

UML Y PATRONES

Introducción al análisis y diseño orientado a objetos



CRAIG LARMAN

UML Y PATRONES

INTRODUCCIÓN AL ANÁLISIS
Y DISEÑO ORIENTADO A OBJETOS

CRAIG LARMAN

Versión en español:

Luz María Henández Rodríguez
Traductora profesional

Con revisión técnica de:

Ing. Humberto Cárdenas Anaya
*Director de la Carrera de Ingeniería en Sistemas Computacionales,
Profesor del Departamento de Sistemas de Información,
Instituto Tecnológico y de Estudios Superiores de Monterrey, Campus Estado de México*

No. de Entrada 053681
27.3.01

PEARSON

PRENTICE
HALL

MÉXICO • ARGENTINA • BOLIVIA • BRASIL • COLOMBIA • COSTA RICA • CHILE • ECUADOR
EL SALVADOR • ESPAÑA • GUATEMALA • HONDURAS • NICARAGUA • PANAMÁ
PARAGUAY • PERÚ • PUERTO RICO • REPÚBLICA DOMINICANA • URUGUAY • VENEZUELA

AMSTERDAM • HARLOW • MIAMI • MUNICH • NUEVA DELHI • MENLO PARK • NUEVA JERSEY
NUEVA YORK • ONTARIO • PARÍS • SINGAPUR • SYDNEY • TOKIO • TORONTO • ZURICH

Addison
Wesley
Longman

Ejemplo de actividades de desarrollo

Planeación
y elaboración

Perfeccionamiento del plan

Definición de los requerimientos

Análisis

Diseño

Construcción

Pruebas

1. Definir el plan preliminar	2. Crear el reporte preliminar de investigación
3. Definir los requerimientos	4. Registrar los términos en el glosario
5. Implementar el prototipo	6. Definir los casos de uso (de alto nivel y esenciales)
7. Definir el modelo conceptual preliminar	8. Definir la arquitectura preliminar del sistema
9. Perfeccionar el plan	

1. Definir los casos esenciales de uso	2. Perfeccionar los diagramas de casos de uso
3. Perfeccionar el modelo conceptual	4. Perfeccionar el glosario
5. Definir los diagramas de secuencia del sistema	6. Definir los contratos de operaciones
7. Definir los diagramas de estado	

1. Definir los casos reales de uso	2. Definir reportes, interfaz de usuario y storyboards
3. Perfeccionar la arquitectura del sistema	4. Definir los diagramas de interacción
5. Definir los diagramas de clases del diseño	6. Definir los enunciados de la base de datos

Patrones generales de software para asignar responsabilidades (GRASP)

Patrón	Descripción
Experto	<p>¿Quién asumirá la responsabilidad en el caso general?</p> <p>Asignar una responsabilidad al experto en información: la clase que posee la información necesaria para cumplir con la responsabilidad.</p>
Creador	<p>¿Quién crea?</p> <p>Asignar a la clase B la responsabilidad de crear una instancia de clase A, si se cumple una de las siguientes condiciones:</p> <ul style="list-style-type: none"> 1. B contiene A 2. B agrega A 3. B tiene los datos de inicialización de A 4. B registra A 5. B utiliza A muy de cerca
Controlador	<p>¿Quién administra un evento del sistema?</p> <p>Asignar la responsabilidad de administrar un mensaje de eventos del sistema a una clase que represente una de las siguientes opciones:</p> <ul style="list-style-type: none"> 1. El negocio o la organización global (un controlador de fachada). 2. El "sistema" global (un controlador de fachada). 3. Un ser animado del dominio que realice el trabajo (un controlador de papeles). 4. Una clase artificial (Fabricación Pura) que represente el caso de uso (un controlador de casos de uso).
Bajo Acoplamiento (evaluativo)	<p>¿Cómo dar soporte a poca dependencia y a una mayor reutilización?</p> <p>Asignar las responsabilidades de modo que se mantenga bajo acoplamiento.</p>
Alta Cohesión (evaluativa)	<p>¿Cómo mantener controlable la complejidad?</p> <p>Asignar las responsabilidades de modo que se mantenga una alta cohesión.</p>
Polimorfismo	<p>¿Quién, cuándo el comportamiento varía según el tipo?</p> <p>Cuando varía el tipo (clase) de alternativas o comportamientos relacionados, asignar la responsabilidad del comportamiento —mediante operaciones polimórficas— a los tipos en que varía el comportamiento.</p>
Fabricación Pura	<p>¿Quién, cuándo uno está desesperado y no quiere violar los patrones Alta Cohesión ni Bajo Acoplamiento?</p> <p>Asignar un conjunto muy alto de cohesión de responsabilidades a una clase artificial que no represente nada en el dominio del problema, a fin de brindar soporte a una alta cohesión, a bajo acoplamiento y a la reutilización.</p>
Indirección	<p>¿Quién, para evitar el acoplamiento directo?</p> <p>Asignar la responsabilidad a un objeto intermedio para que medie entre otros componentes o servicios, de modo que no se acoplen directamente.</p>
No Hables con Extraños (ley de Demeter)	<p>¿Quién, para no conocer la estructura de los objetos indirectos?</p> <p>Asignar la responsabilidad al objeto directo del cliente para colaborar con el objeto indirecto, de modo que el cliente no necesita conocer el objeto indirecto. En el método, los mensajes únicamente pueden enviarse a los siguientes objetos:</p> <ul style="list-style-type: none"> • El objeto <i>this</i> (o el <i>self</i>). • Un parámetro del método. • Un atributo de <i>self</i>. • Un elemento de una colección que sea un atributo de <i>self</i>. • Un objeto creado dentro del método.

RESUMEN DE CONTENIDO

PARTE I INTRODUCCIÓN 1

- | | | |
|---|---|----|
| 1 | ANÁLISIS Y DISEÑO ORIENTADOS A OBJETOS | 3 |
| 2 | INTRODUCCIÓN A UN PROCESO DE DESARROLLO | 17 |
| 3 | DEFINICIÓN DE MODELOS Y ARTEFACTOS | 29 |

PARTE II FASE DE PLANEACIÓN Y DE ELABORACIÓN 33

- | | | |
|---|--|----|
| 4 | CASO DE ESTUDIO: EL PUNTO DE VENTA | 35 |
| 5 | CONOCIMIENTO DE LOS REQUERIMIENTOS | 39 |
| 6 | CASOS DE USO: DESCRIPCIÓN DE PROCESOS | 47 |
| 7 | CLASIFICACIÓN Y PROGRAMACIÓN DE LOS CASOS DE USO | 73 |
| 8 | INICIO DE UN CICLO DE DESARROLLO | 81 |

PARTE III FASE DE ANÁLISIS (1) 83

- | | | |
|----|--|-----|
| 9 | CONSTRUCCIÓN DE UN MODELO CONCEPTUAL | 85 |
| 10 | MODELO CONCEPTUAL: AGREGACIÓN DE LAS ASOCIACIONES | 105 |
| 11 | MODELO CONCEPTUAL: AGREGACIÓN DE LOS ATRIBUTOS | 119 |
| 12 | REGISTRO DE LOS TÉRMINOS EN EL GLOSARIO | 131 |
| 13 | COMPORTAMIENTO DE LOS SISTEMAS: DIAGRAMA DE LA SECUENCIA DEL SISTEMA | 135 |
| 14 | COMPORTAMIENTO DE LOS SISTEMAS: CONTRATOS | 145 |

PARTE IV FASE DE DISEÑO (1) 159

- | | | |
|----|--|-----|
| 15 | DEL ANÁLISIS AL DISEÑO | 161 |
| 16 | DESCRIPCIÓN DE LOS CASOS REALES DE USO | 163 |
| 17 | DIAGRAMAS DE COLABORACIÓN | 167 |
| 18 | GRASP: PATRONES PARA ASIGNAR RESPONSABILIDADES | 185 |
| 19 | DISEÑO DE UNA SOLUCIÓN CON OBJETOS Y PATRONES | 217 |
| 20 | DETERMINACIÓN DE LA VISIBILIDAD | 247 |
| 21 | DIAGRAMAS DE CLASES DEL DISEÑO | 255 |
| 22 | ALGUNOS ASPECTOS DEL DISEÑO DE SISTEMAS | 271 |

PARTE V FASE DE CONSTRUCCIÓN (1) 293

- 23 MAPEO DE LOS DISEÑOS PARA CODIFICACIÓN 295
24 SOLUCIÓN EN PROGRAMA DE JAVA 309

PARTE VI FASE DE ANÁLISIS (2) 315

- 25 ELECCIÓN DE LOS REQUERIMIENTOS DEL CICLO DE DESARROLLO 2 317
26 CÓMO RELACIONAR CASOS MÚLTIPLES DE USO 321
27 EXTENSIÓN DEL MODELO CONCEPTUAL 329
28 GENERALIZACIÓN 335
29 PAQUETES: ORGANIZACIÓN DE LOS ELEMENTOS 349
30 REFINAMIENTO DEL MODELO CONCEPTUAL 355
31 MODELO CONCEPTUAL: RESUMEN 367
32 COMPORTAMIENTO DE LOS SISTEMAS 373
33 MODELADO DEL COMPORTAMIENTO EN LOS DIAGRAMAS DE ESTADO 379

PARTE VII FASE DE DISEÑO (2) 391

- 34 GRASP: MÁS PATRONES PARA ASIGNAR RESPONSABILIDADES 393
35 DISEÑO CON MÁS PATRONES 405

PARTE VIII TEMAS ESPECIALES 425

- 36 OTRA NOTACIÓN DE UML 427
37 PROBLEMAS DEL PROCESO DE DESARROLLO 433
38 ESQUEMAS, PATRONES Y PERSISTENCIA 455

APÉNDICE A. LECTURAS RECOMENDADAS 487

APÉNDICE B. EJEMPLOS DE ACTIVIDADES Y MODELOS DE DESARROLLO 489

BIBLIOGRAFÍA 495

GLOSARIO 497

ÍNDICE 503

CONTENIDO

PARTE I INTRODUCCIÓN 1

- 1 ANÁLISIS Y DISEÑO ORIENTADOS A OBJETOS 3
1.1 Aplicación del lenguaje UML y del análisis y el diseño orientados a objetos 3
1.2 Asignación de responsabilidades 5
1.3 ¿Qué son el análisis y el diseño? 6
1.4 ¿Qué son el análisis y el diseño orientados a objetos? 6
1.5 Una analogía: organización de la empresa MicroChaos 7
1.6 Un ejemplo del análisis y del diseño orientados a objetos 10
1.7 Comparación entre el análisis y el diseño orientados a objetos y los diseños orientados a funciones 14
1.8 Advertencia: el "análisis" y el "diseño" pueden provocar guerras terminológicas 14
1.9 El Unified Modeling Language, UML 15

- 2 INTRODUCCIÓN A UN PROCESO DE DESARROLLO 17
2.1 Introducción 17
2.2 El lenguaje UML y los procesos de desarrollo 19
2.3 Pasos de macronivel 20
2.4 Desarrollo iterativo 20
2.5 La fase de la planeación y de la elaboración 23
2.6 La fase de construcción: ciclos del desarrollo 25
2.7 Decidir cuándo crear artefactos 26

- 3 DEFINICIÓN DE MODELOS Y ARTEFACTOS 29
3.1 Introducción 29
3.2 Sistemas de construcción de modelos 29
3.3 Modelos muestra 30
3.4 Relación entre los artefactos 31

PARTE II FASE DE PLANEACIÓN Y DE ELABORACIÓN 33

- 4 CASO DE ESTUDIO: EL PUNTO DE VENTA 35
4.1 El sistema del punto de venta 35
4.2 Capas arquitectónicas y el énfasis en el caso de estudio 36
4.3 Nuestra estrategia: aprendizaje y desarrollo iterativos 36

- 5 CONOCIMIENTO DE LOS REQUERIMIENTOS 39
5.1 Introducción 39
5.2 Los requerimientos 41
5.3 Presentación general 41
5.4 Clientes 41
5.5 Metas 42
5.6 Funciones del sistema 42
5.7 Atributos del sistema 44
5.8 Otros artefactos en la fase de los requerimientos 46

- 6 CASOS DE USO: DESCRIPCIÓN DE PROCESOS 47
6.1 Introducción 47
6.2 Actividades y dependencias 49

	6.3 Casos de uso 49		11	MODELO CONCEPTUAL: AGREGACIÓN DE LOS ATRIBUTOS 119
	6.4 Actores 52			11.1 Introducción 119
	6.5 Un error común en los casos de uso 53			11.2 Atributos 120
	6.6 Identificación de los casos de uso 53			11.3 Notación de los atributos en el UML 120
	6.7 Caso de uso y procesos del dominio 54			11.4 Tipos de atributos válidos 120
	6.8 Casos de uso, funciones del sistema y rastreabilidad 55			11.5 Tipos de atributos no primitivos 124
	6.9 Diagramas de los casos de uso 55			11.6 Modelado de cantidades y unidades de los atributos 125
	6.10 Formatos de los casos de uso 55			11.7 Atributos del sistema del punto de venta 126
	6.11 Los sistemas y sus fronteras 56			11.8 Atributos en el modelo del punto de venta 127
	6.12 Casos de uso primarios, secundarios y opcionales 58			11.9 Multiplicidad entre VentasLinea de Producto y Producto 128
	6.13 Casos esenciales de uso comparados con los casos reales de uso 58			11.10 Modelo conceptual del punto de venta 129
	6.14 Sobre la notación 61			11.11 Conclusión 129
	6.15 Casos de uso dentro de un proceso de desarrollo 63			
	6.16 Pasos del proceso en un sistema del punto de venta 64			
	6.17 Modelos muestra 71			
7	CLASIFICACIÓN Y PROGRAMACIÓN DE LOS CASOS DE USO 73		12	REGISTRO DE LOS TÉRMINOS EN EL GLOSARIO 131
	7.1 Introducción 73			12.1 Introducción 131
	7.2 Programación de los casos de uso en los ciclos de desarrollo 75			12.2 Glosario 131
	7.3 Clasificación de los casos de uso en la aplicación al punto de venta 76			12.3 Actividades y dependencias 132
	7.4 El caso de uso de arranque 76			12.4 Ejemplo de glosario aplicado al sistema del punto de venta 132
	7.5 Programación de los casos de uso en la aplicación del punto de venta 77			
	7.6 Versiones del caso de uso "Comprar productos" 78			
	7.7 Resumen 80			
8	INICIO DE UN CICLO DE DESARROLLO 81		13	COMPORTAMIENTO DE LOS SISTEMAS: DIAGRAMA DE LA SECUENCIA DEL SISTEMA 135
	8.1 Inicio de un ciclo de desarrollo 81			13.1 Introducción 135
				13.2 Actividades y dependencias 135
				13.3 Comportamiento del sistema 137
				13.4 Diagramas de la secuencia del sistema 137
				13.5 Ejemplo de un diagrama de la secuencia de un sistema 137
				13.6 Eventos y operaciones de un sistema 138
				13.7 Cómo elaborar un diagrama de la secuencia de un sistema 140
				13.8 Diagramas de la secuencia de un sistema y otros artefactos 140
				13.9 Eventos y fronteras de un sistema 141
				13.10 Asignación de nombre a los eventos y a las operaciones de un sistema 142
				13.11 Presentación del texto del caso de uso 143
				13.12 Modelos muestra 144
	PARTE III FASE DE ANÁLISIS (1) 83			
9	CONSTRUCCIÓN DE UN MODELO CONCEPTUAL 85		14	COMPORTAMIENTO DE LOS SISTEMAS: CONTRATOS 145
	9.1 Introducción 85			14.1 Introducción 145
	9.2 Actividades y dependencias 87			14.2 Actividades y dependencias 145
	9.3 Modelos conceptuales 87			14.3 Comportamiento de un sistema 147
	9.4 Estrategias para identificar los conceptos 91			14.4 Contratos 147
	9.5 Conceptos idóneos para el dominio del punto de venta 94			14.5 Ejemplo de contrato: introducirProducto 147
	9.6 Directrices para construir modelos conceptuales 96			14.6 Secciones del contrato 148
	9.7 Solución de los conceptos similares: comparación entre TPDV y Registro 97			14.7 Cómo preparar un contrato 149
	9.8 Construcción de un modelo del mundo <i>irreal</i> 98			14.8 Poscondiciones 150
	9.9 Especificación o descripción de conceptos 99			14.9 El espíritu de las poscondiciones: el escenario y el telón 151
	9.10 Definición de términos en el lenguaje UML 101			14.10 Explicación: poscondiciones de <i>introducirProducto</i> 152
	9.11 Modelos Patrón 103			14.11 ¿Cuán completas deben ser las poscondiciones? 153
10	MODELO CONCEPTUAL: AGREGACIÓN DE LAS ASOCIACIONES 105			14.12 Descripción de los detalles y algoritmos del diseño: notas 153
	10.1 Introducción 105			14.13 Precondiciones 153
	10.2 Asociaciones 105			14.14 Recomendación sobre cómo redactar contratos 154
	10.3 Notación de las asociaciones en el UML 106			14.15 Contratos para el caso de uso <i>Comprar productos</i> 155
	10.4 Identificación de las asociaciones: lista de asociaciones comunes 107			14.16 Contratos para el caso de uso <i>Inicio</i> 157
	10.5 ¿Qué grado de detalle deberían tener las asociaciones? 109			14.17 Cambios del modelo conceptual 158
	10.6 Directrices de las asociaciones 110			14.18 Modelos muestra 158
	10.7 Papeles 110			
	10.8 Asignación de nombre a las asociaciones 111			
	10.9 Asociaciones múltiples entre dos tipos 112			
	10.10 Asociaciones e implementación 113			
	10.11 Asociaciones del dominio del punto de venta 113			
	10.12 Modelo conceptual del punto de venta 115			

PARTE IV FASE DE DISEÑO (1) 159	
15	DEL ANÁLISIS AL DISEÑO 161
	15.1 Conclusión de la fase de análisis 161
	15.2 Inicio de la fase de diseño 162
16	DESCRIPCIÓN DE LOS CASOS REALES DE USO 163
	16.1 Introducción 163
	16.2 Actividades y dependencias 163
	16.3 Casos reales de uso 163
	16.4 Ejemplo: Comprar productos: versión 1 165
	16.5 Modelos muestra 166
17	DIAGRAMAS DE COLABORACIÓN 167
	17.1 Introducción 167
	17.2 Actividades y dependencias 167
	17.3 Diagramas de interacción 169
	17.4 Ejemplo de un diagrama de colaboración: efectuarPago 170
	17.5 Los diagramas de interacción son un artefacto de gran utilidad 170
	17.6 Éste es un capítulo dedicado exclusivamente a la notación 171
	17.7 Lea las directrices de diseño en los siguientes capítulos 171
	17.8 Cómo preparar diagramas de colaboración 172
	17.9 Notación básica de los diagramas de colaboración 173
	17.10 Modelos muestra 183
18	GRASP: PATRONES PARA ASIGNAR RESPONSABILIDADES 185
	18.1 Introducción 185
	18.2 Actividades y dependencias 187
	18.3 Los diagramas de interacción bien diseñados son muy útiles 187
	18.4 Responsabilidades y métodos 187
	18.5 Las responsabilidades y los diagramas de interacción 188
	18.6 Patrones 189
	18.7 GRASP: patrones de los principios generales para asignar responsabilidades 191
	18.8 La notación del UML para los diagramas de clase 192
	18.9 Experto 193
	18.10 Creador 197
	18.11 Bajo acoplamiento 200
	18.12 Alta cohesión 203
	18.13 Controlador 206
	18.14 Responsabilidades, representación de papeles y las tarjetas CRC 215
19	DISEÑO DE UNA SOLUCIÓN CON OBJETOS Y PATRONES 217
	19.1 Introducción 217
	19.2 Diagramas de interacción y otros artefactos 218
	19.3 Modelo conceptual del punto de venta 222
	19.4 Diagramas de colaboración para la aplicación TPDV 222
	19.5 El diagrama de colaboración: introducirProducto 223
	19.6 El diagrama de colaboración: terminarVenta 229
	19.7 El diagrama de colaboración: efectuarPago 233
	19.8 El diagrama de colaboración: Iniciar 238
	19.9 Cómo conectar la capa de presentación y la de dominio 243
	19.10 Resumen 245
20	DETERMINACIÓN DE LA VISIBILIDAD 247
	20.1 Introducción 247
	20.2 Visibilidad entre objetos 247
	20.3 Visibilidad 248
	20.4 Presentación de la visibilidad en el UML 253
21	DIAGRAMAS DE CLASES DEL DISEÑO 255
	21.1 Introducción 255
	21.2 Actividades y dependencias 255
	21.3 Cuándo crear diagramas de clases del diseño 257
	21.4 Ejemplo de un diagrama de clases del diseño 257
	21.5 Diagramas de clases del diseño 257
	21.6 Cómo elaborar un diagrama de clases del diseño 258
	21.7 Comparación entre el modelo conceptual y los diagramas de clases del diseño 259
	21.8 Creación de diagramas de clases del diseño para el punto de venta 259
	21.9 Notación de los detalles de los miembros 268
	21.10 Modelos muestra 270
	21.11 Resumen 270
22	ALGUNOS ASPECTOS DEL DISEÑO DE SISTEMAS 271
	22.1 Introducción 271
	22.2 Arquitectura clásica de tres capas 273
	22.3 Arquitecturas multicapas orientadas a objetos 274
	22.4 Cómo mostrar la arquitectura con paquetes de UML 275
	22.5 Identificación de los paquetes 278
	22.6 Estratos y particiones 278
	22.7 Visibilidad entre las clases de paquetes 279
	22.8 Interfaz de los paquetes de servicios: el patrón Fachada 280
	22.9 Sin visibilidad directa respecto a las ventanas: el patrón de Separación Modelo-Vista 281
	22.10 La comunicación indirecta en un sistema 284
	22.11 Coordinadores de las aplicaciones 287
	22.12 Almacenamiento y persistencia 290
	22.13 Modelos muestra 291
PARTE V FASE DE CONSTRUCCIÓN (1) 293	
23	MAPEO DE LOS DISEÑOS PARA CODIFICACIÓN 295
	23.1 Introducción 295
	23.2 La programación y el proceso de desarrollo 295
	23.3 Mapeo de diseños para codificación 298
	23.4 Creación de las definiciones de clase a partir de los diagramas de clases del diseño 299
	23.5 Creación de métodos a partir de los diagramas de colaboración 302
	23.6 Actualizaciones de las definiciones de clases 305
	23.7 Las clases de contenedor/colección en código 306
	23.8 Manejo de las excepciones y de los errores 306
	23.9 Definición del método Venta-hacerLineaDeProducto 307
	23.10 Orden de la implementación 307
	23.11 Resumen del mapeo del diseño a la codificación 308
24	SOLUCIÓN EN PROGRAMA DE JAVA 309
	24.1 Introducción a la solución convertida en programa 309
PARTE VI FASE DE ANÁLISIS (2) 315	
25	ELECCIÓN DE LOS REQUERIMIENTOS DEL CICLO DE DESARROLLO 2 317
	25.1 Requerimientos del ciclo de desarrollo 2 317
	25.2 Suposiciones y simplificaciones 317

26	CÓMO RELACIONAR CASOS MÚLTIPLES DE USO 321	26.1 Introducción 321 26.2 Cuándo crear casos de uso independientes 321 26.3 Diagramas de casos de uso con las relaciones <i>usa</i> 322 26.4 Documentos de casos de uso con las relaciones <i>usa</i> 323	33.7 Tipos que requieren diagramas de estado 384 33.8 Otros diagramas de estado para la aplicación Punto de Venta 386 33.9 Ejemplificación de eventos externos y de intervalos 387 33.10 Notación complementaria de los diagramas de estado 388
27	EXTENSIÓN DEL MODELO CONCEPTUAL 329	27.1 Nuevos conceptos en el sistema del punto de venta 329	
28	GENERALIZACIÓN 335	28.1 Generalización 335 28.2 Definición de supertipos y de subtipos 336 28.3 Cuándo definir un subtipo 339 28.4 Cuándo definir un supertipo 342 28.5 Jerarquías de los tipos del punto de ventas 342 28.6 Tipos abstractos 345 28.7 Construcción de modelos con estados cambiantes 347 28.8 Jerarquías de clases y herencia 348	
29	PAQUETES: ORGANIZACIÓN DE LOS ELEMENTOS 349	29.1 Introducción 349 29.2 Notación de los paquetes en el UML 350 29.3 Cómo partir el modelo conceptual 351 29.4 Paquetes del modelo conceptual del punto de venta 352	
30	REFINAMIENTO DEL MODELO CONCEPTUAL 355	30.1 Introducción 355 30.2 Tipos asociativos 355 30.3 Agregación y composición 359 30.4 Nombres de los papeles de la asociación 362 30.5 Los papeles como conceptos y los papeles en las asociaciones 363 30.6 Elementos derivados 364 30.7 Asociaciones calificadas 365 30.8 Asociaciones recursivas o reflexivas 366	
31	MODELO CONCEPTUAL: RESUMEN 367	31.1 Introducción 367 31.2 Paquete de los conceptos del dominio 368 31.3 Paquete básico/varios 368 31.4 Pagos 369 31.5 Productos 369 31.6 Ventas 370 31.7 Transacciones de autorización 371	
32	COMPORTAMIENTO DE LOS SISTEMAS 373	32.1 Diagramas de secuencia del sistema 373 32.2 Nuevos eventos del sistema 374 32.3 Contratos 375	
33	MODELADO DEL COMPORTAMIENTO EN LOS DIAGRAMAS DE ESTADO 379	33.1 Introducción 379 33.2 Eventos, estados y transiciones 379 33.3 Diagramas de estado 381 33.4 Diagramas de estado para los casos de uso 382 33.5 Diagramas de estado del sistema 383 33.6 Diagramas de estado de los casos de uso para la aplicación del punto de venta 384	
			PARTE VII FASE DE DISEÑO (2) 391
			34 GRASP: MÁS PATRONES PARA ASIGNAR RESPONSABILIDADES 393
			34.1 GRASP: Patrones generales de software para asignar responsabilidades 393 34.2 Polimorfismo 394 34.3 Fabricación Pura 396 34.4 Indirección 398 34.5 No Hables con Extraños 400
			35 DISEÑO CON MÁS PATRONES 405
			35.1 Introducción 405 35.2 Estado (Pandilla de los Cuatro) 406 35.3 Polimorfismo (GRASP) 411 35.4 Singleton 413 35.5 Agente Remoto y Agente (Pandilla de los Cuatro) 416 35.6 El patrón Fachada y el Agente Dispositivo (Pandilla de los Cuatro) 418 35.7 El patrón Comando (Pandilla de los Cuatro) 421 35.8 Conclusión 424
			PARTE VIII TEMAS ESPECIALES 425
			36 OTRA NOTACIÓN DE UML 427
			36.1 Introducción 427 36.2 Notación general 427 36.3 Interfaces 429 36.4 Diagramas de implementación 429 36.5 Mensajes asincrónicos en los diagramas de colaboración 430 36.6 Interfaces de paquetes 432
			37 PROBLEMAS DEL PROCESO DE DESARROLLO 433
			37.1 Introducción 433 37.2 ¿Por qué molestarnos? 434 37.3 Directrices de un proceso eficiente 434 37.4 Desarrollo iterativo e incremental 435 37.5 Desarrollo orientado a los casos de uso 437 37.6 Énfasis inicial en la arquitectura 437 37.7 Fases del desarrollo 438 37.8 Duración de los ciclos de desarrollo 447 37.9 Aspectos del ciclo de desarrollo 448 37.10 Programación del desarrollo de las capas arquitectónicas 452
			38 ESQUEMAS, PATRONES Y PERSISTENCIA 455
			38.1 Introducción 455 38.2 El problema: objetos persistentes 456 38.3 La solución: un esquema de persistencia 456 38.4 ¿Qué es un esquema (framework)? 457 38.5 Requerimientos del esquema de persistencia 458 38.6 ¿Superclase ObjetoPersistente? 459 38.7 Ideas básicas 459

PREFACIO

- 38.8 Mapeo: patrón *Representación de objetos como tablas* 460
- 38.9 Identificación de objetos: el patrón *Identificador de objetos* 461
- 38.10 Intermediarios: el patrón *Intermediario de base de datos* 462
- 38.11 Diseño de esquemas: el patrón *Método de Plantilla* 463
- 38.12 Materialización: el patrón *Método de Plantillas* 464
- 38.13 Objetos colocados en espacio caché: el patrón *Administración de Caché* 467
- 38.14 Referencias inteligentes: los patrones *Agente Virtual* y *Puente* 468
- 38.15 Agentes Virtuales e Intermediarios de Bases de Datos 473
- 38.16 Cómo Representar las relaciones en tablas 475
- 38.17 El patrón *Instanciación de Objetos Complejos* 476
- 38.18 Operaciones de transacciones 479
- 38.19 Búsqueda de objetos en el almacenamiento persistente 483
- 38.20 Diseños alternos 484
- 38.21 Cuestiones sin resolver 486

APÉNDICE A. LECTURAS RECOMENDADAS 487

APÉNDICE B. EJEMPLOS DE ACTIVIDADES Y MODELOS DE DESARROLLO 489

BIBLIOGRAFÍA 495

GLOSARIO 497

ÍNDICE 503

Diseño de sistemas de objetos robustos y de fácil mantenimiento.

Un mapa que lo guiará a través de los requerimientos, el análisis, el diseño y la codificación.

Uso de la notación UML para exemplificar los modelos de análisis y diseño.

*Mejora de los diseños aplicando los patrones de diseño de la “pandilla de los cuatro” (*gang-of-four*; *GoF*) y *GRASP*.*

Aprendizaje eficiente basado en una exposición muy bien organizada.

Felicidades y gracias por leer este libro. Ponemos en sus manos una guía práctica y un mapa que lo conducirán por el camino del análisis y el diseño orientados a objetos. A continuación le explicamos para qué le servirá.

Primero, como sigue proliferando la tecnología de objetos en el desarrollo del software, hoy aún más, gracias a la adopción generalizada de Java, el dominio del análisis y del diseño orientados a objetos es indispensable para usted si quiere crear sistemas robustos y de fácil mantenimiento orientados a objetos. Esta tecnología abre todo un mundo de oportunidades a los arquitectos, analistas y diseñadores.

Segundo, si el análisis y el diseño orientados a objetos son un tema nuevo para el lector, seguramente querrá saber cómo avanzar por esta área tan compleja: nuestro libro ofrece un mapa de actividades bien definidas, para que vaya dominando gradualmente los requisitos de la codificación.

Tercero, el UML (*Unified Modeling Language*, Lenguaje Unificado de Construcción de Modelos) nació como una notación estándar de la construcción de modelos; por ello le será de gran utilidad familiarizarse con él. En este libro, mediante la notación de UML, aprenderá las técnicas del análisis y el diseño orientados a objetos.

Cuarto, los patrones de diseño ofrecen las soluciones e idiomas “más prácticos” de que los expertos en el diseño orientado a objetos se sirven para crear sistemas. En este libro aprenderá los patrones del diseño, entre ellos los famosos patrones de la pandilla de los cuatro y, sobre todo, los de GRASP, que comunican los principios fundamentales de la asignación de responsabilidades en el diseño orientado a objetos. Al ir aprendiendo y utilizando los patrones, dominará más rápidamente el análisis y el diseño.

Quinto, la estructura y el enfoque de este libro se fundan en muchos años de experiencia en la capacitación y de tutoría en el arte del análisis y diseño orientados a objetos. En el libro se plasma esa experiencia docente: ofrece un enfoque refinado, probado y eficiente que facilita el aprendizaje del tema, con lo cual seguramente los lectores aprovecharán al máximo el tiempo que dediquen a la lectura y al aprendizaje.

Aprendizaje
mediante un
ejercicio realista.

Sexto, se examina a fondo un solo caso de estudio, con el propósito de explicar realista e íntegramente el proceso del análisis y el diseño orientados a objetos, profundizando además en detalles complejos del problema: es éste un ejercicio muy parecido a los casos que se encuentran en la realidad.

Traducción
a código.

Séptimo, se muestra cómo mapear los elementos del diseño orientado a objetos para codificarlos en Java.

Diseño de una
arquitectura
de capas.

Octavo, explica cómo diseñar una arquitectura de capas (o niveles) y relaciona la capa gráfica de la interfaz con las del dominio y los servicios del sistema. Se trata de un tema de importancia práctica que a menudo pasa inadvertido.

Diseño de un
esquema.

Finalmente, se muestra al lector cómo diseñar un esquema orientado a objetos, que además es aplicado específicamente a la creación de un esquema de almacenamiento persistente en una base de datos.

Objetivos

El objetivo global es:

Ayudar a los estudiantes y diseñadores a crear mejores diseños orientados a objetos, recurriendo para ello a principios explicables y a la heurística.

Al estudiar y aplicar la información y los métodos del libro, al lector le será más fácil entender un problema a partir de sus procesos y conceptos, así como diseñar una solución sólida por medio de objetos.

A quiénes está destinado el libro

Este libro se dirige a los siguientes lectores:

- Diseñadores con experiencia en un lenguaje de programación orientado a objetos, pero con pocos conocimientos —o relativamente pocos— en el área del análisis y el diseño orientados a objetos.
- Estudiantes de computación o que toman cursos de ingeniería de software en los cuales se enseña la tecnología de objetos.
- Quienes estén familiarizados con el análisis y el diseño orientados a objetos y que deseen aprender la notación del (*Unified Modeling Language*, Lenguaje Unificado de Construcción de Modelos) y aplicar patrones, o bien que quieran ampliar y perfeccionar sus habilidades de análisis y diseño.

Requisitos

Se suponen y se requieren algunos conocimientos previos para aprovechar mejor este libro:

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

No se definen en este libro los conceptos básicos de la tecnología de objetos.

Organización del libro

La organización del libro se basa en la siguiente estrategia global: introducir los temas del análisis y el diseño orientados a objetos en un orden semejante al de un proyecto de desarrollo de software que se realiza a través de dos ciclos de desarrollo iterativo. En el primero se describen el análisis y el diseño. En el segundo, se exponen nuevos temas de análisis y diseño, mientras que los temas actuales se estudian más a fondo.

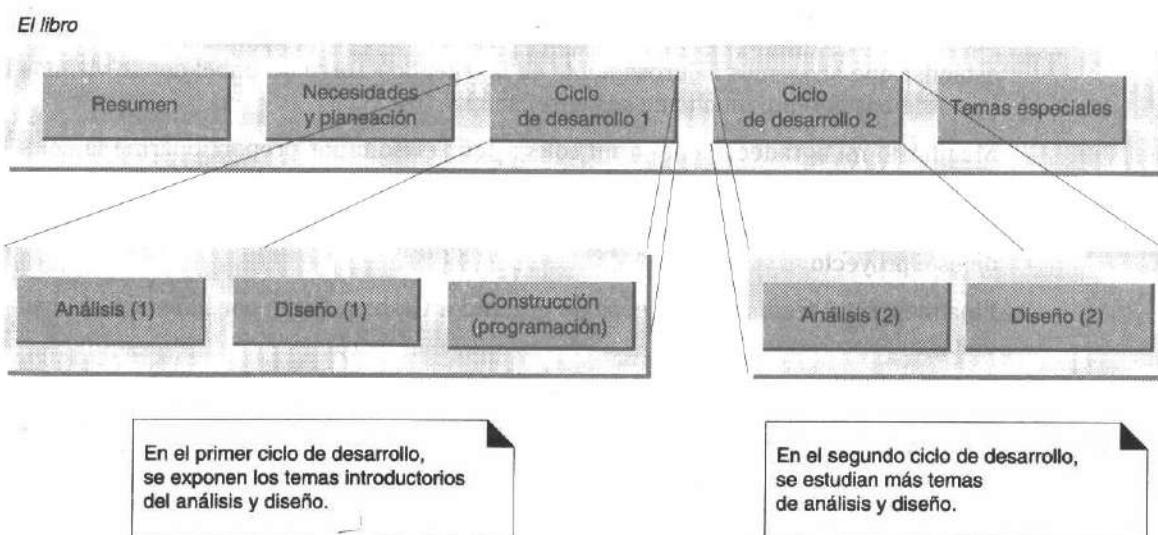


Figura 1. El libro está organizado como un proyecto de desarrollo.

Objetivo del libro

La tecnología de objetos es un área muy prometedora, pero su potencial no se aprovechará plenamente si no se poseen las habilidades apropiadas. Me propongo difundir su adopción eficiente mediante la práctica del análisis y del diseño orientados a objetos y favorecer al mismo tiempo la adquisición de las destrezas relacionadas, porque me he dado cuenta de que son indispensables para la creación y el mantenimiento exitosos de los sistemas importantes.

Reconocimientos

Agradezco a todos los usuarios y estudiosos del programa UML y del análisis y diseño orientados a objetos a quienes he procurado brindar mi ayuda; ellos son mis mejores maestros.

Un testimonio especial de gratitud merecen los revisores del libro (o de parte de él), entre ellos: Kent Beck, Jens Coldewey, Clay Davis, Tom Heruska, Luke Hohmann, David Norris, David Nunn, Brett Schuchert y todo el equipo Mercury. Doy gracias a Grady Booch por revisar el tutorial de la edición en inglés y a Jim Rumbaugh por sus comentarios sobre la relación existente entre el lenguaje UML y el proceso.

Agradezco sus excelentes comentarios acerca del proceso y los modelos a Todd Girvin, John Hebley, Tom Heruska, David Norris, David Nunn, Charles Rego y Raj Wall.

Mi más profunda gratitud a Grady Booch, Ivar Jacobson y Jim Rumbaugh por el desarrollo del *Unified Modeling Language*; por la creación de una notación abierta y estándar que acogemos calurosamente en la auténtica Torre de Babel donde vivimos hoy. Además aprendí mucho de sus enseñanzas.

Manifiesto mi agradecimiento a mi colega Jef Newsom por proporcionarme la solución en Java al estudio de casos.

Gracias a Paul Becker, mi editor de Prentice-Hall, por su inquebrantable fe en el valor de este proyecto.

Finalmente, un testimonio especial de gratitud a Graham Glass por haberme abierto una puerta.

Semblanza del autor

Craig Larman tiene la licenciatura y la maestría en computación, y desde 1978 ha venido desarrollando sistemas grandes y pequeños de software en plataformas que abarcan desde macrocomputadoras hasta microcomputadoras, valiéndose de tecnologías de software que incluyen desde 4GLs hasta programación lógica y programación orientada a objetos.

A principios de los años ochenta, Larman se enamoró de la inteligencia artificial y de las técnicas de programación de los sistemas del conocimiento, área donde tuvo su primer contacto con la programación orientada a objetos (en Lisp). Enseñó y trabajó con la programación en Lisp desde 1984, con Smalltalk desde 1986, con C++ desde 1991 y recientemente con Java, además de impartir varios métodos de análisis y diseño orientados a objetos. Ha ayudado a más de 2,000 estudiantes en el aprendizaje de temas relacionados con la tecnología de objetos.

Actualmente es jefe de instructores en ObjectSpace, compañía que se especializa en el cómputo distribuido, en agentes y en tecnología de objetos.

Si el lector lo desea, puede ponerse en contacto con Craig por correo electrónico: clarman@acm.org.

Convenciones tipográficas

Se utilizan las negritas cuando se presenta un **nuevo término** en una oración.

Se emplean las cursivas para el nombre de una *Clase* o *método*.

Las referencias bibliográficas se hacen escribiendo entre corchetes el apellido del autor y el año de publicación de la obra [Bob67]. En la bibliografía aparecen los datos completos de cada obra.

Se adoptó el criterio de no usar acentos en los modelos conceptuales, diagramas y programas.

Con el operador “--” de resolución del ámbito independiente del lenguaje se indican un método y su clase relacionada así: *ClassName--methodName()*.

PARTE I INTRODUCCIÓN

ANÁLISIS Y DISEÑO ORIENTADOS A OBJETOS

Objetivos

- Comparar y contrastar el análisis y el diseño.
- Definir el análisis y el diseño orientados a objetos.
- Relacionar por analogía el análisis y el diseño orientado a objetos con el fin de organizar una empresa.

1.1 Aplicación del lenguaje UML y del análisis y el diseño orientados a objetos

Desarrolle habilidades en el análisis y diseño orientados a objetos.

¿Qué significa contar con un sistema orientado a objetos que esté bien diseñado? En este libro ayudamos a programadores y estudiantes para que adquieran y utilicen habilidades prácticas en el análisis y el diseño orientados a objetos. Esas destrezas son indispensables para crear sistemas de software bien diseñados, robustos y de fácil mantenimiento, utilizando la tecnología de objetos y los lenguajes de programación orientados a objetos como C++, Java o Smalltalk.

El proverbio “El hábito no hace al monje” se aplica perfectamente a la tecnología de objetos. El hecho de conocer un lenguaje orientado a objetos (Java, por ejemplo) y además tener acceso a una rica biblioteca (como la de Java) es un primer paso necesario pero insuficiente para crear sistemas de objetos. Se requiere además analizar y diseñar un sistema desde la perspectiva de los objetos.

Nota del editor: En esta edición se adoptó el criterio de no usar acentos en los modelos conceptuales, diagramas y programas.

La práctica del análisis y el diseño orientados a objetos se explica con un caso de estudio relativamente exhaustivo que vamos desarrollando a lo largo del libro: ambos aspectos se profundizan lo suficiente para que se traten y resuelvan muchos de los detalles más engorrosos que plantea un problema realista.

"UML" son las siglas de **Unified Modeling Language** (Lenguaje Unificado de Construcción de Modelos), notación (esquemática en su mayor parte) con que se construyen sistemas por medio de conceptos orientados a objetos. En este libro ayudaremos a los programadores para que aprendan la notación de UML y la apliquen a un estudio de casos.

¿Cómo deberían asignarse las responsabilidades a las clases de objetos? ¿Cómo deberían interactuar éstos? ¿Qué papel debe destinárse a cada clase? Estas son preguntas muy importantes cuando se diseña un sistema. Algunas soluciones ya probadas y eficaces de los problemas de diseño pueden expresarse (y se han plasmado) como un conjunto de principios, con heurística o **patrones** (fórmulas de solución de problemas que codifican los principios aceptados del diseño). En este libro enseñamos los patrones y así favorecemos el aprendizaje rápido y la utilización hábil de los tecnicismos fundamentales de esta tecnología.

Debido a las numerosas actividades que posiblemente exijan las condiciones durante la implementación, ¿cómo debería proceder el programador o equipo de programadores? En esta obra presentamos un *ejemplo del proceso de desarrollo* que describe un orden posible de actividades y un ciclo de vida del desarrollo. Pero no prescribe un proceso ni un método definitivos; se limita a ofrecer una muestra de pasos comunes.

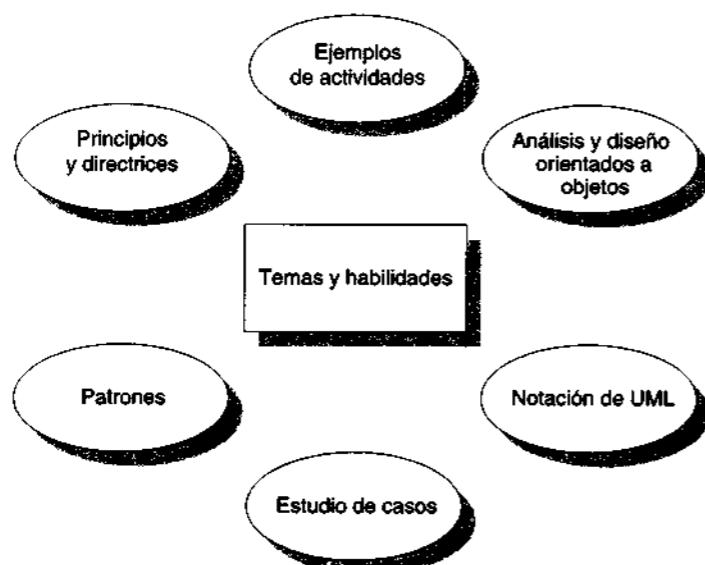


Figura 1.1 Temas y habilidades que se incluyen en el libro.

En conclusión, en este libro se ayuda al programador:

- A aplicar los principios y patrones para aprender a realizar mejores diseños.
- A efectuar varias actividades comunes en el análisis y en el diseño.
- A crear elementos útiles en la notación de UML.

Todo lo anterior lo exemplificamos dentro del contexto de un estudio de casos.

1.2 Asignación de responsabilidades

La asignación de responsabilidades a los componentes es la habilidad más importante del diseño.

Hay muchas posibles actividades y elementos en el análisis y en el diseño, así como gran cantidad de principios y directrices. Supóngase que tuviéramos que elegir una sola habilidad práctica entre todos los temas aquí expuestos: una habilidad tan solitaria como una "isla desierta" por así decirlo. ¿Cuál sería?

La habilidad más importante en el análisis y el diseño orientados a objetos es asignar eficientemente las responsabilidades a los componentes del software.

¿Por qué? Porque es la actividad que debe efectuarse (es ineludible) y que influye más profundamente en la solidez, en la capacidad de mantenimiento y en la reutilización de los componentes del software.

El segundo lugar por orden de importancia aparece la obtención de los objetos o las abstracciones adecuadas. Ambos aspectos son decisivos: ponemos de relieve la asignación de responsabilidades porque tiende a ser más difícil de dominar.

En un objeto real, un programador tal vez no tenga la oportunidad de efectuar alguna otra actividad relacionada con el análisis o con el diseño: el proceso de desarrollo "con prisa por codificar". Pero incluso en tales casos es inevitable la asignación de responsabilidades.

Por ello, en la fase de diseño de este libro ponemos de relieve los principios de la asignación de responsabilidades y ofrecemos las herramientas que le ayudarán al programador a dominarla.

Nueve principios fundamentales de la asignación de responsabilidades, codificados en los patrones de GRASP, se describen y se aplican para ayudarle al lector a dominar este aspecto.

Lo anterior no significa que el resto de las actividades carezcan de importancia. Por ejemplo, es indispensable crear un modelo conceptual que identifique los conceptos (objetos) en el dominio del problema. Pero, en cuanto habilidad de tipo "isla desierta", el hecho de saber asignar responsabilidades es lo que en definitiva hace o destruye a un sistema.

1.3 ¿Qué son el análisis y el diseño?

Análisis:
investigación.

Para crear una aplicación de software hay que describir el problema y las necesidades o requerimientos: en qué consiste el conflicto y qué debe hacerse. El **Análisis** se centra en una *investigación* del problema, no en la manera de definir una solución. Por ejemplo, si se desea un nuevo sistema de información computarizada de una biblioteca, ¿cuáles procesos de la institución se relacionan con su uso?

Diseño: solución.

Para desarrollar una aplicación, también es necesario contar con descripciones detalladas y de alto nivel de la solución lógica y saber cómo satisface los requerimientos y las restricciones. El **Diseño** pone de relieve una *solución lógica*: cómo el sistema cumple con los requerimientos. He aquí un ejemplo: ¿de qué manera el software del sistema de información de la biblioteca capturará y registrará los préstamos de libros? En definitiva, los diseños se implementan en software y en hardware.

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

Análisis orientado a objetos: ¿conceptos?

La esencia del **análisis y el diseño orientados a objetos** consiste en situar el dominio de un problema y su solución lógica dentro de la perspectiva de los objetos (cosas, conceptos o entidades), como se advierte en la figura 1.2.

Diseño orientado a objetos: ¿objetos de software?

Durante el **análisis orientado a objetos** se procura ante todo identificar y describir los objetos —o conceptos— dentro del dominio del problema. Por ejemplo, en el caso del sistema de información de la biblioteca, algunos de los conceptos son *Libro*, *Biblioteca* y *Cliente*.

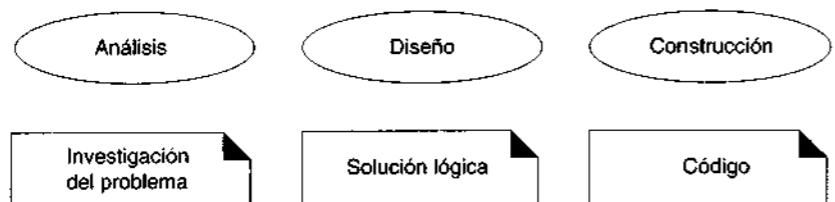


Figura 1.2 Significado de las actividades de desarrollo.

Durante el **diseño orientado a objetos**, se procura definir los objetos lógicos del software que finalmente serán implementados en un lenguaje de programación orientado a objetos. Los objetos tienen atributos y métodos. Así, en el sistema de la biblioteca un objeto de software *Libro* puede tener un atributo *titulo* y un método *imprimir* (figura. 1.3).

Finalmente, durante la **construcción o programación orientada a objetos**, se implementan los componentes del diseño, como una clase *Libro* en C++, Java, Smalltalk o Visual Basic.

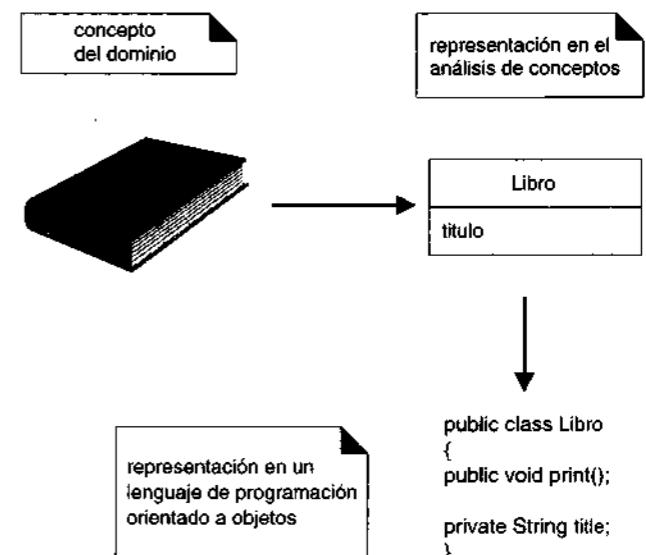


Figura 1.3 La orientación a objetos se centra en la representación de objetos.

1.5 Una analogía: organización de la empresa MicroChaos

La organización de una empresa y el análisis y el diseño orientados a objetos guardan semejanzas.

Algunos de los principales pasos del análisis y del diseño orientados a objetos pueden explicarse por analogía con la forma en que se organiza una compañía.

1.5.1 MicroChaos está creciendo rápidamente...

Imagine ser el fundador y director general de MicroChaos, compañía recién creada y especializada en el software que aplica los modelos matemáticos de la teoría del caos al análisis del mercado accionario (esto ya se ha hecho en el mundo de las finanzas). Su producto principal es *MicroButterfly*.



En el inicio tan prometedor de la compañía todo mundo compartía el trabajo: contestaban el teléfono, surtían los pedidos, escribían los programas de computación. La compañía había alcanzado un éxito extraordinario: había muchos empleados de ingreso reciente y el

personal se había vuelto dispersivo porque realizaba demasiadas tareas diferentes. Ha llegado, pues, el momento de implantar una organización más rigurosa. ¿Cómo procedería usted, en su calidad de director general, para organizar a los empleados y las actividades?

1.5.2 ¿Qué son los procesos de negocios?

El primer paso consiste en analizar lo que debe hacer una empresa —sus procesos de negocios— si quiere seguir funcionando: realizar ventas, pagar a empleados y acreedores, desarrollar programas de computación.

Desde el punto de vista del método del análisis y del diseño orientados a objetos, este paso nos recuerda el **análisis de requerimientos**, en el cual los procesos y las necesidades de los negocios se descubren y se expresan en los **casos de uso**. Los casos son descripciones narrativas textuales de los procesos de una empresa o sistema, como se muestra en seguida:

Caso de uso: Colocar un pedido.

Descripción: Este caso de uso comienza cuando un cliente telefona a un representante de ventas para hacer una compra de MicroButterfly. El representante anota en una nueva orden la información relativa al cliente y al producto.

Identificar los procesos y registrarlos en los casos de uso no es en realidad una actividad del análisis orientado a objetos; de ninguna manera se centra en objetos.

Con todo, se trata de un paso importante y generalizado en los métodos del análisis y diseño orientados a objetos. Asimismo, los casos de uso forman parte del lenguaje UML. A continuación mostramos al lector gráficamente nuestra analogía:

Analogía de la empresa	Ánalisis y diseño orientados a objetos	Documentos relacionados
¿Cuáles son los procesos de negocios?	Ánalisis de requerimientos	Casos de uso

1.5.3 ¿Cuáles son los papeles o las funciones en la organización?

El siguiente paso consiste en identificar los papeles de las personas que intervendrán en los procesos: cliente, representante de ventas, ingeniero de software, etcétera.

En la perspectiva del análisis y del diseño orientados a objetos, este paso nos recuerda al **análisis del dominio orientado a objetos** que expresamos con un **modelo conceptual**. Este modelo presenta las diversas categorías de las cosas en el dominio; no sólo los papeles de las personas sino también todas las cosas de interés, según se observa en la figura 1.4.

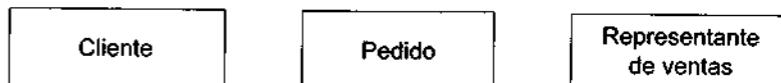


Figura 1.4 Conceptos en la empresa.

Analogía de la empresa	Ánalisis y diseño orientados a objetos	Documentos relacionados
¿Cuáles son los procesos de negocios?	Ánalisis de requerimientos	Casos de uso
¿Cuáles son los papeles de los empleados?	Ánalisis del dominio	Modelo conceptual

1.5.4 ¿Qué funciones cumple cada empleado? ¿Cómo colabora el personal?

Una vez identificados los procesos de su empresa y el personal, es el momento de determinar la manera de cumplir los procesos. Se trata de una actividad de diseño, o sea orientada a las soluciones. Junto con los empleados, defina las responsabilidades de ellos a fin de efectuar las tareas necesarias para llevar a cabo un proceso. También necesita definir de qué manera los empleados colaborarán o compartirán el trabajo.



Desde el punto de vista del análisis y del diseño orientados a objetos, esta actividad se parece al **diseño orientado a objetos** que pone de relieve la **asignación de responsabilidades**. La asignación significa distribuir las funciones y las responsabilidades entre varios objetos de software en la aplicación, del mismo modo que se asignan a los papeles de los empleados. Los objetos de software normalmente colaboran o interactúan para cumplir con sus responsabilidades, como lo hacen las personas.

La descripción de la asignación de las responsabilidades y las interacciones de objetos a menudo se expresan gráficamente con **diagramas de diseño de clase** y con **diagramas de colaboración**; unos y otros muestran la definición de clases y el flujo de mensajes entre los objetos de software.

ANÁLISIS DE REQUERIMIENTOS	ANÁLISIS DEL DOMINIO	DISEÑO DE CLASES
¿Cuáles son los procesos del negocio?	Análisis de requerimientos	Casos de uso
¿Cuáles son los papeles de los empleados?	Análisis del dominio	Modelo conceptual
¿Qué funciones cumplen los empleados? ¿Cómo interactúan ellos?	Asignación de responsabilidades, diseño de interacciones	Diagramas de diseño de clases, diagramas de colaboración

1.6 Un ejemplo del análisis y del diseño orientados a objetos

Un simple ejemplo nos permite ver todo el panorama.

Se necesita abundante información para explicar cabalmente el análisis y el diseño orientados a objetos. Para no perdernos en muchos detalles, ofrecemos al lector una explicación sucinta sobre algunos de los pasos y diagramas usando para ello un ejemplo simple: un “juego de dados” en que un jugador lanza dos dados. Si el total es siete, gana y de lo contrario pierde.



Las notaciones utilizadas forman parte del lenguaje UML.

Observe, por favor, que en el ejemplo no están representados todos los pasos posibles ni los diagramas; tan sólo aparecen los más comunes y esenciales.

1.6.1 Definición de los casos de uso

Para entender los requerimientos se necesita, en parte, conocer los procesos del dominio y el ambiente externo, o sea los factores externos que participan en los procesos. Dichos procesos de dominio pueden expresarse en **casos de uso**, o sea, en descripciones narrativas de los procesos del dominio en un formato estructurado de prosa.



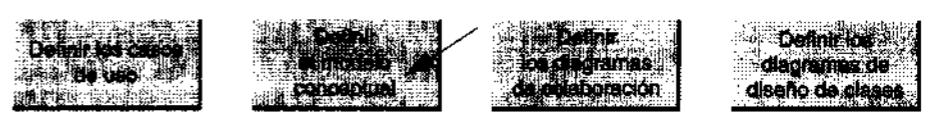
Los casos de uso no son propiamente un elemento del análisis orientado a objetos; se limitan a describir procesos y pueden ser igualmente eficaces en un proyecto de tecnología no orientada a objetos. No obstante, constituyen un paso preliminar muy útil porque describen las especificaciones de un sistema.

Por ejemplo, en el juego de dados el caso de uso de *Juega un Juego*.

Caso de uso:	Juega un Juego.
Participantes:	Jugador.
Descripción:	Este caso de uso comienza cuando el jugador recoge y hace rodar los dados. Si los puntos suman siete, gana y pierde si suman cualquier otro número.

1.6.2 Definición de un modelo conceptual

El análisis orientado a objetos tiene por finalidad estipular una especificación del dominio del problema y los requerimientos desde la perspectiva de la clasificación por objetos y desde el punto de vista de entender los términos empleados en el dominio. Para descomponer el dominio del problema hay que identificar los conceptos, los atributos y las asociaciones del dominio que se juzgan importantes. El resultado puede expresarse en un **modelo conceptual**, el cual se muestra gráficamente en un grupo de diagramas que describen los conceptos (objetos).



Por ejemplo, como se aprecia en la figura 1.5, en el dominio del juego de dados, una parte del modelo conceptual muestra los conceptos *Jugador*, *Dados* y *JuegodeDados*, sus asociaciones y atributos.

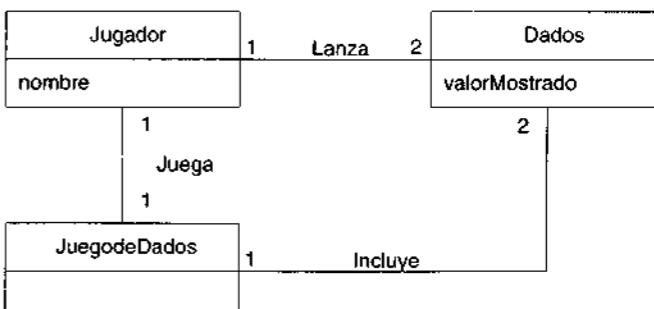
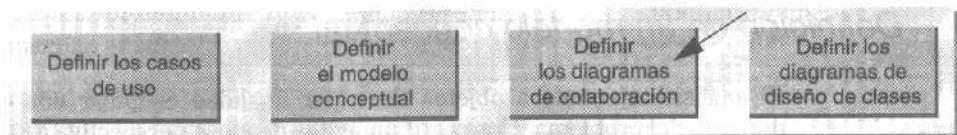


Figura 1.5 Modelo conceptual del juego de dados.

El modelo conceptual no es una descripción de los componentes del software; representa los conceptos en el dominio del problema en el mundo real.

1.6.3 Definición de los diagramas de colaboración

El diseño orientado a objetos tiene por objeto definir las especificaciones lógicas del software que cumplan con los requisitos funcionales, basándose en la descomposición por clases de objetos. Un paso esencial de esta fase es la asignación de responsabilidades entre los objetos y mostrar cómo interactúan a través de mensajes, expresados en **diagramas de colaboración**. Éstos presentan el flujo de mensajes entre las instancias y la invocación de métodos.



Por ejemplo, supongamos que se desea una simulación en el software del juego de dados. El diagrama de colaboración de la figura 1.6 muestra gráficamente el paso esencial del juego, enviando mensajes a las instancias de las clases *Jugador* y *Dado*.

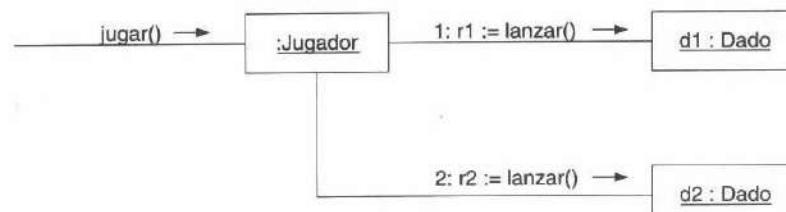


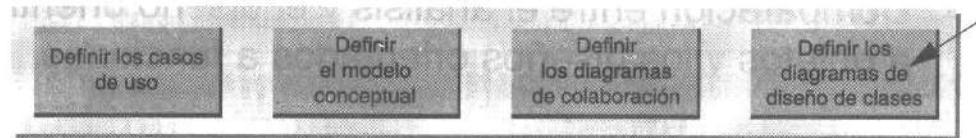
Figura 1.6 Diagrama de colaboración que muestra los mensajes entre los objetos de software.

1.6.4 Definición del diseño de clases

Para definir una clase es preciso contestar varias preguntas:

- ¿Cómo se conectan unos objetos a otros?
- ¿Cuáles son los métodos de una clase?

Si el lector quiere contestar las preguntas anteriores, examine detenidamente los diagramas de colaboración que indican las conexiones necesarias entre objetos, y también los métodos que cada clase de software debe definir. El **diagrama de diseño de clases** es el que expresa esos detalles. Muestra las definiciones de clase que han de implementarse en el software.



Por ejemplo, en el juego de dados, al examinar el diagrama de colaboración, obtenemos el siguiente diagrama del diseño de clases. Puesto que un mensaje *juego* se envía a una instancia *Jugador*, *Jugador* requiere un método *jugar*, mientras que *Dado* requiere un método *lanzar*.

A diferencia del modelo conceptual, este diagrama no muestra gráficamente conceptos del mundo real; describe únicamente los componentes del software.

Para indicar de qué manera los objetos se conectan entre sí a través de atributos, una línea con una flecha en la punta indicará un atributo. Por ejemplo, *JuegodeDados* posee un atributo que apunta a una instancia de un *Jugador*.

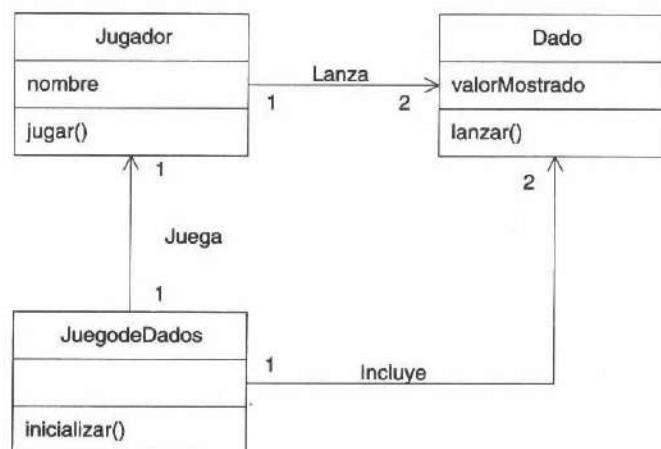


Figura 1.7 Diagrama del diseño de clases para los componentes de software.

El diagrama del diseño de clases de la figura 1.7 no está completo; representa únicamente el inicio de las especificaciones del software que se requieren para definir cabalmente cada clase.

1.6.5 Resumen del ejemplo del juego de dados

El juego de dados es un problema muy simple, que incluimos para centrarnos en algunos de los pasos y artefactos del análisis y diseño orientados a objetos y no en el dominio del problema. En capítulos subsecuentes trataremos más a fondo esos tres aspectos, sirviéndose para ello de un problema más complejo y realista.

1.7 Comparación entre el análisis y el diseño orientados a objetos y los diseños orientados a funciones

Los proyectos de software son complejos, y la estrategia primaria para superar la complejidad es la descomposición (divide y vencerás): dividir el problema en unidades manejables. Antes del advenimiento del análisis y diseño orientados a objetos, el método usual de descomponer un problema eran el **análisis** y el **diseño estructurados**, cuya dimensión de descomposición es fundamentalmente por función o proceso, lo cual origina una división jerárquica de procesos constituidos por subprocesos.

Sin embargo, existen también otras dimensiones de descomposición; el análisis y el diseño orientados a objetos buscan ante todo descomponer un espacio de problema por objetos y no por funciones, como se advierte en la figura 1.8.

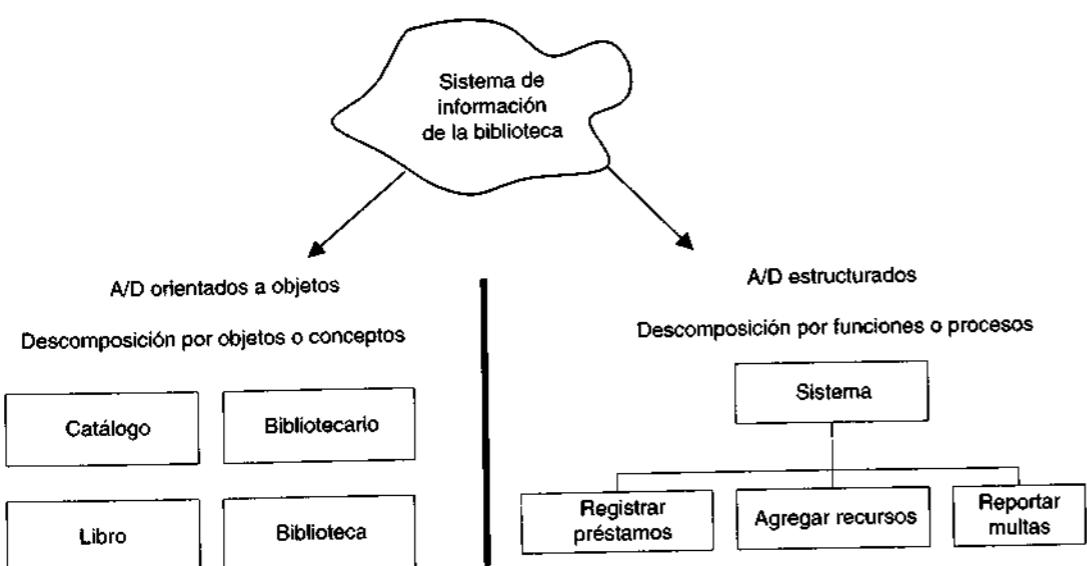


Figura 1.8 Comparación entre la descomposición orientada a objetos y la descomposición orientada a funciones.

1.8 Advertencia: el “análisis” y el “diseño” pueden provocar guerras terminológicas

La división entre análisis y diseño es poco clara; el trabajo de los dos existe en un continuo (figura 1.9), y los profesionales de los métodos del “análisis” y “diseño” clasifican una actividad en varios puntos del continuo. De ahí que no convenga adoptar una actitud rígida ante lo que constituye un paso del análisis o del diseño.

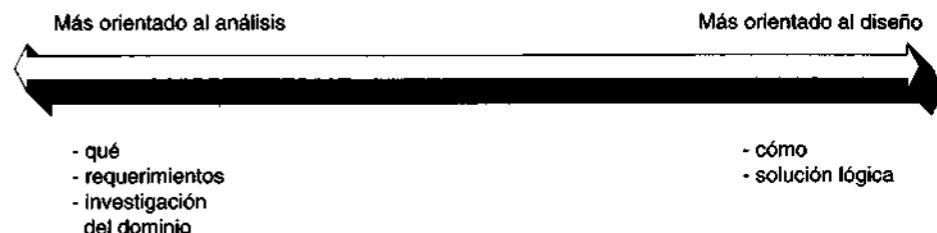


Figura 1.9 Las actividades de análisis y diseño existen en un continuo.

Debatir sobre la definición no resulta constructivo en absoluto, ya que las personas les dan distintos significados a esos dos términos y no se emplean con el mismo sentido en los métodos: lo importante es saber resolver el problema de crear software y darle mantenimiento con mayor eficiencia, con mayor rapidez y a un precio menor.

Con todo, en la práctica conviene contar con una distinción constante entre *investigación* (análisis) y *solución* (diseño), porque es útil tener un paso bien definido que indague la naturaleza del problema antes de buscar la manera de crear una solución. Además genera expectativas de un comportamiento apropiado entre los miembros del equipo; por ejemplo, durante el análisis esperan subrayar la *comprensión* del problema, posponen las cuestiones concernientes a la solución, al desempeño y a otros aspectos.

1.9 El Unified Modeling Language, UML

El UML (Lenguaje Unificado para la Construcción de Modelos) se define como un “lenguaje que permite especificar, visualizar y construir los artefactos de los sistemas de software...” [BJR97]. Es un sistema notacional (que, entre otras cosas, incluye el significado de sus notaciones) destinado a los sistemas de modelado que utilizan conceptos orientados a objetos.

El UML es un estándar incipiente de la industria para construir modelos orientados a objetos. Nació en 1994 por iniciativa de Grady Booch y Jim Rumbaugh para combinar sus dos famosos métodos: el de Booch y el OMT (*Object Modeling Technique*, Técnica de Modelado de Objetos). Más tarde se les unió Ivar Jacobson, creador del método OOSE (*Object-Oriented Software Engineering*, Ingeniería de Software Orientada a Objetos). En respuesta a una petición de OMG (*Object Management Group*, asociación para fijar los estándares de la industria) para definir un lenguaje y una notación estándar del lenguaje de construcción de modelos, en 1997 propusieron el UML como candidato.

Prescindiendo de la aceptación que pueda tener, este lenguaje recibió la aprobación *de facto* en la industria, pues sus creadores representan métodos muy difundidos de la primera generación del análisis y diseño orientados a objetos. Muchas organizaciones dedicadas al desarrollo de software y los proveedores de herramientas de CASE (Computer Aided Software Engineering) lo adoptaron, y muy probablemente se con-

vertirá en el estándar mundial que utilizarán los desarrolladores, los autores y los proveedores de herramientas de CASE.

Este libro no incluye todos los detalles de UML, un conjunto relativamente amplio de notaciones. Se centra en los diagramas de uso frecuente y en las características que más se utilizan dentro de ellos.

Si el lector desea un estudio exhaustivo de la notación UML en sus versiones iniciales, puede consultar la especificación completa en el sitio de Web de Rational Corporation: www.rational.com

Hay por los menos 10 notaciones diferentes de los elementos del análisis y diseño orientados a objetos; ello dificulta una comunicación eficaz y uniforme, la capacidad de aprender y las herramientas de CASE. Los autores del lenguaje UML —Booch, Jacobson y Rumbaugh— hicieron un excelente servicio a la comunidad de la tecnología de objetos al crear un lenguaje estandarizado de modelado que es elegante, expresivo y flexible.

El UML es un lenguaje para construir modelos; no guía al desarrollador en la forma de realizar el análisis y diseño orientados a objetos ni le indica cuál proceso de desarrollo adoptar.

En consecuencia, los metodólogos seguirán definiendo técnicas, modelos y procesos de desarrollo para la creación eficaz de sistemas de software; sólo que ahora pueden hacerlo en un lenguaje común: el UML.

INTRODUCCIÓN A UN PROCESO DE DESARROLLO

Objetivos

- Explicar el motivo del orden de los capítulos posteriores, que siguen los pasos del proceso que se describen en este capítulo.
- Seguir un proceso simple de desarrollo desde los requerimientos hasta la implementación.

2.1

Introducción

Un **proceso de desarrollo de software** es un método de organizar las actividades relacionadas con la creación, presentación y mantenimiento de los sistemas de software.

En este capítulo se da una muy breve introducción a las actividades fundamentales de un proceso, ofreciendo una guía de los pasos principales. Hemos incluido aquí esta explicación por dos motivos:

- En los capítulos subsecuentes se abordan el análisis y el diseño orientados a objetos a partir de esas actividades.
- Al seguir un proceso similar al que presentamos aquí, se sientan las bases para crear un proyecto de desarrollo manejable, reproducible y exitoso.

En el capítulo 37 encontrará el lector más temas concernientes al proceso.

2.1.1 Proceso y modelos que se recomiendan

El lenguaje UML no define un proceso estándar. Sus creadores admiten la importancia de contar con un lenguaje y un proceso muy sólidos para la construcción de modelos. Ofrecen su recomendación de lo que constituye un proceso adecuado en publicaciones aparte de las dedicadas al UML, porque la estandarización del proceso rebasaba entonces el ámbito de la definición del lenguaje.

Tampoco se prescribe en este libro un estándar o proceso nuevo.

