la mayoría de las arquitecturas en capas *no* están acopladas en el mismo sentido limitado que un protocolo de red basado en el Modelo OSI de 7 Capas. En el modelo del protocolo, existe una restricción estricta de que los elementos de la capa N sólo acceden a los servicios de la capa inmediatamente inferior N-1.

Esto raramente se sigue en las arquitecturas de los sistemas de información. Más bien, el estándar es una arquitectura de “capas relajadas” o “de capas transparentes”

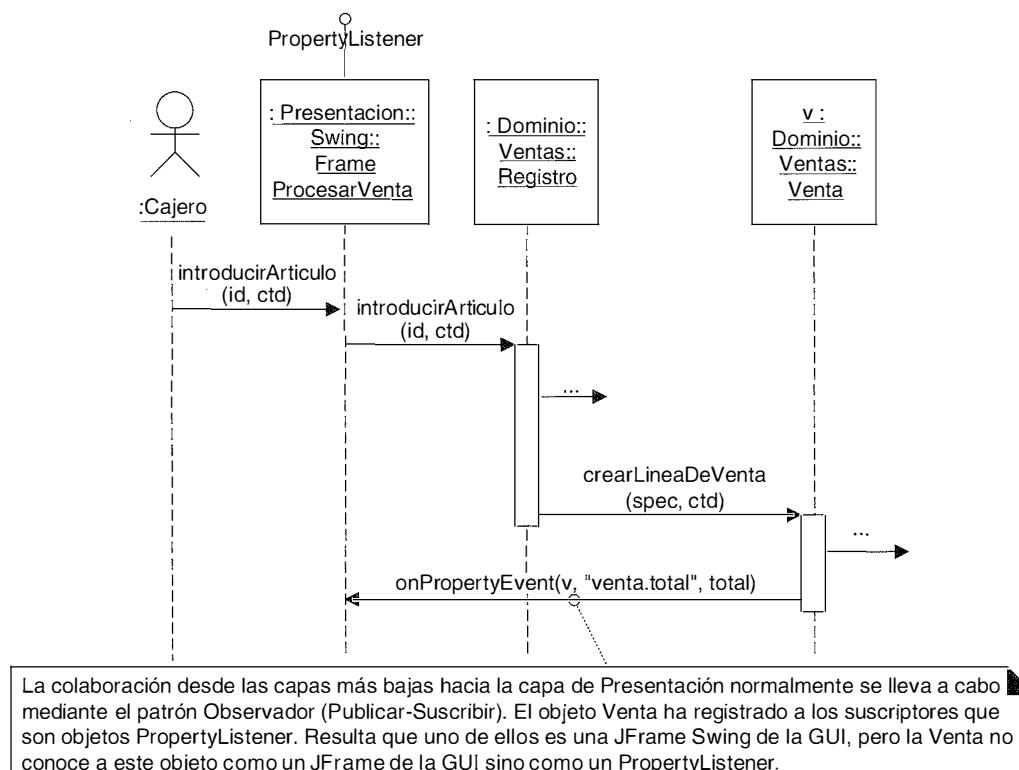


Figura 30.11. Observador para la comunicación “ascendente” con la capa de Presentación.

[BMRSS96], en el que los elementos de una capa colaboran o se acoplan con varias de las otras capas.

Comentarios acerca del acoplamiento típico entre las capas:

- Todas las capas más altas dependen de los Servicios Técnicos y la capa Base.
 - Por ejemplo, en Java todas las capas dependen de los elementos del paquete `java.util`.
- Es sobre todo la capa del Dominio la que depende de la capa de Infraestructura del Negocio.
- La capa de Presentación realiza llamadas a la capa de Aplicación, que solicita los servicios de la capa del Dominio; la capa de Presentación no invoca al Dominio, a menos que no exista la capa de Aplicación.
- Si se trata de una aplicación “*desktop*” de un solo proceso, los objetos software en la capa del Dominio son visibles directamente, o se pasan entre, Presentación, Aplicación, y en menor extensión con, Servicios Técnicos.
 - Por ejemplo, asumiendo que el sistema NuevaEra es de este tipo, un objeto *Venta* y un *Pago* podrían ser visibles directamente en la Capa de Presentación de GUI, y pasarse también al subsistema de Persistencia en la capa de Servicios Técnicos.

- Por otro lado, si se trata de un sistema distribuido, en general se pasa a la capa de Presentación las **réplicas serializables** (también conocidas como **objetos data holder** u **objetos valor**) de los objetos de la capa del Dominio. En este caso, la capa del Dominio se despliega en un ordenador servidor, y los nodos clientes obtienen copias de los datos del servidor.

¿No es peligroso el acoplamiento entre los servicios técnicos y las capas básicas?

Como se presentó en la discusión acerca de los patrones GRASP Variaciones Protegidas y Bajo Acoplamiento, no es el acoplamiento en sí lo que constituye un problema, sino el acoplamiento innecesario en los puntos de variación y evolución que son inestables y costosos de arreglar. Es poco justificable el malgastar el tiempo y el dinero intentando abstraer u ocultar algo que no es probable que cambie, o si lo hace, el coste del impacto sería insignificante. Por ejemplo, si construimos una aplicación con las tecnologías Java, ¿cuál es el beneficio de ocultar en la aplicación el acceso a las librerías de Java? El alto acoplamiento en muchos puntos de las librerías es un problema poco probable, puesto que son (relativamente) estables y ubicuas.

Discusión Además de las cuestiones estructurales y de colaboración de este patrón discutidas anteriormente, entre otras cuestiones encontramos las siguientes.

¿Recursos externos o capa de base de datos externa abajo?

La mayoría de los sistemas confían en recursos o servicios externos, como una base de datos Oracle o un servicio de nombres y de directorio LDAP Novell. Éstos son componentes de implementación *físicos*, no una capa en la vista *lógica* de la arquitectura.

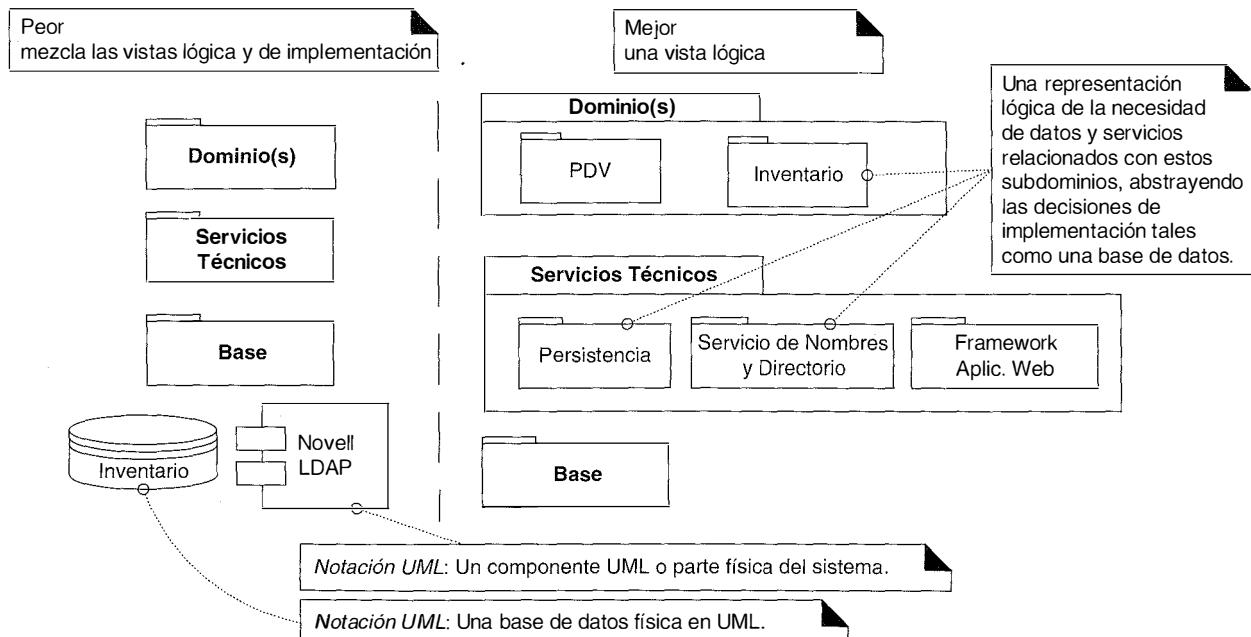


Figura 30.12. Mezcla de vistas de la arquitectura.

Mostrando los recursos externos, como una base de datos específica, en una capa “por debajo de” la capa Base (por ejemplo) mezcla la vista lógica y las vistas de despliegue o implementación de la arquitectura.

Más bien, en función de la vista lógica de la arquitectura y sus capas, se puede ver el acceso a un conjunto de datos persistentes (como los datos del inventario) como un subdominio de la Capa del Dominio —el subdominio del Inventario—. Y los servicios generales que proporcionan el acceso a las bases de datos se podrían ver como una partición del Servicio Técnico —el servicio de Persistencia—. Véase la Figura 30.12.

Vista lógica vs. las vistas de proceso y despliegue de la arquitectura

Las capas de la arquitectura son una vista lógica de la arquitectura, no una vista de despliegue de los elementos en procesos y nodos de procesos. Dependiendo de la plataforma, *todas* las capas podrían desplegarse en el mismo proceso en el mismo nodo, como una aplicación en un PDA portátil, o dispersas por muchos ordenadores y procesos para una aplicación web de gran escala.

En el Modelo de Despliegue del UP que establece la correspondencia entre la arquitectura lógica y los procesos y nodos, influye fuertemente la elección de la plataforma software y hardware y los *frameworks* de aplicación asociados. Por ejemplo, la elección entre J2EE o .NET influye en la arquitectura de despliegue.

Existen muchas formas de organizar estas capas lógicas para el despliegue, y en general el tema de la arquitectura de despliegue sólo se introducirá ligeramente, puesto que no es trivial, queda claramente fuera del alcance de este libro y depende de una discusión detallada de la plataforma software elegida, tal como J2EE.

¿Es opcional la capa de Aplicación?

De existir, la capa de Aplicación contiene los objetos responsables de conocer el estado de la sesión de los clientes, de mediar entre las capas de Presentación y del Dominio, y de controlar el flujo de trabajo.

El flujo podría organizarse controlando el orden de las ventanas o las páginas web, por ejemplo.

En cuanto a los patrones GRASP, forman parte de esta capa los objetos Controlador GRASP como el controlador de fachada de caso de uso. En sistemas distribuidos, forman parte de esta capa componentes tales como los *bean* sesión EJB (y los objetos sesión con estado en general).

En algunas aplicaciones, no es necesaria esta capa. Es útil (ésta no es una lista exhaustiva) cuando se cumple uno o más de los siguientes criterios:

- Se utilizarán diversas interfaces de usuario (por ejemplo, páginas web y una GUI Swing) en el sistema. Los objetos de la capa de Aplicación pueden actuar como Adaptadores que recopilan y reúnen los datos como se necesitan para diferentes UIs, y pueden actuar como Fachadas que envuelven y ocultan el acceso a la capa del Dominio.
- Es un sistema distribuido y la capa del Dominio está en un nodo diferente al de la capa de Presentación, y la comparten múltiples clientes. Normalmente se necesita

mantener la traza del estado de la sesión, y los objetos de la capa de Aplicación son una opción conveniente para esta responsabilidad.

- La Capa del Dominio no puede o no debe mantener el estado de la sesión.
- Existe un flujo de trabajo definido en función del orden controlado de las ventanas o páginas web que se deben presentar.

Pertenencia a un conjunto difuso en capas diferentes

Algunos elementos —son sin ninguna duda— miembros de una capa; una clase *Math*² forma parte de la capa Base. Sin embargo, especialmente entre las capas de Servicios Técnicos y Base, y Dominio e Infraestructura del Negocio, es difícil clasificar algunos elementos, porque la diferencia entre estas capas es, a grandes rasgos, “alto” frente a “bajo”, o “específico” frente “general”, que son un conjunto de términos difusos. Esto es normal, y rara vez es necesario optar por una clasificación definitiva —el equipo de desarrollo podría considerar que un elemento a grandes rasgos forma parte de los Servicios Técnicos y/o la capa Base que se consideran como un grupo, en general conocido como capa de Infraestructura.³

Por ejemplo:

- Suponga que se trata de un proyecto que utiliza las tecnologías de Java, y se ha elegido el framework de *logging*⁴ de libre distribución *Log4J* (parte del proyecto Jakarta). ¿Logging forma parte del Servicio Técnico o de la capa Base? Log4J es un framework general, pequeño, de bajo nivel. Es razonablemente miembro de los dos conjuntos difusos Servicios Técnicos y Base.
- Suponga que se trata de una aplicación web, y se ha elegido el framework para aplicaciones web *Struts* de Jakarta. Struts es un framework técnico, específico, relativamente de alto nivel y grande. Se puede justificar fuertemente que es miembro del conjunto de Servicios Técnicos, y débilmente que es miembro del conjunto Base.

Pero, lo que para una persona es el Servicio Técnico de Alto Nivel, para otras es la Base...

Finalmente, no es el caso de que las librerías que proporcionan una plataforma software sólo representan servicios Básicos de bajo nivel. Por ejemplo, tanto en .NET como J2SE+J2EE, los servicios incluyen funciones de relativamente alto nivel como los servicios de nombres y directorio.

Terminología: niveles, capas y particiones

La noción original de **nivel (*tier*)** en arquitectura era una capa lógica, no un nodo físico, pero la palabra ha pasado a utilizarse ampliamente para referirse a un nodo de procesamiento físico (o una agrupación de nodos), como el “nivel del cliente” (el ordenador del

² *N. del T.* Se refiere a la clase *Math* de Java que reúne funciones matemáticas.

³ Nótese que no existe una convención de nombres bien establecida para las capas, y es común que aparezca en la literatura sobre la arquitectura sobrecarga y contradicciones en los nombres.

⁴ *N. del T.* Permite al programador insertar sentencias log (control de errores) en programas Java sin incurrir en una penalización del rendimiento.

cliente). Esta presentación evitará el término para ser más claros, pero téngalo presente cuando lea literatura sobre arquitectura.

Las **capas** de una arquitectura se dice que representan los cortes verticales, mientras que las **particiones** representan una división horizontal de subsistemas relativamente paralelos de una capa. Por ejemplo, la capa de *Servicios* podría dividirse en particiones tales como *Seguridad* e *Informes* (Figura 30.13).

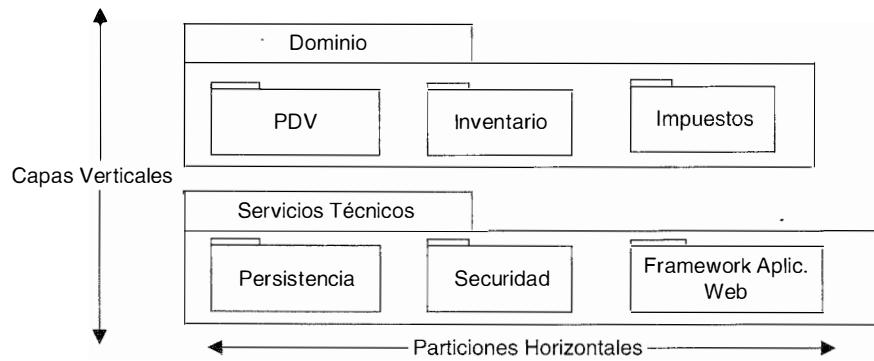


Figura 30.13. Capas y particiones.

Contraindicaciones y Compromisos

- En algunos contextos, añadir capas introduce problemas de rendimiento. Por ejemplo, en un juego en el que se utilizan muchos gráficos y que exige alto rendimiento añadir capas de abstracción e indirección sobre el acceso a los componentes de la tarjeta gráfica podría introducir problemas de rendimiento.
- El patrón Capas es uno entre varios patrones de arquitectura básicos; no es aplicable a todos los problemas. Por ejemplo, un sustituto es Tuberías y Filtros (*Pipes and Filters*) [BMRSS96]. Éste es útil cuando el problema principal de la aplicación implica el procesamiento de alguna cosa mediante una serie de transformaciones, como transformaciones de imagen, y el orden de las transformaciones puede cambiar. Pero incluso en el caso en el que el patrón de la arquitectura de más alto nivel sea Tuberías y Filtros, las tuberías y filtros particulares se pueden diseñar con Capas.

Beneficios

- En general, existe una separación de intereses, una separación entre los servicios de alto y bajo nivel, y de servicios específicos y generales de la aplicación. Esto reduce el acoplamiento y las dependencias, mejora la cohesión, incrementa el potencial para reutilizar e incrementa la claridad.
- La complejidad relacionada se encapsula y se descompone.
- Algunas capas se pueden reemplazar por implementaciones nuevas. Esto generalmente no es posible en las capas de más bajo nivel de Servicios Técnicos o Base (ej. `java.util`), pero podría ser posible en las capas de Presentación, Aplicación y Dominio.
- Las capas más bajas contienen funciones reutilizables.
- Algunas capas (sobre todo de Dominio y Servicios Técnicos) pueden ser distribuidas.
- Se ayuda al desarrollo en equipo debido a la segmentación lógica.

Implementación Implementación de las capas: personas y procesos

Es común y recomendable, en una iteración, contar con un desarrollador especializado en una capa o en un servicio.

Pero, no es el caso de que el equipo completo del proyecto se centre en una capa o servicio en una iteración. Sino más bien, es más habitual implementar cortes verticales entre las capas. Éste es el enfoque del UP en la fase de elaboración: elegir escenarios y requisitos que fuercen, en cada iteración, a cubrir ampliamente muchos paquetes/capas/subsistemas significativos para la arquitectura, para descubrir y estabilizar los elementos importantes de la arquitectura en las primeras iteraciones.

Sin embargo, en este libro, no se ilustra este enfoque en el caso de estudio NuevaEra, porque hacerlo así requeriría una discusión previa de muchos y extensos temas —desde la programación de GUI a la correspondencia objeto-relacional y la optimización de sentencias SQL—. El caso de estudio del libro se ha centrado en el diseño de los objetos de la capa del Dominio, mientras que se reconoce que en realidad se llevaría a cabo un trabajo paralelo para desarrollar otras capas y subsistemas.

Los principios de diseño que se ilustran para el caso de estudio se pueden aplicar prácticamente en todas las capas del diseño.

Vista de implementación: correspondencia de la organización del código fuente con las capas y paquetes

La organización del código fuente forma parte del Modelo de Implementación del UP. Para lenguajes como Java o C#, que proporcionan un soporte sencillo para los paquetes (espacio de nombres, *namespace*), la correspondencia entre los paquetes lógicos y los paquetes de implementación es similar, con notables excepciones cuando se utilizan librerías de terceras partes⁵. De hecho, sólo es en las primeras etapas del desarrollo, cuando los paquetes se dibujan de manera especulativa, pero no se implementan, que hay diferencias significativas.

A lo largo del tiempo, cuando el código base crece, es normal abandonar los primeros dibujos especulativos (como los que acabamos de ver), y en lugar de eso utilizar una herramienta CASE UML para llevar a cabo un proceso de ingeniería inversa que lea el código fuente y genere un diagrama de paquetes. Entonces, estos diagramas de paquetes generados automáticamente, que reflejan de manera precisa el código (el diseño real) se convierten en la base de la vista lógica de la arquitectura.

Utilizando Java como ejemplo para mostrar la correspondencia con los paquetes de la implementación, las capas y paquetes que se ilustraban en la Figura 30.4 podrían corresponderse con los siguientes nombres de paquetes Java:

```
// --- PRESENTACIÓN

com.foo.nuevaera.ui.swing
com.foo.nuevaera.ui.texto
```

⁵ C++ también da soporte a los espacios de nombres, pero resulta difícil utilizar el lenguaje con docenas o cientos de espacios de nombres de grano fino; no así para Java o C#.

```

// --- DOMINIO

    //paquetes relativamente específicos del proyecto NuevaEra
com.foo.nuevaera.dominio.ventas
com.foo.nuevaera.dominio.fijarprecios
com.foo.nuevaera.dominio.servicioacceso
com.foo.nuevaera.dominio.motorreglaspdv

    //paquetes que se pueden diseñar fácilmente como
    //servicios del negocio comunes a múltiples aplicaciones
com.foo.dominio.inventario
com.foo.dominio.pagoacredito


// --- SERVICIOS TÉCNICOS

    //nuestro equipo crea
com.foo.servicios.persistencia

    //de terceras partes
org.apache.log4j
org.apache.soap.rpc
jess


// --- BASE

    //nuestro equipo crea
com.foo.utilidades
com.foo.utilstring

```

Nótese que se ha hecho un esfuerzo por evitar la utilización de los calificadores de los nombres de los paquetes dependientes de la aplicación (“nuevaera”) a menos que sea necesario. Por ejemplo, los paquetes de UI están relacionados con la aplicación NuevaEra, y de ahí que se califiquen con el nombre de la aplicación *com.foo.nuevaera.ui*.*

Para dar soporte a la reutilización, una práctica es nombrar los elementos de manera independiente de la aplicación, cuando sea apropiado. Un ejemplo sencillo, la utilidades de propósito general para los *String* creadas por el equipo NuevaEra, se colocan en *com.foo.utilstring*, en lugar de en *com.foo.nuevaera.utilstring*. Además, *com.foo.utilstring* debería colocarse en el repositorio de código fuente de la compañía al nivel de la compañía, en lugar de enterrarse en las carpetas de código fuente del proyecto NuevaEra. No puede reutilizar lo que no puede ver.

Otro ejemplo, considere los servicios para acceder a los sistemas de terceras partes del inventario y autorización de pago a crédito. Aunque fue el equipo NuevaEra el que los creó al servicio del proyecto NuevaEra, son servicios del negocio generales —uno podría imaginar el acceso a un sistema de inventario desde el interior de otras aplicaciones—; lo mismo para la autorización de pagos a crédito. De ahí que sea preferible *com.foo.dominio.inventario* en lugar de *com.foo.nuevaera.dominio.inventario*.

Por otro lado, el paquete MotorReglasPDV está completamente relacionado con el proyecto del PDV NuevaEra. Así, *com.foo.nuevaera.dominio.motorreglaspdv*.

En caso de duda, califique el paquete con el nombre del proyecto. Siempre se puede cambiar más adelante.

Usos Conocidos Un gran número de sistemas orientados a objetos modernos (desde aplicaciones desktop a sistemas web distribuidos J2EE) se desarrollan con Capas; sería más difícil encontrar uno que no esté desarrollado así, que uno que sí. Retrocediendo más lejos en la historia:

Máquinas virtuales y sistemas operativos

Comenzando en los años sesenta, los arquitectos de sistemas operativos abogaban por el diseño de sistemas operativos en función de capas claramente definidas, donde la capa “más baja” encapsulaba el acceso a los recursos físicos y proporcionaba procesos y servicios de E/S, y las capas más altas invocaban estos servicios. Entre éstos se encuentran Multics [CV65] y el sistema THE [Dijkstra68].

Todavía antes —en los cincuenta— los investigadores sugirieron la idea de una máquina virtual (MV) con un lenguaje máquina universal de bytecodes (por ejemplo, UNCOL [Conway1958]), de manera que las aplicaciones pudieran escribirse en las capas más altas de la arquitectura (y ejecutarse sin tener que recompilarse en diferentes plataformas), sobre la capa de la máquina virtual, que por su parte se asentaría por encima del sistema operativo y los recursos de la máquina. Alan Kay aplicó la arquitectura de capas con MV en Flex un sistema de computación personal basado en la orientación a objetos que marcó un hito [Kay68] y después (1972) Kay y Dan Ingalls en la influyente máquina virtual de Smalltalk [GK76] —la progenitora de las MVs más recientes como la Máquina Virtual de Java—.

Sistemas de información: la arquitectura clásica de tres-niveles

Una primera descripción de la arquitectura de capas para sistemas de información que ha tenido mucha influencia, que incluía una interfaz de usuario y el almacenamiento persistente de los datos, era conocida como **arquitectura de tres niveles** (Figura 30.14), fue descrita en los años setenta en [TK78]. El término no alcanzó popularidad hasta mediados de los noventa, en parte debido a que en [Gartner95] se presentaba como una solución a los problemas asociados con el uso extendido de las arquitecturas de dos niveles.

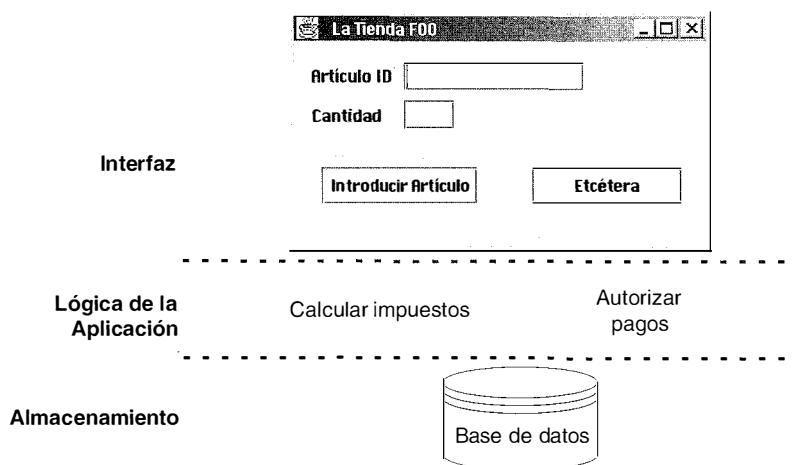


Figura 30.14. Vista clásica de una arquitectura de tres niveles.

El término original ahora es menos común, pero su motivación aún es relevante.

Una descripción clásica de los niveles verticales en la arquitectura de tres niveles es:

1. **Interfaz:** ventanas, informes, etcetera.
2. **Lógica de la Aplicación:** tareas y reglas que dirigen el proceso.
3. **Almacenamiento:** mecanismos de almacenamiento persistente.

La cualidad singular de una arquitectura de tres niveles es la separación de la lógica de la aplicación en un nivel de software intermedio distinto para la lógica. El nivel de la interfaz se encuentra relativamente liberado del procesamiento de la aplicación; las ventanas o las páginas web remiten las peticiones de las tareas al nivel intermedio. El nivel intermedio se comunica con la capa de almacenamiento back-end.

Hubo algún malentendido, se creía que la descripción original implicaba o requería un despliegue físico en tres ordenadores, pero la descripción dada era puramente lógica; la asignación de los niveles a los nodos de ordenadores podría variar de uno a tres. Véase la Figura 30.15.

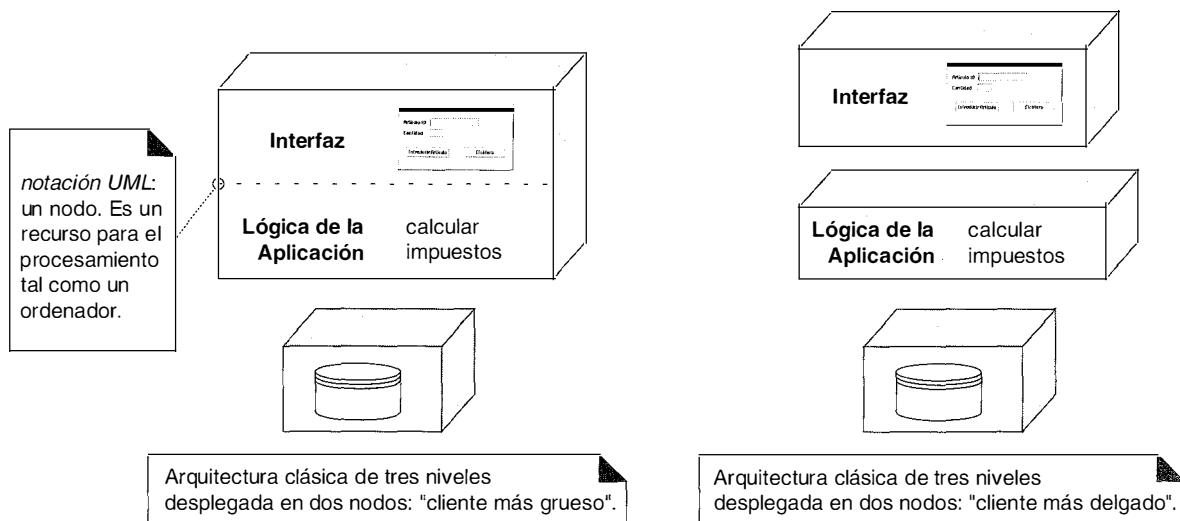


Figura 30.15. Una división lógica en tres niveles desplegada en dos arquitecturas físicas.

El Grupo Gartner contrastó la arquitectura de tres niveles con un diseño **en dos niveles**, en el cual, por ejemplo, la lógica de la aplicación se coloca en las definiciones de las ventanas, que lee y escribe directamente en una base de datos; no existe nivel intermedio que separe la lógica de la aplicación. Las arquitecturas cliente-servidor de dos niveles se hicieron populares especialmente al surgir herramientas como Visual Basic y PowerBuilder.

Los diseños en dos niveles tienen (en algunos casos) la ventaja de un desarrollo rápido inicial, pero puede sufrir los inconvenientes que se presentaron en la sección de *Problemas*. No obstante, hay aplicaciones que son ante todo simples sistemas de uso intensivo de datos CRUD (creación, recuperación, actualización y borrado), para los que esto es una opción adecuada.

- Patrones Relacionados**
- Indirección: las capas pueden añadir un nivel de indirección a los servicios de un nivel inferior.
 - Variaciones Protegidas: las capas pueden proteger contra el impacto de implementaciones que varían.
 - Bajo Acoplamiento y Alta Cohesión: las capas dan soporte efectivo a estos objetivos.
 - La aplicación específica a los sistemas de información orientados a objetos se describe en [Fowler96].

También Conocido Como Arquitectura en Capas [Shaw96, Gemstone00].

30.3. Principio de Separación Modelo-Vista

Este principio se ha discutido varias veces; esta sección lo resume.

¿Qué tipo de visibilidad tendrían que tener otros paquetes de la capa de Presentación? ¿Cómo deberían comunicarse las clases que no son ventanas con las ventanas? Es conveniente que no exista un acoplamiento directo de otros componentes con los objetos ventana puesto que las ventanas están relacionadas con una aplicación específica, mientras que (idealmente) los componentes que no son ventanas podrían reutilizarse en nuevas aplicaciones o conectarse a una nueva interfaz. Éste es el principio de Separación Modelo-Vista.

En este contexto, el **modelo** es un sinónimo de la capa del Dominio de los objetos. La **Vista** es un sinónimo para los objetos de la presentación, como ventanas, applets e informes.

El principio de **Separación Modelo-Vista**⁶ establece que los objetos del modelo (dominio) no deberían conocer *directamente* a los objetos de la vista (presentación), al menos como objetos de la vista. Así, por ejemplo, un objeto *Registro* o *Venta* no debería enviar un mensaje directamente a un objeto ventana de la GUI *FrameProcesarVenta*, pidiéndole que muestre algo, cambie de color, se cierre, etcétera.

Como se discutió anteriormente, una relajación legítima de este principio es el patrón Observador, donde los objetos del dominio envían mensajes a los objetos de la UI vistos sólo en términos de una interfaz como *PropertyListener* o *AlarmaListener*.

Una parte adicional de este principio es que las clases del dominio encapsulan la información y el comportamiento relacionado con la lógica de la aplicación. Las clases de las ventanas son relativamente delgadas; son responsables de la entrada y salida, y capturar los eventos del GUI, pero no mantienen datos ni proporcionan directamente ninguna funcionalidad de la aplicación.

⁶ Éste es un principio clave en el patrón *Modelo-Vista-Controlador* (MVC). El MVC fue originalmente un patrón de pequeña escala de Smalltalk-80, y relacionaba los objetos de datos (modelos), los elementos gráficos de la GUI (vistas), y el manejo de los eventos del ratón y el teclado (controladores). Más recientemente, el término "MVC" también se ha adoptado por la comunidad de diseño distribuido para aplicarlo en los niveles de la arquitectura a gran escala. El Modelo es la Capa del Dominio, la Vista es la Capa de Presentación, y el Controlador son los objetos del flujo de trabajo en la capa de Aplicación.

Los motivos para la Separación Modelo-Vista son los siguientes:

- Dar soporte a definiciones de modelos cohesivos que se centren en los procesos del dominio, en lugar de preocuparse de las interfaces de usuario.
- Permitir separar el desarrollo de las capas del modelo y la interfaz de usuario.
- Minimizar el impacto de los cambios de los requisitos de la interfaz sobre la capa del dominio.
- Permitir que se conecten fácilmente otras vistas a una capa de dominio existente, sin afectar a la capa del dominio.
- Permitir múltiples vistas simultáneas sobre el mismo modelo del dominio, como una vista de la información sobre las ventas en formato tabular o mediante un diagrama de barras.
- Permitir la ejecución de la capa del modelo de manera independiente de la capa de interfaz de usuario, como en un sistema de procesamiento de mensajes o en modo de procesamiento por lotes.
- Permitir trasladar fácilmente la capa del modelo a otro framework de interfaz de usuario.

Separación Modelo-Vista y comunicación “ascendente”

¿Cómo pueden obtener las ventanas la información que tiene que mostrar? Normalmente, es suficiente que envíen mensajes a los objetos del dominio, preguntando sobre la información que luego mostrarán en elementos gráficos —un modelo de **escrutinio (polling)** o **tirar-desde-arriba (pull-from-above)** para mostrar las actualizaciones—.

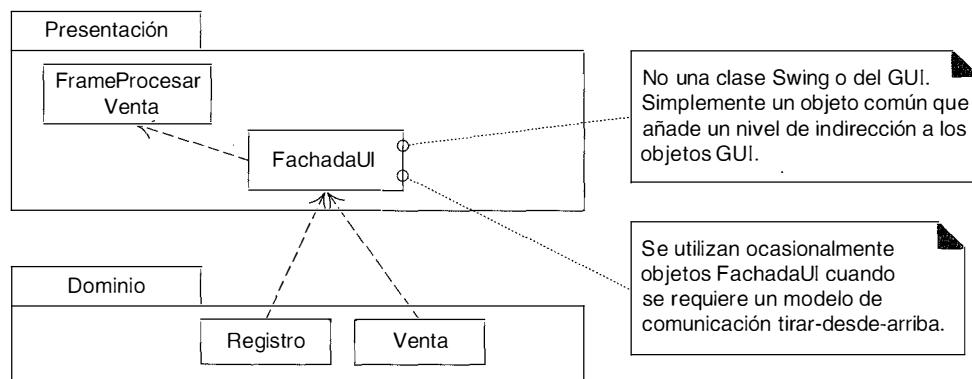


Figura 30.16. Ocasionalmente se utiliza una FachadaUI en la capa de Presentación para los diseños tirar-desde-arriba.

Sin embargo, un modelo de escrutinio a veces es insuficiente. Por ejemplo, comprobar cada segundo cientos de objetos para descubrir que sólo han cambiado uno o dos, lo cual se utiliza entonces para actualizar la información que se muestra en la GUI, no es

eficiente. En este caso es más eficiente que los pocos objetos del dominio que cambian se comuniquen con las ventanas para que provoque que se actualice la información que muestran cuando el estado del objeto del dominio cambia. Las situaciones típicas de este caso son:

- Aplicaciones de vigilancia, como la gestión de redes de telecomunicaciones.
- Aplicaciones de simulación que requieren visualización, como el modelado aerodinámico.

En estas situaciones, se requiere un modelo **empujar-desde-abajo** (*push-from-below*) para mostrar las actualizaciones. Debido a la restricción del patrón Separación Modelo-Vista, necesitamos establecer una comunicación “indirecta” desde los objetos inferiores hacia las ventanas —hacen subir la notificación de actualización desde abajo—.

Existen dos soluciones comunes:

1. El patrón Observador, haciendo que los objetos de la GUI simplemente parezcan objetos que implementan una interfaz como *PropertyListener*.
2. Un objeto fachada de Presentación. Es decir, añadir una fachada en la capa de Presentación que recibe las peticiones desde abajo. Es un ejemplo de la inclusión de Indirección para proporcionar Variaciones Protegidas si cambia la GUI. Por ejemplo, véase la Figura 30.16.

30.4. Lecturas adicionales

Existe abundante literatura sobre las arquitecturas en capas, tanto impresas como en la Web. Una serie de patrones en *Pattern Languages of Program Design*, volumen 1 [CS95] abordó por primer vez el tema en forma de patrones, aunque las arquitecturas en capas se usan y se escribe sobre ellas desde al menos los años sesenta; el volumen 2 continúa con patrones relacionados con las capas adicionales. *Pattern-Oriented Software Architecture* volumen 1 [BMRSS96] trata de manera adecuada el patrón Capas.

Capítulo 31

ORGANIZACIÓN DE LOS PAQUETES DE LOS MODELOS DE DISEÑO E IMPLEMENTACIÓN

*Si estuvieses arando un campo,
¿qué preferirías utilizar? Dos bueyes fuertes o 1.024 gallinas?*

Seymour Cray

Objetivos

- Organizar los paquetes para reducir el impacto de los cambios.
 - Conocer notación UML alternativa para la estructura de los paquetes.
-

Introducción

Si el equipo de desarrollo depende en gran medida de un paquete X, no es conveniente que X sea muy inestable (pasando por muchas versiones nuevas), puesto que incrementa el impacto en el equipo en cuanto a constantes re-sincronizaciones de las versiones y arreglos en el software dependiente que deja de funcionar en respuesta a los cambios en X (**destrozo de versiones**).

Esto nos suena y es obvio, pero algunas veces un equipo no presta atención a la identificación y estabilización de los paquetes de los que más depende, y termina experimentando más destrozo de versiones de lo que es necesario.

Este capítulo se fundamenta en la introducción que se hizo en el capítulo anterior de capas y paquetes, sugiriendo más heurísticas de grano fino para la organización de paquetes, para reducir este tipo de impacto de los cambios. El objetivo es crear un diseño de paquetes físicos robusto.

Uno sufre una organización de paquetes frágil, sensible a las dependencias, mucho más rápidamente en C++ que en Java, debido a las dependencias hiper-sensibles de compilación y enlace en C++; un cambio en una clase tiene un fuerte impacto en las dependencias transitivas, lo que nos lleva a la recompilación de muchas clases, y enlazar de

nuevo¹. Por tanto, estas sugerencias son útiles especialmente en proyectos C++, y menos para proyectos Java, Smalltalk, o C# (como ejemplos).

El útil trabajo de Robert Martin [Martin95], que ha tratado de abordar el diseño físico y el empaquetado de las aplicaciones C++, ha influido en algunas de las guías siguientes.

Diseño físico del código fuente en el Modelo de Implementación

Este asunto es un aspecto del **diseño físico** —el Modelo de Implementación del UP para el empaquetado del código fuente—.

Mientras estamos simplemente dibujando los diagramas del diseño de paquetes en una pizarra o herramienta CASE, podemos colocar de manera arbitraria los tipos en cualquier paquete funcionalmente cohesivo sin ningún impacto. Pero durante el diseño físico del código fuente —la organización de los tipos en unidades de versión física como “paquetes” de Java o C++— nuestras elecciones influirán en el nivel de impacto del desarrollador cuando tengan lugar los cambios en estos paquetes, si existen muchos desarrolladores compartiendo un código base común.

31.1. Guías para la organización de paquetes

Guía: Paquete de secciones verticales y horizontales funcionalmente cohesivas

El principio “intuitivo” básico es dividir en módulos en base a la cohesión funcional —se agrupan los tipos que están fuertemente relacionados en función de su participación en un objetivo común, servicio, colaboración, política y función—. Por ejemplo, todos los tipos en el paquete *FijarPrecios* de NuevaEra están relacionados con la política para fijar los precios a los productos. Las capas y los paquetes en el diseño de NuevaEra se organizan por grupos funcionales.

Normalmente es suficiente con conjeturas informales sobre la agrupación por funciones (“Creo que la clase *LíneaDeVenta* pertenece a *Ventas*”) pero, además, otro indicio de agrupamiento funcional es un grupo de tipos con fuerte acoplamiento interno y débil acoplamiento entre grupos. Por ejemplo, el *Registro* está acoplado fuertemente con la *Venta*, que está fuertemente acoplada con *LíneaDeVenta*.

El acoplamiento interno del paquete, o **cohesión relacional**, puede cuantificarse, aunque tal análisis formal raramente tiene una utilidad práctica. Para los curiosos, una medida es:

$$CR = \frac{\text{NúmeroDeRelacionesInternas}}{\text{NúmeroDeTipos}}$$

¹ En C++ los paquetes podrían realizarse como namespaces, pero es más probable que suponga la organización del código fuente en directorios físicos separados —uno por cada “paquete”—.

Donde el *NumeroDeRelacionesInternas* incluye las relaciones de atributos y parámetros, herencia, e implementaciones de interfaces entre los tipos del paquete.

Un paquete de 6 tipos con 12 relaciones internas tiene CR=2. Un paquete de 6 tipos con 3 relaciones entre los tipos tiene CR=0.5. Los números más altos dan a entender una mayor cohesión o relación en el paquete.

Nótese que esta medida es menos aplicable a los paquetes formados en su mayor parte por interfaces; es más útil para paquetes que contienen algunas clases de implementación.

Un valor CR muy bajo sugiere algunas de las siguientes cuestiones:

- El paquete contiene elementos no relacionados y no está bien factorizado.
- El paquete contiene elementos no relacionados y al diseñador no le importa deliberadamente. Esto es habitual con paquetes de utilidades o servicios dispares (ej. *java.util*), donde no es importante un valor alto o bajo de CR.
- Contiene una o más agrupaciones de subconjuntos de grupos con alto CR, pero el global no lo es.

Guía: Paquete de una familia de interfaces

Coloque una familia de *interfaces* relacionadas funcionalmente en un paquete separado —separado de las clases de implementación. Esto no es fundamental para el caso de una o dos interfaces relacionadas, sino cuando existe una familia de quizás tres o más interfaces. El paquete de la tecnología Java EJB *javax.ejb* es un ejemplo: es un paquete de al menos doce interfaces; las implementaciones están en paquetes separados.

Guía: Paquete por trabajo y por agrupaciones de clases inestables

El contexto para esta discusión es que los paquetes son normalmente la unidad básica del trabajo de desarrollo y de versiones. Es menos común trabajar y lanzar como versión únicamente una clase.

Suponga que 1) existe un paquete grande P1 con treinta clases, y 2) la tendencia en el trabajo es que un subconjunto concreto de diez clases (de C1 a C10) se modifica regularmente y se vuelve a lanzar.

En este caso, reagrupa P1 en P1-a y P1-b, donde P1-b contiene las diez clases sobre las que se trabaja frecuentemente.

De este modo, el paquete se ha reagrupado en subconjuntos más estables y menos estables, o de manera más general, en grupos relacionados con el trabajo. Esto es, si se trabaja de manera conjunta en la mayoría de los tipos de un paquete, entonces es conveniente agruparlos.

Idealmente, menos desarrolladores dependen de P1-b que de P1-a, y agrupando esta parte inestable en un paquete separado, no hay tantos desarrolladores que se vean

afectados por las nuevas versiones de P1-b como por los relanzamientos del paquete P1 original más grande.

Nótese que esta reagrupación se debe a una tendencia de trabajo que surge. Es difícil identificar de manera especulativa una buena estructura de paquetes en las primeras iteraciones. Evoluciona incrementalmente a lo largo de las iteraciones de la elaboración, y debería ser un objetivo de la fase de elaboración (porque es significativo para la arquitectura) estabilizar la mayoría de la estructura del paquete cuando se complete la fase.

Esta guía ilustra la estrategia básica: **reducir una dependencia extendida sobre paquetes inestables**.

Guía: Los más responsables son los más estables

Si los paquetes más responsables (de los que más se depende) son inestables, existe un mayor riesgo de extender el impacto de los cambios por las dependencias. Un caso extremo es, si un paquete de utilidades ampliamente utilizado como *com.foo.utilidades* cambia con frecuencia, podrían dejar de funcionar muchas cosas. Por tanto, la Figura 31.1 ilustra una estructura de dependencia apropiada.

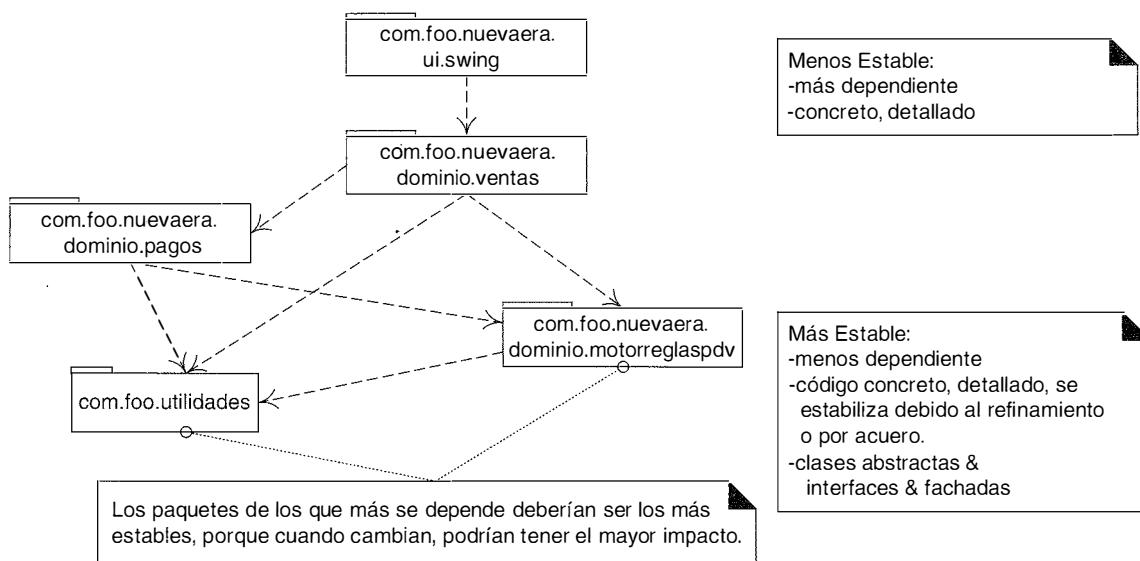


Figura 31.1. Los paquetes más responsables deberían ser más estables.

Visualmente, los paquetes inferiores en este diagrama deberían ser los más estables. Hay formas diferentes de incrementar la estabilidad en un paquete:

- Contiene sólo o en su mayor parte interfaces y clases abstractas.
 - Por ejemplo, *java.sql* contiene ocho interfaces y seis clases, y las clases son en su mayor parte tipos simples y estables como *Time* y *Date*.

- No depende de otros paquetes (es independiente), o depende de otros paquetes muy estables, o encapsula sus dependencias de manera que los dependientes no se ven afectados.
 - Por ejemplo, *com.foo.nuevaera.dominio.motorreglaspdv* oculta la implementación del motor de reglas detrás de un simple objeto fachada. Incluso si la implementación cambia, los paquetes dependientes no se ven afectados.
- Contiene código relativamente estable porque se implementó con mucho cuidado y se refinó antes de lanzar la versión.
 - Por ejemplo, *java.util*.
- Es obligatorio una planificación lenta de cambios.
 - Por ejemplo, *java.lang* el paquete central de las librerías de Java, simplemente no se le permite que cambie con frecuencia.

Guía: Separar los tipos independientes

Organice los tipos que se puedan utilizar independientemente o en contextos diferentes en paquetes separados. Sin una estimación cuidadosa, la agrupación según funcionalidades comunes podría no proporcionar el nivel adecuado de granularidad en la factorización de paquetes.

Por ejemplo, suponga que se ha definido un subsistema para los servicios de persistencia en un paquete *com.foo.servicios.persistencia*. En este paquete hay dos clases de utilidades/auxiliares muy generales *UtilidadesJDBC* y *OrdenesSQL*. Si hay utilidades generales para trabajar con JDBC (servicios Java para el acceso a una base de datos relacional), entonces se pueden utilizar de manera independiente del subsistema de persistencia, en cualquier ocasión que el desarrollador utilice JDBC. Por tanto, es mejor migrar estos tipos a un paquete separado, como *com.foo.utilidades.jdbc*. La Figura 31.2 lo ilustra.

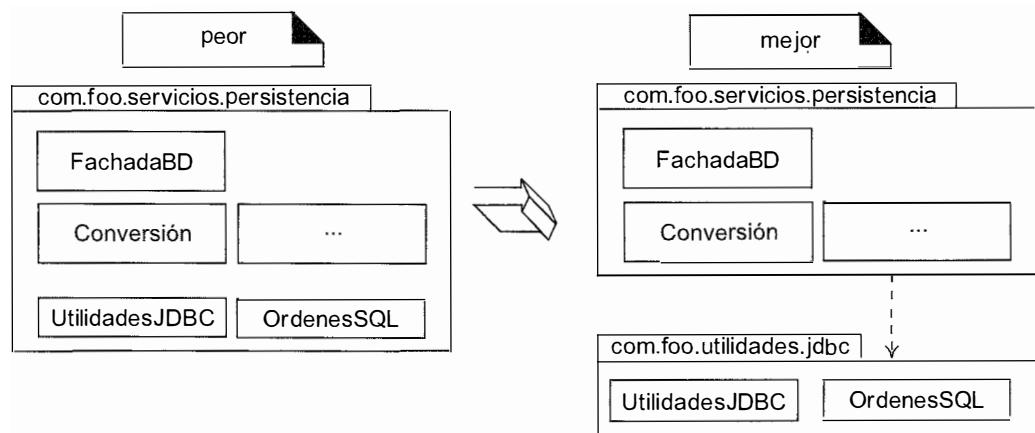


Figura 31.2. Separar los tipos independientes.

Guía: Utilice factorías para reducir la dependencia en paquetes concretos

Una forma de incrementar la estabilidad de los paquetes es reducir la dependencia de clases concretas de otros paquetes. La Figura 31.3 ilustra la situación “anterior”.

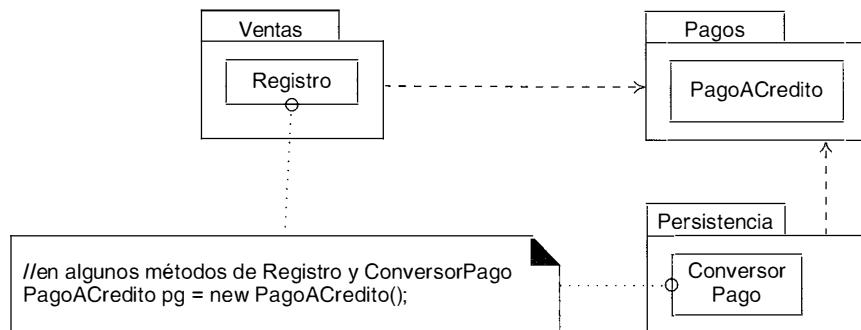


Figura 31.3. Acoplamiento directo de paquetes concretos debido a la creación.

Suponga que tanto el *Registro* como el *ConversorPago* (una clase que almacena/recupera los objetos pago en una base de datos relacional) crean instancias de *PagoACredito* del paquete *Pagos*. Un mecanismo para incrementar la estabilidad a largo plazo de los paquetes *Ventas* y *Persistencia* es detener la creación explícita de objetos de clases concretas definidas en otros paquetes (*PagoACredito* en *Pagos*).

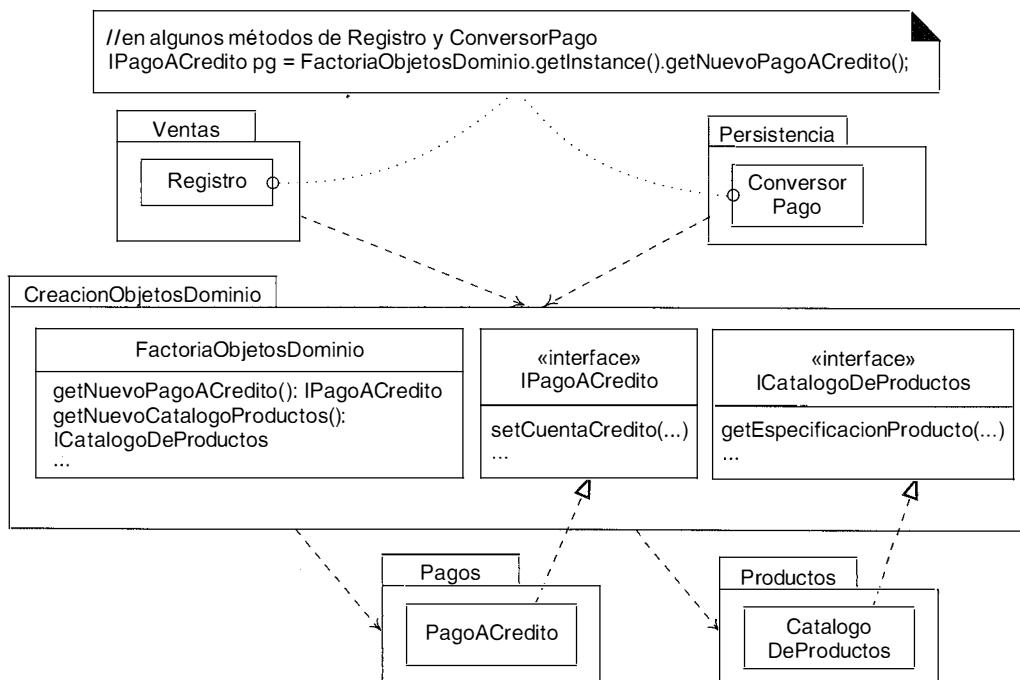


Figura 31.4. Reducción del acoplamiento con un paquete concreto utilizando un objeto factoría.

Podemos reducir el acoplamiento con este paquete concreto utilizando un objeto factoría que crea las instancias, pero cuyos métodos de creación devuelven objetos declarados en términos de interfaces en lugar de clases. Véase la Figura 31.4.

Patrón Factoría de Objetos del Dominio

El uso de factorías de objetos del dominio con interfaces para la creación de *todos* los objetos del dominio es un estilo de diseño común. He visto mencionarlo informalmente en la literatura sobre el diseño como el patrón Factoría de Objetos del Dominio, pero no conozco ninguna referencia en el que se describa formalmente como patrón.

Guía: Paquetes sin ciclos

Si un grupo de paquetes tiene un ciclo de dependencias entonces podrían necesitar que se les tratara como un paquete más grande en términos de una unidad de versión. Esto no es conveniente porque lanzando paquetes mayores (o paquetes agregados) incrementa la probabilidad de afectar a algo.

Hay dos soluciones:

1. Separar los tipos que participan en el ciclo en un paquete nuevo más pequeño.
2. Romper el ciclo con una interfaz.

Los pasos para romper el ciclo con una interfaz son:

1. Redefinir las clases de las que se depende en uno de los paquetes para implementar nuevas interfaces.
2. Definir las nuevas interfaces en un paquete nuevo.
3. Redefinir los tipos dependientes para que dependan de las interfaces del nuevo paquete, en lugar de las clases originales.

La Figura 31.5 ilustra esta estrategia.

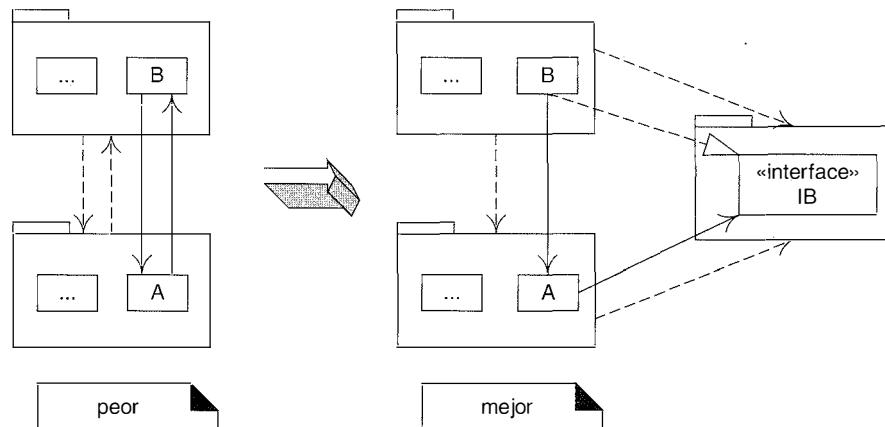


Figura 31.5. Rotura de un ciclo de dependencias.

31.2. Notación adicional de los paquetes en UML

Por último, a propósito de los paquetes, UML proporciona una notación alternativa para ilustrar paquetes internos y externos. Algunas veces es difícil dibujar una caja de paquete externo alrededor de paquetes internos. Las alternativas se muestran en la Figura 31.6.

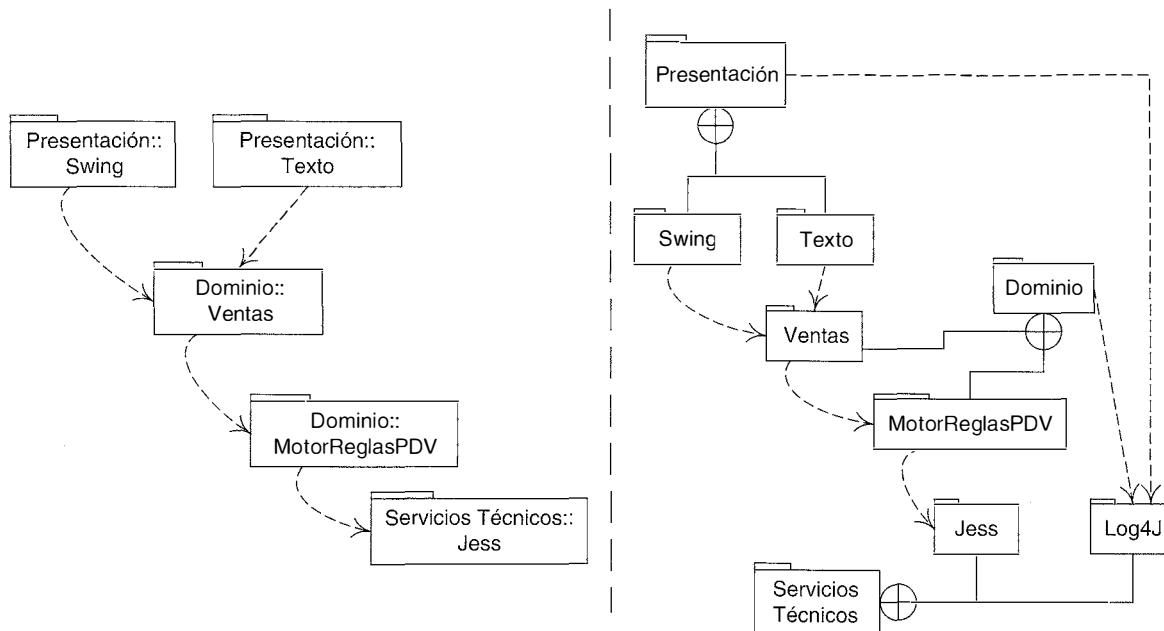


Figura 31.6. Enfoques UML alternativos para mostrar la estructura de paquetes, utilizando nombres de camino UML, o el símbolo de un círculo con una cruz.

31.3. Lecturas adicionales

La mayoría del trabajo detallado —no es sorprendente— sobre las mejoras en el diseño de paquetes para reducir el impacto de las dependencias procede de la comunidad de C++, aunque los principios se pueden aplicar a otros lenguajes. *Designing Object-Oriented C++ Applications Using the Booch Method* de Martin [Martin95] cubre bien el tema, al igual que *Large Scale C++ Software Design* [Lakos96]. El tema también se introduce en *Java 2 Performance and Idiom Guide* [GL99].

Capítulo 32

INTRODUCCIÓN AL ANÁLISIS ARQUITECTURAL Y EL SAD

Error, no hay teclado – presione F1 para continuar.

mensaje inicial de la BIOS del PC

Objetivos

- Crear tablas de factores de la arquitectura.
 - Crear memorándums técnicos que recojan las decisiones acerca de la arquitectura.
 - Conocer los principios básicos del diseño arquitectural.
 - Conocer recursos para aprender los patrones de arquitectura.
-

Introducción

La esencia del análisis arquitectural es identificar los factores que deberían influir en la arquitectura, entender su variabilidad y prioridad, y resolverlos. La parte difícil es conocer qué hay que preguntar, valorando los compromisos y conociendo las muchas formas de resolver un factor significativo desde el punto de vista de la arquitectura, que se extienden desde omisiones benignas, a diseños extravagantes, o a productos de tercera partes.

En el UP, los factores de la arquitectura se recogen en la Especificación Complementaria, y las decisiones de diseño que los resuelven se recogen en el **Documento de la Arquitectura del Software** (SAD, *Software Architecture Document*, descrito con más detalle casi al final de este capítulo).

El análisis arquitectural comienza pronto durante la fase de inicio, y es uno de los puntos de interés de la fase de elaboración; es una actividad de alta prioridad y que tiene mucha influencia en el desarrollo de software. Este tema se ha aplazado hasta este

punto del libro de manera que se pudieran presentar en primer lugar los fundamentos del A/DOO. Es una actividad útil para:

- reducir el riesgo de olvidar algo esencial en el diseño de los sistemas
- evitar dedicar excesivo esfuerzo a cuestiones de poca prioridad
- ayudar a alinear el producto con los objetivos del negocio

Este capítulo es una introducción a los pasos e ideas básicas del análisis arquitectural desde la perspectiva del UP; es decir, al método, en lugar de a los consejos y astucias de los arquitectos expertos. De este modo, no es un libro de recetas de cocina de soluciones de arquitectura —un tema amplio y dependiente del contexto que va más allá del alcance de este libro introductorio—. No obstante, el caso de estudio del PDV NuevaEra que se comenta en el capítulo proporciona ejemplos concretos de soluciones relacionadas con la arquitectura.

32.1. Análisis arquitectural

El **análisis arquitectural** trata de la identificación y resolución de los requisitos no funcionales del sistema (por ejemplo, calidad), en el contexto de los requisitos funcionales.

En el UP, el término comprende tanto la investigación arquitectural (identificación) como el diseño arquitectural (resolución). A continuación se presentan algunos ejemplos de muchas de las cuestiones que se tienen que identificar y resolver al nivel de la arquitectura:

- ¿Cómo afectan en el diseño los requisitos de fiabilidad y tolerancia a fallos?
 - Por ejemplo, en el PDV NuevaEra, ¿para qué servicios remotos (ej. calculador de impuestos) se les podrá mantener el servicio ante fallos mediante servicios locales? ¿Por qué? ¿Proporcionan exactamente los mismos servicios localmente que de manera remota, o existen diferencias?
- ¿Cómo afecta en la rentabilidad el coste de las licencias de los subcomponentes comprados?
 - Por ejemplo, el fabricante del excelente servidor de bases de datos, *SinPistas*, quiere el 2% de cada PDV NuevaEra que se venda, si se utiliza su producto como subcomponente. La utilización de su producto acelerará el desarrollo (y su salida al mercado) porque es robusto y proporciona muchos servicios, y lo conocen muchos desarrolladores, pero esto tiene un precio. ¿En lugar de eso debería utilizar el equipo el servidor de bases de datos *SuSQL* de libre distribución, menos robusto? ¿Con qué riesgo? ¿Cómo limita la facultad de cobrar por el producto NuevaEra?
- ¿Cómo afecta la distribución de los servicios en los requisitos de calidad y los requisitos funcionales?
 - Por ejemplo, utilizando un sistema calculador de impuestos remoto (único y centralizado) reduce la huella de cada cliente NuevaEra, reduce los costes de li-

cencia (sólo se necesita una copia), y minimiza el esfuerzo de configuración para un cliente determinado (cada instalación requiere ajustes semanales debido a cambios en las políticas del negocio y en las de legislación). Sin embargo, el servicio remoto reduce el tiempo de respuesta lo suficiente para que los impuestos sólo se puedan calcular una vez, después de introducir todos los artículos; uno no puede ver un precio parcial con impuestos tras cada línea de venta; y la llamada remota tarda mucho. También crea un único punto de fallo.

- • Cómo afectan al diseño los requisitos de adaptabilidad y de configuración?
 - Por ejemplo, la mayoría de las tiendas difieren en las reglas del negocio que quieren representar en sus aplicaciones de PDV. ¿Cuáles son las variaciones? ¿Cuál es la “mejor” forma de diseñarlas? ¿Cuáles son los criterios para la mejor? ¿Puede NuevaEra ganar más dinero exigiendo que se adapte la programación a cada cliente (y, ¿cuánto esfuerzo supondrá?), o con una solución que permita a los clientes que las adapten ellos mismos fácilmente? ¿“Más dinero” debería ser el objetivo a corto plazo?

Pasos comunes en el análisis arquitectural

Existen varios métodos para el análisis arquitectural. Comunes a la mayoría de ellos son algunas variaciones de los siguientes pasos:

1. Identificar y analizar los requisitos no funcionales que influyen en la arquitectura. Los requisitos funcionales también son relevantes (especialmente en términos de variabilidad o cambio), pero se les presta una atención completa a los no funcionales. En general, todos éstos podrían llamarse **factores de la arquitectura** (también conocidos como **controladores de la arquitectura**, *architectural drivers*).
 - Este paso podría caracterizarse como un análisis de requisitos ordinario, pero puesto que se lleva a cabo en el contexto de la identificación del impacto de la arquitectura y en la toma de decisiones sobre las soluciones de la arquitectura de alto nivel, se considera como parte del análisis arquitectural en el UP.
 - Por lo que se refiere al UP, algunos de estos requisitos se identificarán y recopilarán en líneas generales en la Especificación Complementaria o en los casos de uso durante la fase de inicio. Durante el análisis arquitectural, que tiene lugar al principio de la elaboración, el equipo investiga estos requisitos más detenidamente.
2. Para aquellos requisitos que influyen de manera significativa en la arquitectura, analizar las alternativas y crear soluciones que resuelvan el impacto. Éstas son las **decisiones sobre la arquitectura**.
 - Las decisiones varían desde “eliminar el requisito”, a una solución a medida, a “detener el proyecto”, o a “contratar un experto”.

Esta presentación introduce estos pasos básicos en el contexto del caso de estudio del PDV NuevaEra. Por simplicidad, evita cuestiones del despliegue de la arquitectura tales

como la configuración del hardware y del sistema operativo, que son muy sensibles al contexto y al momento en que se hace.

32.2. Tipos y vistas de la arquitectura

Algunas de las descripciones de arquitectura definen tipos diferentes, como la “arquitectura de aplicación” (asignación de características a componentes) o “arquitectura del sistema” (configuración del hardware y del sistema operativo).

En el UP, existe una especialización de la información parecida, pero se describen en “vistas” de la arquitectura, que resumen y destacan una perspectiva particular. Por ejemplo, la **vista lógica** de la arquitectura, que se introdujo en el Capítulo 30, resume la organización y la funcionalidad de los elementos del software importantes (como las capas) —es similar al término arquitectura de la aplicación—. La **vista de despliegue** resume la topología del sistema, las comunicaciones y la correspondencia entre los elementos ejecutables y los nodos de proceso —es análogo al término arquitectura del sistema.

El UP define seis vistas de la arquitectura, que se describen en detalle casi al final de este capítulo. Concretamente, las vistas combinan texto y diagramas, y —si se describen del todo— se recogen en el SAD.

El análisis arquitectural se relaciona con las vistas de la arquitectura porque las decisiones sobre la arquitectura se reflejan y describen en una o más vistas de la arquitectura.

32.3. La ciencia: identificación y análisis de los factores de la arquitectura

Factores de la arquitectura

Cualquiera y todos los requisitos FURPS+ pueden influir de manera significativa en la arquitectura de un sistema, variando desde la fiabilidad, a la planificación, a las habilidades y a las restricciones de coste. Por ejemplo, un caso de planificación ajustada, habilidades limitadas y dinero suficiente probablemente favorece contratar o alquilar especialistas externos, en lugar de construir todos los componentes dentro de la compañía.

Sin embargo, los factores que influyen con más fuerza en la arquitectura tienden a encontrarse en las categorías FURPS+ de alto nivel de funcionalidad, fiabilidad, rendimiento, soporte, implementación e interfaz (véase el Capítulo 5 para un desglose detallado). Curiosamente, son los atributos de calidad no funcionales (como la fiabilidad o el rendimiento) los que dan a una arquitectura concreta su sabor único, en lugar de sus requisitos funcionales. Por ejemplo, el diseño en el sistema NuevaEra para dar soporte a diferentes componentes de terceras partes con interfaces únicas, y el diseño para dar soporte a la conexión fácil de diferentes conjuntos de reglas del negocio.

En el UP, estos factores con implicaciones en la arquitectura se denominan **requisitos significativos para la arquitectura**. Utilizamos aquí “factores” para simplificar.

Se pueden caracterizar muchos factores técnicos y organizativos como *restricciones* que limitan la solución de alguna manera (como, debe ejecutarse en Linux, o el presupuesto para comprar componentes de terceras partes es X).

Escenarios de calidad

Cuando se definen los requisitos de calidad durante el análisis de los factores de la arquitectura, se recomienda el uso de los **escenarios de calidad**¹, ya que definen respuestas cuantificables (o por lo menos observables) y, por tanto, se pueden verificar. No sirve de mucho establecer de manera vaga “el sistema será fácil de modificar” sin ninguna medida de lo que eso significa.

Cuantificar algunas cosas, como los objetivos de rendimiento y el tiempo entre fallos, son prácticas bien conocidas, pero los escenarios de calidad amplían esta idea y promueve la recopilación de todos (o al menos, la mayoría) los factores como sentencias cuantificables.

Los escenarios de calidad son sentencias cortas de la forma <estímulo> <respuesta cuantificable>; por ejemplo:

- Cuando se envía la venta completa al calculador de impuestos remoto para añadir los impuestos, el resultado se devolverá en 2 segundos “la mayoría” de las veces, medido en un entorno de producción bajo condiciones de carga “media”.
- Cuando llega un informe de errores de un voluntario de una prueba de una versión beta de NuevaEra, responda con una llamada de teléfono en un día laborable.

Nótese que será necesario que el arquitecto de NuevaEra realice un estudio adicional y defina “la mayoría” y “media”; un escenario de calidad no es realmente válido hasta que no se puede probar, lo que implica que se ha especificado completamente. También, observe la calificación en el primer escenario de calidad en función del entorno en el que se aplica. Esto mejora algo la especificación de un escenario de calidad, verifica que lo pasa en un entorno de desarrollo ligeramente cargado, pero falla al evaluarlo en un entorno de producción realista.

Escoja sus Batallas

Una advertencia: la escritura de estos escenarios de calidad puede ser un espejismo de utilidad. Es fácil *escribir* estas especificaciones detalladas, pero no materializarlas. ¿Alguien las probará realmente alguna vez? ¿Cómo y por quién? Se requiere una fuerte dosis de realismo cuando se escriben; no tiene sentido listar muchos objetivos sofisticados si realmente nunca nadie acabará comprobándolos.

Esto está relacionado con la discusión acerca de “escoja sus batallas” que se presentó en un capítulo anterior sobre el patrón Variaciones Protegidas. ¿Cuáles son realmente los escenarios de calidad críticos que causan el éxito o el fracaso? Por ejemplo, en un sistema de reservas de vuelos, es realmente crítico para el éxito del sistema que se completen las

¹ Un término utilizado en varios métodos de la arquitectura promovido por el Instituto de Ingeniería del Software (SEI, *Software Engineering Institute*); por ejemplo, en el método de *Diseño Basado en la Arquitectura*.

transacciones de manera rápida y consistente bajo condiciones de carga alta —debe comprobarse sin lugar a dudas—. En el sistema NuevaEra, la aplicación realmente debe ser tolerante a fallos y mantener el servicio ante los fallos mediante copias locales de los servicios cuando fallan los remotos —sin duda se debe probar y validar debidamente—. Por tanto, céntrese en escribir los escenarios de calidad para las batallas importantes, y siga desde el principio hasta el final un plan para evaluarlos.

Descripción de los factores

Un objetivo importante del análisis arquitectural es entender la influencia de los factores, sus prioridades, y la manera en que varían (necesidad inmediata de flexibilidad y evolución futura). Por tanto, la mayoría de los métodos de la arquitectura (por ejemplo, véase [HNS00]) abogan por la creación de una tabla o árbol con variaciones de la siguiente información (el formato varía dependiendo del método). El siguiente estilo que se muestra en la Tabla 32.1 se denomina **tabla de factores**, que en el UP forma parte de la Especificación Complementaria.

Tabla 32.1. Ejemplo de tabla de factores.

Factor	Medidas y Escenarios de Calidad	Variabilidad (flexibilidad actual y futura evolución)	Impacto del factor (y su variabilidad) en las personas involucradas, arquitectura y otros factores	Prioridad para el éxito	Dificultad o Riesgo
Fiabilidad-Capacidad de recuperación					
Recuperación de los fallos en los servicios remotos	Cuando falla un servicio remoto, restablecer la conexión con él en 1 minuto, bajo una carga normal de la tienda en un entorno de producción.	Flexibilidad actual: nuestro EME dice que son aceptables (y convenientes) los servicios simplificados locales del lado del cliente hasta que sea posible la re-conexión. Evolución: en 2 años, algunas tiendas podrían tener la intención de pagar por una copia local completa de los servicios remotos (como el calculador de impuestos). ¿Probabilidad? Alta.	Impacto alto en el diseño a gran escala. A las tiendas realmente les disgusta cuando fallan los servicios remotos, ya que les impide o restringe el uso de un PDV para realizar las ventas.	A	M
...		

Leyenda: A: Alta. M: Media. EME: Experto en la Materia de Estudio.

Obsérvese el esquema de clasificación: *Fiabilidad—Capacidad de recuperación* (a partir de las categorías del FURPS+). Éste no se presenta como el mejor o el único esquema, pero resulta útil para agrupar los factores de la arquitectura en categorías. Por ejemplo,

certas categorías (como fiabilidad y rendimiento) están fuertemente relacionadas con la identificación y definición de los planes de pruebas y, por tanto, es conveniente agruparlas.

Los valores de los códigos básicos para la prioridad y el riesgo de A/M/B simplemente insinúan que se utilicen algunos códigos que el equipo encuentre útiles; existe una variedad de esquemas de codificación (numéricos y cualitativos) procedentes de diferentes métodos y estándares (como el ISO 9126) de arquitectura. Una advertencia: Si el esfuerzo extra de utilizar un esquema más complejo no le conduce a ninguna acción práctica, no merece la pena.

Factores y los artefactos del UP

El repositorio central de requisitos funcionales en el UP son los casos de uso, y ellos, junto con la Visión y la Especificación Complementaria, son una fuente de inspiración importante cuando se está creando una tabla de factores. En los casos de uso, se deberían revisar los *Requisitos especiales*, las *Variaciones de la tecnología* y los *Temas abiertos*, y reunir sus factores de la arquitectura implícitos o explícitos en la Especificación Complementaria.

Es razonable recoger al principio los factores relacionados con casos de uso en los casos de uso mientras se están creando, debido a que existe una relación obvia, pero al final es más conveniente (en cuanto a la gestión del contenido, traza y legibilidad) reunir todos los factores de la arquitectura en un sitio —en la tabla de factores en la Especificación Complementaria—.

Caso de Uso UC1: Procesar Venta

Escenario principal de éxito:

1. ...

Requisitos especiales:

- El tiempo de respuesta para la autorización de crédito será de 30 segundos el 90% de las veces
- De algún modo, queremos que el sistema se recupere de manera robusta cuando falla el acceso a servicios remotos, como el sistema de inventario.
- ...

Lista de tecnología y variaciones de datos:

- 2a. El identificador del artículo se introduce por medio de un escáner láser de código de barras (si está presente el código de barras) o del teclado.
...

Temas abiertos:

- ¿Cuáles son las variaciones en la ley de impuestos?
- Estudiar las cuestiones sobre la recuperación de los servicios remotos.

32.4. Ejemplo: tabla de factores parcial de la arquitectura del PDV NuevaEra

La tabla de factores parcial de la Tabla 32.2 muestra algunos factores relacionados con la discusión posterior.

Tabla 32.2. Tabla de factores parcial para el análisis arquitectural de NuevaEra.

Factor	Medidas y Escenarios de Calidad	Variabilidad (flexibilidad actual y futura evolución)	Impacto del factor (y su variabilidad) en las personas involucradas, arquitectura y otros factores	Prioridad para el éxito	Dificultad o Riesgo
Fiabilidad—Capacidad de recuperación					
Recuperación de los fallos en los servicios remotos.	Cuando falla un servicio remoto, restablecer la conexión con él en 1 minuto, bajo una carga normal de la tienda en un entorno de producción.	<p>Flexibilidad actual: nuestro EME dice que son aceptables (y convenientes) los servicios simplificados locales del lado del cliente hasta que sea posible la reconexión.</p> <p>Evolución: en 2 años, algunas tiendas podrían tener la intención de pagar por una copia local completa de los servicios remotos (como el calculador de impuestos). ¿Probabilidad? Alta.</p>	<p>Impacto alto en el diseño a gran escala.</p> <p>A las tiendas realmente les disgusta cuando fallan los servicios remotos, ya que les impide o restringe el uso de un PDV para realizar las ventas.</p>	A	M
Recuperación de los fallos en bases de datos de productos remotas.	Como arriba.	<p>Flexibilidad actual: nuestro EME dice que es aceptable (y conveniente) que se almacene en el lado del cliente información acerca de los productos “más comunes” hasta que sea posible la reconexión.</p> <p>Evolución: en 3 años, las soluciones de almacenamiento masivo y replicación en el lado del cliente, serán baratas y efectivas, permitiendo que se mantenga una copia completa permanente y así el uso local. ¿Probabilidad? Alta.</p>	Como arriba.	A	M
Soporte—Adaptabilidad					
Dar soporte a muchos servicios de terceras partes (calculador de impuestos, inventario, RRHH, contabilidad). Variarán en cada instalación.	Cuando se deba integrar un nuevo servicio de terceras partes, se puede hacer con un esfuerzo de 10 días persona.	<p>Flexibilidad actual: como se describe en el factor.</p> <p>Evolución: ninguna.</p>	<p>Se requiere para la aprobación del producto.</p> <p>Poco impacto en el diseño.</p>	A	B

(Continúa)

Tabla 32.2. Tabla de factores parcial para el análisis arquitectural de NuevaEra. (Continuación)

Factor	Medidas y Escenarios de Calidad	Variabilidad (flexibilidad actual y futura evolución)	Impacto del factor (y su variabilidad) en las personas involucradas, arquitectura y otros factores	Prioridad para el éxito	Dificultad o Riesgo
--------	---------------------------------	---	--	-------------------------	---------------------

Soporte—Adaptabilidad (continuación)

¿Dar soporte a terminales PDA inalámbricos para los clientes del PDV?	Cuando se añade el soporte a estos terminales, no requiere que se cambie el diseño de las capas de la arquitectura que no tienen que ver con el UI.	Flexibilidad actual: no se requiere en este momento. Evolución: en 3 años, pensamos que es muy probable que el mercado demande "PDAs" inalámbricos para los clientes del PDV.	Impacto alto en el diseño en cuanto a las variaciones protegidas de muchos elementos. Por ejemplo, los sistemas operativos y las UIs son diferentes en dispositivos pequeños.	B	A
---	---	--	---	---	---

Otros—Legal

Se deben aplicar la legislación de impuestos actual.	Cuando el auditor evalúe si se ajusta, encontrará que se ajusta al 100%. Cuando cambia la legislación de los impuestos, estarán operativas dentro del plazo marcado por el gobierno.	Flexibilidad actual: la conformidad es inflexible, pero la legislación de los impuestos pueden cambiar casi semanalmente debido a que hay muchas leyes y niveles de impuestos del gobierno (nacional, regional,...). Evolución: ninguna.	Fallar en el cumplimiento es un delito. Influye en los servicios de cálculo de impuestos. Es difícil escribir nuestro propio servicio —leyes complejas, cambios constantes, la necesidad de seguir la pista a todos los niveles de gobierno—. Aunque, riesgo fácil/bajo si se compra un paquete.	A	B
--	---	---	---	---	---

Leyenda: A: Alta; M: Media; B: Baja; EME: Experto en la Materia de Estudio.

32.5. El arte: resolución de los factores de la arquitectura

Uno podría decir que la *ciencia* de la arquitectura es la recolección y organización de la información sobre los factores de la arquitectura, como en la tabla de factores. El *arte* de la arquitectura es tomar las decisiones acertadas para resolver estos factores, teniendo en cuenta los compromisos, interdependencias y prioridades.

Los arquitectos expertos conocen una variedad de áreas (por ejemplo, estilos y patrones de arquitectura, tecnologías, productos, peligros y tendencias) y las aplican a sus decisiones.

Registro de alternativas, decisiones y la motivación de la arquitectura

Ignorando por ahora los principios de la toma de decisiones sobre la arquitectura, casi todos los métodos de arquitectura recomiendan mantener un registro de las soluciones alternativas, decisiones, factores que influyen, y el motivo de las cuestiones y decisiones relevantes.

A tales registros se les ha llamado **memorándums técnicos** [Cunningham96], **tarjetas de cuestiones** [HNS00], y **documentos de técnicas de arquitectura** (propuestas de arquitectura del SEI), con diversos grados de formalidad y sofisticación. En algunos métodos, estos memorándums son la base para todavía otro paso de revisión y refinamiento.

En el UP, los memorándums se deben recoger en el SAD.

Un aspecto importante de los memorándums técnicos es la *motivación* o los fundamentos. Cuando un futuro desarrollador o arquitecto necesita modificar el sistema², es enormemente útil entender los motivos que están detrás del diseño, como *por qué* se escogió un enfoque concreto para la recuperación de los fallos de los servicios remotos en el PDV NuevaEra y se rechazaron otros, con el objeto de tomar decisiones con conocimiento de causa sobre los cambios del sistema.

Es importante explicar los motivos para rechazar las alternativas, ya que durante la evolución futura del producto, un arquitecto podría reconsiderar estas alternativas, o por lo menos querer conocer qué alternativas se consideraron, y por qué se escogió una.

A continuación presentamos un memorándum técnico de ejemplo que recoge una decisión sobre la arquitectura para el PDV NuevaEra. Por supuesto, el formato exacto no es importante. Manténgalo simple y recoja únicamente la información que ayudará a los futuros lectores a tomar decisiones con conocimiento de causa cuando estén cambiando el sistema.

Memorándum Técnico

Asunto: Fiabilidad—Recuperación de fallos en los servicios remotos

Resumen de la solución: Ubicación transparente utilizando un servicio de búsqueda, mantenimiento del servicio ante los fallos pasando de remoto a local, y replicación parcial del servicio.

Factores

- Recuperación robusta de los fallos en los servicios remotos (ej. calculador de impuestos, inventario)

² ¡O cuando han pasado cuatro semanas y el arquitecto original ha olvidado su propia motivación!

- Recuperación robusta de los fallos en la base de datos de productos (ej. descripción de productos y precios) remota

Solución

Conseguir variaciones protegidas con respecto a la ubicación de los servicios utilizando un Adaptador creado en una FactoriaDeServicios. Donde sea posible, ofrecer implementaciones locales de servicios remotos, normalmente con comportamiento simplificado o restringido. Por ejemplo, el calculador de impuestos local utilizará porcentajes de impuestos constantes. La base de datos con información de los productos local será una pequeña caché de los productos más comunes. Se almacenarán las actualizaciones del inventario y se remitirán cuando se restablezca la conexión.

Véase también el memorándum técnico de *Adaptabilidad—Servicios de terceras partes* para conocer los aspectos de adaptabilidad de esta solución, porque las implementaciones de los servicios remotos varían en cada instalación.

Para satisfacer los escenarios de calidad de reconexión con los servicios remotos ASAP, utilizar para los servicios objetos Proxy inteligentes, que en cada llamada a un servicio comprobarán si se puede reactivar el servicio remoto y lo redireccionan a éste cuando es posible.

Motivación

¡Las tiendas realmente no quieren dejar de vender! Por tanto, si el PDV NuevaEra ofrece este nivel de fiabilidad y recuperación, será un producto muy atractivo, ya que ninguno de nuestros competidores proporciona esta capacidad. La pequeña caché de productos está motivada por la gran limitación de recursos en el lado del cliente. El calculador de impuestos de terceras partes real no se duplica en el cliente fundamentalmente debido a que el coste de las licencias es más alto y a los esfuerzos de configuración (puesto que cada instalación del calculador requiere ajustes casi semanales). Este diseño también soporta los puntos de evolución de los deseos de futuros clientes y es capaz de replicar los servicios permanentemente como el calculador de impuestos en cada terminal del cliente.

Cuestiones sin resolver

Ninguna

Alternativas consideradas

Una calidad de “nivel oro” del contrato de servicio con los servicios de autorización de crédito para mejorar la fiabilidad. Estaba disponible, pero era demasiado caro.

Nótese como se ilustra en este ejemplo —y es un punto clave— que una decisión sobre la arquitectura descrita en un memorándum técnico podría resolver un grupo de factores, no sólo uno.

Prioridades

Hay una jerarquía de objetivos que guían las decisiones sobre la arquitectura:

1. Restricciones inflexibles, entre las que se encuentran ajustarse a las normas legales y de seguridad.
 - El PDV NuevaEra debe aplicar correctamente las leyes de impuestos.
2. Objetivos del negocio.
 - Demo de las características relevantes listo para la feria POSWorld de Hamburgo dentro de 18 meses.

- Tiene cualidades y características atractivas para los grandes almacenes de Europa (por ejemplo, que soporte diferentes monedas y reglas de negocio a medida).
3. Todos los otros objetivos.
- A menudo se les pueden seguir la pista hacia atrás hasta establecer directamente objetivos del negocio, pero son indirectos. Por ejemplo, se podría establecer la traza entre “fácilmente extensible: puede añadir <alguna unidad funcional> en 10 semanas persona” y el objetivo del negocio de “nueva versión cada seis meses”.

En el UP, muchos de estos objetivos se recogen en el artefacto de Visión. Recuerde que los valores de la *Prioridad para el éxito* en la tabla de factores debería reflejar la prioridad de estos objetivos.

Existe un aspecto distintivo de la toma de decisiones a este nivel frente al diseño de objetos a pequeña escala: uno tiene que considerar simultáneamente más objetivos (y a menudo que influyen globalmente) y sus compromisos. Además, los objetivos del negocio pasan a ser cruciales para las decisiones técnicas (o al menos deberían). Por ejemplo:

Memorándum Técnico

Asunto: Legal—Cumplimiento de la legislación de impuestos

Resumen de la solución: Comprar un componente para el cálculo de impuestos.

Factores

- Se deben aplicar, por ley, la legislación de impuestos actuales.

Solución

Comprar un calculador de impuestos con un acuerdo de licencia para recibir actualizaciones de la legislación de impuestos actuales. Nótese que se podrían utilizar diferentes calculadores en instalaciones distintas.

Motivación

Acelerar la salida al mercado, corrección, requisitos de mantenimiento bajos, y desarrolladores felices (véase las alternativas). Estos productos son costosos, lo que afecta a nuestros objetivos del negocio de política de contención de costes y de política de fijación de precios, pero la alternativa se considera inaceptable.

Cuestiones sin resolver

¿Cuáles son los productos líderes y sus cualidades?

Alternativas consideradas

¿Qué el equipo de NuevaEra construya uno? Se estima que puede llevar mucho tiempo, ser propenso a fallos, y crea una responsabilidad de mantenimiento continua, costosa y sin interés (para los desarrolladores de la compañía), que afecta al objetivo de “desarrolladores felices” (seguramente, el objetivo más importante de todos).

Prioridades y puntos de evolución: ingeniería por defecto o en exceso

Otra característica distintiva de la toma de decisiones sobre la arquitectura es establecer prioridades de acuerdo a la probabilidad de los **puntos de evolución** —puntos de variabilidad o cambio que *podrían* surgir en el futuro—. Por ejemplo, en NuevaEra, existe una posibilidad de que se deseé terminales clientes portátiles e inalámbricos. El diseño para esto influye de manera significativa debido a las diferencias en los sistemas operativos, interfaces de usuario, recursos hardware, etcétera.

La compañía podría gastarse una cantidad enorme de dinero (e incrementar diversos riesgos) para conseguir esta “futura necesidad”. Si en el futuro resulta que no es relevante, hacerlo sería un ejercicio muy caro de sobre-ingeniería. Nótese también que se puede sostener que las futuras necesidades rara vez se dan, puesto que son especulaciones; incluso si ocurre el cambio previsto, es probable que se produzcan algunos cambios en el diseño supuesto.

Por otro lado, las futuras necesidades contra el problema de datos Y2K habría sido dinero bien empleado; en lugar de eso, hubo un esfuerzo de ingeniería por defecto con un resultado tremadamente costoso.

El arte del arquitecto es conocer qué batallas merece la pena pelear —dónde merece la pena invertir en diseños que protejan contra cambios evolutivos—.

Para decidir si se deberían evitar “futuras necesidades” prematuras, considere realmente el escenario de posponer el cambio para el futuro, cuando se requiera. ¿Cuánto del diseño y del código tendrá que cambiar en realidad? ¿Cuál será el esfuerzo? Quizás un estudio detallado del cambio potencial revelará que lo que al principio se consideró una cuestión gigantesca contra la que protegerse, se estima que el esfuerzo será sólo de unas pocas semanas-persona.

Esto es precisamente un problema difícil: “La predicción es muy difícil, especialmente si trata sobre el futuro” (atribuido, aunque sin comprobar, a Niels Bohr).

Principios básicos de diseño de la arquitectura

Los principios básicos de diseño que se han estudiado en gran parte de este libro que se aplicaron al diseño de objetos a pequeña escala, todavía son principios de gran influencia al nivel de la arquitectura a gran escala:

- Bajo acoplamiento.
- Alta cohesión.
- Variaciones protegidas (interfaces, indirección, servicio de búsqueda, etcétera).

Sin embargo, la granularidad de los componentes es mayor —se trata de bajo acoplamiento entre aplicaciones, subsistemas, o procesos, en lugar de entre pequeños objetos—.

Además, a esta escala más amplia, hay más o diferentes mecanismos para conseguir cualidades como el bajo acoplamiento y variaciones protegidas. Por ejemplo, considere el siguiente memorándum técnico:

Memorándum Técnico
Asunto: Adaptabilidad—Servicios de terceras partes

Resumen de la solución: Variaciones Protegidas utilizando interfaces y adaptadores.

Factores

- Dar soporte a muchos y cambiables servicios de terceras partes (calculadores de impuestos, autorización de crédito, inventario...).

Solución

Conseguir variaciones protegidas como sigue: Analizar varios productos para el cálculo de impuestos comerciales (y así sucesivamente para las otras categorías de productos) y construir interfaces comunes para el mínimo común denominador de funcionalidades. Entonces utilizar la Indirección mediante el patrón Adaptador. Es decir, crear un objeto Adaptador de recursos que implementa la interfaz y actúa como conexión y traductor con un calculador de impuestos back-end concreto.

Véase también el memorándum técnico *Fiabilidad—Recuperación de fallos en los servicios remotos* para conocer los aspectos de la ubicación transparente de esta solución.

Motivación

Simple. Comunicación más barata y más rápida que utilizando un servicio de intercambio de mensajes (ver alternativas), y en cualquier evento, un servicio de intercambio de mensajes no se puede utilizar para conectar directamente con el servicio de autorización de crédito externo.

Cuestiones sin resolver

¿Originarán las interfaces del mínimo común denominador un problema imprevisto, tal como ser demasiado limitada?

Alternativas consideradas

Aplicar indirección utilizando un servicio de intercambio de mensajes o publicar-suscribir (ej. una implementación JMS) entre el cliente y el calculador de impuestos, con adaptadores. Pero no se puede utilizar directamente con un autorizador de crédito, costoso (para los fiables), y más fiabilidad en la entrega de mensajes de lo que se necesita en la práctica.

El hecho es que al nivel de la arquitectura, existen normalmente nuevos mecanismos para conseguir variaciones protegidas (y otros objetivos), con frecuencia en colaboración con componentes de terceras partes, como utilizar el Servicio de Mensajes de Java (JMS; *Java Messaging Service*) o el servidor EBJ.

Separación de intereses y localización del impacto

Otro principio básico que se aplica durante el análisis arquitectural es conseguir **separación de intereses**. Esto también es aplicable en la escala de pequeños objetos, pero alcanza relevancia durante el análisis arquitectural.

Los **intereses transversales** son aquéllos con amplia aplicación o influencia en el sistema, como la persistencia de datos o la seguridad. Uno *podría* diseñar el soporte a la persistencia en la aplicación NuevaEra de manera que cada objeto (que contiene código de la lógica de la aplicación) se comunique él mismo con una base de datos para almacenar sus datos. Esto entrelazaría el interés de la persistencia con el de la lógica de la

aplicación, en el código fuente de las clases —así también con la seguridad—. La cohesión disminuye y aumenta el acoplamiento.

En cambio, diseñando para conseguir una separación de intereses se separa el soporte a la persistencia y el soporte a la seguridad en “cosas” separadas (existen mecanismos muy diferentes para esta separación). Un objeto con lógica de la aplicación únicamente tiene lógica de la aplicación, no lógica de persistencia o de seguridad. Igualmente, un subsistema de persistencia se centra en las cuestiones de persistencia, no de seguridad. Un subsistema de seguridad no lleva a cabo la persistencia.

La separación de intereses es una forma de pensar a gran escala acerca del bajo acoplamiento y alta cohesión al nivel de arquitectura. También se aplica a objetos a pequeña escala, porque su ausencia da lugar a objetos sin cohesión que tienen muchas áreas de responsabilidad. Pero es una cuestión de arquitectura especialmente porque los intereses son amplios, y las soluciones conllevan elecciones de diseño importantes y fundamentales.

Existen por lo menos tres técnicas a gran escala para conseguir la separación de intereses:

1. Tratar el interés como un módulo en un componente separado (por ejemplo, un subsistema) e invocar sus servicios.
 - Éste es el enfoque más común. Por ejemplo, en el sistema NuevaEra, el soporte a la persistencia podría colocarse en un subsistema aparte denominado *servicio de persistencia*. Mediante una fachada, puede ofrecer una interfaz público de servicios a otros componentes. Las arquitecturas en capas también ilustran esta separación de intereses.
2. Utilizar decoradores.
 - Éste es el segundo enfoque más común; se dio a conocer en primer lugar en el Servicio de Transacciones de Microsoft, y posteriormente con los servidores EJB. En este enfoque, un interés (como la seguridad) decora otros objetos con un objeto Decorador que encapsula el objeto interno e interpone el servicio. Al Decorador se le denomina **contenedor** en la terminología EJB. Por ejemplo, en el sistema de PDV NuevaEra, el control de seguridad a los servicios externos como el sistema de RRHH se puede conseguir con un contenedor EJB que añade las comprobaciones de seguridad en el Decorador externo, alrededor de la lógica de la aplicación del objeto interno.
3. Utilizar post-compiladores y tecnologías orientadas a aspectos.
 - Por ejemplo, con entidades *bean* EJB uno puede añadir el soporte a la persistencia a clases como *Venta*. Uno especifica en un fichero de descripción de propiedades las características de persistencia de la clase *Venta*. Entonces, un post-compilador (que es otro compilador que se ejecuta después de un compilador “ordinario”) añadirá el soporte a la persistencia necesario en una clase *Venta* modificada (modificando únicamente los byte-codes) o una subclase. El desarrollador continúa viendo la clase original como una clase “ limpia” que sólo contiene la lógica de la aplicación. Otra variación son las tecnologías **orientadas a aspectos** como AspectJ

(www.aspectj.org), que de manera similar da soporte a entrelazar post-compilación de intereses transversales en el código, de manera transparente para el desarrollador. Estos enfoques mantienen la ilusión de la separación durante el trabajo de desarrollo, y entrelazan el interés antes de la ejecución.

Promoción de patrones de arquitectura

Un estudio de los patrones de arquitectura y del modo en el que se podrían aplicar (o aplicar mal) al caso de estudio NuevaEra queda fuera del alcance de este texto introductorio. Sin embargo, a continuación presentamos algunas indicaciones:

Probablemente el mecanismo más común para conseguir bajo acoplamiento, variaciones protegidas y separación de intereses al nivel de la arquitectura es el patrón Capas, que se introdujo en un capítulo anterior. Éste es un ejemplo de la técnica más común de separación —tratar los intereses como módulos, en componentes separados o capas—.

Existe un amplio y creciente cuerpo de material escrito sobre patrones de arquitectura. Estudiarlos es la manera más rápida que conozco de aprender soluciones de la arquitectura. Por favor, vea las lecturas recomendadas.

32.6. Resumen de los temas del análisis arquitectural

Un asunto a destacar es que las cuestiones “arquitecturales” están especialmente relacionadas con los requisitos no funcionales, y conllevan la percepción del contexto del negocio o del mercado de la aplicación. Al mismo tiempo, no se pueden ignorar los requisitos funcionales (por ejemplo, procesar ventas); éstos proporcionan el contexto en el que se deben resolver estas cuestiones. Por otro lado, la identificación de su variabilidad es significativa para la arquitectura.

Un segundo tema es que las cuestiones arquitecturales implican problemas al nivel del sistema, a gran escala y amplios, cuya resolución normalmente conlleva decisiones de diseño a gran escala o fundamentales; por ejemplo, la elección de —o incluso el uso de— un servidor de aplicación.

Un tercer tema en el análisis arquitectural son las interdependencias y compromisos. Por ejemplo, la mejora de la seguridad podría afectar al rendimiento o a la facilidad de uso, y la mayoría de las opciones afectan al coste.

Un cuarto tema del análisis arquitectural es la generación y evaluación de soluciones alternativas. Un arquitecto experto puede ofrecer soluciones de diseño que impliquen la construcción de nuevo software, y también sugerir soluciones (o soluciones parciales) que utilicen software y hardware comercial o de libre distribución. Por ejemplo, la recuperación en un servidor remoto del PDV NuevaEra se puede conseguir mediante el diseño y programación de procesos “guardianes (*watchdog*)”, o quizás a través de la agrupación, duplicación y los servicios para sobreponerse a fallos que ofrecen algunos sistemas operativos y componentes hardware. Los buenos arquitectos conocen los productos hardware y software de tercera partes.

La definición inicial de cuestiones de la arquitectura proporciona el marco para el modo de pensar sobre el tema de la arquitectura: identificando los puntos con implicaciones a gran escala o a nivel del sistema, y resolviéndolos.

Un análisis arquitectural se preocupa de la identificación y resolución de los requisitos no funcionales (p.e. calidad) del sistema, en el contexto de los requisitos funcionales.

32.7. Análisis arquitectural en el UP

Advertencia: Análisis arquitectural en cascada

Con frecuencia, los métodos y libros sobre el análisis arquitectural implícitamente fomentan extensas decisiones de diseño sobre la arquitectura siguiendo un estilo en cascada antes de la implementación. En el desarrollo iterativo y el UP, aplique estas ideas en el contexto de pequeñas etapas, retroalimentación y adaptación, en lugar de pretender resolver completamente la arquitectura antes de programar. Aborde la implementación de las soluciones más arriesgadas o más difíciles en las primeras iteraciones, y ajuste las soluciones sobre la arquitectura en base a la retroalimentación y al conocimiento que adquiere.

Información sobre la arquitectura en los artefactos del UP

- Los factores de la arquitectura (por ejemplo, en una tabla de factores) se recogen en la Especificación Complementaria.
- Las decisiones sobre la arquitectura se recogen en el SAD. Esto incluye los memoriales técnicos y las descripciones de las vistas de la arquitectura.

El SAD y sus vistas de la arquitectura

Además de los diagramas UML de paquetes, clases e interacciones, el SAD es otro artefacto clave del Modelo de Diseño del UP. Éste describe las grandes ideas de la arquitectura, incluyendo las decisiones sobre el análisis arquitectural. Prácticamente, es una *ayuda de aprendizaje* para los desarrolladores que necesitan entender las ideas esenciales del sistema.

Cuando alguien se une al equipo, un jefe de proyecto puede decir, “¡Bienvenido al proyecto NuevaEra! Por favor, vaya al sitio web del proyecto y lea el SAD de 10 páginas para obtener una visión de las ideas más importantes”. Durante una versión posterior, cuando trabaja en el sistema gente nueva, el SAD constituye una ayuda clave para el aprendizaje.

Por tanto, se debe escribir teniendo en mente quien lo leerá y el objetivo: ¿qué necesita decir (y representar en UML) que rápidamente ayudará a alguien a entender las ideas principales del sistema?

La esencia del SAD es un resumen de las decisiones sobre la arquitectura (como con los memorándums técnicos) y las vistas de la arquitectura del UP.

Vistas de la arquitectura en el SAD

Tener una arquitectura es una cosa, describirla es algo más.

En [Kruchten95], se fomenta la influyente idea de describir una arquitectura con múltiples vistas. La idea esencial de una **vista de la arquitectura** es ésta:

Vista de la arquitectura

Una vista de la arquitectura del sistema desde una perspectiva dada; se centra sobre todo en la estructura y modularidad de los componentes fundamentales y en los principales flujos de control [RUP].

Un aspecto importante de la vista que se obvia en esta definición del RUP es la *motivación*. Es decir, una vista de la arquitectura debería explicar por qué la arquitectura es como es.

Una vista de la arquitectura es una ventana sobre el sistema desde una perspectiva particular que destaca la información relevante o ideas claves, e ignora el resto.

Una vista de la arquitectura es una herramienta de comunicación, enseñanza o de reflexión; se representa con texto y diagramas UML.

En el UP, se sugieren seis vistas de la arquitectura (se permiten más, como una vista de seguridad)³. Todas son opcionales, pero se recomienda documentar por lo menos las vistas lógica, de proceso, de caso de uso, y de despliegue. Las seis vistas son:

1. Lógica

- Organización conceptual del software en función de las capas, subsistemas, paquetes, frameworks, clases e interfaces más importantes. También resume la funcionalidad de los elementos del software importantes, como cada subsistema.
- Muestra los escenarios de realización de casos de uso (como diagramas de interacción) destacados que ilustran los aspectos claves del sistema.
- Una vista sobre el Modelo de Diseño del UP, visualizada con diagramas de paquetes, clases e interacción de UML.

2. Proceso

- Procesos e hilos de ejecución. Sus responsabilidades, colaboraciones y la asignación a ellos de los elementos lógicos (capas, subsistemas, clases,...).
- Una vista sobre el Modelo de Diseño del UP, visualizada con diagramas de clases e interacción de UML, utilizando la notación UML para procesos e hilos.

³ Las primeras versiones del UP describían las “4+1” vistas como se definen en [Kruchten95], que evolucionaron a las seis vistas.

3. Despliegue

- Despliegue físico de los procesos y componentes sobre los nodos de proceso, y la configuración de la red física entre los nodos.
- Una vista sobre el Modelo de Despliegue del UP, visualizada con los diagramas de despliegue de UML. Normalmente, la “vista” es simplemente el modelo completo en lugar de un subconjunto, ya que todo es relevante. Véase el Capítulo 38 para conocer la notación para el diagrama de despliegue de UML.

4. Datos

- Vista global del esquema de datos persistentes, la correspondencia del esquema de objetos a datos persistentes (normalmente en una base de datos relacional), el mecanismo de correspondencia de objetos a una base de datos, procedimientos almacenados en la base de datos y disparadores (*triggers*).
- Una vista sobre el Modelo de Datos del UP, visualizada con diagramas de clases de UML que se utilizan para describir un modelo de datos.

5. Casos de uso

- Resumen de los casos de uso más significativos para la arquitectura y sus requisitos no funcionales. Es decir, aquellos casos de uso que, mediante su implementación, cubren una parte significativa de la arquitectura o que influyen en muchos elementos de la arquitectura. Por ejemplo, el caso de uso *Procesar Venta*, cuando se implementa completamente, tiene estas cualidades.
- Una vista sobre el Modelo de Casos de Uso del UP, expresada textualmente y visualizada con los diagramas de casos de uso de UML.

6. Implementación

- En primer lugar, una definición del Modelo de Implementación: a diferencia de otros modelos del UP, que son texto y diagramas, este “modelo” *es* el código fuente real, ejecutables, etcétera. Tiene dos partes: 1) entregables, y 2) cosas que crean a los entregables (como código fuente y gráficos). El Modelo de Implementación está formado por todas estas cosas, incluyendo las páginas web, DLLs, ejecutables, código fuente, etcétera, y su organización —como el código fuente en los paquetes de Java y bytecodes organizados en ficheros JAR—.
- La vista de implementación es una descripción resumida de la organización relevante de los entregables y de las cosas que crean a los entregables (como el código fuente).
- Una vista sobre el Modelo de Implementación del UP, expresada textualmente y visualizada con los diagramas de paquetes y componentes de UML.

Por ejemplo, los diagramas de paquetes y de interacción de NuevaEra que se presentaron en el Capítulo 30 sobre la arquitectura en capas y lógica, muestran las grandes ideas de la estructura lógica de la arquitectura del software. En el SAD, el arquitecto creará una sección denominada *Vista Lógica*, insertará estos diagramas UML, y añadirá

algunos comentarios escritos acerca de para qué es cada paquete y capa, y el motivo que hay detrás del diseño lógico. De igual modo con las vistas de proceso y despliegue.

Una idea clave de las vistas de la arquitectura —que concretamente son texto y diagramas— es que *no* describen *todo* el sistema desde alguna perspectiva, sino sólo ideas destacadas desde esa perspectiva. Una vista es, si se quiere, la descripción “en un minuto de ascensor”: ¿Cuáles son las cosas más importantes que dirías en un minuto en un ascensor a un compañero sobre esta perspectiva?

Podrían crearse vistas de la arquitectura:

- después de que se construya el sistema, como resumen o ayuda de aprendizaje para futuros desarrolladores
- al final de ciertos hitos de la iteración (como al final de la iteración) para que sirva de ayuda para el aprendizaje para el equipo de desarrollo actual, y nuevos miembros
- de manera especulativa, durante las primeras iteraciones, como ayuda en el trabajo de diseño creativo, reconociendo que la vista original cambiará cuando prosiga el diseño y la implementación.

Estructura de ejemplo de un SAD

Documento de la Arquitectura del Software

Representación de la arquitectura

(Resumen del modo en el que se describirá la arquitectura en este documento, como utilizando memorándums técnicos y las vistas de la arquitectura. Esto es útil para alguien que no esté familiarizado con la idea de los memorándums técnicos o las vistas. Nótese que no son necesarias todas las vistas.)

Factores y decisiones de la arquitectura

(Referencia a la Especificación Complementaria para ver la Tabla de Factores. También, el conjunto de memorándums técnicos que resume las decisiones.)

Vista Lógica

(Los diagramas de paquetes de UML y los diagramas de clases de los elementos importantes. Comentarios sobre la estructura a gran escala y la funcionalidad de los componentes principales.)

Vista Proceso

(Diagramas de interacción y de clases de UML que ilustran los procesos e hilos de ejecución del sistema. Agruparlos de acuerdo a los hilos y procesos que interactúan. Comentarios sobre el modo en que funciona la comunicación entre los procesos (ej. mediante RMI de Java).)

Vista de Casos de Uso

(Resumen breve de los casos de uso más significativos desde el punto de vista de la arquitectura. Diagramas de interacción UML para algunas de las realizaciones de los casos de uso significativos para la arquitectura, o escenarios, con comentarios en los diagramas explicando cómo ilustran los elementos importantes de la arquitectura.)

Vista de Despliegue

(Los diagramas de despliegue de UML que muestra los nodos y la asignación de los procesos y componentes. Comentarios sobre la red.)

Fases

Inicio: Si no está claro si es técnicamente posible satisfacer los requisitos significativos de la arquitectura, el equipo debe implementar una **prueba de concepto de la arquitectura** (PDC) para determinar la viabilidad. En el UP, su creación y evaluación se denomina **Síntesis de la Arquitectura**. Esto es distinto de los anteriores experimentos pequeños y simples de programación de PDC para cuestiones técnicas aisladas. Una PDC de la arquitectura cubre ligeramente *muchos* de los requisitos significativos para la arquitectura para evaluar su viabilidad *combinada*.

Elaboración: Un objetivo importante de esta fase es la implementación de los elementos centrales de la arquitectura de riesgo, de esta manera la mayoría del análisis arquitectural se completa durante la elaboración. Normalmente se espera que la mayor parte del contenido de la tabla de factores, memorándums técnicos y el SAD se pueda completar al final de la elaboración.

Transición: Aunque idealmente los factores y decisiones significativas para la arquitectura se resolvieron mucho antes de la transición, el SAD necesitará que se repase y posiblemente que se revise al final de esta fase para asegurar que describe de manera precisa el sistema de despliegue final.

Ciclos de evolución siguientes: Antes de diseñar nuevas versiones, es común volver a visitar los factores de la arquitectura y las decisiones. Por ejemplo, la decisión de la versión 1.0 de crear un único servicio remoto para el cálculo de impuestos, en lugar de un duplicado en cada nodo de PDV, podría haberse motivado por el coste (para evitar múltiples licencias). Pero quizás en el futuro, se reduce el coste de los calculadores de impuestos, y de esta manera, por razones de tolerancia a fallos o de rendimiento, la arquitectura cambia para utilizar varios calculadores de impuestos.

32.8. Lecturas adicionales

Existe un cuerpo creciente de patrones relacionados con la arquitectura, y consejos generales sobre la arquitectura del software. Sugerencias:

- *Pattern-Oriented Software Architecture*, ambos volúmenes.
- *Software Architecture in Practice* [BCK98].
- *Pattern Languages of Program Design*, todos los volúmenes. Cada volumen contiene una sección de patrones relacionados con la arquitectura.
- Artículos disponibles a través de la web sobre patrones de arquitectura (como las arquitecturas J2EE), en Sun, IBM, y otros sitios web.
- Artículos disponibles a través del web sobre arquitectura en el Instituto de Ingeniería del Software (SEI, *Software Engineering Institute*) de la Universidad Carnegie Mellon, que desde hace mucho tiempo es un centro de investigación sobre arquitectura (www.sei.cmu.edu).

Capítulo 33

DISEÑO DE MÁS REALIZACIONES DE CASOS DE USO CON OBJETOS Y PATRONES

En dos ocasiones me han preguntado (miembros del Parlamento), “Por favor, Mr. Babbage, si introduce en la máquina cifras incorrectas, ¿obtendrá la respuesta correcta?” No soy capaz de comprender exactamente el tipo de confusión de ideas que podría provocar tal pregunta.

Charles Babbage

Objetivos

- Aplicar los patrones GRASP y GoF en el diseño.
-

Introducción

Este capítulo estudia algunos diseños parciales para la iteración actual, manejando requisitos tales como el mantenimiento de los servicios ante fallos mediante servicios locales, la gestión de dispositivos de PDV y la autorización de pagos.

33.1. Mantenimiento de los servicios ante los fallos mediante servicios locales; rendimiento con el almacenamiento local

Uno de los requisitos de NuevaEra es un cierto grado de recuperación ante el fallo (*failover*) del servicio remoto, como una base de datos de productos no disponible (temporalmente).