Más importante que seguir un proceso o método oficiales es que el desarrollador adquiera habilidades que le permitan crear un buen diseño, y que las organizaciones favorezcan este tipo de desarrollo de destrezas. Esto se logra dominando varios principios y heurísticos que permiten identificar y abstraer los objetos idóneos, asignándoles después responsabilidades.

En este libro se ofrece una visión bastante representativa de los mejores métodos, de las actividades y modelos básicos en un proceso de desarrollo para sistemas orientados a objetos que se funden en un enfoque iterativo e incremental de los casos de uso. Los pasos y modelos recomendados constituyen, más que una fórmula, un *recorrido* por el panorama de desarrollo de software por medio de la tecnología de objetos. Estas actividades se incluyen como punto de partida para exponer, experimentar y crear un proceso adecuado a las necesidades concretas de la empresa del lector.

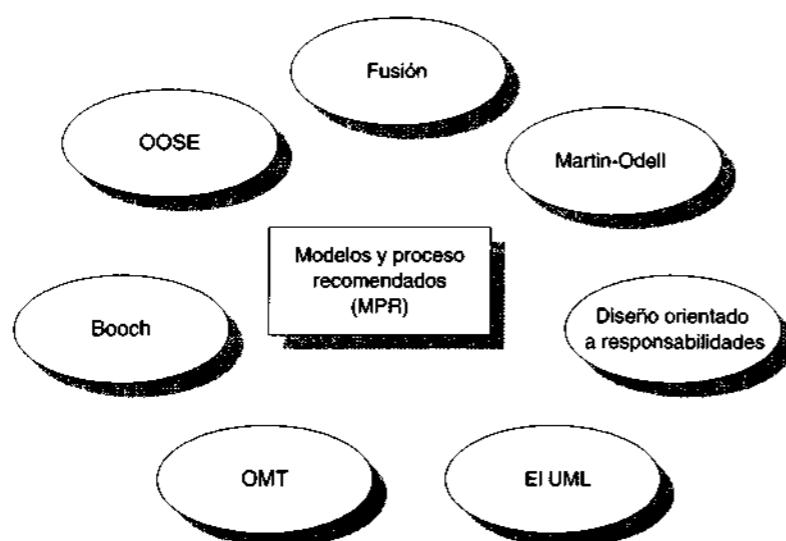


Figura 2.1 Factores que influyen en el proceso y los modelos recomendados en el libro.

No se trata de un método nuevo, sino la descripción de un proceso y de modelos generalmente recomendados (MPR) que utilizan los profesionales y que, con diversos nombres y ligeras modificaciones, se incluyen en otros métodos del análisis y diseño orientado a objetos. [Rumbaugh97]. Para simplificar la explicación, en ocasiones utilizaremos las siglas *MPR* (modelos y proceso recomendados) y también para recalcar la naturaleza cíclica del desarrollo iterativo.

Los factores que más han influido en los MPR aquí descritos (figura 2.1) son la experiencia personal del autor en el desarrollo, el trabajo dedicado a ObjectSpace, el propio UML, Fusion [Coleman94], Martin-Odell [MO95], Responsability-Driven Design [Wirfs-Brock90], OOSE (Objectory) [Jacobson92], OMT [Rumbaugh91] y Booch [Booch94].

2.1.2 Alcance

La descripción de un proceso incluye fundamentalmente las actividades que abarcan desde los requerimientos hasta la presentación o entrega. Además un proceso completo aborda puntos más amplios relacionados con la industrialización del desarrollo de software: ciclo de vida de un producto a largo plazo, documentación, soporte y capacitación, trabajo en paralelo y coordinaciones entre los participantes.¹ En esta introducción se ponen de relieve sólo las actividades básicas, sin entrar en muchos detalles.

En nuestro examen, se prescindirá de algunos pasos y aspectos esenciales del proceso: concepción, planeación, interacción de equipos en paralelo, administración del proyecto, documentación y realización de pruebas.

Omitiremos los pasos anteriores porque son ajenos al propósito fundamental del libro —aprender y aplicar habilidades en el análisis y diseño orientados a objetos— o porque los temas son demasiado extensos como para investigarlos de manera satisfactoria.

2.2 El lenguaje UML y los procesos de desarrollo

El lenguaje UML estandariza los artefactos y la notación, pero no define un proceso oficial de desarrollo. He aquí algunas de las razones que explican esto:

1. Aumentar las probabilidades de una aceptación generalizada de la *notación* estándar del modelado, sin la obligación de adoptar un proceso oficial.
2. La esencia de un proceso apropiado admite mucha variación y depende de las habilidades del personal, de la razón investigación-desarrollo, de la naturaleza del problema, de las herramientas y de muchos otros factores.

¹ Jacobson distingue entre *método*, que abarca los pasos individuales y secuenciales del desarrollo, y *proceso*, que amplía el método para aplicarlo a cuestiones más amplias de industrialización y de desarrollo en equipo [Jacobson92]. Aunque ésta es una distinción importante, no subrayaré la distinción para no complicar aún más un área ya saturada de tecnicismos.

Una vez aclaradas estas razones, procedemos a aplicar los principios generales y los pasos normales que guían un proceso eficaz.

2.3 Pasos de macrónivel

En un nivel alto, los pasos principales en la presentación de una aplicación son los siguientes (figura 2.2):

1. **Planeación y elaboración:** planear, definir los requerimientos, construir prototipos, etcétera.
2. **Construcción:** la creación del sistema.
3. **Aplicación:** la transición de la implementación del sistema a su uso.

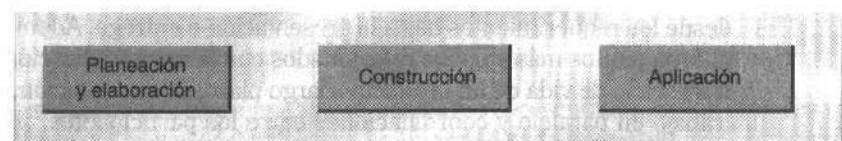


Figura 2.2 Pasos de macrónivel en el desarrollo.

2.4 Desarrollo iterativo

Un ciclo de vida iterativo se basa en el agrandamiento y perfeccionamiento secuencial de un sistema a través de *múltiples* ciclos de desarrollo de análisis, diseño, implementación y pruebas.

El sistema crece al incorporar nuevas funciones en cada ciclo de desarrollo. Tras una fase preliminar de planeación y especificación, el desarrollo pasa a la fase de construcción a través de una serie de ciclos de desarrollo.

En cada ciclo se aborda un conjunto relativamente pequeño de requerimientos, pasando por el análisis, el diseño, la construcción y las pruebas (figura 2.3). El sistema va creciendo con cada ciclo que concluye.

Esto contrasta con el ciclo clásico de la vida en cascada, en el cual las actividades (análisis y diseño, entre otras) se llevan a cabo una vez con todos los requerimientos del sistema.

Entre las ventajas del desarrollo iterativo figuran las siguientes:

- La complejidad nunca resulta abrumadora.
- Se produce retroalimentación en una etapa temprana, porque la implementación se efectúa rápidamente con una parte pequeña del sistema.

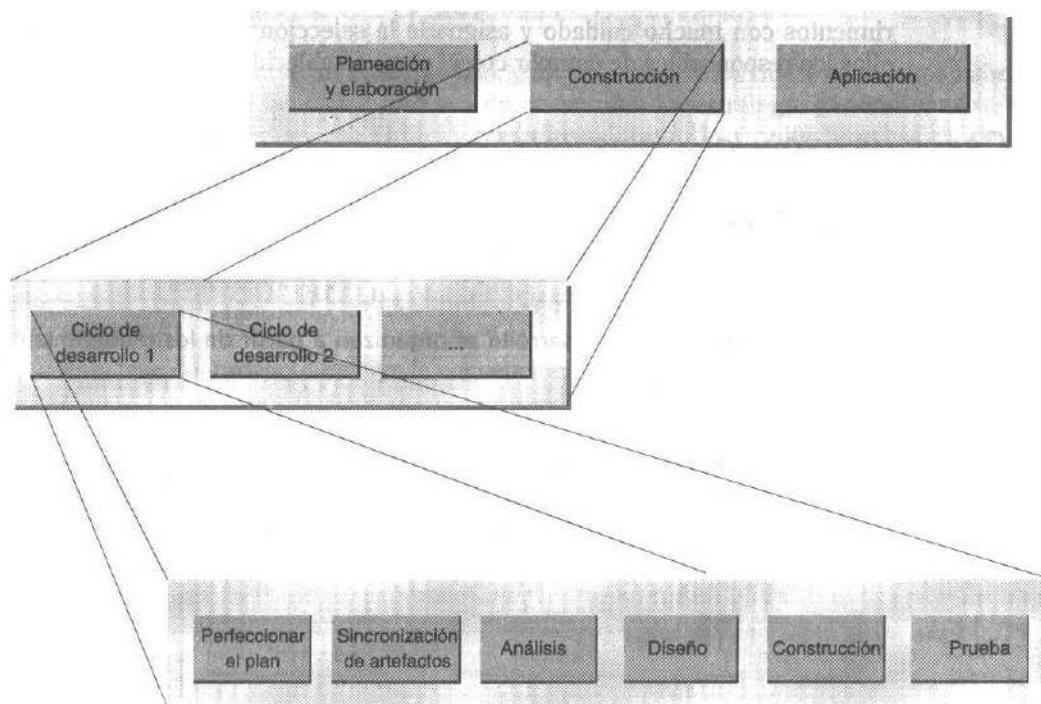


Figura 2.3 Ciclos iterativos de desarrollo.

2.4.1 Fijación de la duración de un ciclo de desarrollo (Time-boxing)

Una estrategia muy útil en los ciclos de desarrollo consiste en limitarlo a un marco temporal, esto es, un lapso rígidamente fijo, digamos cuatro semanas. Todo el trabajo ha de concluirse en ese lapso. Un periodo entre dos semanas y dos meses suele ser conveniente. En un periodo menor sería muy difícil terminar las actividades; en un periodo mayor la complejidad se torna abrumadora y la retroalimentación se retrasa (figura 2.4).

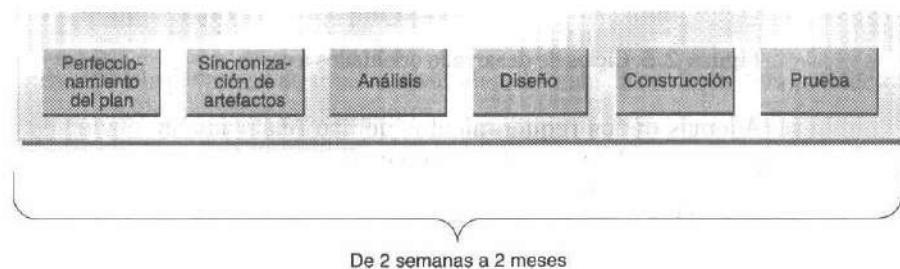


Figura 2.4 Fijación de la duración de un ciclo de desarrollo.

Para tener éxito con un programa de duración fija es necesario escoger los requerimientos con mucho cuidado y asignarle la selección al equipo del desarrollo: son ellos los responsables de cumplir con el plazo establecido.

2.4.2 Casos de uso y los ciclos iterativos del desarrollo

Un caso de uso es una descripción narrativa de un proceso de dominio; por ejemplo, *Obtener libros prestados en una biblioteca*.

Los ciclos iterativos de desarrollo se organizan a partir de los requerimientos del caso de uso.

Dicho de otra manera, se asigna un ciclo de desarrollo para implementar uno o más casos de uso o bien sus versiones simplificadas (por cierto muy comunes cuando el caso completo sería demasiado complejo de abordar en un ciclo) (figura 2.5).

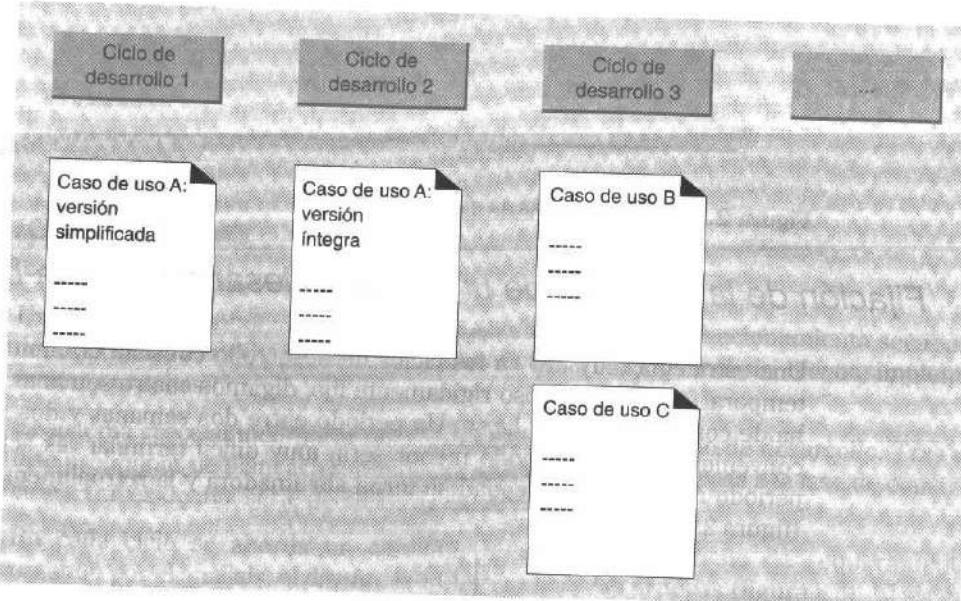


Figura 2.5 Ciclos de desarrollo orientados a casos.

Además de los requerimientos de uso relativamente evidentes que es preciso tener presentes, algunos ciclos —especialmente los primeros— han de centrarse en requerimientos poco evidentes; por ejemplo, la creación de servicios de soporte (persistencia, seguridad y otros).

2.4.3 Clasificación de los casos de uso

Deberían clasificarse los casos de uso, y los que ocupen los niveles más altos habrían de abordarse en los ciclos iniciales de desarrollo. La estrategia general consiste en seleccionar los casos que influyen profundamente en la arquitectura básica, dando soporte al dominio y a las capas de servicios de alto nivel o los que presentan el máximo riesgo.

2.5

La fase de la planeación y de la elaboración

Esta fase del proyecto incluye la concepción inicial, la investigación de alternativas, la planeación, la especificación de requerimientos y otras actividades.

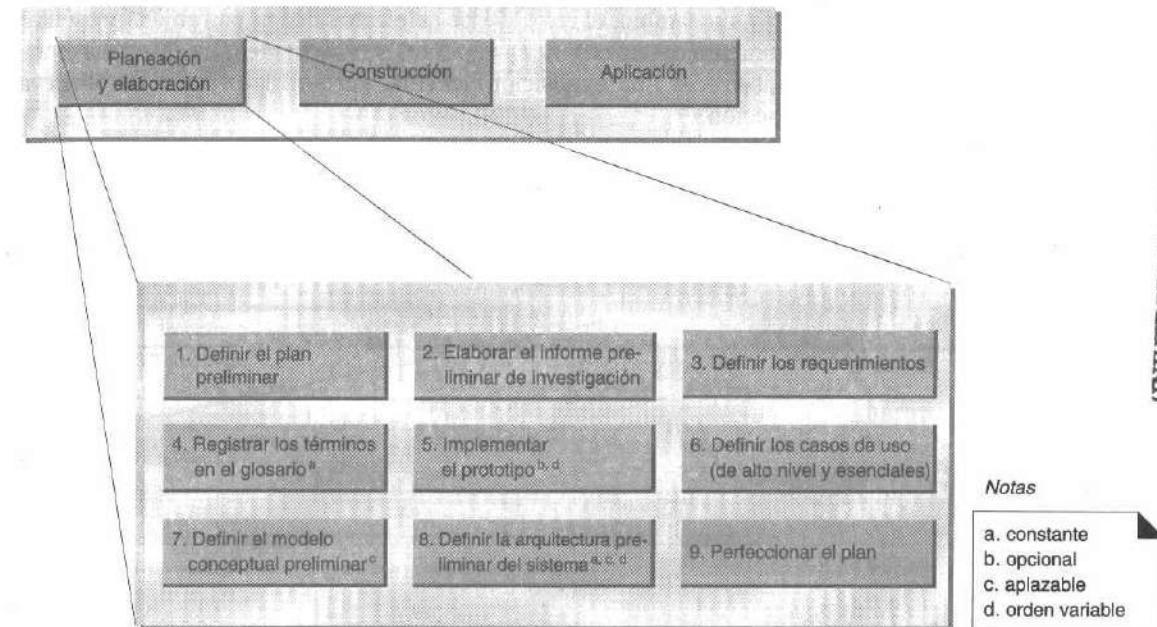


Figura 2.6 Ejemplo de actividades de la fase de planeación y elaboración.

En la figura 2.6 observamos algunas actividades de esta fase. Entre los artefactos generados aquí podemos citar los siguientes:

- *Plan*: programa, recursos, presupuesto, etcétera.
- *Informe preliminar de investigación*: motivos, alternativas, necesidades de la empresa.
- *Especificación de requerimientos*: declaración de los requerimientos.
- *Glosario*: diccionario (nombres de conceptos, por ejemplo) y toda información afín, como las restricciones y las reglas.

- *Prototipo*: sistema de prototipos cuyo fin es facilitar la comprensión del problema, los problemas de alto riesgo y los requerimientos.
 - *Casos de uso*: descripciones narrativas de los procesos de dominio.
 - *Diagramas de casos de uso*: descripción gráfica de todos los casos y de sus relaciones.
 - *Bosquejo del modelo conceptual*: modelo conceptual preliminar cuya finalidad es facilitar el conocimiento del vocabulario del dominio, especialmente en su relación con los casos de uso y con las especificaciones de los requerimientos.

2.5.1 Orden de creación de los artefactos o elementos del desarrollo

Aunque la guía que se muestra en la figura 2.6 puede sugerir un orden lineal en la creación de los artefactos, no es estrictamente así. Podemos preparar en paralelo algunos de ellos, por ejemplo. Esto se aplica especialmente al modelo conceptual, al glosario, a los casos de uso y a los diagramas de los casos. Los casos de uso que son sometidos a examen y, en cambio, el resto de los artefactos tienen por objeto presentar la información proveniente de los casos. En capítulos subsecuentes, los describiremos en orden lineal para ofrecer una exposición sencilla. Pero en la práctica se observa mucha mayor interacción.

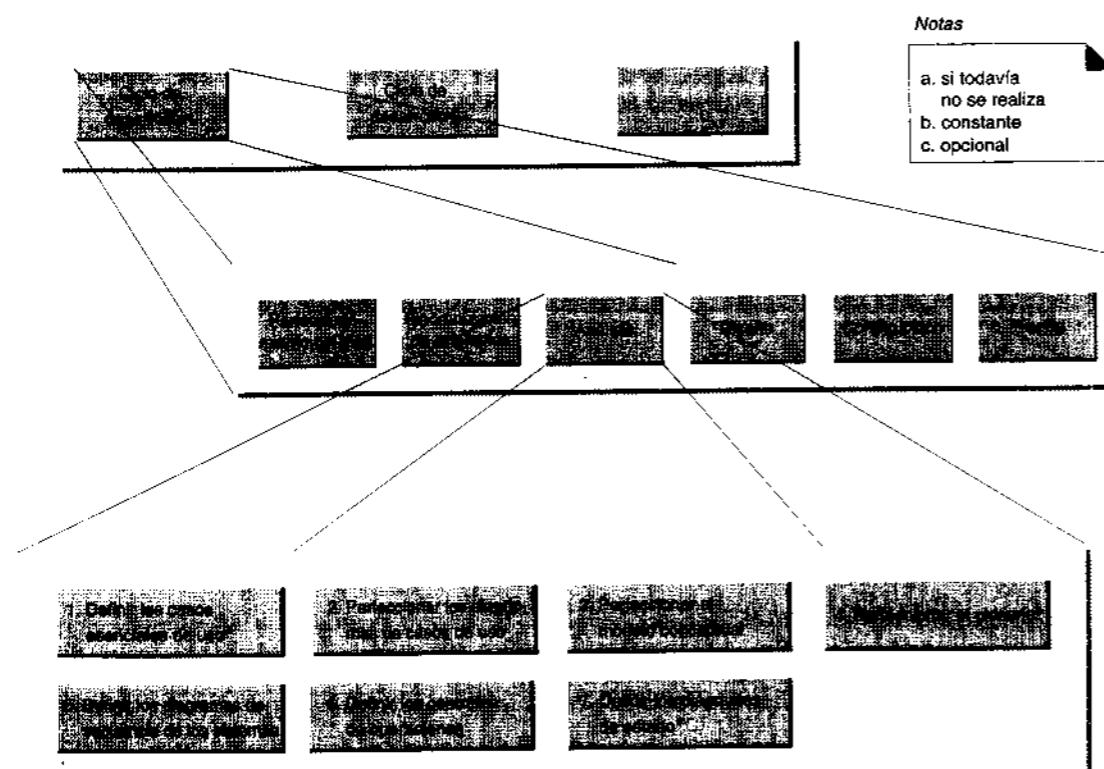


Figura 2.7 Ejemplo de las actividades en la fase de análisis.

La fase de construcción: ciclos del desarrollo

La fase de construcción de un proyecto requiere varios ciclos de desarrollo (probablemente con plazos fijos) a lo largo de los cuales se extiende el sistema. El objetivo final es obtener un sistema funcional de software que atienda debidamente los requerimientos.

En un ciclo individual de desarrollo, los principales pasos se analizan y diseñan, como se señala en las figuras 2.7 y 2.8. Los detalles de los pasos se comentan de manera por-
menorizada en los siguientes capítulos y en el capítulo 37, donde examinaremos otros
aspectos del proceso.

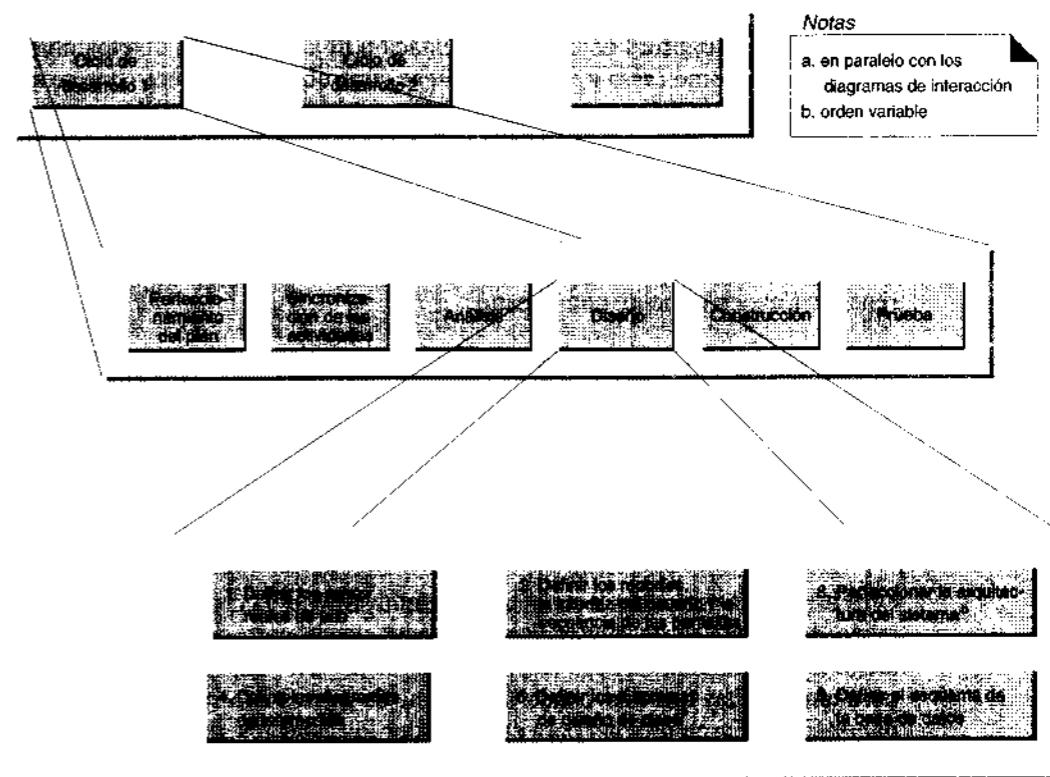


Figura 2.8 Ejemplo de las actividades de la fase de diseño.

2.6.1 Orden de creación de los artefactos en el ciclo de desarrollo

Como en el caso de los artefactos en la fase de requerimientos, el orden lineal deducible de la figura 2.7 no es el que se sigue rigurosamente. Algunos artefactos pueden elaborarse en paralelo; por ejemplo:

- Crear en paralelo un modelo conceptual y un glosario.
- Crear en paralelo los diagramas de interacción y los de diseño de clases.

2.7 Decidir cuándo crear artefactos

En la fase inicial de planeación y elaboración pueden crearse ciertos artefactos, entre ellos un modelo conceptual preliminar (un modelo de conceptos del mundo real) y casos expandidos de uso (descripciones narrativas detalladas de procesos). En la presente sección se examina el momento de su creación.

2.7.1 Cuándo crear el modelo conceptual

El **modelo conceptual** es una representación de conceptos u objetos en el dominio del problema, como *Libro* y *Biblioteca*. Debe controlarse el esfuerzo que se aplique a la producción del modelo conceptual preliminar durante la fase de planeación y elaboración. La meta es lograr un conocimiento básico del vocabulario y de los conceptos que se incluyen en los requerimientos. Por ello, no hace falta una investigación exhaustiva, pues se correría el riesgo de saturar demasiado la investigación desde el principio: exceso de complejidad. En los dominios de problemas amplios —por ejemplo, un sistema de reservaciones para las líneas aéreas—, un modelo conceptual muy completo resultaría excesivamente complicado.

La estrategia intermedia que recomendamos es generar rápidamente un modelo conceptual que se centre en identificar los conceptos obvios expresados en los requerimientos y posponer para más tarde una investigación con detenimiento. Más adelante, en cada ciclo de desarrollo, iremos refinando el modelo conceptual y ampliando los requerimientos referentes al ciclo.

Otra estrategia consiste en suspender definitivamente la creación del modelo hasta el inicio de los ciclos de desarrollo; comenzar desde cero en el ciclo de desarrollo y luego ir ampliándolo en cada ciclo. Se obtiene así la ventaja de aplazar la complejidad, pero también hay una desventaja: se dispone de menos información inicial, que pudiera haber sido de gran utilidad para comprender los aspectos generales, para entender el glosario, al realizar un examen meticuloso y al efectuar estimaciones. En el estudio de casos importantes se adopta este enfoque de posposición en la creación del modelo conceptual, no porque sea nuestro favorito sino porque favorece la explicación gradual de cómo producirlos.

2.7.2 Cuándo crear los casos expandidos de uso

Los **casos de uso de alto nivel** son muy breves, generalmente descripciones de un proceso en dos o tres oraciones. Los **casos expandidos de uso** son descripciones extensas que pueden contener cientos de oraciones con las cuales se realiza la descripción.

Durante la fase de planeación y elaboración, le aconsejamos crear todos los casos de uso de alto nivel, pero escribir sólo los más importantes en un formato expandido (largo), posponiendo el resto hasta el ciclo de desarrollo en que se estudian.

Igual que con el modelo conceptual, la adquisición temprana de información no está exenta de desventajas, pues puede originar una enorme complejidad. Investigar y escribir detalladamente en la fase de planeación y elaboración los casos de uso expandidos ofrece la ventaja de suministrar más información; ésta puede facilitar la compresión, la administración del riesgo, el examen meticuloso y las estimaciones. Pero también ofrece la desventaja de una excesiva complejidad inicial: la investigación producirá miles de detalles nimios. Más aún, los casos expandidos tal vez no sean muy confiables porque la información es incompleta o errónea y porque los requerimientos pueden cambiar constantemente.

Por tanto, la estrategia intermedia que recomendamos es investigar detenidamente, durante la fase de planeación y elaboración, sólo los casos de uso más importantes.

DEFINICIÓN DE MODELOS Y ARTEFACTOS

Objetivos

- Definir los modelos de análisis y de diseño.
- Explicar con ejemplos las dependencias entre los artefactos de análisis y diseño.

3.1 Introducción

En este capítulo se exponen ejemplos de modelos en el análisis y diseño orientados a objetos, así como de la relación de subordinación entre los artefactos en los modelos.

El propósito es ofrecer al lector un panorama general; en capítulos posteriores profundizaremos en los detalles de los modelos y artefactos.

3.2 Sistemas de construcción de modelos

Un sistema (tanto en el mundo real como en el mundo del software) suele ser extremadamente intrincado; por ello es necesario dividir el sistema en partes o fragmentos si queremos entender y administrar su complejidad. Estas partes podemos representarlas como **modelos** que describan y abstraigan sus aspectos esenciales [Rumbaugh97].

Por tanto, un paso útil en la construcción de un sistema de software es el de crear modelos que organicen y comuniquen los detalles importantes del problema de la vida

real con que se relacionan y del sistema a construir. Los modelos deben contener elementos cohesivos y estrechamente interconexos.

UML son las siglas de Unified *Modeling* Language (Lenguaje Unificado de Construcción de Modelos), porque su finalidad es describir modelos de sistemas (del mundo real y del mundo del software), basados en los conceptos de objetos.

Los modelos se componen de otros modelos o **artefactos**, de diagramas y documentos que describen cosas. El UML especifica varios diagramas, entre ellos los diagramas de casos de uso y los diagramas de interacción, que son los artefactos concretos a partir de los cuales creamos los modelos. Estos se visualizan por medio de las **vistas** —proyecciones visuales del modelo—; los diagramas de UML, entre ellos los de interacción, abarcan las vistas de un modelo.

Si queremos caracterizar los modelos, podemos poner de manifiesto la información **estática** o **dinámica** de un sistema. Un **modelo estático** describe las propiedades estructurales del sistema; en cambio, un **modelo dinámico** describe las propiedades de comportamiento de un sistema.

3.3

Modelos muestra

El UML es un lenguaje flexible de modelado que permite definir modelos arbitrarios. En esta sección presentaremos ejemplos de modelos de análisis y de diseño que explican la pertenencia de artefactos concretos a los modelos.

Se trata de modelos *muestra*; no se pretende en absoluto que sean definitivos.

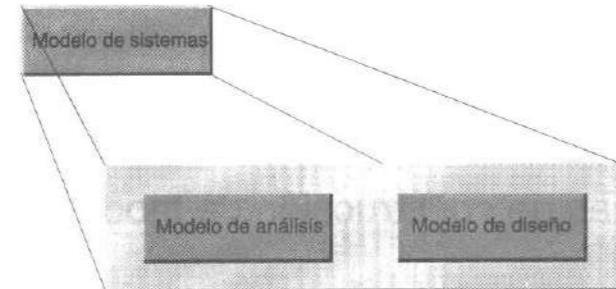


Figura 3.1 El modelo de sistemas.

3.3.1 El modelo del sistema

En este ejemplo el modelo global del sistema (figura 3.1) está constituido por el:

- **Modelo de análisis:** el que se relaciona con una investigación del dominio y del ámbito del problema, pero no con la solución.
 - **Modelo de diseño:** el que se relaciona con la solución lógica.

En capítulos subsecuentes se estudian estos modelos (figura 3.1) con mayor detalle.

3.4

Relación entre los artefactos

Sin importar cómo los artefactos se organicen para construir modelos, se dan dependencias muy importantes entre ellos. Por ejemplo, un diagrama de casos de uso que muestre todos los casos depende de las definiciones de los casos. Conviene entender la dependencia y la influencia entre los artefactos para poder efectuar comprobaciones de consistencia y de rastreabilidad y para poder utilizar eficazmente los artefactos dependientes como punto de partida para crear otros.¹

Por ejemplo, en la figura 3.2 se observan gráficamente las dependencias entre artefactos generados durante la fase de planeación y elaboración. En los siguientes capítulos trataremos más a fondo del significado de los artefactos y sus dependencias.

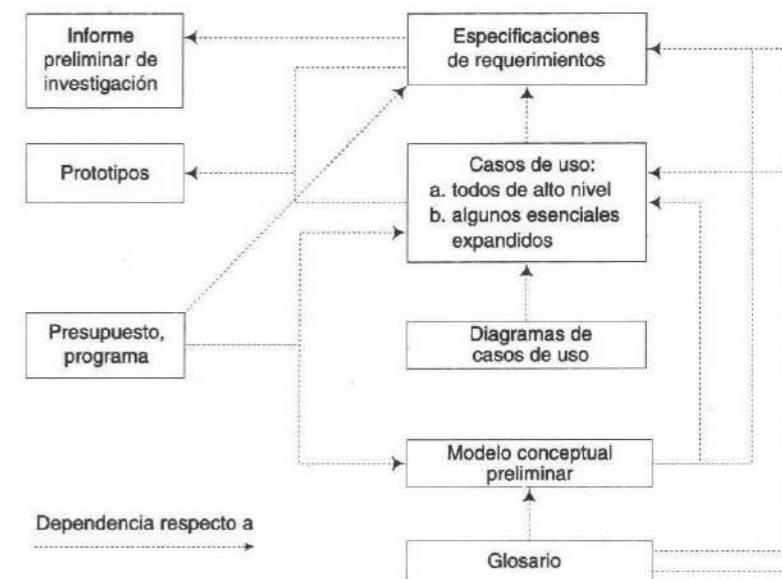


Figura 3.2 Influencia de los artefactos en la fase de planeación y de elaboración.

¹ Si se crea un artefacto que no tenga otros dependientes y si no se usa como entrada de otra cosa, habrá que poner en tela de juicio su valor y el tiempo que se dedicó a su creación.

**PARTE II FASE DE
PLANEACIÓN Y
DE ELABORACIÓN**

CASO DE ESTUDIO: EL PUNTO DE VENTA

Objetivos

- Definir el caso de estudio que se emplea en el libro.

4.1 El sistema del punto de venta

Nuestro principal caso de estudio es el sistema de una terminal (POST) de punto de venta.¹ Esta terminal es un sistema computarizado con el que se registran las ventas y se realizan los pagos; normalmente se utiliza en las tiendas al detalle. Abarca componentes de hardware (una computadora y un lector de código de barras) y software para correr el sistema.

Suponga que se nos ha pedido crear un programa para una terminal de punto de venta. Con una estrategia de desarrollo de incremento iterativo, vamos a realizar las fases de requerimientos, análisis y diseño orientados a objetos e implementación.



Figura 4.1 Terminal de punto de venta.

¹ Un problema también examinado en [Coad95], aunque este trabajo se llevó a cabo de manera independiente y mucho antes que el otro.

¿Por qué escogimos este problema? Por ser representativo de muchos sistemas de información y porque toca problemas comunes que puede encontrar un desarrollador. Ahondaremos lo suficiente en el análisis y en el diseño, para que el lector vea en forma detallada cómo se abordan estos problemas y pueda aplicar el método a otros proyectos.

4.2 Capas arquitectónicas y el énfasis en el caso de estudio

Un sistema típico de información que incluya una interfaz gráfica del usuario y acceso a la base de datos suele presentar un diseño arquitectónico de varios niveles o capas (figura 4.2) como las siguientes:

- **Presentación:** interfaz gráfica; ventanas.
- **Lógica de aplicación - Objetos del dominio del problema:** objetos que representan conceptos del dominio (los objetos de ventas, por ejemplo) que cumplen con los requisitos de aplicación.
- **Lógica de aplicación - Objetos de servicio:** objetos de dominio no relacionados con el problema que prestan servicios de soporte; por ejemplo, interfaz con una base de datos.
- **Almacenamiento:** un mecanismo persistente de almacenamiento; por ejemplo una base de datos relacional u orientada a objetos.

El análisis y diseño orientados a objetos generalmente son más útiles para modelar los niveles lógicos de la aplicación.

Estudiaremos en el capítulo 22 el tema de una arquitectura de capas (o niveles).

El caso de estudio del punto de venta destaca principalmente los objetos del dominio del problema, asignándoles responsabilidades para cumplir con los requisitos de la aplicación. En el capítulo 38 nos serviremos del diseño orientado a objetos para crear un conjunto de objetos del nivel de servicio para conectarnos con una base de datos.

En este método de diseño, la capa de presentación tiene muy poca responsabilidad; se dice que es *delgada*. Las ventanas *no* contienen un código que se encargue de la lógica o procesamiento de la aplicación. Por el contrario, las solicitudes de tarea se envían al dominio del problema y a las capas de servicio.

4.3 Nuestra estrategia: aprendizaje y desarrollo iterativos

Este libro está organizado para seguir una estrategia de desarrollo iterativo. El análisis y el diseño orientados a objetos se aplican al sistema del punto de venta en dos ciclos de desarrollo iterativo en que el primer ciclo se destina a una simple aplicación de las funciones básicas. El segundo amplía la funcionalidad del sistema (véase la figura 4.3).

Presentación



punto menor; explicar cómo conectarse a otras capas

Lógica de aplicación

-objetos del dominio del problema

Venta

Pago

punto primario del caso de estudio

-objetos de servicio

Intermediario-de-Basesdedatos

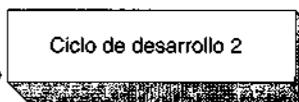
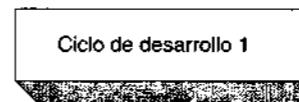
Administrador-de-Seguridad

punto secundario del caso de estudio

Almacenamiento



Figura 4.2 Capas de un sistema ordinario de información orientado a objetos.



En la primera gran sección del libro se estudian sólo las habilidades de análisis y diseño pertinente al primer ciclo.

Se explican más habilidades de análisis y de diseño.

Figura 4.3 Ruta de aprendizaje que siguen los ciclos de desarrollo.

Además del desarrollo iterativo, también se expone de manera iterativa la *presentación* de los temas del análisis y el diseño orientados a objetos, la notación de UML y los patrones. En el primer ciclo de desarrollo del sistema del punto de venta, se describe un grupo básico de temas de análisis y de diseño, así como la notación. El segundo ciclo se expande y ofrece nuevas ideas, la notación de UML y los patrones.

Esta estrategia tiene por objeto proponer un modelo de aprendizaje “justo a tiempo”. El modelo procura explicar primero las ideas de mayor uso, lo más cerca posible del momento en que el lector perciba la necesidad de aprenderlos para poder continuar. Al principio, el libro se centra en las ideas y habilidades fundamentales, sin saturarlo con nueva información que no pueda aplicar de inmediato. Después se explican las habilidades e ideas que se utilizan más raramente.

CONOCIMIENTO DE LOS REQUERIMIENTOS

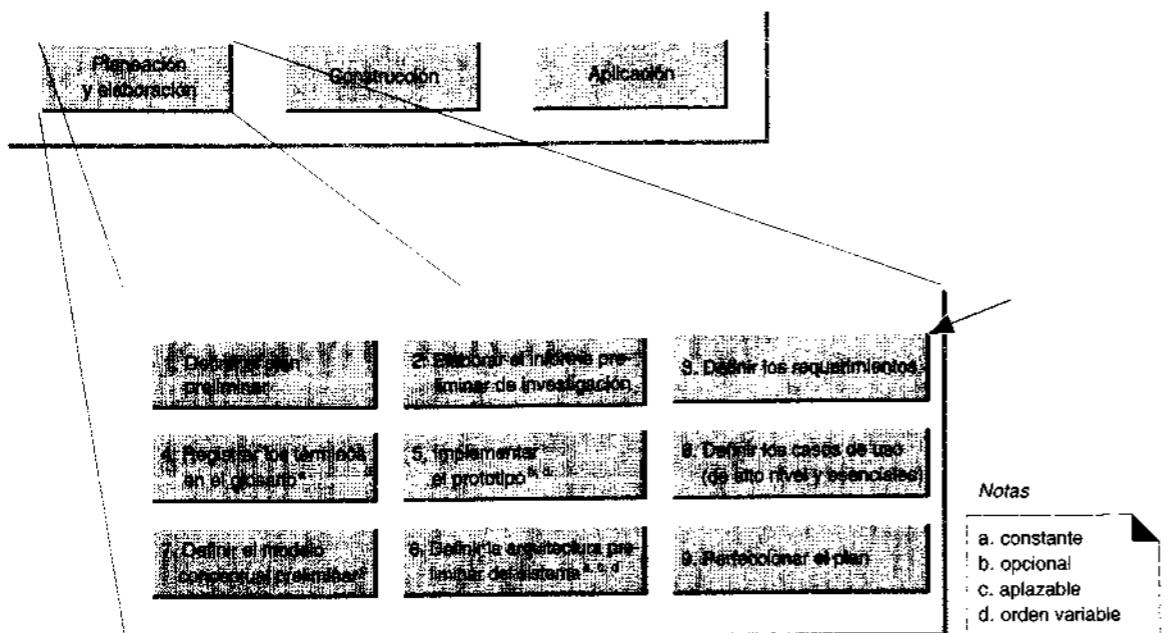
Objetivos

- Crear los artefactos de la fase de requerimientos; por ejemplo, las especificaciones de funciones.
- Identificar y clasificar las funciones del sistema.
- Identificar y clasificar los atributos del sistema y relacionarlos con las funciones.

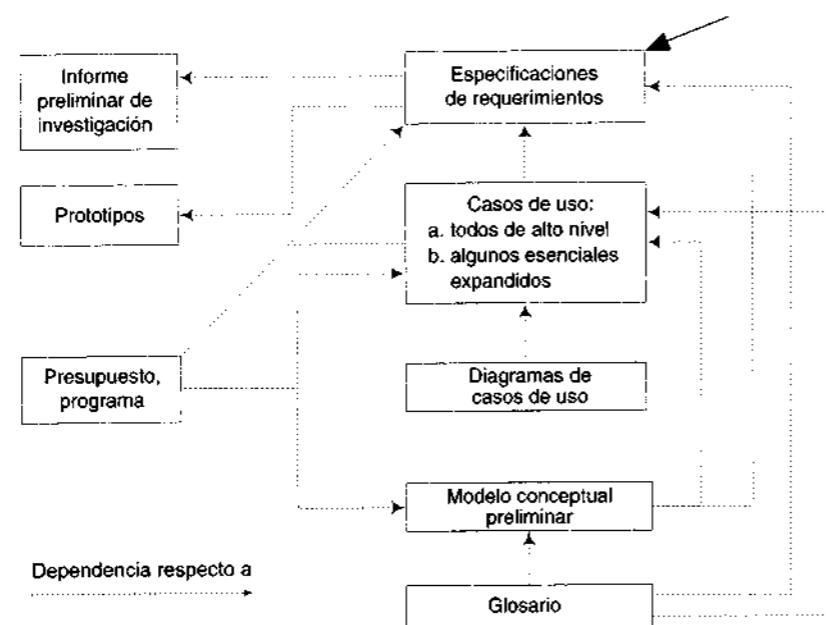
5.1 Introducción

Un proyecto no puede ser exitoso sin una especificación correcta y exhaustiva de los requerimientos. Para ello se necesitan muchas habilidades; un examen riguroso de ellas rebasa el ámbito de este libro, pues nuestro objetivo es que el lector domine el análisis y el diseño orientados a objetos. Pero se ofrece una introducción a los requerimientos de la aplicación punto de venta, porque en la práctica es un paso decisivo, y se mencionan otros artefactos relacionados con la fase de requerimientos. No dudamos en recomendar *Exploring Requirements: Quality Before Design* [GW89], obra que se centra en las habilidades necesarias para dilucidar los requerimientos importantes.

Este capítulo se propone lograr que el lector pueda expresar los requerimientos, no en convertirlo en un experto en el dominio de los sistemas de terminales para tiendas y puntos de venta. Por eso, la lista de las funciones y atributos del sistema no es exhaustiva, sino representativa.



Actividades de la fase de planeación y elaboración.



Dependencias de los artefactos respecto a la fase de planeación y elaboración.

Ninguno de los artefactos que se describen en la presente sección son propios del lenguaje UML; se trata simplemente de documentos comunes de la fase de requerimientos.

5.2 Los requerimientos

Los **requerimientos** son una descripción de las necesidades o deseos de un producto. La meta primaria de la fase de requerimientos es identificar y documentar lo que en realidad se necesita, en una forma que claramente se lo comunique al cliente y a los miembros del equipo de desarrollo. El reto consiste en definirlos de manera inequívoca, de modo que se detecten los riesgos y no se presenten sorpresas al momento de entregar el producto.

Se recomiendan los siguientes artefactos en la fase de requerimientos:

- panorama general
- clientes
- metas
- funciones del sistema
- atributos del sistema

Otros documentos pertinentes, que por cierto no examinaremos en el libro, se mencionan al final del capítulo.

5.2.1 Integración de las piezas del rompecabezas

En nuestro caso de estudio, la definición de los requerimientos aparece muy clara y tajante: la realidad dista mucho de serlo. Por lo regular hay que reunir y asimilar muchos estudios y documentos electrónicos, analizar los resultados de las entrevistas, celebrar reuniones para definir los requerimientos en grupo, etcétera.

5.3 Presentación general

Este proyecto tiene por objeto crear un sistema de terminal para el punto de venta que se utilizará en las ventas al menudeo.

5.4 Clientes

ObjectStore, Inc., detallista multinacional de objetos.

5.5 Metas

En términos generales, la meta es una mayor automatización del pago en las cajas registradoras, dar soporte a servicios más rápidos, más baratos y mejores y a los procesos de negocios. Más concretamente, la meta incluye:

- Pago rápido de los clientes.
- Análisis rápido y exacto de las ventas.
- Control automático del inventario.

5.6 Funciones del sistema

Las **funciones del sistema** son lo que éste habrá de *hacer*, por ejemplo autorizar los pagos a crédito. Hay que identificarlas y listarlas en grupos cohesivos y lógicos.

Con el objeto de verificar que algún X es de verdad una función del sistema, la siguiente oración deberá tener sentido:

El sistema deberá hacer <X>.

Por ejemplo: *El sistema deberá autorizar los pagos a crédito.*

En cambio, los **atributos del sistema** son cualidades no funcionales —entre ellas la facilidad de uso— que a menudo se confunden con las funciones. Nótese que “facilidad de uso” no encaja en la oración de verificación: *El sistema deberá hacer la facilidad de uso.* Los atributos no deben formar parte del documento de las especificaciones funcionales del sistema, sino de un documento independiente que especifica sus atributos.

5.6.1 Categorías de las funciones

Las funciones, como *autorizar pagos a crédito*, han de clasificarse a fin de establecer prioridades entre ellas e identificar las que de lo contrario pasarían inadvertidas (pero que consumen tiempo y otros recursos). Las categorías son:

Evidente	Debe realizarse, y el usuario debería saber que se ha realizado.
Oculta	Debe realizarse, aunque no es visible para los usuarios. Esto se aplica a muchos servicios técnicos subyacentes, como <i>guardar información en un mecanismo persistente de almacenamiento</i> . Las funciones ocultas a menudo se omiten (erróneamente) durante el proceso de obtención de los requerimientos.
Superflua	Opcionales; su inclusión no repercute significativamente en el costo ni en otras funciones.

5.6.2 Funciones básicas

Las siguientes funciones del sistema en la aplicación de la terminal del punto de venta son una muestra representativa; no pretenden en absoluto ser exhaustivas. Nuestro objetivo es entender los detalles del análisis y del diseño, no el funcionamiento de una tienda.

Nombre	Descripción	Categoría
R1.1	Registra la venta en proceso (actual): los productos comprados.	evidente
R1.2	Calcula el total de la venta actual; se incluyen el impuesto y los cálculos de cupón.	evidente
R1.3	Captura la información sobre el objeto comprado usando su código de barras y un lector o usando una captura manual de un código del producto; por ejemplo, un código universal de producto (UPC).	evidente
R1.4	Reduce las cantidades del inventario cuando se realiza una venta.	oculta
R1.5	Se registran las ventas efectuadas.	oculta
R1.6	El cajero debe introducir una identificación y una contraseña para poder utilizar el sistema.	evidente
R1.7	Ofrece un mecanismo de almacenamiento persistente.	oculta

Ref #	Función	Categoría
R1.8	Ofrece mecanismos de comunicación entre los procesos y entre los sistemas.	oculta
R1.9	Muestra la descripción y el precio del producto registrado.	evidente

5.6.3 Funciones de pago

Ref #	Función	Categoría
R2.1	Maneja los pagos en efectivo, capturando la cantidad ofrecida y calculando el saldo deudor.	evidente
R2.2	Maneja los pagos a crédito, capturando la información crediticia a partir de una lectora de tarjetas o mediante captura manual, y autorizando los pagos con el servicio de autorización (externa) de créditos de la tienda a través de una conexión por módem.	evidente
R2.3	Maneja los pagos con cheque, capturando la licencia de conducir mediante captura manual, y autorizando los pagos con el servicio de autorización (externa) de cheques de la tienda a través de la conexión por módem.	evidente
R2.4	Registra los pagos en el sistema de cuentas por cobrar, pues el servicio de autorización de crédito debe a la tienda el monto del pago.	oculta

5.7 Atributos del sistema

Los atributos del sistema son sus características o dimensiones; no son funciones. Por ejemplo:

facilidad de uso	tolerancia a las fallas	tiempo de respuesta
metáfora de interfaz	costo al detalle	plataformas

Los atributos del sistema pueden abarcar todas las funciones (por ejemplo, la plataforma del sistema operativo) o ser específicos de una función o grupo de funciones.

Los atributos tienen un posible conjunto de **detalles de atributos**, los cuales tienden a ser valores discretos, confusos o simbólicos; por ejemplo:

tiempo de respuesta = (psicológicamente correcto)

metáfora de interfaz = (gráfico, colorido, basado en formas)

Algunos atributos del sistema también pueden tener **restricciones de frontera del atributo**, que son condiciones obligatorias de frontera, generalmente en un rango numérico de los valores de un atributo; por ejemplo:

tiempo de respuesta = (cinco segundos como máximo)

He aquí algunos ejemplos más:

Atributo	Detalles y restricciones de frontera
tiempo de respuesta	(restricción de frontera) Cuando se registre un producto vendido, la descripción y el precio aparecerán en cinco segundos.
metáfora de interfaz	(detalle) Ventanas orientadas a la metáfora de una forma y cuadros de diálogo. (detalle) maximiza una navegación fácil con teclado y no con apuntadores.
tolerancia a fallas	(restricción de frontera) debe registrar los pagos a crédito autorizados que se hagan a las cuentas por cobrar en un plazo de 24 horas, aun cuando se produzcan fallas de energía o del equipo.
plataformas del sistema operativo	(detalle) Microsoft Windows 95 y NT.

5.7.1 Atributos del sistema en las especificaciones de funciones

Conviene describir todos los atributos del sistema que se relacionen claramente con las funciones *dentro* de la lista en que se especifican estas últimas. Además, los detalles de los atributos y las restricciones de frontera pueden catalogarse como *obligatorios u opcionales*.¹

¹ Una restricción de frontera suele ser *obligatoria*, pues de lo contrario significaría que no era sólida.

Identificación	Función	Característica	Entorno	Detalles de la descripción	Carácter
R1.9	Mostrar la descripción y el precio del producto registrado.	evidente	tiempo de respuesta	5 segundos como máximo	obligatorio
			metáfora de interfaz	pantallas basadas en formas colorido	obligatorio opcional
R2.4	Registrar los pagos a crédito en el sistema de cuentas por cobrar, pues el servicio de autorización de crédito debe a la tienda el importe del pago.	oculto	tolerancia a fallas	debe registrar en las cuentas por cobrar en un plazo de 24 horas, aun cuando se produzcan fallas de energía o del equipo	obligatorio
			tiempo de respuesta	10 segundos como máximo	obligatorio

5.8 Otros artefactos en la fase de los requerimientos

En este libro se da una introducción muy sucinta a los requerimientos; es un tema que bien podría abarcar libros enteros. Las funciones y los atributos del sistema son los documentos mínimos de los requerimientos, de modo que se necesitan otros artefactos importantes para atenuar el riesgo y entender el problema, a saber:

- *Requerimientos y equipos de enlace*: lista de los que deberían participar en la especificación de las funciones y atributos del sistema, en la realización de entrevistas, de pruebas, de negociaciones y de otras actividades.
- *Grupos afectados*: los que reciben el impacto del desarrollo o aplicación del sistema.
- *Suposiciones*: las cosas cuya verdad se supone.
- *Riesgos*: las cosas que pueden ocasionar el fracaso o retraso.
- *Dependencias*: otras personas, sistemas y productos de los cuales no puede prescindir el proyecto para su terminación.
- *Glosario*: definición de los términos pertinentes; tema que se estudiará en capítulos subsecuentes.
- *Casos de uso*: descripciones narrativas de los procesos del dominio; tema que se verá en capítulos posteriores.
- *Modelo conceptual preliminar*: modelo de conceptos importantes y de sus relaciones; tema que se tratará en capítulos posteriores.

CASOS DE USO: DESCRIPCIÓN DE PROCESOS

Objetivos

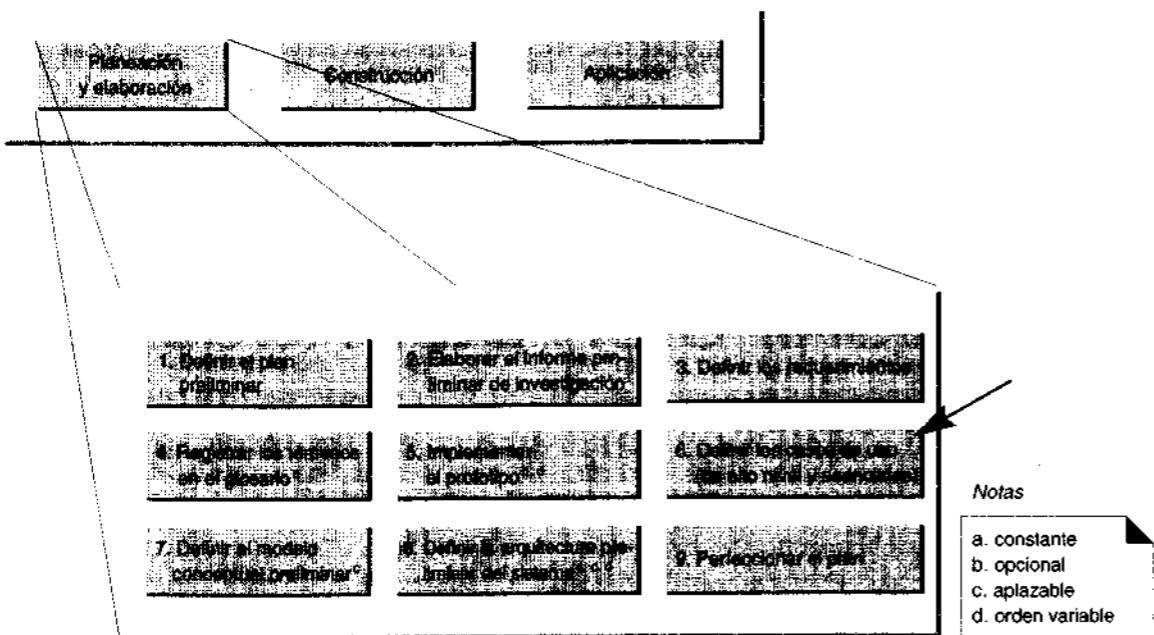
- Identificar y escribir casos de uso.
- Diseñar diagramas de casos de uso.
- Contrastar los casos de uso de alto nivel con los expandidos.
- Contrastar los casos de uso esenciales con los reales.

6.1 Introducción

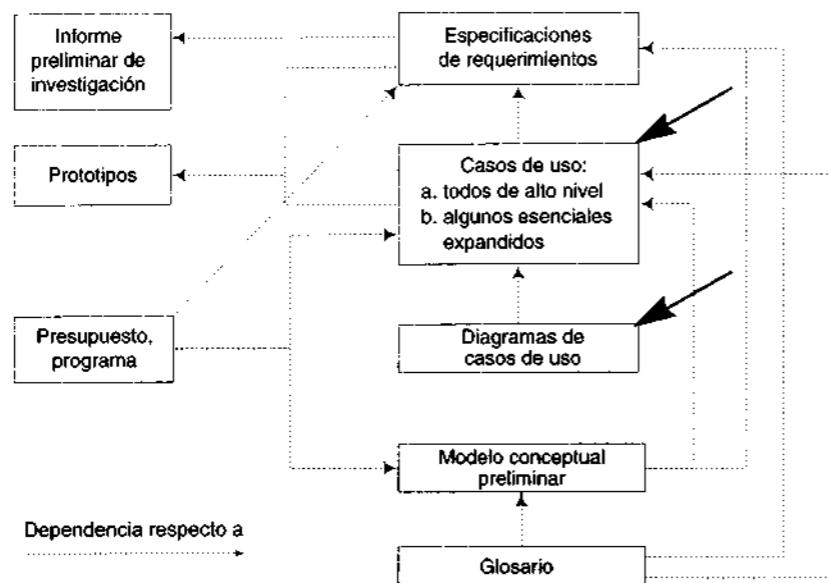
Una técnica excelente que permite mejorar la comprensión de los requerimientos es la creación de casos de uso, es decir, descripciones narrativas de los procesos del dominio. En este capítulo se exponen los conceptos básicos de esta técnica y se dan ejemplos para aplicarlos a la terminal de punto de venta.

En el siguiente capítulo clasificaremos los casos de uso y escogeremos los que utilizaremos en el primer ciclo de desarrollo.

El UML incluye formalmente el concepto de casos de uso y sus diagramas de uso.



Actividades de la fase de planeación y elaboración.



Dependencias de los artefactos respecto a la fase de planeación y elaboración.

6.2

Actividades y dependencias

Los casos de uso requieren tener al menos un conocimiento parcial de los requerimientos del sistema, en teoría expresados en el documento donde se especifican.

6.3

Casos de uso

El caso de uso es un documento narrativo que describe la secuencia de eventos de un actor (agente externo) que utiliza un sistema para completar un proceso [Jacobson92]. Los casos de uso son historias o casos de utilización de un sistema; no son exactamente los requerimientos ni las especificaciones funcionales, sino que ejemplifican e incluyen tácitamente los requerimientos en las historias que narran.

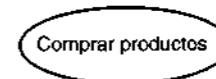


Figura 6.1 Icono del lenguaje UML para un caso de uso.

6.3.1

Ejemplo de un caso de uso de alto nivel: comprar productos

El siguiente caso de uso de alto nivel describe clara y concisamente el proceso de comprar artículos en una tienda cuando se emplea una terminal en el punto de venta.

Caso de uso:	Comprar productos
Actores:	Cliente, Cajero.
Tipo:	Primario (que se explicará luego).
Descripción:	Un Cliente llega a la caja registradora con los artículos que comprará. El Cajero registra los artículos y cobra el importe. Al terminar la operación, el Cliente se marcha con los productos.

Los encabezados y la estructura de este caso de uso son representativos. Sin embargo, el UML no especifica un formato rígido; puede modificarse para atender las necesidades y ajustarse al espíritu de la documentación: ante todo, una comunicación clara.

Conviene comenzar con los casos de uso de alto nivel para lograr rápidamente entender los principales procesos globales.

6.3.2 Ejemplo de un caso expandido de uso: comprar productos con efectivo

Un **caso expandido de uso** muestra más detalles que uno de alto nivel; este tipo de casos suelen ser útiles para alcanzar un conocimiento más profundo de los procesos y de los requerimientos. A menudo se llevan a cabo en un estilo “coloquial” entre los actores y el sistema [Wirfs-Brock93]. Damos en seguida un ejemplo de un caso expandido de *Comprar productos* que ha sido simplificado para manejar únicamente los pagos en efectivo y excluir la administración del inventario (lo hemos hecho para simplificar la explicación en este primer ejemplo).

Caso de uso:	Comprar productos en efectivo
Actores:	Cliente (iniciador), Cajero.
Propósito:	Capturar una venta y su pago en efectivo.
Resumen:	Un Cliente llega a la caja registradora con artículos que desea comprar. El Cajero registra los productos y recibe un pago en efectivo. Al terminar la operación, el Cliente se marcha con los productos comprados.
Tipo:	Primario y esencial.
Referencias cruzadas:	<i>Funciones:</i> R1.1, R1.2, R1.3, R1.7, R1.9, R2.1.

Curso normal de los eventos

Acción del actor	Respuesta del sistema
1. Este caso de uso comienza cuando un Cliente llega a una caja de TPDV (Terminal Punto de Venta) con productos que desea comprar.	
2. El Cajero registra el identificador de cada producto. Si hay varios productos de una misma categoría, el Cajero también puede introducir la cantidad.	3. Determina el precio del producto e incorpora a la transacción actual la información correspondiente. Se presentan la descripción y el precio del producto actual.
4. Al terminar de introducir el producto, el Cajero indica a TPDV que se concluyó la captura del producto.	5. Calcula y presenta el total de la venta.
6. El Cajero le indica el total al Cliente.	

Curso normal de los eventos

Acción del actor	Respuesta del sistema
7. El Cliente efectúa un pago en efectivo —el “efectivo ofrecido”— posiblemente mayor que el total de la venta.	
8. El Cajero registra la cantidad de efectivo recibida.	9. Muestra al cliente la diferencia. Genera un recibo.
10. El Cajero deposita el efectivo recibido y extrae el cambio del pago. El Cajero da al Cliente el cambio y el recibo impreso.	11. Registra la venta concluida.
12. El Cliente se marcha con los artículos comprados.	
Cursos alternos	
■ Línea 2: introducción de identificador inválido. Indicar error.	
■ Línea 7: el cliente no tenía suficiente dinero. Cancelar la transacción de venta.	

6.3.3 Explicación del formato expandido

La parte superior de la forma expandida es información muy sucinta.

Caso de uso:	Nombre del caso de uso
Actores:	Lista de actores (agentes externos), en la cual se indica quién inicia el caso de uso.
Propósito:	Intención del caso de uso.
Resumen:	Repetición del caso de uso de alto nivel o alguna síntesis similar.
Tipo:	1. Primario, secundario u opcional (a explicar). 2. Esencial o real (a explicar).
Referencias cruzadas:	Casos relacionados de uso y funciones también relacionadas del sistema.

La sección intermedia, *curso normal de los eventos*, es la parte modular del formato expandido; describe los detalles de la conversión interactiva entre los actores y el sistema. Un aspecto esencial de la sección es que explica la secuencia más común de los eventos: la historia normal de las actividades y la terminación exitosa de un proceso. No incluye situaciones alternas.

Curso normal de los eventos

Acción del actor	Respuesta del sistema
Acciones numeradas de los actores.	Descripciones numeradas de las respuestas del sistema.

La última sección, *curso alterno de los eventos*, describe importantes opciones o excepciones que pueden presentarse en relación con el curso normal. Si son complejas, podemos expandirlas y convertirlas en nuestros casos de uso.

Cursos alternos

- Alternativas que pueden ocurrir en el número de línea. Descripción de excepciones.

6.4 Actores

El actor es una entidad externa del sistema que de alguna manera participa en la historia del caso de uso. Por lo regular estimula el sistema con eventos de entrada o recibe algo de él. Los actores están representados por el papel que desempeñan en el caso: Cliente, Cajero u otro. Conviene escribir su nombre con mayúscula en la narrativa del caso para facilitar la identificación.



Cliente

Figura 6.2 Icono del lenguaje UML que representa un actor de casos de uso.¹

¹ Aunque el icono estándar es una figura humana estilizada, hay quienes prefieren utilizar un icono con figura de computadora para designar los actores que son sistemas de cómputo y no seres humanos.

En un caso de uso hay un **actor iniciador** que produce la estimulación inicial y, posiblemente, otros **actores participantes**; tal vez convenga indicar quién es el iniciador.

Los actores suelen ser los papeles representados por seres humanos, pero pueden ser cualquier tipo de sistema, como un sistema computarizado externo de bancos. He aquí algunos tipos:

- papeles que desempeñan las personas
- sistemas de cómputo
- aparatos eléctricos o mecánicos

6.5 Un error común en los casos de uso

Un error común en la identificación de los casos de uso consiste en representar los pasos, las operaciones o las transacciones individuales como casos. Por ejemplo, en el dominio de la terminal del punto de venta, podemos definir (incorrectamente) un caso denominado "Imprimir el recibo", cuando en realidad esta operación no es más que un paso de un proceso más amplio del caso *Comprar productos*.

Un caso de uso es una descripción de un proceso de principio a fin relativamente amplia, descripción que suele abarcar muchos pasos o transacciones; normalmente no es un paso ni una actividad individual del proceso.

Es posible dividir las actividades o parte del caso en subcasos (denominados **casos abstractos de uso**), incluso en pasos individuales; pero esto no es lo habitual y lo veremos en el capítulo 26.

6.6 Identificación de los casos de uso

Los siguientes pasos de la identificación de los casos de uso requieren una lluvia de ideas y revisar los documentos actuales sobre la especificación de los requerimientos.

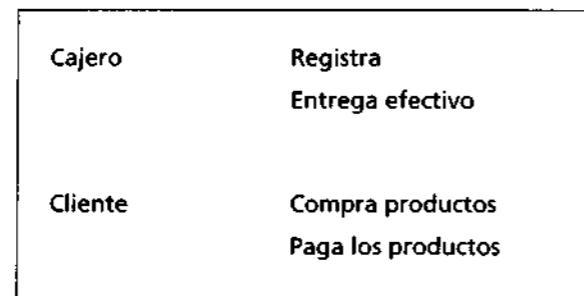
Un método con que se identifican los casos de uso se basa en los actores.

1. Se identifican los actores relacionados con un sistema o empresa.
2. En cada actor, se identifican los procesos que inician o en que participan.

Un segundo método de identificación de los casos de uso se basa en eventos.

1. Se identifican los eventos externos a los que un sistema ha de responder.
2. Se relacionan los eventos con los actores y con los casos de uso.

En la aplicación del punto de venta, algunos actores posiblemente relevantes y los procesos que inician son:



6.7 Caso de uso y procesos del dominio

Un caso de uso describe un proceso, un proceso de negocios por ejemplo. Un **proceso** describe, de comienzo a fin, una secuencia de los eventos, de las acciones y las transacciones que se requieren para producir u obtener algo de valor para una empresa o actor.

A continuación se mencionan algunos procesos:

- Retira efectivo en un cajero automático
- Ordena un producto
- Registra los cursos que se imparten en una escuela
- Verifica la ortografía de un documento con un procesador de palabras
- Realiza una llamada telefónica

6.8

Casos de uso, funciones del sistema y rastreabilidad

Las funciones del sistema identificadas durante la especificación previa de requerimientos deben asignarse a los casos de uso. Además, debe ser posible verificar, mediante la sección *Referencias cruzadas*, que todas las funciones hayan sido asignadas. Con ello se logra un vínculo importante respecto a la rastreabilidad entre los artefactos. En definitiva, todas las funciones y casos de uso del sistema deberían poder rastrearse hasta la implementación y la aplicación de pruebas.

6.9

Diagramas de los casos de uso

En la figura 6.3 se muestra un diagrama de casos de uso para el sistema del punto de venta.

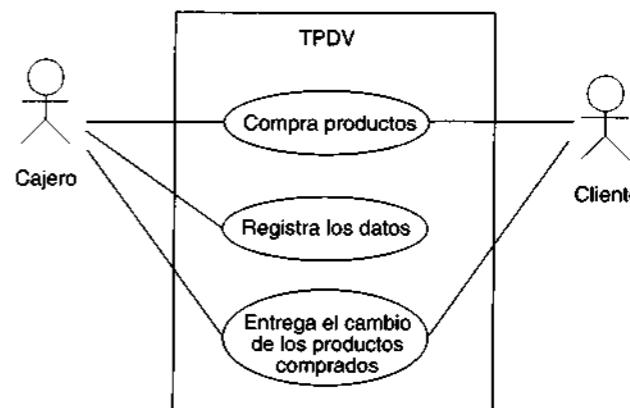


Figura 6.3 Diagrama parcial de casos de uso.

Un **diagrama de casos de uso** explica gráficamente un conjunto de casos de uso de un sistema, los actores y la relación entre éstos y los casos de uso. Estos últimos se muestran en óvalos y los actores son figuras estilizadas. Hay líneas de comunicaciones entre los casos y los actores; las flechas indican el flujo de la información o el estímulo.

El diagrama tiene por objeto ofrecer una clase de diagrama contextual que nos permite conocer rápidamente los actores externos de un sistema y las formas básicas en que lo utilizan.

6.10

Formatos de los casos de uso

En la práctica, los casos de uso pueden expresarse con diverso grado de detalle y de aceptación de las decisiones concernientes al diseño. En otras palabras, un mismo

caso de uso pueden escribirse en diferentes formatos y con diversos niveles de detalle. Más adelante estudiaremos otras formas de clasificarlos y expresarlos en formatos; pero por ahora nos concentraremos en una división fundamental: casos con formato de alto nivel y expandido.

6.10.1 Formato de alto nivel

Un **caso de uso de alto nivel** describe un proceso muy brevemente, casi siempre en dos o tres enunciados. Conviene servirse de este tipo de caso durante el examen inicial de los requerimientos y del proyecto, a fin de entender rápidamente el grado de complejidad y de funcionalidad del sistema. Estos casos son muy sucintos y vagos en las decisiones de diseño.

6.10.2 Formato expandido

Un **caso de uso expandido** describe un proceso más a fondo que el de alto nivel. La diferencia básica con el caso de uso de alto nivel consiste en que tiene una sección destinada al *curso normal de los eventos*, que los describe paso por paso. Durante la fase de especificación de requerimientos, conviene escribir en el formato expandido los casos más importantes y de mayor influencia; en cambio, los menos importantes pueden posponerse hasta el ciclo de desarrollo en el cual van a ser abordados.

6.11 Los sistemas y sus fronteras

Un caso de uso describe la interacción con un “sistema”. Las fronteras ordinarias del sistema son:

- la frontera hardware/software de un dispositivo o sistema de cómputo
- el departamento de una organización
- la organización entera

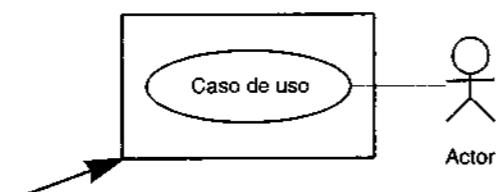


Figura 6.4 Frontera de un caso de uso.

Es importante definir la frontera del sistema para identificar lo que es interno o externo, así como las responsabilidades del sistema. El ambiente externo está representado únicamente por actores.

Estudiaremos un ejemplo de la influencia que tiene seleccionar la frontera del sistema: los pagos en la terminal del punto de venta y la tienda. Si elegimos como “el sistema” la tienda entera o el negocio (figura 6.6), el único actor es el cliente y no el cajero, porque este último es un recurso del sistema del negocio que realiza las funciones. Pero si escogemos como sistema el hardware y el software de la terminal del punto de venta (figura 6.5), se trata como actores al cliente y al cajero.

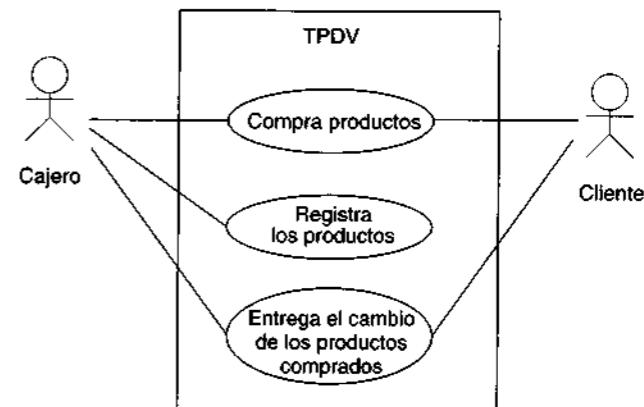


Figura 6.5 Casos de uso y actores cuando el sistema TPDV es la frontera.

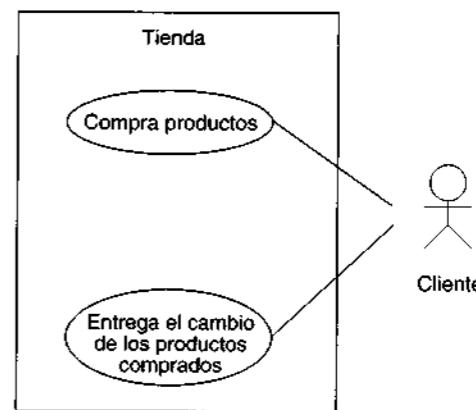


Figura 6.6 Casos de uso y actores cuando tienda es la frontera.