El acceso a la información de los productos es el primer caso que se utiliza para estudiar la estrategia de recuperación y mantenimiento del servicio ante un fallo. Poste-

riormente, se explora el acceso al servicio de contabilidad, que tiene una solución ligeramente distinta.

Revisemos parte del memorándum técnico:

Memorándum Técnico

Asunto: Fiabilidad—Recuperación de fallos en los servicios remotos

Resumen de la solución: Ubicación transparente utilizando un servicio de búsqueda, mantenimiento del servicio ante los fallos pasando de remoto a local, y replicación parcial en el servicio local.

Factores

- Recuperación robusta de los fallos en los servicios remotos (ej. calculador de impuestos, inventario).
- Recuperación robusta de los fallos en la base de datos de productos (ej. descripciones y precios) remota.

Solución

Conseguir variaciones protegidas con respecto a la ubicación de los servicios utilizando un Adaptador creado en una FactoriaDeServicios. Donde sea posible, ofrecer implementaciones locales de servicios remotos, normalmente con comportamiento simplificado o restringido. Por ejemplo, el calculador de impuestos local utilizará porcentajes de impuestos constantes. La base de datos local con información de los productos será una pequeña caché de los productos más comunes. Se almacenarán las actualizaciones del inventario y se remitirán cuando se restablezca la conexión.

Véase también el memorándum técnico de *Adaptabilidad—Servicios de terceras partes* para conocer los aspectos de adaptabilidad de esta solución, porque las implementaciones de los servicios remotos variarán en cada instalación.

Para satisfacer los escenarios de calidad de reconexión con los servicios remotos, utilizar para los servicios objetos Proxy inteligentes, que en cada llamada a un servicio comprueban si se puede reactivar el servicio remoto y lo redireccionan a éste cuando es posible.

Motivación

¡Las tiendas realmente no quieren dejar de vender! Por tanto, si el PDV NuevaEra ofrece este nivel de fiabilidad y recuperación, será un producto muy atractivo, ya que ninguno de nuestros competidores proporciona esta capacidad.

Antes de solucionar los aspectos de recuperación y de mantenimiento del servicio ante los fallos, observe que tanto por razones de rendimiento como para mejorar la recuperación cuando falla la base de datos remota, el arquitecto (yo) ha recomendado una caché local (que persiste de manera fiable en el disco duro local en un simple fichero) de objetos *EspecificacionDelProducto*. Por tanto, siempre se debería buscar en la caché local por si hubiera un “acuerdo de caché” antes de intentar un acceso remoto.

Esto se puede conseguir de manera elegante con nuestro diseño existente del adaptador y la factoría:

1. La *FactoriaDeServicios* siempre devolverá un adaptador a un servicio de información de producto local.
2. El “adaptador” de los productos locales realmente no es un adaptador a otro componente. Implementará él mismo las responsabilidades del servicio local.

3. El valor inicial del servicio local es una referencia a un segundo adaptador del verdadero servicio de productos remoto.
4. Si el servicio local encuentra el dato en su caché, lo devuelve; en otro caso, remite la petición al adaptador del servicio externo.

Obsérvese que existen dos niveles de caché del lado del cliente:

1. El objeto *CatalogoDeProductos* en memoria mantendrá una colección en memoria (como un *HashMap* de Java) de algunos (por ejemplo, 1.000) objetos *EspecificacionDelProducto* que se han obtenido a través del servicio de información de productos. Se puede ajustar el tamaño de esta colección en función de la disponibilidad de memoria local.
2. El servicio de productos local mantendrá una caché persistente más grande (en el disco duro) que mantiene cierta cantidad de información de productos (como 1 o 100 MB de espacio en ficheros). De nuevo, se puede ajustar dependiendo de la configuración local. Esta caché persistente es importante para la tolerancia a fallos, de manera que incluso si la aplicación del PDV cae y se pierde la caché de memoria del objeto *CatalogoDeProductos*, permanece la caché persistente.

Este diseño no afecta al código existente —el nuevo objeto de servicio local se inserta sin afectar al diseño del objeto *CatalogoDeProductos* (que colabora con el servicio de productos)—.

Hasta el momento, no se han introducido nuevos patrones; se han utilizado un Adaptador y una Factoría.

La Figura 33.1 ilustra los tipos del diseño y la Figura 33.2 ilustra la inicialización.

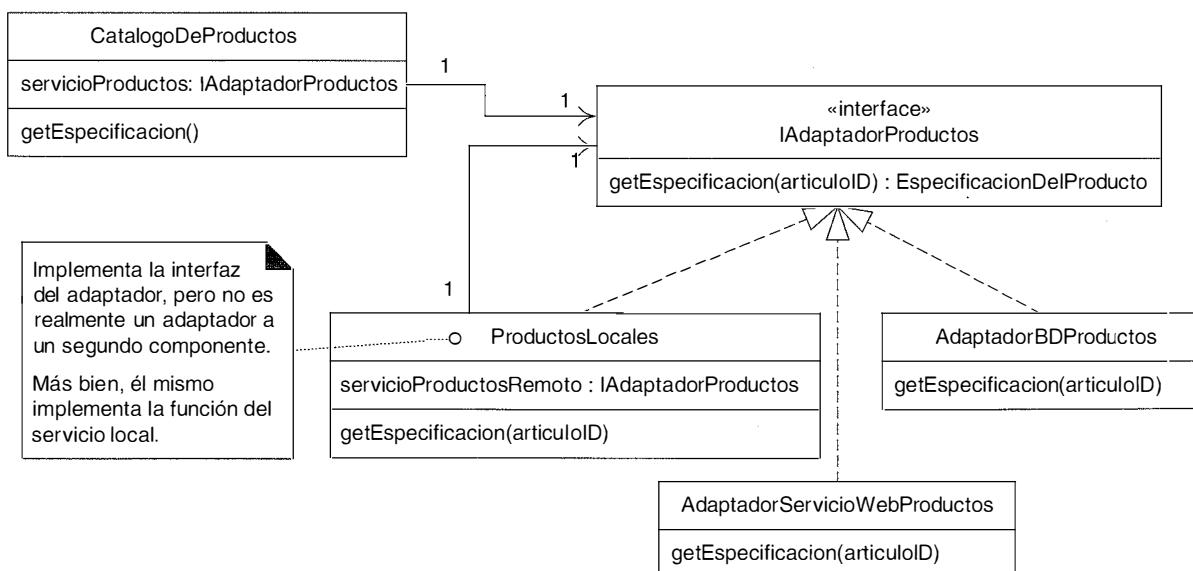


Figura 33.1. Adaptadores para la información de productos.

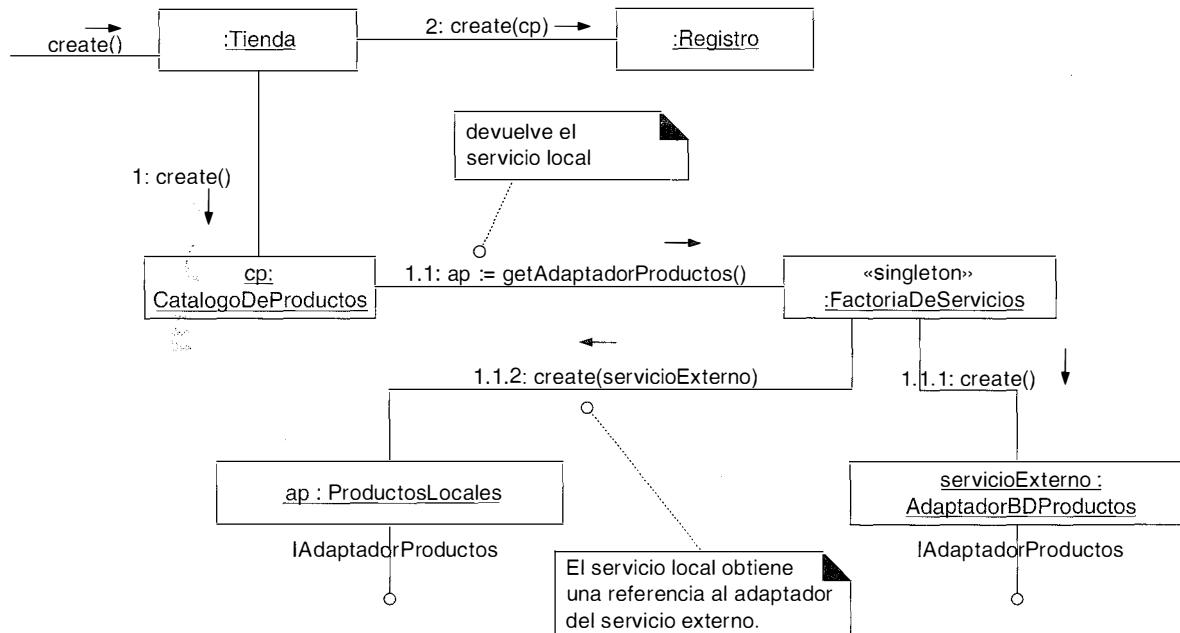


Figura 33.2. Inicialización del servicio de información de productos.

La Figura 33.3 muestra la colaboración inicial desde el catálogo al servicio de productos.

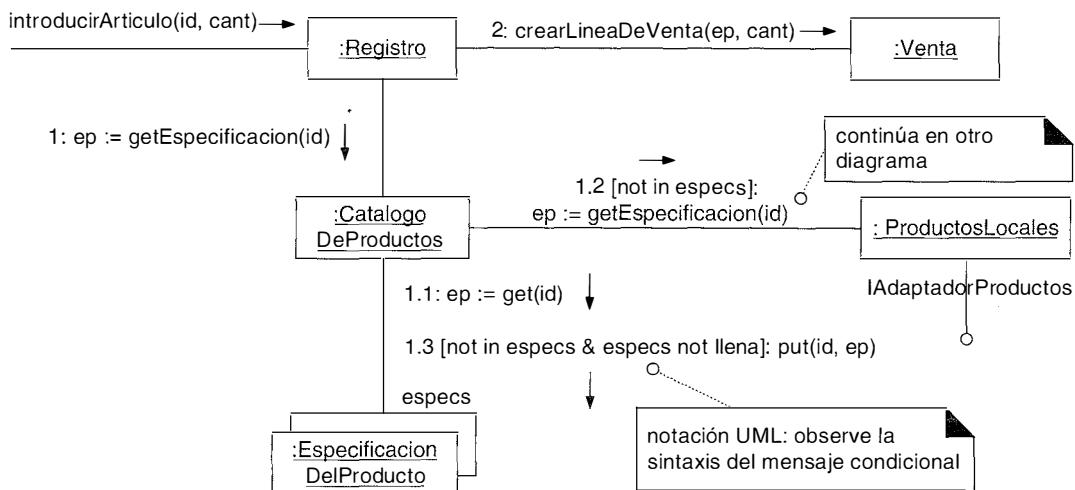


Figura 33.3. Comienzo de la colaboración con el servicio de productos.

Si el servicio de productos local no tiene el producto en su caché, colabora con el adaptador del servicio externo, como se muestra en la Figura 33.4. Nótese que el servicio de productos local almacena los objetos *EspecificacionDelProducto* como verdaderos objetos serializados.

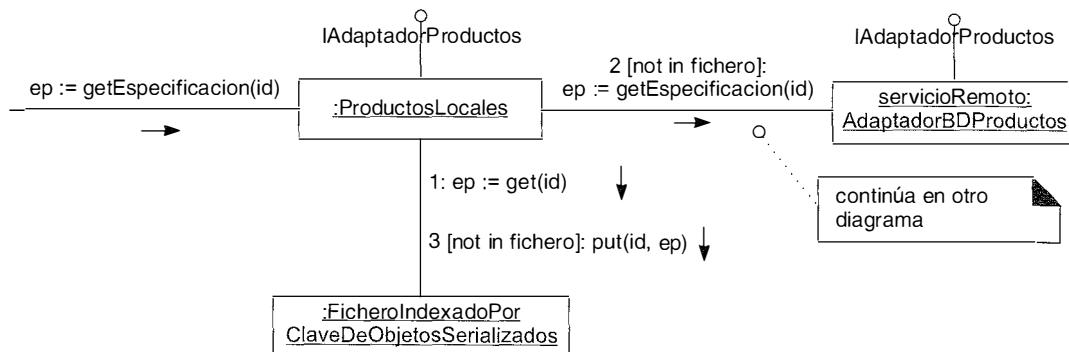


Figura 33.4. Continuación de la colaboración para obtener la información del producto.

Si se cambió el servicio externo de una base de datos a un nuevo servicio Web, sólo es necesario cambiar la configuración de la factoría del servicio remoto. Véase la Figura 33.5.

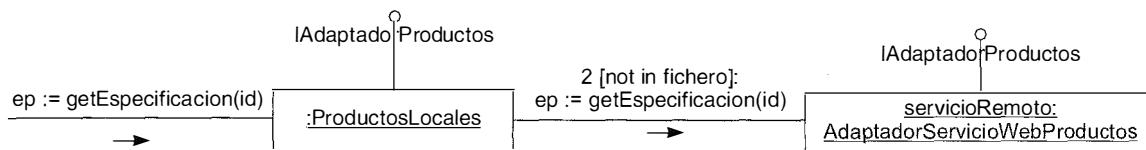


Figura 33.5. Los nuevos servicios externos no afectan al diseño.

Continuando con el caso de la colaboración con el *AdaptadorBDProductos*, interactuará con un subsistema de persistencia que tiene que establecer la correspondencia objeto-relacional (*mapping O-R*) (ver Figura 33.6).

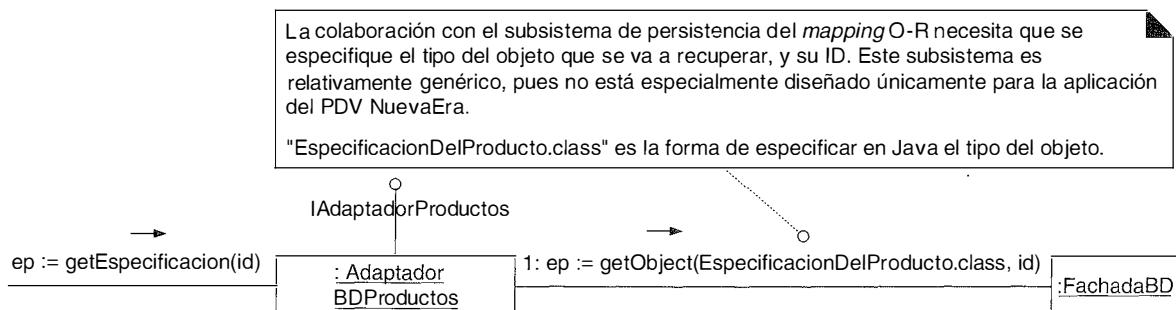


Figura 33.6. Colaboración con el subsistema de persistencia.

Estrategias de almacenamiento en caché

Considere las alternativas para cargar en memoria la caché del *CatalogoDeProductos* y la caché basada en ficheros de *ProductosLocales*: un enfoque es la inicialización perezosa, en el que las cachés se cargan lentamente cuando se recupera la información del producto externo; otro enfoque es la inicialización impaciente, en el que las cachés se cargan du-

rante el caso de uso *PonerEnMarcha*. Si el diseñador no está seguro del enfoque a utilizar y quiere experimentar con las alternativas, una familia de diferentes objetos *Estrategia-Cache* basada en el patrón Estrategia puede solucionar el problema de manera elegante.

Caché antigua

Puesto que los precios de los productos cambian rápidamente, y quizás al antojo del encargado de la tienda, almacenar en la caché el precio de los productos crea un problema —la caché contiene datos antiguos—; esto es siempre una preocupación cuando se replican los datos. Una solución es añadir una operación al servicio remoto que responda con los cambios actuales a lo largo de un día; el objeto *ProductosLocales* la consulta cada n minutos y actualiza su caché.

Hilos en UML

Si el objeto *ProductosLocales* va a solucionar el problema de la cache con datos antiguos cada n minutos, un enfoque de diseño es hacerlo un **objeto activo** que posea un hilo de control. El hilo dormirá durante n minutos, se despertará, el objeto obtendrá los datos, y

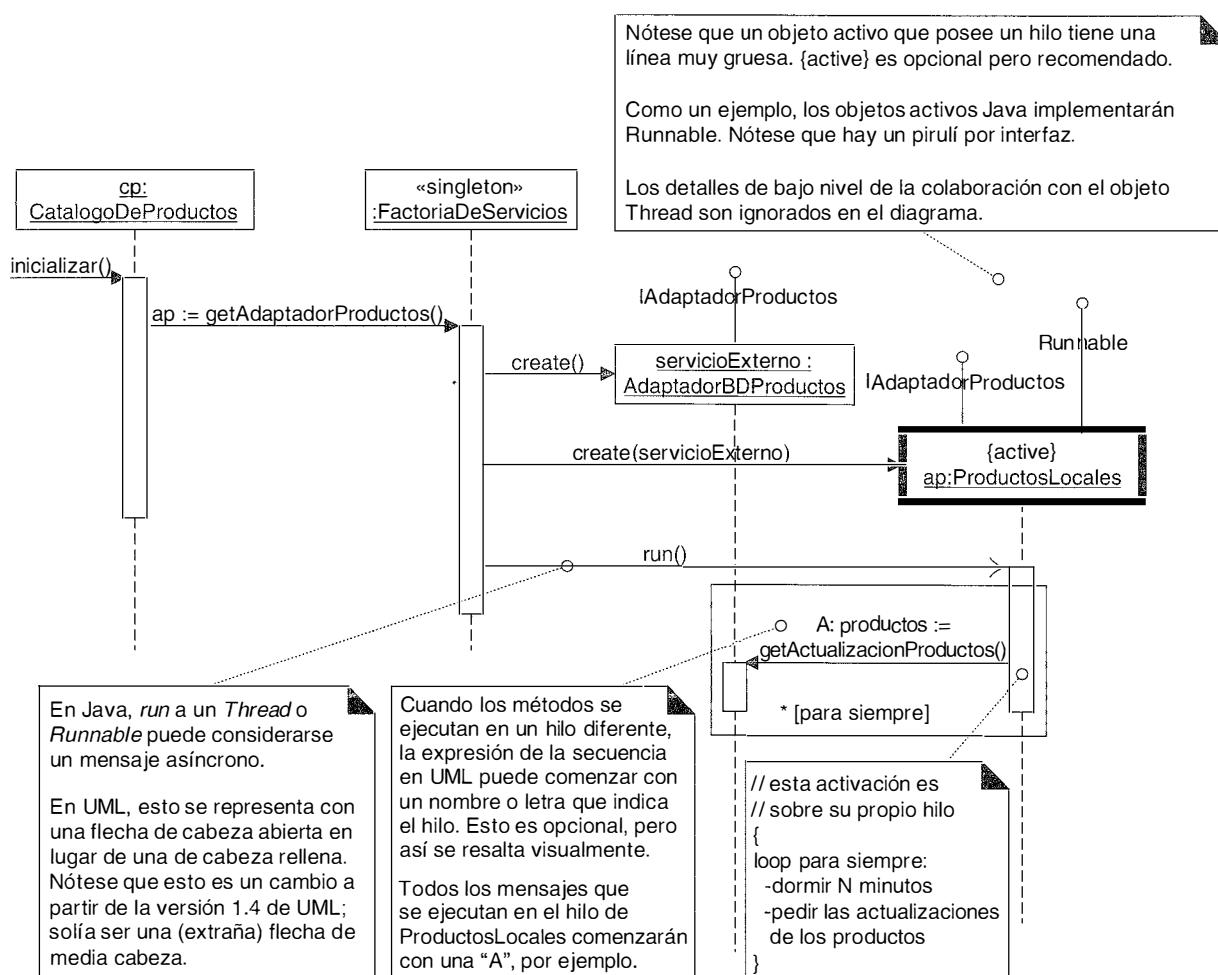


Figura 33.7. Hilos y mensajes asíncronos en UML.

el hilo volverá a dormirse. UML proporciona la notación para representar hilos y las llamadas asíncronas, como se muestra en la Figura 33.7.

En un diagrama de interacción, una instancia de un objeto activo podría etiquetarse con la propiedad */active*. En un diagrama de clases, una clase de objetos activos (una **clase activa**) que posee su propio hilo se puede estereotipar con «*thread*». Véase la Figura 33.8.

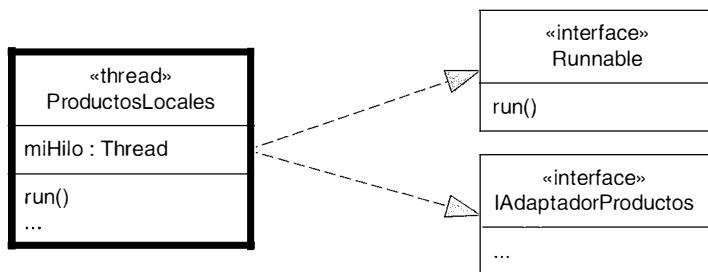


Figura 33.8. Notación para las clases activas.

33.2. Manejo de fallos

El diseño anterior proporciona una solución para el almacenamiento en la caché del lado del cliente de los objetos *EspecificacionDelProducto* en un fichero persistente, para mejorar el rendimiento, y también para proporcionar al menos una solución parcial a la que recurrir si no se puede acceder al servicio de productos externo. Quizás se almacenen en la caché 10.000 productos en el fichero local, que podría satisfacer la mayoría de las peticiones de información de los productos incluso cuando falla el servicio externo.

¿Qué hacemos en el caso de que no se encuentre en la caché y falla el acceso al servicio de productos externo? Suponga que las personas involucradas nos piden que creamos una solución que indique al cajero que introduzca manualmente el precio y la descripción, o cancele la entrada de la línea de venta.

Éste es un ejemplo de una condición de error o fallo, y se utilizará como contexto para describir algunos patrones generales que se ocupan del manejo de los fallos y excepciones. El manejo de excepciones y errores es un tema amplio, y esta introducción únicamente se centrará en algunos patrones específicos para el contexto del caso de estudio. En primer lugar, algo de terminología:

- **Defecto:** el origen último o causa del mal comportamiento.
 - El programador escribe mal el nombre de la base de datos.
- **Error:** una manifestación del defecto en el sistema en ejecución. Los errores se detectan (o no).
 - Cuando estamos invocando al servicio de nombres para obtener una referencia a la base de datos (con el nombre mal escrito), se señala un error.
- **Fallo:** la denegación de un servicio causada por un error.
 - El subsistema de Productos (y el PDV NuevaEra) falla al proporcionar un servicio de información de productos.

Lanzamiento de excepciones

Un enfoque directo para señalar el fallo que se está produciendo es lanzar una excepción.

Las excepciones son apropiadas especialmente cuando estamos tratando con fallos de los recursos (disco, memoria, acceso a la red o a una base de datos y otros servicios externos).

Se lanzará una excepción desde el interior del subsistema de persistencia (realmente, es muy probable que a partir de algo como una implementación JDBC de Java), donde se detecte en primer lugar un fallo al utilizar la base de datos de productos externa. La excepción desenrollará la pila de llamadas hacia atrás hasta un punto apropiado donde se pueda manejar¹.

Suponga que la excepción original (utilizando Java como ejemplo) es *java.sql.SQLException*. ¿Debería propagarse una *SQLException* todo el camino hacia arriba hasta llegar a la capa de presentación? No. No está en el nivel correcto de abstracción. Esto nos lleva a un patrón común para el manejo de excepciones:

Patrón: Convertir Excepciones [Brown01]

En un subsistema, evite la emisión de excepciones de bajo nivel que proceden de los subsistemas inferiores o servicios. Más bien, convierta una excepción de bajo nivel en una que sea significativa para el nivel del subsistema. La excepción de más alto nivel normalmente encapsula la excepción de un nivel inferior, y añade información, para hacer que la excepción sea más significativa de acuerdo con el contexto de los niveles más altos.

Esto es una guía, no una regla absoluta.

Aquí “Excepción” se utiliza en el sentido vernáculo de algo que se puede lanzar; en Java, el equivalente es un *Throwable*.

También conocido como Abstracción de Excepción [Renzel97].

Por ejemplo, el subsistema de persistencia captura una *SQLException* concreta, y (asumiendo que no puede manejarla²) lanza una nueva *BDNoDisponibleException*, que contiene la *SQLException*. Nótese que el *AdaptadorBDProductos* es como una fachada sobre un subsistema lógico para la información de los productos. Por tanto, el *AdaptadorBDProductos* de más alto nivel (como representante de un subsistema lógico) captura la *BDNoDisponibleException* de un nivel inferior y (asumiendo que no puede manejarla) lanza una nueva *InfoProductoNoDisponibleException*, que encapsula la *BDNoDisponibleException*.

¹ No se trata el manejo de excepciones comprobadas frente a las no comprobadas, ya que no es soportado en todos los lenguajes OO conocidos—C++, C# y Smalltalk, por ejemplo.

² Resolver una excepción cerca del nivel donde se produce es un objetivo loable pero difícil, puesto que los requisitos acerca de la manera de manejar un error con frecuencia son específicos de la aplicación.

Considere los nombres de estas excepciones: ¿por qué decir *BDNoDisponibleException* en lugar de *SubsistemaPersistenciaException*? Existe un patrón para esto:

Patrón: Nombre El Problema No El Lanzador [Grosso00]

¿Cómo llamar a una excepción? Asigne un nombre que describa por qué se va a lanzar la excepción, no quién la lanza. La ventaja es que facilita al programador la comprensión del problema, y resalta la similitud fundamental de muchas clases de excepciones (de una manera que no ocurre nombrando al que la lanza).

Excepciones en UML

Éste es un momento adecuado para introducir la notación de UML para el lanzamiento³ y captura de excepciones.

Notación UML: UML tiene una sintaxis "por defecto" para las operaciones. Pero no incluye una solución oficial para mostrar las excepciones que lanza una operación. Existen al menos tres soluciones:

1. UML permite que se utilice para las operaciones la sintaxis de cualquier otro lenguaje, como Java. Además, algunas herramientas CASE UML permiten mostrar por pantalla las operaciones explícitamente en la sintaxis de Java. De este modo,

```
Object get(Clave, Class) throws BDNoDisponibleException, FatalException
```

2. La sintaxis por defecto permite que el último elemento sea un "string de propiedades" (*property string*). Esto es una lista arbitraria de pares propiedad+valor, como {autor=Craig, niños=(Hannah, Haley),...}. Así,

```
put(Object, id){ throws=( BDNoDisponibleException, FatalException)}
```

3. Algunas herramientas CASE UML le permiten a uno especificar (en un cuadro de diálogo especial) las excepciones que lanza una operación.

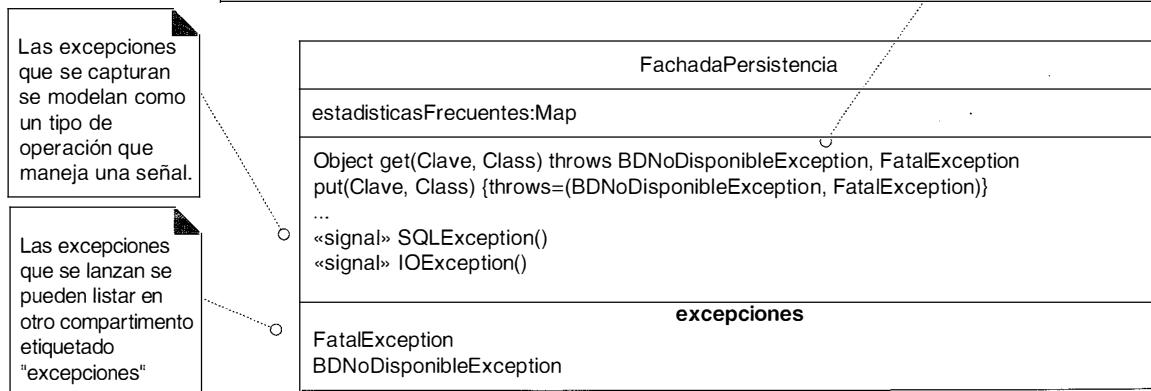


Figura 33.9. Excepciones capturadas y lanzadas por una clase.

³ Oficialmente en UML, uno *envía* una excepción, pero *lanza* es un término adecuado y más familiar.

Dos preguntas típicas sobre la notación UML son:

1. En un diagrama de clases, ¿cómo representamos las excepciones que captura y lanza una clase?
2. En un diagrama de interacción, ¿cómo representamos el lanzamiento de una excepción?

La Figura 33.9 presenta la notación para un diagrama de clases.

En UML, una *Exception* es una especialización de una *Signal*, que es la especificación de una comunicación asíncrona entre objetos. Esto significa que en un diagrama de interacción, las excepciones se representan como **mensajes asíncronos**⁴.

La Figura 33.10 muestra la notación, utilizando como ejemplo la descripción anterior de *SQLException* que se transforma en una *BDNoDisponibleException*.

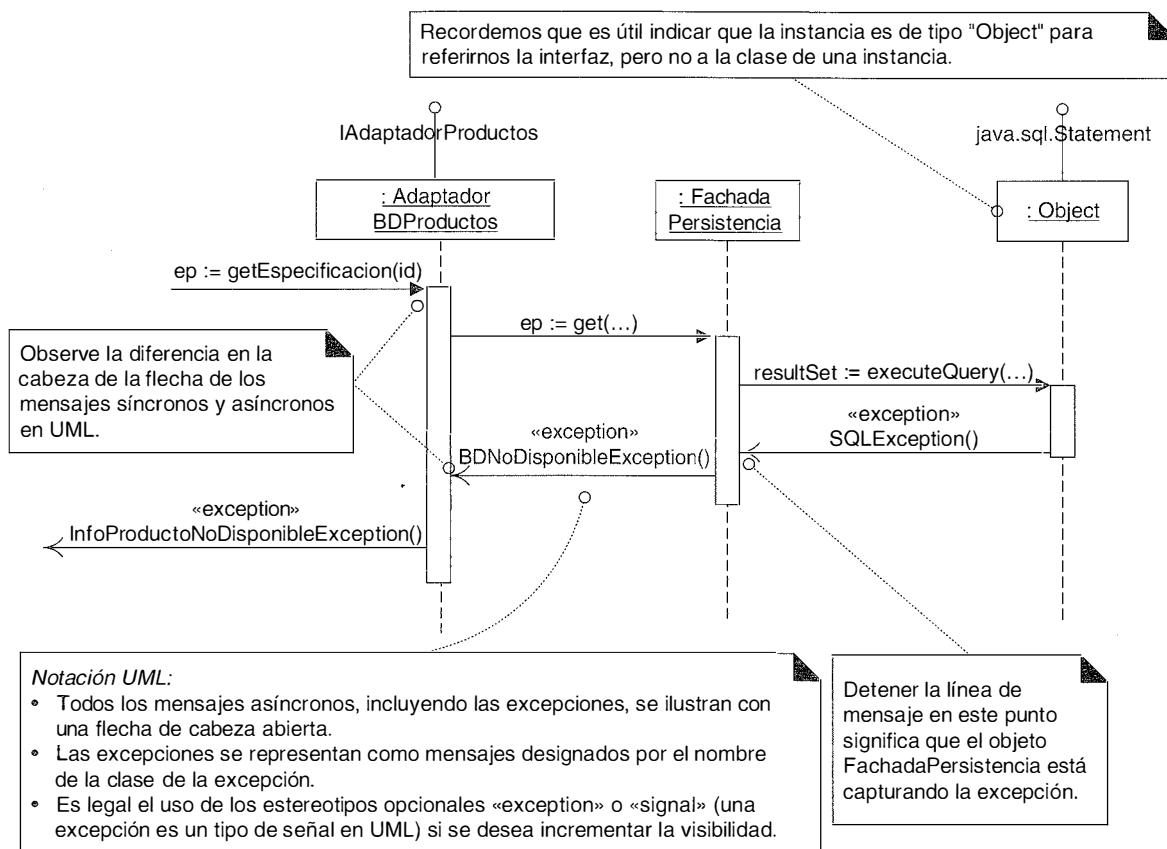


Figura 33.10. Excepciones en un diagrama de interacción.

En resumen, existe notación UML para representar las excepciones. Sin embargo, pocas veces se utiliza.

⁴ Nótese que a partir de la versión 1.4 de UML, cambió la notación para los mensajes asíncronos de una flecha de media cabeza a una de cabeza abierta.

No es que recomendemos evitar las consideraciones iniciales sobre el manejo de excepciones. Más bien lo contrario: al nivel de arquitectura es necesario establecer pronto los patrones básicos, políticas y colaboraciones para el manejo de las excepciones, puesto que es difícil insertar el manejo de las excepciones como una ocurrencia tardía. Sin embargo, muchos desarrolladores consideran que el diseño a bajo nivel del manejo de excepciones concretas se decide de manera más apropiada durante la programación o mediante descripciones de diseño menos detalladas, en lugar de mediante diagramas UML detallados.

Manejo de errores

Se ha considerado una parte del diseño: el lanzamiento de excepciones, en cuanto a la conversión, asignación de nombres y la manera de representarlas. La otra parte es el manejo de una excepción.

Dos patrones que se pueden aplicar en éste y en la mayoría de los casos son:

Patrón: Registro Centralizado de los Errores [Renzel97]

Utilice un objeto central de registro de errores con acceso mediante el patrón Singleton y notifíquelo todas las excepciones. Si se trata de un sistema distribuido, cada singleton local colaborará con un registro de error central. Beneficios:

- Informes consistentes.
- Definición flexible de la información de salida y el formato.

También conocido como Registrador de Diagnóstico (*Diagnostic Logger*) [Harrison98].

Es un patrón sencillo. El segundo es:

Patrón: Diálogo de Error [Renzel97]

Para notificar los errores a los usuarios utilice un objeto que no pertenezca a la interfaz de usuario, independiente de la aplicación y al que se accede siguiendo el patrón Singleton. Actúa como envoltorio (*wrapper*) de uno o más objetos de “diálogo” de la UI (tales como diálogo modal de la UI, consola de texto, emitir un sonido, o generador de voz) y delega la notificación del error a los objetos del UI. De esta manera, la salida podría ir tanto a un diálogo de la GUI como a un generador de voz. También informará de la excepción al registro centralizado de errores. Una Factoría que lee los parámetros del sistema creará los objetos apropiados de la UI. Beneficios:

- Variaciones Protegidas con respecto a los cambios en el mecanismo de salida.
- Estilo consistente de los informes de error; por ejemplo, todas las ventanas de la GUI pueden invocar a este singleton para mostrar el diálogo de error.
- Control centralizado de la estrategia común para la notificación de los errores.
- Ganancia de rendimiento pequeña; si se utiliza un recurso “caro” como un diálogo de la GUI, es fácil ocultarlo y almacenarlo en caché para volver a utilizarlo después, en lugar de volver a crear un diálogo por cada error.

¿Debería un objeto de la UI (por ejemplo, *FrameProcesarVenta*) manejar un error capturando la excepción e informando al usuario? Para aplicaciones con pocas ventanas,

y caminos de navegación entre ventanas simples y estables, este diseño directo es bueno. Esto se cumple actualmente para la aplicación NuevaEra.

Tenga presente sin embargo, que esto sitúa algo de la “lógica de la aplicación” relacionada con el manejo de los errores en la capa de presentación (GUI). El manejo de errores está relacionado con la notificación a los usuarios, luego es lógica, pero es una tendencia que hay que tener en cuenta. No es inherentemente un problema de UI simples con pocas oportunidades de reemplazar la UI, sino que es un punto de fragilidad. Por ejemplo, suponga que un equipo quiere sustituir la UI Swing de Java por el framework de GUI Java MicroView de IBM para ordenadores portátiles. Ahora existe algo de la lógica de la aplicación en la versión Swing que se tiene que identificar y replicar en la versión MicroView. Hasta cierto punto, esto es inevitable en las sustituciones de UI; pero se agravará cuanta más lógica de la aplicación se migre hacia arriba. En general, cuantas más responsabilidades de la lógica de la aplicación que no tiene que ver con el UI se migren a la capa de presentación, mayor es la probabilidad de que se produzcan quebraderos de cabeza relacionados con el diseño o el mantenimiento.

Para sistemas con muchas ventanas y caminos de navegación complejos (quizás incluso cambiantes), existen otras soluciones. Por ejemplo, se puede insertar una capa de aplicación de uno o más controladores entre las capas de presentación y del dominio.

Además, se puede insertar un objeto “mediador en el manejo de la vista” [GHJV95, BMRSS96] que es responsable de mantener una referencia a todas las ventanas abiertas, y conocer las transiciones entre ventanas, dado algún evento E1 (como un error).

Este mediador es de manera abstracta una máquina de estados que encapsula los estados (la ventana que se muestra) y las transiciones entre los estados, en base a los eventos. Podría leer el modelo de transición de estados (ventanas) de un fichero externo, de manera que los caminos de navegación pueden ser dirigidos por los datos (no es necesario ningún cambio en el código fuente). También puede cerrar todas las ventanas de la aplicación, o disponerlas de algún modo o minimizarlas, puesto que tiene una referencia a todas las ventanas.

En este diseño, podría diseñarse un controlador de la capa de aplicación con una referencia a este mediador para el manejo de la vista (por ello, el controlador de la aplicación se acopla “hacia arriba” con la capa de presentación). El controlador de la aplicación podría capturar la excepción y colaborar con el mediador para el manejo de la vista para producir la notificación (basado en el patrón Diálogo de Error). De esta manera, el controlador de la aplicación está involucrado con flujos de trabajo de la aplicación, y se deja fuera de las ventanas algo del manejo lógico de errores.

El diseño detallado del control de la UI y de la navegación quedan fuera del alcance de esta introducción, y el simple diseño de la ventana que captura la excepción será suficiente. Un diseño que utiliza un Diálogo de Error se muestra en la Figura 33.11.

33.3. Mantenimiento de los servicios ante los fallos mediante un Proxy (GoF)

El mantenimiento de los servicios ante los fallos (*failover*) mediante un servicio local para la información de los productos se consiguió insertando el servicio local delante del servicio externo; el servicio local siempre se prueba primero. Sin embargo, este diseño

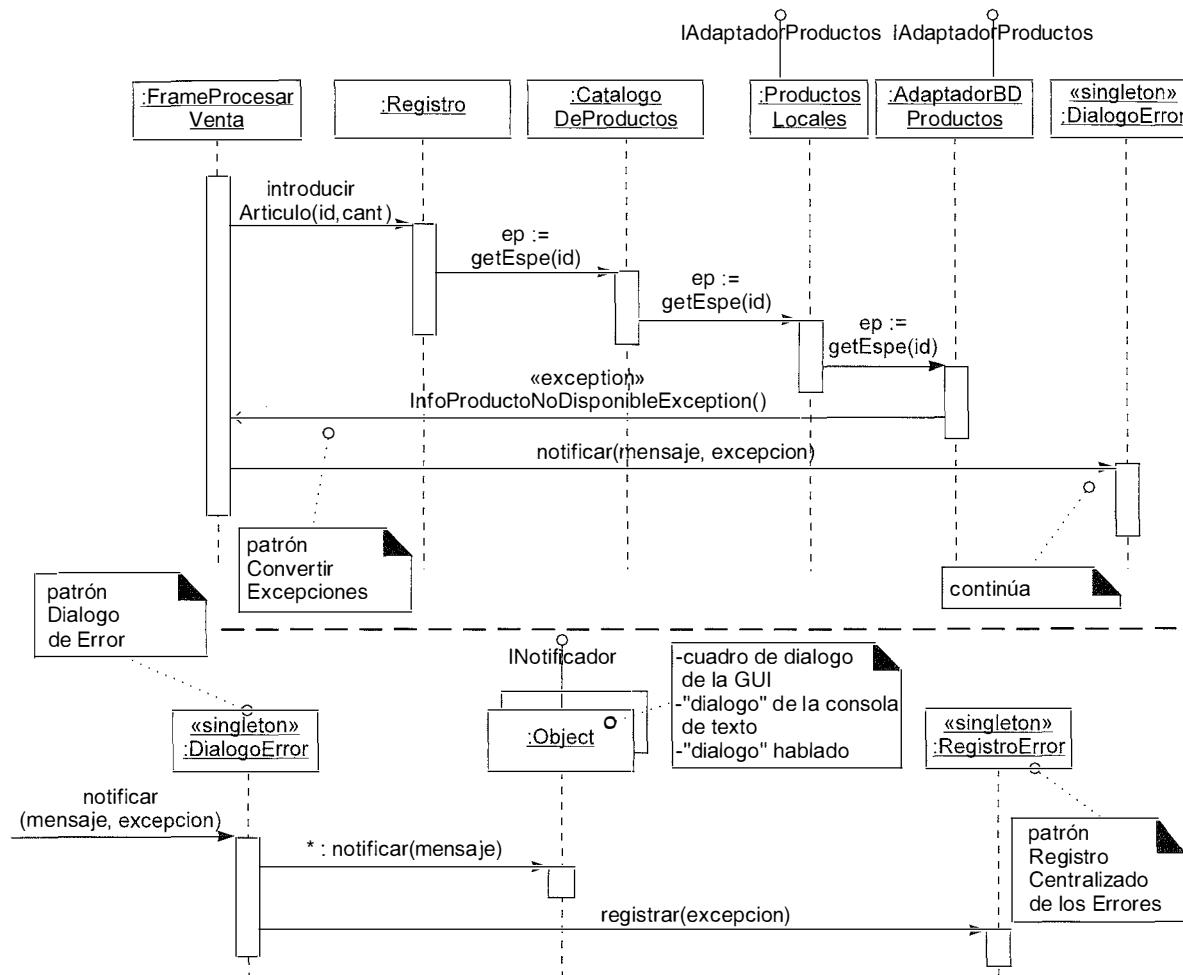


Figura 33.11. Manejo de la excepción.

no es apropiado para todos los servicios; algunas veces se debería probar primero el servicio externo, y la versión local en segundo lugar. Por ejemplo, considere el registro de las ventas en el servicio de contabilidad. El negocio quiere que se registren lo antes posible, para controlar en tiempo real la actividad de la tienda y el registro.

En este caso, otro patrón GoF puede solucionar el problema: Proxy⁵. Proxy es un patrón sencillo, y se utiliza ampliamente su variante **Proxy Remoto**. Por ejemplo, en RMI de Java y en CORBA, un objeto local del lado del cliente (denominado “*stub*”) se invoca para que acceda a los servicios de un objeto remoto. El stub del lado del cliente es un proxy local, o un representante o sustituto del objeto remoto.

Este ejemplo de uso del Proxy en NuevaEra no es la variante de Proxy Remoto, sino la variante **Proxy de Redirección (Redirection Proxy)** (también conocido como **Proxy de Mantenimiento de Servicio, Failover Proxy**).

⁵ *N. del T.* No se ha traducido por el mismo motivo que en el caso del patrón Singleton.

Independientemente de la variante, la estructura de un Proxy es siempre la misma; las variantes tienen que ver con lo que hace el proxy una vez que se le ha llamado.

Un proxy es simplemente un objeto que implementa la misma interfaz que el objeto al que se quiere acceder en realidad, manteniendo una referencia al sujeto real, y suele controlar el acceso a él. La Figura 33.12 muestra la estructura general.

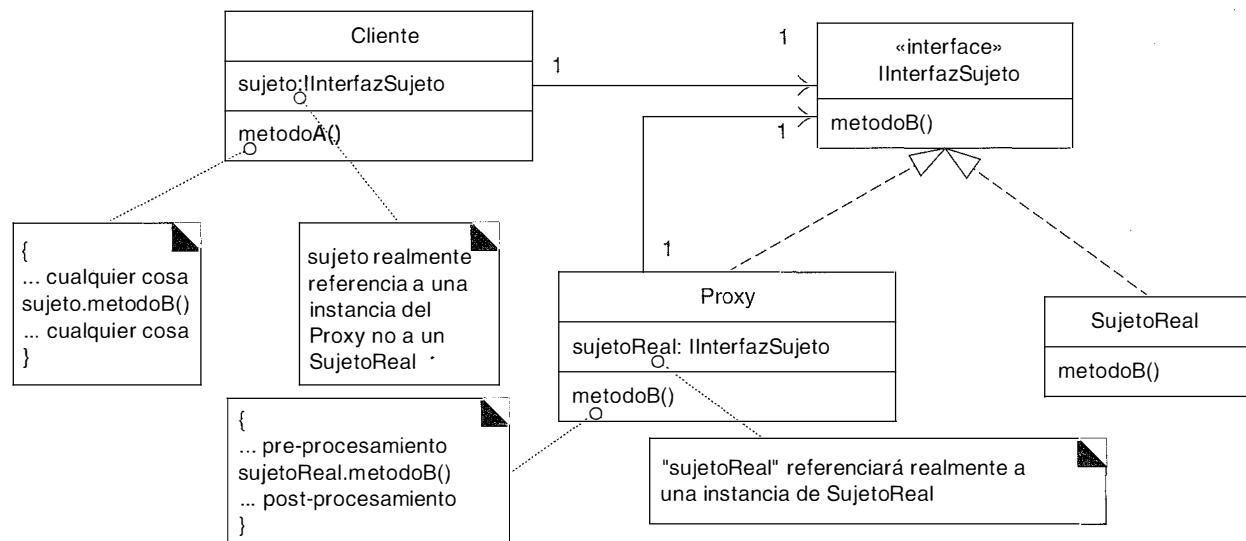
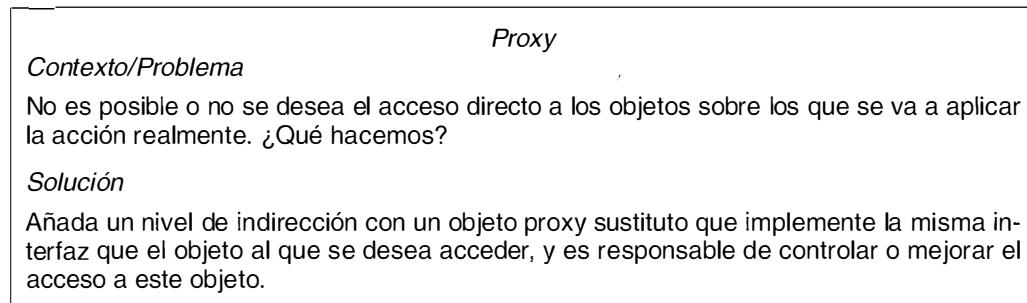


Figura 33.12. Estructura general del patrón Proxy.

Aplicado al caso de estudio de NuevaEra para el acceso al servicio de contabilidad externo, se utiliza un proxy de redirección como sigue:

1. Enviar el mensaje *anotarVenta* al proxy de redirección, tratándolo como si se pensase que es el servicio de contabilidad externo real.
2. Si falla el proxy de redirección al establecer el contacto con el servicio externo (mediante su adaptador), entonces redireccionar el mensaje *anotarVenta* a un servicio local, que almacena localmente las ventas para remitirlas al servicio de contabilidad, cuando esté disponible.

La Figura 33.13 ilustra un diagrama de clases de los elementos interesantes.

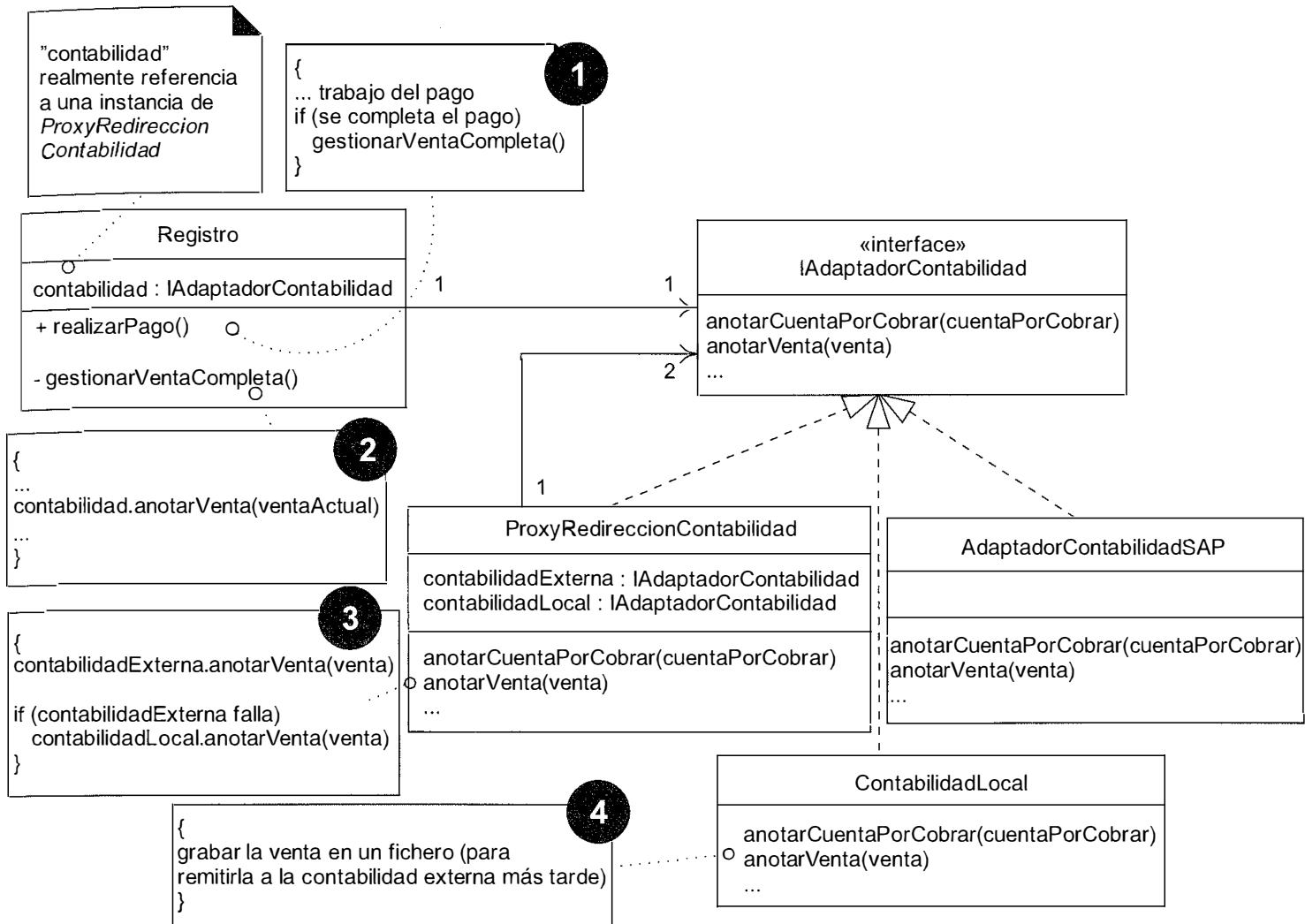


Figura 33.13. Uso en NuevaEra de un proxy de redirección.

Notación UML:

- Para evitar crear un diagrama de interacción que muestre el comportamiento dinámico, obsérvese cómo este diagrama estático utiliza la numeración para mostrar la secuencia de interacción. Normalmente es preferible un diagrama de interacción, pero se presenta este estilo para ilustrar un estilo alternativo.
- Obsérvese los marcadores de visibilidad pública y privada (+, -) junto a los métodos del `Registro`. Si no se indican, significa que no se especifica en lugar de asumir un valor por defecto público o privado. Sin embargo, de acuerdo con la práctica habitual, la mayoría de las personas que leen diagramas UML (y las herramientas CASE que generan código) interpretan la visibilidad no especificada como privada para los atributos y pública para los métodos. Sin embargo, en este diagrama, en particular quiero transmitir el hecho de que `realizarPago` es público, y en cambio, `gestionarVentaCompleta` es privado. El ruido visual y la sobrecarga de información son siempre una preocupación en la comunicación, luego, es conveniente hacer uso de las interpretaciones convencionales para mantener los diagramas simples.

Resumiendo, un proxy es un objeto externo que envuelve un objeto interno, y ambos implementan la misma interfaz. Un objeto cliente (como un `Registro`) no sabe que refe-

rencia a un proxy —se diseña de manera que piense que está colaborando con el sujeto real (por ejemplo, el *AdaptadorContabilidadSAP*)—. El Proxy intercepta las llamadas para mejorar el acceso al sujeto real, en este caso redireccionando la operación a un servicio local (*ContabilidadLocal*) si no está accesible el servicio externo.

33.4. Diseño para los requisitos no funcionales o de calidad

Antes de pasar a la siguiente sección, observe que el trabajo de diseño realizado hasta este punto del capítulo no está relacionado con la lógica del negocio, sino con los requisitos no funcionales o de calidad relacionados con la fiabilidad y recuperación.

Curiosamente —y es un punto clave en la arquitectura del software— es habitual que los diseños den forma antes a los temas, patrones y estructuras de la arquitectura del software de gran escala para resolver los requisitos no funcionales o de calidad, en lugar de a la lógica del negocio básica.

33.5. Acceso a los dispositivos físicos externos con adaptadores; comprar vs. construir

Otro requisito de esta iteración es interactuar con dispositivos físicos que componen un terminal de PDV, como abrir el cajón de caja, entregar el cambio del dispensador de monedas y capturar la firma de un dispositivo para las firmas digitales.

El PDV NuevaEra debe trabajar con una variedad de equipamiento de PDV, que incluye el que vende IBM, Epson, NCR, Fujitsu, etcétera.

Afortunadamente, el arquitecto de software ha investigado algo, y ha descubierto que ahora existe un estándar industrial, UnifiedPOS⁶ (www.nrf-arts.org), que define las interfaces orientadas a objetos estándar (en el sentido de UML) para todos los dispositivos comunes del PDV. Además, existe el JavaPOS (www.javapos.com) —la correspondencia en Java del UnifiedPOS—.

Por tanto, en el Documento de la Arquitectura del Software, el arquitecto añade un memorándum técnico para comunicar esta alternativa significativa para la arquitectura:

Memorándum Técnico
Asunto: Fiabilidad—Control de los dispositivos hardware del PDV

Resumen de la solución: Utilizar el software de Java de los fabricantes de dispositivos que se ajusta a las interfaces estándares de JavaPOS.

Factores

- Controlar correctamente los dispositivos
- Coste de comprar vs. construir y mantener

⁶ *N. del T.* POS es acrónimo de *Point-Of-Sale* (Punto-De-Venta), por tanto en esta sección aparece tanto POS como PDV.

Solución

UnifiedPOS (www.nrf-arts.org) define un modelo de interfaces UML que es un estándar industrial para los dispositivos de PDV. JavaPOS (www.javapos.com) es un estándar industrial que establece la correspondencia del UnifiedPOS a Java. Los fabricantes de dispositivos de PDV (ej. IBM, NCR) venden las implementaciones Java de estas interfaces que controlan sus dispositivos.

Comprárlas en lugar de construirlos.

Utilizar una Factoría que lea una propiedad del sistema para cargar un conjunto de clases de IBM o NCR (etc.) y devuelva instancias basadas en sus interfaces.

Motivación

Basado en una encuesta informal, creemos que funcionan bien, y los fabricantes tienen un proceso de actualización regular para mejorarllos. Es difícil conseguir la experiencia y otros recursos para escribirlos nosotros mismos.

Alternativas consideradas

Escribirlos nosotros mismos —difícil y arriesgado—.

La Figura 33.14 muestra algunas de las interfaces, que se han añadido como otro paquete de la capa del dominio en nuestro Modelo de Diseño.