En la selección de una frontera del sistema influyen las necesidades de la investigación. Si estamos desarrollando un software de aplicación o un dispositivo, será razonable establecer la frontera del sistema en la del hardware y en la del software; por ejemplo, la terminal del punto de venta y sus programas constituye “el sistema”, y el cliente y el cajero son los actores (agentes) externos.

Si estamos efectuando la **reingeniería de procesos de la empresa** —reorganizando los procesos o la empresa para mejorar la competitividad o la calidad—, la selec-

ción de la compañía o de la tienda entera como el sistema es importante. En el sistema TPDV, definiremos “el sistema” como la terminal de punto de venta y su software.

6.12 Casos de uso primarios, secundarios y opcionales

Los casos deberían clasificarse en primarios, secundarios u opcionales. Más adelante, a partir de estas designaciones, clasificaremos nuestro conjunto de casos de uso para establecer prioridades en su desarrollo.

Los **casos primarios de uso** representan los procesos comunes más importantes, como *Comprar productos*.

Los **casos secundarios de uso** representan procesos menores o raros; por ejemplo, *Solicitud de surtir el nuevo producto*.

Los **casos opcionales de uso** representan procesos que pueden no abordarse.

6.13 Casos esenciales de uso comparados con los casos reales de uso

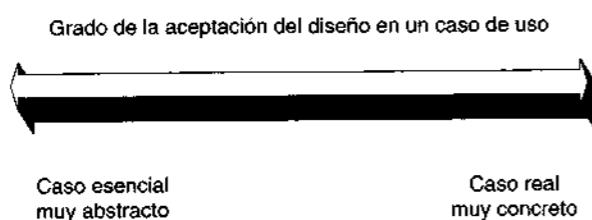


Figura 6.7 Los casos esenciales y reales de uso existen a lo largo de un continuo.

6.13.1 Casos esenciales de uso

Los **casos esenciales de uso** [Constantine97] son casos expandidos que se expresan en una forma teórica que contiene poca tecnología y pocos detalles de implementación; las decisiones de diseño se posponen y se abstraen de la realidad, especialmente las concernientes a la interfaz para el usuario. Un caso de este tipo describe el proceso a partir de sus actividades y motivos esenciales. El grado de abstracción con que se describe existe en un continuo: la descripción puede ser más o menos esencial.

Los casos de alto nivel siempre son de carácter esencial, debido a su brevedad y abstracción.

En seguida se incluye un ejemplo de un caso de *Retiro de efectivo* en un cajero automático, que se expresa en una forma relativamente esencial.

Esencial

Acción de los actores

1. El Cliente se identifica.
3. y así sucesivamente.

Respuesta del sistema

2. Presenta opciones.
4. y así sucesivamente.

La *manera* en que un cliente se identifica cambia con el tiempo —es una decisión de diseño—, pero forma parte del proceso esencial de que la identificación se realice de alguna manera.

Conviene crear casos esenciales de uso al comenzar a investigar los requerimientos, con el propósito de entender mejor el alcance del problema y las funciones necesarias. Este tipo de casos son de gran utilidad porque permiten captar la esencia del proceso y sus motivos fundamentales, sin verse abrumado con detalles de diseño. Suelen también ser correctos durante largo tiempo, ya que excluye las decisiones de diseño y, por lo mismo, su creación favorece la comprensión y el registro de los factores que dan vida a los procesos de un negocio. Una empresa puede recuperar y releer los casos esenciales de uso mucho tiempo después, aplicándolos exitosamente a un nuevo proyecto de desarrollo.

13.2 Casos reales de uso

En cambio, un **caso real de uso** describe concretamente el proceso a partir de su diseño concreto actual, sujeto a las tecnologías específicas de entrada y de salida, etc. Cuando se trata de la interfaz para el usuario, a menudo ofrece presentaciones de pantalla y explica la interacción con los artefactos. A continuación se incluye el caso *Retiro de efectivo* expresado en una forma relativamente *real*.

Real

Acción de los actores

1. El Cliente introduce su tarjeta.
3. Introduce el NIP con un teclado numérico.
5. y así sucesivamente.

Respuesta del sistema

2. Pide el número de identificación personal (NIP).
4. Muestra el menú de opciones.
6. y así sucesivamente.

Nótese que la acción esencial del “Cliente se identifica a sí mismo” del caso de uso se realizó ahora concretamente en la serie de acciones comenzando con “El Cliente introduce su tarjeta”.

En teoría, los casos reales de uso se crean durante la fase de diseño en un ciclo de desarrollo, por ser un artefacto del diseño. En algunos proyectos se prevén las primeras decisiones de diseño concernientes a la interfaz para el usuario; de ahí la necesidad de crear casos reales en la fase inicial de elaboración. Se recomienda hacerlo en la fase de planeación y elaboración por la aceptación prematura de un diseño y la abrumadora complejidad. No obstante, algunas compañías aceptan un contrato de desarrollo, basándose en las especificaciones de la interfaz para el usuario.

En el capítulo 19 se examinan los casos reales en la aplicación al punto de venta.

6.13.3 El caso esencial de uso de la compra de productos

El caso ampliado de uso *Comprar productos* que ya mencionamos tiende a ser un caso esencial. Obsérvese que la descripción no es muy específica respecto a la realización técnica. El caso está escrito de manera que casi podemos imaginar su aplicabilidad después de cien años o hace cien años, lo cual manifiesta que es esencial.

Esencial

Acción de los actores	Respuesta del sistema
1. El Cajero registra el identificador en cada producto. Si hay más de un producto igual, el Cajero puede introducir de igual manera la cantidad. 3. y así sucesivamente.	2. Determina el precio del producto y agrega la información sobre él a la actual transacción de venta. Aparecen la descripción y el precio del producto actual. 4. y así sucesivamente.

6.13.4 El caso real de uso de la compra de productos

A diferencia de una versión esencial del caso de uso, una versión real se compromete con el diseño; una versión completa de ella se explicará en un capítulo posterior. En el siguiente ejemplo de la versión real, nótese la decisión de utilizar un código universal de producto (CUP) con el identificador del producto¹ y una interfaz gráfica para el usuario.

¹ En una fase temprana del análisis no conviene tomar algunas decisiones, como la de utilizar un código universal de producto (CUP) en el caso esencial de uso. El análisis y diseño esencial y real son términos a lo largo de un continuo de abstracción más que extremos. He aquí lo más importante: siempre que se adopta un compromiso durante la fase de análisis, existe la posibilidad de un error prematuro de diseño, sobrecarga de información y menor flexibilidad.

Real

Acción de los actores

1. En cada producto, el Cajero teclea el Código Universal de Productos (CUP) en el campo de entrada del CUP de la Ventana1. Después oprime el botón "Introducir producto" con el ratón u oprimiendo la tecla <Enter>.
3. etcétera.

Respuesta del sistema

2. Muestra el precio del producto y agrega la información sobre él a la actual transacción de venta. La descripción y el precio del producto actual se muestran en el cuadro de Texto 2 de la Ventana1.
4. etcétera.

14 Sobre la notación

14.1 Asignación de nombre a los casos de uso

Al caso de uso se le asigna un nombre que comience con un verbo para subrayar que se trata de un proceso. Por ejemplo:

- Comprar productos
- Introducir un pedido

14.2 Inicio de un caso expandido de uso

Comience un caso expandido con el siguiente esquema:

1. Este caso comienza cuando <Actor> <inicia un evento>

Por ejemplo:

1. Este caso de uso comienza cuando un Cliente llega a un TPDV con productos que desea comprar.

De este modo se estimula una identificación clara del actor y del evento iniciadores.

14.3 Puntos sobre la decisión de notación y sobre la ramificación

Un caso de uso puede contener puntos de decisión. Por ejemplo, en *Comprar productos*, el cliente puede optar por pagar con efectivo, a crédito o con cheque en el momento del pago. Si una de estas trayectorias de decisión es un caso muy representativo y si las otras alternativas son raras, inusuales o excepcionales, el caso típico

deberá ser el único acerca del cual se escribe en el *Curso normal de los eventos* y las opciones han de escribirse en la sección titulada *Alternativas*.

Pero en ocasiones el punto de decisión representa opciones cuya probabilidad es relativamente igual y normal; esto sucede con los tipos de pago en efectivo, con tarjeta de crédito y con cheque. En este caso se utiliza la siguiente estructura notacional:

1. En la sección principal *Curso normal de los eventos*, indique las ramas de las subsecciones.
2. Escriba una subsección en cada rama, utilizando otra vez un *Curso normal de los eventos*. Inicie el evento numerando en 1 cada sección.
3. Si las subsecciones tienen opciones, escríbalas en una sección de alternativas de cada subsección.

Sección: principal

Curso normal de los eventos

Acción de los actores

1. Este caso de uso comienza cuando un Cliente llega a la caja TPDV con los productos que comprará.
2. (Excluidos los pasos intermedios)...
3. El Cliente escoge el tipo de pago:
 - a. Si paga en efectivo, consúltese la sección *Pago en efectivo*.
 - b. Si paga a crédito, consúltese la sección *Pago con tarjeta de crédito*.
 - c. Si paga con cheque, consúltese la sección *Pago con cheque*.
6. El Cajero le entrega el recibo al Cliente.
7. El Cliente se marcha con los productos comprados.

Respuesta del sistema

4. Registra la venta terminada.
5. Imprime un recibo.

Sección: Pago en efectivo

Curso normal de los eventos

Acción de los actores

1. El Cliente da un pago en efectivo —el “efectivo ofrecido”—, posiblemente mayor que el total de la venta.
2. El Cajero registra el efectivo ofrecido.
4. El Cajero deposita el efectivo recibido y extrae la diferencia.

El Cajero entrega al Cliente el cambio del pago.

Respuesta del sistema

3. Muestra al Cliente la diferencia.

Cursos alternativos

- Línea 4: efectivo insuficiente en la caja para pagar la diferencia. Se pide efectivo al supervisor o se pide al Cliente un pago más cercano al total de la venta.

Sección: pago con tarjeta de crédito

Cursos normales y alternos de la historia de pago con tarjeta de crédito.

Sección: pago con cheque

Cursos normales y alternos de la historia de pago con cheque.

6.15 Casos de uso dentro de un proceso de desarrollo

6.15.1 Pasos de la fase de planeación y elaboración

1. Despues de haber listado las funciones del sistema, defina la frontera de éste y luego identifique los actores y los casos de uso.

2. Escriba todos los casos de uso en el formato de *alto nivel*. Clasifíquelos en primarios, secundarios u opcionales.
3. Dibuje un diagrama de caso de uso.
4. Relacione los casos de uso y dé ejemplo de las relaciones en el diagrama correspondiente (más adelante se explican las relaciones de los casos).
5. Escriba en el formato *esencial expandido* los casos de uso más importantes, influyentes y riesgosos, a fin de entender y estimar mejor la naturaleza y las dimensiones del problema. Para evitar análisis complejos posponga la escritura de la forma esencial expandida de los casos de uso menos importantes hasta los ciclos de desarrollo en que serán abordados.
6. En teoría, los casos *reales* deberían posponerse hasta una fase de diseño en el ciclo de desarrollo, porque su creación conlleva decisiones de diseño. Pese a ello, a veces es necesario crear casos reales de uso durante la etapa inicial de los requerimientos si:
 - Las descripciones concretas facilitan notablemente la comprensión.
 - Los Clientes exigen especificar sus procesos en esta forma.
7. Clasifique los casos de uso (que se expondrán en el siguiente capítulo).

6.15.2 Pasos de la fase del ciclo de desarrollo iterativo

1. Fase de análisis: escriba casos esenciales de uso expandidos para los que se han abordado, si todavía no se llevan a cabo.
2. Fase de diseño: escriba casos reales de uso para los que están siendo abordados, en caso de que todavía no se realicen.

6.16 Pasos del proceso en un sistema del punto de venta

Expicaremos algunas de las siguientes actividades en capítulos posteriores, ya que requieren una exposición amplia o pueden aplazarse para evitar una sobrecarga de información. Como nuestra meta es adquirir la habilidad de aplicar los casos y no convertirnos en expertos en tiendas, no escribiremos los detalles de todos los casos.

6.16.1 Identifique los actores y los casos de uso

En la aplicación del punto de venta, defina la frontera del sistema que será el sistema de hardware/software, el caso habitual. Un ejemplo de lista de los actores y procesos relevantes a que dan inicio —que no pretende en absoluto ser completa— incluye:

Cajero	Registra los productos Entrega el cambio
Cliente	Compra productos Paga los productos
Gerente	Inicia Cierra
Administrador del sistema	Incorpora nuevos usuarios

6.16.2 Escriba casos de uso en el formato de alto nivel

Una muestra de casos de uso de alto nivel comprende:

Caso de uso:	Comprar productos
Actores:	Cliente (iniciador), Cajero.
Tipo:	Primario.
Descripción:	Un Cliente llega a una caja con productos que desea comprar. El Cajero registra los productos y obtiene el pago. Al terminar la transacción, el Cliente se marcha con los productos.
Caso de uso:	Inicio de operaciones
Actores:	Gerente.
Tipo:	Primario.
Descripción:	Un gerente activa una TPDV a fin de preparla para que la usen los Cajeros. El Gerente comprueba que la fecha y la hora sean correctos; hecho esto, el sistema está listo para que lo utilice el Cajero.

6.16.3 Dibuje un diagrama de casos de uso

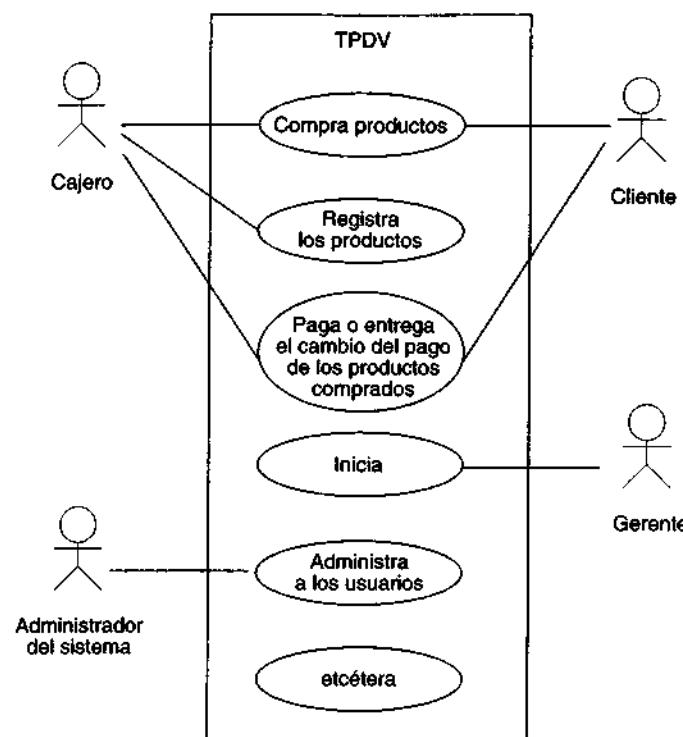


Figura 6.8 Diagrama parcial de un caso de uso, que representa la aplicación TPDV.

6.16.4 Relacione los casos de uso

Este tema lo trataremos en un capítulo posterior.

6.16.5 Escriba algunos casos esenciales expandidos de uso

Entre los casos primarios de uso realmente significativos figuran:

- Comprar productos
- Pagar los productos comprados

Escribir lo anterior en una forma esencial expandida suministrará una mayor información y esclarecimiento de los requerimientos. A continuación se presenta el caso de uso *Comprar productos* en su forma esencial expandida completa:

Casos de uso: comprar productos

Sección: principal

Caso de uso:	Comprar productos
Actores:	Cliente (iniciador), Cajero.
Propósito:	Capturar una venta y su pago.
Resumen:	Un Cliente llega a la caja con productos que desea comprar. El Cajero registra los productos y recibe el pago, que puede ser autorizado. Al terminar la transacción, el Cliente se marcha con los productos.
Tipo:	Primario y esencial.
Referencias cruzadas:	<i>Funciones:</i> R1.1, R1.2, R1.3, R1.7, R1.9, R2.1, R2.2, R2.3, R2.4.
	<i>Casos de uso:</i> el Cajero debe haber terminado el caso de uso: <i>Registrar</i> .

Curso normal de los eventos

Acción de los actores

1. Este caso de uso comienza cuando un Cliente llega a la caja TPDV con productos que desea comprar.
2. El Cajero registra los productos. Si hay más de un producto, también puede introducir la cantidad.
3. Determina el precio del producto y agrega la información sobre él a la actual transacción de venta. Se muestran la descripción y el precio del producto actual.
4. Al terminar la captura de los productos, el Cajero indica a TPDV que terminó la captura de los productos.
5. Calcula y presenta el total de la venta.
6. El Cajero le indica el total al Cliente.

Respuesta del sistema

Curso normal de los eventos**Acción de los actores****Respuesta del sistema**

7. El Cliente escoge la forma de pago:
 - a. Si paga en efectivo, véase la sección *Pagar en efectivo*.
 - b. Si paga con tarjeta de crédito, véase la sección *Pagar con tarjeta de crédito*.
 - c. Si paga con cheque, véase la sección *Pagar con cheque*.
8. Registra la venta terminada.
9. Actualiza los niveles de inventario.
10. Genera un recibo.
11. El Cajero entrega el recibo al cliente.
12. El Cliente se marcha con los productos comprados.

Cursos alternos

- Línea 2: se introduce un identificador inválido del producto. Indique el error.
- Línea 7: el Cliente no pudo pagar. Cancelé la transacción de venta.

Sección: pagar en efectivo**Curso normal de los eventos****Acción de los actores****Respuesta del sistema**

1. El Cliente da un pago en efectivo —el “efectivo ofrecido”—, posiblemente mayor que el total de la venta.

Curso normal de los eventos**Acción de los actores****Respuesta del sistema**

2. El Cajero registra el efectivo ofrecido.
3. Presenta la diferencia al Cliente.
4. El Cajero deposita el efectivo recibido y extrae la diferencia.
- El Cajero le entrega el cambio al Cliente.

Cursos alternos

- Línea 1: el Cliente no tiene suficiente efectivo. Puede cancelar o iniciar otro método de pago.
- Línea 4: la caja no contiene suficiente efectivo para pagar la diferencia. El Cajero pide más efectivo al supervisor o le pide al Cliente otro billete de menor denominación u otra forma de pago.

Sección: pago con tarjeta de crédito**Curso normal de los eventos****Acción de los actores****Respuesta del sistema**

1. El Cliente comunica su información de crédito para pagar con tarjeta.
2. Genera una solicitud de pago con tarjeta de crédito y la envía a un Servicio externo de autorización de crédito.
3. El Servicio de autorización de crédito autoriza el pago.
4. Recibe una respuesta aprobatoria de crédito del Servicio de autorización de crédito.
5. En el sistema de Cuentas por cobrar registra la información sobre el pago con tarjeta de crédito y la respuesta de aprobación. El Servicio de autorización de crédito debe dineros a la Tienda; por tanto, Cuentas por cobrar debe darle seguimiento.
6. Muestra el mensaje aprobatorio de autorización.

Cursos alternos

- Línea 3: solicitud de crédito negada por el Servicio de autorización de crédito. Proponer otro método de pago.

Sección: pago con cheque

Curso normal de los eventos

Acción de los actores	Respuesta del sistema
1. El Cliente extiende un cheque y se identifica.	
2. El Cajero registra la información sobre la identificación y solicita la autorización del pago con cheque.	3. Genera una solicitud de pago con cheque y la envía a un Servicio externo de autorización de cheques.
4. Verifica que el pago haya sido autorizado por el Servicio de autorización de cheques.	5. Recibe una respuesta aprobatoria del Servicio de autorización de cheques. 6. Indica la obtención de la autorización.

Cursos alternos

- Línea 4: verificar solicitud negada por el Servicio de autorización de cheques. Proponer otra forma de pago.

6.16.6 Si es necesario, escriba algunos casos reales de uso

No conviene o no es necesario crear casos de uso real en este momento; este trabajo se realizará durante los ciclos de desarrollo.

6.16.7 Clasifique los casos de uso

Este tema se estudiará en el siguiente capítulo.

6.17 Modelos muestra

Los casos esenciales de alto nivel y los diagramas de casos de uso son miembros del modelo de casos de uso del análisis (figura 6.9).

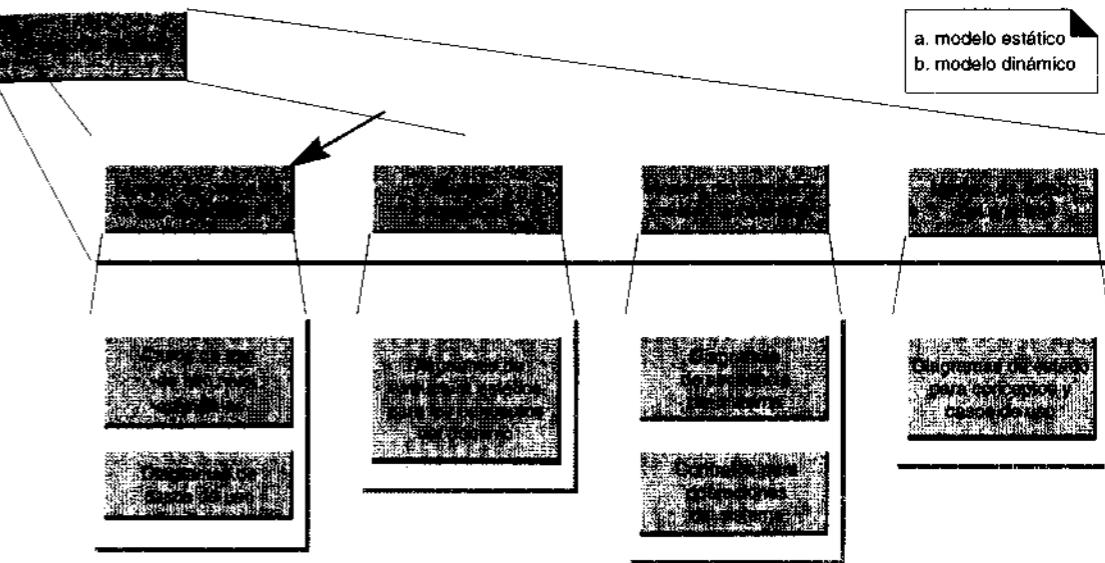


Figura 6.9 Modelo de análisis.

UNIVERSIDAD DE LA REPÚBLICA
FACULTAD DE INGENIERÍA
DEPARTAMENTO DE
DOCUMENTACIÓN Y BIBLIOTECA
MONTEVIDEO - URUGUAY

CLASIFICACIÓN Y PROGRAMACIÓN DE LOS CASOS DE USO

Objetivos

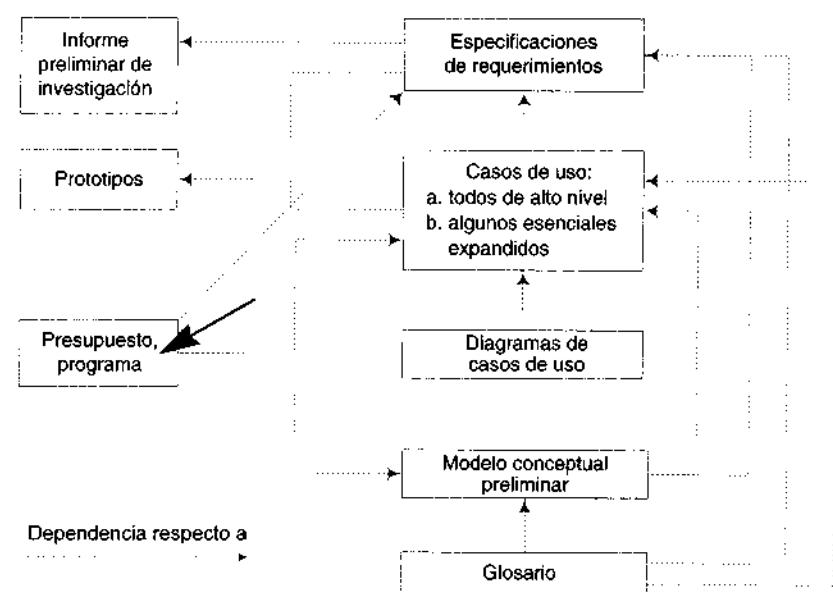
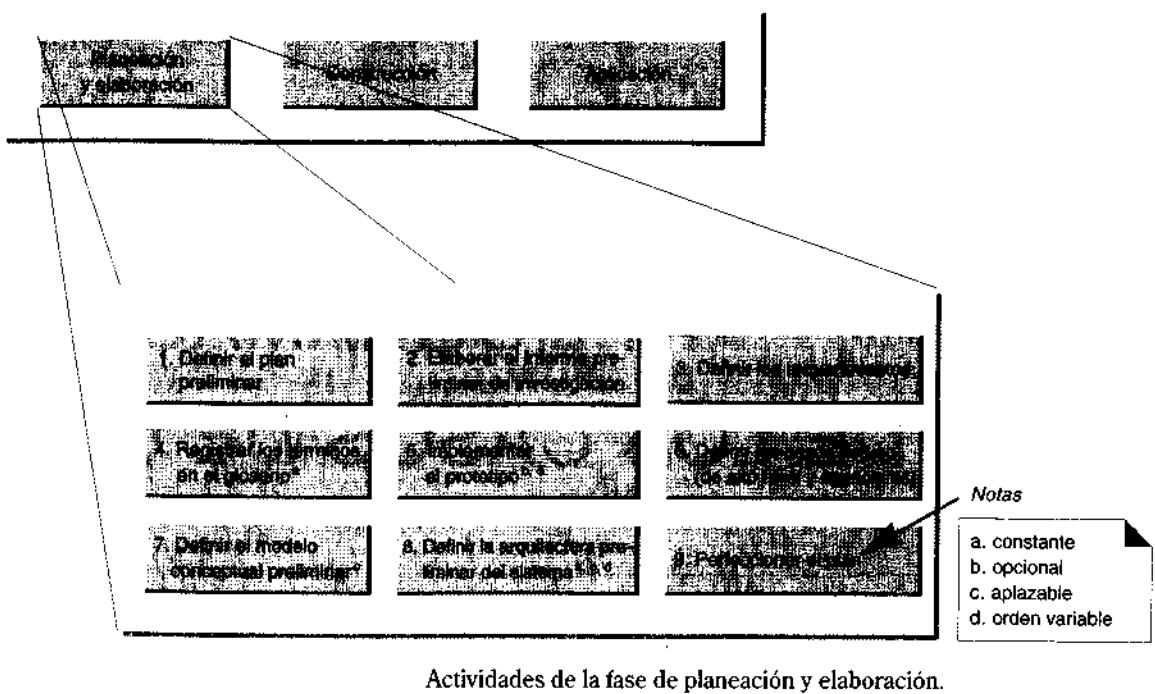
- Clasificar los casos de uso.
- Cuando sea necesario, preparar versiones simplificadas de los casos de uso.
- Asignar los casos de uso a los ciclos de desarrollo.

7.1 Introducción

Las especificaciones de los requerimientos y los casos de uso se definen en la Fase de Planeación y Elaboración. Además, puede crearse un Modelo conceptual preliminar y diseñar una arquitectura también preliminar del sistema, aunque estas actividades se pospondrán en nuestro estudio de casos para ofrecer una introducción más gradual a los temas.

Suponiendo que todos los artefactos deseados hayan sido generados (por ejemplo, la especificación de los requerimientos y los casos de uso), el siguiente paso es iniciar la fase de Construcción en el ciclo de desarrollo iterativo y comenzar a implementar el sistema. En un ciclo de vida de desarrollo iterativo, la tarea de llenar los casos de uso se distribuye entre varios ciclos.

En el presente capítulo se estudia la clasificación y la programación o calendarización de los casos de uso. Una vez concluida esta etapa, estaremos listos para comenzar el primer ciclo de desarrollo y examinar a fondo el análisis y diseño orientados a objetos.



Dependencias de los artefactos respecto a la fase de planeación y elaboración.

7.2 Programación de los casos de uso en los ciclos de desarrollo

7.2.1 Casos de uso y los ciclos de desarrollo

Los ciclos de desarrollo se organizan en torno a los requerimientos de los casos de uso. En otras palabras, se asigna un ciclo para implementar uno o más casos de uso o sus versiones simplificadas, cuando el caso íntegro resulta demasiado complejo para abordarlo en un ciclo (figura 7.1).

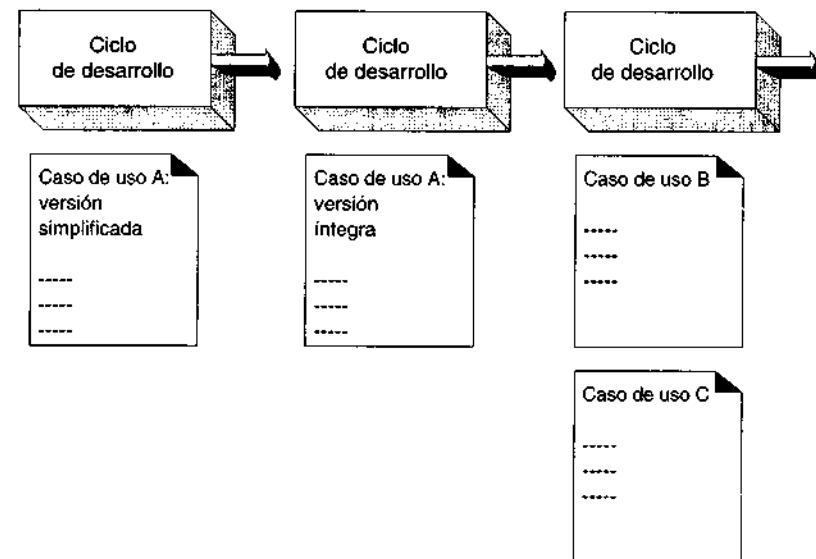


Figura 7.1 Asignación de los casos de uso a los ciclos de desarrollo.

7.2.2 Clasificación de los casos de uso

Es necesario clasificar los casos de uso, y los casos de alto rango han de tratarse al inicio de los ciclos de desarrollo. La estrategia general consiste en escoger primero los casos que influyen profundamente en la arquitectura básica. He aquí algunas de las cualidades que aumenta la clasificación de un caso:

- a. Tener una fuerte repercusión en el diseño arquitectónico; por ejemplo, incorporar muchas clases a la capa del dominio o requerir servicios de persistencia.
- b. Con relativamente poco esfuerzo obtener información e ideas importantes sobre el diseño.
- c. Incluir funciones riesgosas, urgentes o complejas.
- d. Requerir una investigación a fondo o tecnología nueva y riesgosa.
- e. Representar procesos primarios de la línea de negocios.
- f. Apoyar directamente el aumento de ingresos o la reducción de costos.

El esquema taxonómico puede servirse de una clasificación simple y poco rigurosa: alto-mediano-bajo.

El esquema también puede aplicar puntuaciones (posiblemente incrementadas con ponderación), basándose en las cualidades que inciden en la clasificación; por ejemplo:¹

Caso de uso	a	b	c	d	e	f	Suma
Comprar productos	5	3	2	0	5	3	18
y así sucesivamente							

7.3 Clasificación de los casos de uso en la aplicación al punto de venta

Con base en los criterios anteriores de clasificación, a continuación se incluye una clasificación informal y poco rigurosa de los casos de uso de aplicación al punto de venta. De ninguna manera pretendemos dar una lista exhaustiva.

Clasificación	Caso de uso	Justificación
Alto	Comprar productos	Corresponde a los criterios de clasificación más altos.
Mediano	Incorpora nuevos usuarios	Afecta al subdominio de la seguridad.
	Registra los productos comprados	Afecta al subdominio de la seguridad.
	Paga los productos comprados	Proceso importante; afecta a la contabilidad.
Bajo	Pagar	Efecto mínimo en la arquitectura.
	Iniciar	La definición depende de otros casos de uso.
	Cerrar	Efecto mínimo en la arquitectura.

7.4 El caso de uso de arranque

Prácticamente todos los sistemas cuentan con un *Caso de uso de inicio o arranque*. Aunque tal vez no ocupe un nivel alto conforme a otros criterios, es preciso estudiar al menos una versión simplificada de él, al principiar el ciclo de desarrollo para presentar la inicialización supuesta en otros casos. En cada ciclo de desarrollo, éste se

¹ El esquema no es completo, pues tan sólo pretende ofrecer sugerencias.

realizó incrementalmente para satisfacer las necesidades de arranque de otros casos. No vamos a presentar de manera explícita la programación de los pasos de este caso de uso y supondremos que siempre se desarrolla en forma implícita como se necesita.

7.5 Programación de los casos de uso en la aplicación del punto de venta

A partir de la clasificación, el caso *Comprar productos* debería incluirse en el primer ciclo de desarrollo. Como hemos visto, también puede abordarse una versión simple de *Inicio* para soportar los otros casos de uso.

7.5.1 Creación de versiones múltiples de los casos de uso complejos

Siempre que se asigne un caso de uso, es necesario estimar si es posible resolverlo íntegramente en el lapso limitado de un ciclo (cuatro semanas, por ejemplo) o si el trabajo ha de ser distribuido en varios ciclos. En este caso, *Comprar productos* es extremadamente complejo y quizás requiere cinco o más ciclos, suponiendo que cada uno tendrá una duración de cuatro semanas exactamente. Se supone una estrategia de programación de duración o tiempo fijo, en la cual al ciclo de desarrollo se le establece un plazo fijo.

En esta situación el caso se redefine a partir de varias versiones de él, que van abarcando requerimientos cada vez más exhaustivos. Cada versión se limita a incluir lo que se estima una cantidad razonable de trabajo dentro de los confines de la duración fija del ciclo (digamos cuatro semanas). Por ejemplo:

- Comprar productos: versión 1 (pagos en efectivo, sin actualizaciones de inventario, ...)
- Comprar productos: versión 2 (permitir cualquier tipo de pago)
- Comprar productos: versión 3 (versión completa, incluyendo entre otras cosas las actualizaciones del inventario, ...)

Las versiones anteriores se distribuyen después a lo largo de una serie de ciclos de desarrollo, junto con otros casos de uso.

7.5.2 Asignación de los casos de uso

Si nos basamos en la clasificación de los casos y de varias versiones de *Comprar productos*, podríamos asignar algunos casos de uso al ciclo de desarrollo de la figura 7.2.

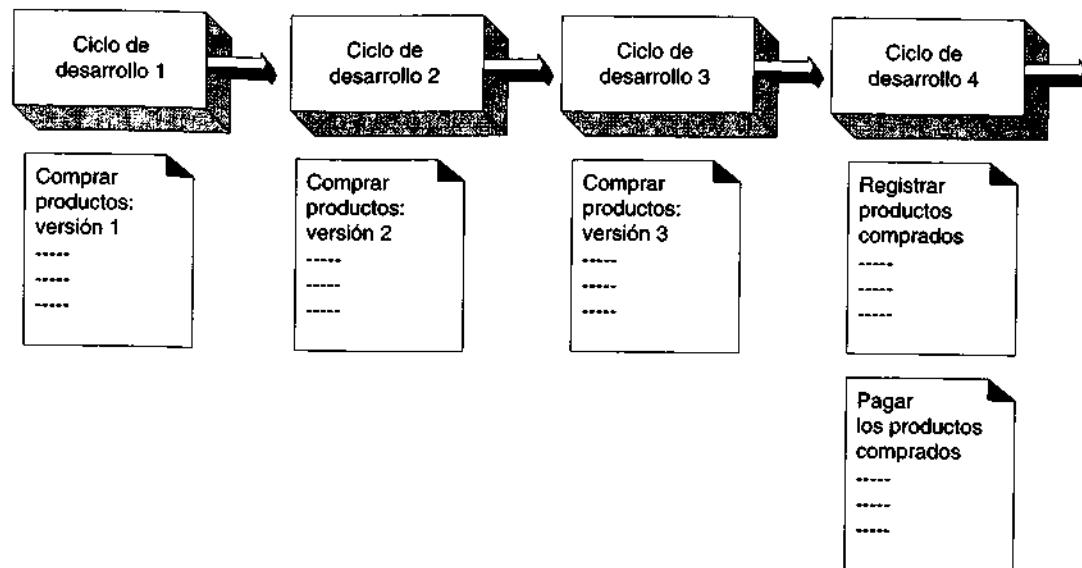


Figura 7.2 Asignación de los casos de uso a los ciclos de desarrollo.

7.6 Versiones del caso de uso “Comprar productos”

Una vez que se ha decidido simplificar el caso y expresarlo, hay que escribir versiones cada vez más complejas. También hay que estipular las simplificaciones, las metas y las suposiciones de cada versión. Las siguientes secciones ofrecen sugerencias valiosas al respecto.

7.6.1 Versión 1 de Comprar productos

Simplificaciones, metas y suposiciones

- Pagos en efectivo exclusivamente.
- Sin mantenimiento de inventario.
- Es una tienda independiente, que no forma parte de una organización más grande.
- Captura manual del código universal de producto (CUP); sin lector de código de barras.
- No se calculan los impuestos.
- Sin cupones.
- El cajero no tiene que registrar las ventas; no se controla el acceso.
- No se lleva un registro de los clientes individuales ni de sus hábitos de compra.

- No se controla la caja de efectivo.
- En el recibo aparecen el nombre y la dirección de la tienda, la fecha y la hora de la venta.
- Ni la identificación del cajero ni la de TPDV aparecen en el recibo.
- Las ventas realizadas se registran en un documento histórico.

Caso de uso:	Comprar productos, versión 1
Actores:	Cliente (iniciador), Cajero.
Propósito:	Capturar una venta y su pago en efectivo.
Resumen:	Un Cliente llega a la caja con productos que desea comprar. El Cajero registra los productos comprados y recibe el pago en efectivo. Al terminar la transacción, el Cliente se marcha con los productos.
Tipo:	Primario y esencial.
Referencias cruzadas:	Funciones: R1.1, R1.2, R1.3, R1.5, R1.7, R1.9, R2.1.

Curso normal de los eventos

- | Acción de los actores | Respuesta del sistema |
|--|---|
| <ol style="list-style-type: none"> 1. Este caso de uso comienza cuando un Cliente llega a un caja de TPDV con productos que desea comprar. 2. El Cajero registra el código universal de producto (CUP) en cada producto.
Si un producto se repite, el Cajero también puede introducir la cantidad. 4. Al terminar de capturar los productos, el Cajero indica a la TPDV que ya concluyó la captura. 6. El Cajero le indica al Cliente el total. 7. El Cliente da un pago en efectivo —el efectivo “ofrecido”—, posiblemente mayor que el total de la venta. | <ol style="list-style-type: none"> 3. Determina el precio del producto y agrega la información correspondiente a la transacción actual.

Presenta la descripción y el precio del producto en cuestión. 5. Calcula el total de la venta y se lo presenta al Cliente. |

Curso normal de los eventos

Acción de los actores	Respuesta del sistema
8. El Cajero registra el efectivo recibido.	9. Muestra al Cliente la diferencia. Genera un recibo.
10. El Cajero deposita el efectivo recibido y extrae la diferencia. El Cajero entrega al cliente el cambio y el recibo impreso.	11. Registra la venta terminada.
12. El Cliente se marcha con los productos comprados.	

7.6.2 Comprar productos: versión 2

Simplificaciones, metas y suposiciones

Las simplificaciones de la versión 1 se aplican también en esta versión, salvo que el pago puede efectuarse en efectivo, con tarjeta de crédito o con cheque. Las dos segundas formas de pago requieren autorización.

Caso de uso: Comprar productos, versión 2

Actores: Cliente (iniciador), Cajero.

Propósito: Capturar una venta y su pago.

Resumen: Un Cliente llega a la caja con productos que desea comprar. El Cajero registra los productos comprados y recibe un pago, que debe recibir autorización. Al terminar la transacción, el Cliente se marcha con los productos comprados, y así sucesivamente.

7.7 Resumen

Ya estamos preparados para hacer la transición a la fase del desarrollo iterativo. En el primer ciclo nos serviremos del caso de uso *Comprar productos: versión 1*.

INICIO DE UN CICLO DE DESARROLLO

Objetivos

- Resumir la transición de la fase de planeación y elaboración a la de construcción iterativa.

8.1 Inicio de un ciclo de desarrollo

Suponga que la fase de planeación y elaboración ha concluido y que los casos de uso han sido identificados, clasificados y programados, por lo menos en la primera pareja de ciclos. Se presenta entonces una transición muy importante: comienza la fase de construcción en la cual se cumplen los ciclos del desarrollo iterativo (figura 8.1). En el primer ciclo se ha decidido examinar una versión simplificada de *Comprar productos* que incluye solamente los pagos en efectivo y no el control de inventario.

Las actividades iniciales del ciclo se relacionan con la administración del proyecto. En el caso general, viene después (o, más probablemente, ocurre en paralelo) una sincronización de la documentación (por ejemplo, los diagramas) a partir del último ciclo con el estado real del código, porque los artefactos de diseño y los códigos difieren invariablemente durante la fase de codificación del último ciclo.

Entonces se inicia la fase de analizar (o de análisis), en la cual se investigan a fondo los problemas del ciclo actual. En esta fase, una de las primeras actividades consiste en desarrollar un modelo conceptual, tema que trataremos en el siguiente capítulo.

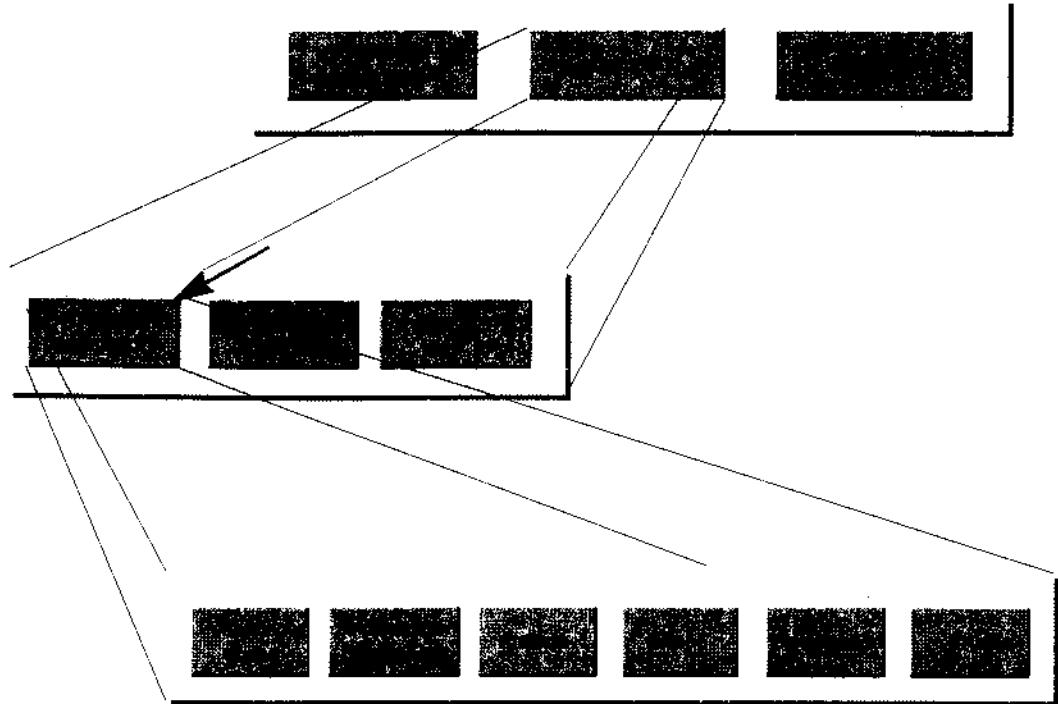


Figura 8.1 Un ciclo de desarrollo.

PARTE III **FASE DE
ANÁLISIS (1)**

UNIVERSIDAD DE LA REPÚBLICA
FACULTAD DE TECNOLOGÍA
DEPARTAMENTO DE
DOCUMENTACIÓN Y BIBLIOTECA
MONTEVIDEO - URUGUAY

CONSTRUCCIÓN DE UN MODELO CONCEPTUAL

Objetivos

- Identificar los conceptos relacionados con los requerimientos del ciclo actual de desarrollo.
- Crear un modelo conceptual preliminar.
- Distinguir entre los atributos correctos y los incorrectos.
- Incorporar conceptos de especificación cuando convenga.
- Comparar los términos: concepto, tipo, interfaz y clase.

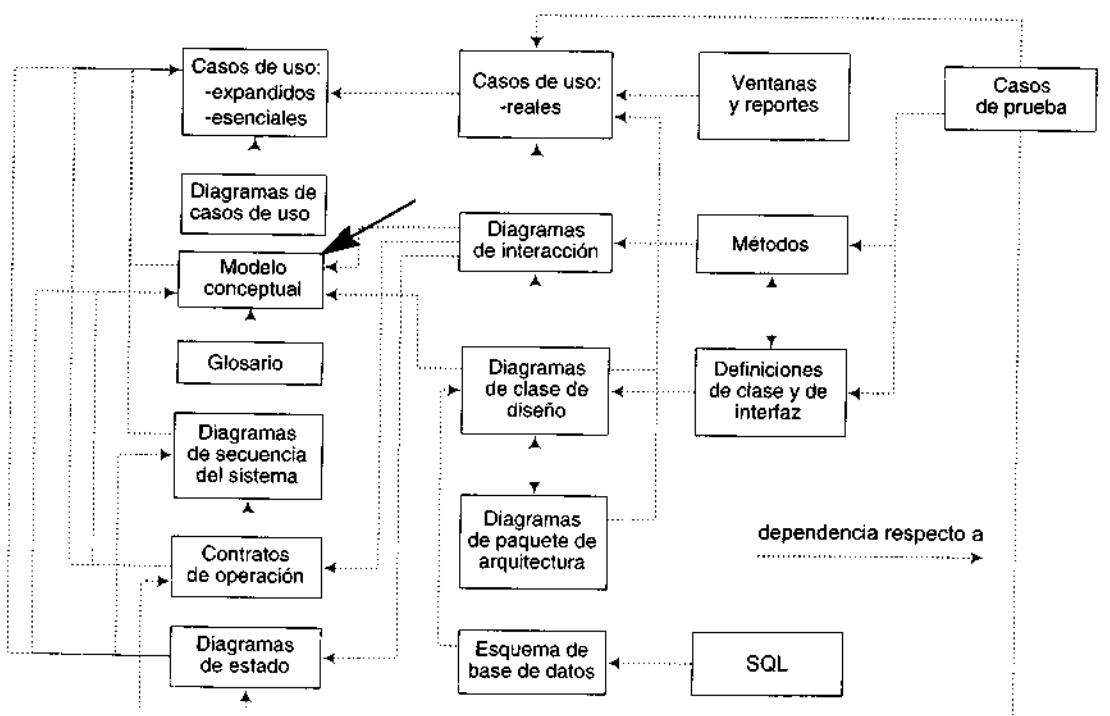
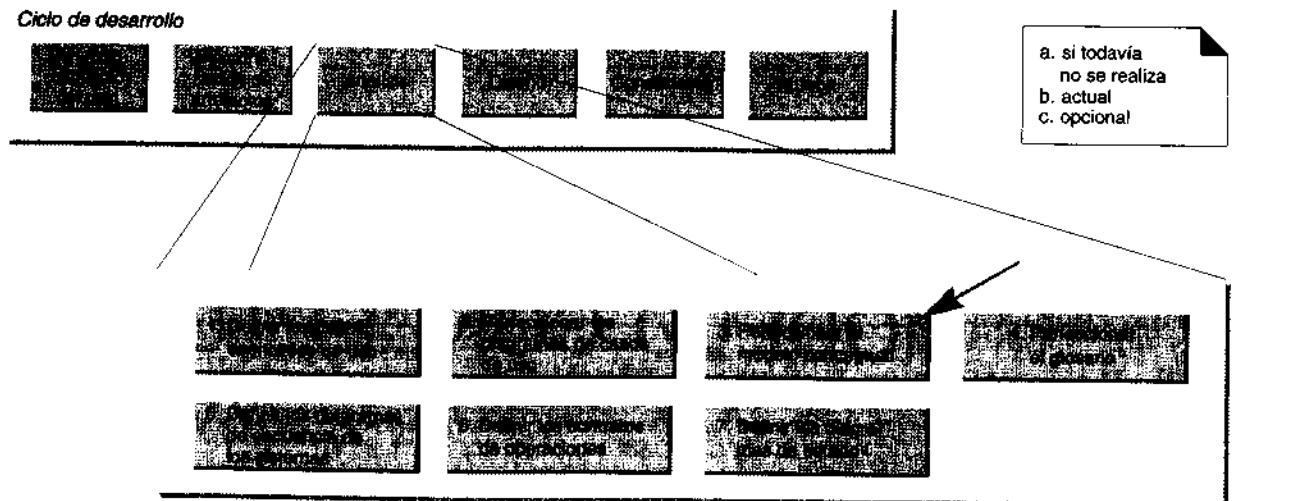
UNIVERSIDAD DE LA REPÚBLICA
FACULTAD DE INGENIERÍA

DIRECCIÓN NACIONAL DE
DOCUMENTACIÓN Y BIBLIOTECA
MONTEVIDEO - URUGUAY

9.1 Introducción

Un modelo conceptual explica (a sus creadores) los conceptos significativos en un dominio del problema; es el artefacto más importante a crear durante el análisis orientado a objetos.¹ En este capítulo estudiaremos los conocimientos preliminares en la creación de modelos conceptuales. En los dos siguientes examinaremos más detenidamente las habilidades relacionadas con la construcción de modelos conceptuales: observar atentamente los atributos y las asociaciones.

¹ Los casos de uso son un importante artefacto del análisis de requerimientos, pero realmente no están orientados a *objetos*. Ponen de relieve la vista del dominio a partir de un proceso.



Dependencias de los artefactos durante la fase de construcción.

Identificar muchos objetos o conceptos constituye la esencia del análisis orientado a objetos, y el esfuerzo se compensa con los resultados conseguidos durante la fase de diseño e implementación.

La identificación de conceptos forma parte de una investigación del dominio del problema. El lenguaje UML contiene la notación en diagramas de estructura estática que explican gráficamente los modelos conceptuales.

Una calidad esencial que debe ofrecer un modelo conceptual es que representa cosas del mundo real, no componentes del software.

9.2 Actividades y dependencias

Una de las primeras actividades centrales de un ciclo de desarrollo consiste en crear un modelo conceptual para los casos de uso del ciclo actual. Esto no puede hacerse si no se cuentan con los casos y con otros documentos que permitan identificar los conceptos (objetos). La creación no siempre es lineal; por ejemplo, el modelo conceptual puede formularse en paralelo con el desarrollo de los casos.

9.3 Modelos conceptuales

El paso esencial de un análisis o investigación orientados a *objetos* es descomponer el problema en conceptos u objetos individuales: las cosas que sabemos. Un **modelo conceptual** es una representación de conceptos en un dominio del problema [MO95, Fowler96]. En el UML, lo ilustramos con un grupo de **diagramas de estructura** donde no se define ninguna operación. La designación de *modelo conceptual* ofrece la ventaja de subrayar fuertemente una concentración en los conceptos del dominio, no en las entidades del software.

Puede mostrarnos:

- conceptos
- asociaciones entre conceptos
- atributos de conceptos

Por ejemplo, en la figura 9.1 vemos un modelo conceptual parcial del dominio de la tienda y las ventas. Explica gráficamente que el concepto de *Pago* y *Venta* son importantes en este dominio del problema, que *Pago* se relaciona con *Venta* en una forma que conviene señalar y que *Venta* tiene fecha y hora. Por ahora no son importantes para nosotros los detalles de la notación.

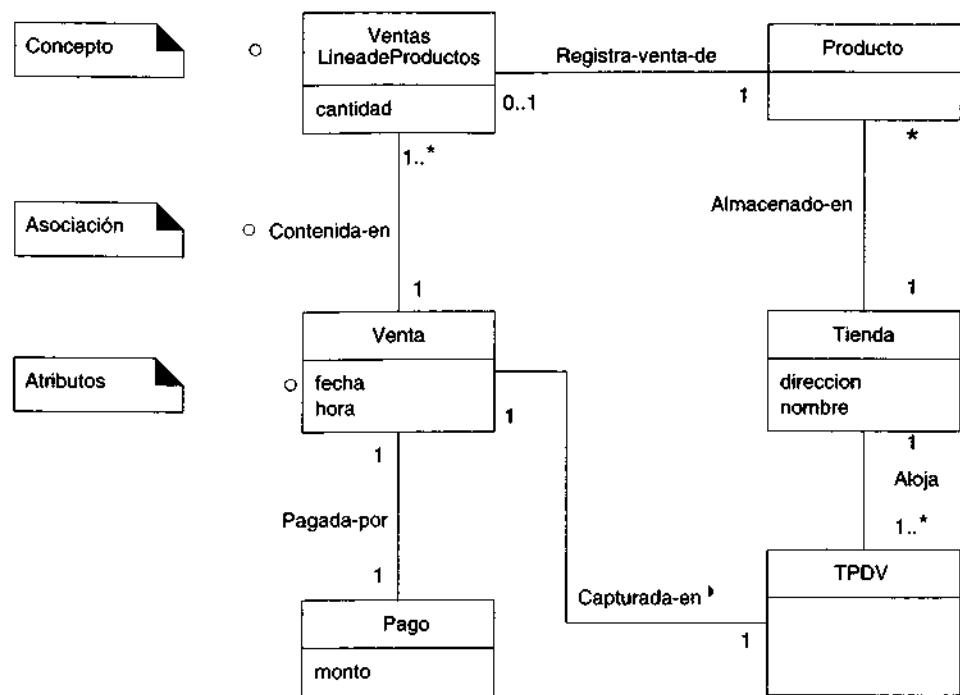


Figura 9.1 Modelo conceptual parcial. Los números en los extremos de la línea indican multiplicidad, la cual se describe en un capítulo subsecuente.

9.3.1 Conocimiento de la nomenclatura del dominio

Además de descomponer el espacio del problema en unidades comprensibles (conceptos), la creación de un modelo conceptual contribuye a esclarecer la terminología o nomenclatura del dominio. Podemos verlo como un modelo que comunica (a los interesados como pueden serlo los desarrolladores) cuáles son los términos importantes y cómo se relacionan entre sí.

9.3.2 Los modelos conceptuales no son modelos de diseño de software

Un modelo conceptual, como se advierte en la figura 9.2, es una descripción del dominio de un problema real, *no* es una descripción del diseño del software, como una clase de Java o de C++ (figura 9.3). Por ello los siguientes elementos no son adecuados en él:

- Los artefactos de software, como una ventana o una base de datos, salvo que el dominio a modelar se refiera a conceptos de software; por ejemplo, un modelo de interfaces gráficas para el usuario.

■ Las responsabilidades o métodos.¹



Figura 9.2 Un modelo conceptual muestra conceptos del mundo real.

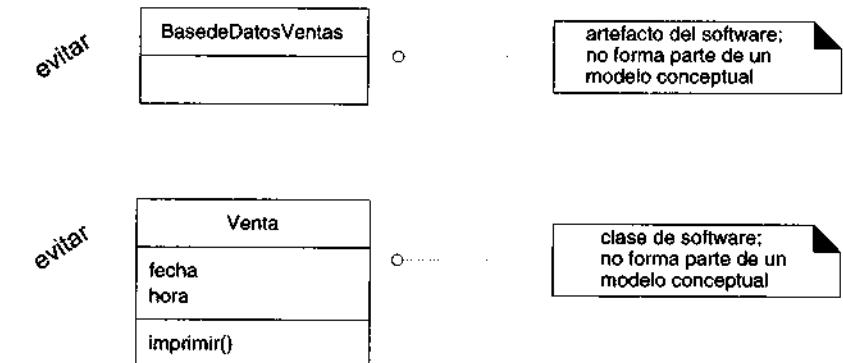


Figura 9.3 Un modelo conceptual no muestra los artefactos o clases del software.

9.3.3 Conceptos

En términos informales el concepto es una idea, cosa u objeto. En un lenguaje más formal, podemos considerarlo a partir de su símbolo, intención² y extensión [MO95] (figura 9.4).

- Símbolo:** palabras o imágenes que representan un concepto.
- Intensión:** la definición del concepto.
- Extensión:** el conjunto de ejemplos a que se aplica el concepto.

Consideremos, por ejemplo, el concepto del evento de una transacción de compra. Podemos optar por designarlo con el símbolo *Venta*. La intensión de una *Venta* puede afirmar que “representa el evento de una transacción de compra y tiene fecha y hora”. La extensión de *Venta* son todos los ejemplos de venta; en otras palabras, el conjunto de todas las ventas.

¹ Las responsabilidades normalmente se relacionan con entidades del software y los métodos siempre lo hacen; pero el modelo conceptual describe conceptos reales, no entidades del software. Durante la fase de *diseño* es muy importante tener en cuenta las responsabilidades; ya que no sólo forman parte de este modelo.

² Intensión: en oposición a extensión, designa el grado de una cualidad.

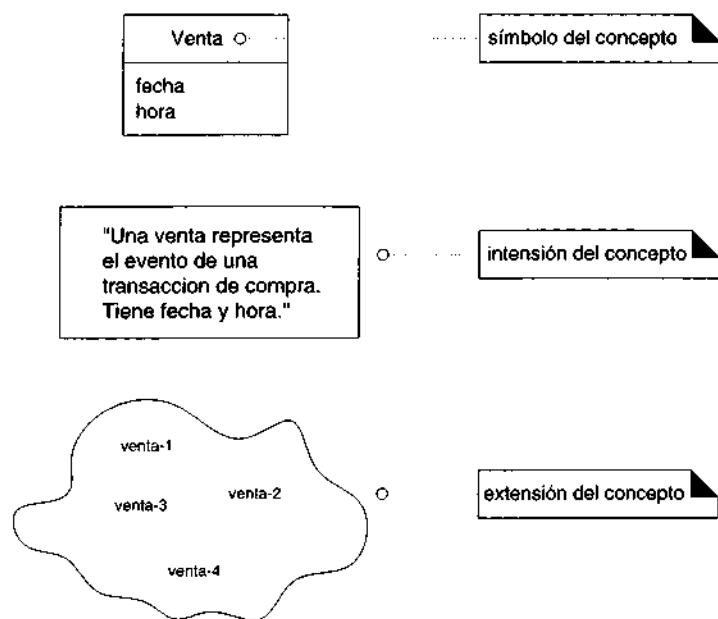


Figura 9.4 El concepto tiene un símbolo, intención y extensión.

Cuando se crea un modelo conceptual, por lo regular la vista del símbolo y de la intención de un concepto es el aspecto de mayor interés práctico.

9.3.4 Los modelos conceptuales y la descomposición

Los problemas de software a veces son complejos; la descomposición —divide y vencerás— es una estrategia que suele utilizarse para resolver la complejidad dividiendo el espacio del problema en unidades comprensibles. En el **análisis estructurado** la dimensión de la descomposición se realiza mediante procesos o *funciones*. En cambio, en el análisis orientado a objetos, se lleva a cabo fundamentalmente con conceptos.

Una distinción fundamental entre el análisis orientado a objetos y el análisis estructurado: división por conceptos (objetos) y no por funciones.

Por tanto, una tarea primordial de la fase de análisis consiste en identificar varios conceptos en el dominio del problema y documentar los resultados en un modelo conceptual.

9.3.5 Conceptos en el dominio del punto de venta

Por ejemplo, en el dominio del problema real de comprar productos en una tienda en una terminal de punto de venta (TPDV) intervienen los conceptos de *Tienda*, *TPDV* y una *Venta*. Por tanto, nuestro modelo conceptual (figura 9.5) puede incluir una *Tienda*, *TPDV* y una *Venta*.

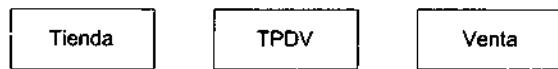


Figura 9.5 Modelo conceptual parcial en el dominio de la tienda.

9.4 Estrategias para identificar los conceptos

Nuestra meta es crear un modelo conceptual de conceptos interesantes o significativos del dominio en cuestión. En este caso, ello significa conceptos relacionados con el caso de uso *Comprar productos, versión 1*. La tarea fundamental será, pues, identificar los conceptos; se proponen dos estrategias.

La siguiente es una directriz de gran utilidad en la identificación de conceptos:

Es mejor exagerar y especificar un modelo conceptual con muchos conceptos refinados que no especificarlo cabalmente.

No piense que un modelo conceptual es más adecuado si tiene menos conceptos; generalmente sucede lo contrario.

Es frecuente omitir conceptos durante la fase inicial de identificación y descubrirlos más tarde cuando se examinen los atributos o asociaciones o durante la fase de diseño. Cuando se detecten, habrá que incorporarlos al modelo conceptual.

No se excluya un concepto simplemente porque los requerimientos no indiquen una necesidad evidente que permita recordar la información acerca de ella (criterio común de la construcción de modelos de datos para diseñar una base de datos relacional, pero no pertinente a la creación de modelos conceptuales) o porque el concepto carezca de atributos. Es perfectamente válido tener conceptos sin atributos o conceptos con un papel puramente de comportamientos en el dominio en vez de un papel informacional.

9.4.1 Obtención de conceptos a partir de una lista de categorías de conceptos

La creación de un modelo conceptual se comienza preparando una lista de conceptos idóneos a partir de la siguiente lista. Contiene muchas categorías comunes que vale la pena tener en cuenta, sin que importe el orden de importancia. Los ejemplos se tomaron de los dominios de la tienda y de las reservaciones de líneas aéreas.

Categoría del concepto	Ejemplos
objetos físicos o tangibles	TPDV Avión
especificaciones, diseño o descripciones de cosas	Especificación de Producto Descripción de Vuelo
lugares	Tienda Aeropuerto
transacciones	Venta, Pago Reservación
línea o renglón de elemento de transacciones	Ventas Línea de Producto
papel de las personas	Cajero Piloto
contenedores de otras cosas	Tienda, Cesto Avión
cosas dentro de un contenedor	Producto Pasajero
otros sistemas de cómputo o electromecánicos externos al sistema	Sistema de Autorización de Tarjeta de Crédito Control de Tráfico Aéreo
conceptos de nombres abstractos	Hambre Acrofobia
organizaciones	Departamento de Ventas Objeto Línea Aérea
eventos	Venta, Robo, Junta Vuelo, Accidente, Aterrizaje

Categoría del concepto	Ejemplos
procesos (a menudo no están representados como conceptos, pero pueden estarlo)	Venta Un Producto Reservación Asiento
reglas y políticas	Política de Reembolso Política de Cancelaciones
catálogos	Catálogo de Producto Catálogo de Partes
registros de finanzas, de trabajo, de contratos de asuntos legales	Recibo, Mayor, Contrato de Empleo Bitácora de Mantenimiento
instrumentos y servicios financieros	Línea de Crédito Existencia
manuales, libros	Manual de Personal Manual de Reparaciones

9.4.2 Obtención de conceptos a partir de la identificación de frases nominales

Otra técnica muy útil (por su simplicidad) propuesta en [Abbot83] consiste en identificar las frases nominales en las descripciones textuales del dominio de un problema y considerarlas conceptos o atributos idóneos.

Este método hay que utilizarlo con mucha prudencia; no es posible encontrar mecánicamente correspondencias entre sustantivo y concepto, y además las palabras del lenguaje natural son ambiguas.

Pese a ello, esta técnica es fuente de inspiración. Los casos expandidos de uso son una excelente descripción que puede conseguirse con este análisis. Por ejemplo, puede usarse el caso de uso *Comprar productos, versión 1*.

Acción de los actores	Respuesta del sistema
1. Este caso de uso comienza cuando un Cliente llega a una caja de TPDV con productos que desea comprar.	
2. El Cajero registra el código universal de productos (CUP) en cada producto . Si hay más de un producto , el Cajero puede introducir también la cantidad .	3. Determina el precio del producto y a la transacción de ventas le agrega la información sobre el producto . Se muestran la descripción y el precio del producto actual .

Algunas de las frases nominales anteriores son conceptos idóneos; algunas pueden ser atributos de conceptos. Por favor, consulte el lector la sección siguiente y el capítulo dedicado a los atributos: en ellos encontrará sugerencias para distinguirlos.

Una debilidad de este enfoque es la imprecisión del lenguaje natural; varias frases nominales pueden designar el mismo concepto o atributo, entre otras ambigüedades que pueden presentarse. Pese a ello, no dudamos en recomendar usarlo en combinación con el método de *Lista de categoría de conceptos*.

9.5 Conceptos idóneos para el dominio del punto de venta

A partir de la *Lista de categoría de conceptos* y del análisis de frases nominales generamos una lista de conceptos adecuados para incluirlos en la aplicación del punto de venta. La lista está sujeta a la restricción de los requerimientos y simplificaciones que se consideren en el momento: los casos simplificados de uso de *Comprar productos, versión 1*.

<i>TPDV</i>	<i>EspecificaciondeProducto</i>
<i>Producto</i>	<i>VentasLineadeProductos</i>
<i>Tienda</i>	<i>Cajero</i>
<i>Venta</i>	<i>Cliente</i>
<i>Pago</i>	<i>Gerente</i>
<i>CatalogodeProductos</i>	

9.5.1 Objetos del informe: ¿se incluye el recibo en el modelo?

El recibo es un registro de una venta y de un pago, así como un concepto relativamente prominente en el dominio de ventas; ¿debe, pues, mostrarse en el modelo? En seguida se mencionan algunos factores que han de tenerse presentes:

- El recibo es un informe de una venta. En general, no conviene incluirlo en un modelo conceptual, ya que toda su información proviene de otras fuentes. Éste es un buen motivo para excluirlo.
- El recibo cumple un papel especial respecto a las reglas de la empresa: al portador le confiere el derecho de devolver los productos adquiridos. Esta es una razón para incorporarlo al modelo.

El recibo se excluirá, porque las devoluciones de productos no se incluyen en este ciclo de desarrollo. Se justificará su inclusión durante el ciclo que aborde el caso *Devolver productos*.

9.5.2 El modelo conceptual del punto de venta (sólo conceptos)

La lista anterior de los nombres de conceptos puede representarse gráficamente (figura 9.6) en la notación del diagrama de estructura estática de UML, a fin de mostrar la génesis del modelo conceptual.



Figura 9.6 Modelo conceptual inicial del dominio del punto de venta.

En capítulos posteriores trataremos de los atributos y asociaciones del modelo conceptual.

9.6 Directrices para construir modelos conceptuales

9.6.1 Cómo construir un modelo conceptual

Aplique los siguientes pasos para crear un modelo conceptual:

Para construir un modelo conceptual:

1. Liste los conceptos idóneos usando la *Lista de categorías de conceptos* y la identificación de la frase nominal relacionadas con los requerimientos en cuestión.
2. Dibújelos en un modelo conceptual.
3. Incorpore las asociaciones necesarias para registrar las relaciones para las cuales debe reservar un espacio en la memoria (tema que se expondrá en un capítulo posterior).
4. Agregue los atributos necesarios para cumplir con las necesidades de información (tema que se tratará en un capítulo posterior).

9.6.2 La asignación de nombres y el modelado de las cosas: el cartógrafo

La estrategia del cartógrafo se aplica a los mapas y a los modelos conceptuales.

Prepare un modelo conceptual inspirándose en la metodología del cartógrafo:

- Utilice los nombres existentes en el territorio.
- Excluya las características irrelevantes.
- No agregue cosas que no existan.

El modelo conceptual es una especie de mapa de conceptos o cosas de un dominio. Este enfoque pone de relieve el papel analítico de un modelo conceptual y sugiere lo siguiente:

- Los cartógrafos se sirven de nombres del territorio —no cambian los nombres de ciudades en sus mapas. En el caso de un modelo conceptual, ello significa *utilizar el vocabulario del dominio cuando se asignan nombres a los conceptos y a los atributos*. Por ejemplo, al desarrollar el modelo de una biblioteca, al cliente se le designará con los nombres que utilice el personal: *Visitante*, *Lector* u otro semejante.

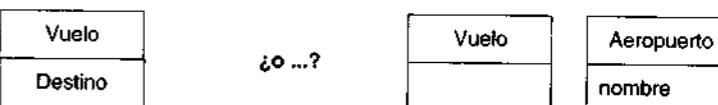
- Un cartógrafo elimina cosas en el mapa en caso de que no las juzgue pertinentes para el propósito que persigue; así, no es necesario que muestre la topografía ni las poblaciones. De modo análogo, un modelo conceptual puede excluir en el dominio del problema los conceptos que no se relacionen con los requerimientos. Por ejemplo, podemos omitir *Pluma* y *BolsadePapel* en nuestro modelo conceptual (con el conjunto actual de requerimientos), por no tener una función importante que sea obvia.
- Un cartógrafo no muestra cosas que no existan; por ejemplo, una montaña inexistente. En forma parecida, el modelo conceptual ha de excluir las cosas que *no* se encuentren en el dominio del problema en cuestión.

9.6.3 Un error que se comete frecuentemente al identificar los conceptos

Tal vez el error más frecuente cuando se crea un modelo conceptual es el de representar algo como atributo, cuando debió haber sido un concepto. Una regla práctica para no caer en él es:

Si en el mundo real no consideramos algún concepto X como número o texto, probablemente X sea un concepto y no un atributo.

Por ejemplo, pongamos el caso del dominio de las reservaciones en líneas aéreas. ¿Debería *Destino* ser un atributo de *Vuelo* o un concepto aparte *Aeropuerto*?



En el mundo real, un aeropuerto de destino no se considera número ni texto: es una cosa masiva que ocupa espacio. Por tanto, *Aeropuerto* debería ser un concepto.

En caso de duda, convierta el atributo en un concepto independiente.

9.7 Solución de los conceptos similares: comparación entre TPDV y Registro

Antaño, mucho antes del advenimiento de las terminales instaladas en el punto de venta, una tienda llevaba un *registro*: un libro donde asentaba las ventas y los pagos. Con el tiempo fue automatizado en un registro mecánico de efectivo. Hoy esa terminal desempeña el papel del registro (figura 9.7).

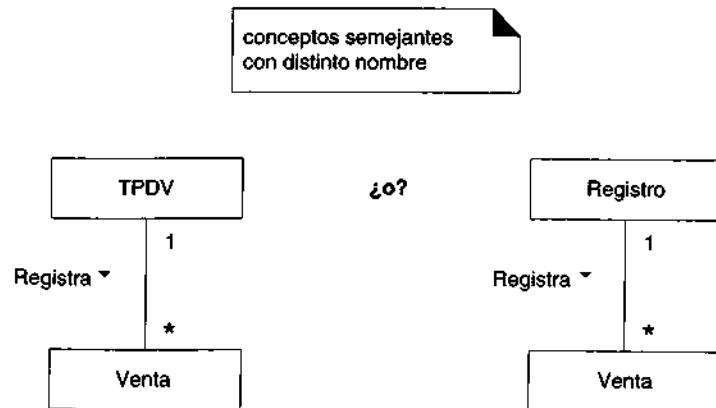


Figura 9.7 TPDV y registro son conceptos similares.

Un registro es una cosa que asienta las ventas y los pagos, pero también lo es la terminal instalada en el punto de ventas. No obstante, el término *registro* parece tener un significado más abstracto y denota una implementación menos orientada que *TPDV*. Pues bien, ¿en el modelo conceptual deberíamos utilizar el símbolo *Registro* en vez de *TPDV*?

Primero, una regla práctica consiste en que un modelo conceptual no es absolutamente correcto ni erróneo, sino de mayor o menor utilidad; es una herramienta de la comunicación.

Conforme al principio del cartógrafo, *TPDV* es un término usual en el territorio, por lo cual es un símbolo útil desde el punto de vista del conocimiento y la comunicación. *Registro* es atractivo y útil, según la meta de crear modelos que representen abstracciones y que no dependan de la implementación.¹

Ambas opciones tienen sus bondades. En este caso de estudio hemos escogido *TPDV* de modo un tanto arbitrario; también pudimos haber escogido *Registro*.

9.8

Construcción de un modelo del mundo *irreal*

Algunos sistemas de software están destinados a dominios que presentan muy poca semejanza con los dominios naturales o con los de las empresas; un ejemplo lo constituye el software de las telecomunicaciones. Todavía es posible crear un modelo conceptual en esos dominios, pero se requiere un alto grado de abstracción y abandonar los diseños comunes.

¹ Nótese que antaño un *registro* era simplemente una implementación posible de la manera de registrar las ventas. Con el tiempo, su significado ha ido generalizándose.

Enumeramos algunos conceptos adecuados que se relacionan con un intercambio de telecomunicaciones: *Mensaje*, *Conexión*, *Diálogo*, *Ruta*, *Protocolo*.

9.9 Especificación o descripción de conceptos

Suponga lo siguiente:

- La instancia de *Elemento* representa una entidad física de una tienda; puede ser incluso un número serial.
- Un *Elemento* tiene una descripción, precio y código universal de producto, los cuales no están registrados en ninguna otra parte.
- Todos los que trabajan en la tienda sufren amnesia.
- Cada vez que se vende un elemento físico, en “terreno del software” se elimina una instancia del software correspondiente a *Elemento*.

Con estas suposiciones, preguntamos: ¿qué sucede en el siguiente escenario?

Existe gran demanda de una nueva hamburguesa: *ObjetoHamburguesa*. La tienda vende todas sus existencias, lo cual significa que todas las instancias de *Elemento* de *ObjetoHamburguesa* se cancelan en la memoria de la computadora.

Ahora bien, ésta es la esencia del problema: si alguien pregunta: “¿Cuánto cuesta el *ObjetoHamburguesa*?", nadie podrá contestarle porque la memoria de su precio se anexó a las instancias inventariadas, que fueron eliminándose conforme se vendían.

Nótese asimismo que el modelo actual, si se implementa en el software tal como se describe, posee datos duplicados y maneja inefficientemente el espacio, porque la descripción, el precio y el código universal de producto se duplica en cada instancia de *Elemento* del mismo producto.

9.9.1 Necesidad de las especificaciones

El problema anterior demuestra la necesidad de un concepto de objetos que son especificaciones o descripciones de otras cosas. Para resolver el problema del *Elemento* lo que se necesita es una *EspecificacioneProducto* (o *EspecificacioneElemento*, *DescripcioneProducto*, ...) concepto que registra la información sobre los elementos. Una *EspecificacioneProducto* no representa un *Elemento*, sino una descripción *acerca de* ellos. Nótese que, aunque todos los elementos inventariados se vendan y se eliminén sus instancias correspondiente de software, se conserva la *EspecificacioneProducto*.

La descripción o especificación de objetos se relacionan bastante con aquella que describen. En un modelo conceptual, se acostumbra estipular que una *EspecificacionX* *describe una X*.

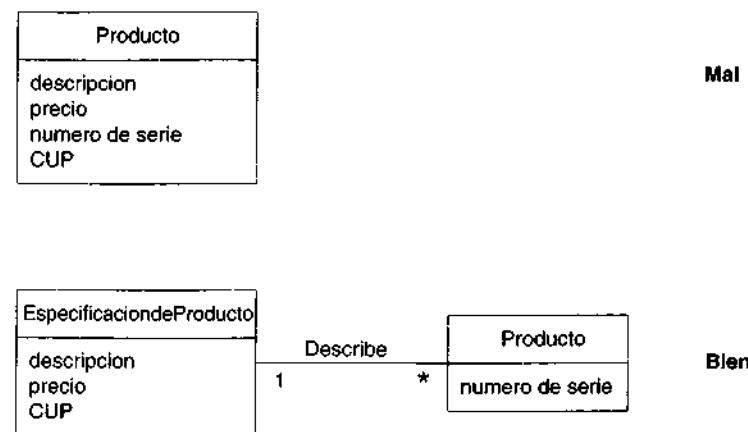


Figura 9.8 Especificaciones de otras cosas. El signo “*” significa una multiplicidad de “muchos”. Indica que una *EspecificaciondeProducto* puede describir muchos (*) *Productos*.

La necesidad de especificar los conceptos es frecuente en los dominios de ventas y productos. También lo es en la manufactura, donde se requiere una *descripción* de lo manufacturado que se distingue de la cosa manufacturada. El tiempo y el espacio han sido incluidos al explicar la causa de la especificación de conceptos, por ser muy comunes; no se trata de un concepto poco usual de la construcción de modelos.

9.9.2 ¿Cuándo se requiere especificar los conceptos?

Las siguientes directrices indican cuándo emplear las especificaciones.

Incorpore una especificación o descripción de conceptos (por ejemplo, *EspecificaciondeProducto*) cuando:

- La eliminación de las instancias de las cosas que describen *Elemento*, por ejemplo, da por resultado una pérdida de la información que ha de conservarse, debido a la asociación incorrecta de la información con lo eliminado.
- Reduce información redundante o duplicada.

9.9.3 Otro ejemplo de especificación

He aquí un último ejemplo: imagine que una compañía aérea sufre el accidente fatal de uno de sus aviones. Suponga que todos los vuelos quedan cancelados por seis meses, mientras se lleva a cabo la investigación. Suponga además que, cuando se cancelan los vuelos, sus correspondientes objetos de software *Vuelo* se eliminan en la memoria de la computadora. Así, todos los objetos *Vuelo* quedan eliminados tras el accidente.

Si el único registro del aeropuerto al cual se dirige un vuelo está en las instancias *Vuelo*, que representan vuelos específicos en determinado día y hora, ya no habrá un registro de qué rutas tiene la línea aérea.

Para resolver este problema, se requiere una *DescripciondeVuelo* (o *EspecificaciondeVuelo*) que describa un vuelo y su ruta, aun cuando no esté programado un vuelo determinado (figura 9.9).

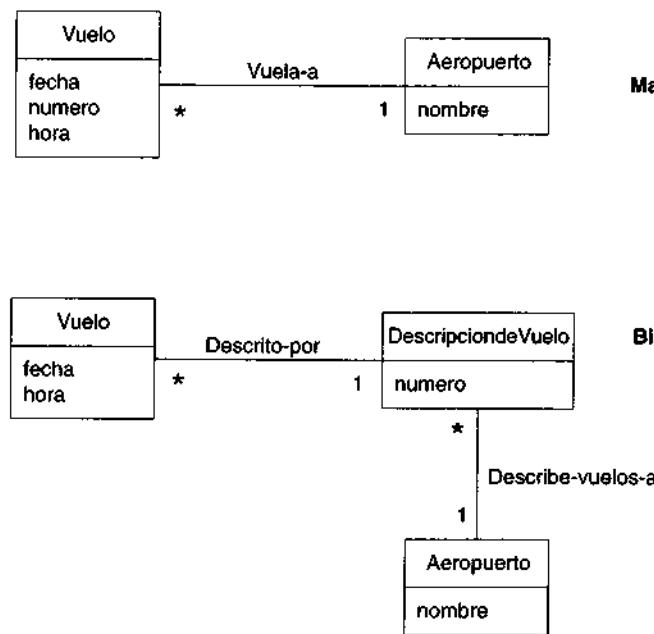


Figura 9.9 Especificaciones de otros elementos.

9.10 Definición de términos en el lenguaje UML

En UML, se emplean los términos “clase” y “tipo”, no así “concepto”. No existe consenso unánime respecto al significado de clase y tipo; así que para evitar ambigüedades el UML define rigurosamente ambos términos tal como se utilizan en su metamodelo (el modelo de UML), aunque otros autores y profesionales usan definiciones alternas y antagónicas.

Cualquiera que sea la definición, lo importante es su utilidad para distinguir entre la perspectiva de un analista de dominios que observa conceptos reales, como venta, y los diseñadores de software que especifican entidades de programas; por ejemplo, la clase *Venta* en Java. El UML sirve, entre otras cosas, para explicar concretamente las perspectivas de notación y terminología muy afines; de ahí la importancia de tener presente qué perspectiva va a adoptarse (un análisis, un diseño o una vista de la implementación).

Para simplificar nuestra exposición, en este libro con el término "concepto" designaremos cosas del mundo real y con el de "clase", las especificaciones e implementaciones del software.

La definición de **clase** en UML es "una descripción de un conjunto de objetos que comparten los mismos atributos, operaciones, métodos, relaciones y semántica" [BJR97]. Algunos autores limitan la definición de clase a una implementación concreta de software, digamos una clase en Java [Fowler96]. Pero, en el UML, este vocablo tiene una acepción más general: abarca especificaciones que anteceden a la implementación. En ese lenguaje, a una clase implementada de software se le llama más concretamente **clase de implementación**.

En UML, una **operación** es "un servicio que puede solicitarse a un objeto para que realice un comportamiento" [BJR97], y **método** es la implementación de una operación que especifica el algoritmo o procedimiento de esta última.

La definición de **tipo** en UML se asemeja a la de clase —describe un conjunto de objetos parecidos con atributos y operaciones—, pero no puede incluir métodos. De ello se deduce que un tipo es una *especificación* de una entidad de software y no una implementación. Ello significa también que un tipo de UML es independiente del lenguaje.

Aunque no sea estrictamente exacto dentro del contexto del lenguaje UML, este libro a veces utiliza los vocablos "concepto" y "tipo" indistintamente, porque en el uso coloquial el vocablo "tipo" a menudo se define como sinónimo de un concepto del mundo real [MO95].

El término **interfaz** se define como un conjunto de operaciones visibles en el exterior. En el UML, puede estar asociada a tipos y clases (y también a paquetes que agrupan elementos). Aunque los conceptos reales pueden tener una interfaz (la de un teléfono, por ejemplo), el término suele emplearse dentro del contexto de una interfaz para entidades de software, como la interfaz *Runnable* de Java.

9.11 Modelos Patrón

El Modelo Conceptual se compone de diagramas estáticos de estructura UML que ilustran conceptos en el dominio, como se muestra en la figura 9.10.

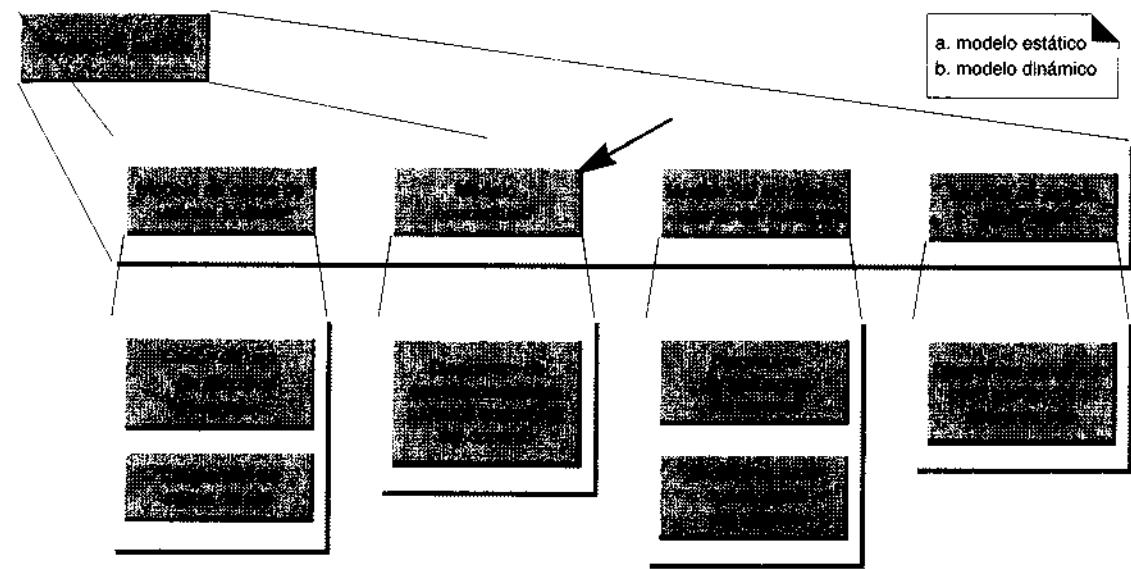


Figura 9.10 El modelo de análisis.

MODELO CONCEPTUAL: AGREGACIÓN DE LAS ASOCIACIONES

Objetivos

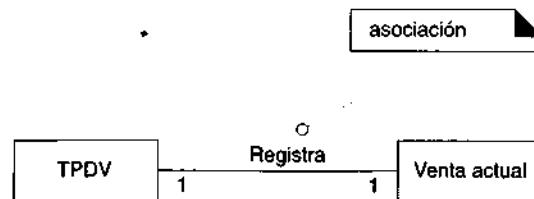
- Identificar las asociaciones de un modelo conceptual.
- Distinguir entre las asociaciones que es necesario conocer y las que se requieren sólo para la comprensión.

10.1 Introducción

Es necesario identificar las asociaciones de los conceptos que se requieren para satisfacer los requerimientos de información de los casos de uso en cuestión y los que contribuyen a entender el modelo conceptual. En el presente capítulo examinaremos la identificación de las asociaciones adecuadas y las incorporaremos al modelo conceptual del sistema del punto de venta.

10.2 Asociaciones

La **asociación** es una relación entre dos conceptos que indica alguna conexión significativa e interesante entre ellos (figura 10.1).

**Figura 10.1** Asociaciones.

En el lenguaje UML se describen como “relaciones estructurales entre los objetos de diversos tipos”.

10.2.1 Criterios de las asociaciones útiles

Las asociaciones que vale la pena mencionar suelen incluir el conocimiento de una relación que ha de preservarse durante algún tiempo: puede tratarse de milisegundos o años según el contexto. En otras palabras, ¿entre qué objetos hemos de tener algún recuerdo de una relación? Por ejemplo, ¿debemos recordar cuáles instancias de *VentasLineadeProducto* están asociadas a la instancia *Venta*? Claro que sí, porque de lo contrario no sería posible reconstruir la venta, imprimir el recibo ni calcular el total de la venta.

Examine la conveniencia de incluir las siguientes asociaciones en un modelo conceptual:

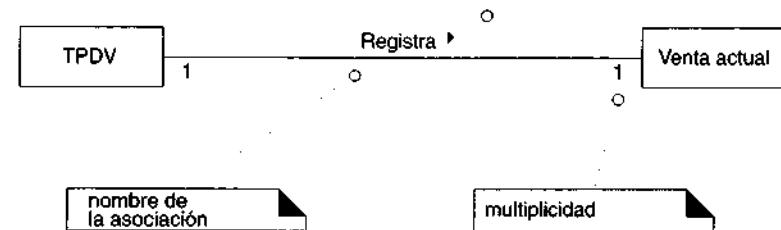
- Las asociaciones en que el conocimiento de la relación ha de ser preservado durante algún tiempo (asociaciones que “deben conocerse”).
- Las asociaciones provenientes de la *Lista de asociaciones comunes*.

En cambio, ¿necesitamos recordar una relación entre una *Venta* actual y un *Gerente*? La respuesta es negativa: los requerimientos no indican que haga falta ese tipo de relación. Tal vez no sea incorrecto mostrar una relación entre una *Venta* y *Gerente*, pero no es indispensable ni útil dentro del contexto de nuestros requerimientos.

10.3 Notación de las asociaciones en el UML

Una asociación se representa como una línea entre conceptos con el nombre de la asociación. Ésta es intrínsecamente bidireccional, o sea es posible un nexo lógico entre los objetos de un tipo y los del otro. Este vínculo es totalmente abstracto; *no* es una afirmación sobre las conexiones entre las entidades del software.

-“flecha de dirección de la lectura”
 -*no* tiene otro significado que el de indicar la dirección en que debe leerse el nombre de la asociación
 -a menudo se excluye

**Figura 10.2** Notación de las asociaciones en el lenguaje UML.

Los extremos de una asociación pueden contener una expresión de multiplicidad que indique la relación numérica entre las instancias de los conceptos.

Una flecha opcional de la dirección de la lectura (o “flecha de la dirección del nombre”) indica la dirección en que debe leerse el nombre de la asociación; no denota la dirección de visibilidad o de navegación. En su ausencia, por convención la asociación se lee de izquierda a derecha o de arriba hacia abajo, aunque el UML no hace de esto una regla (figura 10.2).

La flecha de dirección de la lectura no tiene un valor semántico; tan sólo es una ayuda para leer el diagrama.

10.4 Identificación de las asociaciones: lista de asociaciones comunes

Comience a agregar las asociaciones utilizando la lista anexa. Contiene categorías comunes que normalmente vale la pena incluir. Los ejemplos están tomados de los dominios de la tienda y de la reservación de las líneas aéreas.

Categoría	Ejemplos
A es una parte física de B	Caja—TPDV Ala—Avion
A es una parte lógica de B	VentasLineadeProducto—Venta TramodeVuelo—RutadeVuelo
A está físicamente contenido en B	TPDV—Tienda, Producto—Estante Pasajero—Avion
A está contenido lógicamente en B	DescripciondeProducto—Catalogo Vuelo—ProgramadeVuelo
A es una descripción de B	DescripciondeProducto—Producto DescripciondeVuelo—Vuelo
A es un elemento de linea en una transacción o reporte B	VentasLineadeProducto—Venta TrabajodeMantenimiento—Mantenimiento
A se conoce/introduce/registra/presenta/captura en B	Venta—TPDV Reservacion—ListadePasajeros
A es miembro de B	Cajero—Tienda Piloto—Avion
A es una subunidad organizacional de B	Departamento—Tienda Mantenimiento—Linea Aerea
A usa o dirige a B	Cajero—TPDV Piloto—Avion
A se comunica con B	Cliente—Cajero AgentededeReservaciones—Pasajero
A se relaciona con una transacción B	Pago—Venta Pasajero—Boleto

Categoría	Ejemplos
A es una transacción relacionada con otra transacción B	Pago—Venta Reservacion—Cancelacion
A está contiguo a B	TPDV—TPDV Ciudad—Ciudad
A es propiedad de B	TPDV—Tienda Avion—Linea aerea

10.4.1 Asociaciones de alta prioridad

A continuación se enumeran algunas categorías de alta prioridad que siempre conviene incluir en un modelo conceptual:

- A es una *parte física o lógica* de B.
- A está *física o lógicamente contenido* en B.
- A está *registrado en* B.

10.5 ¿Qué grado de detalle deberían tener las asociaciones?

Las asociaciones son importantes, pero una falla habitual al crear modelos conceptuales es el excesivo tiempo que, durante la investigación, se dedica al intento de descubrirlas. Es indispensable convencerse de lo siguiente:

Es mucho más importante identificar los conceptos que las asociaciones. El tiempo consagrado a la creación del modelo conceptual debería destinarse a identificar los conceptos, no las asociaciones.

10.6 Directrices de las asociaciones

Directrices de las asociaciones:

- Concentrarse en las asociaciones en que el conocimiento de la relación ha de preservarse durante algún tiempo (asociaciones que "es necesario conocer").
- Es más importante identificar los conceptos que las asociaciones.
- Muchas asociaciones tienden a confundir el modelo conceptual en vez de aclararlo. A veces se requiere mucho tiempo para descubrirlas, y los beneficios son escasos.
- No incluir las asociaciones redundantes ni las derivables.

10.7 Papeles

A los extremos de una asociación se les llama **papeles**. Éstos pueden tener:

- nombre
- expresión de multiplicidad
- navegabilidad

En este momento se investiga la multiplicidad, pero las dos características restantes se estudiarán en capítulos posteriores.

10.7.1 Multiplicidad

La **multiplicidad** define cuántas instancias de un tipo *A* pueden asociarse a una instancia del tipo *B* en determinado momento (figura 10.3).

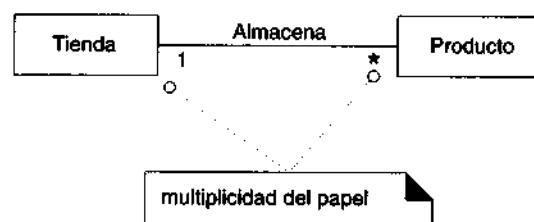


Figura 10.3 Multiplicidad en una asociación.

Por ejemplo, una instancia individual de una *Tienda* puede asociarse a "muchas" instancias (cero o más marcadas con *) de *Producto*.

En la figura 10.4 se ofrecen algunos ejemplos de las expresiones de multiplicidad.

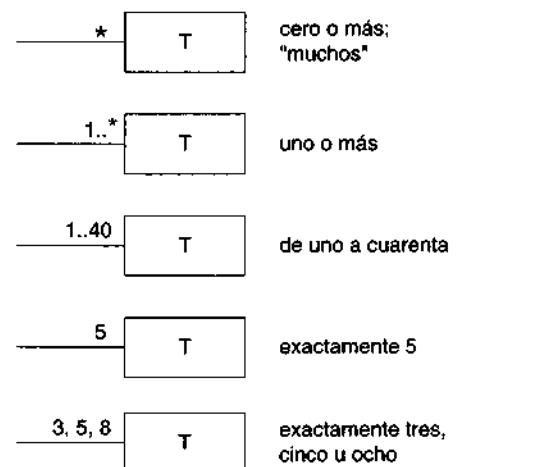


Figura 10.4 Valores de la multiplicidad.

En el UML, el valor de multiplicidad depende del contexto. Rumbaugh [Rumbaugh91] da un excelente ejemplo de *Persona* y *Compañía* en la asociación *Trabaja-para*. Del contexto del modelo dependerá indicar si una instancia de *Persona* se aplica a una o a varias instancias de *Compañía*; al departamento de impuestos le interesan *muchas*; a un sindicato probablemente le interese sólo *una*.

10.8 Asignación de nombre a las asociaciones

Se asigna nombre a una asociación basándose en el formato *NombredelTipo-Frase-Nominal-NombredelTipo*, donde la frase nominal genera una secuencia que es legible y significativa dentro del contexto del modelo.

Los nombres de las asociaciones comienzan con una mayúscula. Una frase nominal debe construirse con guiones.

En la figura 10.5, la dirección por omisión en que debe leerse el nombre de una asociación es de izquierda a derecha o de arriba hacia abajo. No es así en la dirección por omisión de UML, aunque se trata de una convención relativamente usual.

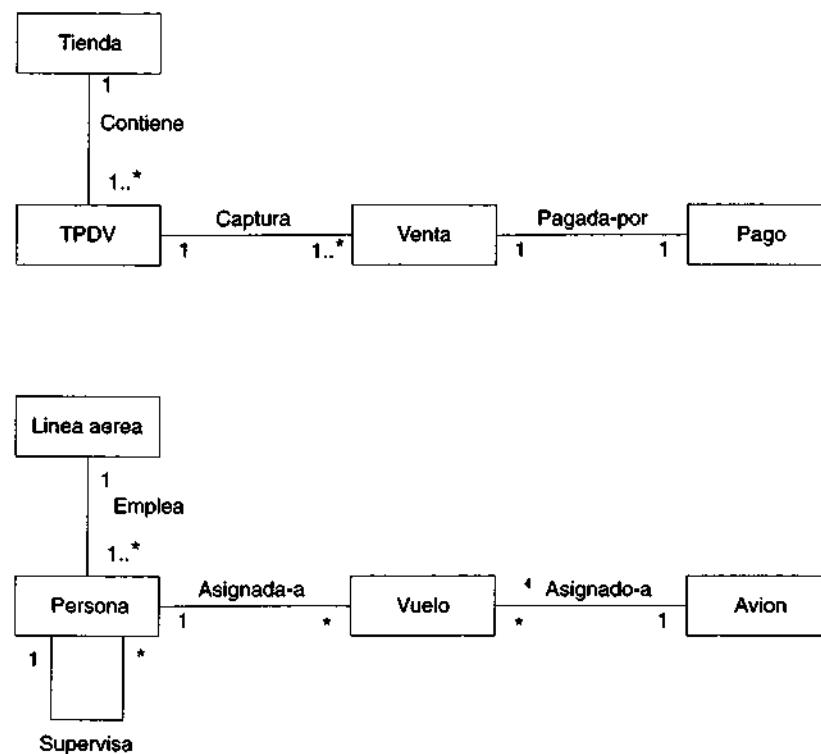


Figura 10.5 Nombres de las asociaciones.

10.9 Asociaciones múltiples entre dos tipos

Dos tipos pueden tener varias asociaciones entre ellos; esto sucede con frecuencia. No hay un ejemplo sobresaliente en nuestro sistema TPDV, pero en el dominio de la línea aérea encontramos uno en las relaciones entre *Vuelo* y *Aeropuerto* (figura 10.6). Las asociaciones *volar-hacia* y *volar-de* son relaciones netamente diferentes que han de mostrarse por separado. Adviértase asimismo que *no se garantiza* que todos los vuelos aterricen en un aeropuerto!

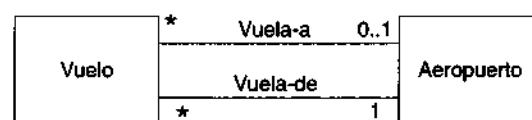


Figura 10.6 Asociaciones múltiples.

10.10 Asociaciones e implementación

Durante la fase de análisis, una asociación *no* es una proposición sobre los flujos de datos, variables de instancia ni conexiones de objetos en una solución de software; es una proposición de que una relación es significativa en un sentido puramente analítico: en el mundo real. En la práctica, muchas de estas relaciones suelen implementarse en los programas como trayectorias de navegación y visibilidad; pero su presencia en una vista investigadora o analítica de un modo conceptual no requiere la implementación.

Cuando se crea un modelo conceptual, podemos definir asociaciones que no se necesitarán en la construcción. Y a la inversa: podemos descubrir asociaciones que han de ser implementadas pero que se omitieron durante la fase de análisis. En tal caso, habría que actualizar el modelo para que incluyan esos descubrimientos.

Más adelante explicaremos las formas de implementar las asociaciones en un lenguaje de programación orientado a objetos (lo usual es servirse de un atributo que apunte a una instancia de la clase asociada), pero por ahora conviene verlas como meras expresiones analíticas, *no* como proposiciones acerca de una solución de base de datos ni de software. Como de costumbre, posponemos las consideraciones de diseño y así nos liberamos de información y decisiones prematuras en el modelo de análisis, a la vez que aumentamos al máximo nuestras opciones para después.

10.11 Asociaciones del dominio del punto de venta

Ahora podemos agregar asociaciones a nuestro modelo conceptual del sistema del punto de venta. Deberíamos incorporar las que indican los requerimientos (los casos de uso, por ejemplo), las que conllevan la necesidad de recordar o que de alguna otra manera nos sugiere nuestra percepción del dominio del problema. Cuando acometamos un nuevo problema, hay que repasar y estudiar las categorías comunes de las asociaciones mencionadas en páginas anteriores, pues representan muchas de las asociaciones pertinentes que generalmente es preciso registrar.

10.11.1 Relaciones inolvidables en la tienda

La siguiente muestra de asociaciones se justifica por la necesidad de conocerlas. Se funda en los casos de uso que hemos venido examinando.

TPDV Captura Venta

Para conocer la venta actual genera un total, e imprime un recibo.

<i>Venta pagada en efectivo</i>	Para saber si se pagó la venta, relaciona la cantidad ofrecida con el total de la venta e imprime un recibo.
<i>CatalogodeProductos registra EspecificaciondeProducto</i>	Para recuperar una <i>EspecificaciondeProducto</i> , con un código universal de producto.

10.11.2 Aplicación de la categoría de la lista de comprobación de las asociaciones

Recorremos la lista de comprobación, basándonos en los tipos anteriormente identificados y teniendo presentes los requerimientos actuales del caso de uso.

Concepto A → Concepto B Relación entre A y B		Aplicación TPDV
A es una parte física de B		no aplicable
A es una parte lógica de B		VentasLineadeProducto—Venta
A está contenido físicamente en B		TPDV—Tienda Producto—Tienda
A está contenido lógicamente en B		EspecificaciondeProducto—CatalogodeProductos CatalogodeProductos—Tienda
A es un descripción de B		EspecificaciondeProducto—Producto
A es una línea de una transacción o reporte B		VentasLineadeProducto—Venta
A se introduce/registra/presenta/captura en B		Ventas(terminadas)—Tienda Venta(actual)—TPDV
A es miembro de B		Cajero—Tienda
A es una subunidad organizacional de B		No aplicable
A usa o dirige a B		Cajero—TPDV Gerente—TPDV Gerente—Cajero, pero probablemente no aplicable.

Concepto A → Concepto B Relación entre A y B	Aplicación TPDV
A se comunica con B	Cliente—Cajero
A se relaciona con una transacción B	Cliente—Pago Cajero—Pago
A es una transacción relacionada con otra transacción B	Pago—Venta
A sigue a B	TPDV—TPDV, pero probablemente no aplicable
A es propiedad de B	TPDV—Tienda

10.12 Modelo conceptual del punto de venta

El modelo conceptual de la figura 10.7 muestra un conjunto de conceptos y asociaciones idóneas para nuestra aplicación al punto de venta. Las asociaciones se tomaron principalmente de la lista de comprobación de asociaciones adecuada.

10.12.1 ¿Se conservan sólo las asociaciones que deben conocerse?

El conjunto de asociaciones que se incluye en el modelo conceptual de la figura 10.7 se obtuvo de manera bastante mecánica a partir de la lista de comprobación. Pero tal vez hay que ser más selectivos con las asociaciones que contendrá nuestro modelo conceptual. Desde el punto de vista de la comunicación, no conviene saturar el modelo con asociaciones que no sean indispensables y que no mejoren nuestro conocimiento. El exceso de ellas no aclara sino que oscurece la situación.

Como señalamos con anterioridad, recomendamos los siguientes criterios para presentar las asociaciones:

- Concentrarse en las asociaciones en que el conocimiento de la relación ha de ser preservado durante algún tiempo (asociaciones "que deben conocerse").
- No incluir las asociaciones redundantes ni las derivables.

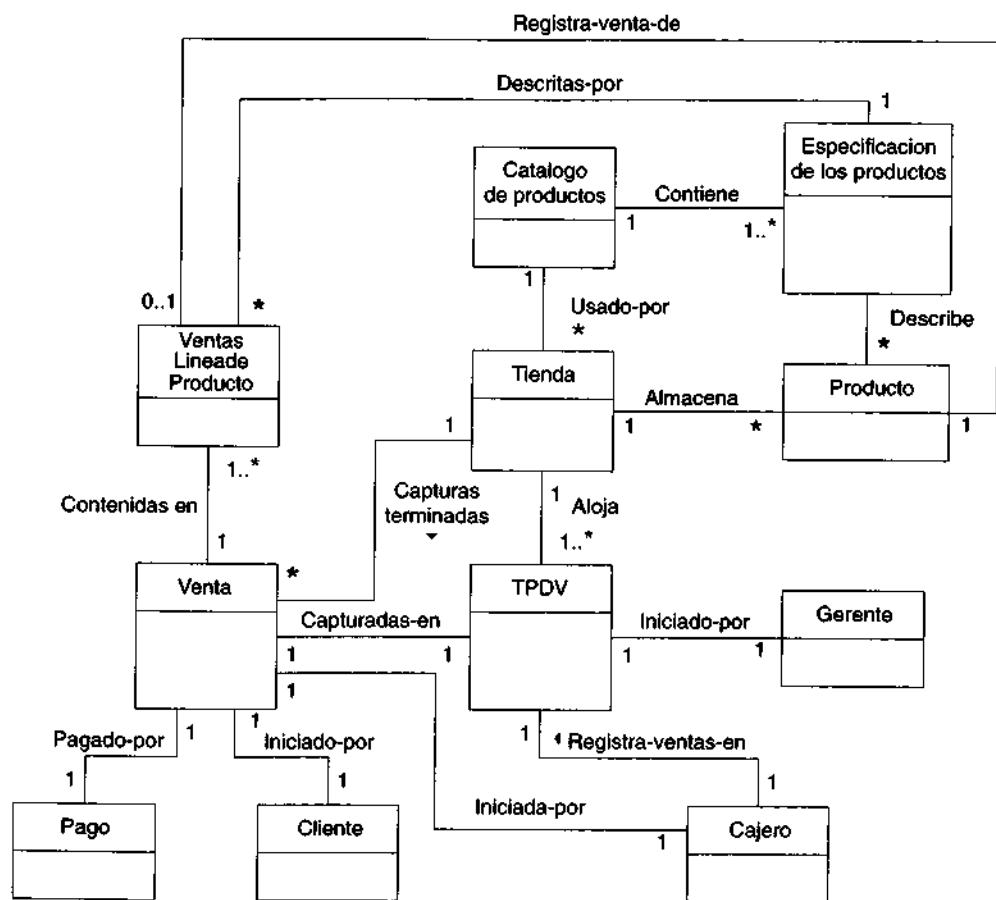


Figura 10.7 Modelo conceptual aplicado al punto de venta.

Conforme a nuestra recomendación, no son indispensables todas las asociaciones que se muestran en cierto momento. Estudie detenidamente la tabla anexa:

Asociación	Explicación
Venta capturada-por Cajero	Los requerimientos no indican la necesidad de conocer ni de registrar al cajero actual. Además, es derivable si existe la asociación TPDV Usado-por Cajero.
TPDV Usado-por Cajero	Los requerimientos no indican la necesidad de conocer ni de registrar al cajero actual.

Asociación	Explicación
TPDV Iniciado-por Gerente	Los requerimientos no indican la necesidad de conocer ni de registrar al gerente que inició un TPDV.
Venta Iniciada-por Cliente	Los requerimientos no indican la necesidad de conocer ni de registrar al cliente actual que inició una venta.
Tienda Almacena Producto	Los requerimientos no indican la necesidad de conocer o mantener la información del inventario.
VentasLineadeProducto Registra-venta-de-Producto	Los requerimientos no indican la necesidad de mantener la información de inventario.

Nótese que la capacidad de justificar una asociación atendiendo a la necesidad de conocerla depende de los requerimientos; un cambio de ellos —por ejemplo, exigir que la identificación del cajero aparezca en un recibo— altera la necesidad de recordar una relación.

A partir del análisis anterior, *tal vez* se justifique suprimir las asociaciones en cuestión.

10.12.2 Las asociaciones que necesitan conocerse y las que facilitan la comprensión

Un criterio estricto de la necesidad de conocer aplicado a la conservación de asociaciones dará origen a un “modelo mínimo de información” sobre lo que se requiere para construir un modelo del dominio del problema, modelo que estará acotado por los requerimientos en cuestión. Pero este procedimiento puede producir un modelo que no facilite (ni a los diseñadores ni a terceros) la comprensión cabal del dominio.

Algunas veces vemos el modelo conceptual no como un modelo riguroso de información, sino como una herramienta de la comunicación, con la cual intentamos entender los conceptos importantes y sus relaciones. Desde esta perspectiva, eliminar algunas asociaciones que no reclamen estrictamente el criterio de la necesidad de conocer puede originar un modelo inadecuado: no comunica las ideas ni las relaciones más importantes.

Esto lo vemos, por ejemplo, en la aplicación al punto de venta: aunque conforme al criterio estricto de la necesidad de conocer, quizás no sea preciso registrar *Venta Iniciada-por Cliente*, su ausencia omite un aspecto importante para comprender el dominio: el hecho de que un cliente genera ventas.

En lo tocante a las asociaciones, un buen modelo ocupa un punto intermedio de un modelo basado en la necesidad mínima de conocimiento y otro que contenga todas las relaciones posibles. ¿Cuál es el criterio básico para juzgar su valor? Uno podría ser:

¿cumple con todos los requerimientos de la necesidad de conocer y además comunica claramente una comprensión esencial de los conceptos importantes en el dominio del problema?

Enfatice las asociaciones que deben conocerse, pero incorpore también las opcionales que se requieren sólo para la comprensión, con el fin de enriquecer el conocimiento básico del dominio.

MODELO CONCEPTUAL: AGREGACIÓN DE LOS ATRIBUTOS

Objetivos

- Identificar los atributos en un modelo conceptual.
- Distinguir entre atributos correctos e incorrectos.

11.1 Introducción

Es necesario identificar los atributos de los conceptos que se necesitan para satisfacer los requerimientos de información de los casos de uso en cuestión. En este capítulo examinaremos la identificación de los atributos idóneos y le agregaremos atributos al modelo conceptual del dominio, que hemos aplicado al punto de venta.

Recuérdese que el modelo conceptual es una representación de cosas reales, no de componentes del software. Cualquier afirmación concerniente a los atributos ha de interpretarse dentro del contexto de entidades del mundo real.

11.2 Atributos

Un atributo es un valor lógico de un dato de un objeto.

Incluya los siguientes atributos en un modelo conceptual:

Aquellos en que los requerimientos (por ejemplo, los casos de uso) indican o llevan la necesidad de recordar información.

Por ejemplo, un recibo de ventas normalmente incluye la fecha y la hora. En consecuencia, el concepto *Venta* requiere los atributos *fecha* y *hora*.

11.3 Notación de los atributos en el UML

Los atributos se muestran en la segunda sección de la sección de conceptos (Figura 11.1). Es opcional indicar su tipo.



Figura 11.1 Concepto y atributos.

11.4 Tipos de atributos válidos

Hay algunas cosas que no deberían representarse como atributos, sino más bien como asociaciones. En esta sección trataremos de los atributos válidos.

11.4.1 Conserve simples los atributos

Los tipos más simples de atributos son los que, en la práctica, suelen considerarse los tipos primitivos de datos. Por lo regular el tipo de un atributo no debería ser un concepto complejo del dominio, como *Venta* o *Aeropuerto*. Por ejemplo, el siguiente atributo *actual* de *TPDV* en el tipo *Cajero* de la figura 11.2 no es idóneo porque con su tipo se indica una *TPDV*, que no es un tipo de atributo simple (digamos *Número* o *Cadena*). La forma más conveniente de expresar que un *Cajero* utiliza una *TPDV* es recurrir a una asociación, no a un atributo.

En un modelo conceptual, es preferible que los atributos sean **atributos simples o valores puros de datos**.

Entre los tipos comunes de atributos simples más frecuentes se cuentan:

Booleano, Fecha, Número, Cadena (Texto), Hora

He aquí otros tipos comunes:

Dirección, Color, Geometría (Punto, Rectángulo, ...), Número telefónico, Número del Seguro Social, Código Universal de Producto (CUP), SKU, CP o códigos postales, tipos enumerados

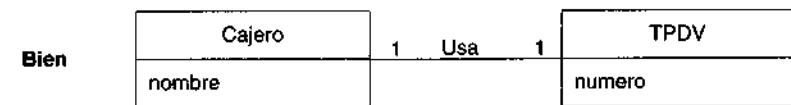
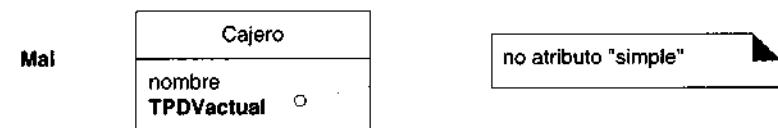


Figura 11.2 Relacione con asociaciones, no con atributos.

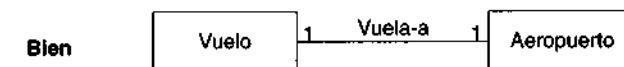


Figura 11.3 No represente como atributos los conceptos complejos de dominio; use asociaciones.

Repetimos un ejemplo anterior: una confusión frecuente consiste en modelar como atributo un concepto complejo del dominio. Así, un aeropuerto de destino no es una cadena de caracteres en realidad; es una realidad compleja que ocupa muchos kilómetros cuadrados de espacio. Por tanto, deberíamos relacionar *Vuelo* con *Aeropuerto* a través de una asociación y no con un atributo, como se indica en la figura 11.3.

Relacione conceptos con una asociación, no con un atributo.

11.4.2 Comparación entre análisis y diseño: ¿qué decir de los atributos en código?

La restricción de que los atributos del modelo conceptual sean únicamente tipos de datos simples *no* significa que los de C++, Java o Smalltalk (miembros de datos, variables de instancias) deban ser sólo de tipos primitivos de datos simples. El modelo conceptual se centra en afirmaciones analíticas puras sobre un dominio del problema, no en entidades de software.

Más adelante, durante las fases del diseño y construcción, veremos que las asociaciones entre los objetos expresados en el modelo conceptual a menudo se implementarán como atributos que apuntan a otros tipos complejos. Pero ésta no es más que una de muchas posibles soluciones de diseño para implementar una asociación; por tanto, la decisión habrá de aplazarse durante la fase de análisis. La etapa de análisis no es el momento de tomar prematuramente decisiones concernientes al diseño.

11.4.3 Valores puros de datos

En términos más generales, los atributos deberían ser **valores puros de datos** (o **Tipos de Datos**, en el lenguaje UML), en los cuales la identidad única no es significativa (dentro del contexto de nuestro modelo o sistema) [Rumbaugh91]. Por ejemplo, no es (generalmente) significativo distinguir entre:

- Instancias aisladas del *Numero* 5.
- Instancias aisladas de la *Cadena* "gato".
- Instancias aisladas de *NumeroTelefonico* que contengan el mismo número.
- Instancias aisladas de *Direccion* que contengan la misma dirección.

En cambio, sí es significativo distinguir (por identidad) entre dos instancias aisladas de una *Persona* cuyos nombres sean "Jill Smith", porque ambas instancias pueden representar diferentes individuos con un mismo nombre.

Por lo que respecta al software, hay pocas situaciones donde compararíamos la dirección de memoria de las instancias de *Numero*, *Cadena*, *NumeroTelefonico* o *Direccion*; sólo las comparaciones basadas en valores son relevantes. En cambio, es posible comparar las direcciones de memoria de las instancias *Persona* y distinguirlas, aun cuando tuvieran los mismos valores de atributos, por ser importante su identidad única.

Un elemento de un tipo puro de datos puede exemplificarse en la sección de otro concepto destinada a los atributos, aunque también es aceptable modelarlo como un concepto diferente.

Los valores puros de datos también se denominan **objetos de valor**.

El concepto de valor puro es útil. Una regla práctica consiste en aplicar la prueba básica de los tipos de atributos "simples": convertirlos en atributo si espontáneamente los concebimos como número, cadena, booleano, fecha u hora (y así sucesivamente); de lo contrario, representarlos como un concepto aparte.

En caso de duda, defina algo como concepto aislado y no como atributo.

11.4.4 Deterioro del diseño: ningún atributo debe incluirse como llave foránea

Los atributos no deberían servir para relacionar conceptos en el modelo conceptual. La violación más frecuente de esta regla consiste en agregar un tipo de **atributo de llave foránea**, lo cual suele hacerse con los diseños de bases de datos relacionales, a fin de asociar dos tipos. Por ejemplo, en la figura 11.4 el atributo *NumeroTPDVactual* en el tipo *Cajero* no es conveniente, porque su propósito es relacionar *Cajero* con un objeto *TPDV*. La mejor manera de expresar que un *Cajero* usa una *TPDV* consiste en recurrir a la asociación, no a un atributo de llave foránea. Una vez más, recuerde el lector relacionar los tipos a través de una asociación y no con un atributo.

Hay muchas formas de relacionar los objetos —las llaves foráneas son una de tantas—; además para evitar el deterioro del diseño, deberíamos posponer hasta la fase del diseño cómo vamos a implementar la relación.

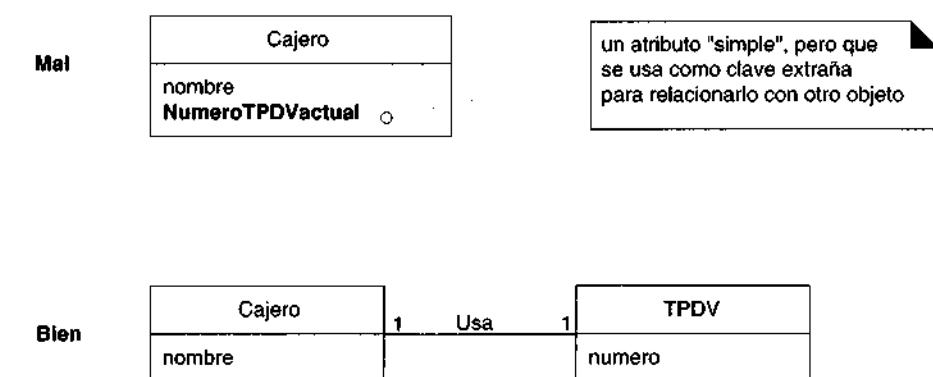


Figura 11.4 No utilice los atributos como claves extrañas.

11.5 Tipos de atributos no primitivos

En el modelo conceptual el tipo de un atributo puede expresarse como no primitivo por sí mismo. Así, en el sistema del punto de venta hay un Código universal de producto (CUP). Suele considerársele como un número. ¿Debería entonces representarse como un tipo no primitivo? Aplique la siguiente directriz:

- Represente como tipo no primitivo lo que inicialmente puede considerarse un tipo primitivo de datos (un número o una cadena, por ejemplo), si:
- Se compone de secciones independientes.
 - número telefónico, nombre de persona
 - Normalmente se asocian a él operaciones como el análisis o la validación.
 - número del seguro social
 - Posee otros atributos.
 - el precio promocional podría tener fecha de inicio y de terminación
 - Es una cantidad con una unidad.
 - el importe del pago tiene una unidad monetaria

Al aplicar las directrices anteriores a los atributos del modelo conceptual del punto de venta, se obtiene el siguiente análisis:

- El código *CUP* debería ser un tipo *CUP* no primitivo, porque puede ser validado verificando la suma y porque puede poseer otros atributos (por ejemplo, el fabricante que lo asignó).
- Los atributos de *precio* y *cantidad* deberían ser los tipos no primitivos *Cantidad*, por ser cantidades en una unidad monetaria.
- El atributo *direccion* debería ser un tipo no primitivo *Direccion*, porque consta de secciones independientes.

Los tipos *CUP*, *Direccion* y *Cantidad* son valores puros de datos (la identidad única no es significativa); así que podemos mostrarlos en la sección de la casilla dedicada a los atributos en vez de relacionarse con una línea de asociación.

11.5.1 ¿Dónde mostrar los tipos no primitivos de atributos y los valores puros?

¿Deberíamos mostrar el tipo de *CUP* como concepto aparte en un modelo conceptual? La respuesta es: depende de lo que queramos destacar en el diagrama. Por ser el *CUP*

un *valor puro de datos* (la identidad única no es importante), podría presentarse en la sección dedicada a atributos en la casilla de conceptos, como se advierte en la figura 11.5. Pero por no ser un tipo primitivo con sus propios atributos y asociaciones, tal vez sería interesante mostrarlo como concepto en su propia caja. No existe en este caso una respuesta correcta; dependerá de la manera en que vamos a utilizar el modelo conceptual como herramienta de comunicación y también dependerá de la importancia del concepto en el dominio.

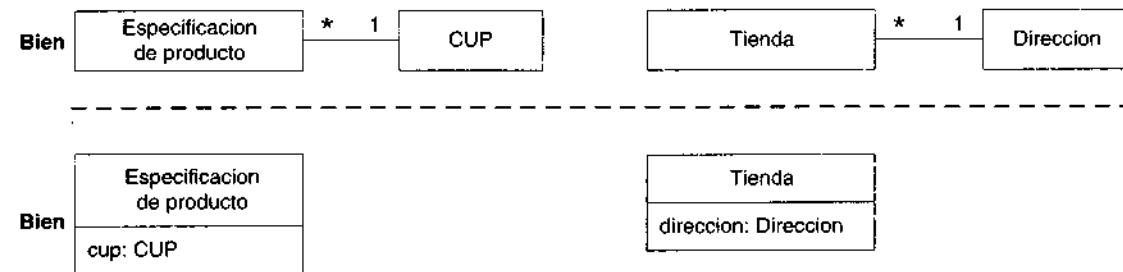


Figura 11.5 Si el tipo de atributo es un valor puro de datos, puede mostrarse en la sección de atributos.

Un modelo conceptual es una herramienta de la comunicación; las decisiones sobre lo que debería mostrarse han de tomarse teniendo presente eso.

11.6 Modelado de cantidades y unidades de los atributos

En términos informales, el importe de un *Pago* puede ser representado como un *Número*. Pero, en general, éste no es un esquema robusto ni flexible porque las *unidades* de un número a menudo son importantes. Considere:

- moneda
- velocidad

Por ejemplo suele requerirse convertir las unidades (por ejemplo, las conversiones del sistema inglés al sistema métrico). Suponiendo que el software del punto de venta esté destinado al mercado internacional, habría que conocer la unidad monetaria de los pagos.

La solución consiste en representar *Cantidad* como un concepto distinto, con una *Unidad* asociada. Y como se supone que las cantidades son valores puros de datos (la identidad única carece de importancia), es aceptable limitarse a presentarlas en la sección de la casilla destinada a los tipos (véase la figura. 11.6).

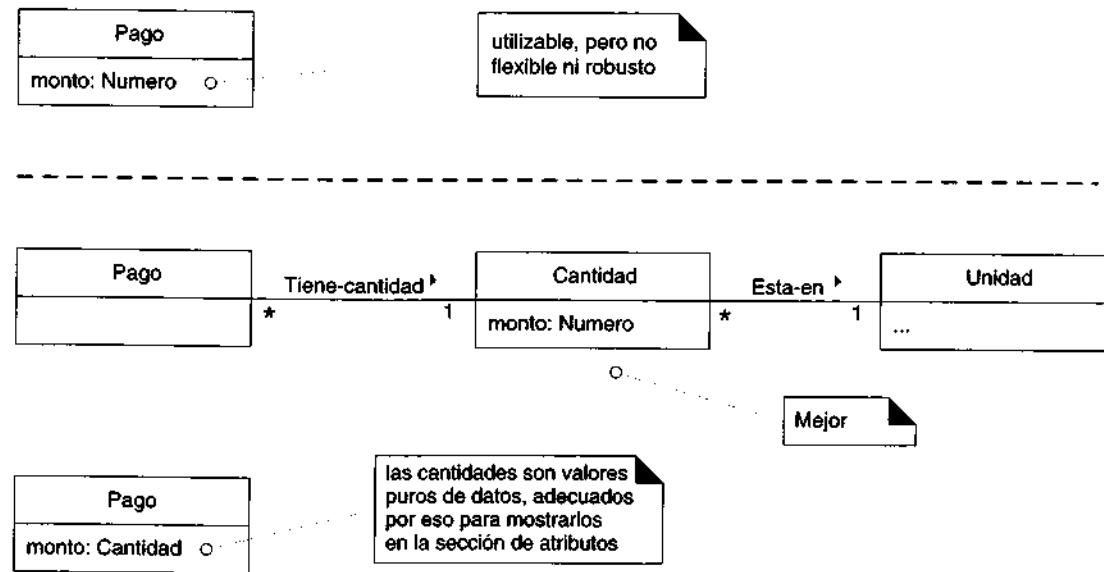


Figura 11.6 Modelado de cantidades.

11.7 Atributos del sistema del punto de venta

Es necesario producir una lista de atributos para los conceptos del dominio del punto de venta. Debería estar reservada primordialmente a los requerimientos y a las simplificaciones en cuestión: los casos simplificados *Comprar Productos: versión 1*.

La lectura llama claramente algunos atributos:

- especificación de los requerimientos
- casos de uso en cuestión
- documentos de simplificaciones, clarificaciones y suposiciones

Por ejemplo, normalmente es necesario registrar la fecha y la hora de una venta. De ahí que el concepto *Venta* requiera los atributos *fecha* y *hora*.

Otros atributos no son evidentes y posiblemente no se los identifique en el análisis. Esto es aceptable: durante las fases de diseño y construcción descubriremos e iremos incorporando el resto de los atributos.

En la siguiente sección se resumen las opciones de atributos indicadas por los casos actuales de uso.

11.8 Atributos en el modelo del punto de venta

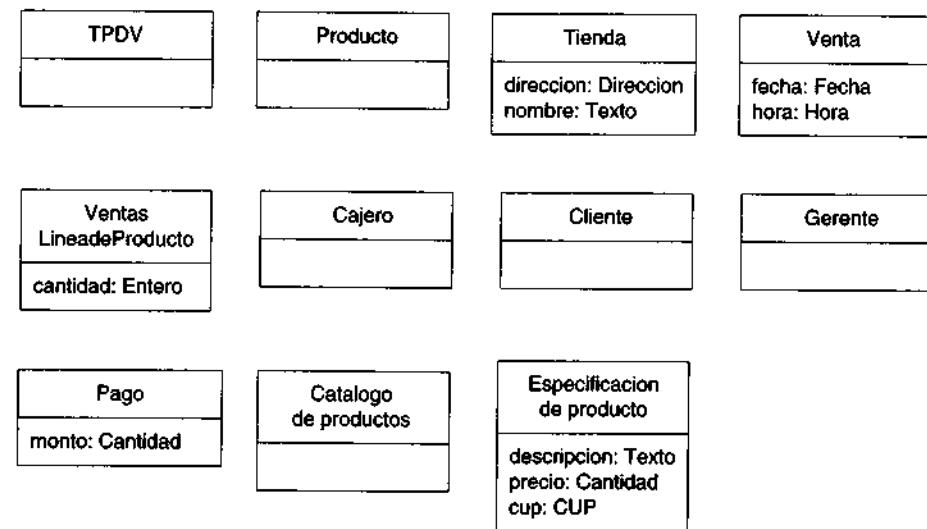


Figura 11.7 Modelo conceptual que muestra los atributos.

11.8.1 Explicación de los atributos TPDV

Pago

importe: hay que capturar un monto (llamado también “importe ofrecido”) para determinar si se dio un pago suficiente y calcular el cambio.

Especifica- ciondeProducto

descripcion: para incluir una descripción en un despliegue.

Venta

CUP: Para consultar EspecificaciondeProducto, una vez capturado un CUP, es necesario relacionarlos con un *CUP*.

precio: para calcular el total de las ventas y mostrar el precio de la línea del producto.

fecha, hora: el recibo es un informe escrito de una venta. Normalmente contiene la fecha y la hora de la venta.

- VentasLineadeProducto* *cantidad*: para registrar la cantidad capturada, cuando hay más de un elemento en la línea del producto (por ejemplo, *cinco* paquetes de pañuelos desechables).
- Tienda* *dirección, nombre*: el recibo requiere el nombre y la dirección de la tienda.

11.9 Multiplicidad entre VentasLineadeProducto y Producto

Es posible que un cajero reciba un grupo de productos afines (seis paquetes de pañuelos desechables, por ejemplo), que introduzca una vez el código universal de producto y luego una cantidad (seis, por ejemplo). En consecuencia, un *VentasLineadeProducto* puede estar asociado a más de una instancia de cada producto. La cantidad que captura el cajero puede quedar registrada como atributo de *VentasLineadeProducto* (figura 11.8). Sin embargo, también puede calcularla a partir del valor real de multiplicidad de la relación; así que puede caracterizarse como un **atributo derivado**, el cual puede ser deducido de otra información. En el lenguaje UML, un atributo de esta clase se denota con el símbolo “/”.

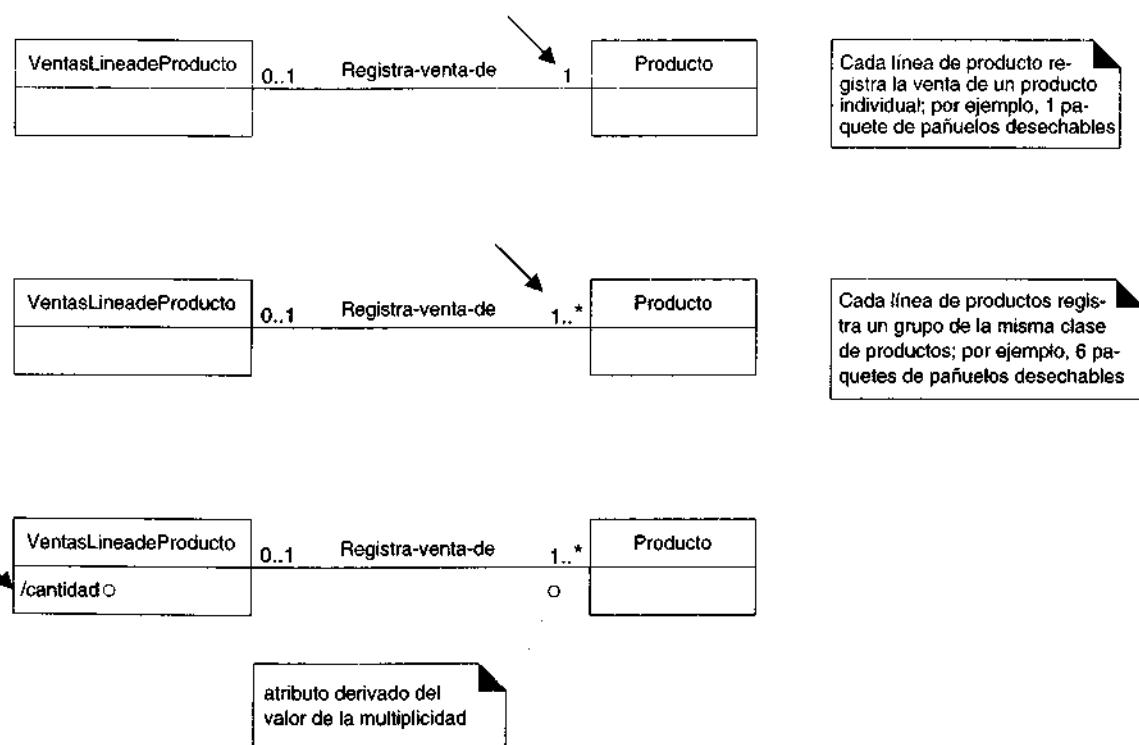


Figura 11.8 Registrar la cantidad de productos vendidos en una línea de producto.

11.10 Modelo conceptual del punto de venta

Al combinar los conceptos, asociaciones y atributos que descubrimos en la investigación anterior, obtenemos el modelo de la figura 11.9.

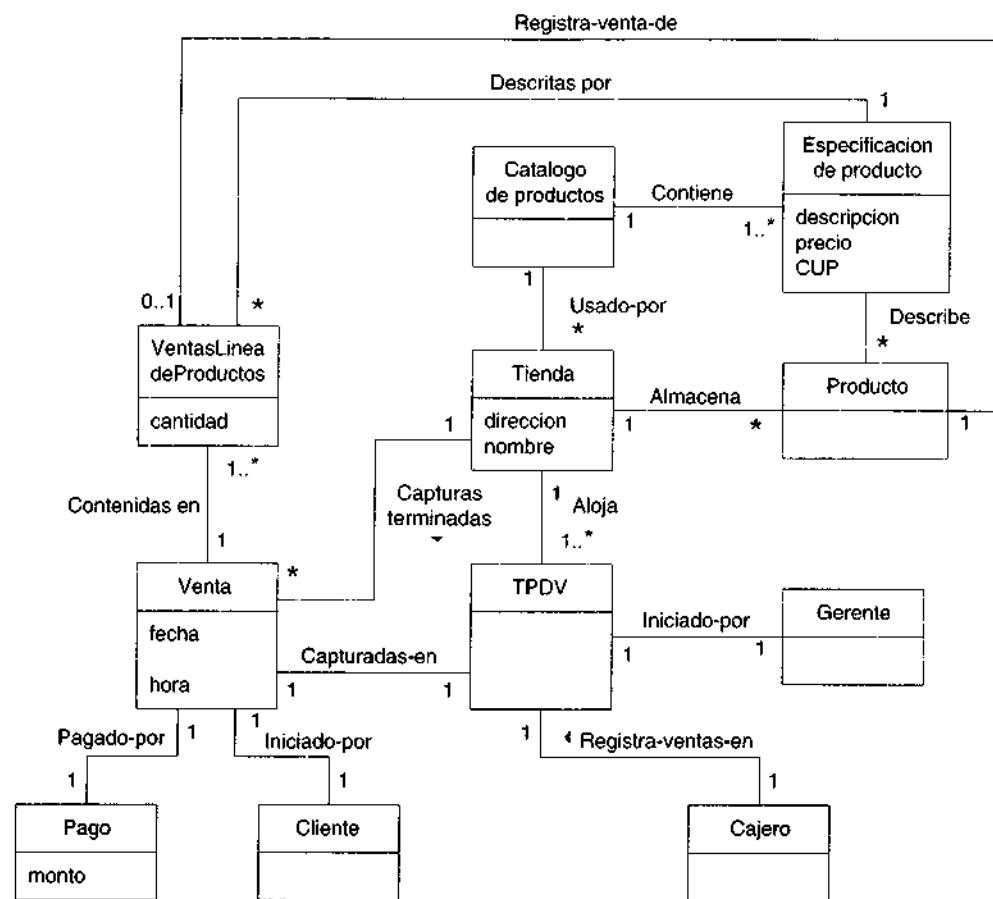


Figura 11.9 Modelo conceptual del dominio del punto de venta.

11.11 Conclusión

Hemos creado un modelo conceptual relativamente útil del dominio de la aplicación al punto de venta. No existe un modelo apropiado para todos los casos o circunstancias. Todos ellos no son más que aproximaciones al dominio que intentamos entender. Un buen modelo conceptual capta las abstracciones esenciales y la información indispensable para comprender el dominio dentro del contexto de los requerimientos actuales; nos facilita además conocer el dominio: sus conceptos, su terminología y sus relaciones.

REGISTRO DE LOS TÉRMINOS EN EL GLOSARIO

Objetivos

- Expresar términos en un glosario.

12.1 Introducción

El glosario es un documento simple en el cual se definen términos. En el presente capítulo ofrecemos un ejemplo de glosario aplicado al dominio del punto de venta.

12.2 Glosario

El **glosario o diccionario modelo** (semejante a un diccionario de datos) incluye y define todos los términos que requieren explicación para mejorar la comunicación y aminorar el riesgo de malos entendidos.

Un significado uniforme y compartido resulta extremadamente importante durante el desarrollo de las aplicaciones, sobre todo cuando muchos miembros del equipo intervienen en el proyecto.

12.3 Actividades y dependencias

El glosario se crea originalmente durante la fase de Planeación y Elaboración conforme vayan generándose los términos; después va perfeccionándose continuamente en cada ciclo de desarrollo al aparecer nuevos vocablos. Por lo regular se realiza junto con la especificación de requerimientos, los casos de uso y el modelo conceptual. Darle mantenimiento es una actividad permanente a lo largo de todo el proyecto.

12.3.1 Reglas y restricciones del dominio

El glosario es un documento muy útil donde se registran las reglas del dominio de la empresa, las restricciones y otros puntos, aunque aquí no examinaremos este aspecto. He aquí otros artefactos en que se registra esta clase de información: los casos de uso, los contratos (que veremos en un capítulo subsecuente) y la incorporación de notas de restricción a los elementos (los conceptos, entre ellos) a los que se aplica la regla o la restricción.

12.4 Ejemplo de glosario aplicado al sistema del punto de venta

No existe un formato oficial de este tipo de glosario. Anexamos aquí una muestra:

Término	Categoría	Comentarios
Comprar productos	caso de uso	Descripción del proceso de un cliente que compra productos en una tienda.
EspecificacioneProducto.descripcion: Texto	atributo	Descripción breve de un producto en una venta, junto con su EspecificacioneProducto asociada.
Producto	tipo	Un producto para venderse en una Tienda.
Pago	tipo	Un pago en efectivo.
EspecificacioneProducto.precio: Cantidad	atributo	El precio de un producto en una venta, junto con su EspecificacioneProducto asociada.
VentasLineadeProducto.cantidad: Entero	atributo	La cantidad comprada de un tipo de Producto.
Venta	tipo	Una transacción de ventas.

Término	Categoría	Comentarios
VentasLineadeProducto	tipo	Una línea de productos de un producto particular comprado en una Venta.
Tienda	tipo	El lugar donde se realiza la venta de productos.
Venta.total: Cantidad	atributo	El gran total de la Venta.
Pago.monto: Cantidad	atributo	El monto que el cliente ofrece o presenta para el pago.
EspecificacioneProducto.cup: CUP	atributo	El código universal de producto del Producto y su EspecificacioneProducto.

COMPORTAMIENTO DE LOS SISTEMAS: DIAGRAMA DE LA SECUENCIA DEL SISTEMA

Objetivos

- Identificar los eventos y las operaciones del sistema.
- Crear el diagrama de la secuencia del sistema para los casos de uso.

13.1 Introducción

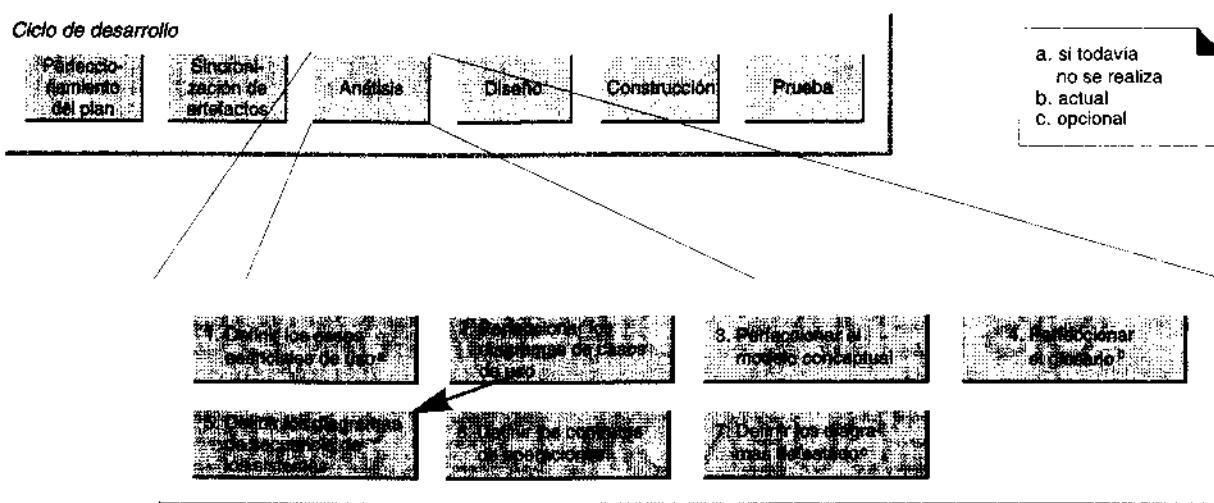
El diagrama de la secuencia de un sistema muestra gráficamente los eventos que fluyen de los actores al sistema. En este capítulo describiremos la forma de elaborarlos.

La creación de los diagramas de la secuencia de un sistema forma parte de la investigación para conocer el sistema; se incluye, pues, dentro del modelo de análisis.

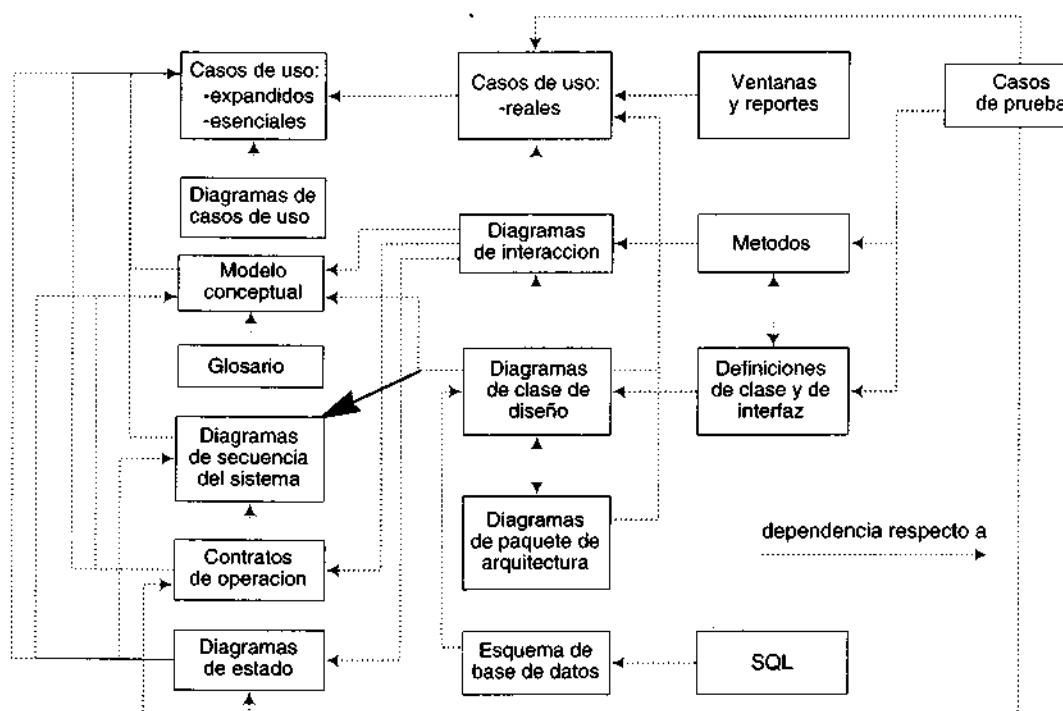
El UML ofrece una notación con los diagramas de la secuencia que muestran gráficamente los eventos que pasan de los actores al sistema.

13.2 Actividades y dependencias

Los diagramas de la secuencia de un sistema se preparan durante la fase de análisis de un ciclo de desarrollo. Su creación depende de la formulación previa de los casos de uso.



Actividades de la fase de análisis dentro de un ciclo de desarrollo.



Dependencias de los artefactos durante la fase de construcción.

13.3 Comportamiento del sistema

Antes de iniciar el diseño lógico de cómo funcionará una aplicación de software, es necesario investigar y definir su comportamiento como una “caja negra”. El **comportamiento del sistema** es una descripción de lo *que* hace, sin explicar la manera en que lo hace. Una parte de la descripción es un diagrama de la secuencia del sistema.

13.4 Diagramas de la secuencia del sistema

Los casos de uso indican cómo los actores interactúan con el sistema de software que es lo que en realidad deseamos crear. Durante la interacción un actor genera eventos dirigidos a un sistema, solicitando alguna operación a cambio. Por ejemplo, cuando un cajero introduce un código universal de producto de un artículo, está pidiendo al sistema TPDV registrar la compra. Con el evento de esa petición se inicia una operación del sistema.

Conviene aislar y explicar gráficamente las operaciones que un actor solicita a un sistema, porque contribuyen de manera importante a entender el comportamiento del sistema. El UML incluye entre su notación los **diagramas de la secuencia** que dan una descripción gráfica de las interacciones del actor y de las operaciones a que da origen.

El **diagrama de la secuencia de un sistema** es una representación que muestra, en determinado escenario de un caso de uso,¹ los eventos generados por actores externos, su orden y los eventos internos del sistema. A todos los sistemas se les trata como una caja negra; los diagramas se centran en los eventos que trasciende las fronteras del sistema y que fluyen de los actores a los sistemas.

El diagrama de la secuencia de un sistema debería prepararse para el curso normal de los eventos de un caso de uso —y posiblemente también de otros—, teniendo en cuenta los cursos opcionales más interesantes.

13.5 Ejemplo de un diagrama de la secuencia de un sistema

El **diagrama de la secuencia de un sistema** describe, en el curso particular de los eventos de un caso de uso, los actores externos que interactúan directamente con el sistema (como caja negra) y con los eventos del sistema generados por los actores (figura 13.1). En el diagrama el tiempo avanza hacia abajo, y el ordenamiento de los eventos debería seguir el orden indicado en el caso de uso.

¹ El escenario de un caso de uso es una instancia o trayectoria realizada por medio del uso: un ejemplo real de su ejecución.

Los eventos del sistema pueden incluir parámetros.

El ejemplo adjunto se refiere al curso normal de los eventos en el caso *Comprar Productos*. Indica que el cajero es el único actor en el sistema TPDV y que genera los eventos del sistema *introducirProducto*, *terminarVenta* y *efectuarPago*.

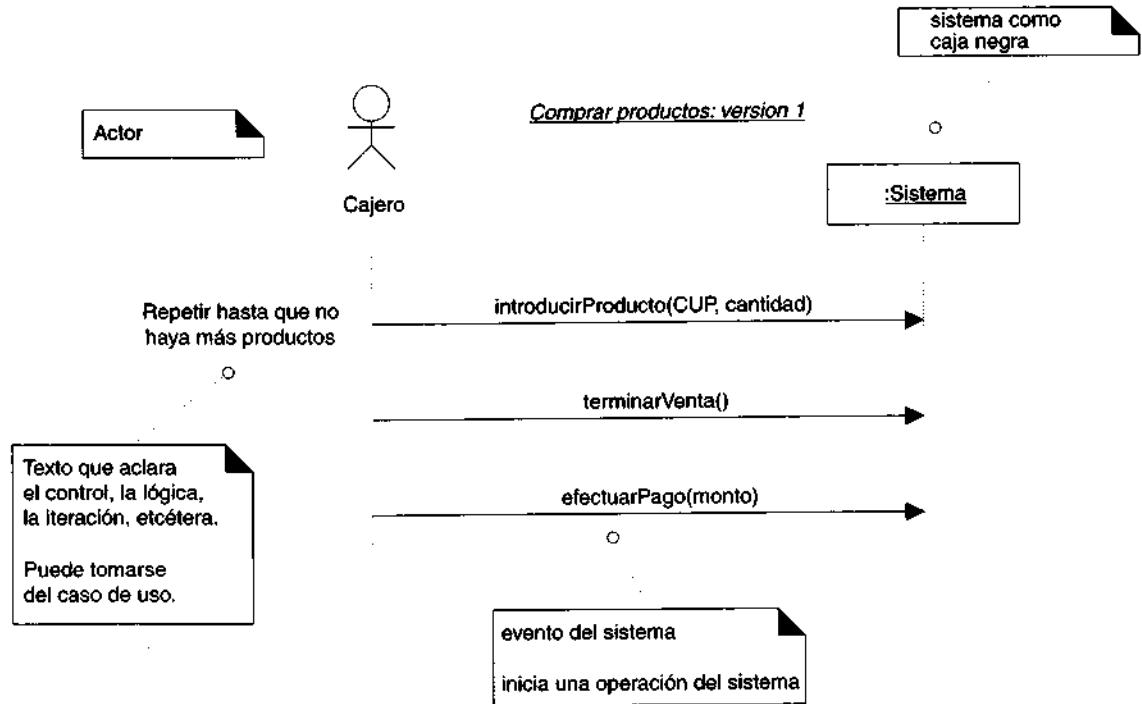


Figura 13.1 Diagrama de la secuencia del sistema para el caso de uso *Comprar productos*.