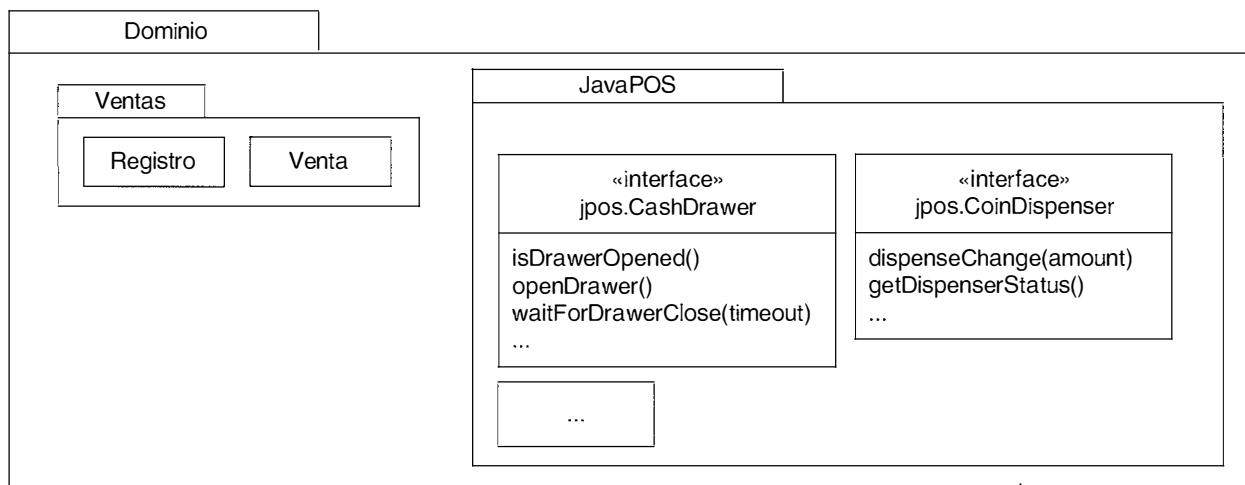


Figura 33.14. Interfaces JavaPOS estándares.

Asuma que los fabricantes importantes de los equipos de PDV ahora suministran las implementaciones JavaPOS. Por ejemplo, si compramos un terminal de PDV IBM con un cajón de caja, dispensador de monedas, etcétera, IBM también nos puede proporcionar las clases Java que implementan las interfaces JavaPOS, y que controlan los dispositivos físicos.

En consecuencia, esta parte de la arquitectura se resuelve comprando componentes software, en lugar de construyéndolos. Fomentar el uso de componentes existentes es una de las buenas prácticas del UP.

¿Cómo funcionan? A bajo nivel, un dispositivo físico tiene un controlador de dispositivo para las operaciones del sistema operativo subyacente. Una clase Java (por ejemplo, una que implemente *jpos.CashDrawer*) utiliza JNI (*Java Native Interface*) para hacer las llamadas a estos controladores de dispositivos.

Esas clases de Java adaptan los controladores de dispositivos de bajo nivel a las interfaces de JavaPOS, y así se pueden caracterizar como objetos Adaptador en el sentido del patrón GoF. También se les puede llamar objetos Proxy —objetos proxy locales que controlan o mejoran el acceso a los dispositivos físicos—.

No es raro que seamos capaces de clasificar un diseño en función de múltiples patrones.

33.6. Factoría Abstracta (GoF) para familias de objetos relacionados

Se comprarán a los fabricantes las implementaciones JavaPOS. Por ejemplo⁷:

```
//controladores IBM
com.ibm.pos.jpos.CashDrawer (implements jpos.CashDrawer)
com.ibm.pos.jpos.CoinDispenser (implements jpos.CoinDispenser)
...
//Controladores NCR
com.ncr.posdrivers.CashDrawer (implements jpos.CashDrawer)
com.ncr.posdrivers.CoinDispenser (implements jpos.CoinDispenser)
...
```

Ahora, ¿cómo diseñamos la aplicación de PDV NuevaEra para utilizar los controladores Java de IBM, si es que se utiliza el hardware de IBM, los controladores NCR si fueran los apropiados, etcétera?

Nótese que hay familias de clases (*CashDrawer+CoinDispenser+...*) que es necesario que se creen, y cada familia implementa las mismas interfaces.

Para esta situación, existe un patrón GoF que se utiliza comúnmente: Factoría Abstracta (*Abstract Factory*).

Factoría Abstracta (*Abstract Factory*)

Contexto/Problema

¿Cómo crear familias de clases relacionadas que implementen una interfaz común?

Solución

Defina una interfaz para la factoría (la factoría abstracta). Defina una clase factoría concreta para cada familia de cosas que hay que crear. Opcionalmente, defina una clase abstracta que implemente la interfaz factoría y proporcione servicios comunes a las factorías concretas que la extienden.

⁷ Éstos son nombres de paquetes ficticios.

La Figura 33.15 ilustra la idea básica, que se mejora en la siguiente sección.

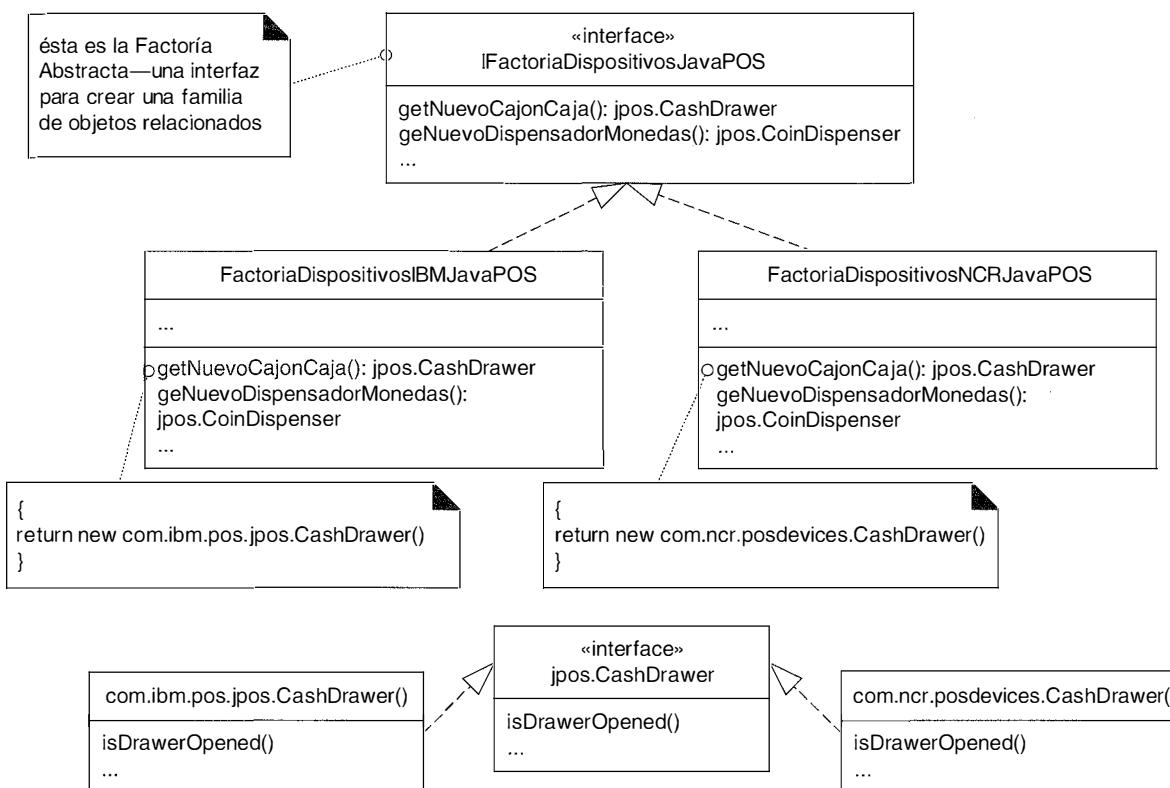


Figura 33.15. Una factoría abstracta básica.

Factoría Abstracta mediante una clase abstracta

Una variación típica de la Factoría Abstracta es crear una clase abstracta factoría a la que se accede utilizando el patrón Singleton; lee una propiedad del sistema para decidir cuál de sus subclases factoría crear, y entonces devuelve la instancia de la subclase apropiada. Esto se utiliza, por ejemplo, en las librerías de Java con la clase `java.awt.Toolkit`, que es una clase abstracta que representa una factoría abstracta para la creación de familias de elementos gráficos del GUI para diferentes sistemas operativos y subsistemas de GUI.

La ventaja de este enfoque es que soluciona este problema: ¿cómo sabe la aplicación qué factoría abstracta utilizar? ¿`FactoriaDispositivosIBMJavaPOS`? ¿`FactoriaDispositivosNCRJavaPOS`?

El siguiente refinamiento soluciona este problema. La Figura 33.16 ilustra la solución.

Con esta clase abstracta como factoría y el método `getInstancia` del patrón Singleton, los objetos pueden colaborar con la superclase abstracta, y obtener una referencia a una de las instancias de sus subclases. Por ejemplo, considere la declaración:

```
cajonCaja=FactoriaDispositivosJavaPOS.getInstancia().getNuevoCajonCaja();
```

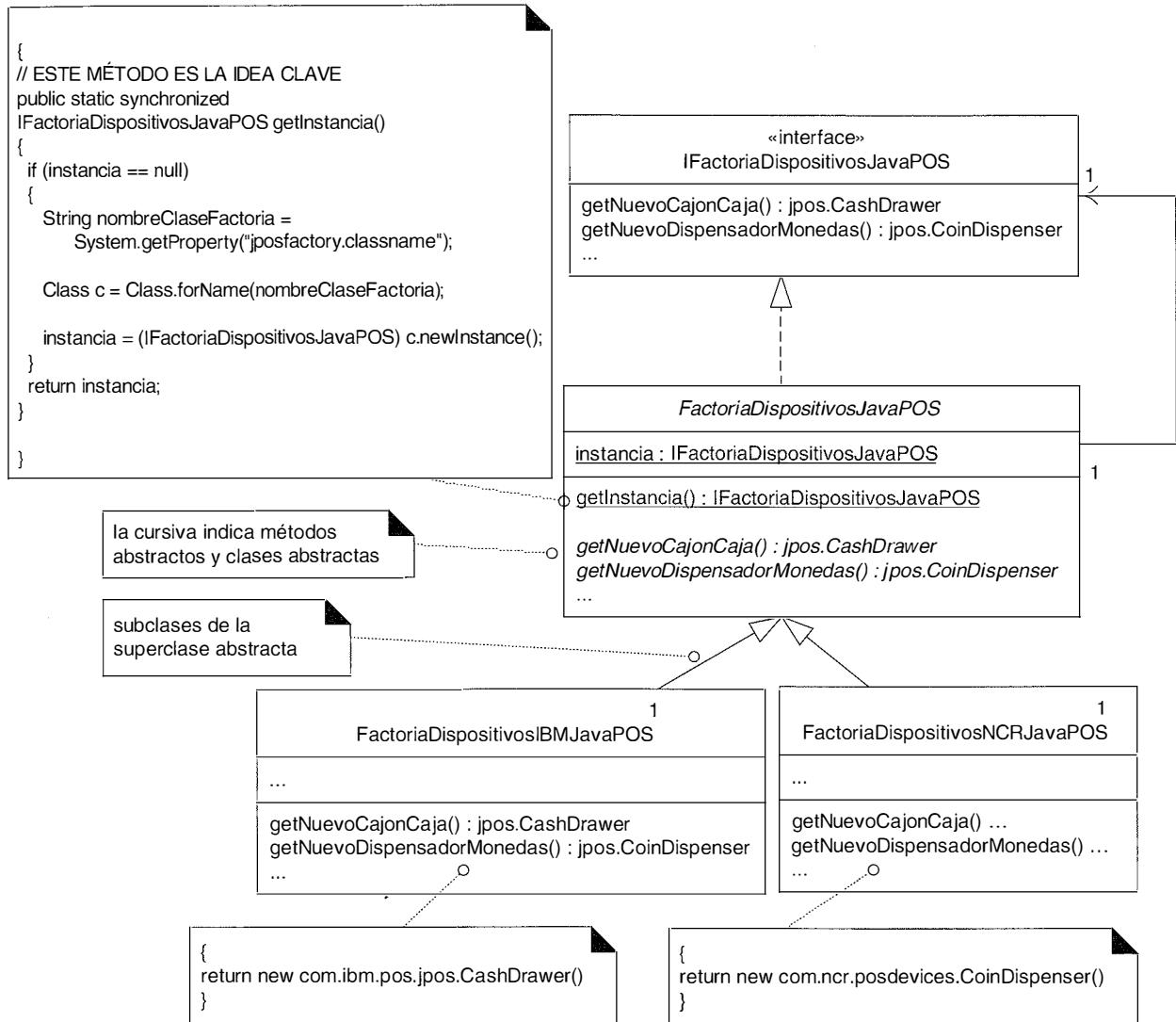


Figura 33.16. Factoría Abstracta mediante una clase abstracta.

La expresión `FactoriaDispositivosJavaPOS.getInstance()` devolverá una instancia de `FactoriaDispositivosIBMJavaPOS` o `FactoriaDispositivosNCRJavaPOS`, dependiendo de la propiedad del sistema que se lea. Obsérvese que cambiando la propiedad del sistema externa “`jposfactory.classname`” (que es el nombre de la clase como un String) en un fichero de propiedades, el sistema NuevaEra utilizará una familia diferente de controladores JavaPOS. Se consiguen Variaciones Protegidas con respecto a los cambios de factoría con un diseño de programación dirigido por los datos (leyendo un fichero de propiedades) y reflexivo, utilizando la expresión `c.newInstance()`.

La interacción con la factoría tendrá lugar en un *Registro*. De acuerdo con el objetivo de salto en la representación bajo, es razonable que el *Registro* software (cuyo nom-

bre sugiere el terminal de PDV global) mantenga una referencia a los dispositivos como el *CajonCaja*. Por ejemplo:

```
class Registro
{
    private jpos.CashDrawer cajonCaja;
    private jpos.CoinDispenser dispensadorMonedas;

    public Registro()
    {
        cajonCaja =
            FactoriaDispositivosJavaPos.getInstancia().getNuevoCajonCaja();
        //...
    }
    //...
}
```

33.7. Gestión de pagos con Polimorfismo y Hacerlo Yo Mismo

Una de las formas habituales de aplicación del polimorfismo (y el Experto en Información) es en el contexto de lo que Peter Coad denomina la estrategia o patrón “Hacerlo Yo Mismo” [Coad95]. Es decir:

Hacerlo Yo Mismo

“Yo (un objeto software) hago las cosas que hace normalmente el objeto actual del que yo soy una abstracción.” [Coad95]

Éste es el estilo de diseño orientado a objetos clásico: los objetos *Circulo* se dibujan ellos mismos, los objetos *Cuadrado* se dibujan ellos mismos, los objetos *Texto* hacen ellos mismos la comprobación ortográfica, y así sucesivamente.

Nótese que el hecho de que un objeto *Texto* compruebe él mismo la ortografía es un ejemplo del Experto en Información: *El objeto que tiene la información relacionada con el trabajo lo hace* (un *Diccionario* también es un candidato, según el Experto).

Hacerlo Yo Mismo y el Experto en Información normalmente nos llevan a la misma elección.

Análogamente, observe que el hecho de que el *Circulo* y el *Cuadrado* se dibujen ellos mismos son ejemplos de Polimorfismo: *Cuando alternativas relacionadas varían según el tipo, asigne la responsabilidad utilizando operaciones polimórficas a los tipos para los que varía el comportamiento*.

Hacerlo Yo Mismo y el Polimorfismo normalmente nos llevan a la misma elección.

No obstante, como se estudió en la discusión acerca de la Fabricación Pura, con frecuencia está contraindicado debido a problemas con el acoplamiento y la cohesión, y en

lugar de eso, un diseñador utiliza fabricaciones puras como estrategias, factorías y otras por el estilo.

A pesar de todo, cuando es apropiado, Hacerlo Yo Mismo es atractivo en parte porque favorece un salto en la representación bajo. El diseño para la gestión de pagos se llevará a cabo con los patrones Hacerlo Yo Mismo y Polimorfismo.

Uno de los requisitos de esta iteración es gestionar múltiples tipos de pagos, que esencialmente significa gestionar los pasos de autorización y contabilidad. Diferentes tipos de pagos se autorizan de distintas formas:

- Los pagos a crédito y débito se autorizan con un servicio de autorización externo. Ambos requieren que se registre una entrada en las cuentas por cobrar —dinero que debe la institución financiera que hace la autorización—.
- En algunas tiendas (es una tendencia en algunos países) se autorizan los pagos en efectivo utilizando un analizador especial de billetes unido al terminal de PDV que comprueba si el dinero es falso. Otras tiendas no lo hacen.
- En algunas tiendas se autorizan los pagos con cheque utilizando un servicio de autorización informatizado. Otras tiendas no autorizan los cheques.

Los *PagosACredito* se autorizan de una forma; los *PagosConCheque* se autorizan de otra. Éste es un caso clásico de Polimorfismo.

De esta manera, como se muestra en la Figura 33.17, cada subclase de *Pago* tiene su propio método *autorizar*.

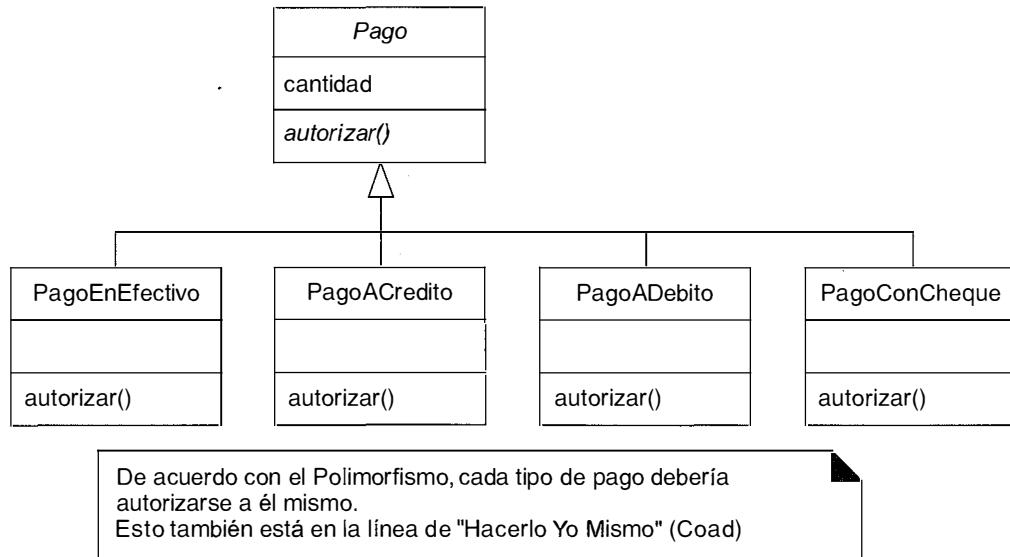


Figura 33.17. Polimorfismo clásico con múltiples métodos *autorizar*.

Por ejemplo, como se ilustra en las Figuras 33.18 y 33.19, una *Venta* instancia un *PagoACredito* o un *PagoConCheque* y le pide que se autorice él mismo.

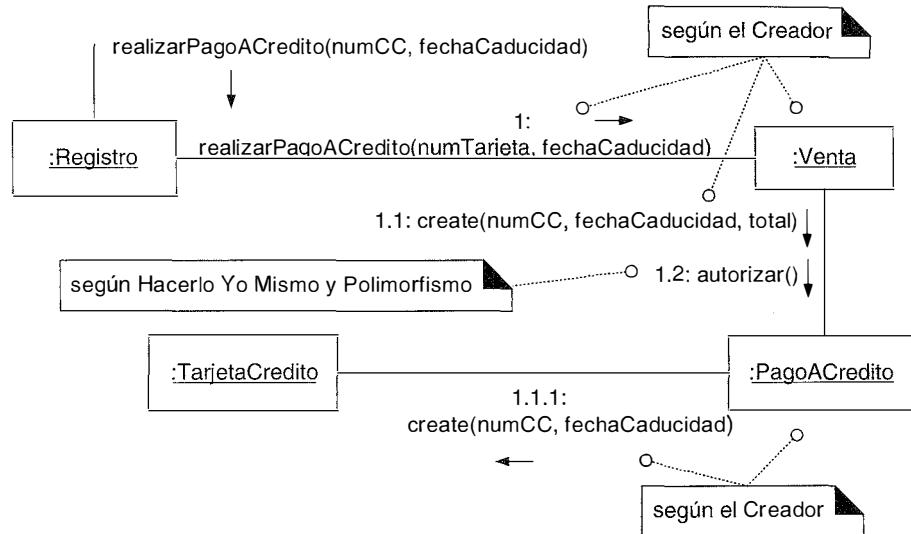


Figura 33.18. Creación de un PagoACredito.

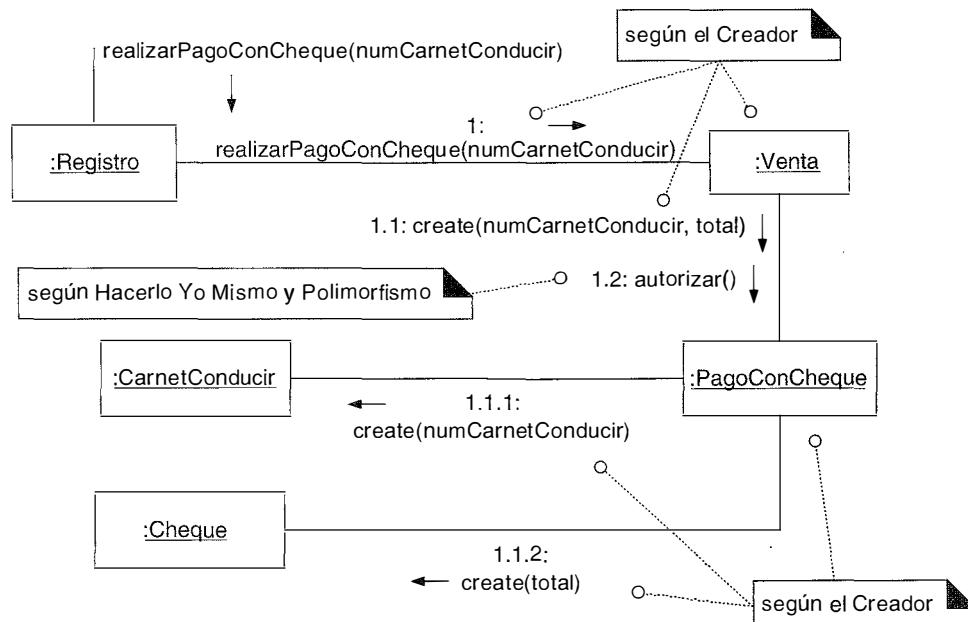


Figura 33.19. Creación de un PagoConCheque.

¿Clases pequeñas?

Considere la creación de los objetos software *TarjetaCredito*, *CarnetConducir* y *Cheque*. Nuestro primer impulso podría ser simplemente registrar los datos que contienen en sus clases de pago relacionadas, y eliminar tales clases pequeñas. Sin embargo, normalmente utilizarlas es una estrategia más beneficiosa; a menudo terminan proporcionando com-

portamiento útil y acaban siendo reutilizables. Por ejemplo, la *TajetaCredito* es un Expero natural en decirnos el tipo de compañía de crédito (Visa, MasterCard, etcétera). Este comportamiento resultará necesario para nuestra aplicación.

Autorización de pago a crédito

El sistema se debe comunicar con un servicio externo de autorización de crédito, y ya hemos creado la base del diseño para dar soporte a esto basada en adaptadores.

Información del dominio sobre el pago a crédito relevante

Establezcamos el contexto para el diseño que vamos a realizar a continuación:

- Los sistemas PDV se conectan físicamente con los servicios externos de autorización de varias formas, entre las que se encuentran líneas telefónicas (que se deben marcar) y conexiones permanentes de Internet de banda ancha.
- Se utilizan diferentes protocolos del nivel de la aplicación y formatos de datos asociados, como Transacciones Electrónicas Seguras (SET, *Secure Electronic Transaction*). Podrían hacerse populares otros nuevos, como XMLPay.
- La autorización de pagos se puede ver como una operación síncrona ordinaria: un hilo de ejecución del PDV se bloquea, esperando una respuesta del servicio remoto (dentro de los límites de un periodo de tiempo de espera).
- Todos los protocolos de autorización de pago llevan el envío de identificadores que identifican de manera única a la tienda (con un “ID de comerciante”), y al terminal del PDV (con un “ID del terminal”). Una respuesta incluye un código de aprobación o denegación, y un ID de transacción único.
- Una tienda podría utilizar diferentes servicios de autorización externos para distintos tipos de tarjetas de crédito (uno para la VISA y otro para la MasterCard). Para cada servicio, la tienda tiene un ID de comerciante distinto.
- El tipo de la compañía de crédito se puede deducir a partir del número de la tarjeta. Por ejemplo, los números que comienzan por 5 son MasterCard; los números que empiezan por 4 son Visa.
- Las implementaciones de los adaptadores protegerán a las capas superiores del sistema contra todas estas variaciones en la autorización de los pagos. Cada adaptador es responsable de asegurar que la transacción de solicitud de la autorización tiene el formato adecuado, y de la colaboración con el servicio externo. Como se discutió en la iteración anterior, la *FactoriaDeServicios* es responsable de enviar la implementación de *IAdaptadorServicioAutorizacionCredito* adecuada.

Un escenario de diseño

La Figura 33.20 comienza la presentación de un diseño con anotaciones que satisfacen estos detalles y requisitos. Los mensajes tienen notas aclaratorias para mostrar el razonamiento.

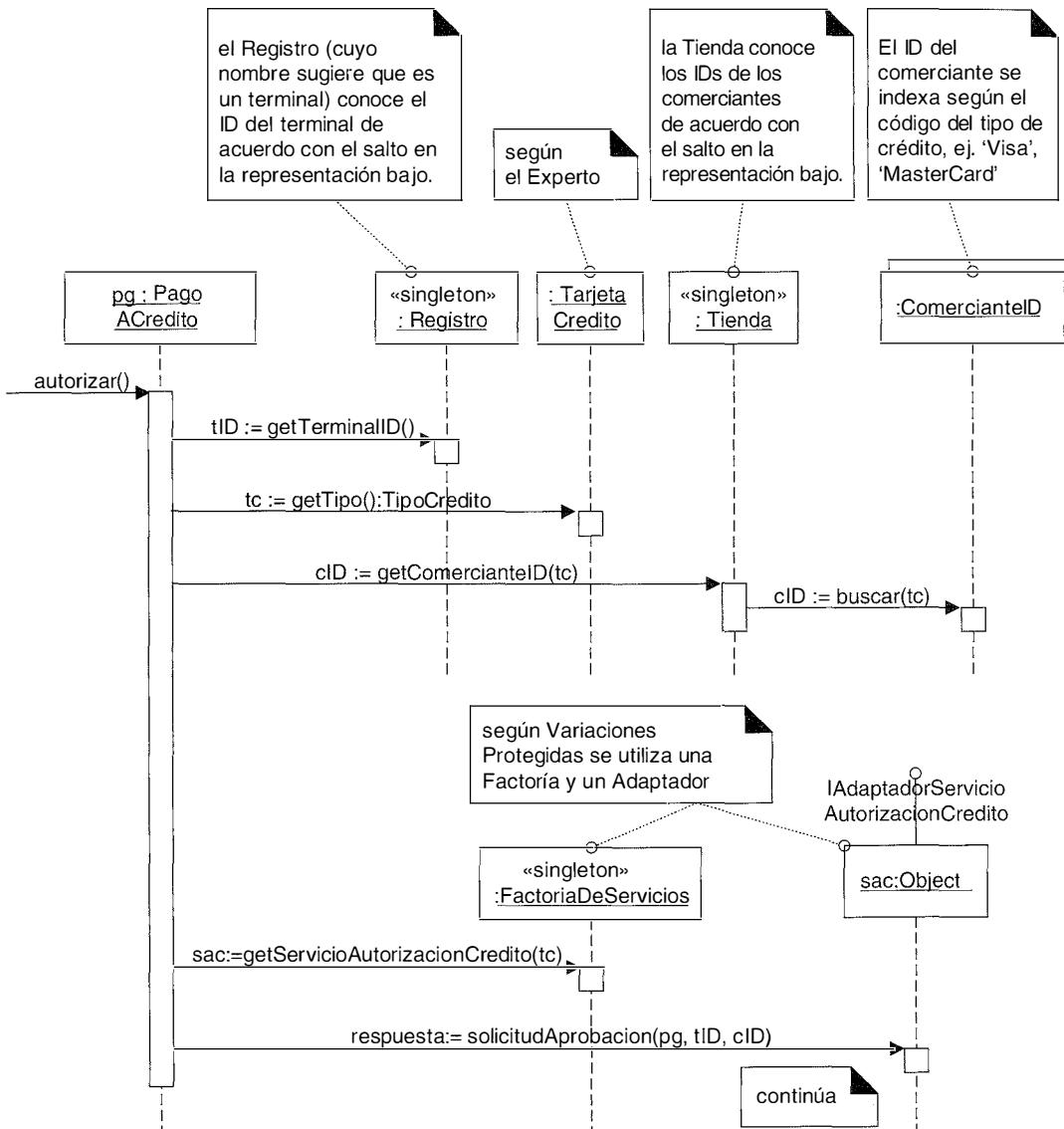


Figura 33.20. Gestión de un pago a crédito.

Una vez que se encuentra el `IAdaptadorServicioAutorizacionCredito` correcto, se le otorga la responsabilidad de completar la autorización, como se muestra en la Figura 33.21.

En el momento que se obtiene la respuesta del `PagoACredito` (al que se le ha dado la responsabilidad de gestionar su finalización de acuerdo con el Polimorfismo y Hacerlo Yo Mismo), asumiendo que se aprueba, completa sus tareas, como se muestra en la Figura 33.22.

Notación UML: Obsérvese en este diagrama de secuencia que se apilaron algunos objetos. Esto es legal, aunque pocas herramientas CASE lo soportan. Es de utilidad para incluirlo en un libro o documento, donde el ancho de página está limitado.

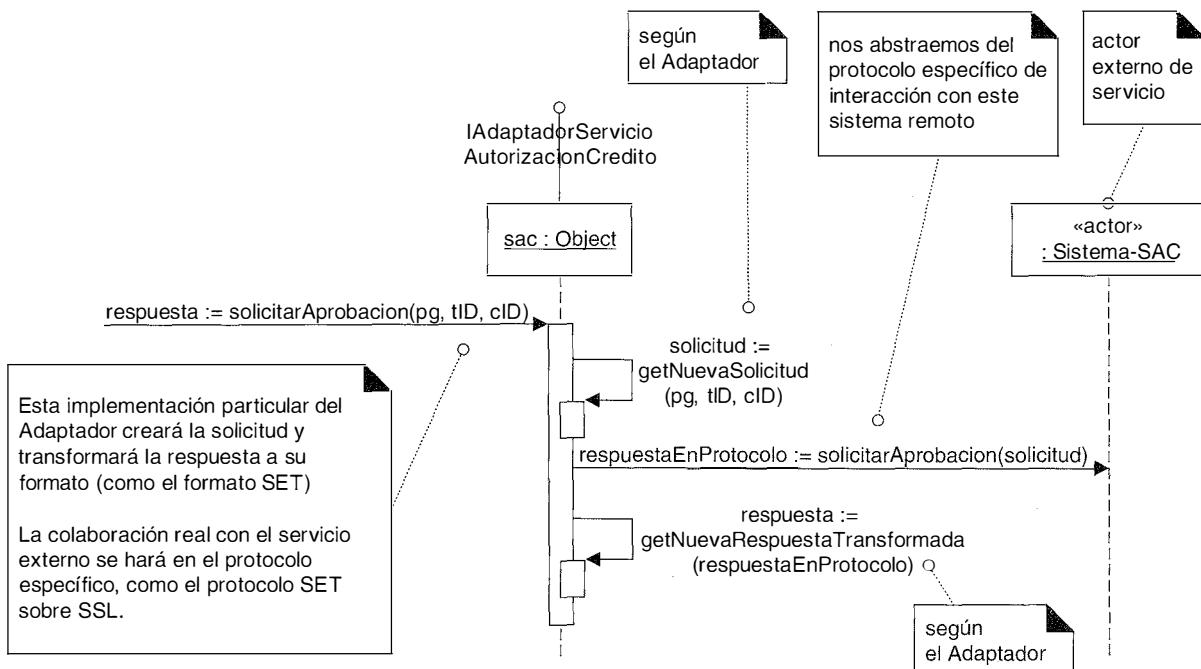


Figura 33.21. Finalización de la autorización.

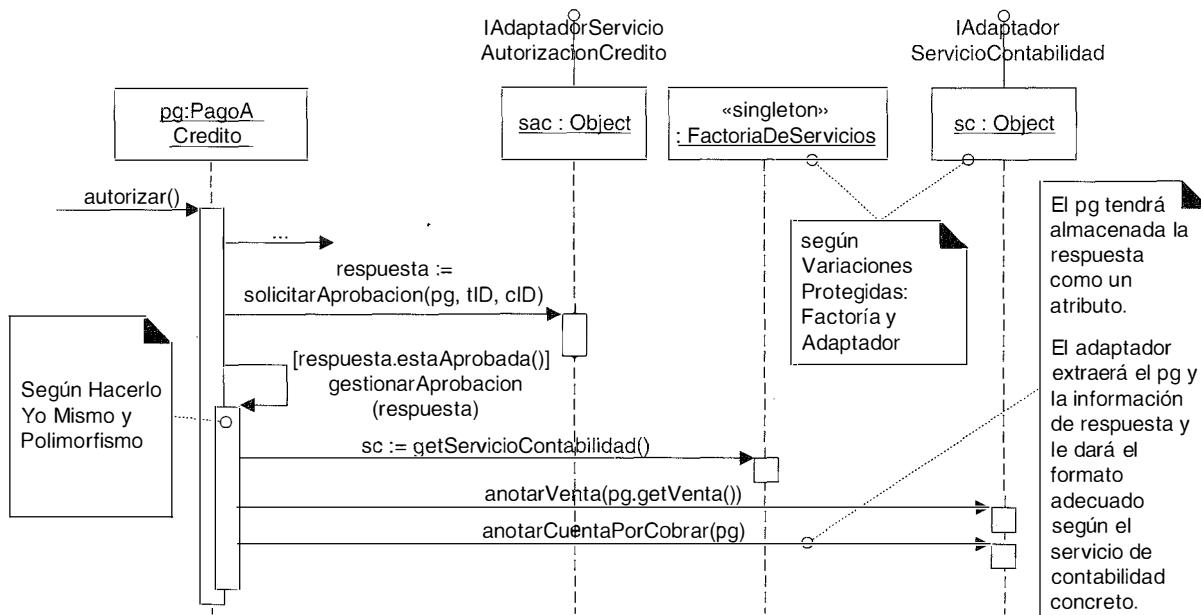


Figura 33.22. Finalización de un pago a crédito aprobado.

33.8. Conclusión

Lo importante de este caso de estudio no era mostrar la solución correcta —no existe una única solución que sea la mejor—, y estoy seguro de que los lectores pueden me-

jorar lo que he propuesto. Espero sinceramente haber demostrado que se puede llevar a cabo el diseño de objetos basado en un razonamiento que siga principios básicos como bajo acoplamiento y la aplicación de patrones, en lugar de ser un proceso misterioso.

Advertencia: Patron-itis

Esta presentación ha utilizado los patrones de diseño GoF en muchos puntos, lo cual es una de las cosas importantes del caso de estudio como ayuda al aprendizaje. Pero, ha habido informes de diseñadores preocupados excesivamente por introducir los patrones de una manera forzada en un frenesí creativo de patron-itis. La conclusión que podemos extraer de esto es que es necesario estudiar los patrones en múltiples ejemplos para digerirlos bien. Un método de aprendizaje extendido es formar un grupo de estudio a la hora de la comida o después del trabajo en el que los participantes comparten las formas que han visto o podrían ver de la aplicación de los patrones, y discutir una sección de un libro sobre patrones.

Capítulo 34

DISEÑO DE UN FRAMEWORK DE PERSISTENCIA CON PATRONES

Le temps est un grand professeur, mais malheureusement il tue tous ses élèves

(El tiempo es un gran profesor, pero desgraciadamente mata a todos sus alumnos).

Hector Berlioz

Objetivos

- Diseñar parte de un framework¹ con los patrones Método Plantilla, Estado y Command.
 - Introducir las cuestiones de la correspondencia (*mapping*) objeto-relacional (O-R).
 - Implementar la materialización perezosa con Proxies Virtuales.
-

Introducción

La aplicación NuevaEra —como la mayoría— requiere que se almacene y recupere la información en mecanismos de almacenamiento persistente, como una base de datos relacional (BDR). Este capítulo presenta el diseño de un framework para el almacenamiento de objetos persistentes.

Normalmente es mejor conseguir o comprar uno de éstos, ya sea un producto independiente o parte de un contenedor que maneja la persistencia para beans entidad (*entity beans*) si se utiliza EJBs y otras tecnologías de Java. Construir un servicio de persistencia O-R de calidad industrial puede llevar mucho esfuerzo persona-año, y existen cuestiones sutiles que requieren una experiencia especializada. Además, las tecnologías como las que se basan en *Java Data Objects* (JDO) ofrecen soluciones parciales.

Por tanto, la intención no es mostrar un framework industrial o sugerir que se ignoren las tecnologías como JDO, sino más bien utilizar un framework de persistencia como

¹ *N. del T.* Los miembros de la comunidad software de habla hispana utilizan el término original en inglés, aunque a veces se traduce framework por marco, elección que no nos parece muy acertada.

medio para explicar el diseño general de frameworks con patrones, puesto que constituye un caso de estudio especialmente bueno. Es también otro ejemplo de utilización de UML para comunicar un diseño software.

Este framework se presenta para introducir el diseño de los frameworks, no como un enfoque recomendado para el diseño de un servicio de persistencia industrial.

34.1. El problema: objetos persistentes

Asuma que en la aplicación NuevaEra, los datos de la *EspecificacionDelProducto* residen en una base de datos relacional. Estos datos deben traerse a la memoria local durante el uso de la aplicación. Los **objetos persistentes** son aquellos que requieren almacenamiento persistente, como las instancias de *EspecificacionDelProducto*.

Mecanismos de almacenamiento y objetos persistentes

Bases de datos de objetos: Si se utiliza una base de datos de objetos para almacenar y recuperar los objetos, no se necesita ningún servicio de persistencia a medida o de terceras partes adicional. Éste es uno de los diversos atractivos de su uso.

Bases de datos relacionales: Debido al predominio de las BDR, a menudo es necesario que se utilicen, en lugar de las bases de datos de objetos que son más convenientes. Si es éste el caso, surgen algunos problemas debido a la incompatibilidad entre la representación de los datos orientada a registros y la orientada a objetos; estos problemas se estudiarán más adelante. Se requiere un servicio especial para establecer la correspondencia O-R (*mapping O-R*).

Otros: Además de las BDR, a veces se desea almacenar los objetos en otros mecanismos de almacenamiento o formatos, como simples ficheros, estructuras XML, ficheros Palm OS PDB, bases de datos jerárquicas, etcétera. Como con las bases de datos relacionales, existen incompatibilidades entre las representaciones de los objetos y estos formatos que no son orientados a objetos. Y como con las BDR, se requieren servicios especiales que hagan que funcionen con objetos.

34.2. La solución: un servicio de persistencia a partir de un framework de persistencia

Un **framework de persistencia** es un conjunto de tipos de propósito general, reutilizable y extensible, que proporciona funcionalidad para dar soporte a los objetos persistentes. Un **servicio de persistencia** (o subsistema) realmente proporciona el servicio, y se creará con un framework de persistencia. Un servicio de persistencia se escribe normalmente para que trabaje con BDR, en cuyo caso también se conoce como **servicio de correspondencia O-R**. Generalmente, un servicio de persistencia tiene que traducir los objetos a registros (o a alguna otra forma de datos estructurada como XML) y guardarlos en una base de datos, y traducir los registros a objetos cuando los recuperamos de la base de datos.

En cuanto a la arquitectura en capas de la aplicación NuevaEra, un servicio de persistencia es un subsistema dentro de la capa de servicios técnicos.

34.3. Frameworks

Aun a riesgo de simplificar en exceso, un framework es un conjunto *extensible* de objetos para funciones relacionadas. El ejemplo prototípico es un framework de GUI, como las AWT o Swing de Java.

La señal de calidad de un framework es que proporciona una implementación para las funciones básicas e invariables, e incluye un mecanismo que permite al desarrollador conectar las funciones que varían, o extender las funciones.

Por ejemplo, el framework Swing de GUI de Java proporciona muchas clases e interfaces para las funciones principales de la GUI. Los desarrolladores pueden incluir elementos gráficos especializados creando subclases de las clases Swing y redefiniendo ciertos métodos. Los desarrolladores también pueden conectar diversos comportamientos de respuesta a los eventos en las clases de los elementos gráficos predefinidas (como *JButton*) registrando oyentes o suscriptores basados en el patrón Observador. Eso es un framework.

En general, un **framework**:

- Es un conjunto cohesivo de interfaces y clases que colaboran para proporcionar los servicios de la parte central e invariable de un subsistema lógico.
- Contiene clases concretas (y especialmente) abstractas que definen las interfaces a las que ajustarse, interacciones de objetos en las que participar, y otras invariantes.
- Normalmente (aunque no necesariamente) requiere que el usuario del framework defina subclases de las clases del framework existentes para utilizar, adaptar y extender los servicios del framework.
- Tiene clases abstractas que podrían contener tanto métodos abstractos como concretos.
- Confía en el **Principio Hollywood** —“*No nos llame, nosotros le llamaremos*”. Esto significa que las clases definidas por el usuario (por ejemplo, nuevas clases) recibirán mensajes desde las clases predefinidas del framework. Estos mensajes normalmente se manejan implementando métodos abstractos en las superclases.

El siguiente ejemplo del framework de persistencia describirá y explicará estos principios.

Los frameworks son reutilizables

Los frameworks ofrecen un alto grado de reutilización —mucho más que con clases individuales—. En consecuencia, si una organización está interesada (¿y quién no lo está?) en incrementar su grado de reutilización del software, entonces debería enfatizar la creación de frameworks.

34.4. Requisitos para el servicio y framework de persistencia

Para la aplicación de PDV NuevaEra, necesitamos un servicio de persistencia que se construya con un framework de persistencia (que se podría utilizar también para crear otros servicios de persistencia). Llámemos al framework FWP (Framework de Persis-

tencia). FWP es un framework simplificado —un framework de persistencia desarrollado de calidad industrial queda fuera del alcance de esta introducción—.

El framework debería proporcionar funciones para:

- almacenar y recuperar los objetos en un mecanismo de almacenamiento persistente
- *confirmar y deshacer (commit y rollback)* las transacciones

El diseño debe ser extensible para dar soporte a diferentes mecanismos de almacenamiento, como BDRs, registros en ficheros simples, o XML en ficheros.

34.5. Ideas claves

Las siguientes ideas claves se estudiarán en las secciones que vienen a continuación:

- **Correspondencia:** Se debe establecer alguna correspondencia (*mapping*) entre una clase y su almacenamiento persistente (por ejemplo, una tabla en una base de datos), y entre los atributos de los objetos y los campos (columnas) en un registro. Es decir, debe existir un **correspondencia de esquemas** entre los dos esquemas.
- **Identidad de objeto:** Los registros y los objetos tienen un único identificador de objeto para relacionar fácilmente los registros con los objetos, y asegurar que no hay duplicados inapropiados.
- **Conversor de base de datos:** Una Fabricación Pura conversor (*mapper*) de base de datos es responsable de la materialización y desmaterialización.
- **Materialización y desmaterialización:** La materialización es el acto de transformar una representación de datos no orientada a objetos (por ejemplo, registros) de un almacenamiento persistente en objetos. La desmaterialización es la actividad opuesta (también conocida como *passivation*).
- **Caché:** Los servicios persistentes almacenan en una caché los objetos materializados por razones de rendimiento.
- **Estado de transacción de los objetos:** Es útil conocer el estado de los objetos en función de sus relaciones con la transacción actual. Por ejemplo, es útil conocer qué objetos se han modificado (están *sucios*) de manera que es posible determinar si es necesario que se guarden de nuevo en su almacenamiento persistente.
- **Operaciones de transacción:** Operaciones confirmar y deshacer (*commit* y *rollback*).
- **Materialización perezosa:** No todos los objetos se materializan de una vez; una instancia particular sólo se materializa bajo demanda, cuando se necesita.
- **Proxies virtuales:** La materialización perezosa se puede implementar utilizando una referencia inteligente que se conoce como proxy virtual.

34.6. Patrón: Representación de Objetos como Tablas

¿Cómo conviertes un objeto en un registro o esquema de base de datos relacional?

El patrón **Representación de Objetos como Tablas** [BW96] propone la definición de una tabla en una BDR por cada clase de objeto persistente. Los atributos de los

objetos que contienen tipos de datos primitivos (número, cadena de texto, booleano, etcétera) se corresponden con las columnas.

Si un objeto sólo tiene atributos de tipos de datos primitivos, la correspondencia es directa. Pero como veremos, las cosas no son tan simples puesto que los objetos podrían tener atributos que hacen referencia a otros objetos complejos, mientras que el modelo relacional requiere que los valores sean atómicos (esto es, la Primera Forma Normal) (ver Figura 34.1).

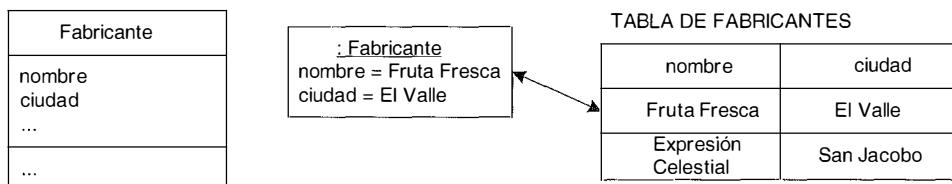


Figura 34.1. Correspondencia entre objetos y tablas.

34.7. Perfil (*Profile*) de modelado de datos en UML

A propósito de las BDR, no es sorprendente que UML se haya convertido en una notación muy utilizada para los **modelos de datos**. Fíjese que uno de los artefactos oficiales del UP es el Modelo de Datos, que forma parte de la disciplina de Diseño. La Figura 34.2 ilustra alguna notación UML para el modelado de datos.

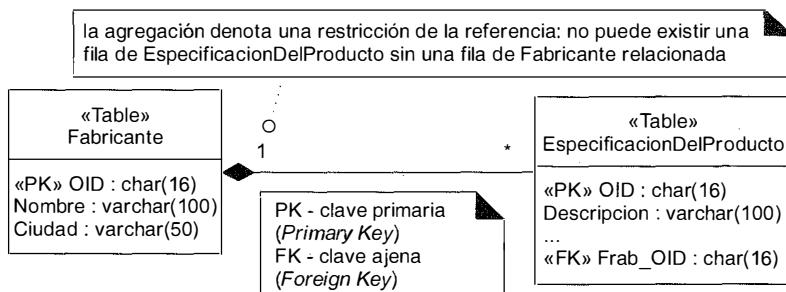


Figura 34.2. Ejemplo del Perfil (*Profile*) de Modelado de Datos en UML.

Estos estereotipos no forman parte del núcleo de UML —son extensiones—. Generalizando, UML tiene el concepto de **profile** (perfil de UML): un conjunto coherente de estereotipos de UML, valores etiquetados y restricciones para un propósito específico. La Figura 34.2 ilustra parte del Perfil de Modelado de Datos de UML propuesto (al OMG); en el momento en el que se escribió este libro no se había aprobado. Un perfil no tiene que estar aprobado por el OMG para ser un perfil, aunque se están enviando algunos de uso muy extendido —como el modelado de datos— para que se aprueben.

34.8. Patrón: Identificador de Objeto

Es conveniente contar con una forma consistente de relacionar los objetos con los registros, y ser capaces de asegurar que la materialización repetida de un registro no da como resultado objetos duplicados.

El patrón **Identificador de Objeto** [BW96] propone asignar un **identificador de objeto** (OID) a cada registro y objeto (o proxy de un objeto).

Un OID normalmente es un valor alfanumérico; cada uno es único para un objeto específico. Existen varios enfoques para generar identificadores únicos para los OIDs, variando desde únicos para una base de datos, a únicos globalmente: generadores de secuencia de bases de datos, la estrategia de generación de claves Alto-Bajo [Ambler00], y otros.

En el campo de los objetos, un OID se representa mediante una interfaz o clase OID que encapsula el valor real y su representación. En un BDR, normalmente se almacena como un valor de tipo carácter de longitud fija.

Cada tabla tendrá un OID como clave primaria, y cada objeto también tendrá (directa o indirectamente) un OID. Si se asocia cada objeto con un OID, y cada tabla tiene un OID como clave primaria, cada objeto se corresponde de manera única con una fila de alguna tabla (ver Figura 34.3).

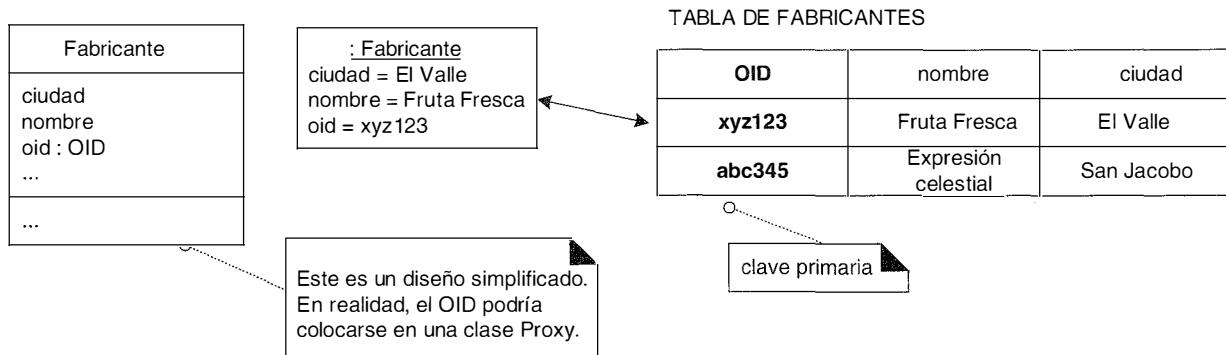


Figura 34.3. Los identificadores de los objetos enlazan objetos y registros.

Ésta es una vista simplificada del diseño. En realidad, el OID podría no colocarse exactamente en el objeto persistente —aunque sea posible—. En lugar de eso, se podría colocar en un objeto Proxy que envuelve al objeto persistente. En el diseño influye la elección del lenguaje.

Un OID también proporciona un tipo de clave consistente para utilizarla en la interfaz con el servicio de persistencia.

34.9. Acceso al servicio de persistencia con una Fachada

El primer paso del diseño de este subsistema es definir una fachada para estos servicios; recordemos que la Fachada es un patrón común para proporcionar una interfaz uniforme a un subsistema. Para empezar, se necesita una operación para recuperar un objeto dado un OID. Pero además del OID, el subsistema necesita conocer el tipo del objeto que se va a materializar; por tanto, también se debe proporcionar el tipo de la clase. La Figura 34.4 ilustra algunas operaciones de la fachada y su uso en colaboración con uno de los adaptadores de servicios de NuevaEra.

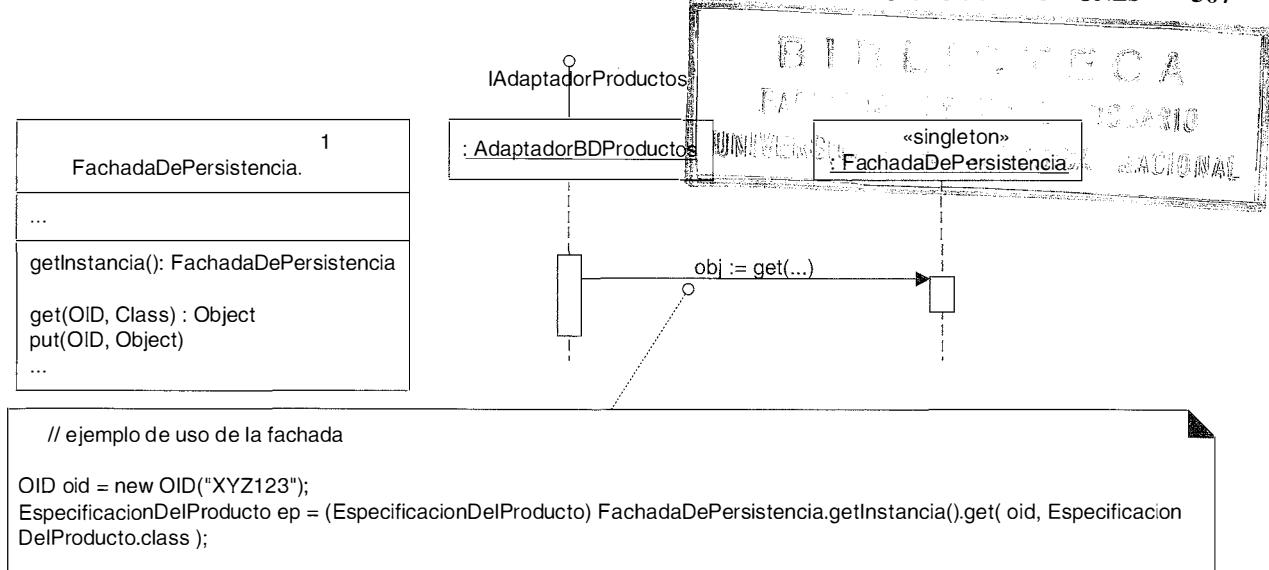


Figura 34.4. La FachadaDePersistencia.

34.10. Correspondencia de los objetos: patrón Conversor (*Mapper*) de Base de Datos o Intermediario (*Broker*) de Base de Datos

La *FachadaDePersistencia* —como se cumple en todas las fachadas— no hace ella misma el trabajo, sino que delega las peticiones en objetos del subsistema.

¿Quién debe ser el responsable de la materialización y desmaterialización de los objetos (por ejemplo, una *EspecificacionDelProducto*) procedentes de un almacenamiento persistente?

El patrón Experto en Información sugiere que la propia clase del objeto (*EspecificacionDelProducto*) persistente es candidata, porque tiene algunos de los datos (los datos que se van a almacenar) que necesita la responsabilidad.

Si una clase de objetos persistentes define el código para almacenarse ella misma en una base de datos, se denomina diseño de **correspondencia directa**. Se puede utilizar la correspondencia directa si el código relacionado de la base de datos se genera y se inyecta automáticamente en la clase mediante un compilador de post-procesamiento, y el desarrollador nunca tiene que ver o mantener este código de base de datos complejo que añade confusión a su clase.

Pero si la correspondencia directa se añade y mantiene de manera manual, tiene varios defectos y tiende a no ser escalable en cuanto a la programación y el mantenimiento. Entre los problemas encontramos:

- Fuerte acoplamiento de la clase de objetos persistentes y el conocimiento del almacenamiento persistente —violación de Bajo Acoplamiento—.
- Responsabilidades complejas en un área nueva y no relacionada con las responsabilidades previas del objeto —violación de Alta Cohesión y mantenimiento de la

separación de intereses—. Cuestiones relacionadas con servicios técnicos se mezclan con otras propias de la lógica de la aplicación.

Estudiaremos un enfoque clásico de **correspondencia indirecta**, que utiliza otros objetos para establecer la correspondencia con los objetos persistentes.

Parte de este enfoque es utilizar el patrón **Intermediario (Broker) de Base de Datos** [BW95]. Éste propone crear una clase que sea responsable de materializar y desmaterializar un objeto almacenado. También se le ha llamado patrón **Conversor (Mapper) de Base de Datos** en [Fowler01], que es un nombre más adecuado que Broker de Bases de Datos, puesto que describe su responsabilidad, y el término “broker” en el diseño de los sistemas distribuidos [BMRSS96] tiene un significado distinto, que existe desde hace mucho tiempo².

Se define una clase diferente que establece la correspondencia para cada clase de los objetos persistentes. La Figura 34.5 ilustra que cada objeto persistente podría tener su propia clase que lleve a cabo la correspondencia, y que podrían existir diferentes tipos de conversores para diferentes tipos de mecanismos de almacenamiento. Un extracto del código:

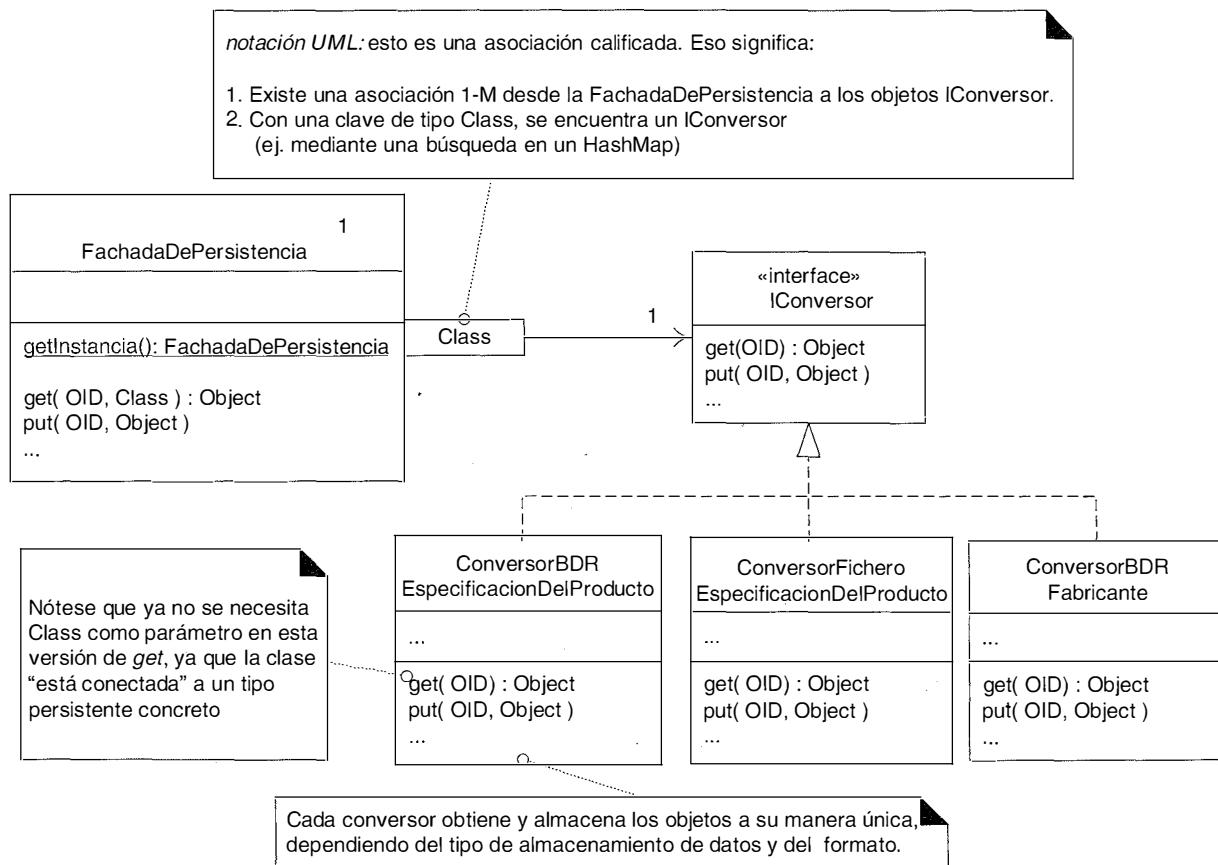


Figura 34.5. Conversores de bases de datos.

² En los sistemas distribuidos, un **broker** es un proceso del servidor front-end que delega las tareas en los procesos del servidor back-end.

```

class FachadaDePersistencia
{
//...
public Object get(OID oid, Class clasePersistente)
{
    //la clave de IConversor es la Clase del objeto persistente
    IConversor conversor = (IConversor)conversores.get(clasePersistente);

    //delega
    return conversor.get(oid);
}
//...
}

```

Aunque este diagrama señala dos conversores para la *EspecificacionDelProducto*, sólo uno de ellos estará activo en un servicio de persistencia en funcionamiento.

Conversores basados en metadatos

Un diseño de conversores más flexible, pero más difícil, se basa en **metadatos** (datos sobre los datos). A diferencia de las clases hechas a mano que establecen la correspondencia de manera individual para diferentes tipos persistentes, los conversores basados en metadatos generan dinámicamente la correspondencia entre un esquema de objetos y otro esquema (como el relacional) en base a la lectura de metadatos que describen la correspondencia, como “La TablaX se corresponde con la Clase Y; la columna Z se corresponde con la propiedad P del objeto” (llega a ser mucho más complejo). Este enfoque es viable en lenguajes con capacidades de programación reflexiva, como Java, C# o Smalltalk, y resulta difícil para aquellos que no las tienen, como C++.

Estableciendo la correspondencia en base a metadatos, podemos cambiar la correspondencia de esquemas en un almacenamiento externo y tendrá efecto en el sistema en marcha, sin que haya que cambiar el código fuente —Variaciones Protegidas con respecto a las variaciones en los esquemas—.

No obstante, una cualidad útil del framework que se ha presentado aquí es que se pueden utilizar los conversores escritos a mano o basados en metadatos, sin afectar a los clientes —encapsulación de la implementación—.

34.11. Diseño del framework con el patrón Método Plantilla

La siguiente sección describe algunas de las características esenciales del diseño de los Conversores de Bases de Datos, que constituyen una parte central del FWP. Estas características de diseño se basan en el patrón de diseño GoF **Método Plantilla** (*Template Method*) [GHJV95]³. Este patrón es una parte esencial del diseño del framework⁴, y es familiar para la mayoría de los programadores OO por la práctica si no por el nombre.

³ Este patrón no está relacionado con las plantillas (*template*) de C++. El patrón describe la *plantilla* de un algoritmo.

⁴ De manera más específica, de los **frameworks de caja blanca**. Normalmente éstos son frameworks orientados a la definición de subclases y jerarquías de clases que requieren que los usuarios conozcan algo acerca de su diseño y estructura; de ahí lo de caja blanca.

La idea es crear un método (el Método Plantilla) en una superclase que define el esqueleto de un algoritmo, con sus partes variables e invariables. El Método Plantilla invoca otros métodos, algunos de los cuales podrían redefinirse en una subclase. De esta manera, las subclases pueden redefinir los métodos que varían para añadir su propio comportamiento único en los puntos de variabilidad (ver Figura 34.6).

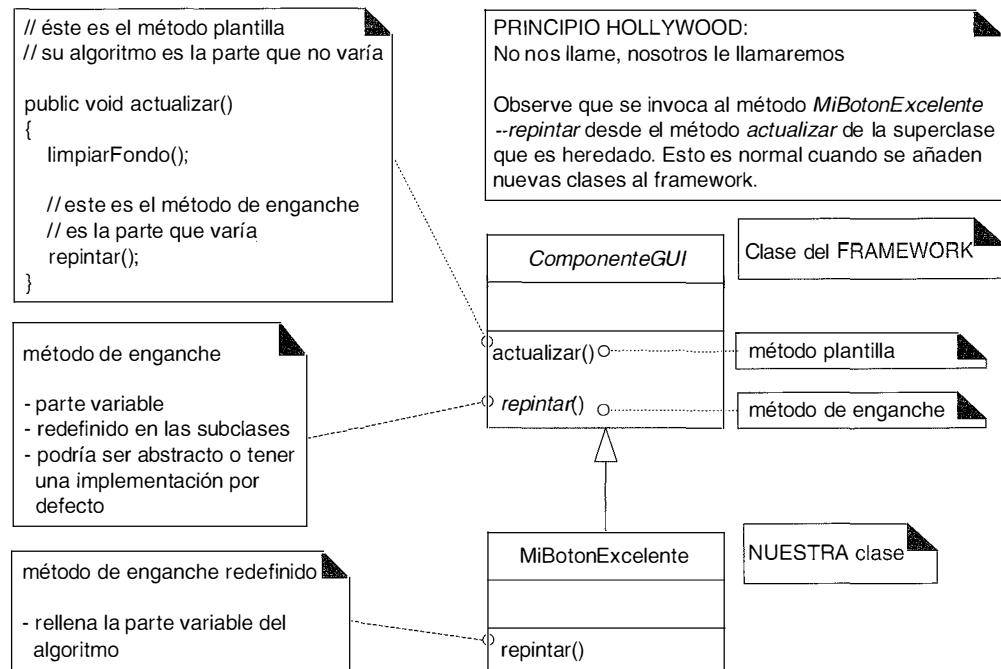


Figura 34.6. Patrón Método Plantilla en un framework de GUI.

34.12. Materialización con el patrón Método Plantilla

Si tuviéramos que programar dos o tres clases para establecer la correspondencia con el mecanismo de almacenamiento permanente, apreciaríamos partes comunes en el código. La estructura básica que se repite del algoritmo para materializar un objeto es:

```
if (objeto está en cache)
    return obj
else
    crear el objeto a partir de su representación en el almacenamiento
    guardar el objeto en la cache
    return obj
```

El punto de variación es la manera de crear el objeto a partir del almacenamiento.

Crearemos el método *get* que será el método plantilla en una superclase abstracta *ConversorPersistenciaAbstracto* que define la plantilla, y utiliza un método “de enganche” (*hook method*) en las subclases para la parte que varía. La Figura 34.7 muestra el diseño esencial.

Como se muestra en este ejemplo, es normal que el método plantilla sea *público*, y que el método de enganche sea *protegido*. El *ConversorPersistenciaAbstracto* e *IConversor* forman parte del FWP. Ahora, un programador de aplicaciones puede incorporar

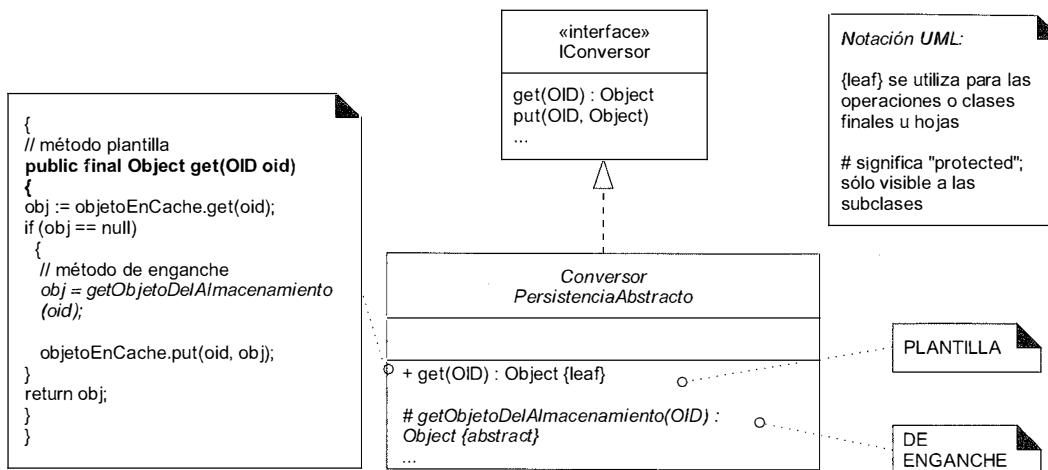
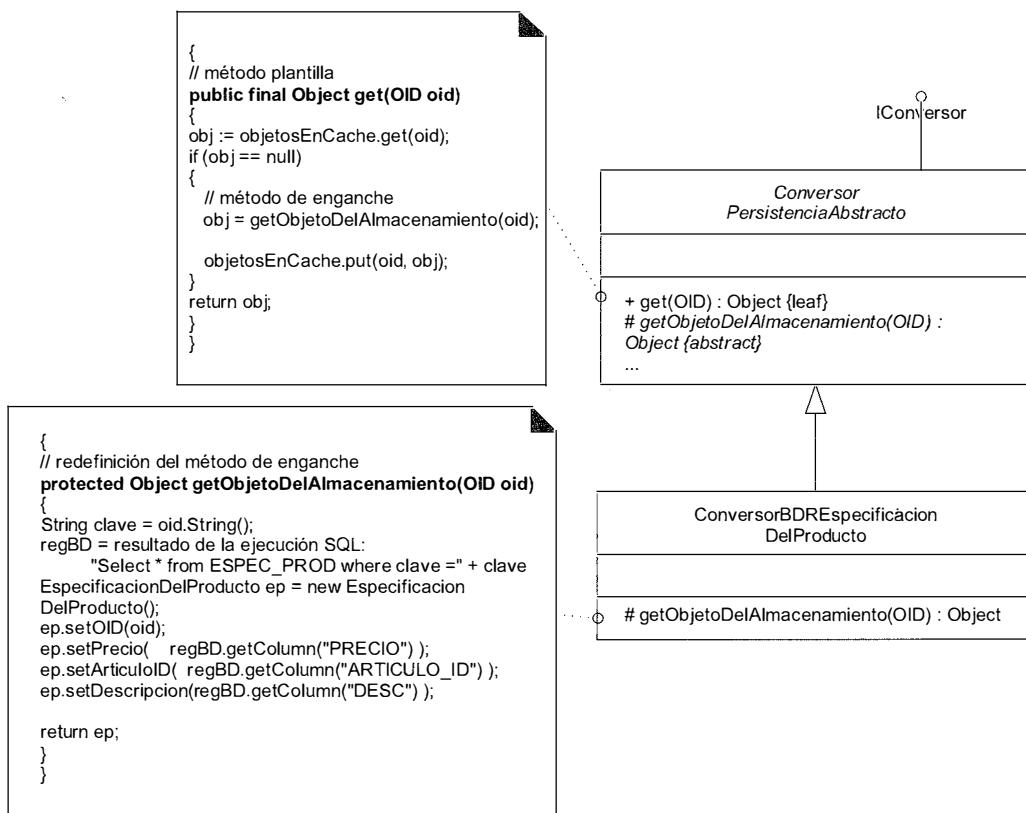


Figura 34.7. Método Plantilla para objetos conversores.

elementos en este framework añadiendo una subclase, y redefiniendo o implementando el método de enganche `getObjetoDelAlmacenamiento`. La Figura 34.8 muestra un ejemplo.

Figura 34.8. Redefinición del método de enganche⁵.

⁵ En Java por ejemplo, el `regBD` que devuelve la ejecución de la consulta SQL sería un `ResultSet` de JDBC.

Asuma que en la implementación del método de enganche de la Figura 34.8, la primera parte del algoritmo —la ejecución del SELECT de SQL— es la misma para todos los objetos, sólo varía el nombre de la tabla de la base de datos⁶. Si se sostiene esta suposición, entonces una vez más, podría aplicarse el patrón Método Plantilla para factorizar por separado las partes que varían y las que no. En la Figura 34.9, la parte artificial es que *ConversorAbstractoBDR--getObjetoDelAlmacenamiento* es un método de enganche con respecto al método *ConversorPersistenciaAbstracto--get*, pero es un método plantilla con respecto al nuevo método de enganche *getObjetoDelRegistro*.

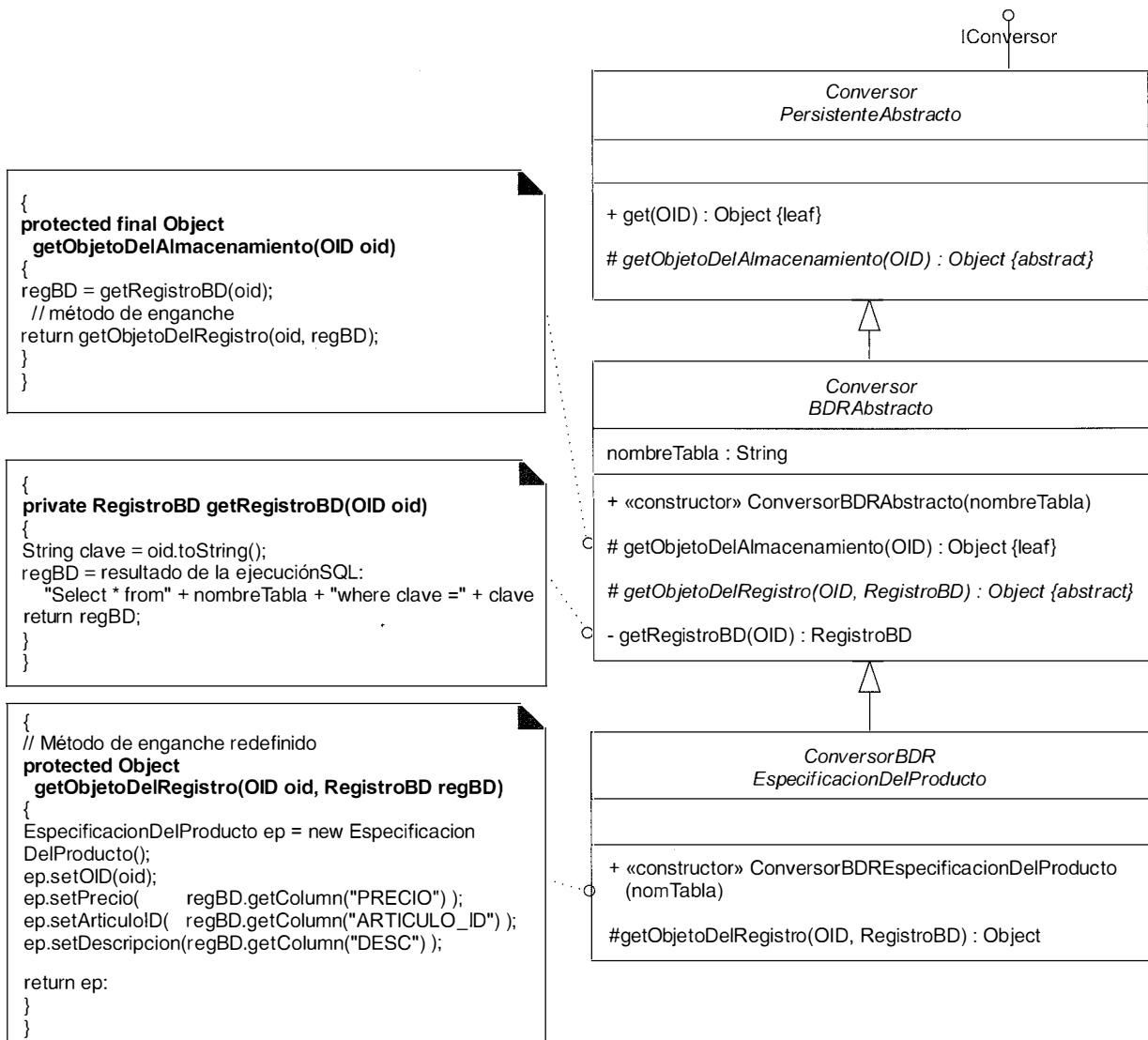


Figura 34.9. Ajustando el código de nuevo con el Método Plantilla.

⁶ En muchos casos, la situación no es tan simple. Un objeto podría derivarse a partir de los datos contenidos en dos o más tablas o a partir de múltiples bases de datos, en cuyo caso, la primera versión del diseño del Método Plantilla es más flexible.

Notación UML: Observe cómo se pueden declarar los constructores en UML. El estereotipo es opcional, y si se utiliza la convención de que el nombre del constructor sea igual al nombre de la clase, probablemente es innecesario.

Ahora *IConversor*, *ConversorPersistenciaAbstracto* y *ConversorBDRAbstracto* forman parte del framework. El programador de la aplicación sólo necesita añadir su subclase, como *ConversorBDREspecificacionDelProducto*, y asegurar que se crea con el nombre de la tabla (se pasa mediante el encadenamiento de constructores hasta el *ConversorBDRAbstracto*).

La jerarquía de clases del Conversor de Base de Datos es una parte esencial del framework; el programador de la aplicación podría añadir nuevas subclases para adaptarlo a nuevos tipos de mecanismos de almacenamiento persistente o nuevas tablas o ficheros específicos en un mecanismo de almacenamiento existente. La Figura 34.10 muestra la estructura de algunos de los paquetes y clases. Nótese que las clases específicas del proyecto NuevaEra no pertenecen al paquete general de servicios técnicos *Persistencia*.

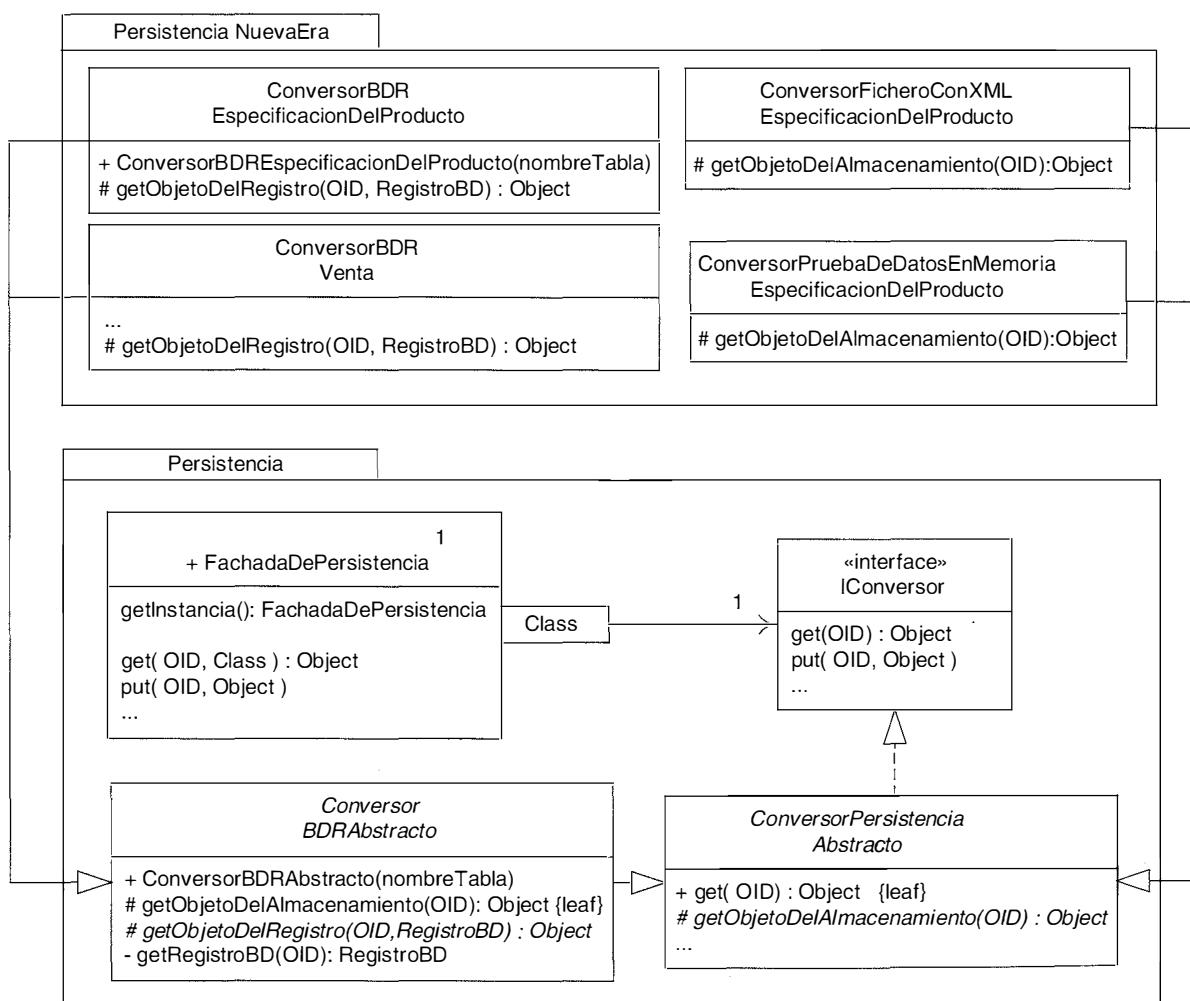


Figura 34.10. El framework de persistencia.

Creo que este diagrama, combinado con la Figura 34.9, ilustra el valor de un lenguaje visual como UML para describir las partes del software; transmite de manera concisa mucha información.

Obsérvese la clase *ConversorPruebaDeDatosEnMemoriaEspecificacionDelProducto*. Tales clases se pueden utilizar para servir objetos con valores definidos directamente en el código para hacer pruebas, sin acceder a ningún almacenamiento persistente externo.

El UP y el Documento de la Arquitectura del Software (SAD)

En cuanto al UP y la documentación, recordemos que el SAD sirve de ayuda para el aprendizaje de futuros desarrolladores, que contiene vistas de la arquitectura con las ideas claves relevantes. La inclusión de diagramas como los de las Figuras 34.9 y 34.10 en el SAD para el proyecto NuevaEra está muy en la línea del tipo de información que un SAD debería contener.

Métodos sincronizados o de guarda en UML

El método *ConversorPersistenciaAbstracto--get* contiene código con secciones críticas que no es seguro en el hilo de ejecución —se podría materializar el mismo objeto de manera concurrente en hilos diferentes—. Como subsistema de servicios técnicos, el servicio de persistencia necesita que se diseñe teniendo presente la seguridad de los hilos. De hecho, el subsistema completo podría estar distribuido en un proceso separado en otro ordenador, transformando la *FachadaDePersistencia* en un objeto servidor remoto, y con muchos hilos ejecutándose simultáneamente en el subsistema, sirviendo a múltiples clientes.

Por tanto, el método debería tener control de la concurrencia de los hilos —si se utiliza Java, se añadirá la palabra clave *synchronized*—. La Figura 34.11 ilustra un método sincronizado en un diagrama de clases.

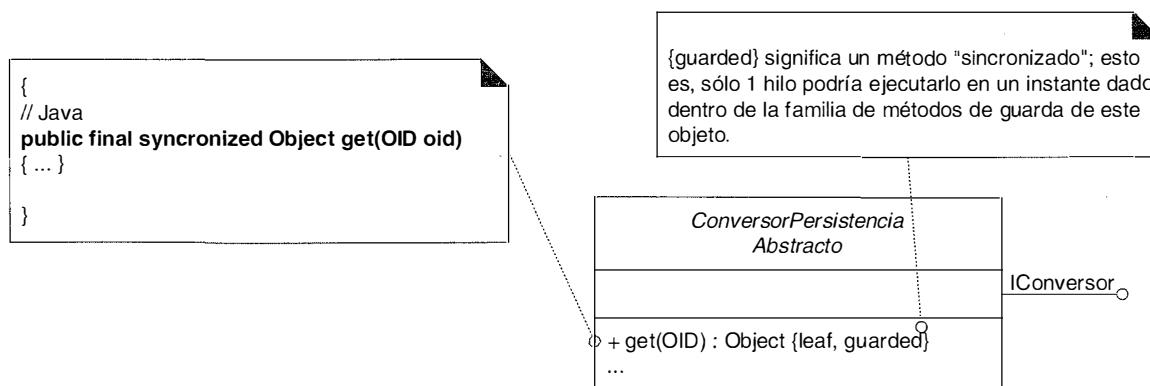


Figura 34.11. Métodos de guarda en UML.

34.13. Configuración de conversores con una FactoriaDeConversores

Análogamente a los ejemplos anteriores de factorías en el caso de estudio, la configuración de la *FachadaDePersistencia* con un conjunto de objetos *IConversor* se puede conseguir con un objeto factoría, *FactoriaDeConversores*. Sin embargo, un pequeño giro, no es conveniente nombrar cada conversor con una operación diferente. Por ejemplo, no es deseable:

```
class FactoriaDeConversores
{
    public IConversor getConversorEspecificacionDelProducto(){...}
    public IConversor getConversorVenta(){...}
    ...
}
```

Esto no soporta Variaciones Protegidas con respecto a la lista creciente de conversores —y crecerá—. En consecuencia, es preferible lo siguiente:

```
class FactoriaDeConversores
{
    public Map getTodosLosConversores(){...}
    ...
}
```

donde las claves del *java.util.Map* (probablemente implementado con un *HashMap*) son los objetos *Class* (los tipos persistentes), y los objetos *IConversor* son los valores.

Entonces, la fachada puede inicializar su colección de objetos *IConversor* como sigue:

```
class FachadaDePersistencia
{
    private java.util.Map conversores =
        FactoriaDeConversores.getInstancia().getTodosLosConversores();
    ...
}
```

La factoría puede asignar un conjunto de objetos *IConversor* utilizando un diseño dirigido por los datos. Es decir, la factoría puede leer las propiedades del sistema para descubrir qué clases *IConversor* debe instanciar. Si se utiliza un lenguaje con capacidades de programación reflexiva, como Java, entonces la instanciación se puede basar en la lectura de los nombres de las clases como cadenas de texto, y la utilización de algo como la operación *Class.newInstance* para crear las instancias. De esta manera, se puede reconfigurar el conversor sin cambiar el código fuente.

34.14. Patrón: Gestión de Caché

Es conveniente mantener los objetos materializados en una caché local para mejorar el rendimiento (la materialización es relativamente lenta) y dar soporte a las operaciones de gestión de las transacciones como *commit*.

El patrón **Gestión de Caché (Cache Management)** [BW96] propone que el Conversor de Base de Datos sea el responsable de mantener esta caché. Si se utiliza un conversor diferente para cada clase de objetos persistentes, cada conversor puede mantener su propia caché.

Cuando se materializan los objetos, se colocan en la caché, con su OID como clave. La siguientes peticiones al conversor para obtener un objeto provocará que el conversor busque primero en la caché, evitando de esta manera materializaciones innecesarias.

34.15. Reunir y ocultar sentencias SQL en una clase

Sentencias SQL embebidas en diferentes clases conversores no es un pecado terrible, pero se puede mejorar. Suponga que en lugar de eso:

- Existe una única clase Fabricación Pura (y es un singleton) *OperacionesBDR* donde se reúnen todas las operaciones SQL (SELECT, INSERT,...).
- Las clases conversores de BDR colaboran con ella para obtener un registro de BD o conjunto de registros (por ejemplo, *ResultSet*).
- Su interfaz es algo parecido a lo siguiente:

```
class OperacionesBDR
{
    public ResultSet getDatosEspecificacionDelProducto(OID oid){...}
    public ResultSet getDatosVenta(OID oid) {...}
    ...
}
```

De manera que, por ejemplo, un conversor tiene código como éste:

```
class ConversorBDREspecificacionDelProducto extends ConversorPersistenciaAbstracto
{
    protected Object getObjectoDelAlmacenamiento(OID oid)
    {
        ResultSet rs = OperacionesBDR.getInstancia().getDatosEspecificacionDelProducto(oid);

        EspecificacionDelProducto ep = new EspecificacionDelProducto();
        ep.setPrecio(rs.getDouble("PRECIO"));
        ep.setOID(oid);
        return ep;
    }
}
```

A partir de esta Fabricación Pura se obtienen los siguientes beneficios:

- Se facilita el mantenimiento y rendimiento que es ajustado por un experto. La optimización del SQL requiere un programador SQL, en lugar de un programador de objetos. Con todo el SQL embebido en esta única clase, facilita al programador en SQL encontrarlo y trabajar con él.
- Encapsulación de los métodos y detalles de acceso. Por ejemplo, SQL construido mediante código podría sustituirse por una llamada a un procedimiento almacenado en la BDR para obtener los datos. O se podría insertar un enfoque más sofisticado basado en los **metadatos** para generar el SQL, en el que se genera el SQL de manera dinámica a partir de la descripción del esquema de los metadatos que se lee de una fuente externa.

Como arquitecto, el aspecto interesante de esta decisión de diseño es que en ella influyen las habilidades del desarrollador. Se llegó a un compromiso entre alta cohesión y la comodidad de un especialista. No todas las decisiones de diseño están motivadas por cuestiones de ingeniería del software “puras” como el acoplamiento y la cohesión.

34.16. Estados transaccionales y el patrón Estado

Las cuestiones relacionadas con el soporte de las transacciones pueden complicarse, pero para mantener las cosas simples de momento —para centrarnos en el patrón GoF Estado— asuma lo siguiente:

- Los objetos persistentes pueden insertarse, eliminarse o modificarse.
- Operar sobre un objeto persistente (por ejemplo, modificarlo) no provoca una actualización inmediata de la base de datos; más bien se debe ejecutar una operación *commit* explícita.

Además, la respuesta a una operación depende del estado de la transacción del objeto. Como ejemplo, las respuestas se podrían mostrar en la máquina de estados de la Figura 34.12.

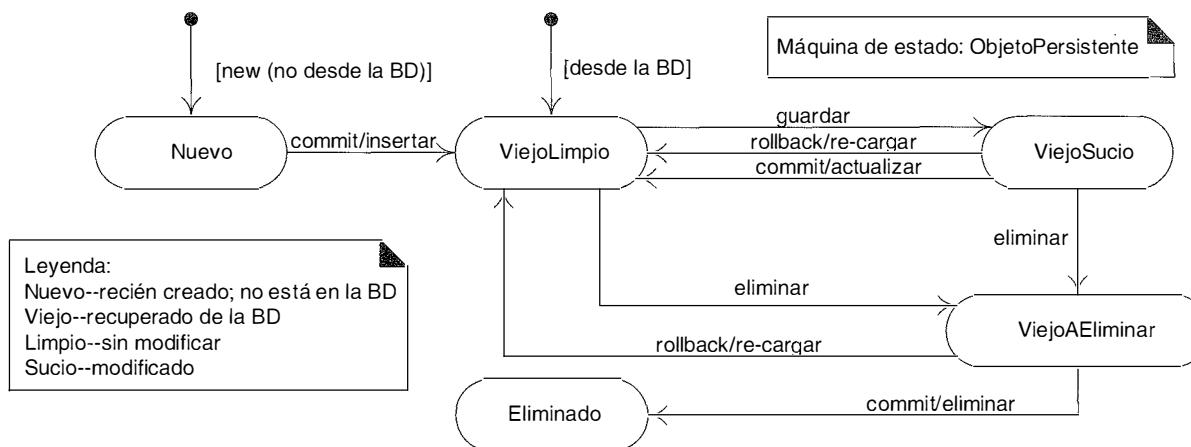


Figura 34.12. Máquina de estados para *ObjetoPersistente*.

Por ejemplo, un objeto “viejo sucio” (*old dirty*) es aquél recuperado de la base de datos y modificado después. En una operación *commit*, debería actualizarse en la base de datos —a diferencia del que se encuentra en el estado “viejo limpio”, con el que no debería hacer nada (porque no ha cambiado)—. En el FWP orientado a objetos, cuando se ejecuta una operación *eliminar* o *guardar*, no origina que se elimine o se guarde inmediatamente en la base de datos; sino, las transiciones de los objetos persistentes al estado apropiado, esperando una operación *commit* o *rollback* para realmente hacer algo.

Como comentario UML, esto es un buen ejemplo de donde es útil una máquina de estados para transmitir información concisa que de otra manera sería difícil de expresar.

En este diseño, asuma que haremos que todas las clases de objetos persistentes extenderán la clase *ObjetoPersistente*⁷, que proporciona los servicios técnicos comunes para la persistencia⁸. Por ejemplo, véase la Figura 34.13.

⁷ [Ambler00b] es una buena referencia acerca de la clase *ObjetoPersistente* y las capas de persistencia, aunque la idea es más antigua.

⁸ Algunas cuestiones relacionadas con la extensión de la clase *ObjetoPersistente* se discuten más tarde. Siempre que una clase de objetos del dominio extienda una clase de los servicios técnicos, se debería hacer una pausa para reflexionar, ya que se está mezclando intereses de la arquitectura (persistencia y lógica de la aplicación).

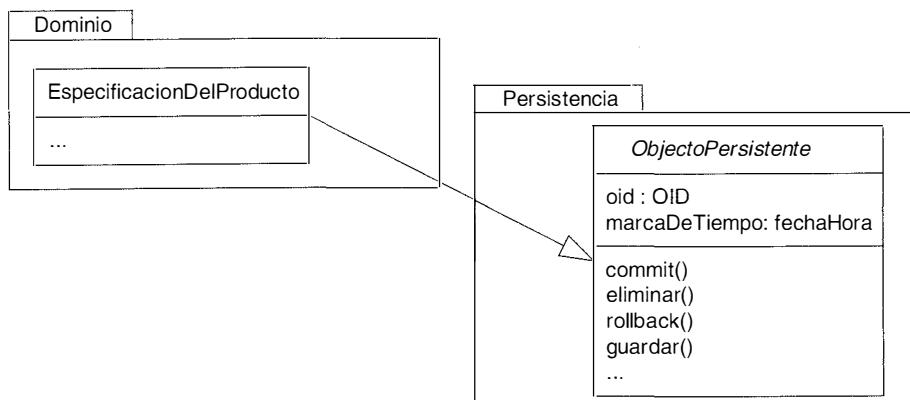


Figura 34.13. Objetos persistentes.

Ahora —y ésta es la cuestión que se resolverá con el patrón Estado— observe que los métodos *commit* y *rollback* requieren una estructura similar de lógica de casos, basada en el código del estado de la transacción. *commit* y *rollback* ejecutan diferentes acciones en cada caso, pero tienen estructuras lógicas similares.

<pre> public void commit() { switch(estado) { case VIEJO_SUCIO: //... break; case VIEJO_LIMPIO: //... break; ... } } </pre>	<pre> public void rollback() { switch(estado) { case VIEJO_SUCIO: //... break; case VIEJO_LIMPIO: //... break; ... } } </pre>
---	---

Una alternativa a esta estructura lógica de casos que se repite es el patrón GoF Estado (*State*).

Estado (State)

Contexto/Problema

El comportamiento de un objeto depende de su estado, y sus métodos contienen la lógica de casos que reflejan las acciones condicionales dependientes del estado. ¿Existe una alternativa a la lógica condicional?

Solución

Cree clases estado para cada estado, que implementan una interfaz común. En lugar de definir en el objeto de contexto las operaciones que dependen del estado, deléguelas en su objeto del estado actual. Asegure que el objeto de contexto siempre refuerce al objeto estado que refleja su estado actual.

La Figura 34.14 ilustra su aplicación en el subsistema de persistencia.

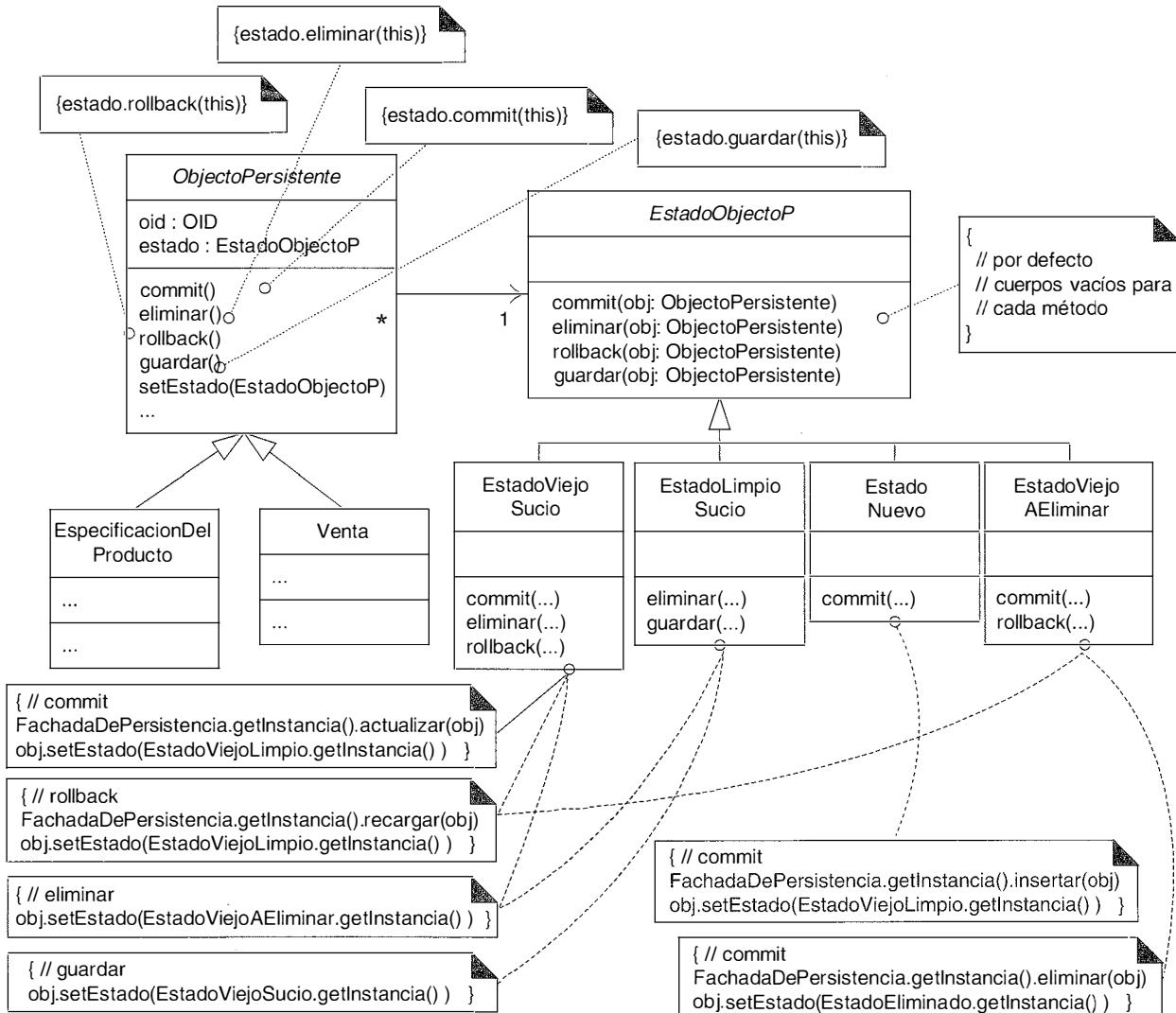


Figura 34.14. Aplicación del patrón Estado⁹.

Los métodos dependientes del estado del *ObjetoPersistente* delegan su ejecución a un objeto estado asociado. Si el objeto de contexto referencia a *EstadoViejoSucio* entonces 1) el método *commit* provocará una actualización de la base de datos, y 2) el objeto de contexto pasará a referenciar a *EstadoViejoLimpio*. Por otro lado, si el objeto de contexto está referenciando a *EstadoViejoLimpio*, se ejecuta el método *commit* heredado como que no hace nada y no hace nada (como se esperaba, puesto que el objeto está limpio).

Observe en la Figura 34.14 que las clases estado y su comportamiento se corresponden con la máquina de estados de la Figura 34.12. El patrón Estado es un mecanismo

⁹ La clase *Borrado* se ha omitido por restricciones de espacio en el diagrama.

para implementar un modelo de transición de estados en software¹⁰. Esto da lugar a que se produzca una transición de un objeto a estados diferentes en respuesta a los eventos.

Un comentario acerca del rendimiento, estos objetos estado no tienen —irónicamente— estado (sin atributos). Por tanto, no es necesario que haya múltiples instancias de una clase —cada una es un singleton—. Miles de objetos persistentes pueden referenciar a la misma instancia *EstadoViejoSucio*, por ejemplo.

34.17. Diseño de una transacción con el Patrón Command

La última sección consideró una vista simplificada de las transacciones. Esta sección amplía la discusión, pero no cubre todas las cuestiones relacionadas con el diseño de las transacciones. Informalmente, una transacción es una unidad de trabajo —un conjunto de tareas— cuyas tareas deben completarse todas con éxito, o no se debe completar ninguna. Es decir, la terminación es atómica.

En cuanto a los servicios de persistencia, las tareas de una transacción incluyen la inserción, actualización y eliminación de los objetos. Una transacción podría contener dos inserciones, una actualización y tres eliminaciones, por ejemplo. Para representar esto, se añade una clase *Transaccion* [Ambler00b]¹¹. Como se señala en [Fowler01], el orden de las tareas de la base de datos dentro de una transacción puede influir en su éxito (y rendimiento).

Por ejemplo:

1. Suponga que la base de datos tiene una restricción de integridad referencial de manera que cuando se actualiza un registro en la TablaA que contiene una clave ajena a un registro de la TablaB, la base de datos requiere que el registro de la TablaB ya exista.
2. Suponga que una transacción contiene una tarea INSERT para añadir el registro de la TablaB, y una tarea UPDATE para actualizar el registro de la TablaA. Si se ejecuta UPDATE antes de INSERT, surge un error de integridad referencial.

La ordenación de las tareas de la base de datos puede ayudar. Algunas cuestiones de la ordenación son específicas del esquema, pero una estrategia general es primero realizar las inserciones, luego las actualizaciones y entonces las eliminaciones.

Tenga en cuenta que el orden en el que una aplicación añade las tareas a una transacción podría no reflejar el mejor orden de ejecución. Las tareas necesitan que se ordenen justo antes de su ejecución.

Esto nos lleva a otro patrón GoF: Command¹².

¹⁰ Existen otros, entre los que se encuentran lógica condicional construida mediante código, intérpretes de máquinas de estados, y generadores de código dirigidos por tablas de estados.

¹¹ Se denomina UnidadDeTrabajo en [Fowler01].

¹² *N. del T.* No se ha traducido por el mismo motivo que en el caso del patrón Singleton.

BIBLIOGRAFÍA

Contexto/Problema
¿Cómo gestionar las solicitudes o tareas que necesitan funciones como ordenar (estableciendo prioridades), poner en cola, retrasar, anotar en registro o deshacer?

Solución
Defina una clase por cada tarea que implemente una interfaz común.

Éste es un patrón sencillo con muchas aplicaciones útiles; las acciones se convierten en objetos, y de esta manera se pueden ordenar, anotar en un registro, poner en cola, etcétera. Por ejemplo, en el FWP, la Figura 34.15 muestra las clases Command (o tareas) para las operaciones de la base de datos.

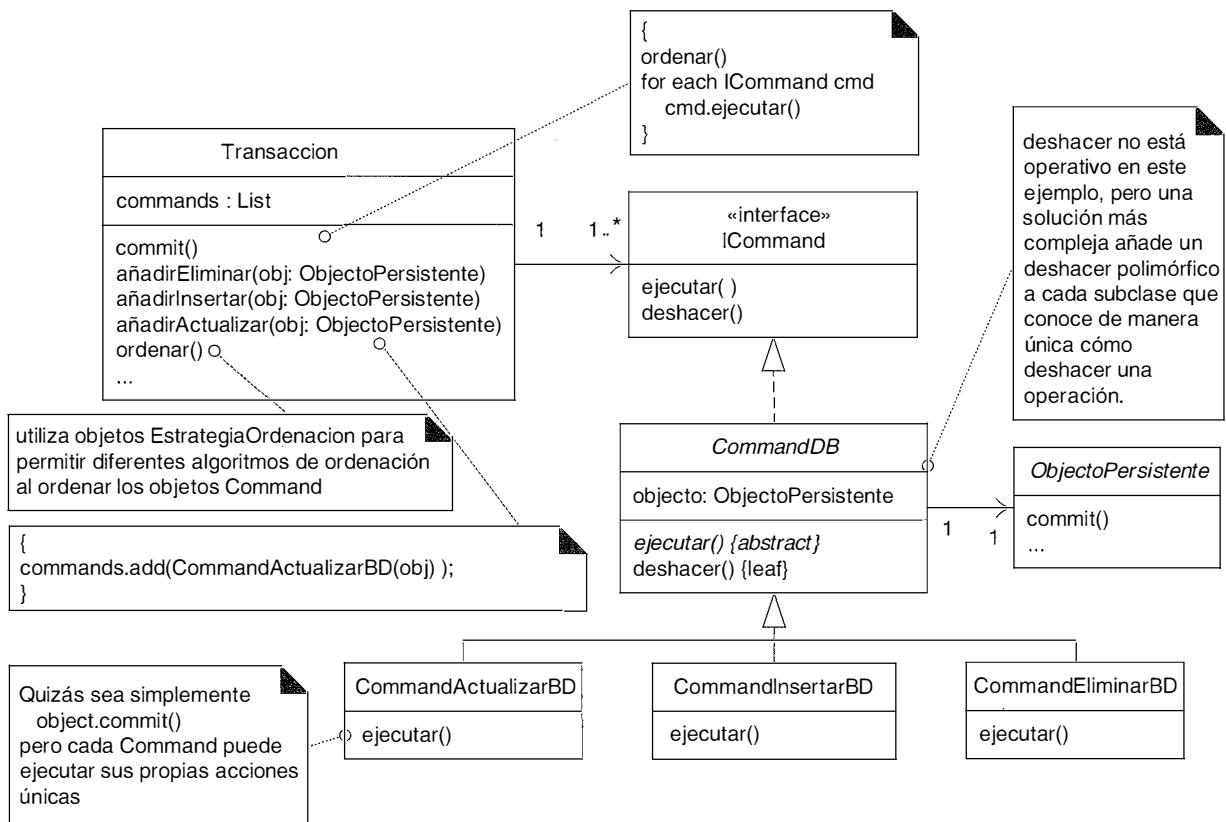


Figura 34.15. Clases Command para las operaciones de la base de datos.

Hay mucho más para completar una solución de transacción, pero la idea clave de esta sección es representar cada tarea o acción de la transacción como un objeto con un método *ejecutar* polimórfico; esto hace accesible un mundo de flexibilidad tratando la respuesta como un objeto en sí.

El ejemplo prototípico de Command es el de las acciones de una GUI, como cortar y pegar. Por ejemplo, el método *ejecutar* de *CommandCortar* realiza la acción cortar, y su

método *deshacer* deshace el corte. *CommandCortar* también retendrá los datos necesarios para llevar a cabo la acción de deshacer. Todos los objetos Command de la GUI se pueden mantener en una pila que recoge la historia de las ejecuciones, de manera que se puedan desapilar por turnos, y deshacerse cada una.

Otro uso típico del patrón Command es para la gestión de las peticiones del lado del servidor. Cuando un objeto servidor recibe un mensaje (remoto), crea un objeto Command para esta petición, y lo entrega al *CommandProcesador* [BMRSS96], que puede poner en cola, anotar en un registro, priorizar y ejecutar los objetos Command.

34.18. Materialización perezosa con un Proxy Virtual

Algunas veces es conveniente diferir la materialización de los objetos hasta que sea absolutamente necesario, normalmente por razones de rendimiento. Por ejemplo, suponga que los objetos *EspecificacionDelProducto* referencian a un objeto *Fabricante*, pero rara vez es necesario que se materialice desde la base de datos. Sólo provocan una petición de la información del fabricante escenarios poco frecuentes, como los escenarios relacionados con las rebajas de los fabricantes en el que se necesita el nombre y la dirección de la compañía.

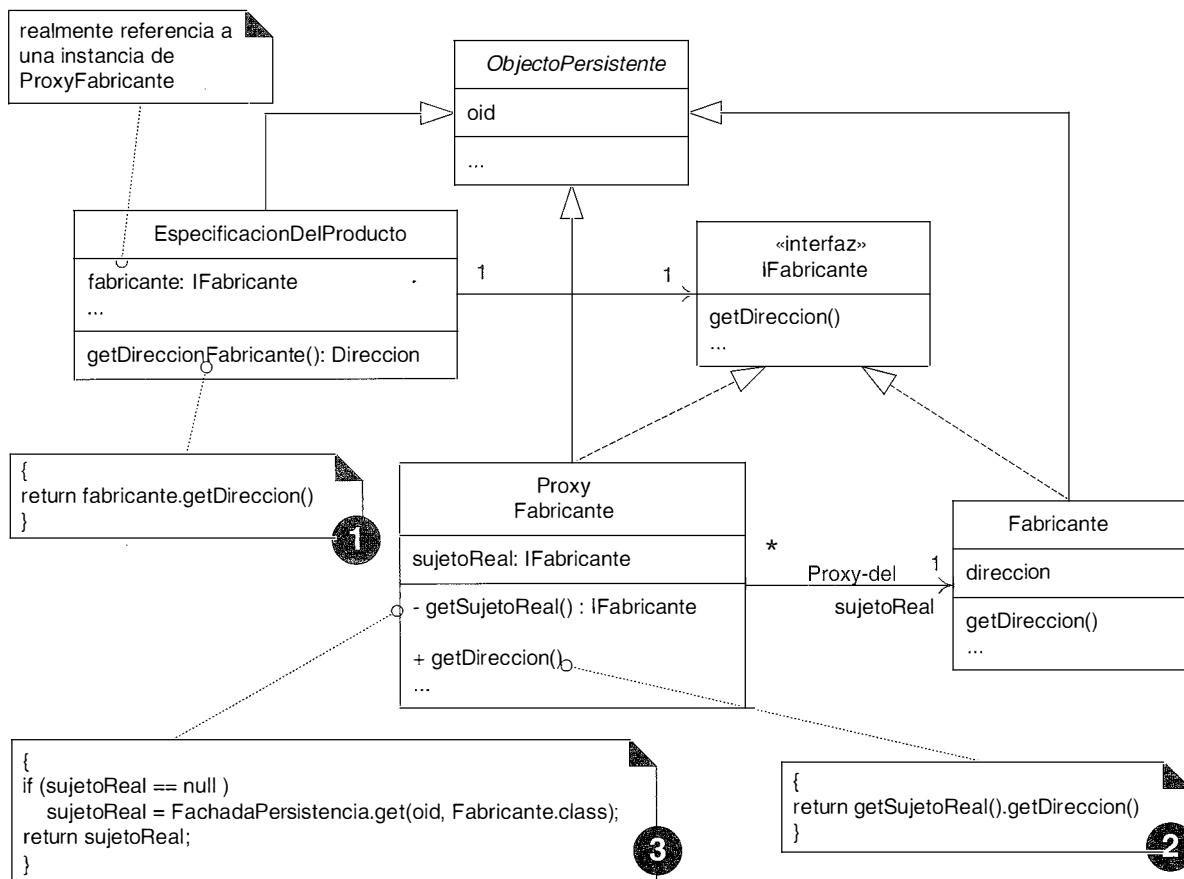


Figura 34.16. Proxy Virtual del Fabricante.

La materialización diferida de los objetos “hijos” se conoce como **materialización perezosa**. La materialización perezosa se puede implementar utilizando el patrón GoF Proxy Virtual —una de las muchas variaciones del Proxy—.

Un **Proxy Virtual** es un proxy para otro objeto (el *sujeto al que se quiere acceder realmente*) que materializa el sujeto real cuando se referencia por primera vez; por tanto, implementa la materialización perezosa. Es un objeto ligero que simboliza el objeto “real” que puede o no ser materializado.

Un ejemplo concreto del patrón Proxy Virtual con la *EspecificacionDelProducto* y el *Fabricante* se muestra en la Figura 34.16. Este diseño se basa en la suposición de que los proxies conocen el OID de sus sujetos reales, y cuando se requiere la materialización, se utiliza el OID para ayudar a identificar y recuperar el sujeto real.