13.6 Eventos y operaciones de un sistema

El **evento de un sistema** es un hecho externo de entrada que un actor produce en un sistema. El evento da origen a una operación de respuesta. La **operación de un sistema** es una acción que éste ejecuta en respuesta a un evento del sistema (figura 13.2). Por ejemplo, cuando el cajero genera el evento *introducirProducto*, causa la ejecución de la operación *introducirProducto*; el nombre del evento y de la operación son idénticos; la distinción reside en que el evento es el estímulo nombrado y la operación es la respuesta.¹

¹ Lo mismo sucede con los mensajes y métodos.

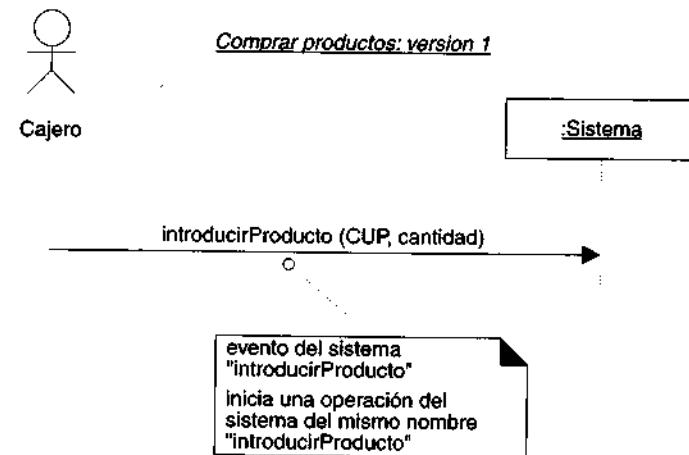


Figura 13.2 Los eventos del sistema inician las operaciones de él.

13.6.1 Registro de las operaciones de un sistema

Para determinar el conjunto de las operaciones requeridas del sistema se identifican sus eventos. Cuando se utilizan parámetros, las operaciones son las siguientes:

- *introducirProducto (CUP, cantidad)*
- *terminarVenta()*
- *efectuarPago (monto)*

¿Dónde deberían registrarse estas operaciones? El lenguaje UML ofrece una notación para registrar las operaciones de un tipo, como se aprecia en la figura 13.3.



Figura 13.3 Notación de las operaciones en el UML.

Con esta notación, las operaciones del sistema pueden agruparse como operaciones del tipo *Sistema* (figura 13.4). Los parámetros son opcionales: pueden incluirse o no.

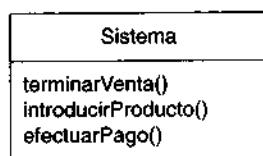


Figura 13.4 Operaciones de un sistema registradas en el tipo *Sistema*.

Este esquema también funciona satisfactoriamente cuando registra las operaciones del sistema de varios sistemas o procesos en una aplicación distribuida; a cada sistema se le asigna un nombre especial (*Sistema1*, *Sistema2*, ...) y también sus operaciones propias.

Observe que la representación del tipo *Sistema* es muy diferente a lo que se expresó en el modelo conceptual. Los elementos de éste representan conceptos del mundo real; en cambio, el tipo *Sistema* es un concepto artificial. Y además muestra las operaciones, cosa totalmente nueva. Ello se debe a que —a diferencia del modelo conceptual, que presenta información estática— estamos describiendo el comportamiento del sistema, que es información dinámica.

13.7 Cómo elaborar un diagrama de la secuencia de un sistema

Para elaborar diagramas de la secuencia de un sistema que describan el curso normal de los eventos en un caso de uso:

1. Trace una línea que represente el sistema como una caja negra.
2. Identifique los actores que operan directamente sobre el sistema. Trace una línea para cada uno de ellos.
3. A partir del curso normal de los eventos del caso de uso identifique los eventos (“externos”) del sistema que son generados por los actores. Muéstrelos gráficamente en el diagrama.
4. A la izquierda del diagrama puede incluir o no el texto del caso de uso.

13.8 Diagramas de la secuencia de un sistema y otros artefactos

La identificación y ordenamiento de los eventos de un sistema para incluirlos en los diagramas de secuencia (figura 13.5) se logran de la revisión de los casos de uso.

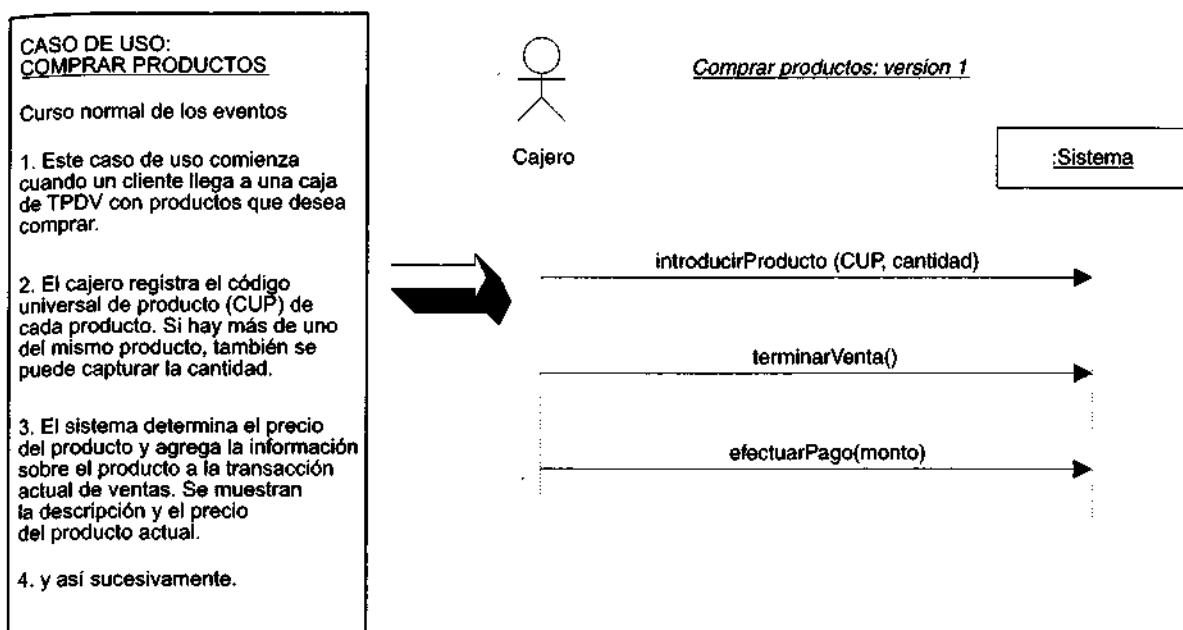


Figura 13.5 Los diagramas de la secuencia de un sistema se dedujeron de los casos de uso.

13.9 Eventos y fronteras de un sistema

Para identificar los eventos de un sistema es necesario tener una idea clara de lo que se quiere al escoger su frontera, según vimos en el capítulo anterior dedicado a los casos de uso. Por lo que respecta al desarrollo del software, la frontera de un sistema suele seleccionarse para que sea el sistema de software (y, posiblemente, del hardware); dentro de este contexto, un evento del sistema es un hecho externo que estimula directamente el software (figura 13.6). En la reingeniería de empresas, la frontera del sistema —y, por tanto, sus eventos— pueden ampliarse para que abarquen los procesos manuales, pero esto rebasa el ámbito de nuestro libro.

Consideremos ahora el caso de uso *Comprar Productos* a fin de identificar los eventos del sistema. Primero debemos determinar los actores que interactúan directamente con el sistema de software. El cliente interactúa con el cajero, pero no directamente con el software TPDV; esto sólo lo hace el cajero. Por tanto, el cliente no es un generador de eventos del sistema; sólo el cajero lo es.

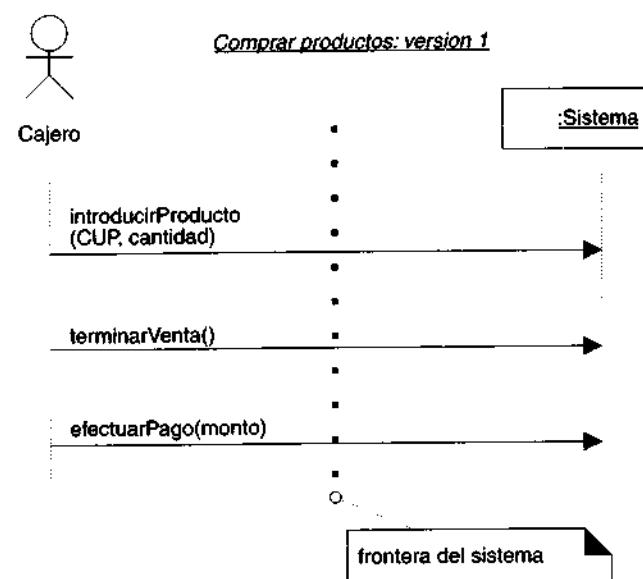


Figura 13.6 Definición de la frontera del sistema.

3.10 Asignación de nombre a los eventos y a las operaciones de un sistema

Los eventos de un sistema (y sus operaciones asociadas) deberían expresarse en el nivel de propósito y no el medio físico de entrada o en el nivel de elementos de la interfaz.

También mejora la claridad si el nombre de un evento del sistema comienza con un verbo (agregar..., introducir..., terminar..., efectuar...), como en la figura 13.7, porque recalca que los eventos están orientados a comandos.

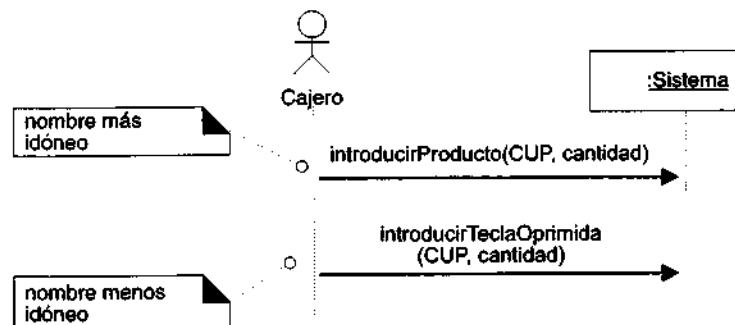


Figura 13.7 Escoja los nombres de los eventos y de las operaciones en un nivel abstracto.

Así, “terminarVenta” es preferible a “introducirTeclaOprimida” porque capta mejor el propósito de la operación: mantiene un carácter abstracto y no se pronuncia respecto a las decisiones de diseño sobre cuál interfaz sirve para capturar el evento del sistema.

En cuanto a expresar las operaciones en el nivel de propósito, procure alcanzar el nivel más alto o la meta final de asignar nombre a la operación. Por ejemplo, respecto a la operación que captura el pago:

<i>introducirImporteOfrecido(monto)</i>	deficiente
<i>introducirPago(monto)</i>	mejor
<i>introducirPago(monto)</i>	quizá mejor aún

13.11 Presentación del texto del caso de uso

En ocasiones conviene mostrar al menos fragmentos del texto del caso de uso dentro del diagrama de la secuencia, con el fin de describir gráficamente su estrecha relación con el caso de uso (figura 13.8).

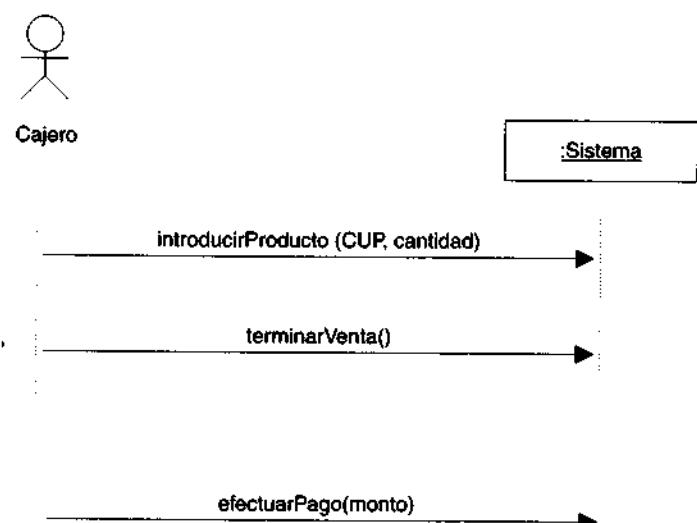


Figura 13.8 Diagrama de la secuencia de un sistema con texto del caso de uso.

13.12 Modelos muestra

Los diagramas de la secuencia de un sistema forman parte del modelo del comportamiento del sistema, como se advierte en la figura 13.9, la cual especifica a qué eventos responde un sistema y qué responsabilidades y poscondiciones tiene sus operaciones correspondientes.

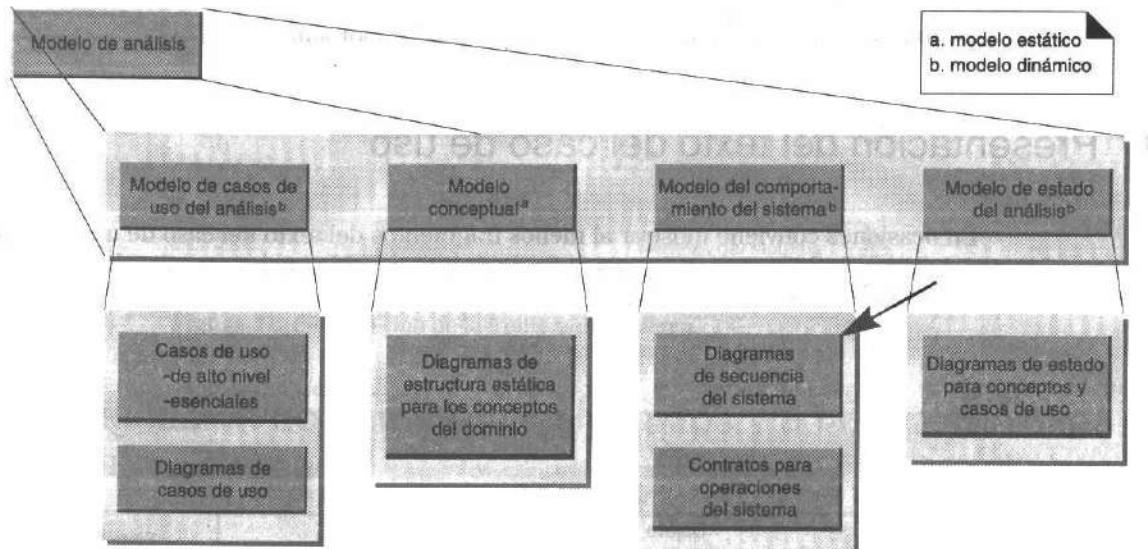


Figura 13.9 Modelo del análisis.

COMPORTAMIENTO DE LOS SISTEMAS: CONTRATOS

Objetivos

- Crear contratos para las operaciones de un sistema.

14.1 Introducción

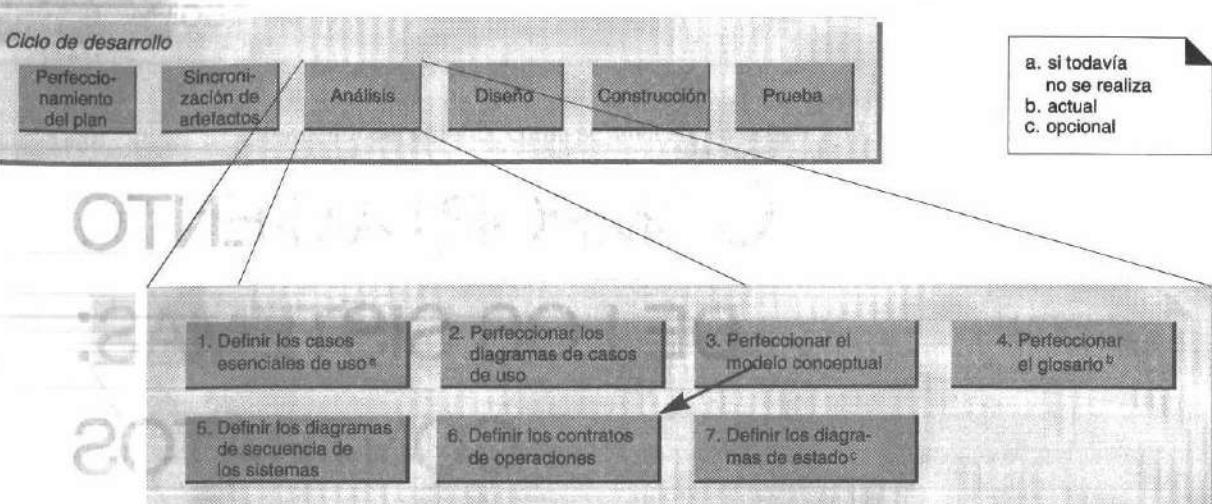
Los contratos contribuyen a definir el comportamiento de un sistema; describen el efecto que sobre él tienen las operaciones. En este capítulo estudiaremos su utilización.

El lenguaje UML ofrece un soporte para definir los contratos, ya que permite definir las precondiciones y las poscondiciones de las operaciones.¹

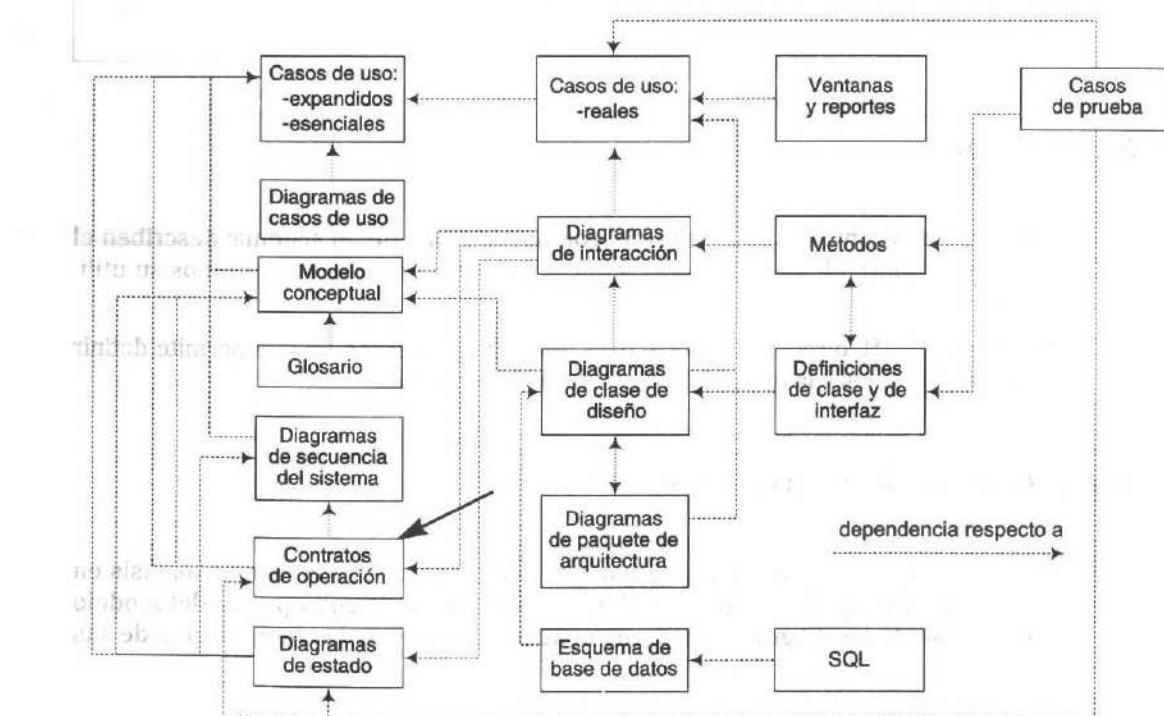
14.2 Actividades y dependencias

Los contratos de operación del sistema se elaboran durante la fase de análisis en un ciclo de desarrollo. Su preparación depende del desarrollo previo del modelo conceptual, de los diagramas de la secuencia del sistema y la identificación de sus operaciones.

¹ En la definición formal del UML —o metamodelo—, las operaciones tienen un conjunto de propiedades predefinidas que incluye sus precondiciones y poscondiciones.



Actividades de la fase de análisis dentro de un ciclo de desarrollo.



Dependencias de los artefactos durante la fase de construcción.

14.3 Comportamiento de un sistema

Antes de emprender el diseño lógico de cómo funcionará una aplicación de software, es necesario investigar y definir su comportamiento como “caja negra”. El **comportamiento de un sistema** es una descripción de *lo que* hace, sin explicar cómo lo hace. Los contratos son documentos muy útiles que describen el comportamiento de un sistema a partir de cómo cambia el estado de un sistema cuando se llama una operación suya.

14.4 Contratos

En la figura 14.1, el diagrama de la secuencia de un sistema muestra los eventos generados por un actor externo, pero no profundiza en los detalles de la funcionalidad asociada con las operaciones invocadas. No contiene los detalles necesarios para entender la respuesta del sistema, o sea su comportamiento.

En términos generales, un **contrato**¹ es un documento que describe lo que una operación se propone lograr. Suele redactarse en un estilo declarativo, enfatizando *lo que* sucederá y no *cómo* se conseguirá. Los contratos suelen expresarse a partir de los cambios de estado de las precondiciones y de las poscondiciones. Puede elaborarse un contrato para un método de una clase de software o para una operación más global del sistema.

El **contrato de operación del sistema** describe los cambios del estado del sistema total cuando se llama una de sus operaciones.



Figura 14.1 Las operaciones del sistema requieren las descripciones del contrato.

Repetimos: un contrato puede emplearse con una operación de alto nivel que se aplica al sistema entero o con un método de bajo nivel de una clase particular. Por ahora nos centraremos en su uso en las operaciones del sistema.

14.5 Ejemplo de contrato: introducirProducto

En el siguiente ejemplo se describe un contrato de la operación *introducirProducto* del sistema.

¹ Bertrand Meyer fue el primero en difundir el término.

Contrato	
Nombre:	introducirProducto (cup: número, cantidad: entero).
Responsabilidades:	Capturar (registrar) la venta de un producto y agregarla a la venta. Desplegar la descripción y el precio del producto.
Tipo:	Sistema.
Referencias cruzadas:	Funciones del sistema: R1.1, R1.3, R1.9. Casos de uso: Comprar productos.
Notas:	Utilizar el acceso superrápido a la base de datos.
Excepciones:	Si el CUP no es válido, indicar que se cometió un error.
Salida:	
Precondiciones:	El sistema conoce el CUP.
Poscondiciones:	

- Si se trata de una nueva venta, se crea una *Venta* (*creación de instancia*).
- Si se trata de una nueva venta, la nueva *Venta* fue asociada a *TPDV* (*asociación formada*).
- Se creó una instancia *VentasLineadeProducto* (*creación de instancia*).
- Se asoció una instancia *VentasLineadeProducto* a la *Venta* (*asociación formada*).
- Se asignó cantidad a *VentasLineadeProducto.cantidad* (*modificación de atributo*).
- Se asoció una instancia *VentasLineadeProducto* a la instancia *EspecificaciondeProducto*, basado esto en la correspondencia del CUP (*asociación formada*).

14.6 Secciones del contrato

Una descripción de las secciones de un contrato se da en el siguiente esquema. No todas las secciones son necesarias; nosotros recomendamos las de *Responsabilidades* y *Poscondiciones*.

Contrato	
Nombre:	Nombre de la operación y parámetros.
Responsabilidades:	Descripción informal de las responsabilidades que debe cumplir la operación.
Tipo:	Nombre del tipo (concepto, clase de software, interfaz).

Referencias cruzadas:	Números de referencia de las funciones del sistema, casos de uso, etcétera.
Notas:	Notas de diseño, algoritmos e información afín.
Excepciones:	Casos excepcionales.
Salida:	No salidas de la Interfaz del Usuario; por ejemplo, mensajes o registros que se envían fuera del sistema.
Precondiciones:	Suposiciones acerca del estado del sistema antes de ejecutar la operación.
Poscondiciones:	
	<ul style="list-style-type: none"> ■ El estado del sistema después de la operación. Esto se explica a fondo en la siguiente sección.

14.7 Cómo preparar un contrato

Aplique la siguiente sugerencia para elaborar contratos.

Para preparar un contrato en los casos de uso:
1. Identifique las operaciones del sistema a partir de los diagramas de su secuencia.
2. Elabore un contrato en cada operación del sistema.
3. Comience redactando la sección de <i>Responsabilidades</i> ; después describa informalmente el propósito de la operación.
4. Complete luego la sección de <i>Poscondiciones</i> , describiendo en forma declarativa los cambios de estado de los objetos en el modelo conceptual.
5. Para describir las poscondiciones utilice las siguientes categorías:
<ul style="list-style-type: none"> <input type="checkbox"/> Creación y eliminación de las instancias. <input type="checkbox"/> Modificación de los atributos. <input type="checkbox"/> Asociaciones formadas y canceladas.

14.7.1 Contratos y otros artefactos

- Los casos de uso sugieren los diagramas de los eventos y de la secuencia del sistema.
- Después se identifican las operaciones del sistema.

- El efecto de las operaciones del sistema se describe en los contratos.

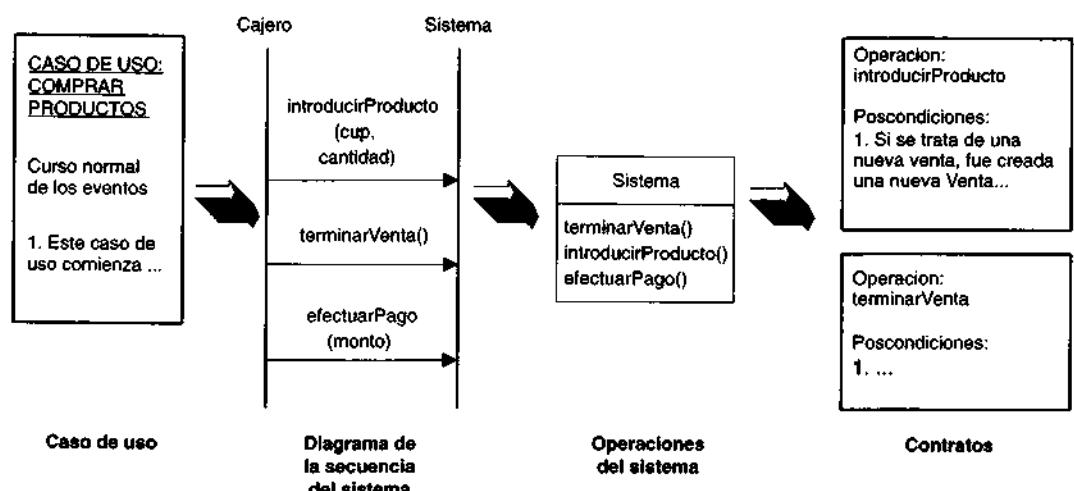


Figura 14.2 Contratos y otros artefactos.

14.8 Poscondiciones

Nótese que las poscondiciones en el ejemplo *introducirProducto* incluyen una clasificación; por ejemplo, *creación de instancias* o *asociación formada*. Después de la sección de *Responsabilidades*, la parte más importante del contrato son las poscondiciones, que estipulan cómo cambió el sistema tras esta operación. No son acciones que deben efectuarse durante la operación; más bien son declaraciones sobre el estado del sistema que se aplican una vez concluida la operación, es decir, *una vez que el humo se ha disipado*.

El UML no impone límites a la manera de expresar las poscondiciones; pero las siguientes categorías de cambio de estado han resultado de gran utilidad en la práctica. Le aconsejamos expresar sus poscondiciones de una manera similar.

Categorías útiles referentes a las poscondiciones del contrato:

- Creación y eliminación de las instancias.
- Modificación de los atributos.
- Asociaciones formadas y canceladas.

El UML no define cómo expresar las poscondiciones; así que puede usted escoger el formato que más le agrade. Lo importante es que adopte una actitud declarativa, orientada al cambio de estado y no a la acción, porque las poscondiciones deberían ser declaraciones sobre los estados o resultados, no una descripción de acciones a realizar.

14.8.1 Las poscondiciones se relacionan con el modelo conceptual

Las poscondiciones se expresan dentro del contexto del modelo conceptual. ¿Qué instancias es posible crear? La respuesta es: las provenientes de ese modelo. ¿Qué asociaciones es posible formar? La respuesta es: las que están en el modelo conceptual. Y así podríamos ir formulando y contestando más preguntas.

Cuando se elaboran contratos, generalmente el lector se percatará de la necesidad de registrar en el modelo conceptual nuevos conceptos, atributos o asociaciones. No se limite usted a la definición precedente del modelo conceptual: mejórelo conforme vaya haciendo más descubrimientos, mientras reflexiona sobre los contratos de las operaciones.

14.8.2 La ventaja de las poscondiciones

Expresado en una forma declarativa de cambios de estado, el contrato constituye una excelente herramienta de investigación, pues permite describir los cambios necesarios para que el sistema funcione sin necesidad de describir *cómo* se logran. En otras palabras, podemos posponer el diseño y la solución del software y concentrarnos analíticamente en *lo que* debe suceder, no en la manera de conseguirlo.

14.9 El espíritu de las poscondiciones: el escenario y el telón

Las poscondiciones deberían describir el estado de un sistema, no las acciones a realizar. Expréselas en tiempo pasado para enfatizar que se trata de declaraciones sobre un cambio pretérito de estado. Por ejemplo:

- Se creó una instancia *VentasLineadeProducto* (mejor); en lugar de
- Crear una instancia *VentasLineadeProducto* (peor).

Reflexione sobre las poscondiciones sirviéndose de la siguiente imagen:

El sistema y sus objetos se presentan en el escenario de un teatro.

1. Tome una fotografía del escenario antes de la operación.
2. Corra el telón del escenario y aplique la operación del sistema (*ruido de fondo con sonidos metálicos, gritos, chillidos...*).
3. Corra el telón y tome una segunda fotografía.

4. Compare las fotografías de antes y después, y exprese como poscondiciones los cambios del estado del escenario (*Se creó la instancia VentasLineadeProducto...*).

14.10 Explicación: poscondiciones de *introducirProducto*

En la siguiente sección analizaremos por qué se utilizan las poscondiciones de la operación del sistema de *introducirProducto*.

14.10.1 Creación y eliminación de instancias

Una vez que el cajero capturó el código universal de producto (CUP) y la cantidad de un producto, ¿qué nuevos objetos han de crearse? Si se trata de una nueva venta, habría que crear una instancia para una nueva *Venta*. Una instancia *VentasLineadeProducto* debería ser creada de modo incondicional. Por tanto:

- Si se trata de una nueva venta, se creó una *Venta* (creación de instancia).
- Se creó una instancia *VentasLineadeProducto* (creación de instancia).

14.10.2 Modificación de atributos

Una vez que la cajera capturó el código universal de producto y la cantidad de un producto, ¿qué atributos de los objetos nuevos o actuales deberían ser modificados? Habría que establecer la cantidad de *VentasLineadeProducto*. Por tanto:

- Se definió para *VentasLineadeProducto.cantidad* el valor de *cantidad* (modificación del atributo).

14.10.3 Asociaciones formadas y canceladas

Una vez que el cajero capturó el código universal de producto y la cantidad de un producto, ¿qué asociaciones entre los objetos nuevos y los actuales debieron haber sido formadas o canceladas? Habría que haber relacionado la nueva instancia *VentasLineadeProducto* con sus *Ventas* y con su *Producto*. Si se trataba de una nueva venta, la *Venta* debió haber sido relacionada con la *TPDV* dentro de la cual es registrada. Por tanto:

- Si se trata de una venta nueva, la nueva *Venta* fue asociada a la *TPDV* (asociación formada).
- Una instancia *VentasLineadeProducto* fue asociada a la *Venta* (asociación formada).
- Una instancia *VentasLineadeProducto* fue asociada a una *EspecificaciondeProducto*, con base en la correspondencia del CUP (asociación formada).

14.11 ¿Cuán completas deben ser las poscondiciones?

En la fase de análisis no es probable —ni siquiera necesario— generar un grupo de poscondiciones completas y exactas de la operación del sistema. Aconsejamos ver en su elaboración la conjectura inicial más acertada, a sabiendas de que los contratos estarán incompletos. Esta creación temprana —aunque incompleta— sin duda es preferible a posponer la investigación hasta la fase de diseño, cuando los creadores se concentrarán en el diseño de una solución más que en averiguar *lo que* debe hacerse.

Algunos de los detalles finos —y quizá hasta los más generales— se descubrirán durante la fase de diseño. No es algo necesariamente malo; en la fase de investigación se debilitan el esfuerzo y la dedicación si se prolongan demasiado tiempo. En la fase de diseño se logran algunos descubrimientos que después pueden guiar la fase de investigación de un ciclo iterativo posterior. Una de las ventajas del desarrollo iterativo es ésta: los descubrimientos hechos en la fase de diseño de un ciclo pueden mejorar la calidad de la investigación y el trabajo de análisis en el siguiente ciclo.

14.12 Descripción de los detalles y algoritmos del diseño: notas

La sección del contrato correspondiente a Notas es el lugar donde pueden hacerse las declaraciones del diseño referentes a la operación. Por ejemplo, si se sabe que se prefiere un algoritmo en particular para manejar la operación, esa sección es el sitio idóneo para su documentación.

14.13 Precondiciones

Las precondiciones definen las suposiciones sobre el estado del sistema al iniciarse la operación. Hay muchas precondiciones que pueden declararse en una operación, pero la experiencia revela que vale la pena mencionar las siguientes:

- Cosas que son importantes probar en el software en algún momento de la ejecución de la operación.
- Cosas que no serán sometidas a prueba, pero de las cuales depende el éxito de la operación. Queremos compartir esta suposición con los futuros lectores del contrato, a fin de subrayar su importancia y de que los lectores se percaten de ella.

14.14 Recomendación sobre cómo redactar contratos

- Una vez anotado el nombre de la operación, llene primero la sección de *Responsabilidades* y luego la de *Poscondiciones*, dejando al final la de *Precondiciones*. Son las tres secciones más importantes del proyecto en cuanto a su uso ulterior. Desde luego, si a un desarrollador no le resulta útil llenar una sección, no lo hará y nada pasará.
- Use la sección de *Notas* para explicar los detalles del diseño; por ejemplo, los algoritmos y los pasos secuenciales de alto nivel.
- Use la sección de *Excepciones* para explicar la reacción ante situaciones raras o especiales.
- Use las siguientes categorías de los cambios de estado en las poscondiciones:
 - Creación y eliminación de instancias.
 - Modificación de atributos.
 - Asociaciones formadas y canceladas.
- Exprese las poscondiciones en forma pasiva declarativa, en pretérito (*fue registrado...*) para destacar la declaración de un cambio de estado en vez del diseño de cómo iba a obtenerse. Por ejemplo:

Fue creada una instancia *VentasLineadeProducto* (bien).

es mejor que

Se creó una instancia *VentasLineadeProducto* (mal).

- La diferencia entre ambas oraciones puede parecer meramente académica y superficial; pero si se emplea la construcción activa en vez de la pasiva, sabemos por experiencia que los desarrolladores rápidamente adoptan la actitud de diseñar la forma en que van a resolver la operación del software. El espíritu del contrato es poner de relieve una declaración de los cambios de estado, absteniéndose de ofrecer los medios de una solución.
- No olvide establecer una memoria entre los objetos actuales y los de creación reciente, definiendo para ello la formación de una asociación. Por ejemplo, si no es suficiente que una nueva instancia *VentasLineadeProducto* se genere cuando ocurre la operación *IntroducirProducto*. Una vez finalizada la operación, deberá ser verdad que la instancia recién creada fue asociada a *Venta*; por tanto,
 - La instancia *VentasLineadeProducto* fue asociada a la *Venta* (asociación formada).

14.14.1 El error más frecuente en la preparación de contratos

El problema más común consiste en olvidar incluir la *formación de asociaciones*. Sobre todo cuando se crean nuevas instancias, muy probablemente será necesario haber establecido las asociaciones a varios objetos. No lo olvide.

14.15 Contratos para el caso de uso *Comprar productos*

14.15.1 Contrato para *introducirProducto*

	Contrato
Nombre:	introducirProducto
	(cup: número, cantidad: entero).
Responsabilidades:	Introducir (registrar) la venta de un producto y agregarlo a la venta. Desplegar la descripción del producto y su precio.
Tipo:	Sistema.
Referencias cruzadas:	Funciones del sistema: R1.1, R1.3, R1.9.
Notas:	Casos de uso: Comprar productos.
Excepciones:	Utilice el acceso superrápido a la base de datos.
Salida:	Si el CUP no es válido, indique que se cometió un error.
Precondiciones:	El sistema conoce el CUP.
Poscondiciones:	

- Si se trata de una nueva venta, una *Venta* fue creada (*creación de instancia*).
- Si se trata de una nueva venta, la nueva *Venta* fue asociada a la *TPDV* (*asociación formada* o *formación de asociaciones*).
- Se creó una instancia *VentasLineadeProducto* (*creación de instancia*).
- Se asoció *VentasLineadeProducto* a la *Venta* (*asociación formada*).
- Se estableció *VentasLineadeProducto.cantidad* con el valor de *cantidad* (*modificación de atributo*).
- La instancia *VentasLineadeProducto* fue asociada a una *EspecificaciondeProducto*, basado esto en la correspondencia del código universal de producto (*asociación formada*).

14.15.2 Contrato para terminarVenta

Contrato	
Nombre:	terminarVenta().
Responsabilidades:	Registrar que es el final de la captura de los productos de la venta y desplegar el total de la venta.
Tipo:	Sistema.
Referencias cruzadas:	Funciones del sistema: R1.2. Casos de uso: Comprar productos.
Notas:	
Excepciones:	Si no está realizándose una venta, indicar que se cometió un error.
Salida:	
Precondiciones:	El sistema conoce el CUP.
Poscondiciones:	

- Estableció *Venta.estaTerminada* en *verdadero* (modificación de atributo).

14.15.3 Contrato para efectuarPago

Contrato	
Nombre:	efectuarPago (monto: Número o Cantidad).
Responsabilidades:	Registrar el pago, calcular el saldo e imprimir el recibo.
Tipo:	Sistema.
Referencias cruzadas:	Funciones del sistema: R2.1. Casos de uso: Comprar productos.
Notas:	
Excepciones:	Si la venta no está concluida, indicar que se cometió un error. Si el monto es menor que la venta total, indicar que se cometió un error.
Salida:	
Precondiciones:	
Poscondiciones:	

- Se creó un *Pago* (creación de instancia).
- Se asignó a *Pago.montoOfrecido* el valor de *monto* (modificación de atributo).
- Se asoció el *Pago* a la *Venta* (relación formada).
- Se asoció la *Venta* a la *Tienda* para agregarla al registro histórico de las ventas terminadas (relación formada).

14.16 Contratos para el caso de uso *Inicio*

14.16.1 Contrato para Inicio

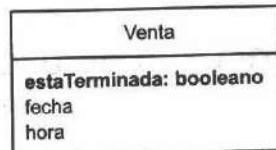
Contrato	
Nombre:	inicio().
Responsabilidades:	Inicializar el sistema.
Tipo:	Sistema.
Referencias cruzadas:	
Notas:	
Excepciones:	
Salida:	
Precondiciones:	
Poscondiciones:	

- Se creó una instancia *Tienda*, *TPDV*, *CatalogodeProductos* y *EspecificacionesdeProducto* (creación de instancia).
- Se asoció *CatalogodeProductos* a *EspecificacionesdeProducto* (asociación formada).
- Se asoció *Tienda* a *CatalogodeProductos* (asociación formada).
- Se asoció *Tienda* a *TPDV* (asociación formada).
- Se asoció *TPDV* a *CatalogodeProductos* (asociación formada).

14.17 Cambios del modelo conceptual

Estos contratos sugieren la existencia de un dato que todavía no ha figurado en el modelo conceptual: la terminación de la captura del producto en la venta. Lo modifica la especificación *terminarVenta*, y la especificación *efectuarPago* lo prueba como precondition.

Una forma de representar esta información es expresarla como un atributo *estaTerminada* (o *capturaEstaTerminada*) de la venta, por medio de un valor booleano:



Se cuenta con otras soluciones alternas para representar el estado cambiante del sistema. Un método es el **patrón de estado**, que estudiaremos en un capítulo subsiguiente.

14.18 Modelos muestra

Los contratos que rigen las operaciones del sistema forman parte del modelo del comportamiento del sistema, el cual describe la interfaz externa y el comportamiento de todo el sistema (figura 14.3).

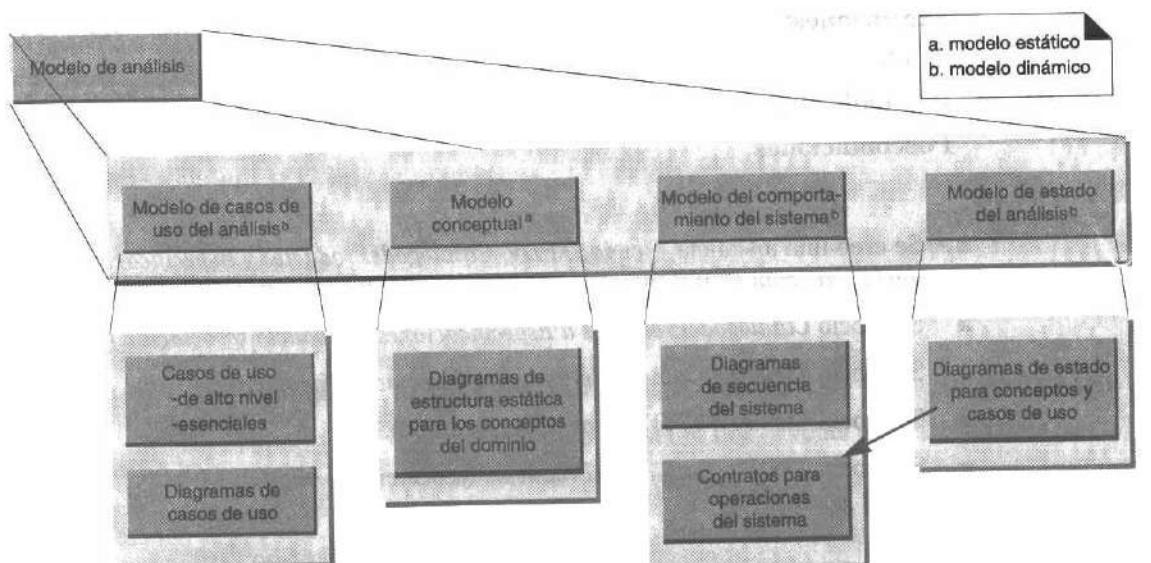


Figura 14.3 Modelo de análisis.

PARTE IV

FASE DEL DISEÑO (1)

DEL ANÁLISIS AL DISEÑO

Objetivos

- Motivar la transición del análisis al diseño.

15.1 Conclusión de la fase de análisis

En la fase de análisis del desarrollo se da prioridad al conocimiento de los requisitos, los conceptos y las operaciones relacionadas con el sistema. A menudo la investigación y el análisis se caracterizan por centrarse en cuestiones concernientes al *qué*: cuáles son los procesos, los conceptos, etcétera.

En el UML hay otros artefactos que sirven para capturar los resultados de una investigación; a continuación se describe un grupo mínimo de ellos (que ya estudiamos en capítulos anteriores):

Artefacto de análisis	Preguntas que se contestan
Casos de uso	¿Cuáles son los procesos del dominio?
Modelo conceptual	¿Cuáles son los conceptos, los términos?
Diagramas de la secuencia de un sistema	¿Cuáles son los eventos y las operaciones del sistema?
Contratos	¿Qué hacen las operaciones del sistema?

15.2 Inicio de la fase de diseño

Durante el ciclo de desarrollo iterativo es posible pasar a la fase de diseño, una vez terminados estos documentos del análisis. Durante este paso se logra una solución lógica que se funda en el paradigma orientado a objetos. Su esencia es la elaboración de **diagramas de interacción**, que muestran gráficamente cómo los objetos se comunicarán entre ellos a fin de cumplir con los requerimientos.

El advenimiento de los diagramas de interacción nos permite dibujar **diagramas de diseño de clases** que resumen la definición de las clases (e interfaces) implementables en software.

En los siguientes capítulos examinaremos la creación de estos artefactos. Los diagramas de interacción son los más importantes de ellos (desde el punto de vista de la preparación de un buen diseño) y exigen gran dedicación y esfuerzo creativos. Para prepararlos hay que aplicar los principios de la asignación de **responsabilidades** y utilizar los **patrones de diseño**. Por tanto, en los siguientes capítulos nos centraremos en esos principios y patrones del diseño orientado a objetos.

La creación de los diagramas de interacción exige conocer:

- Los principios de la asignación de responsabilidades.
- Los patrones del diseño.

DESCRIPCIÓN DE LOS CASOS REALES DE USO

Objetivos

- Crear casos reales de uso.

UNIVERSIDAD DE LA REPÚBLICA
FACULTAD DE INGENIERÍA
DEPARTAMENTO DE
DOCUMENTACIÓN Y BIBLIOTECA
MONTEVIDEO - URUGUAY

16.1 Introducción

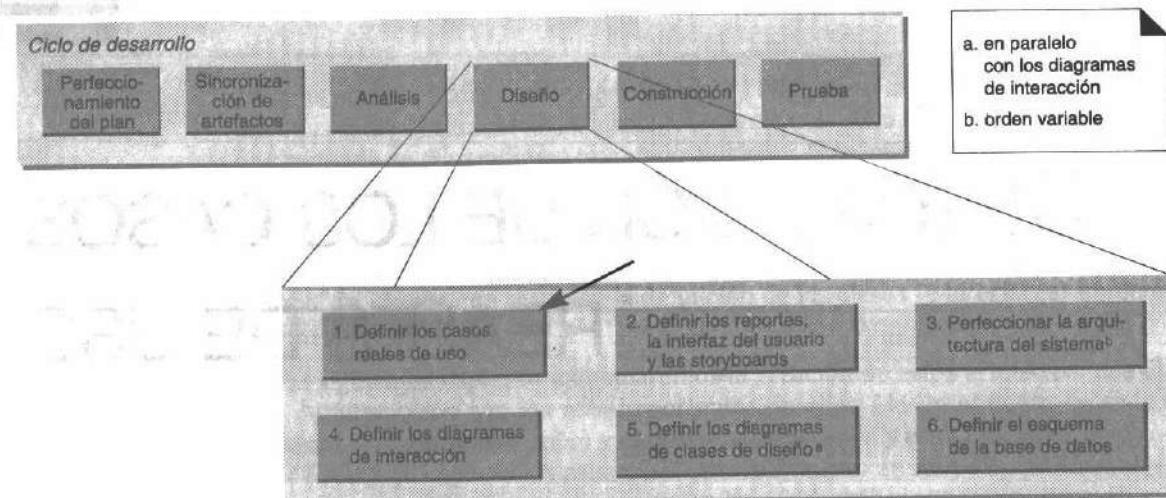
Los casos reales de uso presentan un diseño concreto de cómo se realizará el caso. En el presente capítulo vamos a examinar su creación.

16.2 Actividades y dependencias

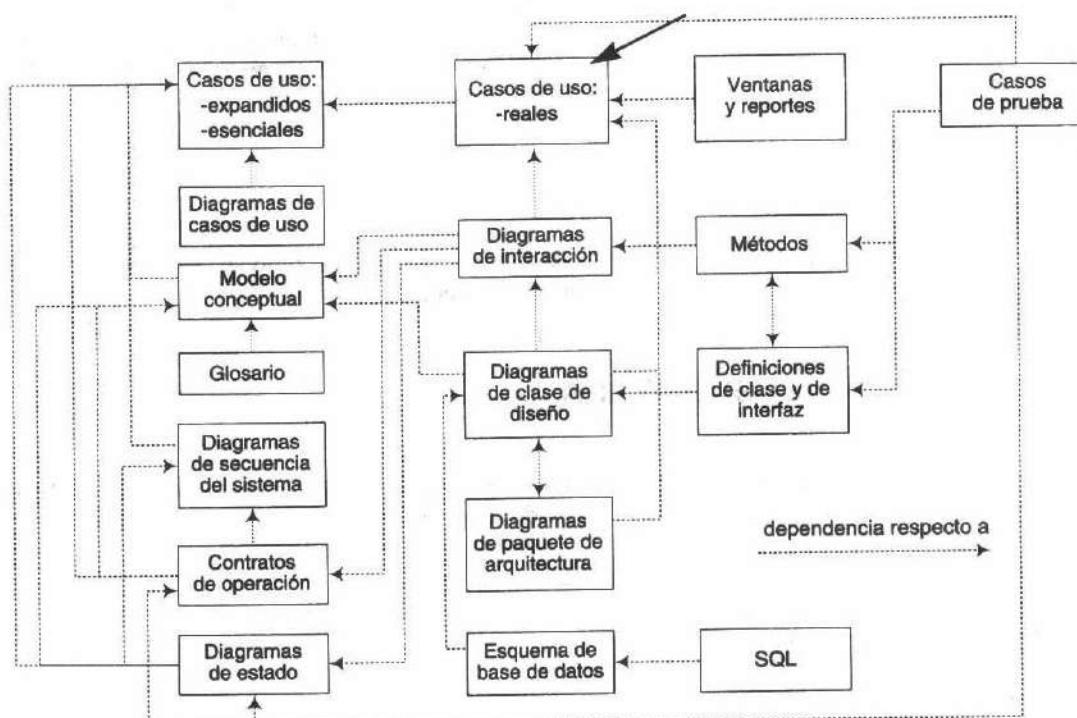
La definición de los casos de uso reales es una de las primeras actividades dentro de un ciclo de desarrollo. Su creación depende de los casos esenciales conexos que hayan sido generados antes.

16.3 Casos reales de uso

Un **caso real de uso** describe el diseño concreto del caso de uso a partir de una tecnología particular de entrada y salida, así como de su implementación global. Por ejemplo, si interviene una interfaz gráfica para el usuario, el caso de uso real incluirá



Actividades de la fase de diseño dentro de un ciclo de desarrollo.



Dependencias de los artefactos durante la fase de construcción.

diagramas de las ventanas en cuestión y una explicación de la interacción de bajo nivel con los artefactos de la interfaz.

Tal vez no sea necesario generarlo. Una alternativa podría consistir en que el diseñador realizara storyboards o secuencias de las pantallas de la interfaz general para el usuario y que después fuera incorporando los detalles durante la implementación.

Los storyboards son de gran utilidad, si antes de la implementación los diseñadores o el cliente necesitan descripciones rigurosamente detalladas.

16.4 Ejemplo: Comprar productos: versión 1

En el siguiente ejemplo y en la figura 16.1, observe la utilización de un esquema de codificación con los artefactos de ventana para obtener una descripción fluida.

Casos de uso:	Comprar productos: versión 1 (efectivo exclusivamente)
Actores:	Cliente (iniciador), Cajero.
Propósito:	Capturar una venta y su pago en efectivo.
Resumen:	Un Cliente llega a la caja con productos que desea comprar. El Cajero registra los productos de la compra y recibe el pago en efectivo. Al terminar la transacción, el Cliente se marcha con los productos comprados.
Tipo:	Primario y real.
Referencias cruzadas:	Funciones: R1.1, R1.2, R1.3, R1.7, R1.9, R2.1.

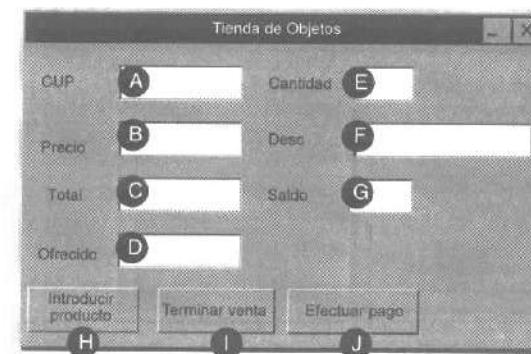


Figura 16.1 Ventana-1.

Curso normal de los eventos	
Acción de los actores	Respuesta del sistema
1. Este caso comienza cuando un Cliente llega a la caja TPDV con objetos que desea comprar.	
2. Con cada producto, el Cajero teclea el código universal de producto (CUP) en A de la Ventana-1. Si hay más de un producto, es opcional capturar la cantidad en E. Se oprime H después de capturar cada producto.	3. Agrega la información sobre el producto a la actual transacción de ventas. La descripción y el precio del producto actual se muestran en B y en F de la Ventana-1.
4. Al terminar de capturar los productos, el Cajero oprime el botón I para indicarle a la TPDV que terminó de capturar los productos.	5. Calcula y presenta en C el total de la venta. 6. ...

16.5 Modelos muestra

Los casos reales de uso son miembros del modelo de caso de uso de diseño

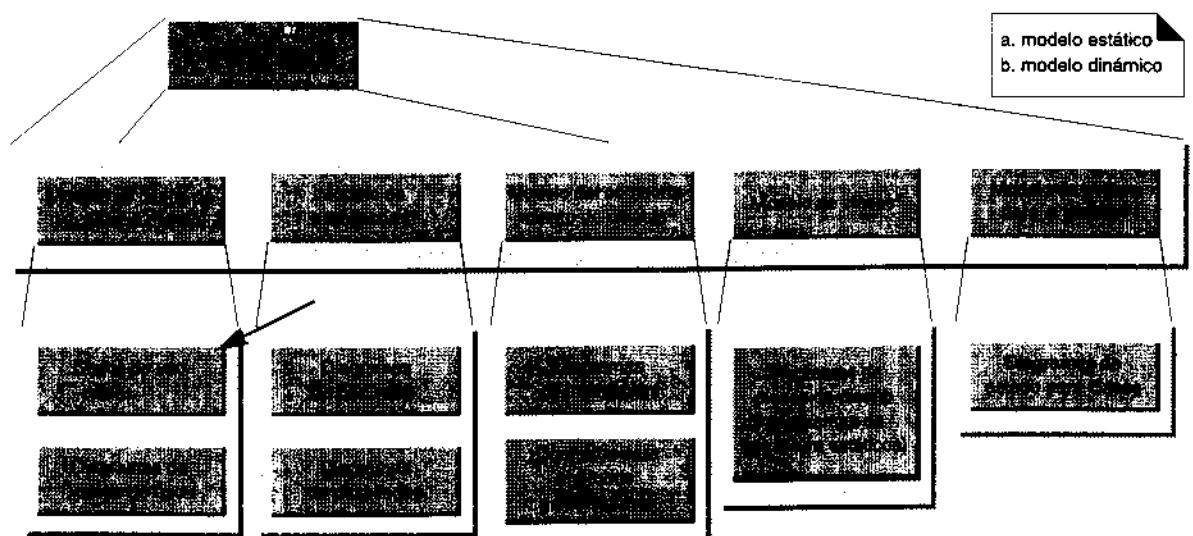


Figura 16.2 El modelo de diseño.

DIAGRAMAS DE COLABORACIÓN

Objetivos

- Leer la notación del UML para un diagrama de colaboración.

17.1 Introducción

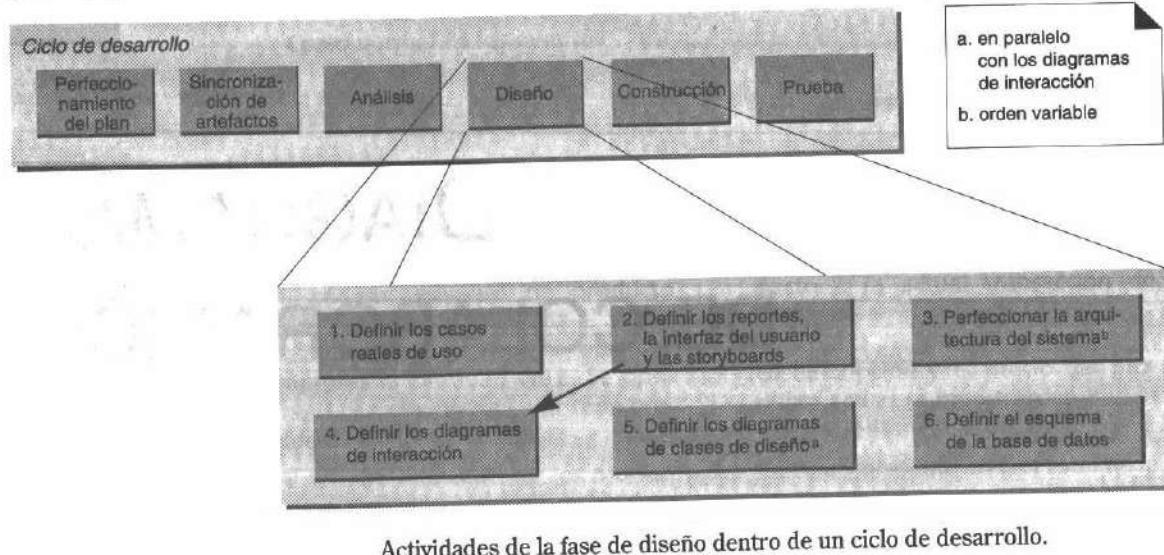
En los contratos de colaboración se incluye una primera conjectura óptima sobre las poscondiciones referentes al *inicio* de las operaciones del sistema: *inicio*, *introducirProducto*, *terminarVenta* y *efectuarPago*. Sin embargo, los contratos no muestran una solución de cómo los objetos de software van a cumplir con ellas.

El UML contiene **diagramas de interacción** que explican gráficamente cómo los objetos interactúan a través de mensajes para realizar las tareas. En el presente capítulo examinaremos su creación para la aplicación del punto de venta.

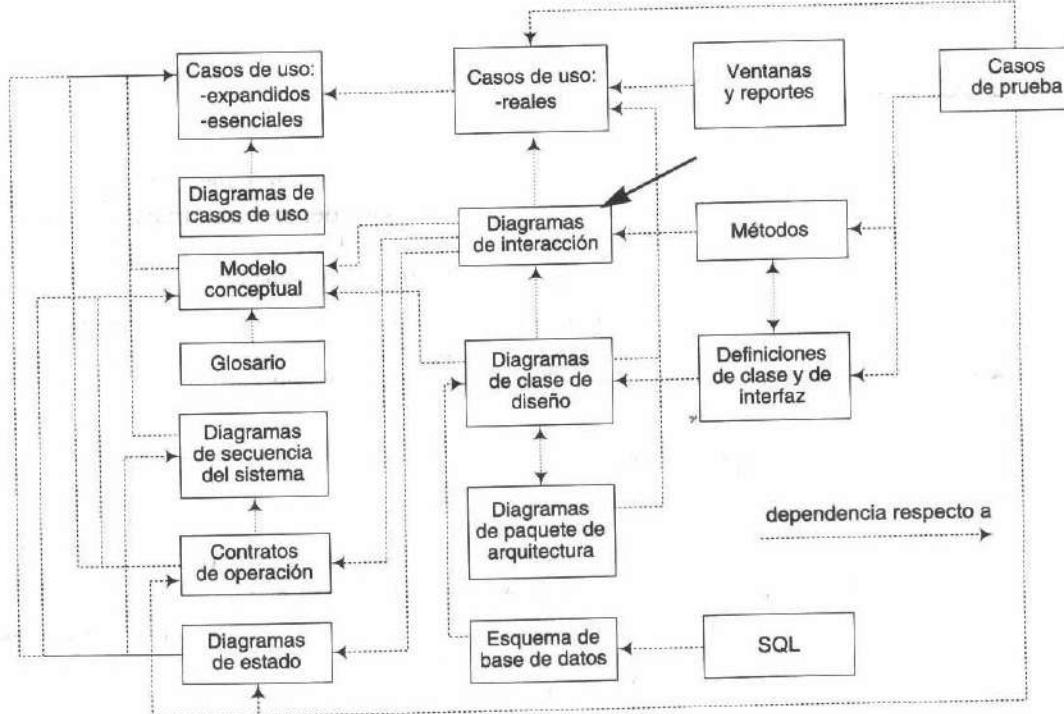
17.2 Actividades y dependencias

Los diagramas de interacción se realizan en la fase de diseño de un ciclo de desarrollo. No se pueden preparar si antes no se generan los siguientes artefactos:

- Un modelo conceptual: a partir de este modelo el diseñador podrá definir las clases del software correspondientes a los conceptos. Los objetos de las clases participan en las interacciones que se describen gráficamente en los diagramas.
- Contratos de la operación del sistema: a partir de ellos el diseñador identifica las responsabilidades y las poscondiciones que han de llenar los diagramas de interacción.



Actividades de la fase de diseño dentro de un ciclo de desarrollo.



Dependencias de los artefactos durante la fase de construcción.

- Casos de uso reales (o esenciales): a partir de ellos el diseñador recaba información sobre las tareas que realizan los diagramas de interacción, además de lo estipulado en los contratos.

17.3 Diagramas de interacción

Un **diagrama de interacción** explica gráficamente las interacciones existentes entre las instancias (y las clases) del modelo de éstas. El punto de partida de las interacciones es el cumplimiento de las poscondiciones de los contratos de operación.

El UML define dos tipos de estos diagramas; ambos sirven para expresar interacciones semejantes o idénticas de mensaje:

1. diagramas de colaboración
2. diagramas de secuencia

Los **diagramas de colaboración** describen las interacciones entre los objetos en un formato de grafo o red, como se aprecia en la figura 17.1.



Figura 17.1 Diagrama de colaboración.

Los **diagramas de secuencia** describen las interacciones en una especie de formato de cerca o muro, como se observa en la figura 17.2.

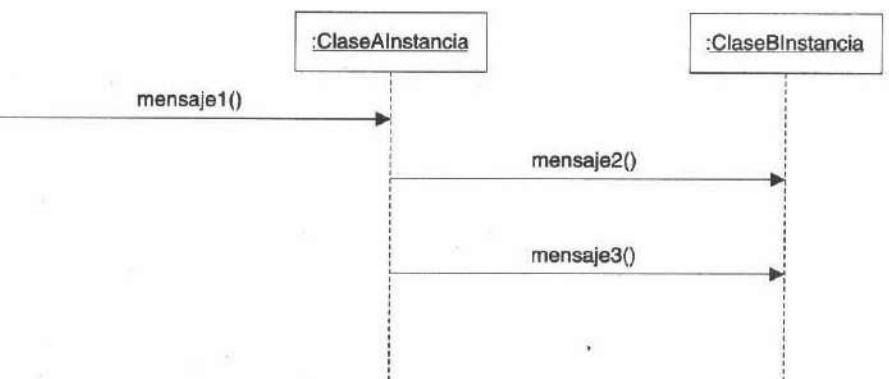


Figura 17.2 Diagrama de secuencia.

En este capítulo y a lo largo del libro nos centraremos en los diagramas de colaboración por su excepcional expresividad, su capacidad de comunicar más información

contextual y su economía de espacio.¹ Pero recuerde el lector que tanto una como otra notación pueden expresar conceptos parecidos.

17.4 Ejemplo de un diagrama de colaboración: efectuarPago

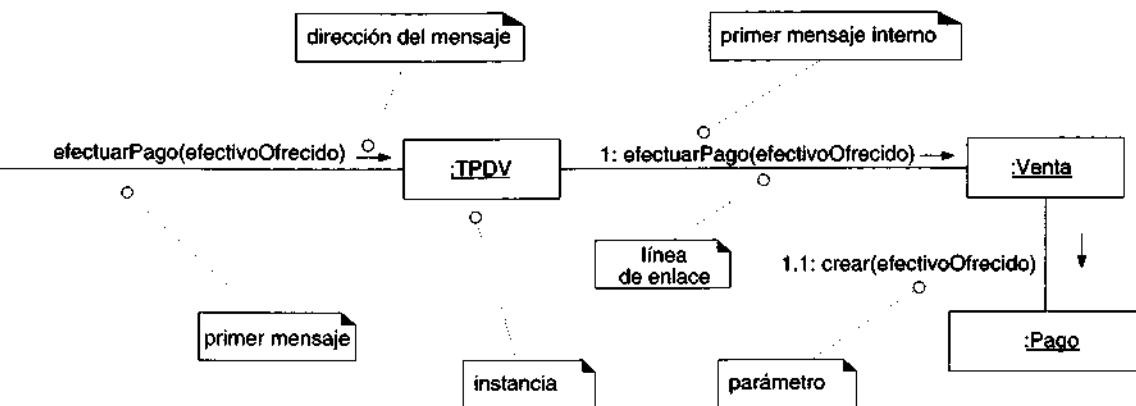


Figura 17.3 Diagrama de colaboración.

El diagrama de colaboración de la figura 17.3 se lee así:

1. El mensaje `efectuarPago` se envía a una instancia de `TPDV`. La instancia corresponde al mensaje `efectuarPago` de la operación del sistema.
2. El objeto `TPDV` envía el mensaje `efectuarPago` a la instancia `Venta`.
3. El objeto `Venta` crea una instancia de un `Pago`.

17.5 Los diagramas de interacción son un artefacto de gran utilidad

Un problema frecuente en los proyectos de la tecnología de objetos consiste en no comprender la utilidad de preparar los diagramas de interacción, la consideración cuidadosa de la asignación de responsabilidades y las habilidades que todo esto requiere. Son muy importantes la asignación de responsabilidades y el diseño de la colaboración entre los objetos.

Un porcentaje considerable de las actividades del proyecto ha de destinarse a esa fase. Más aún, es fundamentalmente durante ella cuando se exige aplicar las habilidades del diseño en los patrones, las expresiones y los principios.

¹ Los diagramas de colaboración nos permiten decir más en un espacio que los diagramas de secuencia y expresar además más información contextual; por ejemplo, el tipo de visibilidad entre los objetos. También resulta más fácil expresar la lógica condicional y la concurrencia.

En cierto modo, la creación de los casos de uso, de los modelos conceptuales y de otros artefactos resulta más fácil que asignar responsabilidades y elaborar diagramas bien diseñados de interacción. Ello se debe a lo siguiente: los sutiles principios del diseño en que se fundan son muchos más numerosos que cualquier análisis orientado a objetos o que cualquier artefacto de diseño.

Los diagramas de interacción constituyen uno de los artefactos más importantes que se generan en el análisis y en el diseño orientados a objetos.

El tiempo y el esfuerzo dedicados a su preparación deberían absorber un porcentaje considerable de la actividad total destinada al proyecto.

Para mejorar la calidad de su diseño, es posible aplicar patrones, principios y expresiones codificados.

Los principios del diseño necesarios para construir eficazmente los diagramas de interacción *pueden* codificarse, explicarse y aplicarse de modo metódico. Esta forma de entender y utilizar los principios del diseño se basa en **patrones**: directrices y principios estructurados. Por tanto, luego de exponer la sintaxis de los diagramas de interacción, nos ocuparemos (en capítulos posteriores) de los patrones de diseño y de su aplicación a los diagramas de interacción.

17.6 Éste es un capítulo dedicado exclusivamente a la notación

En este capítulo nos proponemos describir y resumir la notación del UML para un tipo particular de diagrama de interacción: el diagrama de colaboración. No explicaremos los principios ni las directrices que rigen la elaboración de un diagrama de colaboración bien diseñado.

No es indispensable que el lector conozca todos los detalles de la notación que se ofrecen en este capítulo para proseguir el estudio del libro. No obstante, le recomendamos echar un vistazo a los ejemplos y familiarizarse un poco con ellos antes de pasar a los siguientes capítulos.

17.7 Lea las directrices de diseño en los siguientes capítulos

Hay varios principios de diseño que deben conocerse para poder crear buenos diagramas de interacción. Tras familiarizarse un poco con la notación de éstos, conviene leer los siguientes capítulos dedicados a esos principios y a la forma de aplicarlos.

17.8 Cómo preparar diagramas de colaboración

Aplique las siguientes normas cuando elabore un diagrama de colaboración.

Para preparar un diagrama de colaboración:

1. Elabore un diagrama por cada operación del sistema durante el ciclo actual de desarrollo.
 - En cada mensaje del sistema, dibuje un diagrama incluyéndolo como mensaje inicial.
2. Si el diagrama se torna complejo (por ejemplo, si no cabe holgadamente en una hoja de papel de 8.5 x 11), divídalo en diagramas más pequeños.
3. Diseñe un sistema de objetos interactivos que realicen las tareas, usando como punto de partida las responsabilidades del contrato de operación, las poscondiciones y la descripción de casos de uso. Aplique el GRASP y otros patrones para desarrollar un buen diseño.

17.8.1 Los diagramas de colaboración y otros artefactos

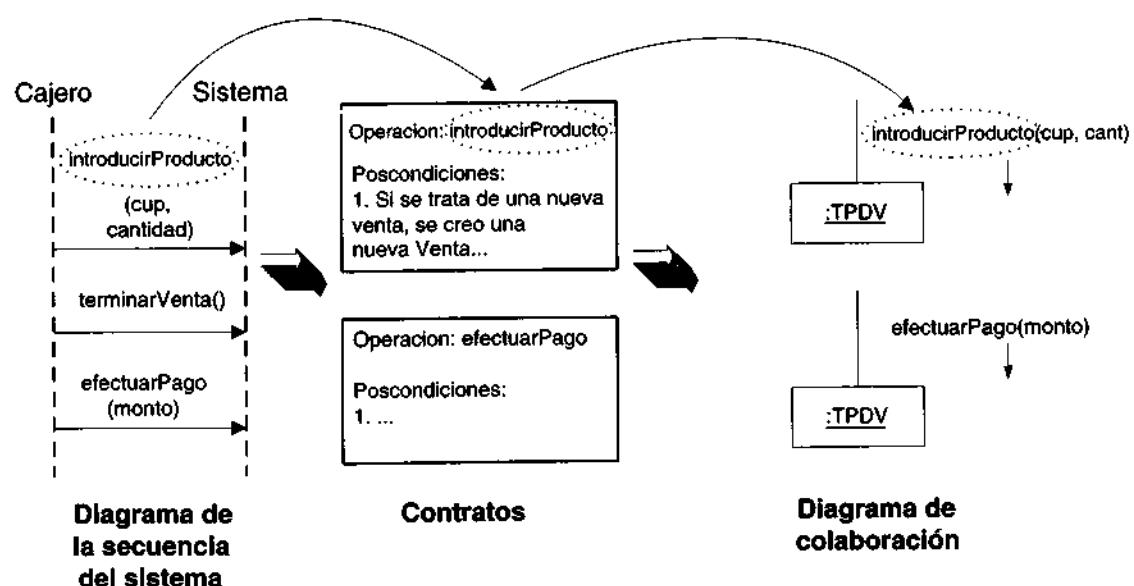


Figura 17.4 Relación entre los artefactos.

Como se observa en la figura 17.4, la relación entre los artefactos incluye lo siguiente:

- Los casos de uso indican los eventos del sistema que se muestran explícitamente en los diagramas de su secuencia.
- En los contratos se describe la mejor conjectura inicial sobre las operaciones del sistema.
- Las operaciones del sistema representan mensajes y éstos originan diagramas que explican gráficamente cómo los objetos interactúan para llevar a cabo las funciones requeridas.

17.9 Notación básica de los diagramas de colaboración

17.9.1 Representación gráfica de las clases y de las instancias

El UML ha adoptado un método simple y uniforme de describir visualmente las instancias para distinguirlas de los tipos (figura 17.5):

- Con cada tipo de elemento del UML (clase, actor, ...), una instancia utiliza el mismo símbolo gráfico usado para representar el tipo, pero se subraya el texto.