Nótese que la *EspecificacionDelProducto* tiene visibilidad de atributo de la instancia *IFabricante*. El *Fabricante* para esta *EspecificacionDelProducto* podría no haberse materializado todavía en memoria. Cuando la *EspecificacionDelProducto* envía un mensaje *getDireccion* al *ProxyFabricante* (creyendo que es el objeto fabricante materializado), el proxy materializa el *Fabricante* real, utilizando el OID del *Fabricante* para recuperarlo y materializarlo.

Implementación de un Proxy Virtual

La implementación de un Proxy Virtual varía según el lenguaje. Los detalles quedan fuera del alcance de este capítulo, pero a continuación presentamos un resumen:

Lenguaje	Implementación del Proxy Virtual
C++	Define una clase plantilla de puntero inteligente (<i>smart pointer</i>). Realmente no se necesita la definición de la interfaz <i>IFabricante</i> .
Java	Se implementa la clase <i>ProxyFabricante</i> . Se define la interfaz <i>IFabricante</i> . Sin embargo, normalmente no se codifican manualmente. Más bien, uno crea un generador de código que analiza las clases a las que se quiere acceder realmente (ej. <i>Fabricante</i>) y genera <i>IFabricante</i> y <i>ProxyFabricante</i> . Otra alternativa de Java es utilizar el API Proxy Dinámico.
Smalltalk	Definir un Proxy Morphing Virtual (o Proxy Fantasma), que utiliza <i>#doesNotUnderstand:</i> y <i>#become:</i> para transformar en el sujeto real. No se necesita la definición de <i>IFabricante</i> .

¿Quién crea el Proxy Virtual?

Observe en la Figura 34.16 que el *ProxyFabricante* colabora con la *FachadaDePersistencia* para materializar su sujeto real. ¿Pero quién crea el *ProxyFabricante*? Respuesta: la clase que establece la correspondencia entre la *EspecificacionDelProducto* y la base de datos. Esta clase es la responsable de decidir, cuándo se materializa un objeto, cuál de los objetos “hijos” debería también materializarse de manera impaciente, y cuáles deberían materializarse de manera perezosa con un proxy.

Considere estas soluciones alternativas: una utiliza materialización impaciente, y la otra materialización perezosa.

```

//MATERIALIZACIÓN IMPACIENTE DEL FABRICANTE

class ConversorBDREspecificacionDelProducto extends ConversorPersistenciaAbstracto
{
protected Object getObjetoDelAlmacenamiento(OID oid)
{
ResultSet rs =
    OperacionesBDR.getInstancia().getDatosEspecificacionDelProducto(oid);

EspecificacionDelProducto ep = new EspecificacionDelProducto();
ep.setPrecio(rs.getDouble("PRECIO"));

//aquí está la esencia

String claveAjenaFabricante = rs.getString("FAB_OID");
OID oidFab = new OID(claveAjenaFabricante);
ep.setFabricante((IFabricante)
    FachadaDePersistencia.getInstancia().get(oidFab, Fabricante.class));
...
}

```

A continuación presentamos la solución de materialización perezosa:

```

//MATERIALIZACIÓN PEREZOSA DEL FABRICANTE

class ConversorBDREspecificacionDelProducto extends ConversorPersistenciaAbstracto
{
protected Object getObjetoDelAlmacenamiento(OID oid)
{
ResultSet rs =
    OperacionesBDR.getInstancia().getDatosEspecificacionDelProducto(oid);

EspecificacionDelProducto ep = new EspecificacionDelProducto();
ep.setPrecio(rs.getDouble("PRECIO"));

//aquí está la esencia

String claveAjenaFabricante = rs.getString("FAB_OID");
OID oidFab = new OID(claveAjenaFabricante);
ep.setFabricante(new ProxyFabricante(oidFab));
...
}

```

34.19. Cómo representar las relaciones en tablas

El código de la sección anterior confía en la clave ajena FAB_OID en la tabla ESPEC_PRODUCTO para conectar con un registro de la tabla FÁBRICANTE. Esto pone de relieve la pregunta: ¿cómo se representan las relaciones de los objetos en el modelo relacional?

La respuesta se da en el patrón **Representación de las Relaciones de los Objetos en Tablas** [BW96], que propone lo siguiente:

- Asociaciones **uno-a-uno**
 - Colocar una clave ajena OID en una o ambas tablas que representan los objetos de la relación.
 - O, crear una tabla asociación que recoja los OIDs de cada uno de los objetos de la relación.
- Asociaciones **uno-a-muchos**, como una colección
 - Crear una tabla asociativa que registre los OIDs de cada uno de los objetos de la asociación.
- Asociaciones **muchos-a-muchos**
 - Crear una tabla asociativa que registre los OIDs de cada uno de los objetos de la asociación.

34.20. Superclase ObjetoPersistente y separación de intereses

Una solución de diseño parcial típica para proporcionar persistencia a los objetos es crear una clase abstracta de servicios técnicos *ObjetoPersistente* de la que heredan todos los objetos persistentes (ver Figura 34.17). Tal clase, normalmente define atributos para la persistencia, como un OID único, y métodos para almacenarlos en la base de datos.

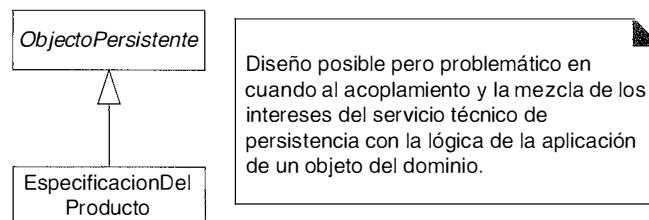


Figura 34.17. Problemas con la superclase ObjetoPersistente.

Esto no es incorrecto, pero adolece de la debilidad del acoplamiento entre la clase y la clase *ObjetoPersistente* —las clases del dominio terminan extendiendo una clase de los servicios técnicos—.

Este diseño no ilustra una clara separación de intereses. Más bien, se mezclan los intereses de los servicios técnicos con los intereses de la lógica del negocio de la capa del dominio en virtud de su extensión.

Por otro lado, la “separación de intereses” no es una virtud absoluta que se debe seguir a cualquier precio. Como se discutió cuando se presentaron las Variaciones Protegidas, los diseñadores necesitan escoger sus batallas en los puntos en los que realmente sea probable una inestabilidad costosa. Si en una aplicación particular, haciendo que la clase sea una subclase de *ObjetoPersistente* nos lleva a una solución ordenada y fácil y no da lugar a problemas de diseño o mantenimiento a largo plazo, ¿por qué no? La respuesta está en entender la evolución de los requisitos y el diseño de la aplicación. También influye el lenguaje: aquéllos con herencia simple (como Java) *consumen* su única preciada superclase.

34.21. Cuestiones sin resolver

Esto ha sido una breve introducción a los problemas y soluciones de diseño de un framework y servicio de persistencia. Se han encubierto muchas cuestiones importantes, entre las que se encuentran:

- Desmaterialización de los objetos.
 - Brevemente, los conversores deben definir los métodos *putObjetoEnAlmacenamiento*. La desmaterialización de jerarquías de composición requiere la colaboración entre múltiples conversores y el mantenimiento de tablas asociativas (si se utiliza un BDR).
- Materialización y desmaterialización de las colecciones.
- Consultas a grupos de objetos.
- Gestión completa de transacciones.
- Gestión de los errores cuando falla una operación de la base de datos.
- Acceso multiusuario y estrategias de bloqueo.
- Seguridad—control del acceso a la base de datos.

Parte 6

TEMAS ESPECIALES

Capítulo 35

SOBRE EL DIBUJO DE DIAGRAMAS Y LAS HERRAMIENTAS

Las burbujas no explotan.

Bertrand Meyer

Objetivos

- Aprender consejos para dibujar los diagramas UML en un proyecto.
 - Ilustrar algunas funciones comunes de las herramientas CASE para UML.
-

Introducción

En un proyecto real, la realización del análisis y diseño mientras se dibujan los diagramas UML no ocurre ordenadamente como en las páginas de un libro. Tiene lugar en el contexto de un equipo de desarrollo de software ocupado, que trabaja en oficinas o habitaciones, hacen garabatos en pizarras y quizás utilizando una herramienta, y a menudo con tendencia a querer comenzar a programar en lugar de trabajar con detalle algunos detalles mediante la realización de diagramas. Si la herramienta para UML, o el proceso para dibujar los diagramas, resultan molestos y engorrosos, o parece que tiene menos valor que la programación, se evitarán.

Este capítulo propone mantener un equilibrio entre la programación y el dibujo de diagramas, y fomentar un entorno de soporte para hacer que la tarea de dibujar los diagramas sea cómoda y útil en lugar de difícil.

35.1. Diseño especulativo y razonamiento visual

Los diseños que se ilustran mediante diagramas UML serán incompletos, y sólo servirán como “trampolín” para la programación. Demasiados diagramas antes de programar nos lleva a una pérdida de tiempo en direcciones de diseño especulativas, o una pérdida de

tiempo obsesionados con herramientas para UML. No hay nada como el código real para decirte qué funciona. Bertrand Meyer lo dice mejor: “Las burbujas no explotan”.

No obstante, promuevo enérgicamente que se anticipa *alguna* idea mediante la realización de diagramas antes de programar, y se que puede ser útil, especialmente para explorar las estrategias de diseño más importantes. La pregunta interesante es “¿cuánto tiempo se tiene que dedicar a dibujar los diagramas antes de programar?” En parte la respuesta está en función de la experiencia y estilo cognitivo de los diseñadores.

Algunas personas son buenas para el razonamiento espacial/visual y expresar sus pensamientos acerca del diseño del software en un lenguaje visual complementa su naturaleza; otros no. Un gran porcentaje del cerebro está dedicado al razonamiento y procesamiento visual o simbólico, en lugar de al procesamiento textual (código). Los lenguajes visuales como UML juegan con una capacidad natural de la mente en la mayoría de la gente. A aquellos a los que se les ha enseñado UML obviamente les resultará más sencillo que a los que no. Y, en general, los diseñadores de objetos con más experiencia pueden diseñar de manera efectiva dibujando sin perderse en especulaciones no realistas, debido a su experiencia y juicio. Aplicados por expertos, los diagramas pueden ayudar a un grupo a moverse más rápidamente hacia un diseño apropiado, debido a la capacidad de ignorar los detalles y centrarse en los verdaderos problemas.

Una excepción a esta sugerencia de dibujar los diagramas “ligeros” son los sistemas que se modelan de manera natural como máquinas de estados. Hay algunas herramientas CASE que pueden hacer un trabajo impresionante generando código completo basado en máquinas de estados UML detalladas para todas las clases. Pero no todos los dominios encajan de manera natural en un enfoque fuertemente centrado en las máquinas de estados; por ejemplo, las máquinas de control y telecomunicaciones a menudo se ajustan bien, los sistemas de información de gestión normalmente no.

35.2. Sugerencias para dibujar los diagramas de UML en el proceso de desarrollo

Nivel de esfuerzo

Como guía, considere realizar los diagramas en parejas durante los siguientes períodos, antes de programar de manera seria en la iteración.

Iteración de 2-semanas	De medio día a un día casi al comienzo de la iteración (ej. lunes o martes)
Iteración de 4-semanas	Uno o dos días cerca del comienzo

En ambos casos, la realización de los diagramas no tiene que parar después de este esfuerzo inicial enfocado a este fin, los desarrolladores podrían dirigirse —idealmente en parejas— “a la pizarra” durante sesiones cortas para esbozar ideas antes de programar más. Y podrían realizar otra sesión más larga de medio día a mitad de la iteración, cuando tropiecen con un problema complejo en el ámbito de su tarea inicial, o terminen su primera tarea y pasen a la segunda.

Otras sugerencias

- Dibuje en parejas, no a solas. Lo más importante, la sinergia nos conduce a diseños mejores. En segundo lugar, la pareja aprende rápidamente técnicas de diseño el uno del otro y, por tanto, ambos llegan a ser mejores diseñadores. Es difícil mejorar como diseñador de software cuando uno diseña individualmente. Cambie de manera regular el compañero de dibujo/diseño para estar periodos de tiempo amplios expuestos al conocimiento de los demás.
- Para aclarar un punto al que se ha hecho alusión varias veces, en los procesos iterativos (como el UP), los programadores son también los diseñadores; no hay un equipo separado que dibuja los diseños y se los entrega a los programadores. Los desarrolladores se ponen su sombrero de UML, y dibujan un poco. Entonces se ponen su sombrero de programador e implementan, y continúan diseñando mientras programan.
- Si hay diez desarrolladores, suponga que hay cinco equipos de dibujo trabajando durante un día en pizarras diferentes. Si el arquitecto dedica tiempo a rotar por los cinco equipos, llegará a ver puntos de dependencia, conflictos e ideas de un equipo que sirven para otro. El arquitecto entonces puede actuar de enlace para llegar a una armonía entre los diseños y clarificar las dependencias.
- Contrate un escritor técnico para el proyecto y enséñele algo de la notación UML y los conceptos básicos del A/DOO (de manera que pueda entender el contexto). Dispóngase de la ayuda del escritor en la realización del “trabajo engorroso” con las herramientas CASE para UML, en los procesos de ingeniería inversa generando los diagramas a partir del código, en la impresión y presentación de grandes diagramas impresos con plotter, etcétera. Los desarrolladores dedicarán su tiempo (más caro) a hacer lo que hacen mejor: idear diseños y programar. Un escritor técnico les ayuda realizando la gestión de los diagramas, además de las verdaderas responsabilidades de la escritura técnica tales como el trabajo en el documento para el usuario final. Esto se conoce como el patrón Analista Mercenario [Coplien95a].
- Organice el área de desarrollo con muchas pizarras amplias distribuidas muy cerca unas de otras.
- Generalizando, maximice el entorno de trabajo para dibujar cómodamente en las paredes. Cree un entorno “amigable” para dibujar y colgar los diagramas. No puede esperar que se consiga una cultura de modelado visual con éxito en un entorno donde los desarrolladores se están peleando por dibujar en pizarras pequeñas de 60x90 cm, monitores de ordenador de tamaño corriente o trozos de papel. Para dibujar de manera cómoda hacen falta espacios muy amplios y abiertos —físicos o virtuales—.
- Como accesorio de las pizarras, utilice finas hojas blancas de plástico con “adherencia estática” (viene en paquetes de 20 o más) que se pueden colocar en las paredes; éstas se encuentran disponibles en muchas papelerías. Las hojas permanecen unidas a la pared mediante adherencia estática, y se puede utilizar como una pizarra con un rotulador que se pueda borrar. Se puede empapelar una pared con estas hojas para crear “pizarras” temporales, masivas. He dirigido grupos donde

hemos empapelado todas las paredes —de arriba abajo— de la sala del proyecto con estas hojas, y lo encontramos de gran ayuda para la comunicación.

- Si utiliza una pizarra para los dibujos de UML, utilice un dispositivo (existe al menos uno en el mercado) que capture los dibujos hechos a mano y los transmita a un ordenador como un fichero gráfico. Un diseño involucra una parte receptora en la esquina de la pizarra que captura la imagen para enviarla al ordenador y unas fundas especiales transmisoras en las que se insertan los rotuladores.
- De manera alternativa, si utiliza una pizarra para los dibujos de UML, utilice una cámara digital para capturar las imágenes, normalmente en dos o tres secciones. Ésta es una práctica para dibujar los diagramas, bastante común y efectiva.
- Otra tecnología de pizarra es una pizarra “que imprime”, que normalmente es una pizarra de dos lados con un escáner y una impresora conectada. También son útiles.
- Imprima las imágenes UML dibujadas a mano (capturadas mediante una cámara o un dispositivo de pizarra) y cuélguelas de manera visible *muy* cerca de las estaciones de trabajo para la programación. Lo importante de los diagramas es inspirar la dirección de la programación, de manera que los programadores puedan echarles un vistazo mientras están programando. Si se dibujan pero “se entierran”, no tenía mucho sentido dibujarlos.
- Si dibuja los diagramas UML a mano, utilice una notación simple elegida para acelerar y facilitar la realización de los diagramas.
- Incluso si lleva a cabo el diseño creativo en una pizarra, utilice una herramienta CASE para UML para generar los diagramas de paquetes y de clases mediante un proceso de ingeniería inversa a partir del código fuente (de la última iteración) por lo menos al comienzo de cada iteración posterior. Entonces, utilice estos diagramas generados mediante un proceso de ingeniería inversa como punto de partida para el siguiente diseño creativo.
- Imprima periódicamente los diagramas de paquetes y de clases interesantes/inestables/difíciles, generados recientemente mediante un proceso de ingeniería inversa, en un tamaño ampliado (para facilitar que se vea) con un plotter que pueda imprimir en papel continuo de 90 ó 120 cm de ancho. Cuélguelos en las paredes muy cerca de los desarrolladores como ayuda visual. El escritor técnico, si está presente, puede hacer este trabajo. Anime a los desarrolladores a que dibujen y hagan garabatos sobre los diagramas durante el trabajo de diseño creativo.
- Con respecto a la ingeniería inversa, pocas herramientas para UML soportan el proceso de ingeniería inversa para generar diagramas de secuencia —no únicamente diagramas de clases— a partir del código fuente. Si está disponible, utilice una para generar los diagramas de secuencia para los escenarios significativos para la arquitectura, imprimálos a tamaño grande con el plotter, y cuélguelos para facilitar que se vean.
- Si utiliza una herramienta CASE para UML (de hecho, hágalo para todo el trabajo de programación), utilice una estación de trabajo con un monitor dual (dos monitores de pantalla plana de tamaño ordinario son más baratas que un único monitor

de pantalla plana de tamaño grande). Los sistemas operativos modernos soportan (al menos) tarjetas de vídeo duales y de esta manera dos monitores. Organice sus ventanas dentro de la herramienta para UML entre los dos monitores. ¿Por qué? Un monitor pequeño reprime psicológicamente y creativamente en lo que se refiere a los dibujos y lenguajes visuales porque el espacio del área visual es demasiado pequeño y estrecho. Un desarrollador puede caer en la aptitud desmotivada de “el diseño ha terminado porque la ventana está llena, y parece demasiado desordenado”.

- Cuando utilice una herramienta CASE para UML y realice el diseño creativo por parejas o pequeños grupos, conecte dos proyectores de ordenador a las dos tarjetas de vídeo del ordenador y alinee las proyecciones en la pared de manera que el equipo pueda ver y trabajar con un espacio amplio para el área visual. Un área pequeña y unos diagramas difíciles de ver son impedimentos psicológicos y sociales para el diseño visual en colaboración de los pequeños grupos.

35.3. Herramientas y características de ejemplo

Este libro es imparcial respecto a la herramienta

Sería algo extraño no mencionar ninguna herramienta CASE (*Computer Aid Software Engineering*, ingeniería del software asistida por ordenador) para UML, porque este libro trata en parte de la realización de dibujos en UML, que tiene lugar con una herramienta CASE, o en una pizarra. Al mismo tiempo, no se pueden cubrir por igual todas las herramientas, y las evaluaciones apropiadas quedan fuera del alcance de este libro. Para ser imparcial:

Este libro no avala ninguna herramienta CASE para UML. Los siguientes ejemplos sólo sirven para ilustrar algunas de las características típicas y claves que podemos encontrar en las herramientas CASE para UML.

Las herramientas se ajustan de manera inconsistente a UML

Pocas herramientas dibujan toda la notación de UML correctamente, y conforme con la versión actual de la especificación de UML —realmente con cualquier versión—. Aunque esto estaría bien, no debería ser un factor en la elección de una herramienta, puesto que es mucho más importante su funcionalidad y facilidad de uso.

Ejemplo uno

En las Figuras 35.1 y 35.2 se utiliza Together de TogetherSoft para ilustrar y definir dos funciones claves de una herramienta CASE para UML: **ingeniería directa e inversa**. Estas funciones son lo que distinguen esencialmente una herramienta CASE para UML de una herramienta para dibujar.

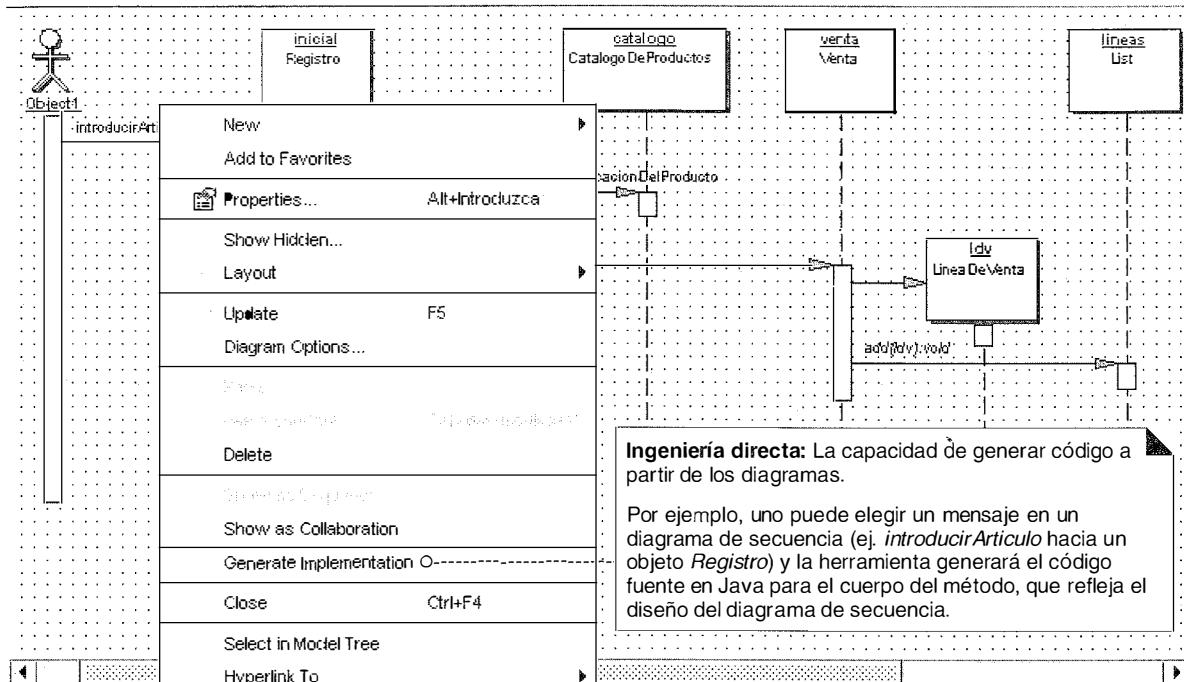


Figura 35.1. Ingeniería directa.

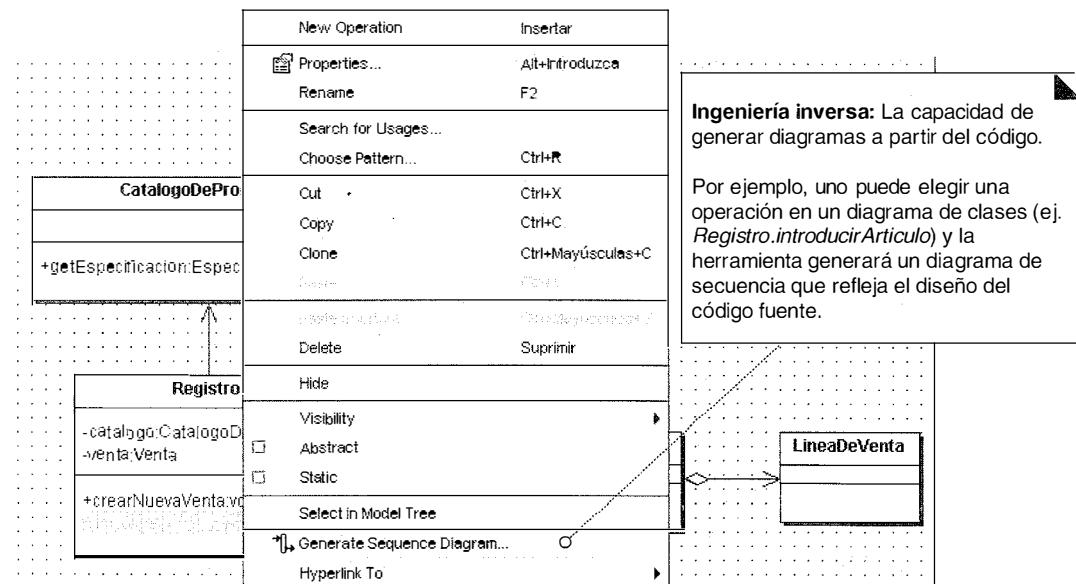


Figura 35.2. Ingeniería inversa.

35.4. Ejemplo dos

En las Figuras 35.3 y 35.4 se utiliza Rational Rose para ilustrar algunas otras funciones básicas de una herramienta CASE para UML.

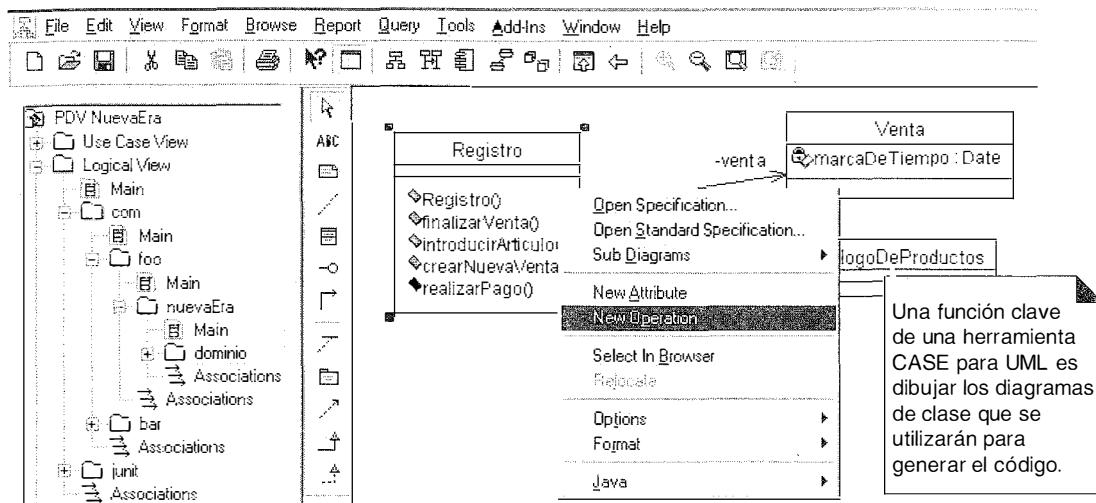


Figura 35.3. Creación de diagramas de clase.

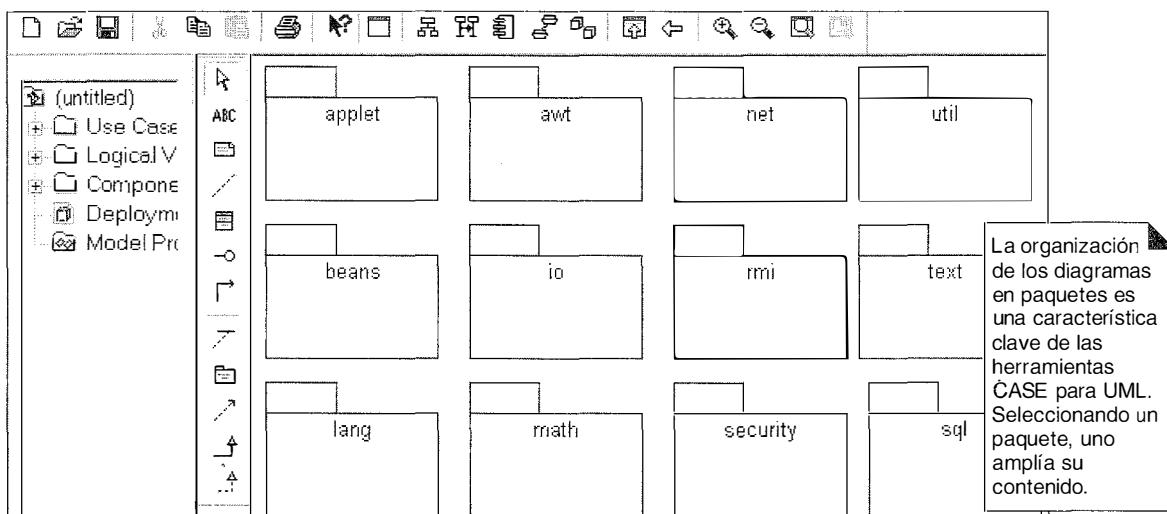


Figura 35.4. Gestión de paquetes.

Peticiones a los vendedores de herramientas CASE para UML

Sugiero que los consumidores hagan cuatro peticiones a los vendedores de herramientas CASE para UML:

1. Implementación correcta de la notación actual de UML en la herramienta.
2. Que el propio equipo de desarrollo de la herramienta CASE haya dibujado, leído y revisado seriamente diagramas UML (incluyendo el proceso de ingeniería inversa de los diagramas) en el proceso de construcción de la propia herramienta para UML.

3. Utilizar la versión N de la herramienta para UML para crear la versión N+1.
4. Proporcionar el soporte para los procesos de ingeniería directa e inversa de los diagramas de secuencia; la mayoría de las herramientas sólo lo soportan para el diagrama de clases.

Microsoft aboga por que los creadores de las herramientas “coman su propia comida para perros”. Buen consejo.

Capítulo 36

INTRODUCCIÓN A CUESTIONES RELACIONADAS CON LA PLANIFICACIÓN ITERATIVA Y EL PROYECTO

La predicción es muy difícil, especialmente si es acerca del futuro.

anónimo

Objetivos

- Priorizar los requisitos y los riesgos.
 - Comparar y contrastar la planificación adaptable y predictiva.
 - Definir el Plan de Fase y Plan de Iteración del UP.
 - Introducir las herramientas para mantener la traza de los requisitos para el desarrollo iterativo.
 - Sugerir el modo de organizar los artefactos del proyecto.
-

Introducción

Las cuestiones relacionadas con la gestión y planificación del proyecto son temas amplios, pero sirve de ayuda abordar un breve estudio sobre algunas preguntas claves relacionadas con el desarrollo iterativo y el UP, como:

- ¿Qué hacemos en la siguiente iteración?
- ¿Cómo mantener la traza de los requisitos en el desarrollo iterativo?
- ¿Cómo organizamos los artefactos del proyecto?

36.1. Priorización de los requisitos

Criterios que dirigen las primeras iteraciones: riesgo, cobertura, naturaleza crítica, desarrollo de habilidades

¿Qué hacemos en las primeras iteraciones? Organice los requisitos y las iteraciones de acuerdo con el riesgo, cobertura y naturaleza crítica [Kruchten00]. El riesgo del requisito comprende tanto la complejidad técnica como otros factores, como incertidumbre del esfuerzo, especificación pobre, problemas políticos, o facilidad de uso. Debemos diferenciar la priorización de los riesgos de los requisitos de la priorización de los riesgos del proyecto, que se estudiará en una sección posterior.

La cobertura implica que todas las partes importantes del sistema por lo menos se han tratado brevemente en las primeras iteraciones —quizás una implementación “en anchura y superficial” a través de muchos componentes—. La naturaleza crítica se refiere a las funciones de alto valor para el negocio; es decir, las funciones principales deberían tener al menos implementaciones parciales para los escenarios principales de éxito en las primeras iteraciones, incluso si no son técnicamente arriesgadas.

En algunos proyectos, otro criterio es el desarrollo de habilidades —un objetivo es ayudar al equipo a dominar nuevas habilidades como adoptar las tecnologías de objetos—. En tales proyectos, el desarrollo de habilidades es un factor de gran peso para establecer las prioridades, que tiende a reorganizar las iteraciones en requisitos de menos riesgo o más simples en las primeras iteraciones, motivado por el aprendizaje en lugar de por objetivos de reducción del riesgo.

¿Qué priorizamos?

El UP está dirigido por casos de uso, que incluye la práctica de priorizar los casos de uso (y escenarios de casos de uso) para la implementación. También se expresan algunos requisitos como características de alto nivel no relacionadas con un caso de uso específico, normalmente porque abarcan muchos casos de uso o son un servicio general, como los servicios de registro. Estas funciones que no pertenecen a ningún caso de uso se recogerán en la Especificación Complementaria. Por tanto, incluya en una lista de priorización tanto los casos de uso como las características de alto nivel.

Requisito	Tipo	...
Procesar Venta	CU	...
Registrar	Característica	...
...

Métodos cualitativos de grupo para la priorización

En base a los anteriores criterios, se priorizan los requisitos, y los de alta prioridad se manejan en las primeras iteraciones. La priorización podría ser informal y cualitativa, generada en una reunión de grupo por miembros conscientes de estos criterios.

Sugerencia

Para priorizar informalmente los requisitos, tareas, o riesgos por medio de una reunión de grupo, utilice la “votación con puntos” de manera iterativa. Liste los ítems en una pizarra. Cada uno coge, por ejemplo, 20 puntos adhesivos. Como grupo, y en silencio (para reducir la influencia), todos se aproximan a la pizarra y aplican los puntos a los ítems, reflejando las prioridades del que vota. En cuanto se termine, se ordenan y discuten. Entonces haga una segunda vuelta de votación con puntos silenciosa para reflejar las percepciones actualizadas en base a la votación de la primera vuelta y a la discusión. Esta segunda vuelta proporciona la retroalimentación y adaptación según la cual se mejoran las decisiones.

La priorización de los requisitos o el riesgo se hará antes de la iteración 1, pero se repetirá de nuevo antes de la iteración 2, y así sucesivamente.

Métodos cuantitativos para la priorización

La discusión de grupo y algo como la votación con puntos para priorizar los requisitos o el riesgo probablemente son suficientes —un enfoque cualitativo difuso—. Para los de mente más cuantitativa, se han utilizado variaciones sobre lo siguiente. Los valores y pesos del ejemplo son sólo sugerencias; lo importante es que los valores numéricos y los pesos se pueden utilizar para razonar acerca de las prioridades.

Requisito	Tipo	SA	Riesgo	Naturaleza crítica	Suma P.
Procesar Venta	CU	3	2	3	15
Registrar	Caract	3	0	1	7
Gestionar Devoluciones	CU	1	0	0	2
...

	Peso	Rango
SA:Significativo para la Arquitectura	2	0-3
Riesgo: tecnol. complejo, nuevo,...	3	0-3
Naturaleza crítica: valor alto negocio <i>inicial</i>	1	0-3

En cualquier proyecto, los valores exactos no deberían tomarse con demasiada seriedad; en cuanto se termine, se pueden utilizar las puntuaciones numéricas para ayudar a agrupar los requisitos en conjuntos difusos de prioridad alta, media y baja. Claramente, parece importante trabajar en el caso de uso *Procesar Venta* en las primeras iteraciones.

Los números no lo dicen todo. Aunque registrar es una característica simple, de bajo riesgo, es significativo para la arquitectura porque tiene que integrarse completamente en el código desde el principio. Sería difícil y disminuye la integridad de la arquitectura el añadirlo a posteriori.

Priorización de los requisitos del PDV NuevaEra

En base a algunos métodos para priorizar, es posible establecer grupos difusos de requisitos. En términos de los artefactos del UP, esta clasificación se recoge en el Plan de Desarrollo del Software del UP.

Prioridad	Requisitos (caso de uso o característica)	Comentario
Alta	Procesar Venta Registrar ...	Puntuación alta en todos los criterios de clasificación. Generalizado. Difícil de añadir tarde. ...
Media	Gestionar Usuarios Autenticar Usuarios ...	Influye en el subdominio de seguridad. Proceso importante pero no demasiado difícil. ...
Baja	Cerrar Caja Desconectar ...	Fácil; efecto mínimo en la arquitectura. ídem. ...

Los casos de uso “Poner en Marcha” y “Desconectar”

Prácticamente todos los sistemas tienen un caso de uso *Poner en Marcha*, implícito si no está explícito. Aunque su prioridad podría no ser alta de acuerdo a otros criterios, es necesario abordar por lo menos una visión simplificada de *Poner en Marcha* en la primera iteración de manera que se proporcione la inicialización que asumen otros casos de uso. En cada iteración, se desarrolla incrementalmente el caso de uso *Poner en Marcha* para satisfacer las necesidades de comienzo de los otros casos de uso. De manera análoga, los sistemas a menudo tienen un caso de uso *Desconectar*. En algunos sistemas, es bastante complejo, como desconectar un commutador de telecomunicaciones activo. En términos de la planificación, si son sencillos, estos casos de uso se pueden listar informalmente en el Plan de Iteración, como “implementar la puesta en marcha y desconexión como se necesite”. Obviamente, versiones más complejas necesitan que se cuiden más los requisitos y la planificación.

Advertencia: Planificación del proyecto vs. objetivos de aprendizaje

El objetivo del libro es ofrecer una ayuda para el aprendizaje de una introducción al análisis y diseño, en lugar de que funcione realmente el proyecto del PDV NuevaEra. Por tanto, se han tomado algunas licencias en la elección de lo que se aborda en las primeras iteraciones del caso de estudio, motivado por objetivos educativos en lugar de por objetivos del proyecto.

36.2. Priorización de los riesgos del proyecto

Un método útil para priorizar los riesgos del proyecto global es estimar su probabilidad e impacto (en coste, tiempo o esfuerzo). La estimación podría ser cuantitativa (que normalmente son muy especulativas) o simplemente cualitativas (por ejemplo, alto-medio-bajo, basado en discusiones y votaciones con puntos en grupo). Los peores riesgos son, naturalmente, aquellos tanto probables como de impacto alto. Por ejemplo:

Riesgo	Probabilidad	Impacto	Ideas para atenuarlo
Número insuficiente y calidad de desarrolladores orientados a objetos expertos.	A	A	Leer el libro. Contratar consultores temporales Educación en aulas & tutorías Diseño y programación por parejas
Demostración no preparada para la próxima convención POS-World en Hamburgo.	M	A	Contratar consultores temporales que sean especialistas en el desarrollo de sistemas de PDV en Java. Identificar requisitos “atractivos” que queden bien en una demostración y darles prioridad, sobre los otros. Maximizar el uso de componentes prefabricados.
...

En términos de los artefactos del UP, esto forma parte del Plan de Desarrollo de Software.

36.3. Planificación adaptable vs. predictiva

Una de las grandes ideas del desarrollo iterativo es la adaptación basada en la retroalimentación, en lugar de intentar predecir y planificar *en detalle* el proyecto completo. En consecuencia, en el UP, uno crea un Plan de Iteración sólo para la *siguiente* iteración. Más allá de la siguiente iteración se deja abierto el plan detallado, para que se ajuste de manera adaptable en el futuro (ver Figura 36.1). Además de fomentar el comportamiento flexible y oportunista, una razón sencilla para no planificar el proyecto completo en detalle es que en el desarrollo iterativo no todos los requisitos, detalles de diseño y, por tanto, las etapas, se conocen al comienzo del proyecto¹. Otra es la preferencia de confiar en el juicio de planificación del equipo conforme ellos proceden. Finalmente, suponga que hubiese un plan detallado de grano fino al comienzo del proyecto, y el equipo se “desvía” de él por adquirir una nueva percepción de cómo ejecutar el proyecto de la mejor manera. Desde el exterior, esto podría verse como un tipo de fallo, cuando de hecho es justo lo contrario.

¹ Tampoco se conocen realmente o de manera fiable en un proyecto “en cascada”, aunque se planifique de manera detallada el proyecto completo como si se conocieran.

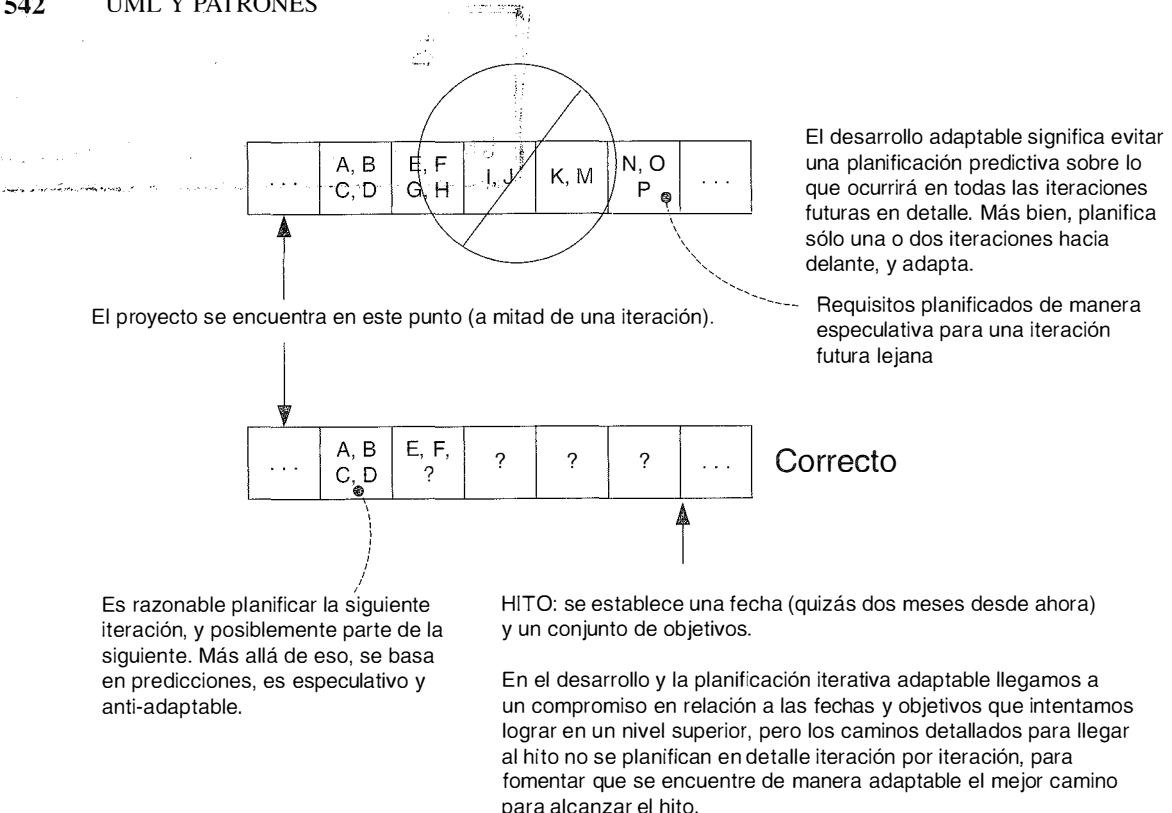


Figura 36.1. Los hitos son importantes, pero evite la planificación detallada que predice un futuro lejano.

Sin embargo, *existen* todavía objetivos e hitos; el desarrollo adaptable no significa que el equipo no sabe hacia dónde va, o las fechas de los hitos y objetivos. En el desarrollo iterativo, el equipo todavía llega a un compromiso en cuanto a las fechas y objetivos, pero el camino detallado para alcanzarlos es flexible. Por ejemplo, el equipo NuevaEra podría establecer un hito de que en tres meses se completarán los casos de uso *Procesar Venta*, *Gestionar Devoluciones* y *Autenticar Usuarios*, y las características de registrar y reglas conectables. Pero —y éste es el punto clave— no se define en detalle el plan o camino de grano fino de las iteraciones fijadas en dos semanas para ese hito. No se fija el orden de las etapas, o lo que se va a hacer en cada iteración a lo largo de los siguientes tres meses. Más bien, únicamente se planifica la siguiente iteración de dos semanas, y el equipo se adapta paso a paso, trabajando para cumplir los objetivos para la fecha de entrega. Por supuesto, las dependencias en componentes y recursos naturalmente restringen algo el orden del trabajo, pero no todas las actividades necesitan que se planifiquen en detalle de grano fino.

El personal involucrado externo ve un plan de macro-nivel (como al nivel de tres meses) al que el equipo se compromete. Pero la organización de micro-nivel se deja al mejor —y adaptable— juicio del equipo, cuando se beneficie de nuevas percepciones (ver Figura 36.1).

Finalmente, aunque en el UP se prefiere la planificación adaptable de grano fino, cada vez más es posible planificar con éxito hacia delante dos o tres iteraciones (con niveles crecientes de desconfianza) conforme los requisitos y la arquitectura se estabilizan, el equipo madura, y los datos se recogen a la velocidad del desarrollo.

36.4. Planes de Fase y de Iteración

A un macro-nivel, es posible establecer fechas y objetivos de los hitos; pero a micro-nivel, el plan para los hitos se deja flexible excepto para el futuro inmediato (por ejemplo, las siguientes cuatro semanas). Estos dos niveles se reflejan en el **Plan de Fase** y **Plan de Iteración** del UP, ambos forman parte del Plan de Desarrollo del Software compuesto. El Plan de Fase expone las fechas y objetivos de los hitos de macro-nivel, tales como los hitos del final de las fases y de las pruebas del sistema piloto a mitad de la fase. El Plan de Iteración define el trabajo para la iteración actual y la siguiente —no todas las iteraciones— (ver Figura 36.2).

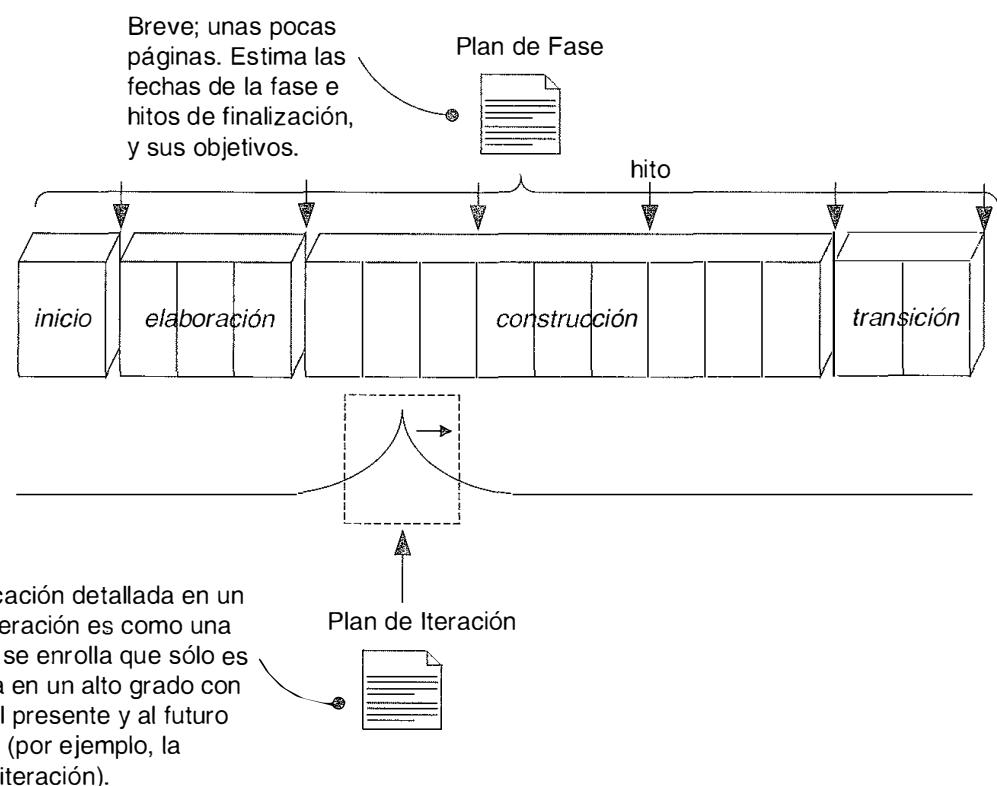


Figura 36.2. Planes de Fase y de Iteración.

Durante la fase de inicio, los hitos estimados en el Plan de Fase son vagos “estimados en base a conjeturas”. A medida que se progresó en la elaboración, las estimaciones se mejoran. Un objetivo de la fase de elaboración es tener, a su terminación, suficiente información realista para que el equipo llegue a un compromiso en relación con las fechas y objetivos de los hitos importantes para el final de la construcción y transición (esto es, la entrega del proyecto).

36.5. Plan de Iteración: ¿qué hacemos en la siguiente iteración?

El UP está dirigido por los casos de uso, lo que en parte implica que se organiza el trabajo alrededor de la finalización de los casos de uso. Es decir, se asigna una iteración para que implemente uno o más casos de uso, o escenarios de casos de uso cuando el

caso de uso completo sea demasiado complejo para una iteración. Y puesto que algunos requisitos no se expresan como casos de uso, sino como características, como registrar y reglas del negocio conectables, éstas también se deben asignar a una o más iteraciones (ver Figura 36.3).

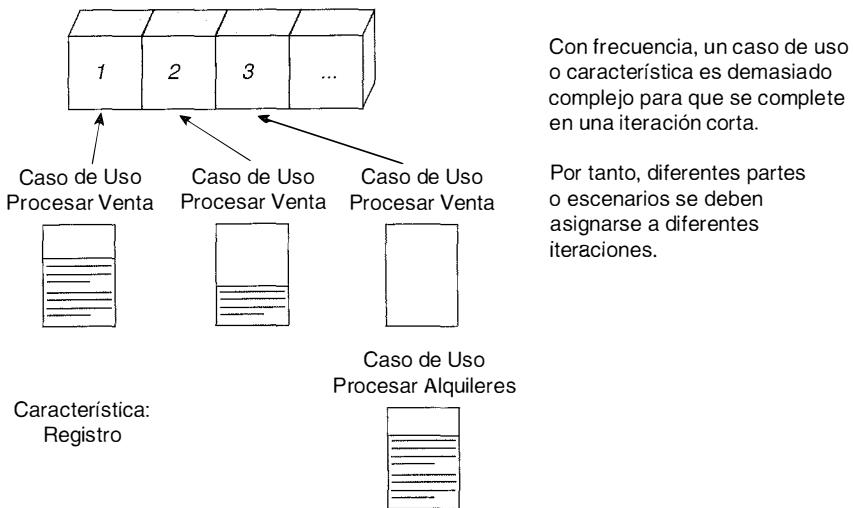


Figura 36.3. Trabajo asignado a una iteración.

Normalmente, se dedica la primera iteración de la elaboración a innumerables tareas generales como instalación y ajuste de herramientas y componentes, aclarar los requisitos, etcétera.

La priorización de los requisitos guía la elección del trabajo inicial. Por ejemplo, el caso de uso *Procesar Venta* es claramente importante. Por tanto, empezamos a abordarlo en la primera iteración. Aunque, no se implementan todos los escenarios de *Procesar Venta* en la primera iteración. Más bien, se escogen algunos escenarios simples, de camino feliz, como el de pago sólo en efectivo. Aunque el escenario es simple, su implementación comienza a desarrollar algunos elementos centrales del diseño.

Durante las iteraciones de la elaboración se abordarán requisitos diferentes, significativos para la arquitectura, relacionados con este caso de uso, forzando al equipo a tratar brevemente muchos aspectos de la arquitectura: las capas importantes, la base de datos, la interfaz de usuario, la interfaz entre los subsistemas importantes, etcétera. Esto nos lleva a la creación en las primeras etapas de una implementación “en anchura y superficial” a través de muchas partes del sistema —un objetivo común en la fase de elaboración—.

36.6. Traza de los requisitos a través de las iteraciones

La tarea de crear el primer Plan de Iteración nos trae una cuestión relevante en el desarrollo iterativo, que se ilustra en la Figura 36.3.

Como se indicó en la última sección, no se implementarán todos los escenarios de *Procesar Venta* en la primera iteración. De hecho, este complejo caso de uso podría tar-

dar en completarse muchas iteraciones de dos semanas a lo largo de un periodo de seis meses. Cada iteración abordará nuevos escenarios o partes de escenarios.

Cuando no es posible llevar a cabo todos los escenarios de un caso de uso en una iteración, surge un problema en la traza de los requisitos. ¿Cómo recoge uno qué partes del caso de uso se han completado, en cuáles se está trabajando actualmente, o todavía no se han hecho? Una herramienta de requisitos construida para este trabajo proporciona una solución.

Un ejemplo es la herramienta de Rational, RequisitePro, y merece la pena dedicar un momento a su estudio para entender cómo trabajan estas herramientas para mantener la traza de los casos de uso completados parcialmente a lo largo de las iteraciones. No es que propongamos esta herramienta, sino que se ofrece la presentación para ilustrar una solución a este importante problema de mantener la traza.

Un ejemplo de herramienta de gestión de requisitos

RequisitePro está integrada con Microsoft Word de manera que uno podría introducir y editar los requisitos en Word, seleccionar una frase y definir la frase seleccionada como un requisito al que se le va a seguir la pista en RequisitePro.

Cada requisito puede tener una variedad de atributos, como el estado, riesgo, etcétera (ver Figuras 36.4 y 36.5). Con una herramienta como ésta, se puede manejar el problema de mantener la traza de la terminación parcial de los casos de uso a lo largo de las iteraciones.

Todas las sentencias de los escenarios principales de éxito y las extensiones se pueden representar individualmente como requisitos a los que se les va a seguir la pista, y

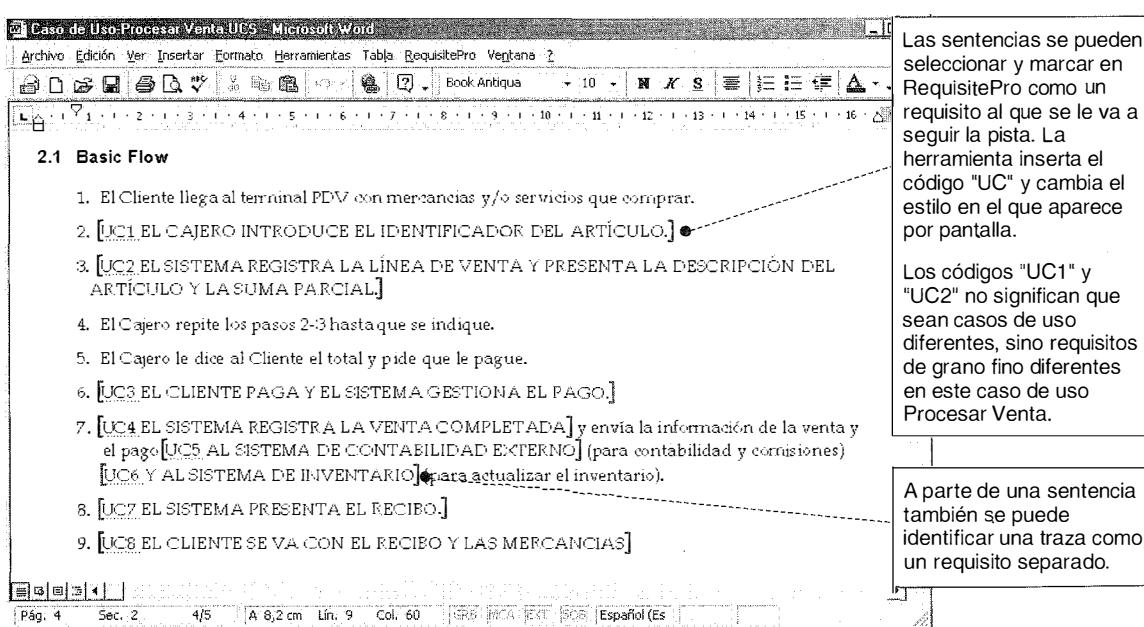


Figura 36.4. Etiquetado básico de frases de los casos de uso como requisitos.