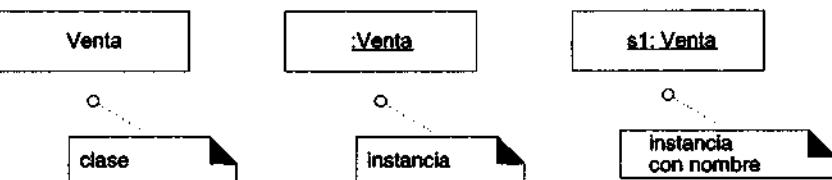


Figura 17.5 Clase e instancias.

Por tanto, para incluir la instancia de una clase en un diagrama de interacción, se recurre al símbolo gráfico usual de la casilla de la clase, sólo que el nombre se subraya. Además, en un diagrama de colaboración, al nombre de la clase siempre se le anteponen dos puntos.

Finalmente, un nombre de instancia sirve para identificarla de modo inequívoco.

17.9.2 Representación gráfica de los vínculos

El vínculo (o enlace) es una trayectoria de conexión entre dos instancias; indica alguna forma de navegación y visibilidad que es posible entre las instancias (figura 17.6). En un lenguaje más formal, el vínculo es una instancia de una asociación. Si vemos dos instancias en una relación de cliente/servidor, una trayectoria de navegación del cliente al servidor significa que los mensajes pueden enviarse del primero al segundo. Así, existe un vínculo —o trayectoria de navegación— entre TPDV y una Venta, a lo largo del cual pueden fluir los mensajes; por ejemplo, el mensaje *agregarPago*.

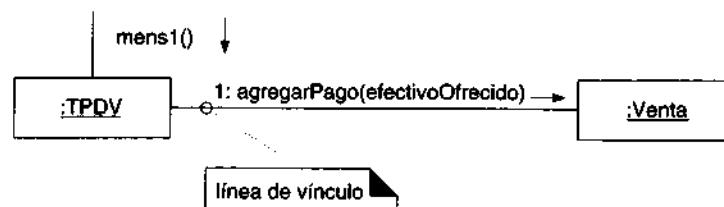


Figura 17.6 Líneas de vínculo (enlace).

17.9.3 Representación gráfica de los mensajes

Los mensajes entre objetos pueden representarse por medio de una flecha con un nombre y situada sobre una línea del vínculo. A través de éste puede fluir un número indefinido de mensajes (figura 17.7). Se agrega un número de secuencia que indique el orden consecutivo de los mensajes en la serie actual de control.

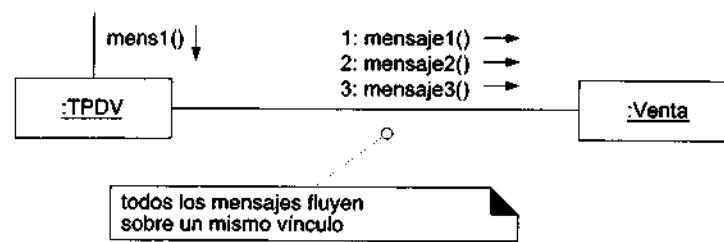


Figura 17.7 Mensajes.

17.9.4 Representación gráfica de los parámetros

Los parámetros de un mensaje pueden anotarse dentro de paréntesis después del nombre del mensaje (figura 17.8). Es opcional incluir o no el tipo de parámetro en cuestión.

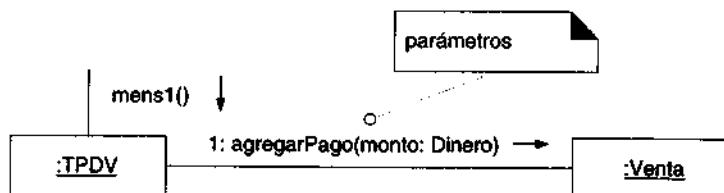


Figura 17.8 Parámetros.

17.9.5 Representación gráfica del mensaje de devolver valor

Puede incluirse un valor de retorno anteponiéndole al mensaje un nombre de variable de esa instrucción y un operador de asignación (':=') (figura 17.9). Es opcional mostrar el tipo del valor de retorno.

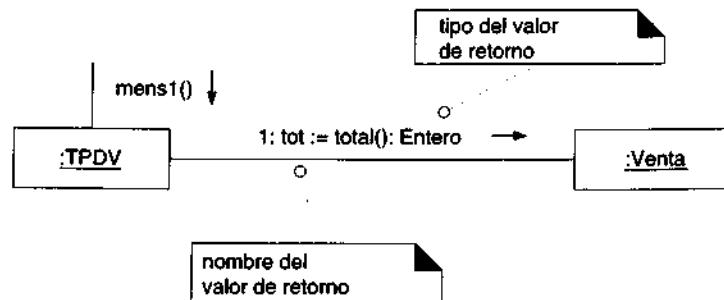


Figura 17.9 Devolver valores.

17.9.6 Sintaxis de los mensajes

El lenguaje UML cuenta con una sintaxis estándar para los mensajes:

retorno ::= mensaje(parametro : tipoParametro) : tipoRetorno

No obstante, como se aprecia en la figura 17.10, es legal servirse de otra sintaxis como la de Java o la de Smalltalk. Recomendamos emplear la sintaxis estándar de UML a fin de que los diagramas de colaboración sigan siendo un lenguaje relativamente independiente.

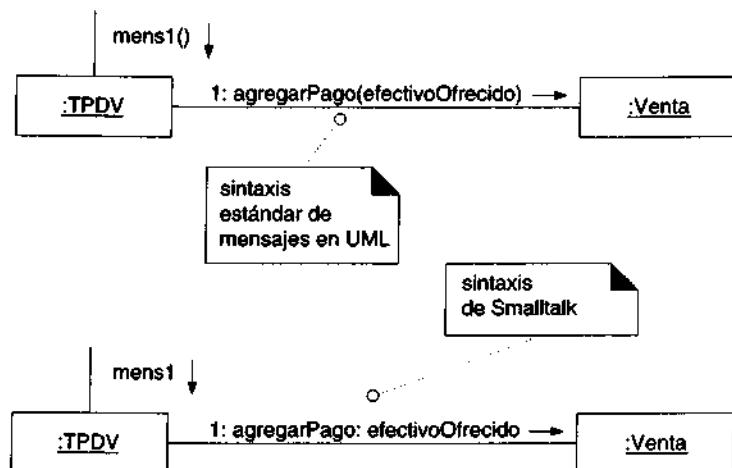


Figura 17.10 Los mensajes pueden expresarse en varias sintaxis.

17.9.7 Representación gráfica de los mensajes al “emisor” o a “esto”

Puede enviarse un mensaje de un objeto a sí mismo (figura 17.11).

Esto lo muestra gráficamente un vínculo consigo mismo, donde el mensaje fluye a lo largo del vínculo.

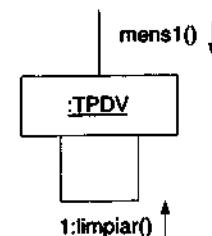


Figura 17.11 Mensajes a “esto”.

17.9.8 Representación gráfica de la iteración

La iteración se indica posponiendo un asterisco (*) al número de secuencia.

Ese símbolo significa que, dentro de un ciclo, el mensaje va a ser enviado repetidamente al receptor (figura 17.12).

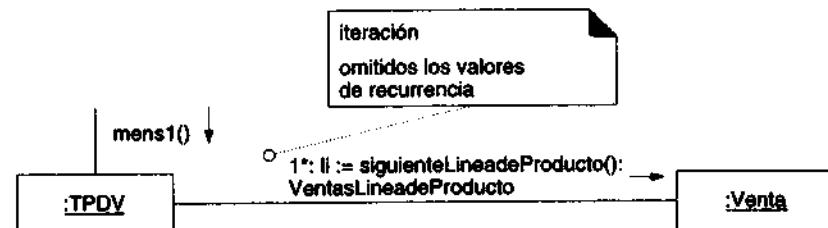


Figura 17.12 Iteración.

También es posible incluir una cláusula de iteración que indique los valores de recurrencia (figura 17.13).

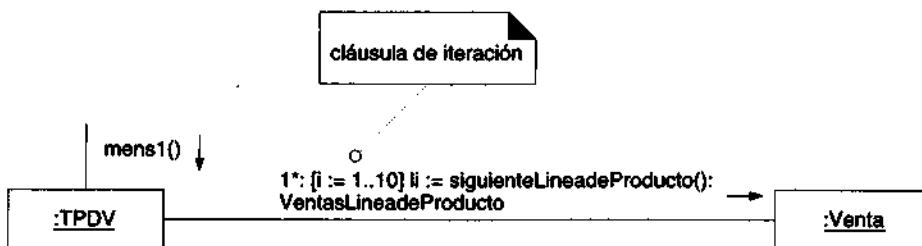


Figura 17.13 Cláusula de iteración.

Si se expresa más de un mensaje que ocurre dentro de la misma cláusula de iteración (por ejemplo, una serie de mensajes en un ciclo *for*), se repetirá la cláusula con cada mensaje (figura 17.14).

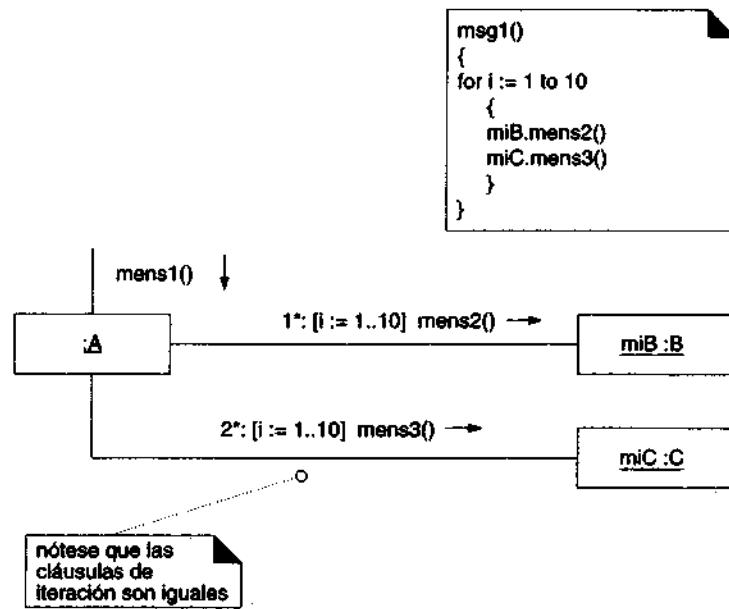


Figura 17.14 Mensajes múltiples dentro de la misma cláusula de iteración.

17.9.9 Representación gráfica de la creación de instancias

El mensaje de creación independiente del lenguaje es *crear*, que se muestra en el momento de ser enviado a la instancia que vamos a generar (figura 17.15).

Aunque en la mayoría de los lenguajes orientados a objetos, las instancias suelen generarse utilizando un mensaje *nuevo* (u operador) con la clase y no con una instancia, esta notación ocupa menos espacio aunque no sea completamente exacta.

Es opcional que la nueva instancia contenga o no un símbolo «*nuevo*».¹

El mensaje *crear* puede contener parámetros, lo cual indica la transferencia de los valores iniciales. En Java, de ese modo se indican —por ejemplo— los parámetros de constructores.

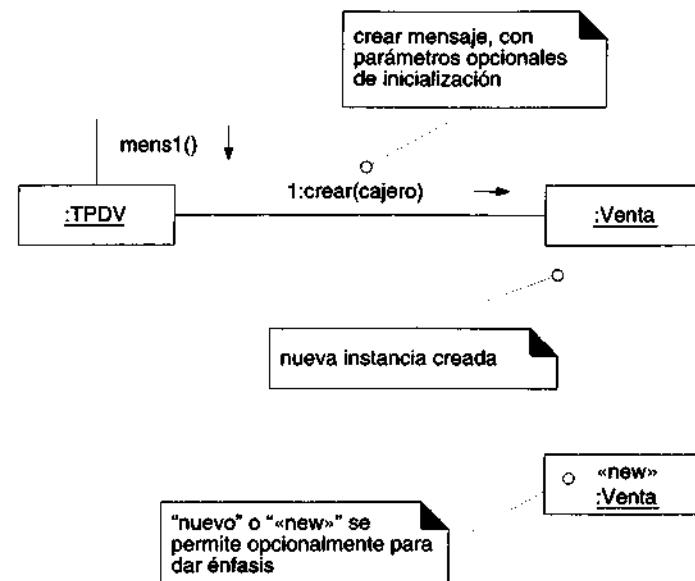


Figura 17.15 Creación de instancias.

En varios lenguajes, el mensaje *crear* se traduce así:

Creado por	
C++	Asignación automática u operador <i>new</i> seguido de una llamada a un constructor.
Java	operador <i>new</i> seguido de una llamada a un constructor.
Smalltalk	mensaje <i>new</i> o una variación de <i>new</i> : seguido del mensaje <i>initialize</i> .

¹ Un estereotipo del UML que estudiaremos más adelante.

17.9.10 Representación gráfica de la secuencia de número de los mensajes

El orden de los mensajes se indica con un **número de secuencia**, como se aprecia en la figura 17.16. El esquema de la numeración es:

1. El primer mensaje no se numera. Así, *mens1()* no lleva número.
2. El orden y el anidamiento de los mensajes siguientes se indican con un esquema legal de numeración, donde a los mensajes anidados se les ha antepuesto un número. La anidación se denota anteponiendo el número del mensaje de entrada al del mensaje de salida.

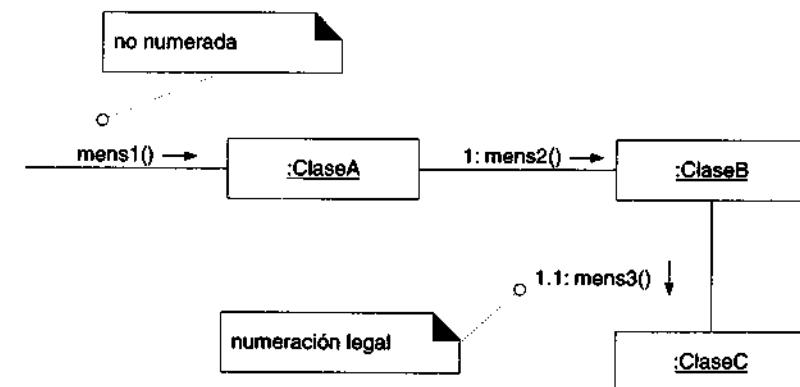


Figura 17.16 Numeración de secuencias.

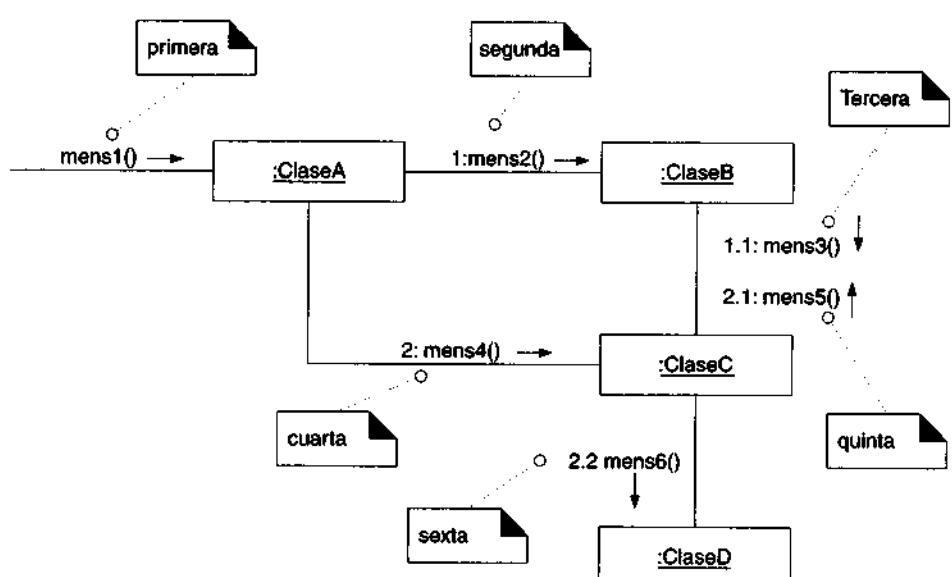


Figura 17.17 Numeración compleja de secuencias.

17.9.11 Representación gráfica de los mensajes condicionales

Un mensaje condicional (figura 17.18) se indica posponiendo al número de la secuencia una cláusula condicional entre corchetes, en forma parecida a como se hace con una cláusula de iteración. El mensaje se envía sólo si la cláusula se evalúa como *verdadera*.

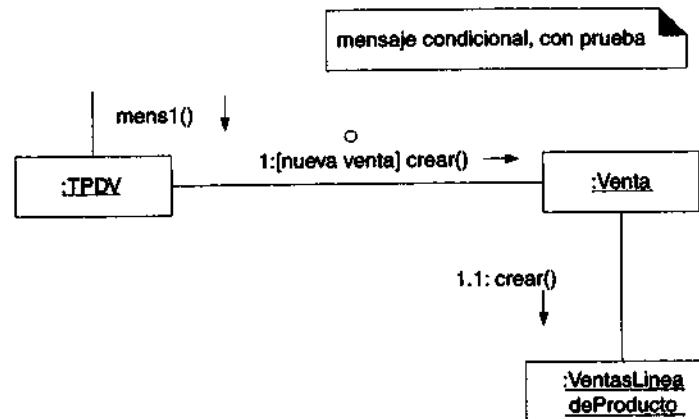


Figura 17.18 Mensaje condicional.

17.9.12 Representación gráfica de trayectorias condicionales mutuamente excluyentes

El ejemplo de la figura 17.19 contiene los números de secuencia con trayectorias condicionales que se excluyen mutuamente.

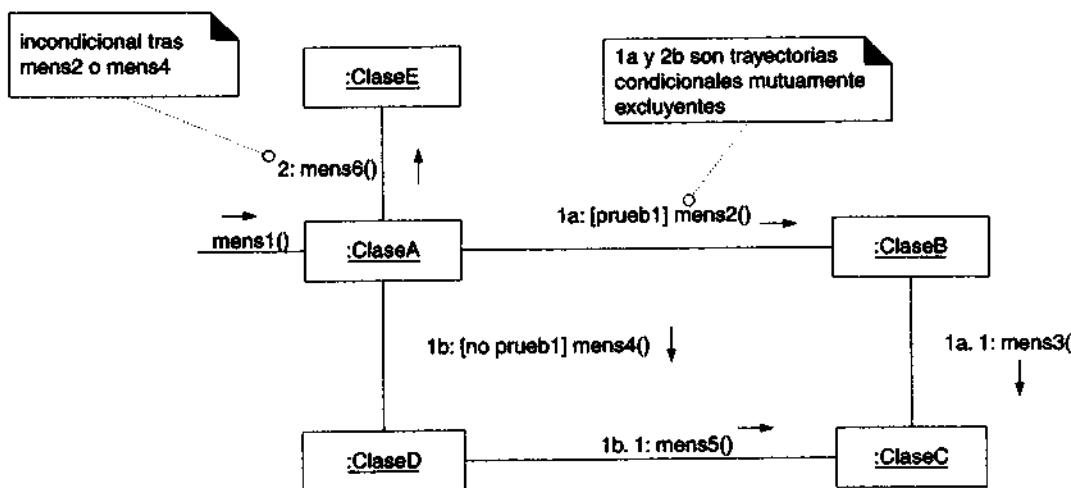


Figura 17.19 Mensajes mutuamente excluyentes.

En este caso es necesario modificar las expresiones de la secuencia con una letra de la trayectoria condicional. Por convención, la primera letra en usarse es *a*. La figura 17.19 establece qué tanto *1a* como *1b* podrían ejecutarse después de *mens1()*. Ambas son el número de la secuencia 1 porque pueden ser el primer mensaje interno.

Nótese que a los subsecuentes mensajes anidados todavía se les sigue anteponiendo la secuencia de sus mensajes externos. Así *1b.1* es un mensaje anidado dentro de *1b*.

17.9.13 Representación gráfica de las colecciones

Un **multiobjeto**, o conjunto de instancias, puede dibujarse como un ícono de pila según se observa en la figura 17.20.



Figura 17.20 Un multiobjeto.

Un multiobjeto suele implementarse como un grupo de instancias guardadas en un contenedor u objeto colección; por ejemplo, un *vector* de la STL de C++, un *Vector* de Java o una *ColecciónOrdenada* (*OrderedCollection*) de Smalltalk. Pero no necesariamente se implementa así; representa tan sólo un conjunto lógico de instancias.

17.9.14 Representación gráfica de los mensajes dirigidos a multiobjetos

Un mensaje dirigido a un ícono de multiobjeto indica que se envía al objeto colección. Así, en la figura 17.21 el mensaje *tamaño* está siendo enviado a la instancia *java.util.Vector* para solicitar el número de elementos del *Vector*.

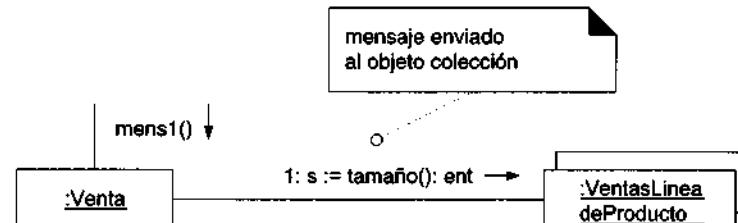


Figura 17.21 Mensaje dirigido a un multiobjeto.

En el lenguaje UML, los mensajes dirigidos a un multiobjeto *no* se transmite a todos los elementos (como sucedía en las versiones anteriores del UML).

En la figura 17.22 se explican gráficamente los mensajes a un multiobjeto y a un elemento.

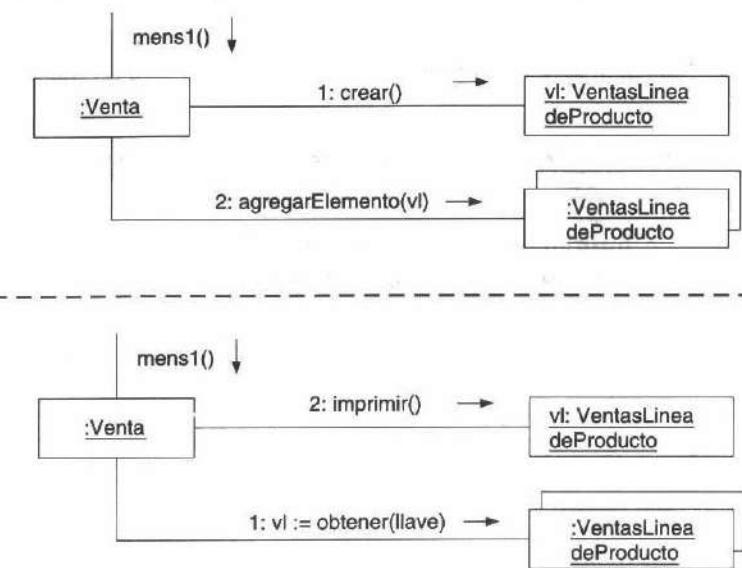


Figura 17.22 Mensajes dirigidos a un multiobjeto y a un elemento.

17.9.15 Representación gráfica de los mensajes dirigidos a un objeto clase

Los mensajes pueden ser dirigidos a la propia clase y no a una instancia, con el fin de llamar los métodos de la clase. Por ejemplo, en Java éstos se implementan como métodos estáticos; en Smalltalk son métodos de clases.

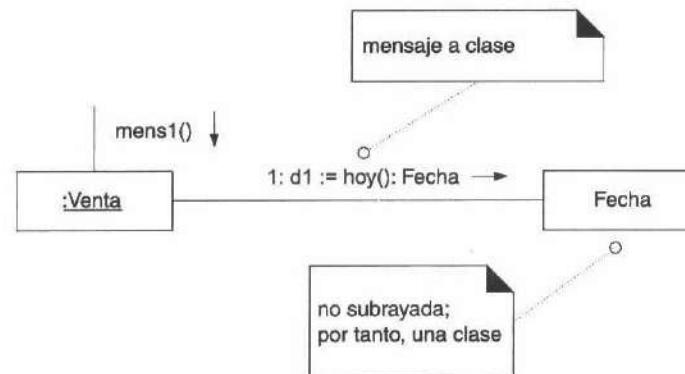


Figura 17.23 Mensajes a un objeto clase (llamada a método estático).

Los mensajes se representan de un modo muy simple: se incluyen dentro de una casilla de clases cuyo nombre no esté subrayado; se indica con ello que el mensaje va a ser enviado a una clase y no a una instancia (figura 17.23).

En consecuencia, es importante ser consistente en subrayar los nombres de las instancias cuando se desea denotar una instancia; pues de lo contrario pueden interpretarse erróneamente los mensajes dirigidos a las instancias y los dirigidos a las clases.

17.10 Modelos muestra

Los diagramas de interacción son miembros del modelo del comportamiento de objetos, ya que describen el comportamiento de los objetos del software.

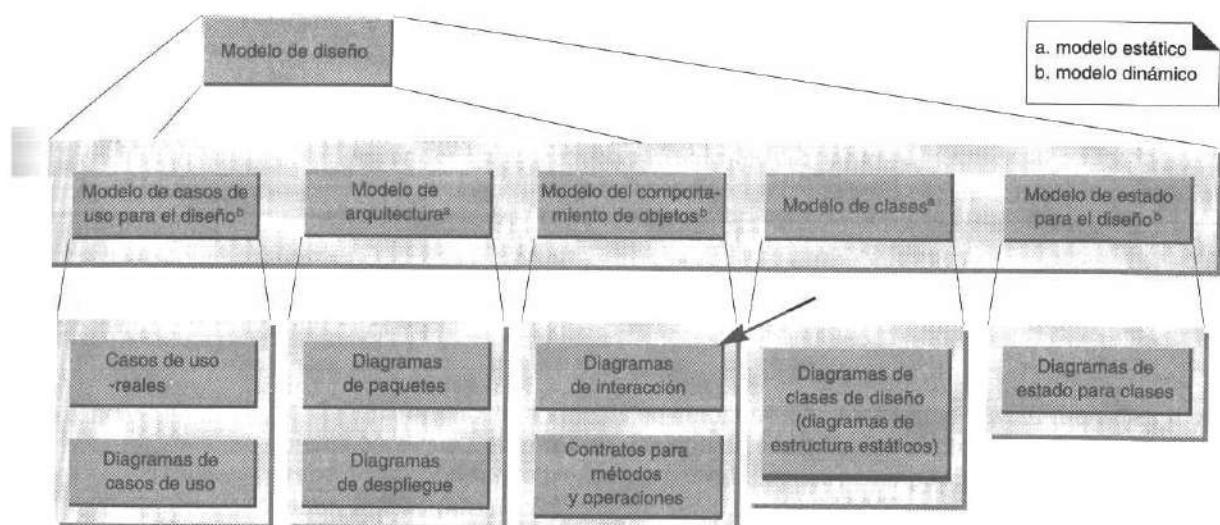


Figura 17.24 El modelo de diseño.

GRASP: PATRONES PARA ASIGNAR RESPONSABILIDADES

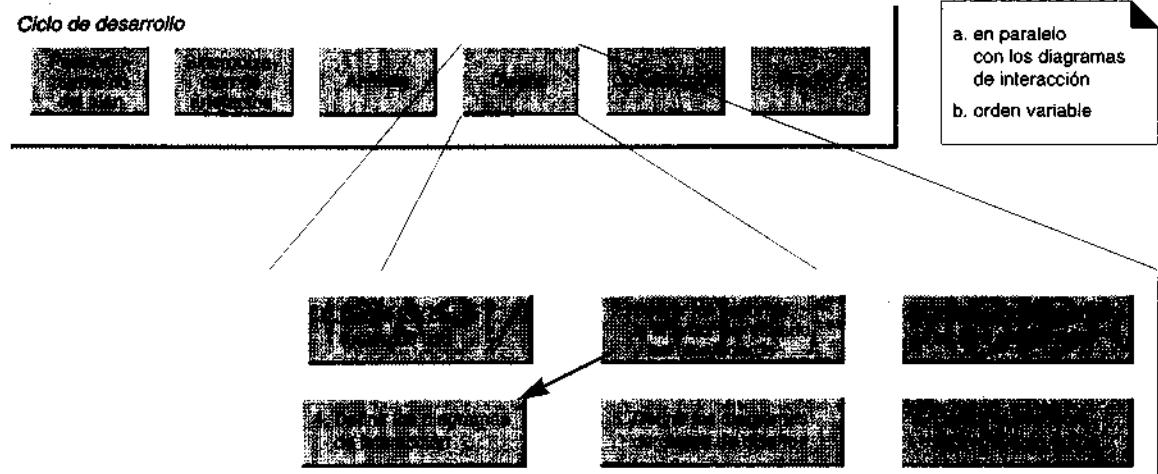
Objetivos

- Definir patrones.
- Aprender a aplicar cinco patrones GRASP.

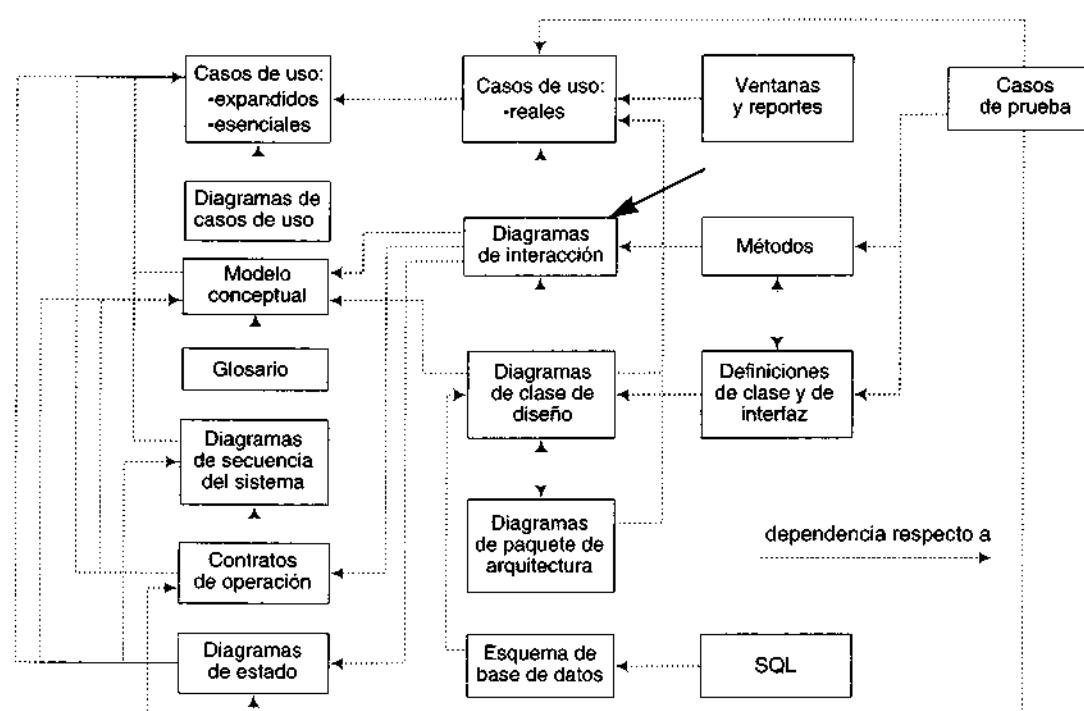
18.1 Introducción

Un sistema orientado a objetos se compone de objetos que envían mensajes a otros objetos para que lleven a cabo las operaciones. En los contratos se incluye una conjectura inicial óptima sobre las responsabilidades y las poscondiciones de las operaciones *inicio*, *introducirProducto*, *terminarVenta* y *efectuarPago*. Los diagramas de interacción describen gráficamente la solución —a partir de los objetos en interacción— que estas responsabilidades y poscondiciones satisfacen.

La calidad de diseño de la interacción de los objetos y la asignación de responsabilidades presentan gran variación. Las decisiones poco acertadas dan origen a sistemas y componentes frágiles y difíciles de mantener, entender, reutilizar o extender. Una implementación hábil se funda en los principios cardinales que rigen un buen diseño orientado a objetos. En los patrones GRASP se codifican algunos de ellos, que se aplican al preparar los diagramas de interacción, cuando se asignan las responsabilidades o durante ambas actividades.



Actividades de la fase de diseño dentro de un ciclo de desarrollo.



Dependencias de los artefactos durante la fase de construcción.

18.2 Actividades y dependencias

Los patrones a los que nos referimos se aplican durante la elaboración de los diagramas de interacción, al asignar las responsabilidades a los objetos y al diseñar la colaboración entre ellos.

18.3 Los diagramas de interacción bien diseñados son muy útiles

A continuación repetimos algunos puntos ya expuestos en el capítulo anterior cuando hablamos de los diagramas de colaboración:

- Los diagramas de interacción son algunos de los artefactos más importantes que se preparan en el análisis y diseño orientados a objetos.
- Es muy importante asignar acertadamente las responsabilidades al momento de elaborar los diagramas de interacción.
- El tiempo y el esfuerzo que se dedican a su elaboración, así como un examen riguroso de la asignación de responsabilidades, deberían absorber parte considerable de la fase de diseño de un proyecto.
- Los patrones, principios y expresiones especializadas codificados sirven para mejorar la calidad del diseño.

Los principios del diseño que se requieren para construir buenos diagramas de interacción pueden codificarse, explicarse y utilizarse en forma metódica. Esta manera de entender y usar los principios del diseño se funda en los *patrones con que se asignan las responsabilidades*.

18.4 Responsabilidades y métodos

Booch y Rumbaugh definen la **responsabilidad** como “un contrato u obligación de un tipo o clase” [BJR97]. Las responsabilidades se relacionan con las obligaciones de un objeto respecto a su comportamiento. Esas responsabilidades pertenecen, esencialmente, a las dos categorías siguientes:

1. conocer
2. hacer

Entre las responsabilidades de un objeto relacionadas con **hacer** se encuentran:

- hacer algo en uno mismo
- iniciar una acción en otros objetos

- controlar y coordinar actividades en otros objetos

Entre las responsabilidades de un objeto relacionadas con **conocer** se encuentran:

- estar enterado de los datos privados encapsulados
- estar enterado de la existencia de objetos conexos
- estar enterado de cosas que se puede derivar o calcular

Las responsabilidades se asignan a los objetos durante el diseño orientado a objetos. Por ejemplo, puede declararse que “una *Venta* es responsable de imprimirse ella misma” (un hacer) o que “una *Venta* tiene la obligación de conocer su fecha” (un *conocer*). Las responsabilidades relacionadas con “conocer” a menudo pueden inferirse del modelo conceptual por los atributos y asociaciones explicadas en él.

La granularidad de la responsabilidad influye en su traducción a clases y métodos. La responsabilidad de “brindar acceso a las bases relacionales de datos” puede incluir docenas de clases y cientos de métodos. En cambio, la de “imprimir una venta” tal vez no incluya más que un método o unos cuantos.

Responsabilidad no es lo mismo que método: los métodos se ponen en práctica para cumplir con las responsabilidades. Éstas se implementan usando métodos que operen solos o en colaboración con otros métodos y objetos. Así, la clase *Venta* podría definir uno o varios métodos que imprimen una instancia *Venta*; digamos el método *imprimir*. Para cumplir con esa responsabilidad, la *Venta* puede colaborar con otros objetos, entre ellos el envío de un mensaje a los objetos *VentasLineadeProducto*, pidiéndoles que se impriman ellos mismos.

18.5 Las responsabilidades y los diagramas de interacción

Este capítulo tiene por objeto ayudarle al lector a aplicar los principios fundamentales que rigen la asignación de responsabilidades a objetos. En los artefactos del UML, las responsabilidades (implementadas como métodos) suelen tenerse en cuenta al momento de preparar los diagramas de interacción, cuya notación ya examinamos en el capítulo anterior.

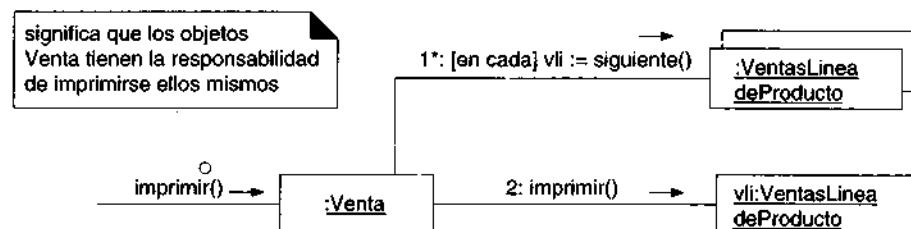


Figura 18.1 Las responsabilidades y los métodos están relacionados.

La figura 18.1 indica que a los objetos *Venta* se les ha asignado la responsabilidad de imprimirse ellos mismos, la cual se llama con un mensaje *imprimir* y se cumple con el método correspondiente *imprimir*. Más aún, para atender esta responsabilidad hay que colaborar con los objetos *VentasLineadeProducto*, pidiéndoles que impriman.

En resumen, los diagramas de interacción muestran las decisiones referentes a la asignación de responsabilidades entre los objetos. Cuando se preparan, se toman decisiones sobre la asignación que se reflejan en los mensajes que son enviados a varias clases de objetos. En el presente capítulo expondremos ampliamente los principios fundamentales —expresados en los patrones GRASP— para guiar las decisiones sobre la asignación de responsabilidades. Esas decisiones se reflejarán después en los diagramas de interacción.

18.6 Patrones

Los diseñadores expertos en orientación a objetos (y también otros diseñadores de software) van formando un amplio repertorio de principios generales y de expresiones que los guían al crear software. A unos y a otras podemos asignarles el nombre de **patrones**, si se codifican en un formato estructurado que describe el problema y su solución, y si se les asigna un nombre. A continuación ofrecemos un ejemplo de patrón.

Nombre del patrón:	Experto.
Solución:	Asignar una responsabilidad a la clase que tiene la información necesaria para cumplirla.
Problema que resuelve:	¿Cuál es el principio fundamental en virtud del cual asignaremos las responsabilidades a los objetos?

En la terminología de objetos, el **patrón** es una descripción de un problema y su solución que recibe un nombre y que puede emplearse en otros contextos; en teoría, indica la manera de utilizarlo en circunstancias diversas.¹ Muchos patrones ofrecen orientación sobre cómo asignar las responsabilidades a los objetos ante determinada categoría de problemas.

Expresado lo anterior con palabras más simples, el **patrón** es una pareja de problema/solución con un nombre y que es aplicable a otros contextos, con una sugerencia sobre la manera de usarlo en situaciones nuevas.

“El patrón de un individuo puede ser la estructura primitiva de otra persona”, máxima con que en la tecnología de objetos se explica la vaguedad de los patrones [GHJV94]. En nuestro estudio de los patrones no abordaremos el tema de lo que conviene llamar patrón; nos centraremos en el valor pragmático de utilizar ese estilo como medio de presentar y recordar los principios tan útiles de la ingeniería del software.

¹ La notación formal de los patrones nació con los patrones arquitectónicos de Christopher Alexander [AIS77]. En los años ochenta, Kent Beck y Ward Cunningham hicieron su aplicación al software [Beck94, Coplien95].

18.6.1 Los patrones no suelen contener ideas nuevas

Los patrones no se proponen descubrir ni expresar nuevos principios de la ingeniería del software. Todo lo contrario: intentan codificar el conocimiento, las expresiones y los principios *ya existentes*: cuanto más trillados y generalizados, tanto mejor. En consecuencia, los patrones GRASP —que describiremos aquí— no introducen ideas novedosas; son una mera codificación de los principios básicos más usados.

18.6.2 Los patrones tienen nombre

En teoría, todos los patrones poseen nombres muy sugestivos. El asignar nombre a un patrón, a un método o a un principio ofrece las siguientes ventajas:

- Apoya el agrupamiento y la incorporación del concepto a nuestro sistema cognitivo y a la memoria.
- Facilita la comunicación.

Darle nombre a una idea compleja —digamos a un patrón— es un ejemplo de la fuerza de la abstracción: convierte una forma compleja en una forma simple con sólo eliminar los detalles. Por tanto, los patrones GRASP poseen nombres concisos como *Experto*, *Creador*, *Controlador*.

18.6.3 La asignación de nombre a los patrones mejora la comunicación

Cuando se le da nombre a un patrón, un simple nombre nos permite discutir con otros un principio en todos sus aspectos. Considere la siguiente conversación entre dos diseñadores de software que emplean una nomenclatura común de patrones (*Experto*, *Separacion Vista-Modelo* y otros términos afines) para tomar una decisión sobre un diseño:

Alfredo: “En tu opinión, ¿a qué objeto deberíamos asignar la responsabilidad de imprimir una *Venta*? Creo que convendría asignarla a *Separacion Vista-Modelo*. ¿Qué te parece una *VistadeReportedeVentas*?”

Teresa: “En mi opinión, *Experto* es una mejor solución por ser una impresión simple y porque Ventas tiene toda la información que se requiere en salida impresa. Creo que le asignaremos esta responsabilidad a la *Venta*.”

Alfredo: “Está bien, hagámoslo.”

Agrupar las expresiones especializadas y los principios del diseño con nombres de uso común facilita la comunicación y le confiere a la búsqueda un nivel de abstracción más alto.

18.7 GRASP: patrones de los principios generales para asignar responsabilidades

Resumimos a continuación la introducción anterior:

- Asignar correctamente las responsabilidades es muy importante en el diseño orientado a objetos.
- La asignación de responsabilidades a menudo se asignan en el momento de preparar los diagramas de interacción.
- Los patrones son parejas de problema/solución con un nombre, que codifican buenos principios y sugerencias relacionados frecuentemente con la asignación de responsabilidades.

Dicho esto, podemos proceder a examinar los patrones GRASP.

Pregunta: ¿Qué son los patrones GRASP?

Respuesta: Los patrones GRASP describen los principios fundamentales de la asignación de responsabilidades a objetos, expresados en forma de patrones.

Es importante entender y poder aplicar estos principios durante la preparación de un diagrama de interacción, pues un diseñador de software sin mucha experiencia en la tecnología de objetos debe dominarlos cuanto antes: constituyen el fundamento de cómo se diseñará el sistema.

GRASP es un acrónimo que significa General Responsibility Assignment Software Patterns (patrones generales de software para asignar responsabilidades).¹ El nombre se eligió para indicar la importancia de *captar* (*grasping*) estos principios, si se quiere diseñar eficazmente el software orientado a objetos.

18.7.1 Cómo aplicar los patrones GRASP

En las siguientes secciones explicaremos los primeros cinco patrones de GRASP:

- Experto
- Creador
- Alta Cohesión
- Bajo Acoplamiento
- Controlador

¹ Desde el punto de vista técnico, deberíamos escribir “Patrones GRAS” en vez de “Patrones GRASP” pero la segunda expresión suena mejor en inglés, idioma en que fue acuñado el término.

Hay otros patrones de los que nos ocuparemos en un capítulo posterior, pero vale la pena dominar los cinco anteriores porque se refieren a cuestiones y a aspectos fundamentales del diseño.

Por favor, estudie los patrones que siguen; observe cómo se emplean en los diagramas muestra de interacción y luego aplíquelos al preparar otros diagramas. Comience dominando *Experto*, *Creador*, *Controlador*, *Alta Cohesión* y *Bajo Acoplamiento*. Más tarde irá aprendiendo los patrones restantes.

18.8 La notación del UML para los diagramas de clase

Los diagramas de clase en UML presentan las clases de software en contraste con los conceptos de dominio. La casilla de clase consta de tres secciones; la tercera contiene los métodos de la clase, como se aprecia en la figura 18.2.

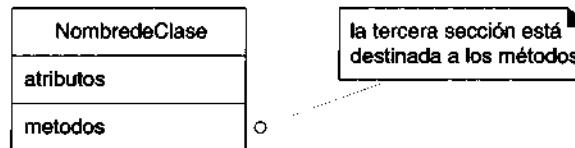


Figura 18.2 Las clases del software muestran los nombres de los métodos.

Los detalles de esta notación se abordan en un capítulo ulterior. En la siguiente exposición sobre los patrones, utilizaremos esporádicamente esta modalidad de la casilla de clase.

18.9 Experto

Solución Asignar una responsabilidad al experto en información: la clase que cuenta con la información necesaria para cumplir la responsabilidad.

Problema ¿Cuál es el principio fundamental en virtud del cual se asignan las responsabilidades en el diseño orientado a objetos?

Un modelo de clase puede definir docenas y hasta cientos de clases de software, y una aplicación tal vez requiera el cumplimiento de cientos o miles de responsabilidades. Durante el diseño orientado a objetos, cuando se definen las interacciones entre los objetos, tomamos decisiones sobre la asignación de responsabilidades a las clases. Si se hacen en forma adecuada, los sistemas tienden a ser más fáciles de entender, mantener y ampliar, y se nos presenta la oportunidad de reutilizar los componentes en futuras aplicaciones.

Ejemplo En la aplicación del punto de venta, alguna clase necesita conocer el gran total de la venta.

Comience asignando las responsabilidades con una definición clara de ellas.

A partir de esta recomendación se plantea la pregunta:

¿Quién es el responsable de conocer el gran total de la venta?

Desde el punto de vista del patrón Experto, deberíamos buscar la clase de objetos que posee la información necesaria para calcular el total. Examinemos detenidamente el modelo conceptual parcial de la figura 18.3.

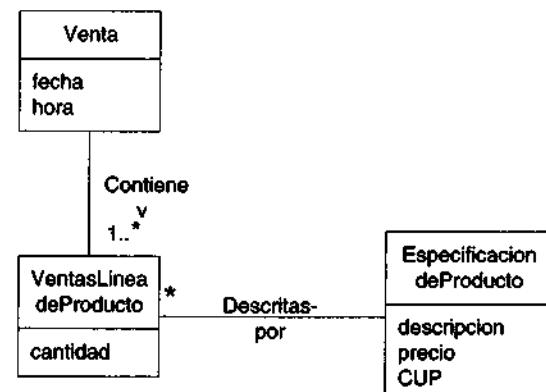


Figura 18.3 Asociaciones de la Venta.

¿Qué información hace falta para calcular el gran total? Hay que conocer todas las instancias *VentasLineadeProducto* de una venta y la suma de sus subtotales. Y esto lo

conoce únicamente la instancia *Venta*; por tanto, desde el punto de vista del Experto, *Venta* es la clase correcta de objeto para asumir esta responsabilidad; es el *experto en información*.

Como se señaló con anterioridad, es dentro del contexto de la preparación de los diagramas de interacción (por ejemplo, los de colaboración) donde surgen estas cuestiones concernientes a la responsabilidad. Imagine que vamos a empezar a dibujar diagramas para asignar responsabilidades a los objetos. El diagrama parcial de colaboración y el de clases de la figura 18.4 describen gráficamente las decisiones que hemos adoptado hasta ahora.

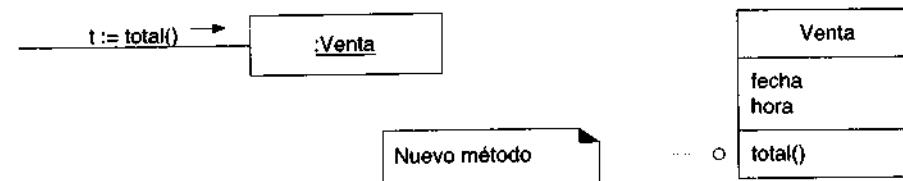


Figura 18.4 Diagrama parcial de colaboración.

Todavía no terminamos. ¿Qué información hace falta para determinar el subtotal de la línea de productos? Se necesitan *VentasLineadeProducto.cantidad* y *EspecificaciondeProducto.precio*. *VentasLineadeProducto* conoce su cantidad y su correspondiente *EspecificaciondeProducto*; por tanto, desde la perspectiva de patrón Experto, *VentasLineadeProducto* debería calcular el subtotal; es el *experto en información*.

Por lo que respecta a un diagrama de colaboración, lo anterior significa que la *Venta* necesita enviar mensajes de *subtotal* a cada *VentasLineadeProducto* y sumar los resultados; el diseño en cuestión se incluye en la figura 18.5.

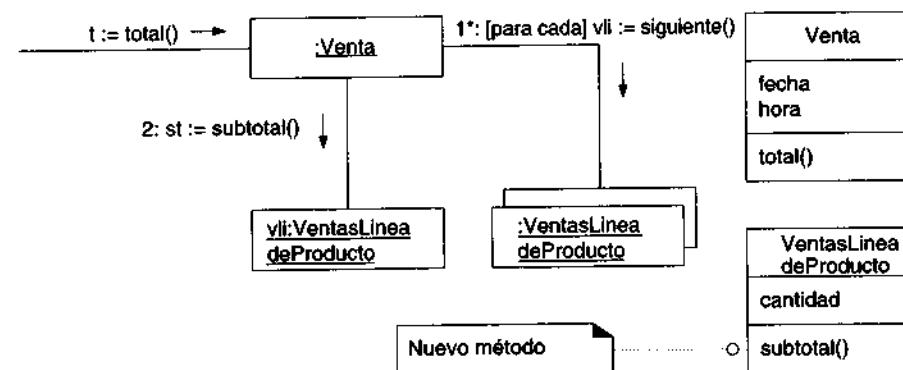


Figura 18.5 Cálculo del total de la venta.

VentasLineadeProducto no puede cumplir la responsabilidad de conocer y dar el subtotal, si no conoce el precio del producto. *EspecificaciondeProducto* es un Experto en información para contestar su precio; por tanto, habrá que enviarle un mensaje preguntándole el precio. El diseño correspondiente aparece en la figura 18.6.

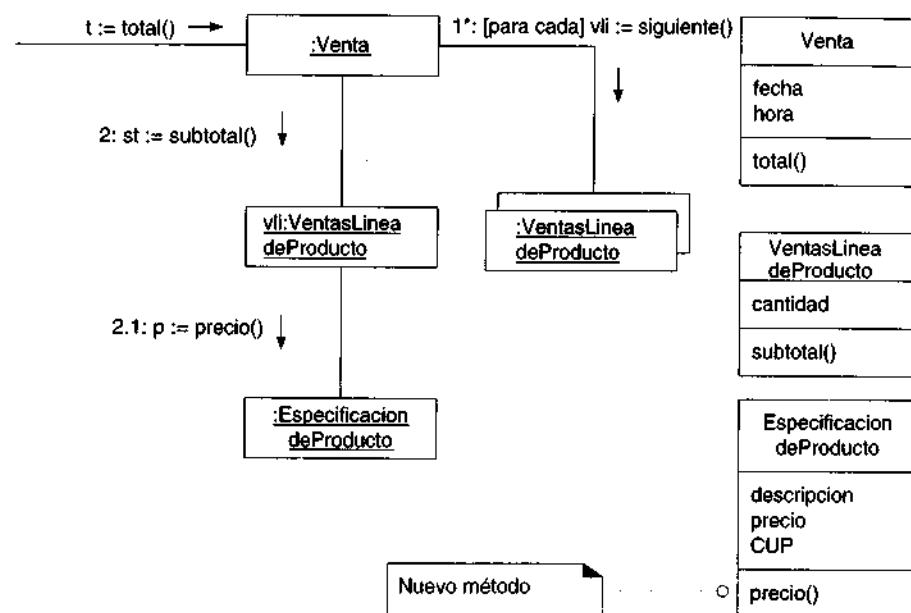


Figura 18.6 Cálculo del total de Venta.

En conclusión, para cumplir con la responsabilidad de conocer y dar el total de la venta, se asignaron tres responsabilidades a las tres clases de objeto así:

<i>Venta</i>	conoce el total de la venta
<i>VentasLineadeProducto</i>	conoce el subtotal de la línea de producto
<i>EspecificaciondeProducto</i>	conoce el precio del producto

El contexto donde las responsabilidades se consideraron y se decidieron fue el dibujo de un diagrama de colaboración. Después, la sección dedicada a métodos en un diagrama de clases puede resumir los métodos.

El principio en virtud del cual se asignaron las responsabilidades fue el patrón Experto: lo colocamos junto con el objeto que posee la información necesaria para cumplirlas.

Explicación Experto es un patrón que se usa más que cualquier otro al asignar responsabilidades; es un principio básico que suele utilizarse en el diseño orientado a objetos. Con él no se pretende designar una idea oscura ni extraña; expresa simplemente la "intuición" de que los objetos hacen cosas relacionadas con la información que poseen.

Nótese que el cumplimiento de una responsabilidad requiere a menudo información distribuida en varias clases de objetos. Ello significa que hay muchos expertos "parciales" que colaboraron en la tarea. Por ejemplo, el problema del total de la venta exigirá finalmente la colaboración de tres clases. Siempre que la información se encuentre esparcida en varios objetos, éstos habrán de interactuar a través de mensajes para compartir el trabajo.

El patrón Experto da origen a diseño donde el objeto de software realiza las operaciones que normalmente se aplican a la cosa real que representa: a esto Peter Coad lo llama estrategia de "Hacerlo yo mismo" [Coad95]. Por ejemplo, en el mundo real una venta no nos indica su total sin ayuda de aparatos electromecánicos; se trata de un concepto inanimado. Alguien calcula el total de la venta. Pero en el terreno del software orientado a objetos, todos los objetos del software están "vivos" o "animados", y pueden asumir responsabilidades y hacer cosas. Básicamente, hacen cosas relacionadas con la información de que disponen. A esto yo lo llamo principio de "Animación" en el diseño orientado a objetos; es como estar en una caricatura donde todo tiene vida.

El patrón Experto —como tantas otras cosas en la tecnología de objetos— ofrece una analogía con el mundo real. Acostumbramos asignar responsabilidad a individuos que disponen de la información necesaria para llevar a cabo una tarea. Por ejemplo, en una empresa ¿quién será el encargado de preparar un estado de pérdidas y ganancias? El empleado que tiene acceso a toda la información necesaria para elaborarlo: quizá el director financiero. Y del mismo modo que los objetos del software colaboran porque la información está esparcida, lo mismo sucede con el personal de una empresa. El director financiero podrá pedir a los encargados de cuentas por cobrar y de cuentas por pagar que generen reportes individuales sobre créditos y deudas.

Beneficios

- Se conserva el encapsulamiento, ya que los objetos se valen de su propia información para hacer lo que se les pide. Esto soporta un **bajo acoplamiento**, lo que favorece al hecho de tener sistemas más robustos y de fácil mantenimiento. (Bajo Acoplamiento es un patrón GRASP que examinaremos más adelante.)
- El comportamiento se distribuye entre las clases que cuentan con la información requerida, alejando con ello definiciones de clase "sencillas" y más cohesivas que son más fáciles de comprender y de mantener. Así se brinda soporte a una **alta cohesión** (patrón del que nos ocuparemos luego).

Otras formas de designar este patrón "Juntar responsabilidades y la información", "Lo hace el que conoce", "Animación", "Lo hago yo mismo", "Unir los servicios a los atributos sobre los que operan".

18.10 Creador

Solución Asignarle a la clase B la responsabilidad de crear una instancia de clase A en uno de los siguientes casos:

- B *agrega* los objetos A.
- B *contiene* los objetos A.
- B *registra* las instancias de los objetos A.
- B *utiliza* específicamente los objetos A.
- B *tiene los datos de inicialización* que serán transmitidos a A cuando este objeto sea creado (así que B es un Experto respecto a la creación de A).

B es un *creador* de los objetos A.

Si existe más de una opción, prefiera la clase B que *agregue* o *contenga* la clase A.

Problema ¿Quién debería ser responsable de crear una nueva instancia de alguna clase?

La creación de objetos es una de las actividades más frecuentes en un sistema orientado a objetos. En consecuencia, conviene contar con un principio general para asignar las responsabilidades concernientes a ella. El diseño, bien asignado, puede soportar un bajo acoplamiento, una mayor claridad, el encapsulamiento y la reutilizabilidad.

Ejemplo En la aplicación del punto de venta, ¿quién debería encargarse de crear una instancia *VentasLineadeProducto*? Desde el punto de vista del patrón Creador, deberíamos buscar una clase que agregue, contenga y realice otras operaciones sobre este tipo de instancias. Examine atentamente el modelo conceptual parcial de la figura 18.7.

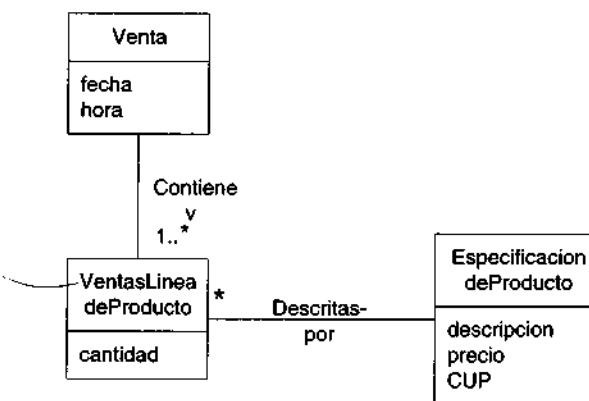


Figura 18.7 Modelo conceptual parcial.

Una *Venta* contiene (en realidad, agrega) muchos objetos *VentasLineadeProducto*; por ello, el patrón Creador sugiere que *Venta* es idónea para asumir la responsabilidad de crear las instancias *VentasLineadeProducto*.

Lo anterior nos permite diseñar las interacciones de los objetos en un diagrama de colaboración, según se observa en la figura 18.8.



Figura 18.8 Creación de un objeto *VentasLineadeProducto*.

Esta asignación de responsabilidades requiere definir en *Venta* un método de *hacerLineadeProducto*.

Una vez más, el contexto donde se consideraron y se asignaron estas responsabilidades fue el momento de dibujar un diagrama de colaboración. Después, en la sección destinada a métodos en un diagrama de clases, podemos resumir los resultados de la asignación de responsabilidades, realizadas concretamente como métodos.

Explicación El patrón Creador guía la asignación de responsabilidades relacionadas con la creación de objetos, tarea muy frecuente en los sistemas orientados a objetos. El propósito fundamental de este patrón es encontrar un creador que debemos conectar con el objeto producido en cualquier evento. Al escogerlo como creador, se da soporte al bajo acoplamiento.

El Agregado *agrega* la Parte, el Contenedor *contiene* el contenido, el Registro *registra*. En un diagrama de clases se registran las relaciones muy frecuentes entre las clases. El patrón Creador indica que la clase incluyente del contenedor o registro es idónea para asumir la responsabilidad de crear la cosa contenida o registrada. Desde luego, se trata tan sólo de una directriz.

Nótese que el concepto de **agregación** se utilizó al examinar el patrón Creador. Es un tema que trataremos más ampliamente en un capítulo posterior; pero damos aquí una definición sucinta: la agregación incluye cosas que están en una sólida relación de parte-todo o de parte-estructura; por ejemplo, Cuerpo agrega Pierna y Párrafo agrega oración.

En ocasiones encontramos un patrón creador buscando la clase con los datos de inicialización que serán transferidos durante la creación. Éste es en realidad un ejemplo del patrón Experto. Los datos de inicialización se transmiten durante la creación a través de algún método de inicialización, como un constructor en Java que cuenta con parámetros.

Suponga, por ejemplo, que una instancia *Pago* al momento de ser creada necesita inicializarse con el total *Venta*. Puesto que *Venta* conoce el total, es un buen candidato para ser el parámetro creador de *Pago*.

Beneficios ■ Se brinda soporte a un **bajo acoplamiento** (que describiremos más adelante), lo cual supone menos dependencias respecto al mantenimiento y mejores oportunidades de reutilización. Es probable que el acoplamiento no aumente, pues la clase *creada* tiende a ser visible a la clase *creador*, debido a las asociaciones actuales que nos llevaron a elegirla como el parámetro adecuado.

Patrones Conexos ■ Bajo acoplamiento.
■ Parte-todo [BMRSS96] describe un patrón para definir los objetos que soportan el encapsulamiento de sus componentes.

UNIVERSIDAD DE LA REPÚBLICA
FACULTAD DE INGENIERÍA
DEPARTAMENTO DE
DOCUMENTACIÓN Y BIBLIOTECA
MONTEVIDEO - URUGUAY

18.11 Bajo acoplamiento

Solución Asignar una responsabilidad para mantener bajo acoplamiento.

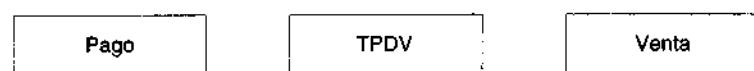
Problema ¿Cómo dar soporte a una dependencia escasa y a un aumento de la reutilización?

El **acoplamiento** es una medida de la fuerza con que una clase está conectada a otras clases, con que las conoce y con que recurre a ellas. Una clase con bajo (o débil) acoplamiento no depende de muchas otras; “muchas otras” depende del contexto, pero no lo estudiaremos aquí por el momento.

Una clase con alto (o fuerte) acoplamiento recurre a muchas otras. Este tipo de clases no es conveniente: presentan los siguientes problemas:

- Los cambios de las clases afines ocasionan cambios locales.
- Son más difíciles de entender cuando están aisladas.
- Son más difíciles de reutilizar porque se requiere la presencia de otras clases de las que dependen.

Ejemplo Examine con mucha atención el siguiente diagrama parcial de clases desde la perspectiva de la aplicación a la terminal del punto de venta:



Suponga que necesitamos crear una instancia *Pago* y asociarla a *Venta*. ¿Qué clase se encargará de hacer esto? Puesto que una instancia *TPDV* “registra” un *Pago* en el dominio del mundo real, el patrón Creador indica que *TPDV* es un buen candidato para producir el *Pago*. La instancia *TPDV* podría entonces enviarle a *Venta* el mensaje *agregarPago*, transmitiendo al mismo tiempo el nuevo *Pago* como parámetro. En la figura 18.9 se incluye un posible diagrama parcial de colaboración que representa gráficamente esto.

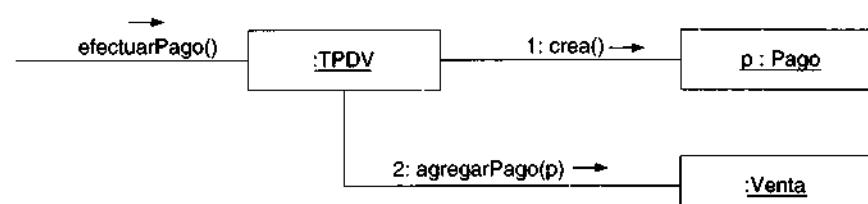


Figura 18.9 TPDV crea Pago.

Esta asignación de responsabilidades acopla la clase *TPDV* al conocimiento de la clase *Pago*. En la figura 18.10 se muestra una solución alterna para crear el *Pago* y asociarlo a la *Venta*.

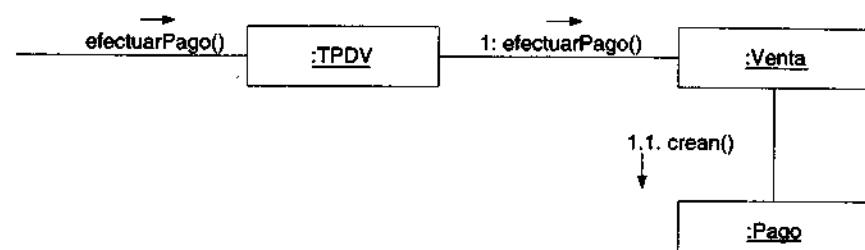


Figura 18.10 Venta crea Pago.

¿Qué diseño, basado en la asignación de responsabilidades, brinda soporte al patrón Bajo Acoplamiento? En ambos casos suponemos que *Venta* terminará acoplándose al conocimiento de un *Pago*. El diseño 1, donde la instancia *TPDV* crea *Pago*, incorpora el acoplamiento de *TPDV* a *Pago*; en cambio, el diseño 2, donde la *Venta* realiza la creación de un *Pago*, no incrementa el acoplamiento. Tomando como única perspectiva la del acoplamiento, el diseño 2 es preferible porque se conserva un menor acoplamiento global. Éste es un ejemplo donde dos patrones —Bajo Acoplamiento y Creador— pueden sugerir soluciones distintas. En la práctica, el grado de acoplamiento no puede considerarse aisladamente de otros principios como Experto y Alta Cohesión. Sin embargo, es un factor a considerar cuando se intente mejorar un diseño.

Explicación El Bajo Acoplamiento es un principio que debemos recordar durante las decisiones de diseño: es la meta principal que es preciso tener presente siempre. Es un *patrón evaluativo* que el diseñador aplica al juzgar sus decisiones de diseño.

En los lenguajes orientados a objetos como C++, Java y Smalltalk, las formas comunes de acoplamiento de *TipoX* a *TipoY* son las siguientes:

- *TipoX* posee un atributo (miembro de datos o variable de instancia) que se refiere a una instancia *TipoY* o al propio *TipoY*.
- *TipoX* tiene un método que a toda costa referencia una instancia de *TipoY* o incluso el propio *TipoY*. Suele incluirse un parámetro o una variable local de *TipoY* o bien el objeto devuelto de un mensaje es una instancia de *TipoY*.
- *TipoX* es una subclase directa o indirecta de *TipoY*.
- *TipoY* es una interfaz y *TipoX* la implementa.

El Bajo Acoplamiento estimula asignar una responsabilidad de modo que su colocación no incremente el acoplamiento tanto que produzca los resultados negativos propios de un alto acoplamiento.

El Bajo Acoplamiento soporta el diseño de clases más independientes, que reducen el impacto de los cambios, y también más reutilizables, que acrecientan la oportunidad de una mayor productividad. No puede considerarse en forma independiente

de otros patrones como Experto o Alta Cohesión, sino que más bien ha de incluirse como uno de los principios del diseño que influyen en la decisión de asignar responsabilidades.

El acoplamiento tal vez no sea tan importante, si no se busca la reutilización. Para apoyar una mejor reutilización de los componentes al hacerlos más independientes, el entero contexto de las metas de reuso ha de tenerse en cuenta antes de intentar reducir al mínimo el acoplamiento. Por ejemplo, a veces se dedica demasiado tiempo a la obtención de componentes reutilizables en futuros proyectos "utópicos", a pesar de que no se sabe con certeza que se necesitarán. Con ello no queremos decir que se pierde tiempo al tratar de reutilizarlos, sólo que han de atenderse las consideraciones de costo-beneficio.

Una subclase está acoplada firmemente con su superclase. La decisión de derivarla de una superclase ha de ser estudiada con mucho cuidado, pues es una forma muy sólida de acoplamiento. Supongamos, por ejemplo, que es necesario guardar persistentemente los objetos en una base de datos relacional o de objetos. En este caso, está bastante generalizado (aunque no se recomienda) el diseño para crear una superclase abstracta denominada *ObjetoPersistente* del cual se derivan otras clases. Esta subclase tiene la desventaja de acoplar estrechamente objetos del dominio con un servicio en particular y la ventaja de una herencia automática del comportamiento de persistencia, una especie de "matrimonio por conveniencia". Y esto rara vez es una buena decisión en las relaciones.

No existe una medida absoluta de cuándo el acoplamiento es excesivo. Lo importante es que el diseñador pueda determinar el grado actual de acoplamiento y si surgirán problemas en caso de incrementarlo. En términos generales, han de tener escaso acoplamiento las clases muy genéricas y con grandes probabilidades de reutilización.

El caso extremo de Bajo Acoplamiento ocurre cuando existe poco o nulo acoplamiento entre las clases. Ello no conviene porque una metáfora esencial en la tecnología de objetos es un sistema de objetos conectados que se comunican entre sí a través de mensajes. Si el Bajo Acoplamiento se lleva a los extremos, dará origen a un diseño deficiente por producir objetos incoherentes, atiborrados y complejos que hacen todo el trabajo, con muchos otros objetos muy pasivos y de acoplamiento cero que funcionan como meros depósitos de datos. Un grado moderado de acoplamiento entre las clases es normal y necesario si quiere crearse un sistema orientado a objetos, donde las tareas se realicen por colaboración entre objetos conectados.

- Beneficios**
- no se afectan por cambios de otros componentes
 - fáciles de entender por separado
 - fáciles de reutilizar

18.12 Alta cohesión

Solución Asignar una responsabilidad de modo que la cohesión siga siendo alta.

Problema ¿Cómo mantener la complejidad dentro de límites manejables?

En la perspectiva del diseño orientado a objetos, la **cohesión** (o, más exactamente, la cohesión funcional) es una medida de cuán relacionadas y enfocadas están las responsabilidades de una clase. Una alta cohesión caracteriza a las clases con responsabilidades estrechamente relacionadas que no realicen un trabajo enorme.

Una clase con baja cohesión hace muchas cosas no afines o un trabajo excesivo. No conviene este tipo de clases pues presentan los siguientes problemas:

- son difíciles de comprender
- son difíciles de reutilizar
- son difíciles de conservar
- son delicadas: las afectan constantemente los cambios

Las clases con baja cohesión a menudo representan un alto grado de abstracción o han asumido responsabilidades que deberían haber delegado a otros objetos.

Ejemplo El mismo ejemplo que hemos utilizado en el patrón Bajo Acoplamiento puede analizarse en el caso de Alta Cohesión.

Suponga que necesitamos crear una instancia de *Pago* (en efectivo) y asociarla a la *Venta*. ¿Qué clase será la responsable de hacerlo? Como *TPDV* registra un *Pago* en el dominio del mundo real, el patrón Creador sugiere que *TPDV* es un buen candidato para que genere *Pago*. Entonces la instancia *TPDV* podría enviar a *Venta* un mensaje *agregarPago*, transmitiendo al mismo tiempo el nuevo *Pago* como parámetro, según se muestra en la figura 18.11.

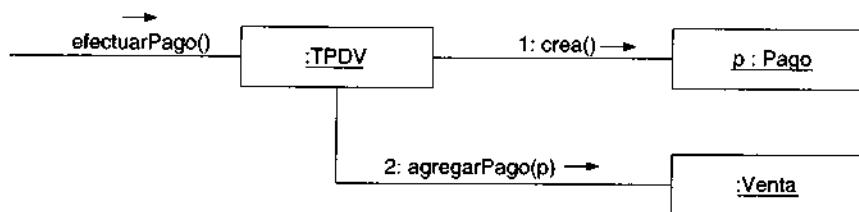


Figura 18.11 TPDV crea Pago.

Esta asignación de responsabilidades coloca en la clase *TPDV* la realización del pago. *TPDV* asume parte de la responsabilidad de realizar la operación del sistema *efectuarPago*.

En este ejemplo aislado, lo anterior es aceptable. Pero si seguimos haciendo que la clase *TPDV* se encargue de efectuar algún trabajo o la mayor parte del que se relaciona con un número creciente de operaciones del sistema, se irá saturando con tareas y terminará por perder la cohesión.

Imagine que el sistema tuviera 50 operaciones, todas ellas recibidas por la clase *TPDV*. Si lleva a cabo el trabajo relacionada con cada una, con el tiempo el sistema será una objeto “saturado” y sin cohesión. Lo importante no es que esta tarea individual de creación de *Pago* haga que *TPDV* pierda su cohesión, sino que sea parte de un panorama más general de la asignación global de responsabilidades y pueda indicar la tendencia a una menor cohesión.

En cambio, como se advierte en la figura 18.12, el segundo diseño delega a la *Venta* la responsabilidad de crear el pago, que soporta una mayor cohesión de *TPDV*. El segundo diseño brinda soporte a una alta cohesión y a un bajo acoplamiento; de ahí que sea conveniente.

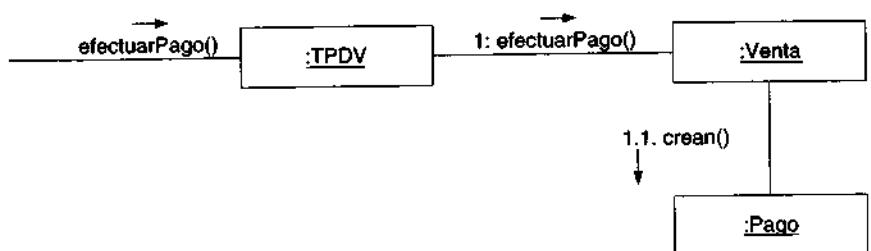


Figura 18.12 Venta crea Pago.

En la práctica, el nivel de cohesión no puede ser considerado independientemente de otras responsabilidades ni de otros principios como los patrones Experto y Bajo Acoplamiento.

Explicación Como el patrón Bajo Acoplamiento, también Alta Cohesión es un principio que debemos tener presente en todas las decisiones de diseño: es la meta principal que ha de buscarse en todo momento. Es un patrón evaluativo que el desarrollador aplica al valorar sus decisiones de diseño.

Grady Booch señala que se da una alta cohesión funcional cuando los elementos de un componente (clase, por ejemplo) “colaboran para producir algún comportamiento bien definido” [Booch94].

A continuación se mencionan algunos escenarios que ejemplifican los diversos grados de la cohesión funcional:

1. *Muy baja cohesión*. Una clase es la única responsable de muchas cosas en áreas funcionales muy heterogéneas.

- Suponga que existe una clase llamada *RDB-RPC-Interfaz*, la única encargada de interactuar con las bases relacionales de datos y de atender las llamadas de procedimientos remotos. Éstas son dos áreas funcionales radicalmente distintas, y cada una requiere mucho código de soporte. Las responsabilidades deberían repartirse en una familia de clases relacionada con el acceso RDB y en otra familia relacionada con el soporte RPC.

2. *Baja cohesión*. Una clase tiene la responsabilidad exclusiva de una tarea compleja dentro de un área funcional.

- Suponga que existe una clase llamada *RDBInterfaz*, la única encargada de interactuar con bases relacionales de datos. Los métodos de una clase están todos relacionados, pero hay muchos de ellos y una enorme cantidad de código de soporte; puede haber cientos o miles de métodos. La clase debería dividirse en una familia de clases ligeras que compartan el trabajo para ofrecer el acceso RDB.

3. *Alta cohesión*. Una clase tiene responsabilidades moderadas en un área funcional y colabora con las otras para llevar a cabo las tareas.

- Suponga que existe una clase denominada *RDBInterfaz*, que se encarga de una parte de la interacción con bases relacionales de datos. Interactúa con una docena de clases afines mediante el acceso RDB para recuperar objetos y guardarlos.

4. *Cohesión moderada*. Una clase tiene un peso ligero y responsabilidades exclusivas en unas cuantas áreas que están relacionadas lógicamente con el concepto de clase, pero no entre ellas.

- Suponga que existe una clase denominada *Compañía*, la única responsable de: a) conocer a sus empleados b) conocer la información financiera. Estas dos áreas no están estrechamente relacionadas entre ellas, aunque lógicamente lo están con el concepto de compañía. Además, el número total de métodos públicos es pequeño, lo mismo que el código de soporte.

Una regla práctica es la siguiente: una clase de alta cohesión posee un número relativamente pequeño, con una importante funcionalidad relacionada y poco trabajo por hacer. Colabora con otros objetos para compartir el esfuerzo si la tarea es grande.

Una clase con mucha cohesión es útil porque es bastante fácil darle mantenimiento, entenderla y reutilizarla. Su alto grado de funcionalidad, combinada con una reducida cantidad de operaciones, también simplifica el mantenimiento y los mejoramientos. La ventaja que significa una gran funcionalidad también soporta un aumento de la capacidad de reutilización.

El patrón Alta Cohesión —como tantas otras cosas en la tecnología de objetos— presenta semejanzas con el mundo real. Todos sabemos que, si alguien asume demasiadas responsabilidades —sobre todo las que debería delegar—, no será eficiente. Esto se observa en algunos gerentes que no han aprendido a delegar. Muestran baja cohesión; prácticamente ya están “desligados”.

- Beneficios**
- Mejoran la claridad y la facilidad con que se entiende el diseño.
 - Se simplifican el mantenimiento y las mejoras en funcionalidad.
 - A menudo se genera un bajo acoplamiento.
 - La ventaja de una gran funcionalidad soporta una mayor capacidad de reutilización, porque una clase muy cohesiva puede destinarse a un propósito muy específico.

18.13 Controlador

Solución Asignar la responsabilidad del manejo de un mensaje de los eventos de un sistema a una clase que represente una de las siguientes opciones:

- el “sistema” global (*controlador de fachada*).
- la empresa u organización global (*controlador de fachada*).
- algo en el mundo real que es activo (por ejemplo, el papel de una persona) y que pueda participar en la tarea (*controlador de tareas*).
- un manejador artificial de todos los eventos del sistema de un caso de uso, generalmente denominados “Manejador<NombreCasodeUso>” (*controlador de casos de uso*).

Utilice la misma clase de controlador con todos los eventos del sistema en el mismo caso de uso.

Corolario: Nótese que en esta lista no figuran las clases “ventana”, “aplicación”, “vista” ni “documento”. Estas clases *no* deberían ejecutar las tareas asociadas a los eventos del sistema; generalmente las reciben y las delegan al controlador.

Problema ¿Quién debería encargarse de atender un evento del sistema?

Un **evento del sistema** es un evento de alto nivel generado por un actor externo; es un evento de entrada externa. Se asocia a **operaciones del sistema**: las que emite en respuesta a los eventos del sistema. Por ejemplo, cuando un cajero que usa un sistema de terminal en el punto de venta oprime el botón “Terminar Venta”, está generando un evento sistémico que indica que “la venta ha terminado”. De modo similar, cuando un escritor que usa un procesador de palabras pulsa el botón “revisar ortografía”, está produciendo un evento que indica “realizar una revisión ortográfica”.

Un **Controlador** es un objeto de interfaz no destinada al usuario que se encarga de manejar un evento del sistema. Define además el método de su operación.

Ejemplo En la aplicación del punto de venta se dan varias operaciones del sistema, como se advierte en la figura 18.13.

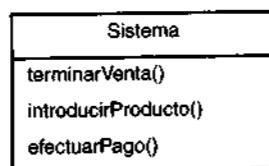


Figura 18.13 Operaciones del sistema asociadas a los eventos sistémicos.

Durante el análisis del comportamiento del sistema, sus operaciones son asignadas al tipo *Sistema*, para indicar que son operaciones del sistema. Pero ello *no* significa que una clase llamada *Sistema* las ejecute durante el diseño.

Más bien, durante el diseño, a la clase *Controlador* se le asigna la responsabilidad de las operaciones del sistema (figura 18.14).

¿Quién debería ser el controlador de eventos sistemáticos como *introducirProducto* y *terminarVenta*?

¿Qué clase de objeto debe encargarse de manejar este mensaje de eventos del sistema?

Un controlador.

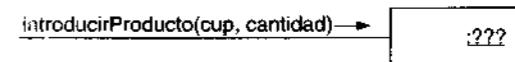


Figura 18.14 ¿Controlador de introducirProducto?

De acuerdo con el patrón Controlador, disponemos de las siguientes opciones:

representa el “sistema” global	<i>TPDV</i>
representa la empresa u organización global	<i>Tienda</i>
representa algo en el mundo real que está activo (por ejemplo, el papel de una persona) y que puede intervenir en la tarea	<i>Cajero</i>
representa un manejador artificial de todas las operaciones del sistema de un caso de uso.	<i>ManejadordeComprar-Productos</i>

Por lo que respecta a los diagramas de colaboración, lo anterior significa que se usará uno de los ejemplos de la figura 18.15.

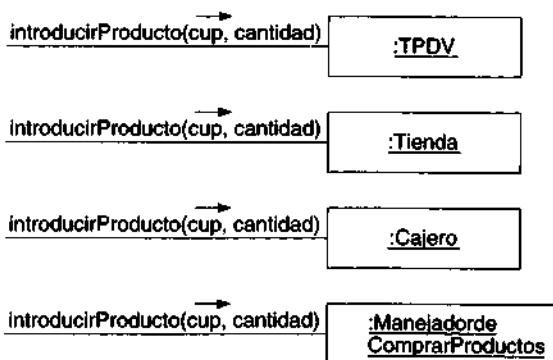


Figura 18.15 Decisiones del controlador.

En la decisión de cuál de las cuatro clases es el controlador más apropiado influyen también otros factores como la cohesión y el acoplamiento. La sección de la explicación profundiza este aspecto.

Durante el diseño, las operaciones del sistema detectadas en el análisis de su comportamiento se asignan a una o más clases de controladores como *TPDV*, según se observa en la figura 18.16.

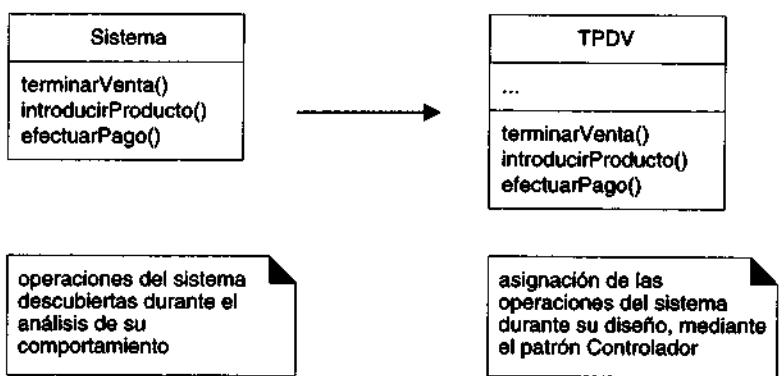


Figura 18.16 Asignación de las operaciones del sistema.

Explicación La mayor parte de los sistemas reciben eventos de entrada externa, los cuales generalmente incluyen una interfaz gráfica para el usuario (IGU) operado por una persona. Otros medios de entrada son los mensajes externos —entre ellos un comutador de telecomunicaciones para procesar llamadas— o las señales procedentes de sensores como sucede en los sistemas de control de procesos.

En todos los casos, si se recurre a un diseño orientado a objetos, hay que elegir los controladores que manejen esos eventos de entrada. Este patrón ofrece una guía para tomar decisiones apropiadas que generalmente se aceptan.

La misma clase controlador debería utilizarse con todos los eventos sistemáticos de un caso de uso, de modo que podamos conservar la información referente al estado del caso. Esta información será útil —por ejemplo— para identificar los eventos del sistema fuera de secuencia (entre ellos, una operación *efectuarPago* antes de *terminarVenta*). Puede emplearse varios controladores en los casos de uso.

Un defecto frecuente al diseñar controladores consiste en asignarles demasiada responsabilidad. Normalmente un controlador debería delegar a otros objetos el trabajo que ha de realizarse mientras coordina la actividad. Favor de consultar la sección *Problemas y soluciones* donde se dan más detalles al respecto.

La primera categoría de controlador es un controlador de fachada que representa al “sistema” global. Es una clase que, para el diseñador, representa de alguna manera al sistema entero. Podría tratarse de una unidad física, como una clase *TPDV*, *InterruptordeTelecomunicaciones* o *Robot*; una clase que representa todo el sistema de software, como *SistemadeInformacionAlmenudeo*, *SistemadeManejodeMensajes* o *ControladordeRobot*, o cualquier otro concepto que, por decisión del diseñador, denote el sistema global.

Los controladores de fachada son adecuados cuando el sistema sólo tiene unos cuantos eventos o cuando es imposible redirigir los mensajes de los eventos del sistema a otros controladores, como sucede en un sistema de procesamiento de mensajes.

Si se recurre a la cuarta categoría de controlador —un “manejador artificial de casos de uso”—, habrá entonces un controlador para cada caso. Adviértase que éste no es un objeto del dominio; es un concepto artificial cuyo fin es dar soporte al sistema (una *fabricación pura*, en términos de los patrones de GRASP). Por ejemplo, si la aplicación del punto de venta contiene casos de uso como “Comprar Productos” y “Devolver Productos”, habrá una clase *ManejadordeComprarProductos* y una clase *ManejadordeDevolverProductos*.

¿Cuándo deberíamos escoger un controlador de casos de uso? Es una alternativa que debe tenerse en cuenta si el hecho de asignar las responsabilidades en cualquiera de las otras opciones de controlador genera diseños de baja cohesión o alto acoplamiento. Esto ocurre generalmente cuando un controlador empieza a “saturarse” con demasiadas responsabilidades. Un controlador de casos de uso constituye una buena alternativa cuando hay muchos eventos de sistema entre varios procesos: asigna su manejo a clases individuales controlables, además de que ofrece una base para reflexionar sobre el estado del proceso actual.

Un corolario importante del patrón Controlador es que los objetos externos de conexión (por ejemplo, los objetos ventana, los applets —o pequeñas aplicaciones—) y la capa de presentación no deberían tener la responsabilidad de llevar a cabo los eventos del sistema.

Dicho con otras palabras, las operaciones del sistema —las cuales reflejan los procesos de la empresa o el dominio— deberían manejarse en la capa de dominio de los objetos y no en las de interfaz, presentación o aplicación. Véase un ejemplo en la siguiente sección titulada *Problemas y Soluciones*.

- Beneficios**
- *Mayor potencial de los componentes reutilizables.* Garantiza que la empresa o los procesos de dominio sean manejados por la capa de los objetos del dominio y no por la de la interfaz. Desde el punto de vista técnico, las responsabilidades del controlador podrían cumplirse en un objeto de interfaz, pero esto supone que el código del programa y la lógica relacionada con la realización de los procesos del dominio puro quedarían incrustados en los objetos interfaz o ventana. Un diseño de interfaz-como-controlador reduce la posibilidad de reutilizar la lógica de los procesos del dominio en aplicaciones futuras, por estar ligada a una interfaz determinada (por ejemplo, un objeto similar a una ventaja) que rara vez puede utilizarse en otras aplicaciones. En cambio, el hecho de delegar a un controlador la responsabilidad de la operación de un sistema entre las clases del dominio soporta la reutilización de la lógica para manejar los procesos afines del negocio en aplicaciones futuras.
 - *Reflexionar sobre el estado del caso de uso.* A veces es necesario asegurarse de que las operaciones del sistema sigan una secuencia legal o poder razonar sobre el estado actual de la actividad y las operaciones en el caso de uso subyacente. Por ejemplo, tal vez deba garantizarse que la operación *efectuarPago* no ocurra mientras no se concluya la operación *terminarVenta*. De ser así, esta información sobre el estado ha de capturarse en alguna parte; el controlador es una buena opción, sobre todo si se emplea a lo largo de todo el caso (cosa que recomendamos).

Problemas y soluciones

Controladores saturados

Una clase controlador mal diseñada presentará baja cohesión: está dispersa y tiene demasiadas áreas de responsabilidad; recibe el nombre de **controlador saturado**. Entre los signos de saturación podemos mencionar los siguientes:

- Hay una sola clase controlador que recibe todos los eventos del sistema y éstos son excesivos. Tal situación suele ocurrir si se escoge un controlador de papeles o de fachada.
- El controlador realiza él mismo muchas tareas necesarias para cumplir el evento del sistema, sin que delegue trabajo. Esto suele constituir una violación de los patrones Experto y Alta Cohesión.
- Un controlador posee muchos atributos y conserva información importante sobre el sistema o dominio —información que debería haber sido redistribuida entre otros objetos— y también duplica la información en otra parte.

Hay varias formas de resolver el problema de un controlador saturado, entre las que se encuentran las siguientes:

1. Agregar más controladores: un sistema no necesariamente debe tener uno solamente. Además de los controladores de fachada, recomendamos emplear los controladores de papeles o los de casos de uso. Considere, por ejemplo, una aplicación con

muchos eventos sistémicos como podría serlo el sistema de reservaciones de una línea aérea. Puede incluir los siguientes controladores:

Controladores de papeles	Controladores de casos de uso
Agente de Reservaciones	Manejador de Hacer Reservación
Programador de Vuelos	Manejador Administración de Programación
Analista de Tarifas	Manejador Administración de Tarifas

2. Diseñe el controlador de modo que delegue fundamentalmente a otros objetos el desempeño de las responsabilidades de la operación del sistema.

Advertencia: los controladores de papeles pueden conducir a la obtención de diseños deficientes

Asignar una responsabilidad a un objeto de papeles humanos en una forma que imite lo que ese papel realiza en el mundo real (por ejemplo, el objeto de software *Cajero* que maneja *efectuarPago*) es aceptable, a condición de que el diseñador conozca perfectamente los posibles riesgos y los evite. En particular, se corre el peligro de generar un controlador poco coherente de papeles que no delegue. Un buen diseño orientado a objetos “les da vida”, al asignarles responsabilidades, aunque representen cosas inanimadas del mundo real. Si escoge un controlador de papeles, no caiga en la trampa de diseñar objetos de tipo humano que realicen todo el trabajo; opte más bien por delegar.

En términos generales, aconsejamos usar poco los controladores de papeles.

La capa de presentación no maneja los eventos del sistema

Un corolario importante —lo repetimos— del patrón Controlador es que los objetos de la interfaz (por ejemplo, objetos de ventanas, applets) y la capa de presentación no deberían encargarse de manejar los eventos del sistema. En otras palabras, las operaciones de éste —que reflejan los procesos de la empresa o del dominio— han de realizarse en la capa del dominio de objeto y no en la de interfaz, la de presentación ni la de aplicación.

Un ejemplo es un diseño en Java que se sirve de un applet para mostrar la información.

Suponga que la aplicación del punto de venta tiene una ventana que muestra la información de ventas y que capture las operaciones del cajero. Con un patrón Controlador, la figura 18.17 describe gráficamente una relación aceptable entre el applet, el Controlador y otros objetos de la parte del sistema correspondiente al punto de venta (con simplificaciones).

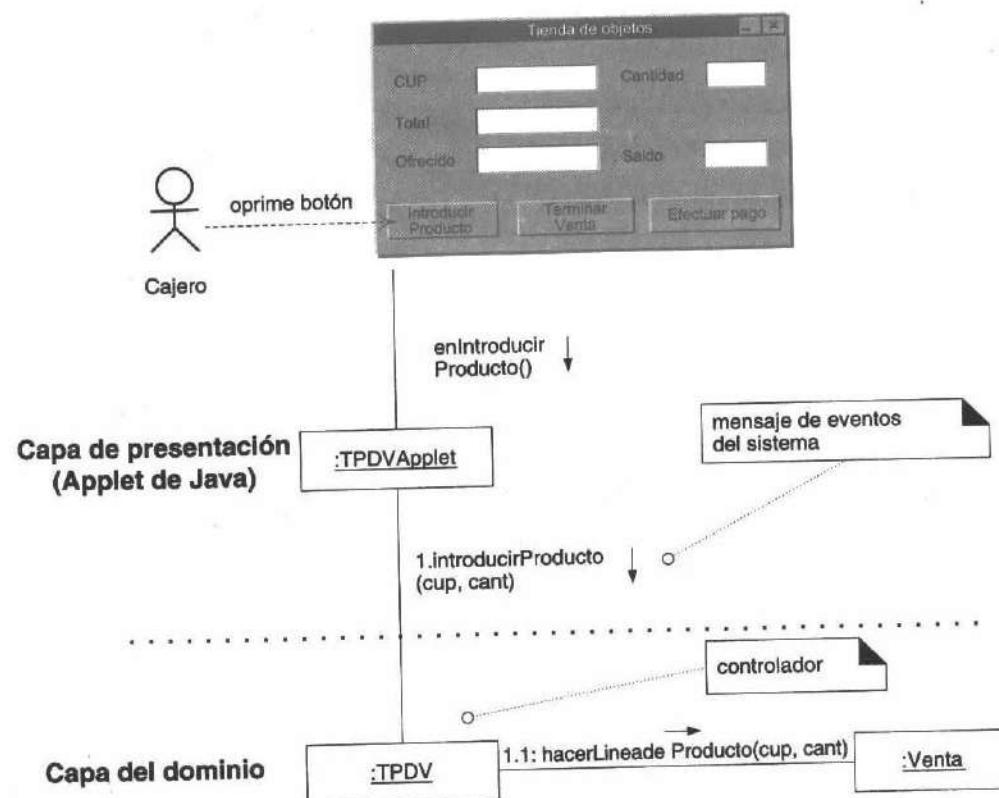


Figura 18.17 Acoplamiento adecuado de la capa de presentación a la capa del dominio.

Nótese que la clase *TPDVApplet* —parte de la capa de presentación— transmite un mensaje *introducirProducto* al objeto *TPDV*. No intervino en el procesamiento de la operación ni la decisión de cómo manejarla, el applet se limitó a delegarla a la capa del dominio.

El potencial de reutilización aumenta al asignar, mediante el patrón Controlador, la responsabilidad de las operaciones del sistema a los objetos situados en la capa del dominio y no en los soportes de la capa de presentación. Si un objeto capa de interfaz (como *TPDVApplet* manejó una operación del sistema —que representa parte de un proceso de negocios—, la lógica de la operación quedará contenida en una interfaz (de tipo ventana, por ejemplo), que tiene pocas posibilidades de reutilización por su acoplamiento con una interfaz y aplicaciones específicas.

En consecuencia, no recomendamos el diseño de la figura 18.18.

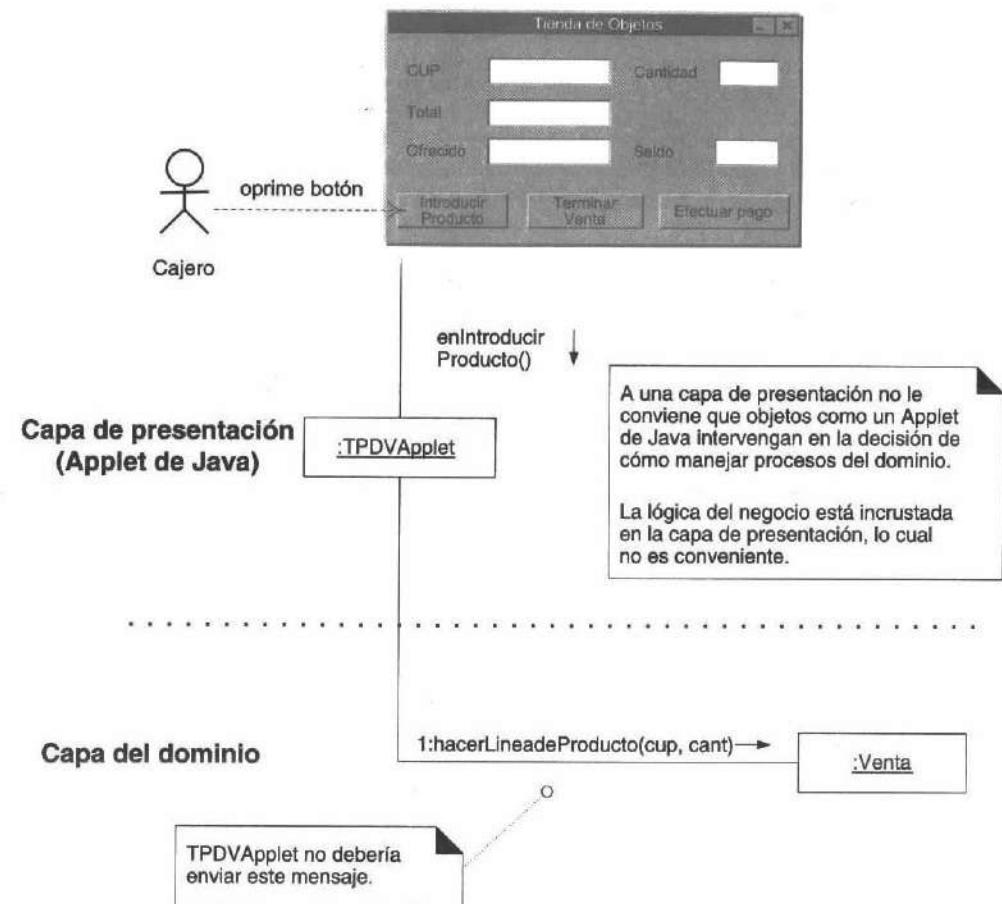


Figura 18.18 Acoplamiento inadecuado de la capa de presentación a la capa del dominio.

Asignar la responsabilidad de la operación del sistema a un controlador del objeto dominio facilita volver a utilizar la lógica del programa que soporte el proceso afín del negocio en aplicaciones futuras. Facilita asimismo desconectar la capa de interfaz y usar otro esquema u otra tecnología de interfaz o bien ejecutar el sistema en un modo "lotes" fuera de línea.

Sistemas del manejo de mensajes y el patrón Comando

Muchas aplicaciones no cuentan con una interfaz para el usuario y, en cambio, reciben mensajes provenientes de algún otro sistema externo; son sistemas de manejo de mensajes. Un conmutador de telecomunicaciones es un ejemplo muy conocido. El mensaje puede estar codificado en un registro, en una secuencia de bytes sin procesar o en un mensaje "real" dirigido a un objeto, si se usa un mecanismo de comunicación interprocesos orientado a objetos como CORBA.

En una aplicación con una interfaz para usuario (una ventana, por ejemplo), la ventana puede decidir cuál será el objeto controlador. Varias ventanas pueden colaborar con los controladores, sobre todo si se recurre a un controlador de casos de uso.

En cambio, el diseño de la interfaz y del controlador es diferente en una aplicación de manejo de mensajes. Tratándose de un caso simple, recomendamos el siguiente diseño. Si el lector desea un diseño más complejo y de muchas características, consulte el patrón Emisor-Receptor en [BMRSS96].

Para manejar los mensajes de eventos del sistema en un sistema de este tipo:

1. Defina un solo controlador para todos los mensajes de eventos; puede ser un controlador de fachada o un controlador individual de casos de uso, con un nombre como *ManejadordeMensajes*.
2. Use el patrón Comando [GHJV95] para contestar la consulta.

El patrón Comando especifica la definición de una clase en cada mensaje o comando, todas ellas con el método *ejecutar*. El controlador creará una instancia Comando correspondiente al mensaje del evento del sistema y le enviará un mensaje *ejecutar*. Las clases comando cuentan con un método único *ejecutar* que especifica las acciones de él, como se aprecia en la figura 18.19.¹

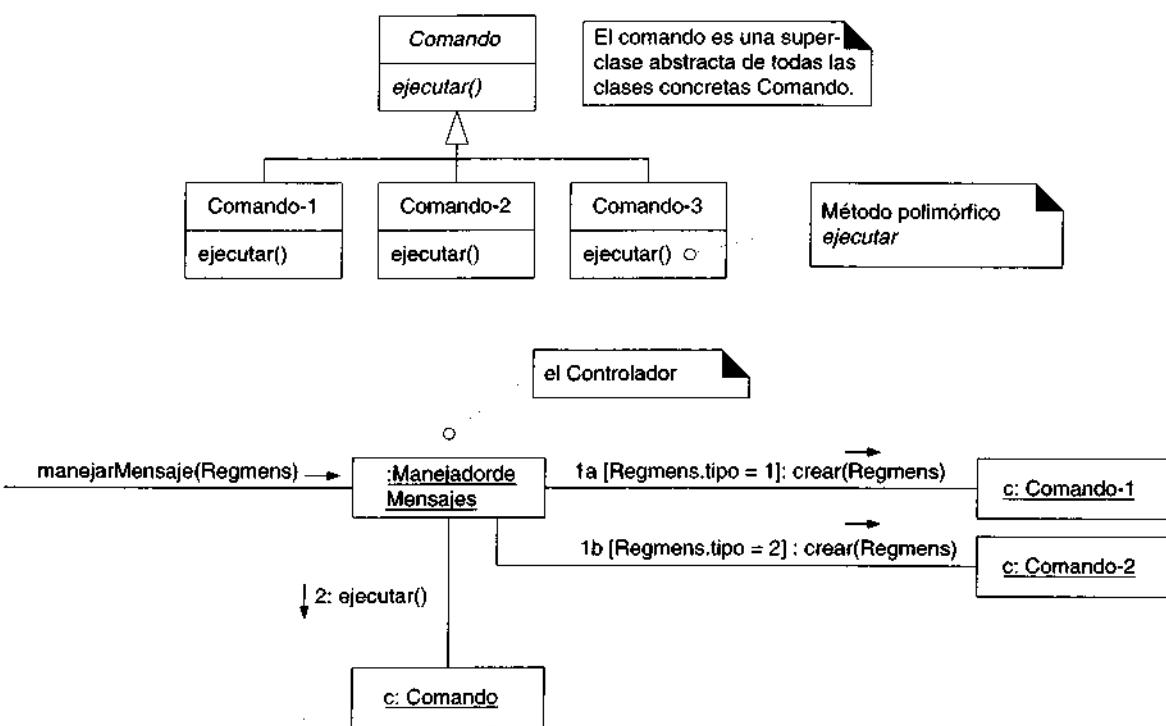


Figura 18.19 El controlador y el uso del patrón Comando.

¹ También es posible un Comando Abstracto *interfaz* en contraste con una superclase abstracta.

Patrones relacionados

- **Comando:** en un sistema de manejo de mensajes; un objeto aislado Comando puede representar y manejar los mensajes [GHJV95].
- **Fachada:** la selección de un objeto que represente un sistema u organización global para que sea controlador lo hace un tipo Fachada [GHJV95].
- **Emisor-Receptor:** éste es un patrón de Siemens [BMRSS96] útil en los sistemas de manejo de mensajes.
- **Capas:** éste es un patrón Siemens [BRMSS96]. Colocar la lógica del dominio en su capa y no en la capa de presentación es parte del patrón Capas.
- **Artefacto puro:** es otro patrón de GRASP. El artefacto puro es una clase artificial, no un concepto del dominio. Un controlador de casos de uso es un tipo de Artefacto Puro.

18.14 Responsabilidades, representación de papeles y las tarjetas CRC

Aunque no sean una parte formal del lenguaje UML, las **tarjetas CRC** [BC89] (siglas en inglés de *Class-Responsibility-Collaborator* Clase-Responsabilidad-Colaborador) son otra herramienta con la cual a veces se asignan las responsabilidades y se indica una colaboración con otros objetos. Fueron inventadas por Kent Beck y Ward Cunningham, principales promotores para que los diseñadores de objetos piensen en términos más abstractos sobre la asignación de responsabilidades y de las colaboraciones.

Las tarjetas CRC son tarjetas de índices, una por cada clase, sobre las cuales se abrevian las responsabilidades de la clase y se anota una lista de los objetos con los que colaboran para desempeñarlas. Suelen elaborarse en una sesión de grupos pequeños donde los participantes **representan papeles** de las diversas clases. Cada grupo tiene las tarjetas CRC correspondientes a las clases cuyo papel está desempeñando.

La representación de papeles hace divertido aprender a pensar en función de objetos y de responsabilidades. Pero las tarjetas basadas en texto tienen una capacidad limitada para registrar las colaboraciones en forma exhaustiva. Los diagramas de colaboración y los de clase describen mejor las conexiones entre los objetos y el contexto general.

Recomendamos por su utilidad las actividades grupales de diseño dirigidas a asignar responsabilidades y a representar papeles; los resultados pueden anotarse después en las tarjetas CRC o en diagramas. Las tarjetas son un método de registrar los resultados de la asignación de responsabilidades y de las colaboraciones. El registro es más satisfactorio si se utiliza este tipo de diagramas. El valor real no está constituido por las tarjetas, sino por el análisis de la asignación de responsabilidades y por la actividad relacionada con la representación de papeles.

DISEÑO DE UNA SOLUCIÓN CON OBJETOS Y PATRONES

Objetivos

- Aplicar los patrones GRASP para asignar responsabilidades a las clases.
- Utilizar la notación de UML en los diagramas de colaboración para describir gráficamente el diseño de la interacción de objetos.

19.1 Introducción

En este capítulo explicaremos cómo crear los diagramas de interacción (en este caso, diagramas de colaboración) para la aplicación del punto de venta. Estudiaremos también su relación con los artefactos anteriores, entre ellos los contratos y el modelo conceptual. Prestamos especial atención a la aplicación de los patrones GRASP para alcanzar una solución bien diseñada.

En el presente capítulo se exponen los principios, mediante el ejemplo del punto de venta, en virtud de los cuales un diseñador orientado a objetos asigna las responsabilidades y establece las interacciones de ellos, una habilidad fundamental en el desarrollo orientado a objetos.

En la práctica, los diseñadores se percatan de que la preparación de los diagramas de interacción es uno de los pasos más lentos (y reditables). Adviértase además lo siguiente:

La asignación de responsabilidades y la elaboración de los diagramas de interacción representan uno de los pasos más importantes en la fase del diseño.

Con toda intención ofrecemos una exposición detallada: se busca exemplificar de modo exhaustivo que no hay decisiones "mágicas" ni injustificables en el diseño orientado a objetos. Podemos explicar y aprender razonablemente otras asignaciones de responsabilidades y la elección de interacciones de objetos.

Las siguientes indicaciones le servirán para repasar los pasos de la preparación de un diagrama de interacción.

Para elaborar diagramas de interacción:

1. Prepare un diagrama individual para cada operación del sistema en el paso iterativo actual.
 - Por cada evento del sistema construya un diagrama que lo incluya como mensaje inicial.
2. Si el diagrama se vuelve complejo, divídalo en otros más pequeños.
3. Con las responsabilidades contractuales, las poscondiciones y la descripción del caso de uso como punto de partida, diseñe un sistema de objetos interactuantes para que realicen las tareas. Aplique los patrones GRASP y otros para elaborar un buen diseño.

19.2 Diagramas de interacción y otros artefactos

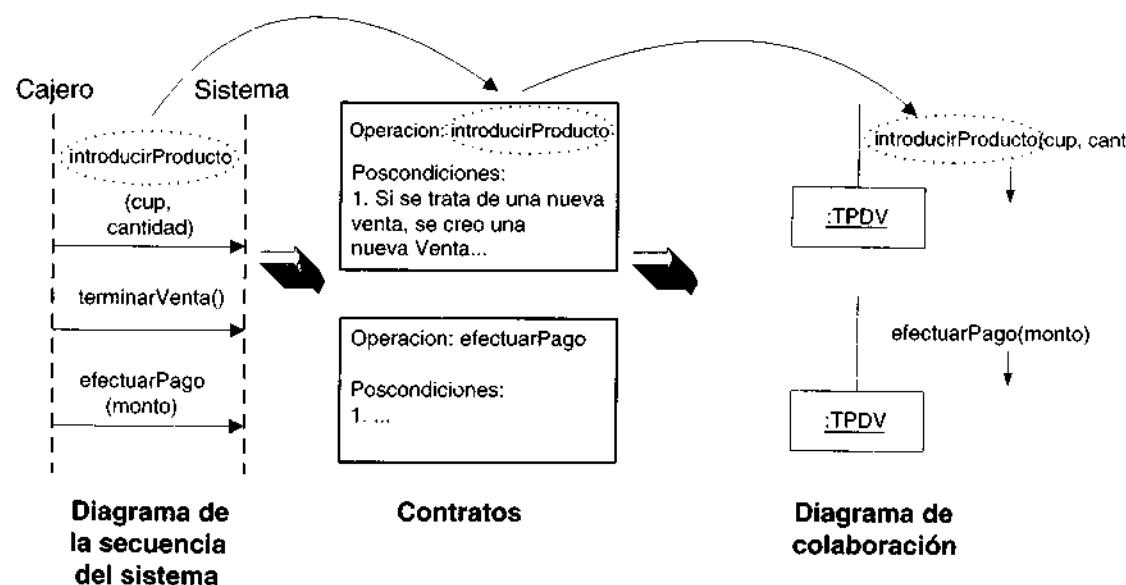


Figura 19.1 Relación entre artefactos desde la perspectiva de los eventos de un sistema.

La figura 19.1 muestra visualmente la relación entre los artefactos.

- Los casos de uso indican los eventos del sistema que se muestran explícitamente en los diagramas de la secuencia de él.
- La mejor conjetura inicial sobre el efecto producido por los eventos del sistema se describe en los contratos de sus operaciones.
- Los eventos del sistema representan mensajes iniciadores de los diagramas de interacción que describen visualmente cómo los objetos interactúan para realizar las tareas en cuestión.
- Los diagramas de interacción incluyen la interacción de los mensajes entre los objetos que se definen en el modelo conceptual y otras clases de objetos.

19.2.1 Los diagramas de interacción y los eventos del sistema

En la iteración actual de la aplicación del punto de venta estamos considerando dos casos de uso y sus eventos sistemáticos asociados:

- Comprar Productos
 - introducirProducto
 - terminarVenta
 - efectuarPago
- Inicio
 - inicio

Por cada evento del sistema construya un diagrama de colaboración cuyo mensaje inicial sea el de sus eventos.

Habrá, pues, cuatro diagramas de interacción por lo menos: uno por cada evento del sistema. Con el patrón Controlador, la clase `TPDV` puede escogerse como controlador para que maneje los eventos (figura 19.2).

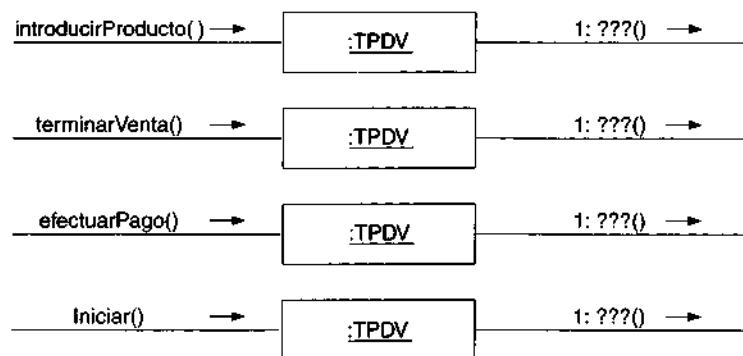


Figura 19.2 Eventos del sistema.

19.2.2 Los diagramas de interacción y los contratos

Con las responsabilidades y las poscondiciones del contrato, así como también los casos de uso como punto de partida, diseñe un sistema de objetos interactuantes para que realicen las tareas.

Por cada operación del sistema existe un contrato que estipula lo que deben lograr las operaciones. Por ejemplo:

Contrato	
Nombre:	introducirProducto (cup : número cantidad : entero).
Responsabilidades:	Capturar (registrar) la venta de un producto y agregarla a la venta. Desplegar la descripción del producto y su precio.
Poscondiciones:	

- Si se trata de una nueva venta, se creó una `Venta` (*creación de instancia*).
- Si se trata de una nueva venta, se asoció la nueva `Venta` a `TPDV` (*asociación formada*).
- y así sucesivamente...

En cada contrato revisamos los cambios de estado de las responsabilidades y las poscondiciones, diseñando al mismo tiempo interacciones de mensajes —incluidos en el diagrama de colaboración— para atender los requerimientos. Nótese que, en caso de omitir la preparación del contrato, de todos modos deberíamos elaborar los diagramas de interacción retornando a los casos de uso y reflexionando sobre lo que debemos lograr. No obstante, los contratos organizan y aislan la información en un formato funcional, al mismo tiempo que estimulan la investigación en la fase de análisis y no en la de diseño.

Por ejemplo, en esta operación parcial del sistema `introducirProducto`, en la figura 19.3 incluimos un diagrama parcial de colaboración que satisface el cambio de estado de la creación de `Venta`.

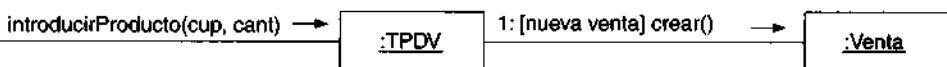


Figura 19.3 Diagrama parcial de colaboración.

19.2.3 Las poscondiciones no son más que una estimación

Los diagramas pueden prepararse con el propósito de cumplir las poscondiciones del contrato. Pero es indispensable reconocer que las poscondiciones definidas de antemano no son sino una excelente conjeta o estimación inicial de lo que se pretende alcanzar. Tal vez no sean muy exactas. Lo mismo sucede con el modelo conceptual: es un punto de partida que contendrá errores y omisiones. En los contratos debemos ver un mero punto de partida para establecer lo que se hará, pero sin sentirnos obligados por ellos. Lo más probable es que algunas poscondiciones no sean necesarias y que haya algunas tareas por realizar que aún no hemos descubierto. Una ventaja del desarrollo iterativo radica en que brinda un soporte espontáneo a la detección de nuevos resultados del análisis y del diseño durante las fases de solución y de construcción.

El desarrollo iterativo se propone alcanzar un nivel “razonable” de información en la fase de análisis, llenando después los detalles en la del diseño. De modo análogo, busca también lograr un nivel “razonable” de resultados del diseño durante la fase de diseño, llenando después los detalles en la de implementación (codificación). La definición de “razonable” es, naturalmente, cuestión de opinión; en este libro procuramos presentar un grado razonable de actividad en cada paso.

19.2.4 Los diagramas de colaboración y el modelo conceptual

Algunos de los objetos que interactúan a través de mensajes en los diagramas de colaboración provienen del modelo conceptual. En parte, la elección de la asignación acertada de las responsabilidades mediante los patrones GRASP se funda en la información contenida en el modelo conceptual. Según mencionamos en páginas anteriores, el actual modelo conceptual difícilmente será perfecto: se prevén errores y omisiones. Usted descubrirá conceptos que antes no estaban incluidos, omitirá los que ya habían sido identificados y hará lo mismo con las asociaciones y con los atributos.

19.3 Modelo conceptual del punto de venta

A manera de recordatorio incluimos la figura 19.4, donde el lector podrá examinar el modelo conceptual en la aplicación al punto de venta. Se trata de objetos que pueden participar en las interacciones referentes a ellos. ¿Acaso los únicos modelos de los objetos en interacción son los que figuran en este modelo? De ninguna manera; en la fase de diseño conviene descubrir otros tipos que no se incluyeron en los análisis anteriores. Y cuando se descubran los incorporaremos al modelo.

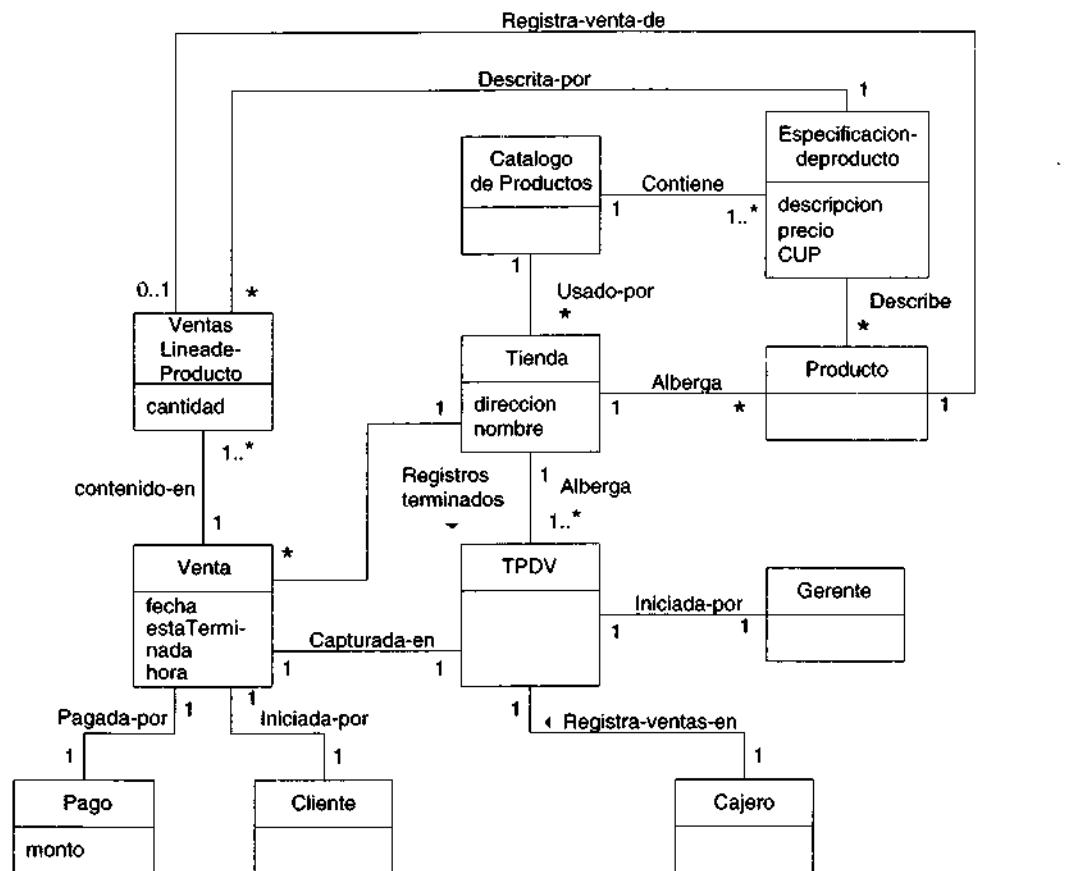


Figura 19.4 Modelo conceptual del punto de venta.

19.4 Diagramas de colaboración para la aplicación TPDV

En esta sección estudiaremos las decisiones y selecciones hechas al preparar un diagrama de colaboración a partir de los patrones GRASP. Nuestras explicaciones son deliberadamente pormenorizadas, pues queremos demostrar que no se da una adhesión irreflexiva cuando se crean diagramas de colaboración bien diseñados: su construcción se basa más bien en principios justificables.

19.5 El diagrama de colaboración: introducirProducto

La operación del sistema *introducirProducto* ocurre cuando un cajero captura el CUP (código universal de producto) y la cantidad del objeto a comprar. A continuación se incluye integralmente un contrato; recuerde el lector que se trata de una estimación inicial y quizás no sea completa:

Contrato	
Nombre:	introducirProducto (cup : número cantidad: entero).
Responsabilidades:	Capturar (registrar) la venta de un producto y agregarla a la venta. Desplegar la descripción del producto y su precio.
Tipo:	Sistema.
Referencias cruzadas:	Funciones del sistema: R1.1, R1.3, R1.9.
Notas:	Casos de uso: Comprar productos.
Excepciones:	Usar el acceso superrápido a la base de datos.
Salida:	Si el CUP no es válido, indicar que se cometió un error.
Precondiciones:	El sistema conoce el CUP.
Poscondiciones:	<ul style="list-style-type: none"> ■ Si se trata de una nueva venta, se crea una <i>Venta</i> (<i>creación de instancia</i>) ■ Si se trata de una nueva venta, se asocia la nueva <i>Venta</i> a la instancia <i>TPDV</i> (<i>formación de asociación</i>) ■ Se creó una instancia <i>VentasLineadeProducto</i> (<i>creación de instancia</i>) ■ Se asoció la instancia <i>VentasLineadeProducto</i> a la instancia <i>Venta</i> (<i>formación de asociación</i>) ■ Se asignó a <i>VentasLineadeProducto.cantidad</i> el valor en <i>cantidad</i> (<i>modificación de atributo</i>) ■ Se asoció la instancia <i>VentasLineadeProducto</i> a <i>EspecificaciondeProducto</i> basado esto en la correspondencia del CUP (<i>formación de asociación</i>)

Se construirá un diagrama de colaboración que cumple con las poscondiciones de *introducirProducto*. Como se señaló en el capítulo dedicado a los patrones GRASP, en cada elección de un mensaje se requiere asignar una responsabilidad; los patrones GRASP servirán para escoger y justificar los mensajes que se envíen a una clase de objetos.

19.5.1 Selección de la clase Controlador

Nuestra primera decisión se referirá a la elección del controlador que se encargue del mensaje de las operaciones del sistema *introducirProducto*. Desde el punto de vista de este patrón, disponemos de las siguientes opciones:

representar todo el sistema (controlador de fachada)	<i>TPDV</i> , o una clase nueva como <i>SistemaAlMenudeo</i>
representar la empresa o la organización total (controlador de fachada)	<i>Tienda</i>
representar algo del mundo real que sea activo (por ejemplo, el papel de una persona) y que pueda participar en la tarea (controlador de papeles)	<i>Cajero</i>
representa un manejador artificial de todas las operaciones del sistema en un caso de uso (controlador de casos de uso).	<i>ManejadordeComprar-Productos</i>

Es satisfactorio elegir un controlador de fachada como *TPDV*, si el sistema ofrece pocas operaciones y si ese patrón no asume demasiadas responsabilidades (en otras palabras, si empieza a perder cohesión). Escoger un controlador de papeles o de casos de uso es una decisión acertada, cuando el sistema tiene demasiadas operaciones y queremos distribuir las responsabilidades para que las clases controlador no se aborren ni pierdan su orientación central (o sea, que sean cohesivas). En este caso, *TPDV* será suficiente porque el sistema tiene pocas operaciones.

Es importante darse cuenta que se trata de una instancia de *TPDV* de un objeto en "territorio de software". No es una terminal real de punto de venta, sino una abstracción de software que representa el registro.

Así pues, el diagrama de colaboración de la figura 19.5 comienza enviando el mensaje *introducirProducto* a una instancia de *TPDV* mediante un CUP y un parámetro de cantidad.

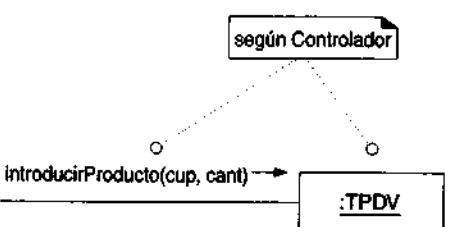


Figura 19.5 Aplicación del patrón Controlador de GRASP.

19.5.2 Presentación visual de la descripción y del precio

En virtud de un principio del diseño denominado **separación de modelo-vista**, no compete a los objetos del dominio (como *TPDV* o *Venta*) comunicarse con la capa de interfaz para el usuario (las ventanas gráficas, por ejemplo). De ahí que por ahora no nos ocupemos de la descripción del producto ni de su precio.

Lo único que se requiere en lo tocante a las responsabilidades del despliegue de la información es conocer los datos, como se verá en este caso.

19.5.3 Realización de una nueva venta

Es necesario examinar los requerimientos, expresados en los contratos, que han de atenderse en el software para lograr una ejecución correcta del sistema. Dos de las poscondiciones contractuales establecen lo siguiente:

- Si se trata de una nueva venta, se creó una *Venta* (*creación de instancia*).
- Si se trata de una nueva venta, se asoció la nueva *Venta* a *TPDV* (*asociación formada*).

Lo anterior significa una responsabilidad en la creación, y el patrón Creador de GRASP indica la asignación de la responsabilidad de generar una clase que agregue, contenga o registre la que va a ser creada.

Al analizar el modelo conceptual, o sea al reflexionar sobre los objetos del dominio, se descubre que en una instancia *TPDV* podemos ver el registro de una *Venta*;¹ por tanto, *TPDV* es un buen candidato para crear una *Venta*. Y al hacer que la cree, con el tiempo podremos asociar las dos, de modo que en operaciones futuras *TPDV* será una referencia a la instancia actual *Venta*.

Además de lo que acabamos de decir, cuando se crea la *Venta*, hay que generar una colección vacía (un contenedor como un *Vector* de Java) para que registre todas las instancias futuras *VentasLineadeProducto* que vayamos agregando. La instancia *Venta* le dará mantenimiento a esta colección, lo cual implica que según el patrón Creador la *Venta* es idónea para crearla.

En conclusión, la instancia *TPDV* crea la *Venta* y ésta, a su vez, produce una colección vacía, que está representada por un multiobjeto en el diagrama de colaboración.

¹ Nótese que *TPDV* es más o menos sinónimo de *registro*: un registro de ventas.

Entonces, el diagrama de colaboración de la figura 19.6 ilustra el diseño.

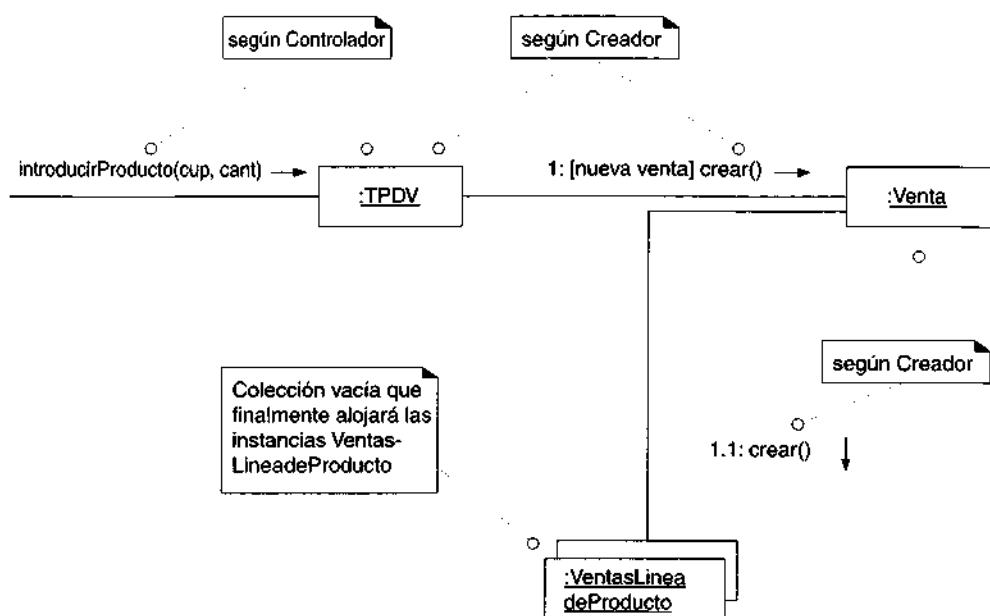


Figura 19.6 Creación de ventas.

19.5.4 Creación de una nueva instancia *VentasLineadeProducto*

Algunas otras poscondiciones contractuales de *introducirProducto* establecen lo siguiente:

- Se creó una instancia *VentasLineadeProducto* (*creación de instancia*).
- Se asoció *VentasLineadeProducto* a la *Venta* (*asociación formada*).
- Se asignó el valor a *VentasLineadeProducto.cantidad* de *cantidad* (*modificación de atributo*).
- Se asoció la instancia *VentasLineadeProducto* con una *EspecificacioneProduct* a partir de la correspondencia en el *CUP* (*asociación formada*).

Lo anterior indica la responsabilidad de crear una instancia *VentasLineadeProducto*. El análisis del diagrama de clases revela que una *Venta* contiene los objetos *VentasLineadeProducto*; por tanto, el patrón Creador es un buen candidato para generar la línea de producto. Y al hacer que la *Venta* cree *VentasLineadeProducto*, podemos asociar la *Venta* con ésta a lo largo del tiempo al guardar la nueva instancia en su colección de línea de producto. Las poscondiciones indican que la nueva instancia *VentasLineadeProducto* necesita una cantidad al ser creada; por eso, la instancia *TPDV* ha de transmitirla a la *Venta*, que a su vez la transfiera como parámetro en el mensaje *crear*.

Así pues, basados en el patrón Creador a la *Venta* se le envía un mensaje *hacerLineadeProducto* para que produzca una instancia *VentasLineadeProducto*. La *Venta* crea una *VentasLineadeProducto* y luego almacena la nueva instancia en su colección permanente.

Los parámetros del mensaje *hacerLineadeProducto* son la *cantidad*, para que *VentasLineadeProducto* pueda registrarla, y también la *EspecificacioneProduct* que concuerde con el *CUP*.

19.5.5 Obtención de una instancia *EspecificacioneProduct*

Necesitamos asociar la instancia *VentasLineadeProducto* a *EspecificacioneProduct* que concuerde con el *CUP* que se recibe. Esto significa que se requiere recuperar una *EspecificacioneProduct* basada en la correspondencia del *CUP*.

Antes de ver *cómo* lograr la consulta, es muy importante determinar *quién* se encargará de ella. Un primer paso de gran utilidad es el siguiente:

Comience asignando las responsabilidades definiéndolas con claridad.

Replantearmos el problema en los siguientes términos:

¿Quién debería asumir la responsabilidad de conocer una instancia *EspecificacioneProduct* basado en que coincida el valor del *CUP*?

No estamos ante un problema de creación ni ante el de escoger un controlador. En la generalidad de los casos, el patrón Experto de GRASP es el principio que ha de aplicarse en primer lugar. Este patrón indica que el objeto con la información necesaria para desempeñar la responsabilidad debería encargarse de ella. ¿Quién está enterado de todo lo concerniente a *EspecificacioneProduct*? El análisis del diagrama de clases revela que *CatalogodeProductos* contiene lógicamente todas las *EspecificacionesdeProductos* y así, de acuerdo con el patrón Experto, *CatalogodeProductos* es idóneo para asumir esta responsabilidad de consulta.

19.5.6 Visibilidad ante un *CatalogodeProductos*

¿Quién debería enviar el mensaje *especificación* al *CatalogodeProductos* para solicitar una *EspecificacioneProduct*?

Es razonable suponer que una instancia *TPDV* y una de *CatalogodeProductos* fueron creadas durante el caso inicial de uso *Início* y que existe una conexión permanente entre el objeto *TPDV* y el objeto *CatalogodeProductos*. Esta suposición nos permitirá concluir que *TPDV* puede enviar el mensaje *especificación* a *Catálogo de productos*.

Ello significa la existencia de otro concepto en el diseño orientado a objetos: el de visibilidad. La **visibilidad** es la capacidad de un objeto para “ver” o tener una referencia a otro.

Para que un objeto envíe a otro un mensaje, debe ser visible al segundo.

Supondremos que *TPDV* tiene una conexión o referencia permanente a *CatalogodeProductos*; por tanto, es visible para esta instancia y de ahí que pueda enviarle mensajes como *especificación*.

En el siguiente capítulo trataremos más a fondo el tema de la visibilidad.

19.5.7 Recuperación de EspecificacioneProducto a partir de una base de datos

En la versión final de una aplicación real al punto de venta, difícilmente todas las *EspecificacionesdeProducto* estarán en la memoria. Lo más probable es que se encuentren almacenadas en una base de datos relacional o de objetos y que sean recuperadas previa solicitud. No obstante, para no hacer compleja la exposición pospondremos el estudio de los problemas concernientes a la recuperación partiendo de una base de datos. Supondremos que todas las *EspecificacionesdeProducto* se hallan en la memoria. En el capítulo 38 abordaremos el tema del acceso a la base de datos de objetos persistentes.

19.5.8 El diagrama de colaboración introducirProducto

Una vez explicado quién debería crear la instancia *VentasLineadeProducto*, recuperar una *EspecificacioneProducto* y realizar otras tareas, el diagrama de colaboración de la figura 19.7 refleja las decisiones sobre la asignación de responsabilidades y sobre la manera en que los objetos deberían interactuar. Observe que se reflexionó detenidamente para llegar a él, con base en los patrones GRASP; el diseño de las interacciones de los objetos y la asignación de responsabilidades requieren una seria deliberación.

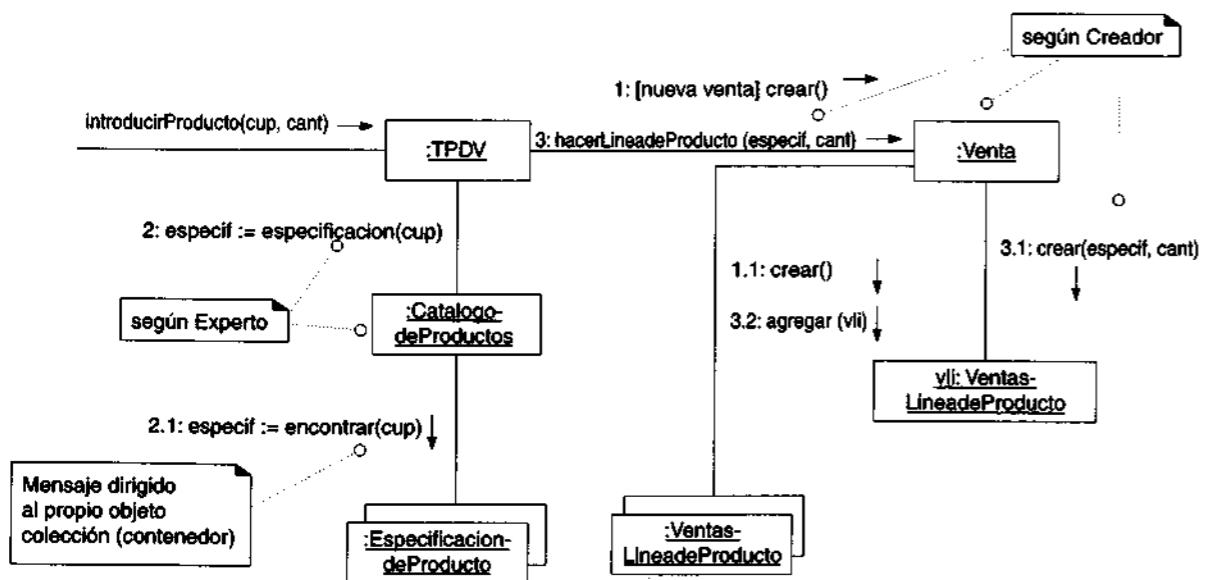


Figura 19.7 El diagrama de colaboración introducirProducto.

19.5.9 Mensajes a multiobjetos

Adviértase lo siguiente: la interpretación predeterminada de un mensaje enviado a un multiobjeto es que se envía implicitamente a todos los elementos de la colección/contenedor, pero también puede interpretarse como un mensaje dirigido al propio objeto colección. Esto ocurre especialmente en las operaciones genéricas de colección como *encontrar* y *agregar*. Por ejemplo, en el diagrama de colaboración *introducirProducto*:

- El mensaje *encontrar* (2.1) enviado al multiobjeto *EspecificacioneProducto* se dirige una vez a la estructura de datos colección representada por el multiobjeto (un *Vector* en Java).
- El mensaje *crear* (1.1) enviado al multiobjeto *VentasLineadeProducto* tiene por objeto generar la estructura de datos colección representada por el multiobjeto (un *Vector* en Java, por ejemplo); su finalidad no es crear una instancia de la clase *VentasLineadeProducto*.
- El mensaje *agregar* (3.2) enviado al multiobjeto *VentasLineadeProducto* tiene por objeto incorporar un elemento a la estructura de datos colección representada por el multiobjeto.

19.6 El diagrama de colaboración: terminarVenta

La operación sistemática *terminarVenta* ocurre cuando un cajero oprime un botón para indicar la conclusión de una venta. He aquí el contrato íntegro:

Contrato

Nombre:	terminarVenta().
Responsabilidades:	Registrar que se terminó de capturar los artículos de la venta y presentar visualmente el total de la venta.
Tipo:	Sistema.
Referencias cruzadas:	Funciones del sistema: R1.2.
Notas:	Casos de uso: Comprar Productos.
Excepciones:	Si está efectuándose una venta, indicar que se cometió un error.
Salida:	
Precondiciones:	El sistema conoce el CUP.
Poscondiciones:	
■	Se asignó a <i>Venta.estaTerminada</i> el valor <i>verdadero</i> (modificación de atributo).

Se elaborará un diagrama de colaboración para cumplir las poscondiciones de *terminarVenta*. Como señalamos en el capítulo dedicado a los patrones GRASP, cada decisión sobre mensajes implica asignar una responsabilidad, y los patrones GRASP se emplearán para escogerlos y justificarlos.

19.6.1 Elección de la clase Controlador

Nuestra primera decisión se refiere a la asignación de la responsabilidad del mensaje *terminarVenta* de la operación del sistema. Basándonos en el patrón Controlador de GRASP, como lo hicimos con *introducirProducto*, seguiremos utilizando *TPDV* como controlador.

19.6.2 Establecimiento del atributo *Venta.estaTerminada*

Las poscondiciones contractuales establecen lo siguiente:

- Se ha establecido *Venta.estaTerminada* en *verdadero* (modificación de atributo).

Como siempre, el patrón Experto debería ser el primero a considerar, salvo que se trate de un problema de controlador o de creación (pero no es así).

¿Quién debería encargarse de asignar el valor *verdadero* al atributo *estaTerminada* de la *Venta*?

Basándonos en el patrón Experto, debería ser la propia *Venta*, porque posee el atributo *estaTerminada* y le da mantenimiento. Así, *TPDV* enviará a *Venta* el mensaje *seTermina* para asignarle el valor *verdadero*. Obsérvese en la figura 19.8 el uso de una nota restrictiva que contiene seudocódigo para aclarar el propósito de la operación *darPorTerminada*; esta última se recomienda cuando es necesario explicar los detalles de una operación.

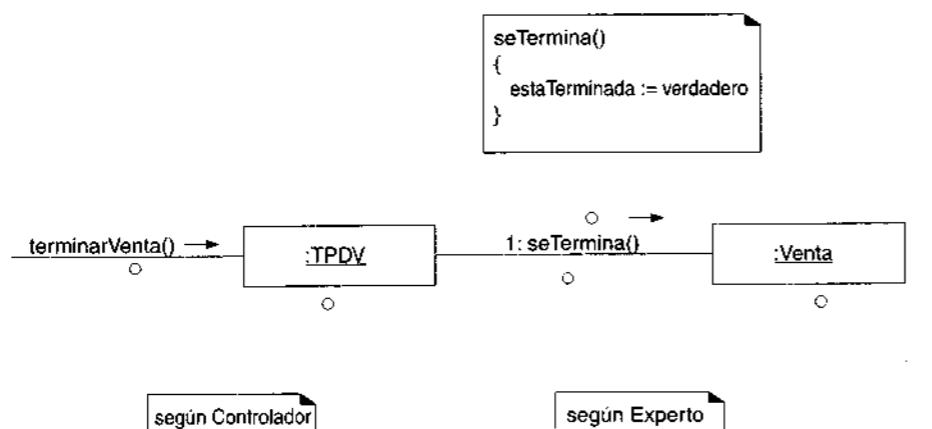


Figura 19.8 Terminación de la captura de un producto.

19.6.3 La presentación visual de información

En las responsabilidades del contrato *terminarVenta* se estipula que el total de la venta deberá mostrarse. Aclaremos la cuestión de quién debería encargarse de presentar visualmente la información, en una ventana gráfica por ejemplo. En síntesis, la capa de

los objetos del dominio —como *Venta*, *Tienda*, etc.— no debiera conocer las ventanas o la capa de presentación ni tener comunicación con ellas. Esto constituye la esencia del patrón *Separación Modelo-Vista*, que se explica en el capítulo 22.

Durante la preparación de los diagramas de colaboración, no se preocupe por mostrar la información salvo en la medida en que se conoce la que se necesita.

Por tanto, prescinda de los requerimientos que requieren presentar información, con excepción de lo siguiente:

Asegúrese de que toda la información que ha de mostrarse visualmente se conozca y pueda obtenerse de los objetos del dominio.

Por ejemplo, el requerimiento anterior indica que algún objeto debe conocer el total de la venta.

19.6.4 Cálculo del total de la venta

Gracias a que contamos con el patrón Separación Modelo-Vista, no es necesario preocuparnos de cómo mostrar el total de la venta, pero debemos cerciorarnos de conocer el total. Obsérvese que ninguna clase conoce en este momento el total; por ello, con un diagrama de colaboración debemos preparar un diseño de las interacciones de los objetos que atienden este requerimiento.

Como siempre, el patrón Experto habrá de ser el primero que consideremos a menos que nos enfrentemos a un problema de controlador o de creación (pero no es así).

Probablemente nos demos cuenta de que la *Venta* debería ser la encargada de conocer su total, pero examinemos el siguiente análisis para hacer muy claro —mediante un ejemplo— el proceso de razonamiento con que se obtiene un Experto.

1. Establezca la responsabilidad:
 - ¿Quién debería asumir la responsabilidad de conocer el total de la venta?
2. Sintetice la información requerida:
 - El total de la venta es la suma de los subtotales de todas las líneas de producto de la venta.
 - Subtotal de la línea de producto de la venta := cantidad de la línea de producto * precio del producto.
3. Incluya la información necesaria para desempeñar esta responsabilidad y las clases que conocen esta información.

<i>EspecificaciondeProducto.precio</i>	<i>EspecificaciondeProducto</i>
<i>VentasLineadeProducto.cantidad</i>	<i>VentasLineadeProducto</i>
todas las <i>VentasLineadeProducto</i> en la <i>Venta</i> actual	<i>Venta</i>

Damos en seguida un análisis pormenorizado:

- ¿Quién debería asumir la responsabilidad de calcular el total de la *Venta*? Según el patrón Experto, tendría que ser la propia *Venta*, porque conoce todo lo concerniente a las instancias *VentasLineadeProducto* cuyos subtotales tienen que sumarse para calcular el total de la venta. Por consiguiente, *Venta* deberá tener la responsabilidad de conocer su total, que se implementó como una operación *total*.
- Para que una *Venta* calcule su total necesita el subtotal de cada instancia *VentasLineadeProducto*. ¿Quién debería asumir la responsabilidad de calcular el subtotal de esa instancia? Según el patrón Experto, debería ser la propia *VentasLineadeProducto*, porque conoce la cantidad y porque *EspecificaciondeProducto* está asociada a ella. Por tanto, *VentasLineadeProducto* tendrá la responsabilidad de conocer su subtotal, que se implementó como una operación *subtotal*.
- Para que *VentasLineadeProducto* calcule su subtotal, necesita el precio de *EspecificaciondeProducto*. ¿Quién debería asumir la responsabilidad de ofrecer este precio? De acuerdo con el patrón Experto, debería ser la propia *EspecificaciondeProducto*, porque encapsula el precio como atributo. Por tanto, *EspecificaciondeProducto* tendrá la responsabilidad de conocer su precio, que se implementó como una operación *precio*.

Aunque el análisis anterior es trivial en este caso y aunque no hace falta un grado de elaboración tan complicada, la misma estrategia con que se localiza un patrón Experto puede y debiera aplicarse en situaciones más difíciles. El lector se dará cuenta de que, una vez aprendidos los patrones GRASP, podrá realizar rápidamente este tipo de razonamiento en forma mental.

19.6.5 El diagrama de colaboración total de *Venta*

Una vez explicado lo anterior, conviene construir un diagrama de colaboración que indique lo que sucede cuando a una *Venta* se le envía un mensaje de *total*. El primer mensaje del diagrama es *total*, pero observe que no es un evento del sistema. De ello extraemos la siguiente observación:

No todo diagrama de colaboración comienza con un mensaje de eventos del sistema; puede empezar con cualquier mensaje cuyas interacciones deseé mostrar el diseñador.

El diagrama de colaboración requerido se incluye en la figura 19.9. Primero, el mensaje de *total* se envía a la instancia *Venta*. Ésta enviará entonces un mensaje de *subtotal* a las instancias relacionadas de *VentasLineadeProducto*. Y a su vez esta instancia enviará un mensaje de *precio* a su correspondiente *EspecificaciondeProducto*.

Dado que las operaciones aritméticas (por lo regular) no se explican gráficamente a través de mensajes, podemos describir los detalles de los cálculos incorporando al diagrama restricciones que definan los cálculos.

¿Quién enviará a la *Venta* el mensaje de *total*? Lo más probable es que lo haga un objeto de la capa de presentación; por ejemplo un Applet de Java.

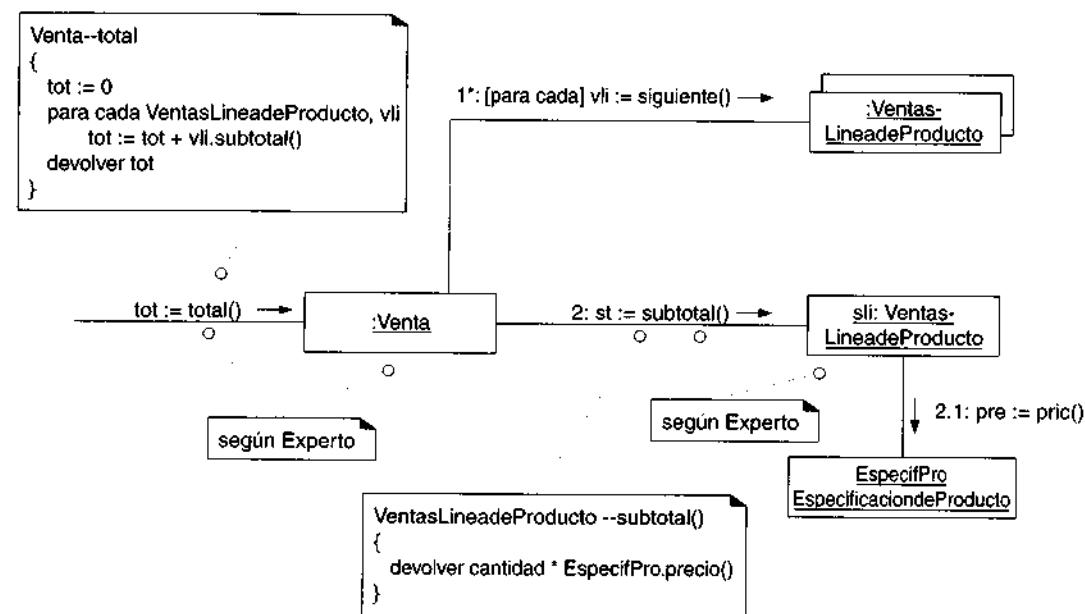


Figura 19.9 El diagrama de colaboración *venta-total*.