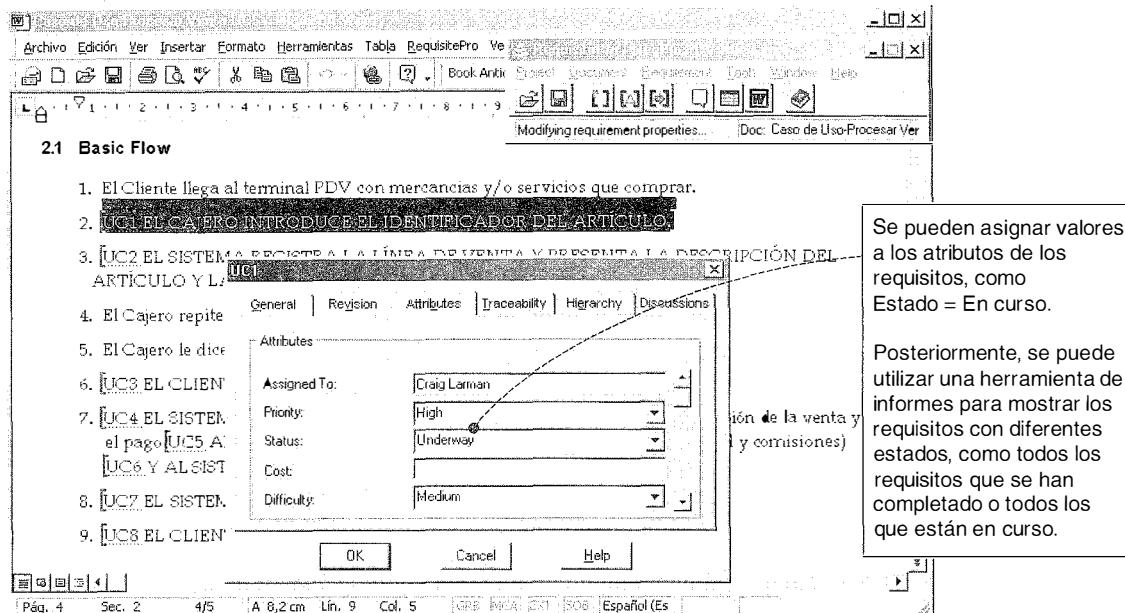


Figura 36.5. Cada requisito etiquetado tiene muchos atributos.

cada uno se puede identificar con varios valores de estado como *propuesto*, *aprobado*, etcétera.

36.7. La (in) validez de las primeras estimaciones

Basura entra, basura sale. Las estimaciones hechas con información poco fiable y difusa son poco fiables y difusas. En el UP se entiende que las estimaciones que se hacen durante la fase de inicio no son de fiar (esto se cumple para todos los métodos, pero el UP lo reconoce). Las estimaciones iniciales de la fase de inicio proporcionan una guía sobre si merece la pena que se haga un estudio real en la elaboración, para generar una buena estimación. Después de la primera iteración de la elaboración contamos con alguna información realista para producir una estimación aproximada. Después de la segunda iteración, la estimación comienza a cobrar credibilidad (ver Figura 36.6).

Las estimaciones útiles requieren de la investigación en algunas iteraciones de la elaboración.

Esto no significa que sea imposible o no merezca la pena intentar hacer estimaciones precisas al principio. Si es posible, muy bien. Sin embargo, la mayoría de las organizaciones no encuentran que éste sea el caso, por razones entre las que se encuentran la continua introducción de nuevas tecnologías, aplicaciones nuevas, y muchas otras complicaciones. Por tanto, el UP aboga por realizar un poco de trabajo realista en la elaboración antes de generar las estimaciones que se utilizan para la planificación del proyecto y la elaboración de los presupuestos.

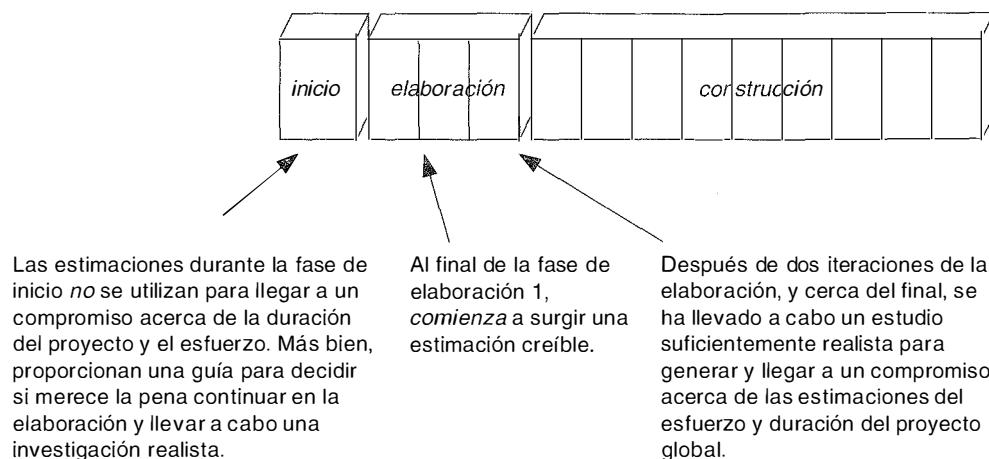


Figura 36.6. Estimación y fases del proyecto.

36.8. Organización de los artefactos del proyecto

El UP organiza los artefactos en función de las disciplinas. El Modelo de Casos de Uso y la Especificación Complementaria están en la disciplina de Requisitos. El Plan de Desarrollo del Software forma parte de la disciplina de Gestión del Proyecto, etcétera. Por tanto, organice carpetas en su control de versiones y sistema de directorios para reflejar las disciplinas, y coloque los artefactos de una disciplina en la carpeta de la disciplina relacionada (ver Figura 36.7).

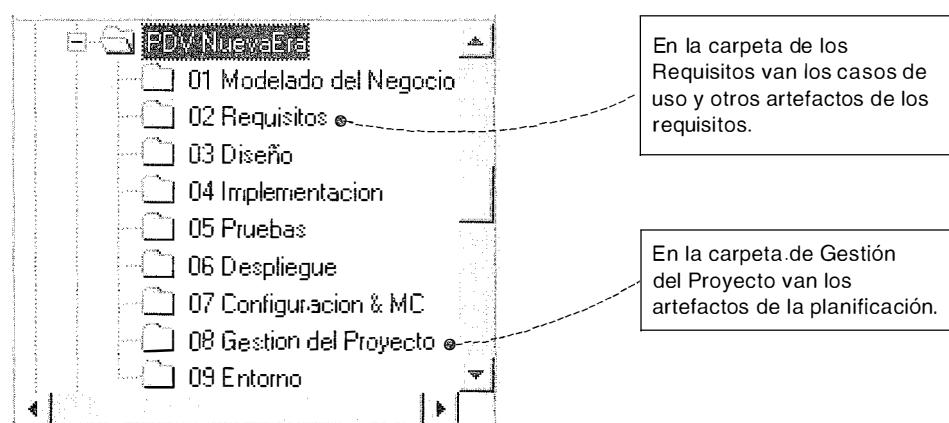


Figura 36.7. Organice los artefactos del UP en carpetas que se corresponden con sus disciplinas.

Esta organización funciona para la mayoría de los elementos que no son de la implementación. Algunos artefactos de la implementación, como la base de datos actual o los ficheros ejecutables, se encuentran comúnmente en diferentes ubicaciones debido a diversos motivos de la implementación.

Sugerencia

Después de cada iteración, utilice la herramienta de control de versiones para crear un punto de control etiquetado y congelado de todos los elementos en estas carpetas (incluyendo el código fuente). Habrá una versión “Elaboración-1”, “Elaboración-2”, y así sucesivamente, de cada artefacto. Para una estimación posterior de la velocidad del equipo (en éste o en otro proyecto), estos puntos de control proporcionan datos reales de la cantidad de trabajo que se hizo en cada iteración.

36.9. Algunas cuestiones de la planificación de la iteración del equipo

Equipos de desarrollo en paralelo

Un proyecto grande normalmente se divide en trabajos de desarrollo en paralelo, donde varios equipos trabajan en paralelo. Una manera de organizar los equipos es a lo largo de líneas de la arquitectura: por capas y subsistemas. Otra estructura organizacional es por conjuntos de características, que podrían muy bien corresponderse a la organización de la arquitectura.

Por ejemplo:

- Equipo de la capa del dominio (o equipo del subsistema del dominio)
- Equipo de la interfaz de usuario
- Equipo de internacionalización
- Equipo de servicios técnicos (equipo de persistencia, etc.)

Equipos en iteraciones de diferente longitud

Algunas veces, el desarrollo de un subsistema (como el servicio de persistencia) a cualquier nivel significativamente utilizable requiere mucho tiempo, especialmente durante sus primeras etapas. En lugar de alargar la longitud de la iteración global para todos los equipos, una alternativa es mantener la iteración corta (en general, un objetivo

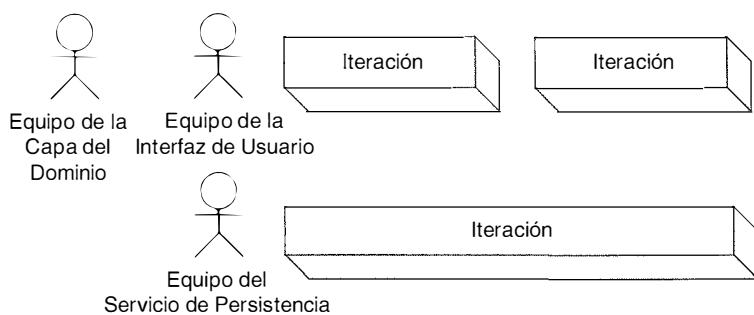


Figura 36.8. Diversas longitudes de las iteraciones.

respetable) para la mayoría de los equipos, y de longitud doble para el equipo “más lento” (ver Figura 36.8).

Velocidad del equipo y adopción incremental del proceso

Además de necesitar iteraciones más largas para los equipos muy grandes, otro motivo para alargar una iteración (por ejemplo, de tres semanas a cuatro), está relacionado con la velocidad y experiencia del equipo. Un equipo para el que son nuevas muchas de las prácticas y las tecnologías, naturalmente irá más despacio, o necesitará más tiempo para completar una iteración. Equipos con menos experiencia se beneficiarán de menos iteraciones *ligeramente* más largas que los equipos con más experiencia.

Nótese que el desarrollo iterativo proporciona un mecanismo para mejorar la estimación de la velocidad: el progreso real en las primeras iteraciones da a conocer las estimaciones para las iteraciones posteriores.

Relacionado con esto está la estrategia **adopción incremental del proceso**. En las primeras iteraciones, los equipos con menos experiencia asumen un conjunto pequeño de prácticas. A medida que los miembros del equipo los digieren y los dominan, añaden más —¡asumiendo que son útiles! Por ejemplo, en las primeras iteraciones el equipo podría construir y probar un sistema una vez al día. En iteraciones posteriores, podrían adoptar integraciones continuas y pruebas del sistema (que suceden muchas veces cada día) con una herramienta de integración continua como la de libre distribución Cruise Control (cruisecontrol.sourceforge.net).

36.10. No se entendió la planificación en el UP cuando...

- Todas las iteraciones se planifican de manera especulativa en detalle, prediciendo el trabajo y los objetivos de cada iteración.
- Se espera que las primeras estimaciones de la fase de inicio o de la primera iteración de la elaboración sean fiables, y se utilizan para llegar a compromisos del proyecto a largo plazo; generalizando, se esperan estimaciones fiables con investigaciones triviales o ligeras.
- Los problemas sencillos o las cuestiones de bajo riesgo se abordan en las primeras iteraciones.

Si la estimación de una organización y el proceso de planificación se parece algo al que se presenta a continuación, no se entendió la planificación que propone el UP:

1. Al comienzo de una fase de planificación anual, se identifican nuevos sistemas o características a un alto nivel; por ejemplo, “Sistemas Web para la gestión de la contabilidad”.
2. A los encargados técnicos se les da poco tiempo para estimar de manera especulativa el esfuerzo y la duración de grandes proyectos, caros o arriesgados, que involucran con frecuencia nuevas tecnologías.

3. Se establece la planificación y presupuesto de los proyectos para el año.
4. Las personas involucradas se preocupan cuando los proyectos reales no se corresponden con las estimaciones originales. Ir al Paso 1.

Este enfoque carece de una estimación realista y refinada iterativamente basada en investigaciones serias como promueve el UP.

36.11. Lecturas adicionales

Software Project Management: A Unified Framework de Royce proporciona una perspectiva iterativa y del UP sobre la gestión y planificación de proyectos.

Surviving Object-Oriented Projects: A Manager's Guide de Cockburn contiene más información útil acerca de la planificación iterativa, y la transición a los proyectos iterativos que hacen uso de la tecnología de objetos.

The Rational Unified Process: An Introduction de Kruchten contiene capítulos útiles dedicados especialmente a la gestión y planificación de proyectos en el UP.

Una advertencia, hay algunos libros que se plantean discutir la planificación para el “desarrollo iterativo” o el “Proceso Unificado” que realmente esconden un enfoque de planificación en cascada o predictivo.

Rapid Development [McConnell96] ofrece una excelente visión global de muchas prácticas y cuestiones relacionadas con la gestión y planificación del proyecto, y riesgos del proyecto.

Capítulo 37

COMENTARIOS ACERCA DEL DESARROLLO ITERATIVO Y EL UP

Deberías utilizar el desarrollo iterativo sólo en los proyectos en los que quieras tener éxito.

Martin Fowler

Objetivos

- Introducir y ampliar algunos temas del UP.
 - Introducir otras prácticas aplicables al desarrollo iterativo.
 - Estudiar cómo el ciclo de vida iterativo puede ayudar a reducir algunos problemas del desarrollo.
-

37.1. Buenas prácticas y conceptos del UP adicionales

La idea más importante que hay que apreciar y poner en práctica en el UP es el desarrollo adaptable, con iteraciones cortas con una duración fija. Entre las buenas prácticas y conceptos claves adicionales del UP encontramos:

- **Abordar las cuestiones de alto riesgo y valor en las primeras iteraciones:** Por ejemplo, si el nuevo sistema es un servidor de aplicaciones que tiene que gestionar 2.000 clientes concurrentes con tiempo de respuesta de la transacción por debajo del segundo, no espere muchos meses (o años) para diseñar e implementar este requisito de alto riesgo. Más bien, céntrese rápidamente en diseñar, programar y probar los componentes esenciales del software y la arquitectura para esta cuestión arriesgada; deje el trabajo fácil hasta las iteraciones posteriores. La idea es disminuir los riesgos altos en las primeras iteraciones, de manera que el proyecto no “fracase tarde”, que es una característica de los proyectos en cascada que retrasan los temas difíciles y arriesgados hasta etapas posteriores del ciclo de vida. Es mejor “fracasar pronto”, si es que fracasa, haciendo primero las cosas difíciles. De

este modo, se dice que el UP está **dirigido por el riesgo**. Finalmente, obsérvese que el riesgo llega de muchas formas: falta de habilidades o recursos, desafíos técnicos, facilidad de uso, política, etcétera. Todas estas formas influyen en lo que se va a abordar en las primeras iteraciones.

- **Usuarios involucrados continuamente:** El desarrollo iterativo y el UP tratan de dar pequeños pasos rápidamente y obtener retroalimentación. Esto requiere una continua atención e implicación por parte de las personas involucradas del negocio y los expertos en la materia de estudio para clarificar y dirigir el proyecto. Al principio, el personal del negocio podría sentir que esto es una imposición. Sin embargo, existe una correlación entre la mayoría de los proyectos que han fracasado y la falta de implicación de los usuarios [Standish94], y este enfoque le proporciona al personal del negocio la capacidad de dar forma al software como realmente lo necesitan. En proyectos donde el “usuario” puede ser cualquiera, como un nuevo sitio web o producto del consumidor, grupos interesados podrían actuar como sustitutos.
- **Atención en las primeras etapas a construir una arquitectura básica cohesiva:** Es decir, el UP está **centrado en la arquitectura**. Esto está relacionado con abordar las cuestiones de alto riesgo en las primeras iteraciones, puesto que normalmente establecer los elementos básicos de la arquitectura es un elemento arriesgado o crítico. Las primeras iteraciones se centran en una implementación de la arquitectura “en anchura y superficial”, estableciendo las cuestiones de diseño importantes, y los subsistemas con sus interfaces y responsabilidades. El equipo “investigará” en áreas verticalmente profundas sobre requisitos específicos difíciles o arriesgados, como el requisito para transacciones por debajo del segundo con 2.000 clientes concurrentes.
- **Verificar continuamente la calidad, desde el principio y con frecuencia:** La calidad en este contexto abarca satisfacer o superar correctamente los requisitos en un proceso sostenible y repetible, con software de fácil mantenimiento y escalable. Una razón para una campaña temprana, continua e intensiva de pruebas, inspección y aseguramiento de la calidad es que el coste de un defecto tardío se incrementa de manera no lineal a través de las fases de un proyecto. Además, el desarrollo iterativo se basa en la retroalimentación y la adaptación; por tanto, las pruebas y evaluación realistas en las primeras etapas son actividades críticas para obtener retroalimentación significativa. Esto difiere del proyecto en cascada, donde la etapa de aseguramiento de la calidad se hace cerca del final del proyecto, cuando la respuesta es la más difícil y costosa. En el UP, la verificación de la calidad se integra de manera continua desde el principio, de manera que no hay grandes sorpresas cerca del final del proyecto. Observe que en el UP, la verificación de la calidad también se refiere a la calidad del proceso —cada iteración, evaluando lo bien que lo está haciendo el equipo—.
- **Aplicar casos de uso:** Informalmente, los casos de uso son historias escritas del uso de un sistema. Son mecanismos para explorar y recoger los requisitos funcionales, a diferencia del estilo más antiguo de la lista de funciones o la lista “El sistema deberá hacer...”. El UP recomienda que se apliquen los casos de uso como la forma principal para la captura de los requisitos, y como una fuerza que dirige la planificación, diseño, pruebas y la escritura de la documentación del usuario final.

- **Modelar el software visualmente:** Un porcentaje muy importante del cerebro de las personas está implicado en el procesamiento visual, que es una de las motivaciones que hay detrás de la presentación visual o gráfica de la información [Tufte92]. Por tanto, es conveniente emplear no sólo lenguajes textuales (como texto o código), sino también lenguajes simbólicos, basados en diagramas, visuales con orientación espacial, como UML, porque esto se beneficia de las ventajas naturales del cerebro¹. Además, la *abstracción* es una práctica útil para razonar sobre los diseños del software y comunicarlos, porque esto nos permite centrarnos en los aspectos importantes, mientras ocultamos o ignoramos los detalles que producen ruido. Un lenguaje visual como UML nos permite visualizar y razonar sobre los modelos abstractos del software, pasando rápidamente al diseño con esquemas en forma de diagramas de las grandes ideas. Pero como se estudiará más tarde, existe un “punto dulce en UML” entre realizar pocos o demasiados diagramas.
- **Gestión de requisitos cuidadosa:** Esto no significa hacer uso del estilo en cascada de definir completamente y congelar los requisitos en la primera fase del proyecto. Más bien, implica no ser descuidado; es decir, ser habilidoso en la elicidadación, escritura, asignación de las prioridades, establecimiento de las trazas y mantenimiento de la pista a lo largo del ciclo de vida de los requisitos, normalmente con el soporte de una herramienta. Esto parece obvio, pero da la impresión de que rara vez se pone en práctica bien. La gestión de requisitos pobre es un factor común en proyectos que tienen problemas [Standish94].
- **Control de cambios:** Esta práctica comprende varias ideas: Primero, gestión de peticiones de cambios. Aunque un proyecto UP iterativo acepta los cambios, no acepta el caos. Cuando surge la solicitud de un nuevo requisito durante las iteraciones, en lugar de un jovial “¡Por supuesto, sin problemas!” hay una evaluación racional del esfuerzo y el impacto, y si se acepta, se modifica la planificación. También incluye la idea de mantener la pista del ciclo de vida de todas las peticiones de cambio (solicitado, en curso,...). Segundo, gestión de configuraciones. Se utilizan herramientas de gestión de configuraciones y construcción para dar soporte a la frecuente (idealmente, por lo menos diariamente) integración y pruebas del sistema, desarrollo en paralelo, áreas de trabajo de desarrolladores y configuraciones separadas, y control de versiones —desde el comienzo del proyecto—. En el UP, todos los bienes del proyecto (no sólo el código) deberían estar bajo el control de configuraciones y versiones.

37.2. Las fases de construcción y transición

Construcción

La elaboración termina cuando se han resuelto las cuestiones de alto riesgo, se ha completado el núcleo central o esqueleto de la arquitectura y se han entendido “la mayoría” de los requisitos. Al final de la elaboración, es posible estimar de manera más realista el esfuerzo y duración restante del proyecto.

¹ Éste también es un motivo para utilizar colores en los diagramas (a no ser que algún miembro del equipo sea daltónico). Por ejemplo, véase [CDL99].

Le sigue la **fase de construcción**, cuyo propósito es esencialmente terminar de construir la aplicación, realizar pruebas alpha, preparar las pruebas beta (en la fase de transición) y preparar el despliegue, mediante actividades tales como la escritura de las guías de usuario y la ayuda on-line. A veces, se resume como poner la “carne sobre el esqueleto” creado en la elaboración. Mientras que la elaboración se puede caracterizar como que construye el núcleo del sistema arrriesgado y significativo para la arquitectura, la construcción se puede describir como que construye el resto. Como antes, el desarrollo avanza por medio de una serie de iteraciones en las que se fija la duración. En cuanto al personal, se recomienda que se utilice un equipo pequeño y cohesivo durante la elaboración, y luego ampliar el tamaño del equipo durante la construcción; además probablemente habrá más equipos en paralelo desarrollando durante esta fase.

Transición

La construcción termina cuando se considera que el sistema está preparado para el despliegue operacional, y se han completado todos los materiales de soporte, como las guías de usuario, materiales de aprendizaje, etcétera. Le sigue la **fase de transición**, cuyo propósito es poner el sistema en producción. Esto podría incluir actividades como pruebas beta, reaccionar a la retroalimentación de las pruebas beta, pequeños ajustes, conversión de datos, cursos de entrenamiento, marketing para el lanzamiento del producto, funcionamiento en paralelo del antiguo y el nuevo sistema, y cosas por el estilo.

37.3. Otras prácticas interesantes

Ésta no es una lista exhaustiva, pero entre las prácticas interesantes —no documentadas explícitamente en el UP— que han sido de valor en los proyectos iterativos se encuentran:

- El patrón de proceso **SCRUM** [BDSSS00]; véase también www.controlchaos.com. El más concreto es una reunión de SCRUM diaria de pie de “15 minutos”. El jefe del proyecto pregunta a cada persona: 1) las cosas hechas desde la última reunión; 2) los objetivos para el día siguiente; y 3) los obstáculos para que el jefe los elimine. Yo también he preguntado a cada miembro las percepciones relevantes que quiere compartir con el equipo. La reunión fomenta el comportamiento del equipo adaptable que emerge, medidas del progreso de grano fino, comunicación de alta densidad y la socialización de los proyectos. Otras ideas claves son: el equipo está libre de todas las distracciones externas, no se le añade ningún trabajo adicional (desde fuera del equipo) durante una iteración, y el trabajo de gestión es eliminar todos los obstáculos y distracciones, de manera que el equipo pueda centrarse.
- Algunas prácticas de la **Programación Extrema** (XP, *Extreme Programming*) [Beck00], como **programar probando primero**: escribir una prueba de unidad *antes* del código que se va a probar, y escribir las pruebas para casi todas las clases. Si estamos trabajando con Java, **JUnit** (www.junit.org) es un framework de pruebas conocido de libre de distribución. Escriba un poco de las pruebas, escriba un poco de código, haga que pase las pruebas, repita. Es esencial que se escriban las pruebas *en primer lugar* para experimentar el valor de este enfoque.

- **Integración continua**, otra práctica de XP; véase [Beck00] para obtener una introducción y www.martinfowler.com para los detalles. El UP incluye la buena práctica de integrar el sistema completo al menos una vez en cada iteración. Esto se abrevia con frecuencia como la práctica de la **construcción diaria**. Integración *continua* lo acorta todavía más, integrando todo el nuevo código que se registra (al menos) cada pocas horas. Aunque esto se puede hacer manualmente, una alternativa es utilizar un entorno que automatice la integración continua y las pruebas en una máquina para la construcción rápida que ejecuta un proceso demonio. Periódicamente se despierta (como cada dos minutos) y busca nuevo código para registrar, que dispara la ejecución de un *script* de prueba y reconstrucción. Está disponible libremente un sistema de integración continua para los proyectos Java denominado **Cruise Control** de libre distribución en SourceForge (cruisecontrol.sourceforge.net).

37.4. Motivos para fijar la duración de una iteración

Hay por lo menos cuatro motivos para fijar la duración de una iteración.

Primero, la **ley de Parkinson**. Parkinson con decepción observó que “El trabajo se expande de manera que rellena el tiempo disponible para su terminación” [Parkinson58]. Fechas de finalización distantes o difusas (por ejemplo, dentro de seis meses), agravan este efecto. Cerca del comienzo del proyecto, puede parecer que hay mucho tiempo para proceder sin prisa. Pero si la fecha de finalización para la siguiente iteración es sólo dentro de *dos semanas*, y se debe colocar un sistema parcial ejecutable y probado para esta fecha, el equipo tiene que centrarse, tomar decisiones y moverse.

Segundo, **asignar prioridades y decisión**. Las iteraciones cortas con la duración fija fuerzan al equipo de desarrollo a tomar decisiones teniendo en cuenta la prioridad del trabajo y los riesgos, identificar cuáles tienen el valor más alto para el negocio o técnico, y estimar algo de trabajo. Por ejemplo, si está embarcado en la primera iteración, que se ha elegido que dure exactamente cuatro semanas, no hay mucha libertad para ser vago —se deben tomar decisiones concretas sobre lo que se hará realmente en las cuatro primeras semanas—.

Tercero, **satisfacción del equipo**. Las iteraciones cortas con la duración fija nos llevan a una rápida y repetida sensación de terminación, competencia y finalización. En ciclos ordinarios de dos o cuatro semanas, el equipo tiene la experiencia de terminar algo, en lugar de trabajar lentamente durante meses sin terminar nada. Estos factores psicológicos son importantes para la satisfacción del trabajo individual, y para generar confianza en el equipo.

Cuarto, **confianza del personal involucrado**. Cuando un equipo llega a un compromiso público de producir algo ejecutable y estable en un periodo corto de tiempo, en una fecha concreta, como dentro de dos semanas, y lo hace, el personal del negocio y otras personas implicadas incrementan su confianza en el equipo y en el proyecto.

37.5. El ciclo de vida secuencial en “cascada”

A diferencia del ciclo de vida iterativo del UP, una antigua alternativa es el ciclo de vida secuencial, lineal, o “en cascada” [Royce70], asociado a procesos grandes y predictivos.

En la práctica habitual, un ciclo de vida en cascada define etapas parecidas a las siguientes:

1. Clarificar, recoger y llegar a un compromiso de un conjunto de requisitos finales.
2. Diseñar un sistema en base a estos requisitos.
3. Implementar, basado en el diseño.
4. Integrar módulos dispares.
5. Evaluar y probar para conseguir corrección y calidad.

Un proceso de desarrollo basado en el ciclo de vida en cascada se asocia con estos comportamientos o aptitudes:

- Definición completa y cuidadosa de un artefacto (por ejemplo, los requisitos o el diseño) antes de pasar a la siguiente etapa.
- Comprometerse a un conjunto congelado de requisitos detallados.
- La desviación de los requisitos o el diseño durante las etapas posteriores indican un fallo porque no se ha sido lo suficientemente habilidoso o exhaustivo. ¡La próxima vez, intente con más empeño hacerlo bien!

Un proceso en cascada es similar al enfoque de la ingeniería según el cual se construyen los edificios o los puentes. Su adopción hace que el desarrollo del software parezca más estructurado y análogo a la ingeniería en algún otro campo. Durante algún tiempo, el proceso en cascada fue el enfoque que aprendieron más desarrolladores de software, directores, autores y profesores cuando eran estudiantes (y después repetían), sin investigar de manera crítica su aplicabilidad al desarrollo del software.

Algunas cosas se deberían construir como los edificios —tales como, bien... edificios— pero, normalmente, no el software.

Como se mencionó en el capítulo preliminar acerca del UP, el estudio de dos años presentado en el *MIT Sloan Management Review*, acerca de los proyectos software exitosos identificó cuatro factores comunes para el éxito: el desarrollo iterativo, en lugar del ciclo de vida en cascada, era el primero de la lista [MacCormack01].

Algunos problemas con el ciclo de vida en cascada

La metáfora de la construcción ha sobrevivido a su utilidad. Es el momento de cambiar de nuevo. Si, como creo, las estructuras conceptuales que construimos hoy son demasiado complicadas para que se especifiquen con precisión por adelantado, y demasiado complejas para que se construyan sin errores, entonces debemos utilizar un enfoque radicalmente diferente (desarrollo iterativo, incremental).

—Frederick Brooks, “No Silver Bullet”, *The Mythical Man-Month*

En una cierta escala de tiempo, realizar parte de los requisitos antes del diseño, y parte del diseño antes de la implementación, es inevitable y sensato. Para un proyecto corto de dos meses, se puede utilizar un ciclo de vida secuencial. Y una iteración individual en el desarrollo iterativo es como un proyecto en cascada corto.

Sin embargo, comienzan a aparecer las dificultades cuando la escala de tiempo se alarga. La complejidad pasa a ser alta, las decisiones especulativas se incrementan y complican, no existe retroalimentación, y en general las cuestiones de alto riesgo no se están abordando lo suficientemente pronto. Por definición, uno intenta todo o la mayoría del trabajo de los requisitos para el sistema completo antes de avanzar, y la mayoría del diseño antes de pasar a lo siguiente.

Se abordan etapas largas en las que se toman muchas decisiones sin el beneficio de una retroalimentación concreta a partir de implementaciones y pruebas reales. En la escala de un mini-proyecto de dos semanas (es decir, una iteración), se puede utilizar una secuencia lineal requisitos-diseño-implementación; el grado de compromisos especulativos de algunos requisitos y del diseño no se encuentra en la zona peligrosa. Sin embargo, a medida que se amplía la escala, así lo hace la especulación y el riesgo.

Entre los problemas relacionados con un proceso en cascada a la escala del proyecto completo encontramos:

- Mitigación del riesgo atrasada; abordando tarde problemas de alto riesgo o difíciles.
- Especulación e inflexibilidad de los requisitos y el diseño
- Alta complejidad
- Baja adaptabilidad

Mitigación de algunos problemas con el ciclo de vida iterativo

El desarrollo iterativo no es una bala mágica para el desafío del desarrollo del software. Pero, ayuda a reducir algunos problemas agravados por un ciclo de vida en cascada lineal.

Problema: Mitigación del riesgo atrasada

El riesgo aparece de muchas formas: el diseño incorrecto, el conjunto de requisitos equivocado, un entorno político extraño, falta de habilidades o recursos, facilidad de uso, etcétera.

En un ciclo de vida en cascada, *no* existe un intento activo de identificar y mitigar en primer lugar las cuestiones de más riesgo. Por ejemplo, la arquitectura incorrecta para un sitio web de alta carga y alta disponibilidad puede causar retrasos costosos, o cosas peores. En un proceso en cascada, la validación de la adecuación de la arquitectura tiene lugar *mucho tiempo* después de que se especifiquen (inevitablemente de manera imperfecta) todos los requisitos y todo el diseño, durante la importante etapa posterior de la implementación. Esto podría ser muchos meses e incluso años después de la fase de inicio del proyecto (ver Figura 37.1). Y no hay escasez de historias donde equipos separados hayan construido subsistemas a lo largo de un largo periodo, y entonces intentaran integrar éstos y comenzar las pruebas del sistema global cerca del final del proyecto —previsiblemente con resultados dolorosos—.

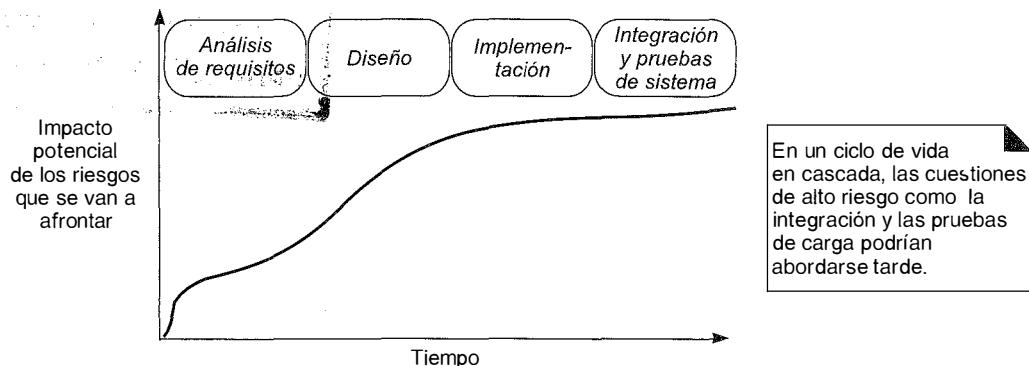


Figura 37.1. Ciclo de vida en cascada y los riesgos.

Mitigación

En cambio, en el desarrollo iterativo el objetivo es identificar y mitigar las cuestiones de más riesgo cuanto antes. Los riesgos altos podrían encontrarse en el diseño del núcleo de la arquitectura, la facilidad de uso de las interfaces, personal involucrado que no se implica. Sea lo que sea, se abordan en primer lugar. Como se ilustra en la Figura 37.2, las primeras iteraciones se centran en disminuir el riesgo. Continuando con el ejemplo anterior del sitio web con carga alta, en un enfoque iterativo, antes de realizar mucho estudio en otros requisitos o trabajo de diseño, el equipo, en primer lugar, diseña, implementa y prueba de manera realista lo suficiente del núcleo de la arquitectura para demostrar que se encuentran en el camino adecuado con respecto a la carga y a la disponibilidad. Si las pruebas demuestran que están equivocados, adaptan el diseño básico en las primeras etapas del proyecto, en lugar de cerca del final.

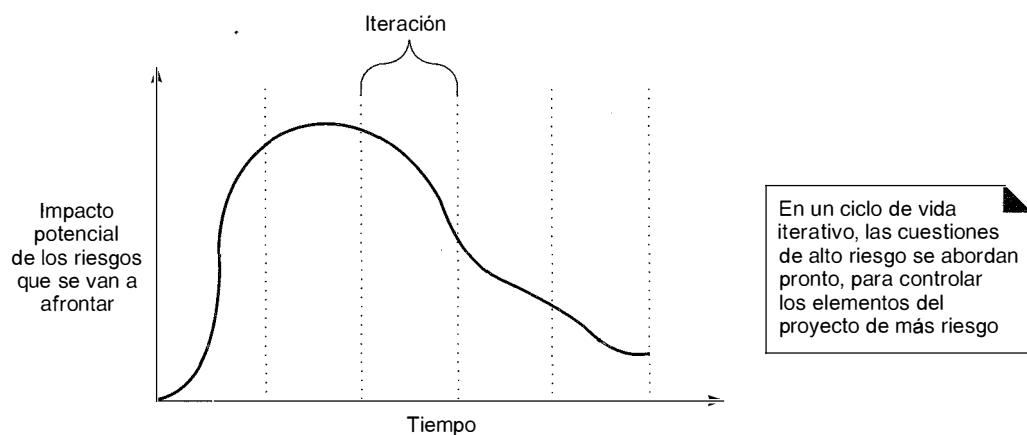


Figura 37.2. Ciclo de vida iterativo y los riesgos.

Problema: Especulación e inflexibilidad de los requisitos

Una suposición fundamental en un proceso en cascada es que los requisitos se pueden especificar por completo y después se congelan en la primera fase de un proyecto. En ta-

les proyectos, se hace un esfuerzo por realizar en primer lugar el análisis de requisitos completo, culminando en un conjunto de artefactos de requisitos que se revisan y se “dan por concluidos”.

Normalmente resulta ser una suposición que tiene algún fallo. El esfuerzo para obtener todos los requisitos definidos y dados por concluidos antes del trabajo de diseño e implementación es probable, irónicamente, que incremente las dificultades del proyecto en lugar de mejorarlas. También lo hace difícil responder más tarde en un proyecto a una nueva oportunidad del negocio por medio de un cambio en el software.

Garantizado, hay algunos proyectos que necesitan que se realice en primer lugar un esfuerzo para especificar completamente y con precisión los requisitos. Esto es especialmente cierto cuando el software se acopla con la construcción de componentes físicos. Entre los ejemplos encontramos los dispositivos de aviación y médicos. Pero nótese que incluso en este caso, el desarrollo iterativo puede todavía aplicarse con provecho al proceso de diseño e implementación.

La investigación que suscita mayor interés en la crítica del mito de ser capaz de definir en primer lugar, con éxito, todos los requisitos procede de [Jones97]. Como se ilustra en la Figura 37.3, en este amplio estudio de 6.700 proyectos, los requisitos arrastrados —aquejlos que no se anticiparon cerca del comienzo— son un hecho significativo de la vida del desarrollo del software, variando desde alrededor del 25% en proyectos de tamaño medio, hasta el 50% en los más amplios; Boehm y Papaccio presentan conclusiones basadas en investigaciones análogas en [BP88]. Las actitudes en cascada, que luchan en contra de (o simplemente niegan) este hecho, asumiendo que los requisitos y los diseños se pueden especificar y congelar, son incongruentes con la mayoría de las realidades de los proyectos.

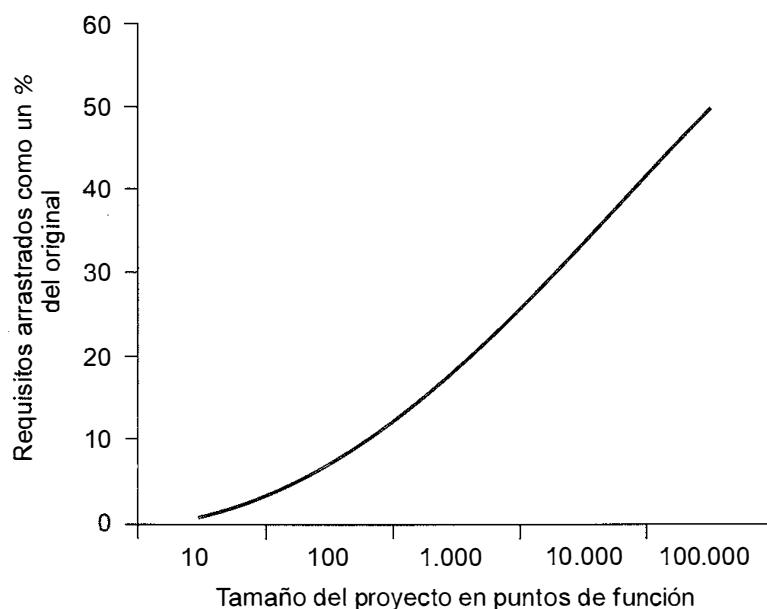


Figura 37.3. Los requisitos que cambian son una fuerza inevitable del desarrollo².

² Los puntos de función describen la complejidad del sistema con una métrica independiente del lenguaje de programación (véase www.ifpug.org).

Por tanto, “la única constante es el cambio”, normalmente porque:

- Las personas involucradas cambian de idea o no pueden imaginarse lo que quieren hasta que no ven un sistema concreto³.
- El mercado cambia.
- La especificación validada correctamente, detallada, y precisa es un desafío psicológico y organizativo para la mayoría de las personas involucradas [Kruchten00].

Y así, existen problemas previsibles y que se ve con frecuencia que surgen en los proyectos en cascada. Puesto que en realidad el cambio significativo es inevitable, éstos incluyen:

- Como se describió anteriormente, descubrimiento y mitigación de altos riesgos retrasado.
- Un sentimiento negativo entre los miembros del equipo de “vivir una ficción” o fracaso del proyecto, ya que la realidad de los cambios no se corresponde con el ideal.
- La realización de una amplia (y costosa) inversión en el diseño e implementación incorrectos (puesto que se basa en requisitos incorrectos).
- Falta de sensibilidad ante los deseos de los usuarios, o las oportunidades de mercado, que cambian.

Mitigación

En el desarrollo iterativo, no se especifican todos los requisitos antes del diseño y la implementación, y no se estabilizan los requisitos hasta después de, al menos, varias iteraciones. Por ejemplo:

Primero, se define un subconjunto de los requisitos básicos, por ejemplo, en un taller de requisitos de dos días. Entonces, el equipo selecciona un subconjunto de éstos para el diseño y la implementación (normalmente en base al riesgo o valor de negocio más alto). Despues de una iteración de cuatro semanas, el personal involucrado se reúne en un segundo taller de requisitos de uno o dos días, revisan de manera intensiva el sistema parcial y aclaran y modifican sus solicitudes. Tras una segunda iteración (más corta) de dos semanas en la que se implementa incrementalmente el sistema, las personas involucradas se reúnen en un tercer taller de requisitos, y refinan de nuevo. A estas alturas, los requisitos comienzan a estabilizarse y representar el verdadero alcance y las intenciones clarificadas de las personas involucradas. A estas alturas, es posible determinar un plan algo realista y una estimación del trabajo restante. Estas iteraciones podrían caracterizarse como parte de la fase de elaboración del UP.

Todavía se admiten cambios posteriores en los requisitos. Sin embargo, la interacción en las primeras iteraciones del trabajo de implementación paralelo y el análisis de requisitos que obtiene retroalimentación a partir de la implementación parcial, nos lleva a una mejor definición de los requisitos en la fase de elaboración.

³ Barry Boehm ha llamado a esto el efecto “Lo sabré cuando lo vea”.

Problema: Especulación e inflexibilidad del diseño

Otra idea central del ciclo de vida en cascada es que la arquitectura y la mayor parte del diseño pueden y deberían especificarse por completo en la segunda fase importante de un proyecto, una vez que se han aclarado los requisitos. En tales proyectos, se hace un esfuerzo por describir a fondo la arquitectura completa, los diseños de los objetos, la interfaz de usuario, el esquema de base de datos, etcétera, antes del que comience la implementación. Algunos problemas asociados con esta suposición:

1. Puesto que los requisitos cambiarán, el diseño original podría no ser fiable.
2. Herramientas, componentes y entornos inmaduros o mal entendidos, hacen las decisiones de diseño especulativas y arriesgadas; podría demostrarse que son incorrectas sobre la implementación porque “no se suponía que el servidor de aplicación fuera a hacer eso...”.
3. En general, la falta de retroalimentación para probar o desaprobar el diseño, hasta mucho tiempo después de que se tomaran las decisiones de diseño.

Mitigación

Estos problemas se mitigan en el desarrollo iterativo construyendo rápidamente parte del sistema y validando el diseño y los componentes de terceras partes mediante las pruebas.

37.6. Ingeniería de usabilidad y diseño de interfaces de usuario

No existe probablemente ninguna técnica con mayor disparidad entre su importancia para el éxito en el desarrollo de software y la falta de una atención y educación formal que la **ingeniería de usabilidad** y el diseño de las interfaces de usuario (UI). Aunque fuera del alcance de esta introducción al A/DOO y el UP, nótese que el UP no incluye el reconocimiento de esta actividad; modelos de usabilidad y UI forman parte de la disciplina de Requisitos. En terminología del UP, los **guiones de los casos de uso** se pueden utilizar para describir de manera abstracta los elementos de la interfaz, y la navegación entre ellos, ya que se relacionan con los escenarios de los casos de uso.

Entre los libros útiles encontramos *Software for Use* de Constantine y Lockwood, *The Usability Engineering Lifecycle* de Mayhew, y *GUI Blooper*s de Johnson.

37.7. El Modelo de Análisis del UP

El UP contiene un artefacto denominado **Modelo de Análisis**; no es necesario y pocos lo crean. Quizás al **Modelo de Análisis** no se le ha asignado el nombre ideal, ya que es realmente un tipo de modelo de diseño. En el uso convencional (por ejemplo, véase [Coleman+94, MO95, Fowler96]), un modelo de análisis sugiere esencialmente un modelo de objetos del dominio —un estudio y descripción de los conceptos del dominio—. Pero el “Modelo de Análisis” del UP es una primera versión del Modelo de Diseño del UP describiendo objetos del software que colaboran, con responsabilidades. Citando textualmente, “El modelo de análisis

es una abstracción, o generalización, del diseño” [Kruchten00]. Y, “Un modelo de análisis puede verse como un primer corte de un modelo de diseño” [JBR99].

El equipo del producto del RUP destaca que es opcional y que su valor no es frecuente, y no fomenta que se cree habitualmente —ya que es otro conjunto más de diagramas que hay que crear antes de la implementación, y los metodólogos y arquitectos expertos rara vez lo utilizan—.

37.8. El producto del RUP

El producto del RUP es un conjunto de documentación basada en el web cohesivo y bien diseñado (páginas HTML) vendido por Rational Software que describe el Proceso Unificado de Rational, un refinamiento actualizado y detallado del UP, que es más general. Describe todos los artefactos, actividades y roles, proporciona guías e incluye plantillas para la mayoría de los artefactos (ver Figura 37.4).

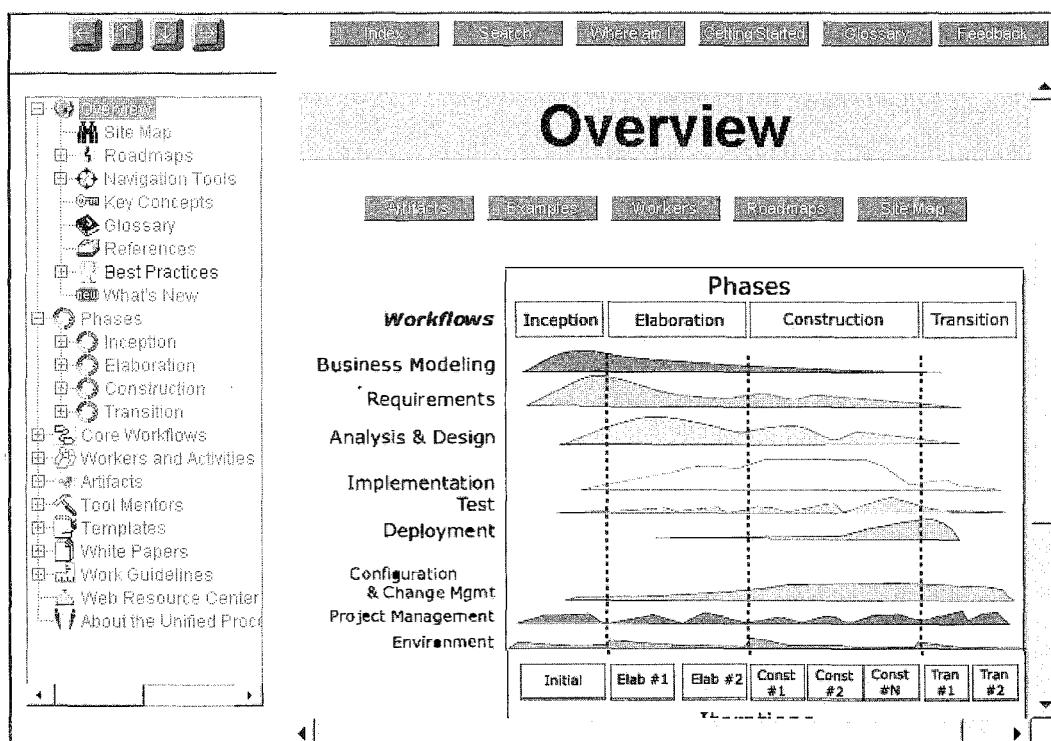


Figura 37.4. El producto del RUP.

El UP se puede aplicar o adoptar con la ayuda de tutores del proceso y libros; las ideas básicas, como el desarrollo iterativo, se describen en éste y en otros libros. En consecuencia, no es necesario poseer el producto del RUP. No obstante, algunas organizaciones consideran que colocar este producto basado en el web (y sus plantillas) en su intranet (respetando la licencia) en una localización visible es un mecanismo simple y efectivo para extender gradualmente su adopción. Que una organización pase a un nue-

vo proceso de desarrollo, más allá de un nivel superficial, requiere que se le apoye de varios modos. Además de tutores para el proceso, proyectos pilotos y seminarios, la documentación y las plantillas basadas en el web proporcionadas por el producto del RUP, sin duda son ayudas útiles que merece la pena que se evalúen.

37.9. Los desafíos y mitos de la reutilización

El UP está desarrollado con los proyectos de la tecnología de objetos (TO) en mente, y con frecuencia se promueve la adopción de la TO para conseguir la **reutilización** del software. Conseguir reutilizar de manera significativa es un objetivo loable, pero difícil. Es una función de mucho más que adoptar la TO y escribir clases; la TO no es más que una tecnología que la posibilita mediante un conjunto de cambios técnicos, organizativos y sociales que tienen que ocurrir para poder apreciar una reutilización significativa. Ciertamente, las librerías de clases para los servicios técnicos, como las librerías de la tecnología Java, proporcionan un gran ejemplo de reutilización, pero me estoy refiriendo a la dificultad de reutilizar código creado en una organización, no a las librerías básicas.

En una encuesta a organizaciones que han adoptado la TO, se les preguntó el valor real de su adopción. Curiosamente, la reutilización estaba *al final* de la lista [Cutter97]. Entre los expertos y organizaciones con experiencia en la TO, no es una sorpresa: saben que la popular descripción de la prensa de la TO para reutilizar es, en cierto modo, un mito; la mayoría de las organizaciones lo ven poco. Esto no implica que no sea un objetivo valioso, o que no haya reutilización —merece la pena, y ha habido algo—. Pero no los altos niveles de reutilización que sugieren muchos artículos y libros. Y muchos desarrolladores con experiencia en la TO puede contarle una historia de guerra sobre el intento desorientado a gran escala de una organización para crear las grandes “librerías reutilizables” o servicios para la compañía, gastando un año y millones de dólares, y terminando con un proyecto fracasado, o uno que perdió el rumbo. La reutilización es difícil, y se puede sostener que es una función de cuestiones más sociales y organizativas que técnicas.

¿Significa esto que la TO no tiene valor? En absoluto, pero su valor se ha asociado incorrectamente sobre todo con la reutilización, en lugar de al modo en el que ayuda de manera más perceptible en la práctica: flexibilidad, facilidad de cambio y manejo de la complejidad. La misma encuesta [Cutter97] lista los valores sobresalientes que realmente se experimentan al adoptar la TO: mantenimiento de la aplicación más simple y ahorro de costes. Los sistemas de objetos —si se diseñan bien— son relativamente más fáciles o más rápidos de modificar y extender, que si se utilizan tecnologías no orientadas a objetos. Esto es importante; muchas organizaciones encuentran que la mayoría de los costes globales a largo plazo de una aplicación tienen que ver con la revisión y el mantenimiento, no con el desarrollo original y, por tanto, son importantes las estrategias que permitan reducir los costes de revisión. Aunque es racional querer reducir los costes de desarrollo de nuevos sistemas, es irónico que pocas de las personas involucradas pregunten lo que sigue, “¿Cómo podemos reducir los costes de revisión y mantenimiento?” cuando con frecuencia es lo más caro. Es aquí donde puede contribuir la TO, además de su fuerza y elegancia al abordar sistemas complejos.

Capítulo 38

MÁS NOTACIÓN UML

38.1. Notación general

Especificaciones de estereotipos y propiedades con etiquetas

Los estereotipos se utilizan en UML para clasificar un elemento (ver Figura 38.1).

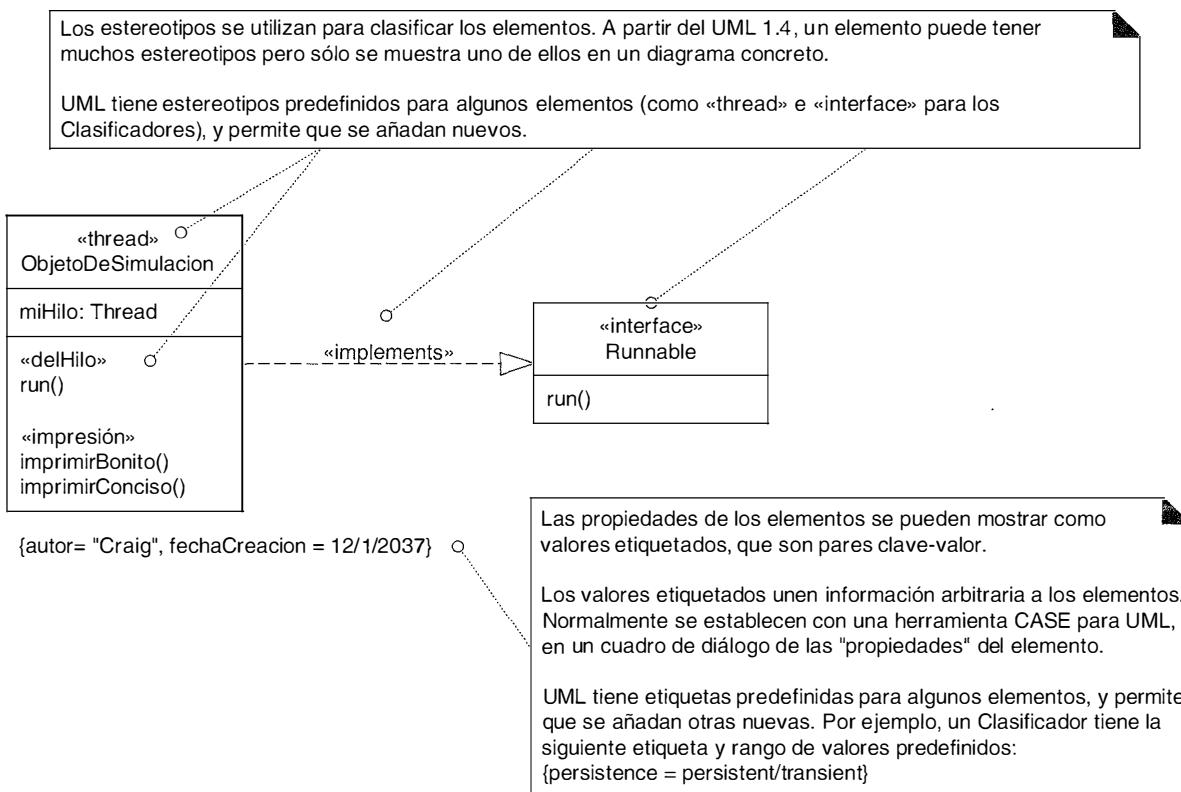


Figura 38.1. Estereotipos y propiedades.

Interfaces de paquetes

Se puede representar un paquete que implementa una interfaz (ver Figura 38.2).

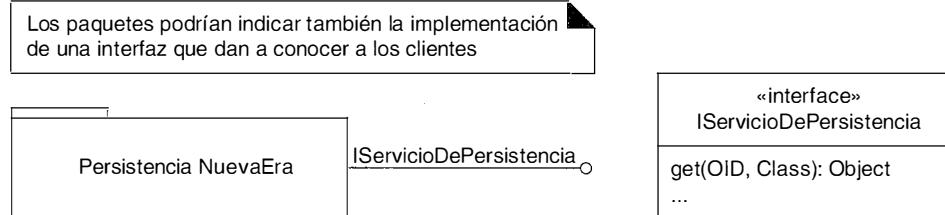


Figura 38.2. Interfaz de un paquete.

Dependencia

Pueden existir dependencias entre elementos cualesquiera, pero probablemente se utilizan con más frecuencia entre en los diagramas de paquetes de UML para ilustrar las dependencias de los paquetes (ver Figura 38.3).

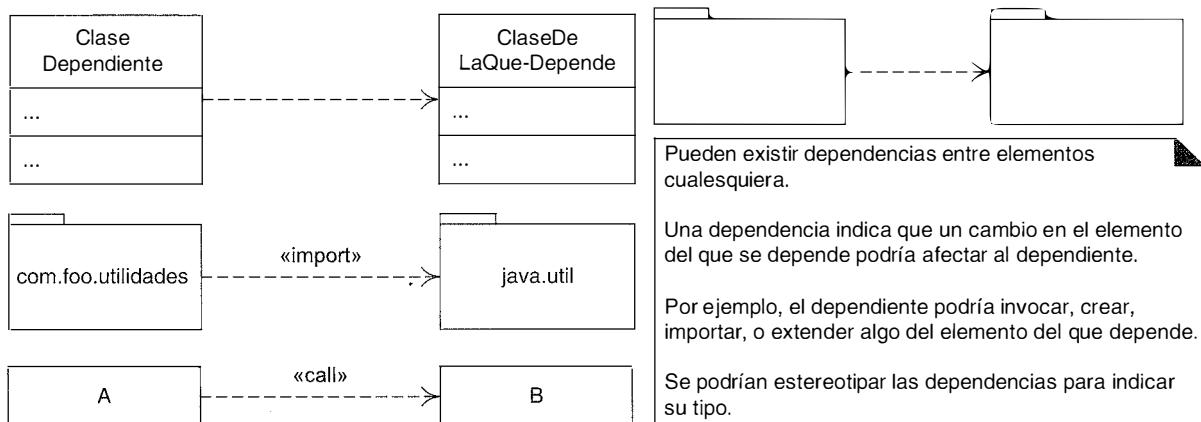


Figura 38.3. Dependencias.

38.2. Diagramas de implementación

UML define varios diagramas que se pueden utilizar para ilustrar los detalles de implementación. El que se utiliza más comúnmente es un diagrama de despliegue, para ilustrar el despliegue de los componentes y procesos en los nodos de proceso.

Diagramas de componentes

Citando textualmente: Un **componente** representa una parte de un sistema modular, desplegable, y reemplazable, que encapsula la implementación y expone un conjunto de interfaces [OMG01]. Podría ser, por ejemplo, código fuente, binario o ejecutable. Entre los

ejemplos encontramos navegadores o servidores HTTP; una base de datos, una DLL, o un fichero JAR (como para un Enterprise Java Bean). Los componentes en UML se representan en los diagramas de despliegue en lugar de independientemente. La Figura 38.4 ilustra algo de la notación común.

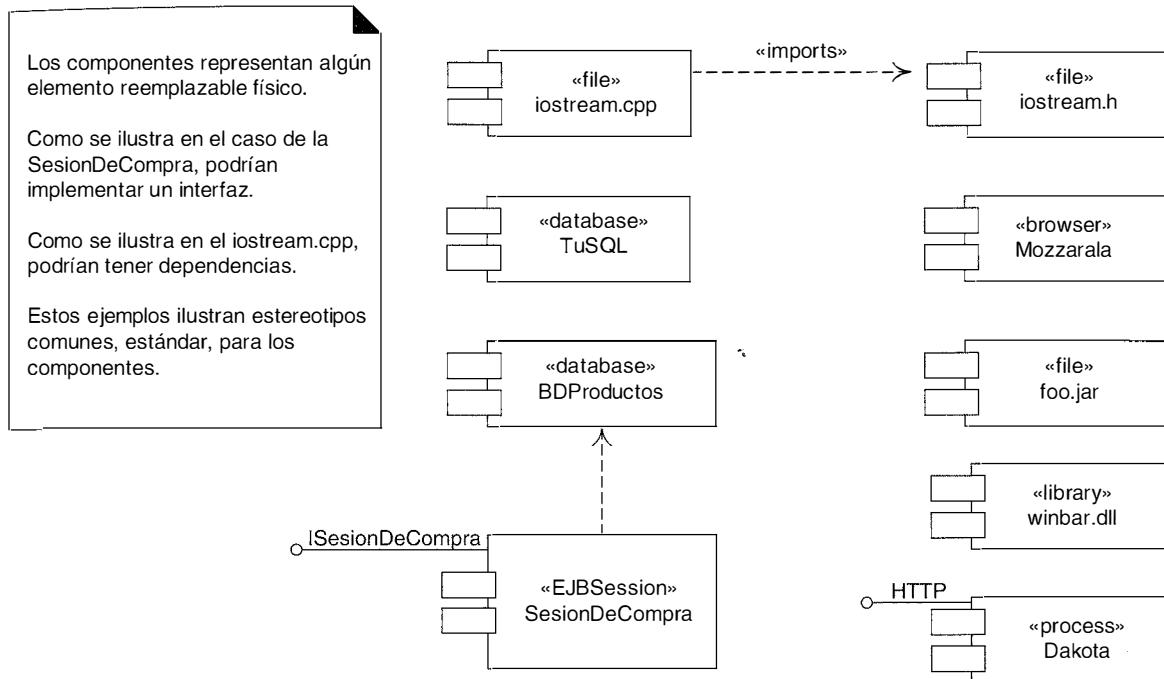


Figura 38.4. Componentes en UML.

Diagramas de despliegue

Un diagrama de despliegue muestra cómo se configuran las instancias de los componentes y los procesos para la ejecución run-time en las instancias de los **nodos** de proceso (algo con memoria y servicios de proceso; ver Figura 38.5).

38.3. Clase plantilla (parametrizada, genérica)

En la Figura 38.6 se muestran las clases plantilla y su instancia.

Algunos lenguajes, como C++, soportan las clases plantilla, genéricas o parametrizadas. Además, esta característica se añadirá al lenguaje Java. Por ejemplo, en C++, `bla<String, Persona>` declara la instanciación de la clase plantilla con claves de tipo `String`, y valores de tipos `Persona`.

38.4. Diagramas de actividades

Un **diagrama de actividades** de UML ofrece una notación rica para representar una secuencia de actividades. Podría aplicarse a cualquier propósito (como para mostrar los pasos de un algoritmo), pero se considera especialmente útil para visualizar los flujos de

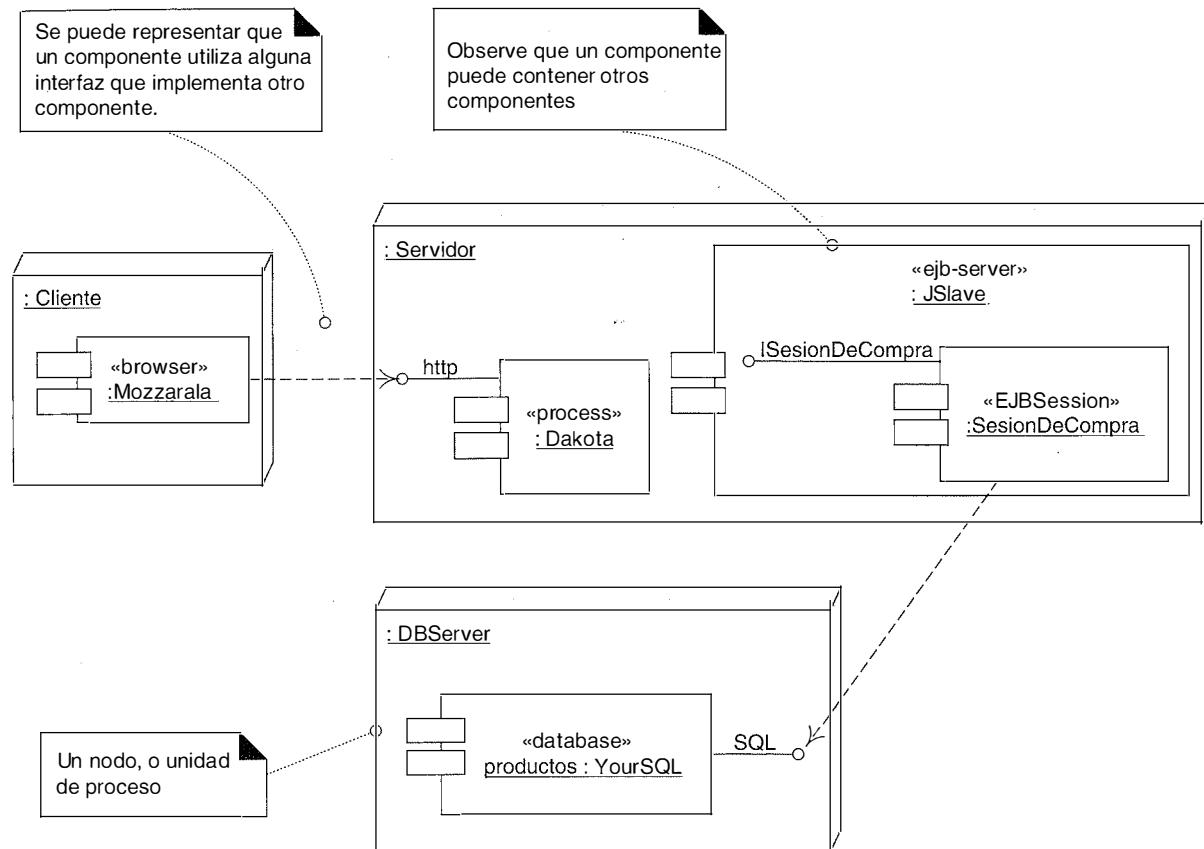


Figura 38.5. Un diagrama de despliegue.

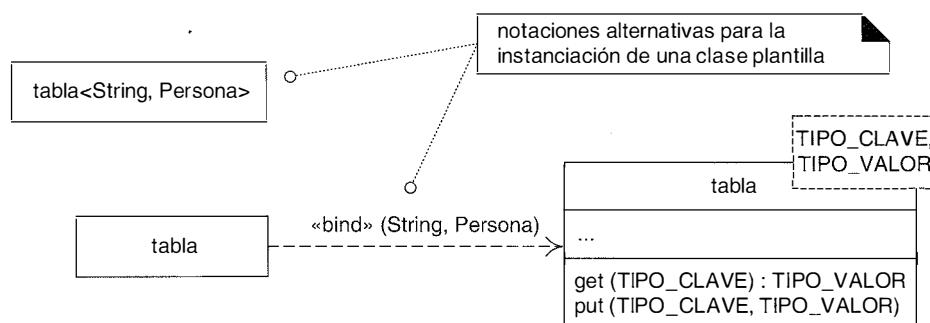


Figura 38.6. Clases plantilla.

trabajo y los procesos del negocio, o casos de uso. Uno de los flujos de trabajo (disciplinas) del UP es el **Modelado del Negocio**; su propósito es entender y comunicar “la estructura y la dinámica de la organización en el que se va a desplegar un sistema” [RUP]. Un artefacto clave de la disciplina del Modelado del Negocio es el **Modelo de Objetos del Negocio** (un superconjunto del Modelo del Dominio del UP), que visualiza esencialmente cómo funciona un negocio utilizando diagramas de clases, secuencia y actividades de UML. De esta manera, los diagramas de actividades se aplican especialmente dentro de la disciplina del Modelado del Negocio del UP.

Entre la notación destacada encontramos actividades concurrentes, calles y relaciones de flujo acción-objeto, como se ilustra en la Figura 38.7 (adaptado a partir de [OMG01, FS00]). Formalmente, un diagrama de actividad se considera un tipo especial de diagrama de estados de UML en el que los estados son acciones, y las transiciones de los eventos se disparan automáticamente al completarse la acción.

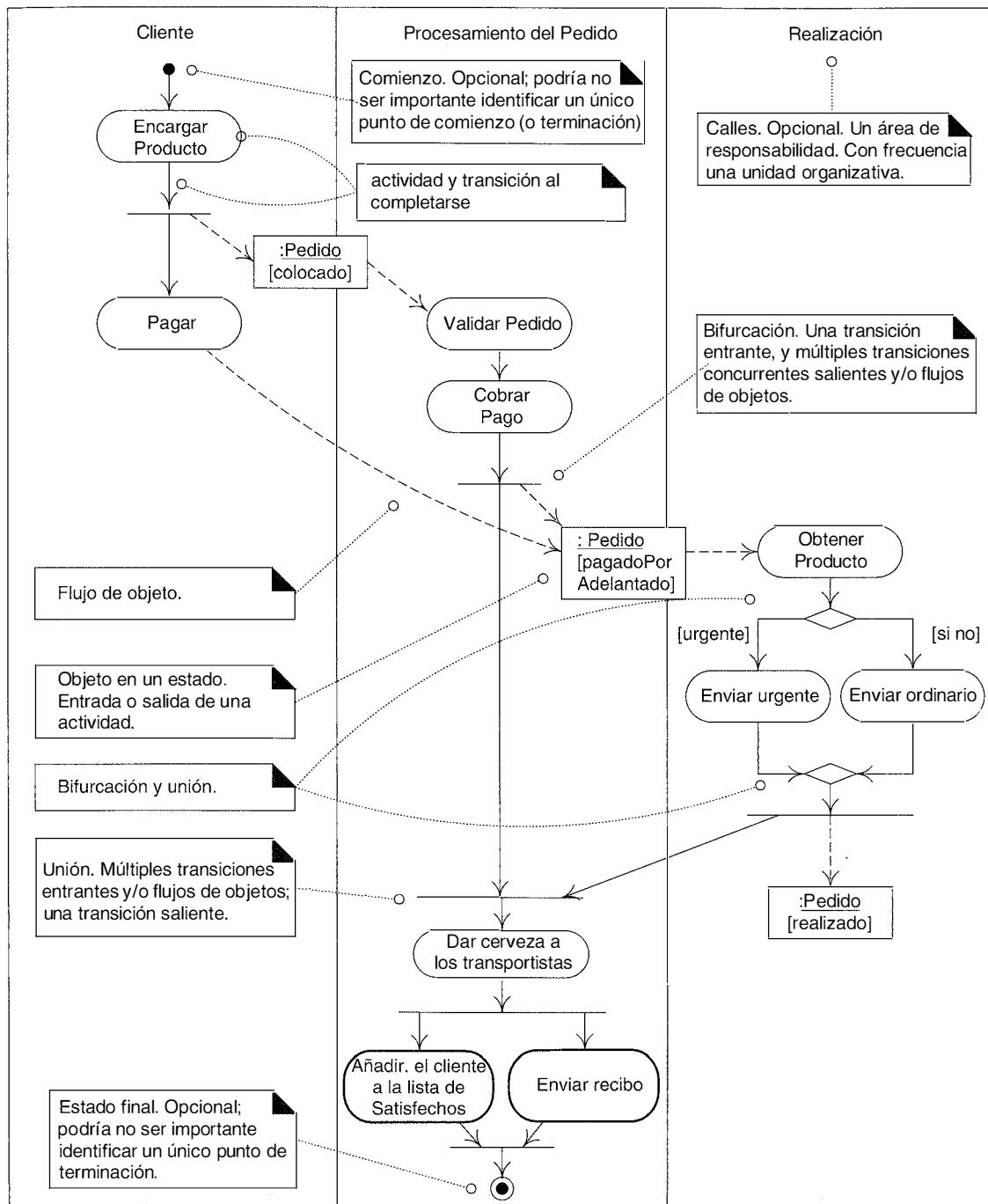


Figura 38.7. Diagrama de actividades.

Bibliografía

- Abbot83 Abbott, R. 1983. Program Design by Informal English Descriptions. *Communications of the ACM* vol. 26(11).
- AIS77 Alexander, C., Ishikawa, S., and Silverstein, M. 1977. *A Pattern Language—Towns-Building-Construction*. Oxford University Press.
- Ambler00 Ambler, S. 2000. *The Unified Process—Elaboration Phase*. Lawrence, KA.: R&D Books
- Ambler00a Ambler, S., Constantine, L. 2000. Enterprise-Ready Object IDs. *The Unified Process—Construction Phase*. Lawrence, KA.: R&D Books
- Ambler00b Ambler, S. 2000. Whitepaper: *The Design of a Robust Persistence Layer For Relational Databases*. www.amblysoft.com.
- BDSSS00 Beedle, M., Devos, M., Sharon, Y., Schwaber, K., and Sutherland, J. 2000. SCRUM: A Pattern Language for Hyperproductive Software Development. *Pattern Languages of Program Design* vol. 4. Reading, MA.: Addison-Wesley
- BC87 Beck, K., and Cunningham, W. 1987. *Using Pattern Languages for Object-Oriented Programs*. Tektronix Technical Report No. CR-87-43.
- BC89 Beck, K., and Cunningham, W. 1989. A Laboratory for Object-oriented Thinking. *Proceedings of OOPSLA 89*. SIGPLAN Notices, Vol. 24, No. 10.
- BCK98 Bass, L., Clements, P., and Kazman, R. *Software Architecture in Practice*. Reading, MA.: Addison-Wesley.
- Beck94 Beck, K. 1994. Patterns and Software Development. *Dr. Dobbs Journal*. Feb 1994.
- Beck00 Beck, K. 2000. *Extreme Programming Explained—Embrace Change*. Reading, MA.: Addison-Wesley.
- BF00 Beck, K., Fowler, M., 2000. *Planning Extreme Programming*. Reading, MA.: Addison-Wesley.
- BJ78 Bjørner, D., and Jones, C. editors. 1978. The Vienna Development Method: The Meta-Language, *Lecture Notes in Computer Science*. vol. 61. Springer-Verlag.
- BJR97 Booch, G., Jacobson, I., and Rumbaugh, J. 1997. The UML specification documents. Santa Clara, CA.: Rational Software Corp. See documents at www.rational.com.
- BMRSS96 Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P.,and Stal, M. 1996. *Pattern-Oriented Software Architecture: A System of Patterns*. West Sussex, England: Wiley.
- Boehm88 Boehm, B. 1988. A Spiral Model of Software Development and Enhancement. *IEEE Computer*. May 1988.

- Boehm00+ Boehm, B., et al. 2000. *Software Cost Estimation with COCOMO II*. Englewood Cliffs, NJ.: Prentice-Hall.
- Booch94 Booch, G., 1994. *Object-Oriented Analysis and Design*. Redwood City, CA.: Benjamin/Cummings.
- Booch96 Booch, G., 1996. *Object Solutions: Managing the Object-Oriented Project*. Menlo Park, CA.: Addison-Wesley.
- BP88 Boehm, B., and Papaccio, P. 1988. Understanding and Controlling Software Costs. *IEEE Transactions on Software Engineering*. Oct 1988.
- BRJ99 Booch, G., Rumbaugh, J., and Jacobson, I., . 1999. *The Unified Modeling Language User Guide*. Reading, MA.: Addison-Wesley.
- Brooks75 Brooks, F., 1975. *The Mythical Man-Month*. Reading, MA.: Addison-Wesley.
- Brown01 Brown, K., 2001. The *Convert Exception* pattern is found online at the Portland Pattern Repository, <http://c2.com>.
- BW95 Brown, K., and Whitenack, B. 1995. *Crossing Chasms, A Pattern Language for Object-RDBMS Integration*, White Paper, Knowledge Systems Corp.
- BW96 Brown, K., and Whitenack, B. 1996. Crossing Chasms. *Pattern Languages of Program Design* vol. 2. Reading, MA.: Addison-Wesley.
- CD94 Cook, S., and Daniels, J. 1994. *Designing Object Sysetms*. Englewood Cliffs, NJ.: Prentice-Hall.
- CDL99 Coad, P., De Luca, J., Lefebvre, E. 1999. *Java Modeling in Color with UML*. Englewood Cliffs, NJ.: Prentice-Hall.
- CL99 Constantine, L., and Lockwood, L. 1999. *Software for Use: A Practical Guide to the Models and Methods of Usage-Centered Design*. Reading, MA.: Addison-Wesley.
- CMS74 Constantine, L., Myers, G., and Stevens, W. 1974. Structured Design. *IBM Systems Journal*, vol. 13 (No. 2, 1974), pp. 115-139.
- Coad95 Coad, P. 1995. *Object Models: Stategies, Patterns and Applications*. Englewood Cliffs, NJ.: Prentice-Hall.
- Cockburn92 Cockburn, A. 1992.. Using Natural Language as a Metaphoric Basis for Object-Oriented Modeling and Programming. *IBM Technical Report TR-36.0002*, 1992.
- Cockburn97 Cockburn, A. 1997. Structuring Use Cases with Goals. *Journal of Object-Oriented Programming*, Sep-Oct, and Nov-Dec. SIGS Publications.
- Cockburn01 Cockburn, A. 2001. *Writing Effective Use Cases*. Reading, MA.: Addison-Wesley.
- Coleman+94 Coleman, D., et al. 1994. *Object-Oriented Development: The Fusion Method*. Englewood Cliffs, NJ.: Prentice-Hall.
- Constantine68 Constantine, L. 1968. Segmentation and Design Strategies for Modular Programming. In Barnett and Constantine (eds.), *Modular Programming: Proceedings of a National Symposium*. Cambridge, MA.: Information & Systems Press.
- Constantine94 Constantine, L. 1994. Essentially Speaking. *Software Development* May. CMP Media.
- Conway58 Conway, M. 1958. Proposal for a Universal Computer-Oriented Language. *Communications of the ACM*. 5-8 Volume 1, Number 10, October.
- Coplien95 Coplien, J. 1995. *The History of Patterns*. See <http://c2.com/cgi/wiki?HistoryOfPatterns>.
- Coplien95a Coplien, J. 1995. A Generative Development-Process Pattern Language. *Pattern Languages of Program Design* vol. 1. Reading, MA.: Addison-Wesley.

- CS95 Coplien, J., and Schmidt, D. eds. 1995. *Pattern Languages of Program Design* vol. 1. Reading, MA.: Addison-Wesley.
- Cunningham96 Cunningham, W. 1996. EPISODES: A Pattern Language of Competitive Development. *Pattern Languages of Program Design* vol. 2. Reading, MA.: Addison-Wesley.
- Cutter97 Cutter Group. 1997. *Report: The Corporate Use of Object Technology*.
- CV65 Corbato, F., and Vyssotsky, V. 1965. Introduction and overview of the Multics system. *AFIPS Conference Proceedings* 27, 185-196.
- Dijkstra68 Dijkstra, E. 1968. The Structure of the THE-Multiprogramming System. *Communications of the ACM*, 11(5).
- Eck95 Eck, D. 1995. *The Most Complex Machine*. A K Pators Ltd.
- Fowler96 Fowler, M. 1996. *Analysis Patterns: Reusable Object Models*. Reading, MA.: Addison-Wesley.
- Fowler00 Fowler, M. 2000. Put Your Process on a Diet. *Software Development*. December. CMP Media.
- Fowler01 Fowler, M. 2001. Draft patterns on object-relational persistence services. www.martin-fowler.com.
- FS00 Fowler, M., and Scott, K. 2000. *UML Distilled*. Reading, MA.: Addison-Wesley.
- Gartner95 Schulte, R., 1995. *Three-Tier Computing Architectures and Beyond*. Published Report Note R-401-134. Gartner Group.
- Gemstone00 Gemstone Corp., 2000. A set of architectural patterns at www.javasuccess.com.
- GHJV95 Gamma, E., Helm, R., Johnson, R., and Vlissides, J. 1995. *Design Patterns*. Reading, MA.: Addison-Wesley.
- Gilb88 Gilb, T. 1988. *Principles of Software Engineering Management*. Reading, MA.: Addison-Wesley.
- GK00 Guiney, E., and Kulak, D. 2000. *Use Cases: Requirements in Context*. Reading, MA.: Addison-Wesley.
- GK76 Goldberg, A., and Kay, A. 1976. *Smalltalk-72 Instruction Manual*. Xerox Palo Alto Research Center.
- GL00 Guthrie, R., and Larman, C. 2000. *Java 2 Performance and Idiom Guide*. Englewood Cliffs, NJ.: Prentice-Hall.
- Grady92 Grady, R. 1992. *Practical Software Metrics for Project Management and Process Improvement*. Englewood Cliffs, NJ.: Prentice-Hall.
- Grosso00 Grosso, W., 2000. *The Name The Problem Not The Thrower* exceptions pattern is found online at the Portland Pattern Repository, <http://c2.com>.
- GW89 Gause, D., and Weinberg, G. 1989. *Exploring Requirements*. NY, NY.: Dorset House.
- Harrison98 Harrison, N., 1998. Patterns for Logging Diagnostic Messages. *Pattern Languages of Program Design* vol. 3. Reading, MA.: Addison-Wesley.
- Hay96 Hay, D. 1996. *Data Model Patterns: Conventions of Thought*. NY, NY.: Dorset House.
- Highsmith00 Highsmith, J. 2000. *Adaptive Software Development: A Collaborative Approach to Managing Complex Systems*. NY, NY.: Dorset House.
- HNS00 Hofmeister, C., Nord, R., and Soni, D. 2000. *Applied Software Architecture*. Reading, MA.: Addison-Wesley.

- Jackson95 Jackson, M. 1995. *Software Requirements and Specification*. NY, NY.: ACM Press.
- Jacobson92 Jacobson, I., et al. 1992. *Object-Oriented Software Engineering: A Use Case Driven Approach*. Reading, MA.: Addison-Wesley.
- JAH00 Jeffries, R., Anderson, A., Hendrickson, C. 2000. *Extreme Programming Installed*. Reading, MA.: Addison-Wesley.
- JBR99 Jacobson, I., Booch, G., and Rumbaugh, J. 1999. *The Unified Software Development Process*. Reading, MA.: Addison-Wesley.
- Jones97 Jones, C., 1997. *Applied Software Measurement*. NY, NY.: McGraw-Hill.
- Jones98 Jones, C. 1998. *Estimating Software Costs*. NY, NY.: McGraw-Hill.
- Kay68 Kay, A. 1968. *FLEX, a flexible extensible language*. M.Sc. thesis, Electrical Engineering, University of Utah. May. (Univ. Microfilms).
- Kovitz99 Kovitz, B. 1999. *Practical Software Requirements*. Greenwich, CT.: Manning.
- Kruchten00 Kruchten, P. 2000. *The Rational Unified Process—An Introduction*. 2nd edition. Reading, MA.: Addison-Wesley.
- Kruchten95 Kruchten, P. 1995. The 4+1 View Model of Architecture. *IEEE Software* 12(6).
- Lakos96 Lakos, J. 1996. *Large-Scale C++ Software Design*. Reading, MA.: Addison-Wesley.
- Lieberherr88 Lieberherr, K., Holland, I., and Riel, A. 1988. Object-Oriented Programming: An Objective Sense of Style. *OOPSLA 88 Conference Proceedings*. NY, NY.: ACM SIGPLAN.
- Liskov88 Liskov, B. 1988. Data Abstraction and Hierarchy, *SIGPLAN Notices*, 23,5 (May, 1988).
- LW00 Leffingwell, D., and Widrig, D. 2000. *Managing Software Requirements: A Unified Approach*. Reading, MA.: Addison-Wesley.
- MacCormack01 MacCormack, A. 2001. Product-Development Practices That Work. *MIT Sloan Management Review*. Volume 42, Number 2.
- Martin95 Martin, R. 1995. *Designing Object-Oriented C++ Applications Using the Booch Method*. Englewood Cliffs, NJ.: Prentice-Hall.
- McConnell96 McConnell, S. 1996. *Rapid Development*. Redmond, WA.: Microsoft Press.
- MO95 Martin, J., and Odell, J. 1995. *Object-Oriented Methods: A Foundation*. Englewood Cliffs, NJ.: Prentice-Hall.
- Moreno97 Moreno, A.M. Object Oriented Analysis from Textual Specifications. *Proceedings of the 9th International Conference on Software Engineering and Knowledge Engineering*, Madrid, June 17-20 (1997).
- MP84 McMenamin, S., and Palmer, J. 1984. *Essential Systems Analysis*. Englewood Cliffs, NJ.: Prentice-Hall.
- MW89 1989. *The Merriam-Webster Dictionary*. Springfield, MA.: Merriam-Webster.
- Nixon90 Nixon, R. 1990. *Six Crisis*. NY, NY.: Touchstone Press.
- OMG01 Object Management Group, 2001. *OMG Unified Modeling Language Specification*. www.omg.org.
- Parkinson58 Parkinson, N. 1958. *Parkinson's Law: The Pursuit of Progress*, London, John Murray.
- Parnas72 Parnas, D. 1972. On the Criteria To Be Used in Decomposing Systems Into Modules, *Communications of the ACM*, Vol. 5, No. 12, December 1972. ACM.
- PM92 Putnam, L., and Myers, W. 1992. *Measures for Excellence: Reliable Software on Time, Within Budget*. Yourdon Press.

- Pree95 Pree, W. 1995. *Design Patterns for Object-Oriented Software Development*. Reading, MA.: Addison-Wesley.
- Renzel97 Renzel, K., 1997. *Error Handling for Business Information Systems: A Pattern Language*. Online at <http://www.objectarchitects.de/arcus/cookbook/exhandling/>
- Rising00 Rising, L. 2000. *Pattern Almanac 2000*. Reading, MA.: Addison-Wesley.
- RJB99 Rumbaugh, J., Jacobson, I., and Booch, G. 1999. *The Unified Modeling Language Reference Manual*. Reading, MA.: Addison-Wesley.
- Ross97 Ross, R. 1997. *The Business Rule Book: Classifying, Defining and Modeling Rules*. Business Rule Solutions Inc.
- Royce70 Royce, W. 1970. Managing the Development of Large Software Systems. *Proceedings of IEEE WESCON*. Aug 1970.
- Rumbaugh91 Rumbaugh, J., et al. 1991. *Object-Oriented Modelling and Design*. Englewood Cliffs, NJ.: Prentice-Hall.
- RUP The Rational Unified Process Product. The browser-based online documentation for the RUP, sold by Rational Corp.
- Rumbaugh97 Rumbaugh, J. 1997. Models Through the Development Process. *Journal of Object-Oriented Programming* May 1997. NY, NY: SIGS Publications.
- Shaw96 Shaw, M. 1996. Some Patterns for Software Architectures. *Pattern Languages of Program Design* vol. 2. Reading, MA.: Addison-Wesley.
- Standish94 Jim Johnson. 1994. *Chaos: Charting the Seas of Information Technology*. Published Report. The Standish Group
- SW98 Schneider, G., and Winters, J. 1998. *Applying Use Cases: A Practical Guide*. Reading, MA.: Addison-Wesley.
- TK78 Tsichiritzis, D., and Klug, A. The ANSI/X3/SPARC DBMS framework: Report of the study group on database management systems. *Information Systems*, 3 1978.
- Tufte92 Tufte, E. 1992. *The Visual Display of Quantitative Information*. Graphics Press.
- VCK96 Vlissides, J., et al. 1996. *Patterns Languages of Program Design* vol. 2. Reading, MA.: Addison-Wesley.
- Wirfs-Brock93 Wirfs-Brock, R. 1993. Designing Scenarios: Making the Case for a Use Case Framework. *Smalltalk Report* Nov-Dec 1993. NY, NY: SIGS Publications.
- WK99 Warmer, J., and Kleppe, A. 1999. *The Object Constraint Language: Precise Modeling With UML*. Reading, MA.: Addison-Wesley.
- WWW90 Wirfs-Brock, R., Wilkerson, B., and Wiener, L. 1990. *Designing Object-Oriented Software*. Englewood Cliffs, NJ.: Prentice-Hall.

Glosario

- abstracción** El acto de reunir las cualidades esenciales o generales de cosas similares. También, las características esenciales provenientes de una cosa.
- acoplamiento** Una dependencia entre elementos (como clases, paquetes, subsistemas), típicamente resultado de la colaboración entre los elementos para proporcionar un servicio.
- agregación** Una propiedad de una asociación que representa una relación todo-parte y (normalmente) control del tiempo de vida.
- análisis** Un estudio de un dominio que da como resultado los modelos que describen sus características estáticas y dinámicas. Se centra en las cuestiones del “qué” en lugar del “cómo”.
- análisis orientado a objetos** El estudio de un dominio del problema o sistema en función de los conceptos del dominio, como las clases conceptuales, asociaciones y cambios de estado.
- arquitectura** Informalmente, una descripción de la organización, motivación y estructura de un sistema. Están implicados muchos niveles diferentes de arquitecturas en el desarrollo de los sistemas software, desde la arquitectura hardware física a la arquitectura lógica de un framework de aplicación.
- asociación** Una descripción de un conjunto de enlaces relacionados entre objetos de dos clases.
- asociación calificada** Una asociación cuyos miembros se partitionan según el valor del calificador.
- asociación recursiva** Una asociación donde la fuente y el destino son la misma clase de objeto.
- atributo** Una característica o propiedad de una clase a la que se le asigna un nombre.
- atributo de clase** Una característica o propiedad que es la misma para todas las instancias de una clase. Esta información se almacena normalmente en la definición de la clase.
- clase** En UML, “El descriptor de un conjunto de objetos que comparten los mismos atributos, métodos, relaciones y comportamiento” [RJB99]. Podría utilizarse para representar elementos del software o conceptuales.

clase abstracta Una clase que se puede utilizar sólo como superclase de alguna otra clase; no se puede crear ningún objeto de una clase abstracta salvo como instancia de una subclase.

clase concreta Una clase que puede tener instancias.

clase contenedora Una clase designada para guardar y manipular una colección de objetos.

clasificación La clasificación define una relación entre una clase y sus instancias. La correspondencia de la clasificación identifica la extensión de una clase.

colaboración Dos o más objetos que participan en una relación cliente/servidor para proporcionar un servicio.

composición La definición de una clase en la que cada instancia consta de otros objetos.

concepto Una categoría de ideas o cosas. En este libro, se usa para designar cosas del mundo real en lugar de entidades del software. La intención de un concepto es una descripción de sus atributos, operaciones y semántica. La extensión de un concepto es el conjunto de instancias u objetos de ejemplo que son miembros del concepto. A menudo se define como sinónimo de clase del dominio.

constructor Un método especial que se invoca en el momento de la creación de una instancia de una clase en C++ o en Java. Con frecuencia el constructor realiza acciones de inicialización.

contrato Define las responsabilidades y postcondiciones que se aplican al uso de una operación o método. También se utiliza para referirse al conjunto de todas las condiciones relacionadas con una interfaz.

delegación La noción de que un objeto puede emitir un mensaje a otro objeto como respuesta a un mensaje. El primer objeto, por tanto, delega la responsabilidad en el segundo objeto.

derivación El proceso de definir una nueva clase referenciando a una clase existente y después añadiendo atributos y métodos. La clase existente es la superclase; nos referimos a la nueva clase como la subclase o clase derivada.

diseño Un proceso que utiliza los productos del análisis para producir una especificación para implementar un sistema. Una descripción lógica de cómo trabajará un sistema.

diseño orientado a objetos La especificación de una solución software lógica en función de objetos software, como clases, atributos, métodos y colaboraciones.

dominio Una delimitación formal que define una materia o un área de interés específica.

encapsulación Un mecanismo que se utiliza para ocultar los datos, la estructura interna y los detalles de implementación de algunos elementos, como un objeto o un subsistema. Todas las interacciones con un objeto se realizan a través de una interfaz pública de operaciones.

enlace Una conexión entre dos objetos; una instancia de una asociación.

estado La condición de un objeto entre eventos.

evento Una ocurrencia relevante.

extensión El conjunto de objetos a los que se aplica un concepto. Los objetos de la extensión son los ejemplos o instancias de los conceptos.

framework Un conjunto de clases abstractas y concretas que colaboran entre sí y que se pueden utilizar como plantilla para solucionar una familia de problemas relacionados. Normalmente se extiende por medio de la definición de subclases para el comportamiento específico de una aplicación.

generalización La actividad de identificar elementos en común entre conceptos y definir las relaciones de una superclase (concepto general) y subclase (concepto especializado). Es una manera de construir clasificaciones taxonómicas entre conceptos que entonces se representan en jerarquías de clases. Las subclases conceptuales son conformes con las superclases conceptuales en cuanto a la intensión y extensión.

herencia Una característica de los lenguajes de programación orientados a objetos mediante la cual se podrían especializar las clases a partir de superclases más generales. La subclase adquiere automáticamente las definiciones de los atributos y métodos de las superclases.

identidad de objeto La característica de que la existencia de un objeto es independiente de cualquier valor asociado con el objeto.

instancia Un miembro individual de una clase. En UML se denomina objeto.

instanciación La creación de una instancia de una clase.

intensión La definición de un concepto.

interfaz Un conjunto de signaturas de operaciones públicas.

jerarquía de clases Una descripción de las relaciones de herencia entre las clases.

lenguaje de programación orientado a objetos Un lenguaje de programación que soporta los conceptos de encapsulación, herencia, y polimorfismo.

mensaje Mecanismo por medio del cual se comunican los objetos; normalmente una solicitud de ejecución de un método.

metamodelo Un modelo que define otros modelos. El metamodelo de UML define los tipos de elementos de UML, como Clasificador.

método En UML, la implementación específica o algoritmo de una operación para una clase. Informalmente, el procedimiento software que se puede ejecutar como respuesta a un mensaje.

método de clase Un método que define el comportamiento de la propia clase, a diferencia del comportamiento de sus instancias.

método de instancia Un método cuyo alcance es una instancia. Se invoca enviando un mensaje a una instancia.

modelo Una descripción de las características estáticas y/o dinámicas de un área de estudio, descritas mediante una serie de vistas (normalmente diagramas o texto).

multiplicidad El número de objetos que se permite que participen en una asociación.

objeto En UML, una instancia de una clase que encapsula estado y comportamiento.

Más informalmente, un ejemplo de una cosa.

objeto activo Un objeto con su propio hilo de control.

objeto persistente Un objeto que puede sobrevivir al proceso o hilo de ejecución que lo crea. Un objeto persistente existe hasta que se elimina explícitamente.

OID Identificador de objeto.

operación En UML, “una especificación de una transformación o consulta que se puede invocar para que la ejecute un objeto” [RJB99]. Una operación tiene una firma, especificada por el nombre y los parámetros, y se invoca por medio de un mensaje. Un método es la implementación de una operación con un algoritmo específico.

operación polimórfica La misma operación implementada de manera diferente por dos o más clases.

patrón Un patrón es una descripción de un problema, la solución, cuándo aplicar la solución y el modo de aplicar la solución en contextos nuevos, al que se le asigna un nombre.

persistencia El almacenamiento permanente de un objeto.

polimorfismo El concepto de que dos o más clases de objetos pueden responder al mismo mensaje de formas diferentes, utilizando operaciones polimórficas. También, la capacidad de definir operaciones polimórficas.

postcondición Una restricción que debe ser verdad tras terminar una operación.

precondición Una restricción que debe ser verdad antes de que se solicite una operación.

privado Un mecanismo de alcance que se utiliza para restringir el acceso a los miembros de la clase de manera que otros objetos no pueden verlos. Normalmente se aplica a todos los atributos y a algunos métodos.

público Un mecanismo de alcance que se utiliza para hacer que los miembros sean accesibles a otros objetos. Normalmente se aplica a algunos métodos, pero no a los atributos, puesto que los atributos públicos violan la encapsulación.

receptor El objeto al que se le envía un mensaje.

responsabilidad Un servicio o grupo de servicios de hacer o conocer, proporcionados por un elemento (como una clase o subsistema); una responsabilidad contiene uno o más de los objetivos u obligaciones de un elemento.

restricción Una limitación o condición sobre un elemento.

rol El extremo de una asociación al que se le asigna un nombre para indicar su propósito.

subclase Una especialización de otra clase (la superclase). Una subclase hereda los atributos y métodos de la superclase.

subtipo Una subclase conceptual. Una especialización de otro tipo (el supertipo) que es conforme con la intención y extensión del supertipo.

superclase Una clase a partir de la cual otras clases heredan atributos y métodos.

supertipo Una superclase conceptual. En una relación generalización-especialización, el tipo más general; un objeto que tiene subtipos.

transición Una relación entre estados que tiene lugar si ocurre el evento especificado y se cumple la condición de guarda.

transición de estado Un cambio de estado de un objeto; algo que se puede indicar mediante un evento.

valores de datos puros Tipos de datos para los que no es significativa la identidad de instancia única, como los números, booleanos y las cadenas de texto.

variable de instancia Como se utiliza en Java y en Smalltalk, un atributo de una instancia.

visibilidad La capacidad de ver o tener referencia a un objeto.

Índice alfabético

A

acoplamiento, 214
actor, 67
 apoyo, 67
 negocio, 72
 pasivo, 67
 principal, 67
adaptador, 322
adopción incremental del proceso, 551
agregación, 386
 compartida, 388
 de composición, 385, 387
alta cohesión, 217
análisis, 6
 definición, 6
 ejemplo del juego de dados, 7
estructurado, 125
orientado a objetos, 6
y diseño orientado a objetos
arquitectura, 418, 452
 análisis de la, 418, 452
base de la, 105
decisiones sobre la, 453
diseño de la, 418
ejecutable, 105
en capas, 420
factores de la, 453
intereses transversales, 464
investigación de la, 418
memorándums técnicos, 460
patrones de, 419
principios de diseño de, 463
promoción de factores de, 466

prototipo, 105
prueba-de-conceptos de la, 471
separación de intereses, 464
síntesis, 471
software, 418
tabla de factores, 456
tarjetas de cuestiones, 463
tipos, 454
tres niveles, 438
vista, 468
 casos de uso, 469
 datos, 469
 despliegue, 469
 implementación, 469
 lógica, 468
 proceso, 468
artefacto
 especificación Complementaria, 79
 glosario, 94
 visión, 79
artefactos, 20
 organización, 549
asociación, 145
 calificada, 393
 centrarse en asociaciones necesito-conocer, 154
 criterios para asociaciones útiles, 146
 de prioridad alta, 148
 enlace, 190
 guías, 148
 localización con lista, 147
 múltiples asociaciones entre tipos, 152
 multiplicidad, 149
 navegabilidad, 273

- nivel de detalle, 150
 - nombres, 151
 - de los roles, 390
 - notación UML, 146
 - reflexiva, 394
 - atributo, 157
 - derivado, 164
 - no claves ajena, 162
 - notación UML, 158
 - referencia, 285
 - simple, 158
 - tipo de dato, 159
 - tipos
 - no primitivos, 160
 - tipos válidos, 158
 - y cantidades, 162
 - atributos de calidad, 41
- B**
- bajo Acoplamiento, 214
 - beneficios del desarrollo iterativo, 17
- C**
- caja de activación, 195
 - calidad
 - verificación continua, 554
 - calificador, 393
 - capa de dominio, 325
 - característica del sistema, 92
 - caso de uso, 45
 - abstracto, 365
 - adicional, 366
 - base, 365
 - breve, 47
 - completo, 47
 - concreto, 364
 - cuándo crear casos de uso abstractos, 365
 - de caja negra, 46
 - del negocio, 72
 - del sistema, 72
 - diagrama de estados, 409
 - para, 409
 - estilo esencial, 66
 - extensión (*extend*), 365
 - inclusión (*include*), 363
 - informal, 47
 - instancia, 45
 - objetivo de usuario, 58
 - postcondición, 53
 - precondición, 52
 - proceso del negocio elemental, 57
 - y el proceso de desarrollo, 72
 - casos de Cambio, 318
 - centrado en la arquitectura, 554
 - ciclo de vida
 - en cascada, 557
 - iterativo, 14
 - mitigación de problemas del ciclo de vida en cascada, 558
 - mitigación de los problemas con el iterativo, 559
 - problemas, 558
 - clase
 - abstracta, 380, 381
 - activa, 479
 - asociación, 384, 385
 - conceptual, 137
 - & abstracta, 380
 - definiciones, 138
 - diagrama, 268
 - diseño, 137
 - en UML, 138
 - genérica, 569
 - implementación, 138
 - jerarquía, 382
 - notación UML, 189
 - parametrizada, 569
 - partición, 375
 - particionar, 376
 - plantilla, 569
 - significado UML, 137
 - software, 138
 - transformación a partir de DCD, 284
 - clases de implementación, 138
 - clasificador, 137
 - cohesión, 217
 - relacional, 444
 - colección
 - iteración sobre una colección en UML, 198
 - command, 499
 - componente, 568
 - comportamiento
 - clase, 202
 - sistema, 114
 - visión general, 114
 - composite, 337
 - compuesto, 385

- conceptos
 de especificación o descripción, 133
 similares, 132
 descubrimiento con identificación de nombres, 128
 error en la identificación, 131
 extensión, 124
 intensión, 124
 símbolo, 124
 versus rol, 391
 construcción, 555
 construcciones diarias, 557
 contenedor (Decorador), 465
 contrato
 descripción de las secciones, 169
 ejemplo, 168
 guías, 173
 postcondición, 169
 control de cambios, 555
 controlador (*Controler*), 221
 aplicación, 237
 saturado, 226
 convertir excepciones, 480
 correspondencia de esquemas, 504
 CRC, 229
 creador (*Creator*), 211
 aplicación, 238
 cruise control para integraciones continuas, 557
 cuestiones de planificación, 550
- D**
- DCD, 268
 definición de
 defecto, 479
 error, 479
 fallo, 479
 desarrollo
 adaptable, 16
 dirigido por
 casos de uso, 72
 el riesgo, 553
 beneficios, 17
 planificación, 539
 iterativo e incremental, 14
 descomposición
 de la representación, 310
 del comportamiento, 310
 diagrama
 de actividad, 569
 de clases de diseño, 268
 añadir métodos, 270
 DCD, 268
 ejemplo, 268
 información del tipo, 273
 mostrar
 navegabilidad, 273
 relaciones de dependencia, 276
 notación para los miembros, 277
 y multiobjetos, 272
 de colaboración, 186
 condicionales mutuamente exclusivos, 193
 creación de instancias, 191
 ejemplo, 187
 enlaces, 190
 iteración, 193
 sobre una colección, 194
 mensaje
 a self, 190
 a un objeto clase, 194
 mensajes, 195
 condicionales, 192
 número de secuencia, 191
 secuencia de mensajes, 191
 de componentes, 568
 de contexto, 69
 de despliegue, 569
 de estados, 408
 acciones de la transición, 413
 condiciones de guarda, 413
 ejemplo, 411
 estados anidados, 414
 para casos de uso, 409
 visión general, 407
 de implementación, 568
 de interacción
 clase, 189
 instancia, 189
 sintaxis de los mensajes, 189
 de secuencia, 186
 caja de activación, 195
 condicional mutuamente exclusivo, 198
 creación de instancias, 196
 del sistema, 114
 mostrar el texto del caso de uso, 118
 destrucción de objetos, 196
 iteración, 198
 sobre una colección, 198

sobre una serie de mensajes, 198
 líneas de vida, 196
 mensaje
 a self, 196
 a una clase, 199
 condicional, 197
 mensajes, 195
 retorno, 195
 dibujar diagramas, 532
 sugerencias, 533
 diccionario de datos, 41
 disciplina, 20
 de diseño, 20
 de entorno, 20
 de modelado del Negocio, 570
 de requisitos, 20
 y fases, 20
 y flujo de trabajo, 20
 diseño, 6
 de interfaz de usuario (UI), 563
 dirigido
 por los datos, 327
 por responsabilidades, 230
 especulativo, 532
 físico, 444
 por Contrato, 177
 diseños modulares, 220
 documento
 de la arquitectura del software (SAD), 467
 de técnicas de la arquitectura, 460

E

EBP, 57
 elaboración, 19
 enlace, 190
 escenario, 45
 de calidad, 455
 especialización, 371
 especificación
 de operación, 176
 de propiedades, 567
 estado, 407
 modelado, 381
 estereotipo, 69
 estilo de casos de uso esencial, 66
 estilos, 419
 estimación, 549
 estimar, 549
 estrategia (*Strategy*), 332

evento, 407
 de tiempo, 412
 del sistema, 222
 asignación de nombres, 117
 externo, 412
 interno, 412
 excepciones en UML, 481
 experto, 207
 aplicación, 240
 en información, 207
 extensión, 124

F

fabricación pura, 308
 factoría (*Factory*), 326
 abstracta (*Abstract Factory*), 490
 fachada (*Facade*), 346
 fases del UP, 19
 fijación de la duración, 18
 motivación, 557
 flujo de trabajo, 20
 y disciplina, 20
 focos de control, 195
 framework de persistencia, 502, 503
 de caja blanca, 509
 ideas claves, 504
 materialización, 510
 patrón
 identificador de objeto, 505
 manejo de la caché, 515
 representación de objetos como tablas, 504
 representación de relaciones en tablas, 524
 requisitos, 503

G

generalización, 372
 conformidad, 374
 notación
 para las clases abstractas, 381
 UML, 372
 partición, 375
 pruebas de validez de las subclases, 375
 visión general, 371
 y clases conceptuales, 373
 y conjuntos de clases conceptuales, 373
 guiones de casos de uso, 563

H

hacerlo Yo Mismo, 493
 herencia, 341
 herramientas CASE para UML, 535
 hilos en UML, 478

I

implementación, 20
 independiente del estado, 411
 indirección, 312
 ingeniería
 de usabilidad, 563
 directa, 535
 inversa, 536
 inicialización
 impaciente, 330
 perezosa, 330
 inicio, 19
 instancia
 notación UML, 184
 integración continua, 557
 cruise control, 557
 construcciones diarias, 557
 intensión, 124
 intereses transversales, 464
 interfaz, 307
 paquete, 568
 intervalos de tiempo, 390
 iteraciones, 14

J

jerarquía de clases, 371
 junit, 556

L

límite del sistema, 116

M

marco de desarrollo, 22
 memorándums técnicos, 460
 mensaje
 asíncrono, 481, 482
 notación UML, 191
 metadatos, 509
 método, 176

a partir del diagrama de colaboración, 287

de guarda, 514
 estáticos, 194
 plantilla (*Template Method*), 509
 sincronizados, 514

modelado

 de datos, 505
 visual, 555

modelo

 conceptual, 122
 de análisis, 563
 de casos de Uso, 43
 de datos, 505
 de delegación de eventos, 348
 de diseño, 182
 vs. Modelo de dominio, 269

de dominio, 122

 conceptos similares, 132
 encontrar conceptos, 127
 estrategia del cartógrafo, 130

modelado

 de lo irreal, 133
 de los cambios de estado, 381

 organización en paquetes, 396
 vocabulario del dominio, 123
 vs. Modelo de diseño, 269

de implementación, 444

de objetos del Negocio, 570

modelos

 de datos, 505
 de objetos
 de análisis, 122
 del dominio, 122

momento-intervalo, 390

multiobjeto, 194

multiplicidad, 149

N

navegabilidad, 273
 nivel (*tier*), 434
 de objetivo de usuario, 58
 nodos, 569
 nombre de camino, 426
 notas en UML, 243

O

objetivo
 de subfunción, 60

- de usuario, 58
 - objeto
 - activo, 478
 - del dominio inicial, 252
 - en UML, 189
 - líneas de vida, 196
 - persistente, 502
 - objetos
 - control, 225
 - data holder, 432
 - entidad, 225
 - frontera, 225
 - persistentes, 502
 - valor, 432
 - observador (*Observer*), 348
 - OCL, 176
 - ocultamiento de información, 319
 - operación del sistema, 222
 - operaciones, 176
 - del sistema, 167
 - organización de artefactos, 549
- P**
- paquete, 348
 - dependencias, 395
 - guías para la organización, 444
 - interfaz, 568
 - notación, 450
 - UML, 394
 - pertenencia, 395
 - referencia, 395
 - patrón, 4
 - adaptador (*Adapter*), 322
 - alta cohesión, 217
 - bajo acoplamiento, 214
 - capas (*Layers*), 420
 - command, 499
 - composite, 337
 - controlador, 221
 - convertir excepciones , 481
 - creador, 211
 - estado (*State*), 175
 - estrategia (*Strategy*), 332
 - experto, 207
 - fabricación pura, 308
 - factoría (*Factory*), 326
 - abstracta (*Abstract Factory*), 490
 - fachada (*Facade*), 346
 - hacerlo Yo Mismo, 493
 - indirección, 312
 - método plantilla (*Template Method*), 509
 - nombres, 205
 - observador (*Observer*), 348
 - polimorfismo, 306
 - para pagos, 493
 - Proxy, 484
 - de redirección, 485
 - remoto, 485
 - virtual, 522
 - publicar-suscribir, 348
 - separación modelo-vista, 239
 - Singleton, 328
 - variaciones protegidas, 313
 - patrones
 - de análisis, 127
 - de arquitectura, 419
 - de diseño, 419
 - de estilo, 419
 - de la “pandilla de los cuatro” (*Gang of Four*), 322
 - GRASP
 - alta cohesión, 217
 - bajo acoplamiento, 214
 - controlador, 221
 - creador, 211
 - experto, 207
 - fabricación pura, 308
 - indirección, 312
 - polimorfismo, 306
 - variaciones protegidas, 313
 - plan de
 - desarrollo de software, 107
 - fase, 24
 - iteración, 24
 - planificación
 - adaptable, 543
 - vs. Predictiva, 543
 - cuestiones de planificación, 550
 - iterativa, 539
 - polimorfismo, 306
 - postcondición, 169
 - en casos de uso, 53
 - una metáfora, 171
 - precondición
 - en casos de uso, 52
 - principio
 - abierto-Cerrado, 319
 - de Sustitución de Liskov, 315
 - Hollywood, 503

- priorización
 de los requisitos, 540
 de los riesgos, 543
- proceso
 adaptable, 24
 ágil, 24
 de desarrollo de software, 13
 del negocio elemental, 57
 iterativo, 14
 pesado, 23
 predictivo, 23
 unificado, 13
 de rational, 14
- programación
 extrema, 556
 orientada al aspecto, 465
- programar probando primero, 556
- Proxy, 485
 de redirección, 485
 remoto, 485
 virtual, 522
- PSL, 315
- publicar-suscribir, 348
- punto
 de evolución, 463
 de extensión, 366
 de variación, 318
- R**
- razonamiento visual, 531
- realizaciones de casos de uso, 72
- reglas del negocio, 86
- relación
 de dependencia, 568
 de extensión entre casos de uso (*extend*), 365
 de inclusión entre casos de uso (*include*), 362
- réplicas, 431
- requisitos, 39
 funcionales, 41
 en el modelo de casos de uso, 44
 gestión, 555
 no funcionales en la especificación complementaria, 80
 priorización, 540
 significativos para la arquitectura, 454
- traza, 546
- visión general, 39
- responsabilidades, 202
 conocer, 202
 hacer, 202
 importancia de, 6
 patrones, 204
 y diagramas de interacción, 203
 y métodos, 202
- restrictiones, 85
- retorno en diagramas de secuencia, 195
- reutilización, 565
- riesgo, 543
- rol, 149
 nombre, 285
 versus concepto, 391
- RUP, 14
- producto, 564
- S**
- SAD, 467
- salto en la representación, 138
- SCRUM, 556
- separación
 de intereses, 464
 modelo-vista, 239
- símbolo, 124
- Singleton, 328
 notación abreviada UML, 329
- solicitud de Cambio107
- subclase, 341
 conformidad, 374
 creación, 375
 partición, 376
 pruebas de validez, 375
- superclase
 creación, 377
- SWEBOK, 42
- T**
- tabla de factores, 457
- tarea de usuario, 57
- tarjetas de cuestiones, 460
- tipo de dato, 158
- transición, 556
- U**
- UML, 10
 herramientas CASE, 535

- modelado visual, 555
 - perfil, 506
 - de modelado de datos, 505
 - perfiles, 505
 - sugerencias para dibujar los diagramas, 532
 - visión general, 10
 - UP, 14
 - ágil, 24
 - buenas prácticas y conceptos, 553
 - fases, 19
 - usuarios comprometidos, 554
- atributo, 263
- global, 265
- local, 265
- parámetro, 263
- por defecto en UML, 278
- vista
- de casos de uso, 469
 - de datos, 469
 - de despliegue, 469
 - de implementación, 469
 - lógica, 454
 - proceso, 468

V

- variaciones protegidas, 313
- visibilidad, 262

X

- XP, 556