19.7 El diagrama de colaboración: efectuarPago

La operación sistemática *efectuarPago* ocurre cuando un cajero captura el efectivo ofrecido como pago. He aquí íntegramente el contrato:

Contrato

Nombre:	efectuarPago (monto: Número o Cantidad).
Responsabilidades:	Registrar el pago, calcular el saldo e imprimir el recibo.
Tipo:	Sistema.
Referencias cruzadas:	Funciones del sistema: R2.1.
Notas:	Casos de uso: Comprar Productos.
Excepciones:	<p>Si la venta no está terminada, indicar que se cometió un error.</p> <p>Si el monto es menor que el total de la venta, indicar que se cometió un error.</p>
Salida:	
Precondiciones:	
Poscondiciones:	
<ul style="list-style-type: none"> ■ Se creó un <i>Pago</i> (creación de instancia). ■ Se asignó a <i>Pago.montoOfrecido</i> el valor del <i>monto</i> (modificación de atributo). ■ Se asoció <i>Pago</i> a <i>Venta</i> (relación formada). ■ Se asoció la <i>Venta</i> a <i>Tienda</i> para agregarla al registro histórico de ventas terminadas (relación formada). 	

Se construirá un diagrama de colaboración para cumplir con las poscondiciones de *efectuarPago*. Como se señaló en el capítulo dedicado a los patrones GRASP, una decisión sobre los mensajes implica asignar una responsabilidad, y esos patrones servirán para escogerlos y justificarlos.

19.7.1 Elección de la clase Controlador

Nuestra primera decisión se refiere al manejo de la responsabilidad del mensaje de operación del sistema *efectuarPago*. Con base en el patrón Controlador de GRASP, igual que en el caso de *introducirProducto*, seguiremos empleando *TPDV* como Controlador. Es frecuente utilizar el mismo controlador a lo largo de un caso de uso.

19.7.2 Creación de Pago

Una de las poscondiciones contractuales estipula lo siguiente.

- Se creó un *Pago* (creación de instancia).

Se trata de una responsabilidad relativa a la creación y, por tanto, debería aplicarse el patrón Creador de GRASP.

¿Quién registra, agrega, utiliza más ampliamente o contiene un *Pago*? Como resulta atractivo afirmar que la instancia *TPDV* registra lógicamente un *Pago*, se pensará que es idóneo para ser usado. Además, cabe suponer que una *Venta* utilizará estrechamente un *Pago*; de ahí la posibilidad de que sea un buen candidato.

Otra manera de encontrar un creador consiste en servirse del patrón Experto para determinar quién es el Experto en los datos de inicialización: en este caso, el monto ofrecido. *TPDV* es el controlador que recibe el mensaje *efectuarPago(montoOfrecido)* de la operación del sistema; por eso al principio incluirá la cantidad ofrecida. En consecuencia, *TPDV* vuelve a ser un buen candidato.

En conclusión, tenemos dos candidatos:

- *TPDV*
- *Venta*

Consideremos algunas de las consecuencias de estas posibilidades respecto a los patrones Alta Cohesión y Bajo Acoplamiento de GRASP. Si escogemos la *Venta* para crear el *Pago*, el trabajo (o las responsabilidades) de *TPDV* será más ligero, pues dará origen a una definición más simple de *TPDV*. Además no necesita saber que existe una instancia *Pago*, porque podemos registrarla indirectamente mediante la *Venta* y obtendremos así un menor acoplamiento con *TPDV*. Ello, a su vez, nos permite construir el diagrama de colaboración de la figura 19.10.

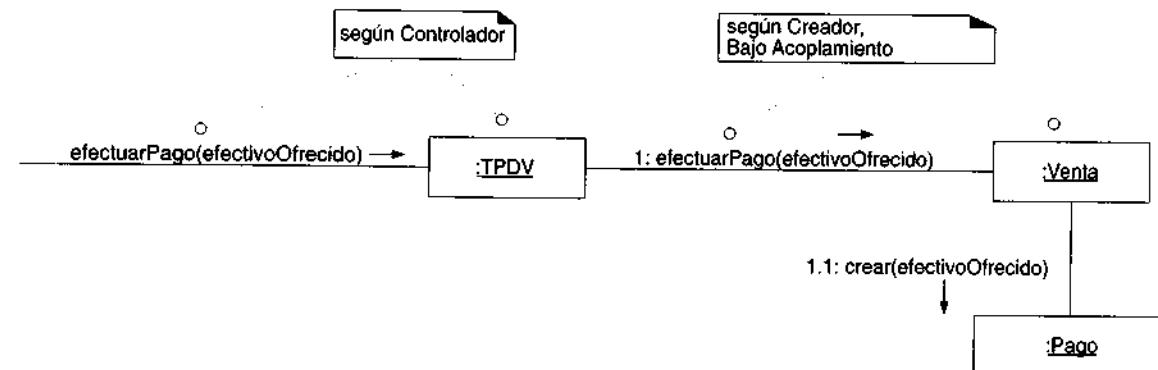


Figura 19.10 Diagrama de colaboración *TPDV-efectuarPago*.

Este diagrama de colaboración cumple con las poscondiciones del contrato: se creó el *Pago* asociado a *Venta* y se estableció su *montoOfrecido*.

19.7.3 Registro de la venta

Una vez formulados, los requerimientos estipulan que la venta debería ser colocada en un registro histórico o bitácora. Como de costumbre, el patrón Experto tendría que ser el primero en considerarse, salvo que se trate de un problema de controlador o de creación (pero no es así); además habría que expresar la responsabilidad:

¿Quién asume la responsabilidad de conocer todas las ventas registradas y de llevar a cabo el registro?

Es lógico que una *Tienda* conozca todas las ventas registradas, pues están estrechamente relacionadas con sus finanzas. Entre otras opciones mencionamos la inclusión de los conceptos clásicos de la contabilidad, como un *LibroMayordeVentas*. Es conveniente usar un objeto *LibroMayordeVentas* conforme vaya creciendo el diseño y la *Tienda* vaya perdiendo cohesión (figura 19.11).

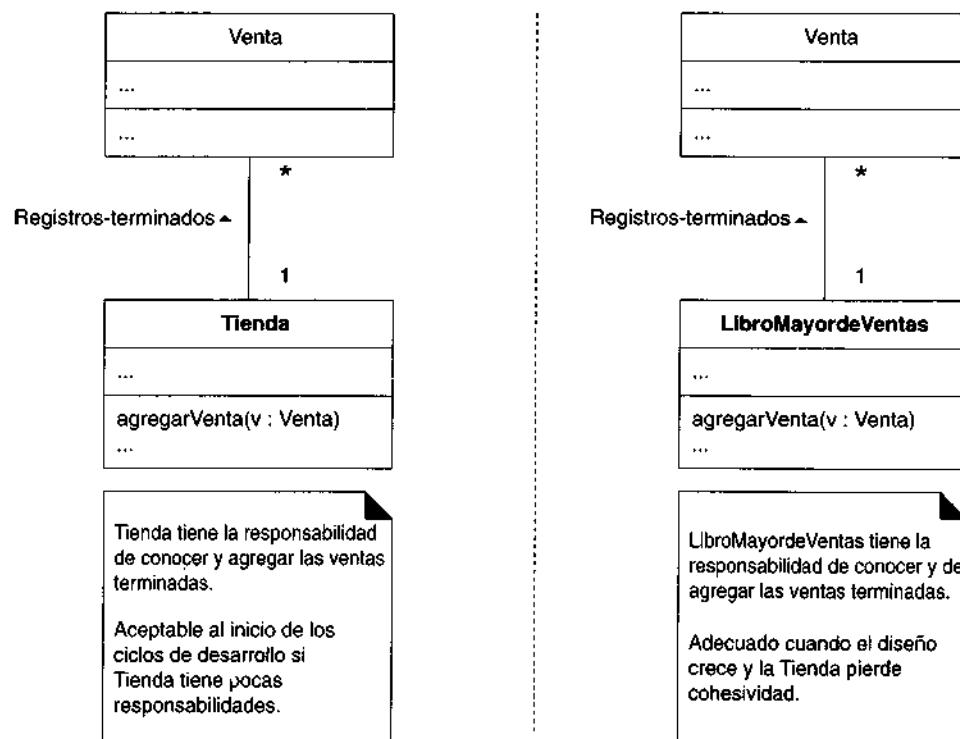


Figura 19.11 ¿Quién asumirá la responsabilidad de conocer las ventas terminadas?

Nótese asimismo que las poscondiciones del contrato indican relacionar la *Venta* con la *Tienda*. Éste es un ejemplo donde las poscondiciones tal vez no sean lo que en ver-

dad deseamos conseguir con el diseño. Acaso no hayamos pensado antes en un *LibroMayordeVentas*, pero ahora que lo hicimos hemos decidido utilizarlo en vez de una *Tienda*. De ser así, en teoría incorporaríamos también *LibroMayordeVentas* al modelo conceptual. Cabe esperar este tipo de descubrimiento y cambio durante la fase de diseño. Por fortuna, el desarrollo iterativo ofrece un ciclo de vida para el cambio continuo.

En este caso, no abandonaremos el plan original de usar la *Tienda*.

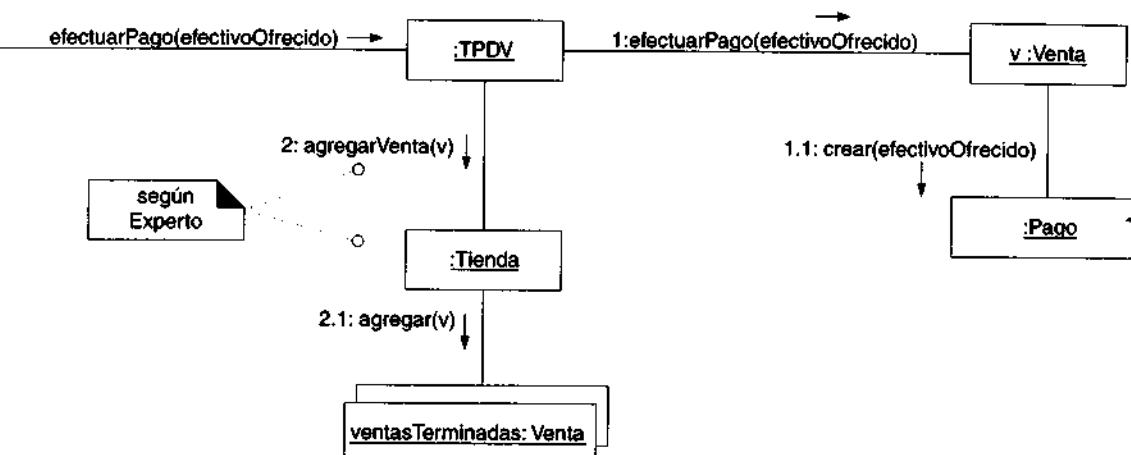


Figura 19.12 Registro de una venta terminada.

19.7.4 Cálculo del saldo

La sección dedicada a las responsabilidades estipula que el saldo ha sido calculado; entonces se imprimirá en un recibo y se mostrará en un monitor.

Debido al principio de Separación Modelo-Vista, no debe preocuparnos cómo se presentará visualmente el saldo, pero sí es necesario asegurarnos de que se conozca. Adviértase que ninguna clase lo conoce en este momento; por ello, mediante un diagrama de colaboración hay que construir un diseño de las interacciones de los objetos que cumpla con este requerimiento.

Como es habitual, el patrón Experto será el primero en considerarse, salvo que se trate de un problema de controlador o de creación (pero no es así), y habrá que formular la responsabilidad en los siguientes términos:

¿Quién asume la responsabilidad de conocer el saldo?

Para calcular el saldo necesitamos el total de la venta y el pago en efectivo ofrecido. Por tanto, *Venta* y *Pago* son Expertos parciales en la solución de este problema.

Si el *Pago* es el principal responsable de conocer el saldo, necesitará ser visible a la *Venta* para pedirle su total. Como por ahora no la conoce, con este método aumentará el acoplamiento global del diseño: no soportará el patrón Bajo Acoplamiento.

En cambio, si la *Venta* es el principal responsable de conocer el saldo, necesitará ser visible al *Pago* para pedirle su efectivo ofrecido. Dado que la *Venta* ya es visible al *Pago* —como su creador—, con este enfoque no se aumenta el acoplamiento global; de ahí que sea un diseño preferible.

En consecuencia, el diagrama de colaboración de la figura 19.13 ofrece una solución para conocer el saldo.

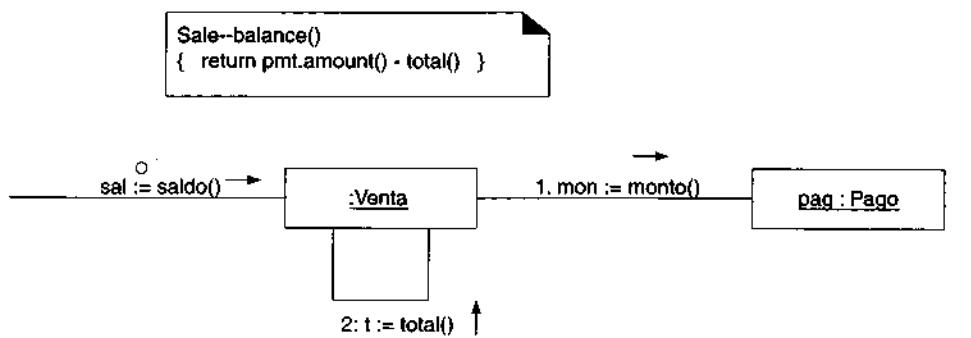


Figura 19.13 El diagrama de colaboración venta-saldo.

19.8 El diagrama de colaboración: Iniciar

19.8.1 ¿Cuándo crear el diagrama de colaboración Iniciar?

La mayoría de los sistemas, si no es que todos, tienen un caso de uso *Iniciar* y alguna operación inicial relacionada con el comienzo de la aplicación. Aunque es la primera en ser ejecutada, se ha considerado la conveniencia de posponer la preparación del respectivo diagrama de colaboración hasta después de considerar el resto de las operaciones del sistema. Con ello se garantiza el descubrimiento de información importante referente a las actividades de inicialización que se requieren para brindar soporte a los diagramas ulteriores de interacción.

Prepare al final el diagrama de colaboración *Iniciar*.

19.8.2 Cómo se inician las aplicaciones

La operación *Iniciar* representa en forma abstracta la fase inicializadora de la ejecución cuando se comienza una aplicación. Si queremos saber cómo diseñar un diagrama de colaboración para esa operación, conviene conocer los contextos donde puede ocurrir la inicialización. Depende del lenguaje de programación y del sistema operativo la manera en que comienza una aplicación y en que se inicializa.

En todos los casos, una expresión de diseño común consiste en crear al final un **objeto inicial del dominio**: el primero del dominio del problema que generamos. En su método de inicialización (un constructor en Java, por ejemplo), es entonces el encargado de producir el resto de los objetos del dominio del problema y de inicializarlos.

El lugar donde se crea el objeto inicial del dominio dependerá del lenguaje de programación orientado a objetos y también del sistema operativo. Así, en un applet de Java su inicio puede hacer que los atributos (variables de instancia) sean instanciados. Uno de ellos puede ser el objeto inicial del dominio; por ejemplo una *Tienda* que crea otros objetos como una *TPDV*.

```

public class TPDVApplet extends Applet
{
  public void init()
  {
    tpdv = tienda.obtenerTPDV();
  }

  // Tienda es el objeto inicial del dominio.
  // El constructor de Tienda crea
  // otros objetos del dominio
  private Tienda tienda = new Tienda();

  private TPDV tpdv;
  private Venta venta;
}
  
```

19.8.3 Interpretación de la operación del sistema Iniciar

La exposición anterior sobre las formas en que comienza una aplicación muestra que la operación del sistema *Iniciar* es una abstracción independiente del lenguaje. En la fase de diseño se observa variación en el sitio donde se produce el objeto inicial y en el hecho de que asuma o no el control del proceso. El objeto inicial del dominio no suele hacerlo si hay una interfaz gráfica para el usuario, pero sí en caso contrario.

El diagrama de colaboración de la operación *Iniciar* describe lo que sucede cuando se crea el objeto inicial del dominio del problema y, opcionalmente, lo que sucede si asume el control. No incluye ninguna actividad anterior ni subsecuente en la capa de objetos destinada al interfaz gráfico para el usuario, si es que existe.

Por tanto, la operación *Iniciar* puede interpretarse así:

1. En un diagrama de colaboración, envíe un mensaje *crear()* para producir el objeto inicial del dominio.
2. (opcional) Si el objeto inicial va a asumir el control del proceso, en un segundo diagrama de colaboración envíe un mensaje *ejecutar* (u otro equivalente).

19.8.4 Aplicación de la operación Iniciar al punto de venta

La operación sistemática *Iniciar* ocurre cuando un supervisor enciende el sistema de punto de venta y el software se ejecuta. Supongamos que contamos con una interfaz gráfica para el usuario y que un objeto de la capa de presentación (por ejemplo, una instancia applet de Java) se encargará de producir el objeto inicial del dominio del problema.¹ Supongamos asimismo que el objeto inicial *no* asume el control del proceso; éste permanece en el applet tras haber sido creado el objeto inicial del dominio. Así pues, el diagrama de colaboración de la operación *Iniciar* puede reinterpretarse exclusivamente como un mensaje *crear()* que se envía para producir el objeto inicial.

19.8.5 Elección del objeto inicial del dominio

¿Cuál debería ser la clase del objeto inicial del dominio?

Escoja como objeto inicial del dominio:

- Una clase que represente todo el sistema de información lógico.
- Una clase que represente íntegramente el negocio u organización.

En la selección de estas opciones pueden influir consideraciones referentes a los patrones Alta Cohesión y Bajo Acoplamiento.

Por tanto, en la aplicación del punto de venta, entre las decisiones razonables acerca del objeto inicial se encuentran:

Todo el Sistema de información lógico	<i>TPDV, SistemaInformacionalMenudeo</i>
Negocio u organización global	<i>Tienda</i>

En esta aplicación se escogió la *Tienda* como objeto inicial.

19.8.6 Objetos persistentes: *EspecificacioneProducto*

En la aplicación realista, las instancias *EspecificacioneProducto* residirán en un medio de almacenamiento persistente: una base de datos relacional o de objetos. Durante la operación *Iniciar*, si no hay más que unos cuantos de esos objetos, podemos cargarlos todos en la memoria directa de la computadora. Pero si hay muchos, cargarlos todos

¹ En un caso real, probablemente no se trate de un applet de Java, pero cuando se sirve de ejemplo ofrece un estándar simple y muy conocido.

requeriría demasiada memoria o tiempo. Una alternativa —quizá la más viable— consiste en cargar las instancias individuales en la memoria conforme vaya necesitándose.

El diseño de cómo, previa solicitud, cargar dinámicamente los objetos y almacenarlos en la memoria a partir de una base de datos es simple si se utiliza una base de datos de objetos, pero difícil si se utiliza una base de datos relacional. Por ahora vamos a posponer la solución de este problema y haremos la suposición simplificadora de que todas las instancias *EspecificacioneProducto* pueden crearse “mágicamente” en la memoria por medio del objeto *CatalogodeProductos*.

En el capítulo 38 examinaremos el tema de los objetos persistentes y la manera de cargarlos en la memoria.

19.8.7 El diagrama de colaboración guardar--crear() (store-create)

He aquí el contrato *Iniciar*:

Contrato

- | | |
|-----------------------|-------------------------|
| Nombre: | <i>Iniciar()</i> . |
| Responsabilidades: | Inicializar el sistema. |
| Tipo: | Sistema. |
| Referencias cruzadas: | |
| Notas: | |
| Excepciones: | |
| Salida: | |
| Precondiciones: | |
| Post-condiciones: | |
- Se crearon *Tienda*, *TPDV*, *CatalogodeProductos* y *EspecificacionesdeProducto* (*creación de instancia*).
 - Se asoció *CatalogodeProductos* a *EspecificacionesdeProducto* (*asociación formada*).
 - Se asoció *Tienda* a *CatalogodeProductos* (*asociación formada*).
 - Se asoció *Tienda* a *TPDV* (*asociación formada*).
 - Se asoció *TPDV* a *Catálogo de productos* (*asociación formada*).

Como hemos señalado, lo anterior se interpretará como lo que sucede a raíz de la creación del objeto inicial —la *Tienda*— cuando se envía un mensaje *crear*.

En la figura 19.14 se incluye un diagrama de colaboración que cumple con estas poscondiciones. La *Tienda* fue elegida para crear *CatalogodeProductos* y *TPDV* mediante el patrón Creador. También se eligió *CatalogodeProductos* para producir *EspecificacionesdeProducto*.

Observe que la creación de todas las instancias *EspecificaciondeProducto* y su incorporación a un contenedor se realizan en una sección de repetición, indicada por un asterisco (*) colocado después de los números de secuencia.

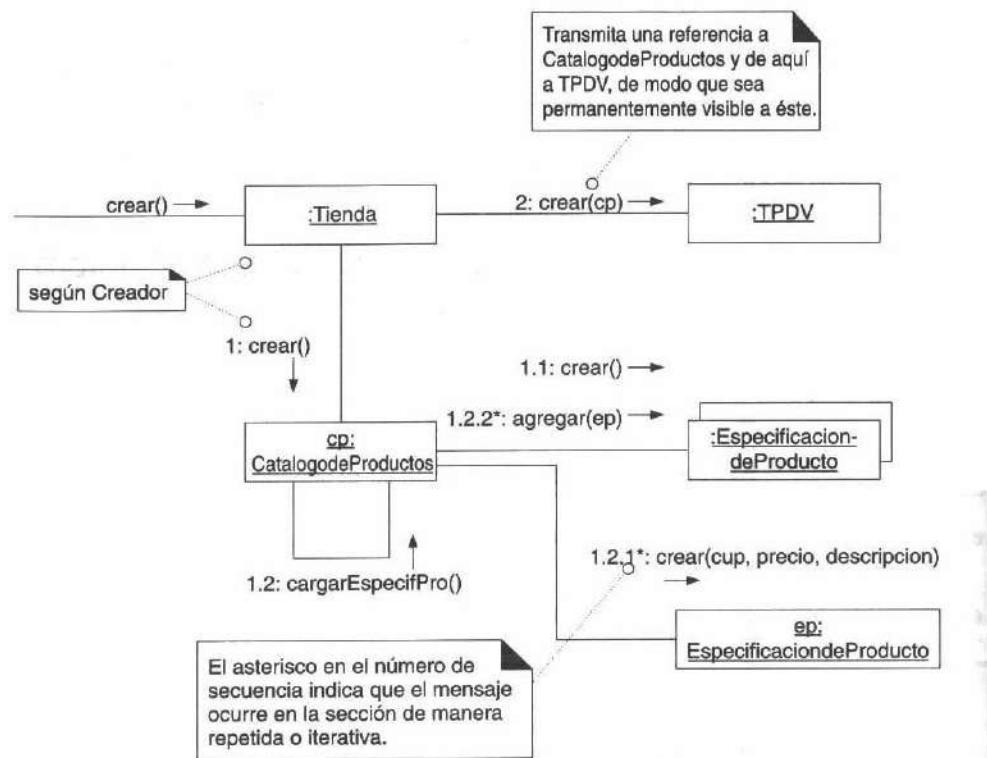


Figura 19.14 Creación del objeto inicial del dominio y de objetos posteriores.

Una discrepancia interesante entre el análisis y el diseño se exemplifica en el hecho de que la *Tienda* sólo crea *un* objeto *TPDV*. Desde el punto de vista analítico, una tienda real puede albergar *muchas terminales* en el punto de venta. Pero ahora estamos considerando un diseño de software, no lo que ocurre en el mundo real. La *Tienda* representada en un diagrama de colaboración no es una tienda real; es un objeto de software. El objeto *TPDV* de un diagrama tampoco es una terminal real; es un objeto de software. En nuestros requerimientos actuales, el software *Tienda* no necesita crear más que una sola instancia del software *TPDV*.

En la fase de análisis y de diseño, la multiplicidad entre las clases de objetos tal vez no sea la misma.

19.9 Cómo conectar la capa de presentación y la de dominio

Como hemos visto, si interviene una interfaz gráfica para el usuario —digamos un applet de Java que forma parte del coordinador de aplicaciones y de la capa de presentación—, el applet se encargará de comenzar la producción del objeto inicial del dominio, como se aprecia en la figura 19.15. Primero, el applet crea el objeto inicial del dominio (*una instancia Tienda*), que a su vez produce una instancia *TPDV*. Luego solicita a la *Tienda* una referencia a la instancia *TPDV* y la guarda en un atributo, para que applet pueda enviarle directamente un mensaje.

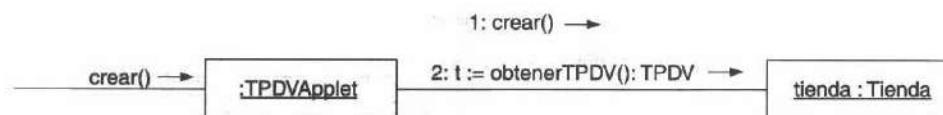


Figura 19.15 Conexión de las capas de presentación y de dominio.

Una vez que applet tiene una conexión con la instancia *TPDV*, puede enviarle mensajes de eventos del sistema, entre ellos *introducirProducto* y *terminarVenta* (figura 19.16).

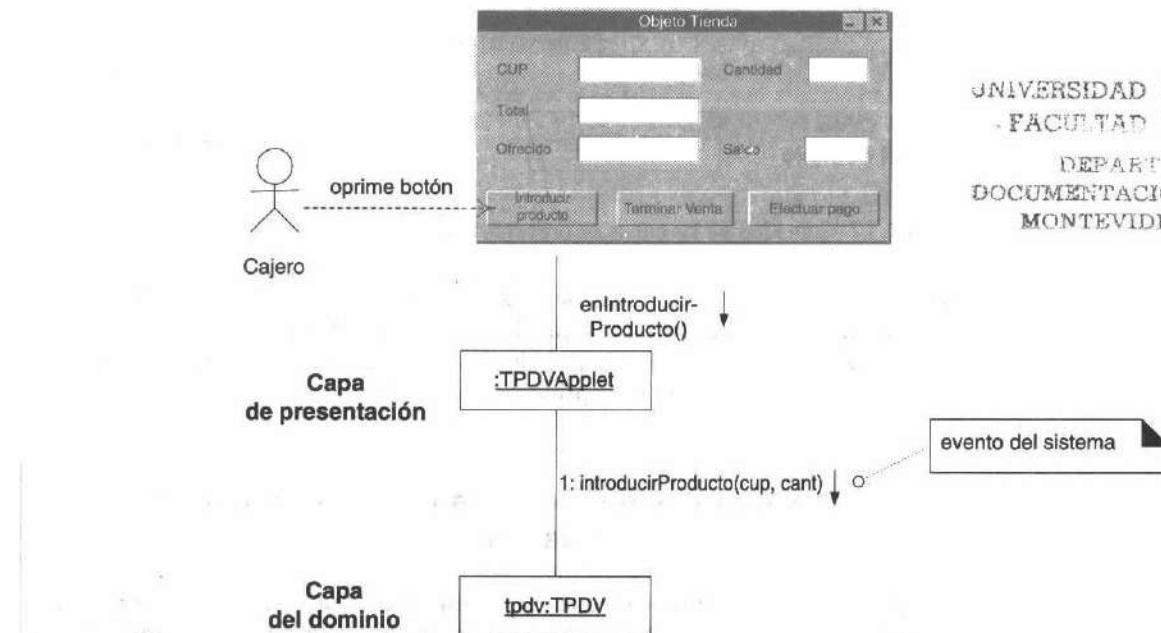


Figura 19.16 Conexión de las capas de presentación y dominio.

En el caso del mensaje *introducirProducto*, la ventana habrá de exhibir el total actual después de cada captura. Como se ve en la figura 19.17, después que *TPDVApplet* envía el mensaje *introducirProducto* al objeto *TPDV*:

1. Obtiene una referencia a la *Venta* (si es que todavía no tiene una).
2. Almacena la referencia *Venta* en un atributo.
3. Envía un mensaje de *total* a la *Venta* con el fin de conseguir la información necesaria para presentar en la ventana el total actual.

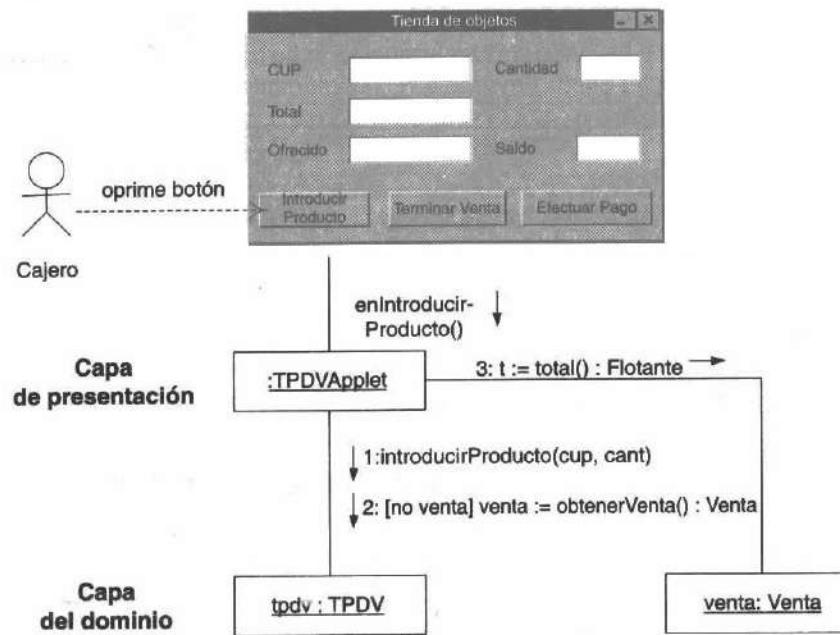


Figura 19.17 Conexión de las capas de presentación y dominio.

Adviértase que, en estos diagramas de colaboración, el applet de Java (*TPDVApplet*), que forma parte de la capa de presentación, no maneja la lógica de la aplicación. A través de *TPDV* envía peticiones de trabajo (las operaciones del sistema) a la capa del dominio. Esto nos lleva al siguiente principio de diseño:

Responsabilidades de las capas de presentación y de dominio

La capa de presentación no debería tener responsabilidades lógicas de dominio. Tan sólo debería encargarse de las tareas de presentación (interfaz), entre ellas actualizar los elementos contenidos en la ventana.

La capa de presentación debería enviarle a la capa del dominio las peticiones de las tareas orientadas al dominio, que es la que se ocupa de ellas.

19.10 Resumen

El diseño de las interacciones de los mensajes y la asignación de responsabilidades constituyen la esencia de un diseño orientado a objetos. Las decisiones que se tomen en él ejercerán un impacto muy profundo sobre la extensibilidad, la claridad y la capacidad de mantenimiento de un sistema de software orientado a objetos, así como en el grado y la calidad de los componentes reutilizables. Contamos con algunos principios que nos permiten tomar decisiones respecto a la asignación de responsabilidades. Los patrones GRASP resumen algunos de los más generales y comunes de los cuales se sirven los diseñadores orientados a objetos.

En este capítulo hemos examinado la aplicación de dichos patrones para crear clases bien diseñadas y un sistema de objetos que interactúan.

DETERMINACIÓN DE LA VISIBILIDAD

Objetivos

- Identificar cuatro tipos de visibilidad.
- Diseñar para lograr la visibilidad.
- Dar ejemplos de los tipos de visibilidad en la notación de UML.

20.1 Introducción

La visibilidad es la capacidad de un objeto para ver otro o hacer referencia a él. En el presente capítulo estudiaremos los problemas de diseño relacionados con la visibilidad.

20.2 Visibilidad entre objetos

Los diagramas de colaboración creados para los eventos del sistema (*introducirProducto*, por ejemplo) describen gráficamente los mensajes entre objetos. Para que un objeto emisor envíe un mensaje a un objeto receptor, el emisor tiene que ser visible a éste: debe tener alguna clase de referencia o apuntador a él.

Por ejemplo, el mensaje *especificacion* enviado de una instancia *TPDV* a una instancia *CatalogodeProductos* significa que ésta es visible a aquélla, como se muestra en la figura 20.1.

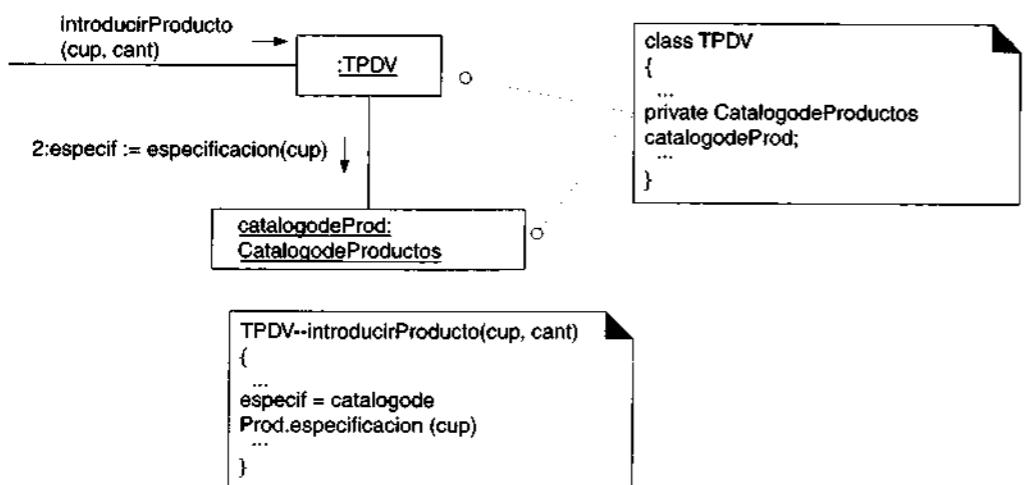


Figura 20.1 Se requiere visibilidad de la instancia TPDV a CatalogodeProductos.¹

Cuando se construye un diseño de objetos que interactúan, es preciso asegurarse de que exista la visibilidad necesaria pues de otro modo no podrá darse soporte a la interacción de los mensajes.

El UML cuenta con una notación especial para mostrar la visibilidad; en este capítulo explicaremos varios tipos de visibilidad y su representación gráfica.

20.3 Visibilidad

En el lenguaje cotidiano, la **visibilidad** es la capacidad de un objeto para “ver” o hacer referencia a otro. En un sentido más general, se relaciona con la cuestión del alcance o ámbito: ¿es un recurso (digamos una instancia) dentro del ámbito de otro? Hay cuatro formas comunes en que podemos conseguir la visibilidad del objeto *A* al objeto *B*:

1. *Visibilidad de atributos*: *B* es un atributo de *A*.
2. *Visibilidad de parámetros*: *B* es un parámetro de un método de *A*.
3. *Visibilidad declarada localmente*: se declara que *B* es un objeto local en un método de *A*.
4. *Visibilidad global*: en alguna forma *B* es visible globalmente.

¹ En este ejemplo de código y en otros subsecuentes, puede simplificarse el lenguaje por razones de brevedad y de claridad.

Consideramos la visibilidad por lo siguiente:

Para que un objeto *A* envíe un mensaje a un objeto *B*, debe ser visible a aquél.

Por ejemplo, para preparar un diagrama de colaboración donde se envíe un mensaje de la instancia *TPDV* a la instancia *CatalogodeProductos*, la primera deberá tener visibilidad de la segunda. Una solución usual consiste en que una referencia a la instancia *CatalogodeProductos* sea conservada como un atributo de *TPDV*.

20.3.1 Visibilidad de atributos

Existe **visibilidad de atributos** de *A* a *B* cuando *B* es un atributo de *A*. Se trata de una visibilidad relativamente permanente porque persiste mientras existan *A* y *B*. Es una forma muy común de visibilidad en los sistemas orientados a objetos.

Por ejemplo, en una definición de *TPDV* en una clase de Java, esta instancia puede tener visibilidad de atributo para un *CatalogodeProductos* por ser un atributo (variable de instancia en Java) de *TPDV*.

```

public class POST
{
    ...
    private ProductCatalog prodCatalog;
    ...
}
    
```

Se requiere esta visibilidad porque, en el diagrama de colaboración *introducirProducto* de la figura 20.2, se necesita una instancia *TPDV* que envíe el mensaje *especificacion* a una instancia *CatalogodeProductos*.

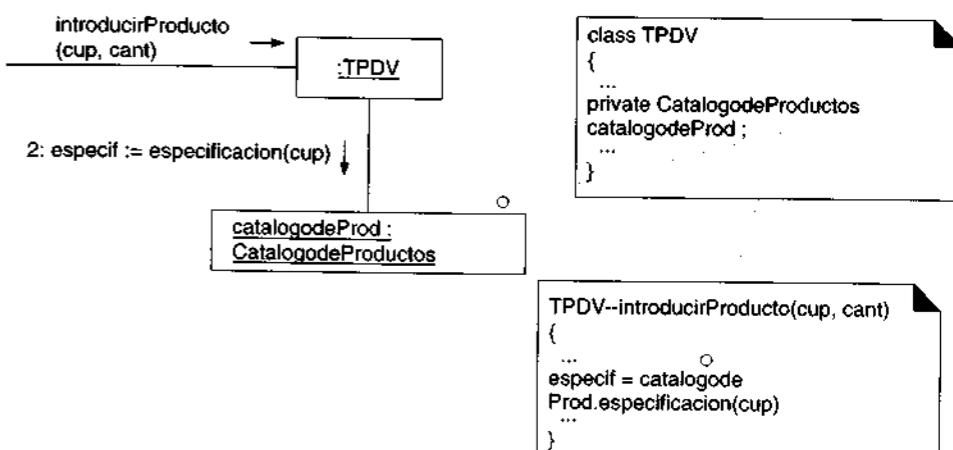


Figura 20.2 Visibilidad de atributos.

20.3.2 Visibilidad de los parámetros

Existe **visibilidad de parámetros** de A a B cuando B se transmite como un parámetro a un método de A. Se trata de una visibilidad relativamente temporal porque persiste sólo dentro del ámbito del método. Después de la visibilidad de atributos, es la segunda forma más común de visibilidad en los sistemas orientados a objetos.

He aquí un ejemplo: cuando el mensaje *hacerLineadeProducto* se envía a una instancia *Venta*, una instancia *EspecificaciondeProducto* es transferida como parámetro. Dentro del ámbito del método *hacerLineadeProducto*, la *Venta* tiene visibilidad de parámetro para *EspecificaciondeProducto* (figura 20.3).

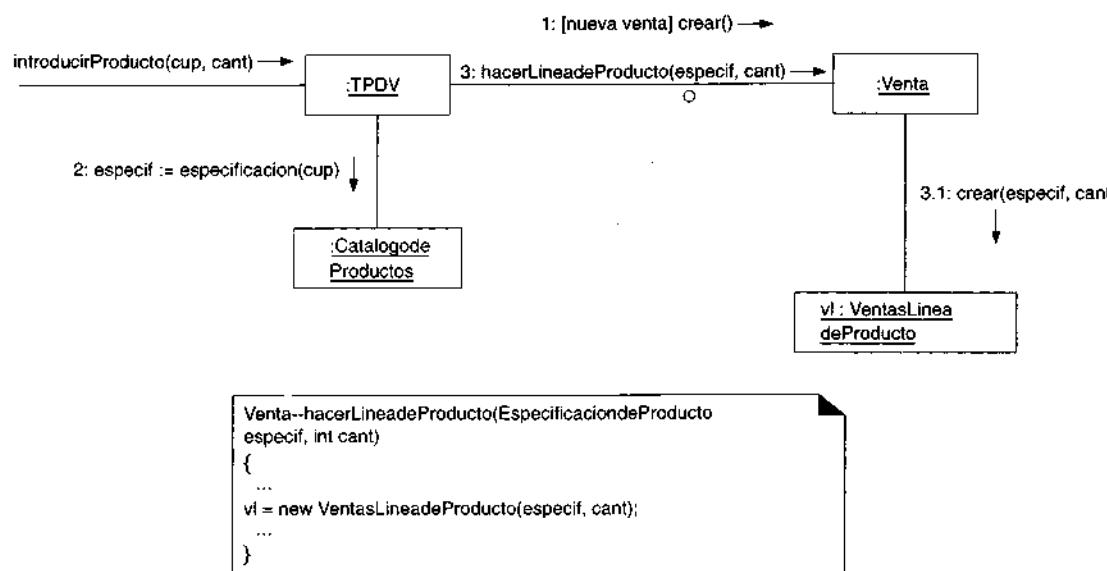


Figura 20.3 Visibilidad de parámetros.

Con frecuencia la visibilidad de parámetros se transforma en visibilidad de atributos. Así, cuando la *Venta* crea una nueva instancia *VentasLineadeProducto*, transmite una *EspecificaciondeProducto* a su método de inicialización (en C++ o en Java sería su **constructor**). Dentro del método de inicialización, se asigna el parámetro a un atributo y con ello se logra la visibilidad de atributos (figura 20.4).

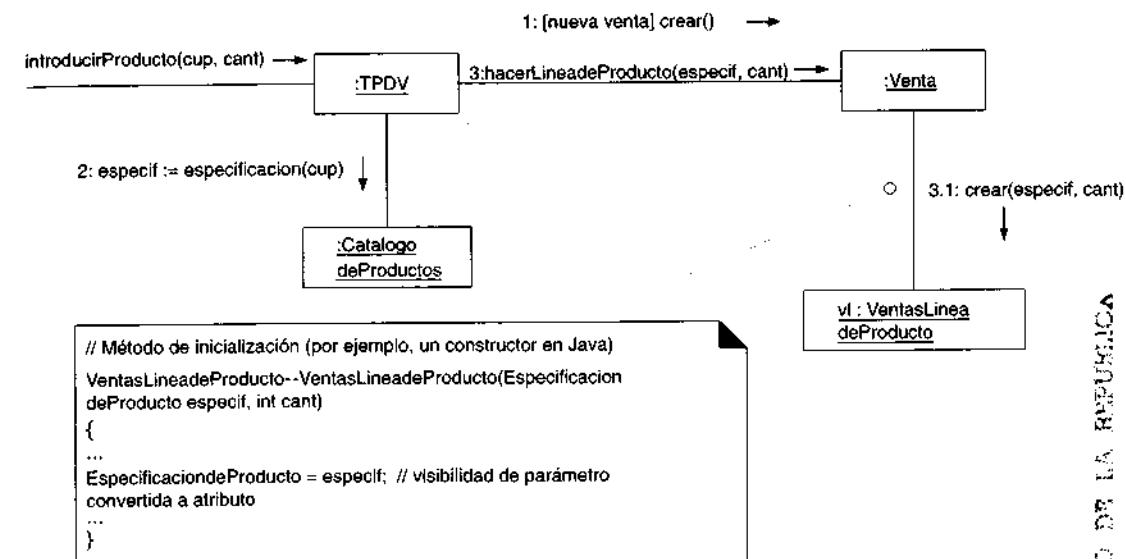


Figura 20.4 Transformación de visibilidad de parámetros a visibilidad de atributos.

20.3.3 Visibilidad declarada localmente

Existe una **visibilidad declarada localmente** de A a B, cuando se declara que B es un objeto local dentro de un método de A. Se trata de una visibilidad relativamente temporal porque persiste sólo dentro del ámbito del método. Después de la visibilidad de parámetros, es la tercera forma más común de visibilidad en los sistemas orientados a objetos. A continuación se mencionan dos medios habituales con que se logra esta clase de visibilidad:

1. Crear una nueva instancia local y asignarla a una variable local.
 2. Asignar a una variable local el objeto devuelto proveniente de la llamada a un método.

Se produce una variación de (2) cuando el método no declara de manera explícita una variable, pero existe implícitamente una como resultado de un objeto devuelto procedente de la invocación de un método.

Igual que en la visibilidad de parámetros, a menudo la visibilidad declarada localmente es transformada en visibilidad de atributo.

Un ejemplo de la segunda variación se encuentra en el método *introducirProducto* de la clase *TPDV* (figura 20.5).

Un ejemplo de la segunda variación se encuentra en el método *introducirProducto* de la clase *TPDV* (figura 20.5).

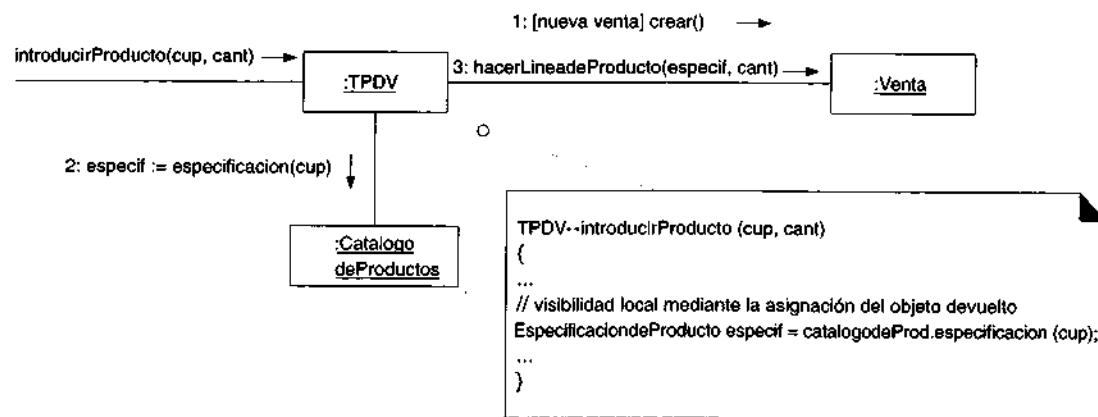


Figura 20.5 Visibilidad local.

20.3.4 Visibilidad global

Existe **visibilidad global** de A a B cuando B es global para A. Se trata de una visibilidad relativamente permanente, porque persiste mientras existan A y B. Es el tipo de visibilidad menos frecuente en los sistemas orientados a objetos.

La forma más obvia —aunque menos conveniente— de alcanzar la visibilidad global consiste en asignar una instancia a una variable global.

El método usual de conseguir la visibilidad global es utilizar el patrón **Singleton** [GHJV95], del cual nos ocuparemos en el capítulo 35.

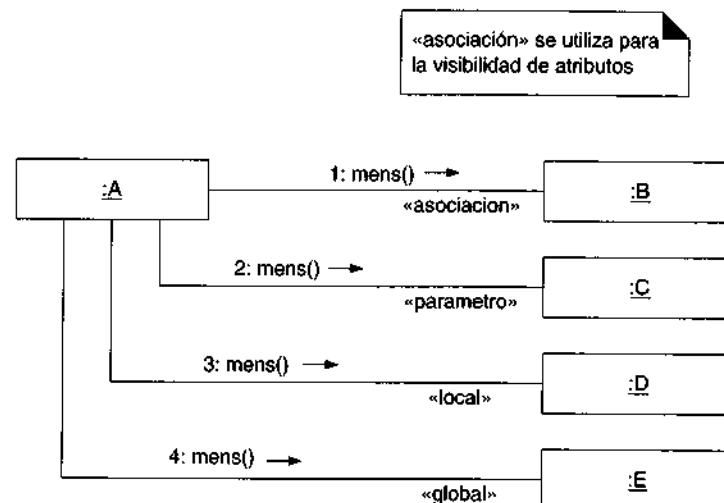


Figura 20.6 Estereotipos de implementación para la visibilidad.

20.4 Presentación de la visibilidad en el UML

El lenguaje UML cuenta con una notación que describe la implementación de varios tipos de visibilidad en un diagrama de colaboración (figura 20.6). Estos elementos decorativos son opcionales y normalmente no se requieren; son de gran utilidad cuando hace falta la clarificación.

DIAGRAMAS DE CLASES DEL DISEÑO

Objetivos

- Crear diagramas de clases del diseño.
- Identificar las clases, los métodos y las asociaciones que deben incluirse en un diagrama de clases de este tipo.

21.1 Introducción

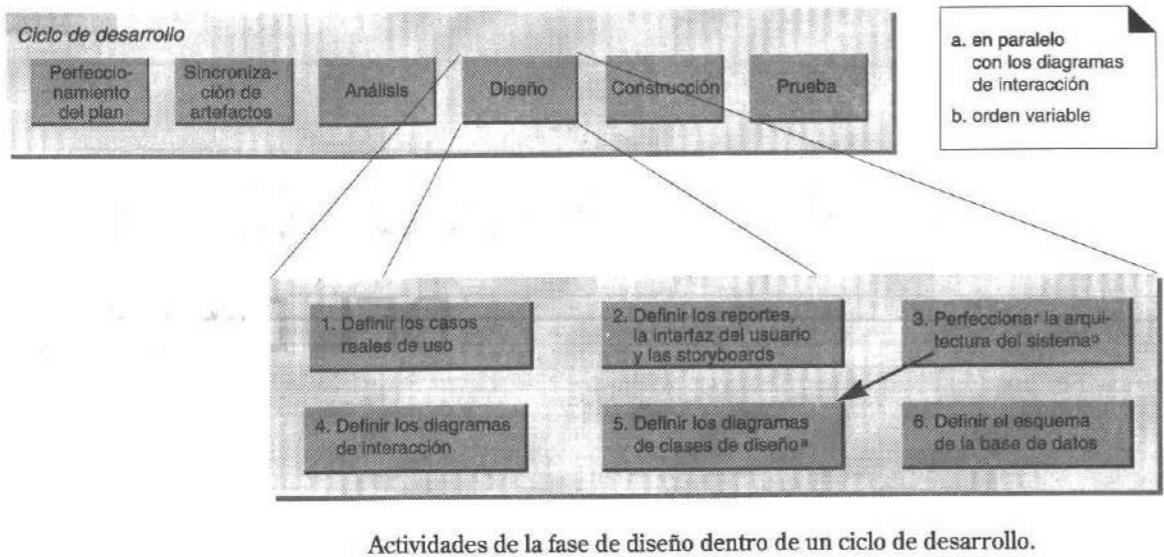
Una vez terminados los diagramas de interacción para el ciclo actual de desarrollo de la aplicación de punto de venta, podemos identificar la especificación de las clases de software (y las interfaces) que participan en la solución de software y complementarlas con detalles de diseño, por ejemplo los métodos.

El lenguaje UML ofrece una notación que muestra los detalles de diseño en los diagramas de estructura —o clase— estática; en este capítulo vamos a estudiarla y a elaborar los diagramas de clases del diseño.

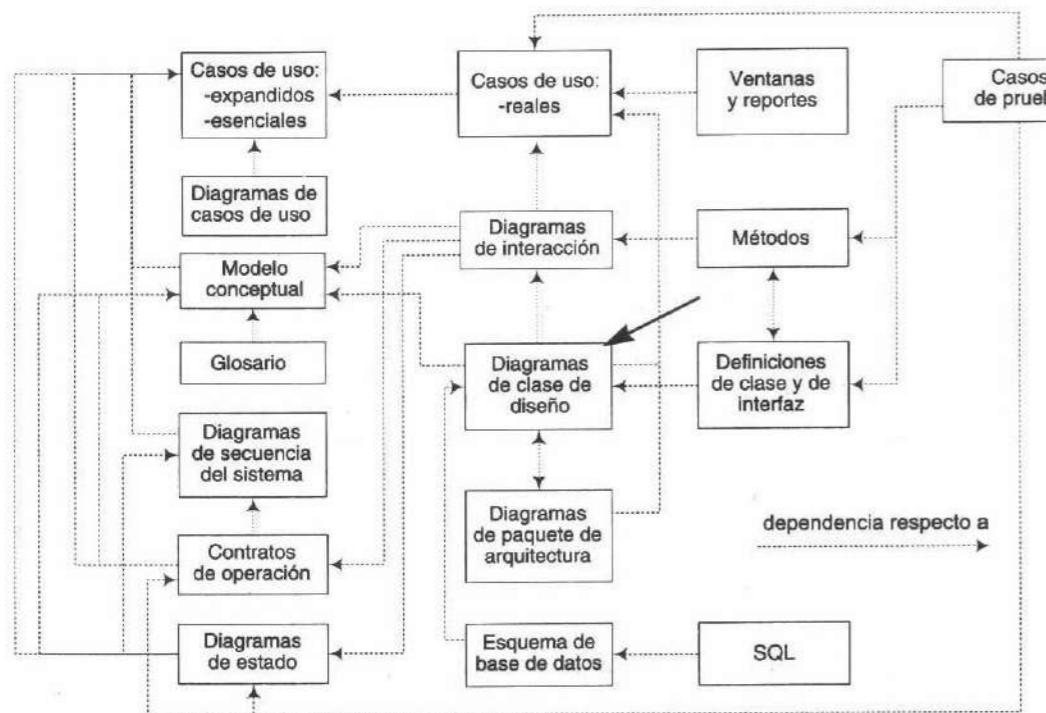
21.2 Actividades y dependencias

La definición de este tipo de diagramas se lleva a cabo en la fase de diseño del ciclo de desarrollo. Su preparación exige crear antes:

- Diagramas de interacción: a partir de ellos el diseñador identifica las clases de software que intervienen en la solución, así como los métodos de las clases.
- Modelo conceptual: a partir de éste el diseñador agrega detalles a la definición de las clases.



Actividades de la fase de diseño dentro de un ciclo de desarrollo.



Dependencias de los artefactos durante la fase de construcción.

21.3 Cuándo crear diagramas de clases del diseño

Aunque nuestra exposición de los diagramas de clases del diseño *viene después* de su elaboración, en la práctica suelen prepararse al mismo tiempo que los diagramas de interacción. Podemos bosquejar muchas clases, nombres de métodos y relaciones al inicio de la fase de diseño, aplicando los patrones de asignación de responsabilidades antes de dibujar los diagramas de interacción. Frente a las tarjetas CRC son una notación alterna de carácter más gráfico, que sirven para registrar las responsabilidades y los colaboradores.

21.4 Ejemplo de un diagrama de clases del diseño

El diagrama de la figura 21.1 contiene una definición parcial de software de las clases *TPDV* y *Venta*.

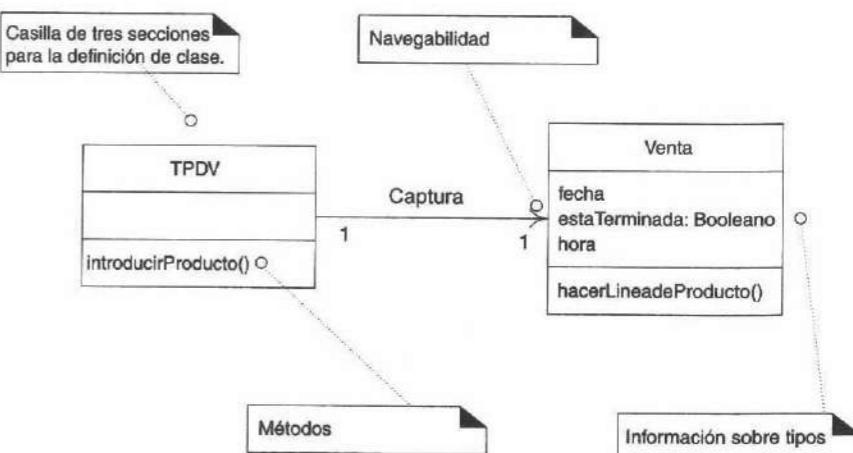


Figura 21.1 Ejemplo de diagrama de clases del diseño.

Además de las asociaciones y de los atributos básicos, se ha ampliado el diagrama para incluir —por ejemplo— los métodos de cada clase, la información sobre el tipo de los atributos, su visibilidad y la navegación entre objetos.

21.5 Diagramas de clases del diseño

El **diagrama de clases del diseño** describe gráficamente las especificaciones de las clases de software y de las interfaces (las de Java, por ejemplo) en una aplicación. Normalmente contiene la siguiente información:

- clases, asociaciones y atributos
- interfaces, con sus operaciones y constantes
- métodos
- información sobre los tipos de los atributos
- navegabilidad
- dependencias

A diferencia del modelo conceptual, un diagrama de este tipo contiene las definiciones de las entidades del software en vez de conceptos del mundo real. El UML no define concretamente un elemento denominado “diagrama clases del diseño”, sino que se sirve de un término más genérico: “diagrama de clases”. Yo opté por incluir el término “diseño” en “diagrama de clases” para recalcar que se trata de una perspectiva desde el punto de vista del diseño de las entidades de software y no de una concepción analítica sobre los conceptos del dominio.

21.6 Cómo elaborar un diagrama de clases del diseño

Aplique la siguiente estrategia para construir un diagrama de clases del diseño:

Para preparar un diagrama de clases orientado al diseño:

1. Identifique todas las clases que participan en la solución del software. Para ello analice los diagramas de interacción.
2. Dibújelas en un diagrama de clases.
3. Duplique los atributos provenientes de los conceptos asociados del modelo conceptual.
4. Agregue los nombres de los métodos analizando los diagramas de interacción.
5. Incorpore la información sobre los tipos a los atributos y a los métodos.
6. Agregue las asociaciones necesarias para dar soporte a la visibilidad requerida de los atributos.
7. Agregue flechas de navegabilidad a las asociaciones para indicar la dirección de la visibilidad de los atributos.
8. Agregue las líneas de relaciones de dependencia para indicar la visibilidad no relacionada con los atributos.

21.7 Comparación entre el modelo conceptual y los diagramas de clases del diseño

En el modelo conceptual una *Venta* no representa una definición de software; más bien es una abstracción de un concepto del mundo real acerca del cual queremos afirmar algo. En cambio, los diagramas de clases del diseño expresan —para el sistema computarizado— la definición de clases como componentes del software. En estos diagramas una *Venta* representa una clase de software (figura 21.2).

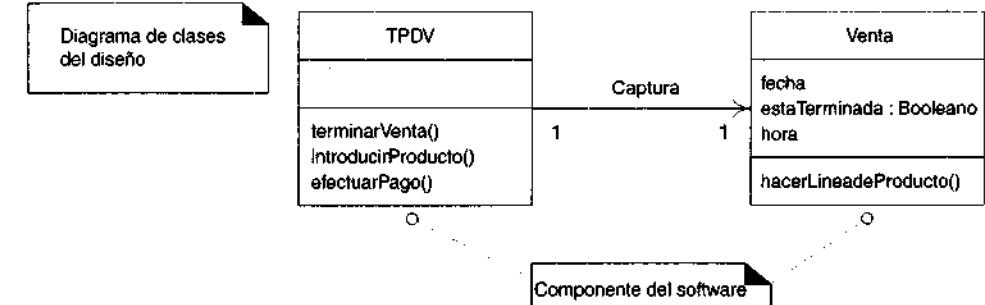
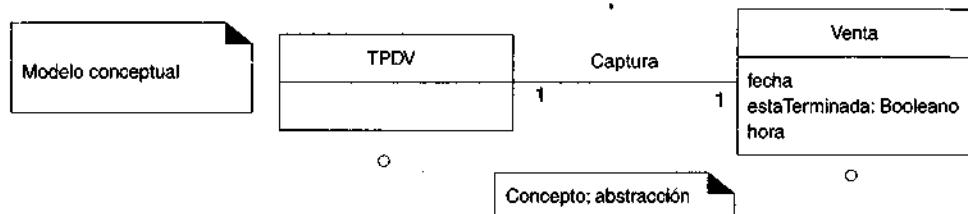


Figura 21.2 Comparación entre el modelo conceptual y el diagrama de clases del diseño.

21.8 Creación de diagramas de clases del diseño para el punto de venta

21.8.1 Identificación de las clases del software y su ilustración

El primer paso en la preparación de estos diagramas como parte del modelo de la solución consiste en identificar las clases que intervienen en la solución del software.

Podemos encontrarlas con sólo examinar todos los diagramas de interacción y listar las clases mencionadas. En las aplicaciones de TPDV ellas son:

TPDV	Venta
CatalogodeProductos	EspecificaciondeProductos
Tienda	VentasLineadeProducto
Pago	

El siguiente paso consiste en dibujar un diagrama de clases para estas clases e incluir en el modelo conceptual los atributos ya identificados (figura 21.3).

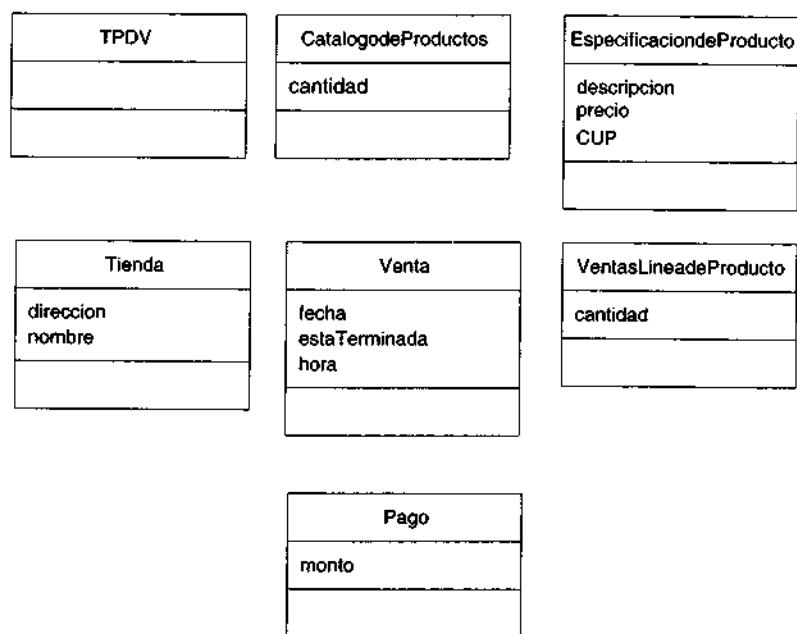


Figura 21.3 Clases de software en la aplicación.

Nótese que muchos de los conceptos del modelo conceptual, entre ellos *Cajero*, *Gerente* y *Producto*, no aparecen en el diseño. No es necesario —en el actual ciclo de desarrollo— representarlos en el software. Pero en ciclos ulteriores podemos incorporarlos al diseño, conforme vayamos descubriendo nuevos requerimientos y casos de uso.

1.8.2 Agregar los nombres de los métodos

Para identificar los métodos de las clases se analizan los diagramas de colaboración. Por ejemplo, si el mensaje *hacerLineadeProducto* se envía a una instancia de la clase *Venta*, entonces ésta deberá definir el método *hacerLineadeProducto* (figura 21.4).

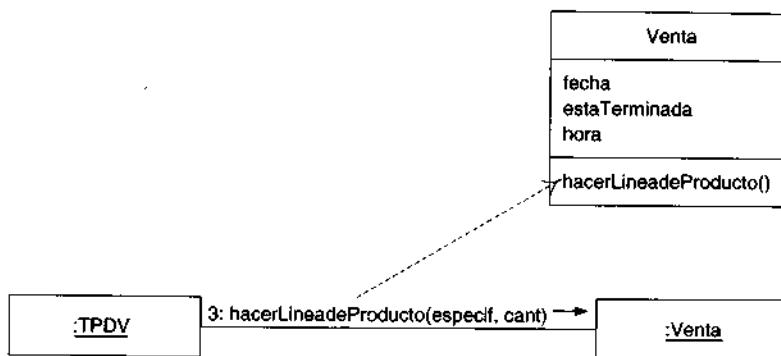


Figura 21.4 Nombres de los métodos tomados de los diagramas de colaboración.

En términos generales, el conjunto de los mensajes enviados a la clase X a través de los diagramas de colaboración indica la mayoría de los métodos que ha de definir la clase X.

Al observar detenidamente todos los diagramas de colaboración referentes a la aplicación del punto de venta, obtenemos la asignación de los métodos que aparecen en la figura 21.5.

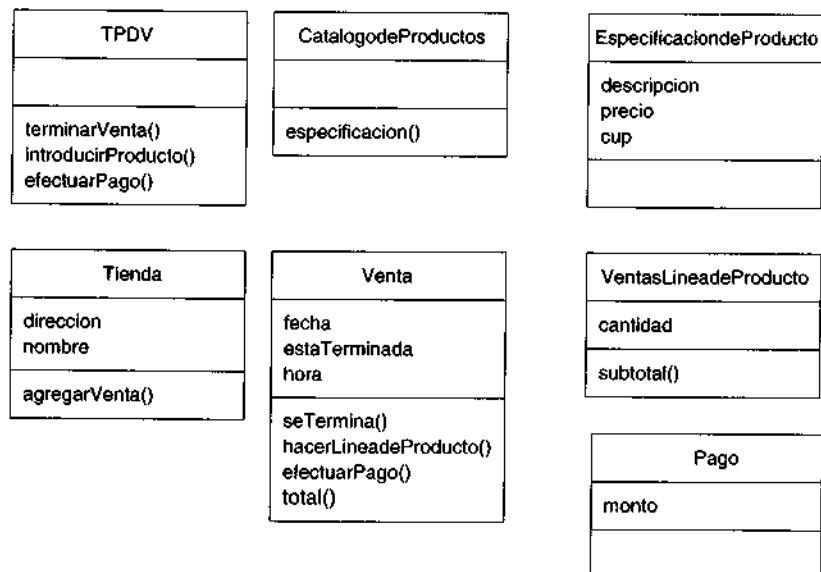


Figura 21.5 Métodos en la aplicación.

21.8.3 Nombres de los métodos: aspectos

Los siguientes aspectos especiales deben tenerse en cuenta respecto a los nombres de los métodos:

- Interpretación del mensaje *crear()*
- Descripción de los métodos de acceso
- Interpretación de los mensajes dirigidos a multiobjetos
- Sintaxis dependiente del lenguaje

21.8.4 Nombres de los métodos: crear

El mensaje *crear* es la forma independiente del lenguaje UML con que se indica instantiación e inicialización. Cuando traducimos el diseño a un lenguaje de programación orientado a objetos, hay que expresarlo en términos de sus expresiones de instantiación e inicialización. Ni Java, ni C++ ni tampoco Smalltalk cuentan con un método *crear*. Por ejemplo, en C++ implica una asignación automática o una asignación en el almacenamiento libre mediante el operador *new*, seguido de una llamada al constructor. En Java implica invocar el operador *new*, seguido de la llamada a un constructor.

Por sus interpretaciones tan heterogéneas, y también por ser la inicialización una actividad extraordinariamente común, se acostumbra omitir los métodos relacionados con la creación y los constructores procedentes del diagrama de clases del diseño.

21.8.5 Nombres de los métodos: métodos de acceso

Los **métodos de acceso** son aquellos que recuperan (método accesor) o establecen (método mutador) el valor de los atributos. Una estructura común consiste en contar con un accesor y con un mutador por cada atributo y declarar privados todos los atributos (para obligar el encapsulamiento). Normalmente estos métodos no se incluyen en la descripción del diagrama de clase por la elevada razón ruido a valor que generan: para N atributos hay $2N$ métodos sin el menor interés.

Por ejemplo, no se muestra el método *precio* de *Especificaciondeproductos* (*u obtenerPrecio*) aunque está presente, por ser *precio* un método accesor simple.

21.8.6 Nombres de los métodos: multiobjetos

Un mensaje a un multiobjeto se interpreta como si estuviera destinado al objeto contenedor/colección.

Por ejemplo, el siguiente mensaje *encontrar* dirigido a un multiobjeto ha de interpretarse como si estuviera destinado a un objeto contenedor/colección; por ejemplo, una *tabla Hashtable* (*Cálculo de Dirección*), un *map* (*mapa*) de C++ o un *Dictionary* (*Diccionario*) de Smalltalk (figura 21.6).

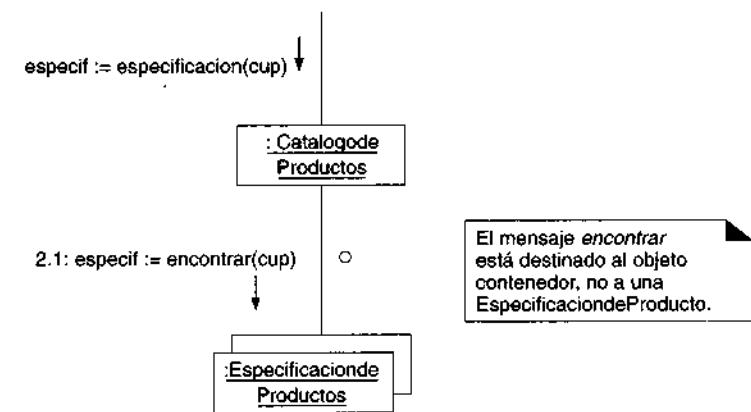


Figura 21.6 Mensaje a un multiobjeto.

Por tanto, el método *encontrar* no forma parte de la clase *EspecificaciondeProducto*, sino más bien de la definición de la clase *Hashtable* o *Dictionary*. Es, pues, incorrecto agregar *encontrar* como método a la clase *EspecificaciondeProducto*.

Estas clases contenedor/colección (como *java.util.Vector* y *java.util.Hashtable*) son clases predefinidas de las bibliotecas, y rara vez sirve mostrarlas explícitamente en el diagrama respectivo porque incorporan ruido y aportan escasa información nueva.

21.8.7 Nombres de los métodos: sintaxis dependiente del lenguaje

Algunos lenguajes, entre ellos Smalltalk, tienen una forma sintáctica para los métodos distintos al formato básico de *nombredelMetodo (listadeParametros)*. Recomendamos utilizar el formato básico UML, aun cuando el lenguaje de la implementación planeada aplique otra sintaxis. En teoría, la traducción debería tener lugar en el momento de generar el código y no al elaborar los diagramas de clase. Pese a ello, el lenguaje UML admite otra sintaxis para especificar el método, la de Smalltalk por ejemplo.

21.8.8 Agregar más información sobre los tipos

Es opcional mostrar el tipo de los atributos, los parámetros del método y los valores de devolver método. La cuestión de si debe incluirse o no esta información debería considerarse dentro del siguiente contexto:

El diagrama de clases orientado al diseño debería elaborarse teniendo en cuenta la audiencia.

- Si vamos a crearlo en una herramienta CASE con generación automática de código, se requerirán detalles completos y exhaustivos.
- Si vamos a crearlo para que lo lean los diseñadores del software, el exceso de detalles puede influir negativamente en la razón ruido a valor.

Por ejemplo, ¿es necesario mostrar toda la información sobre los parámetros y su tipo? Respuesta: depende de cuán evidente sea la información para los destinatarios.

En la figura 21.7, el diagrama de clases del diseño y destinado al punto de venta muestra información exhaustiva sobre los tipos.

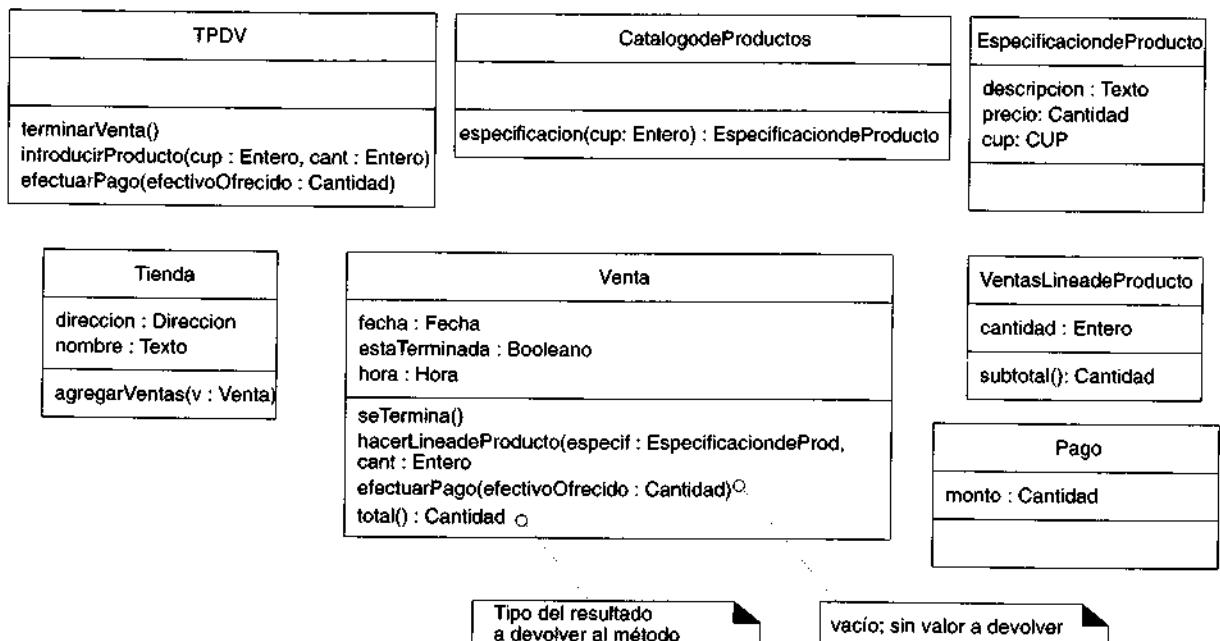


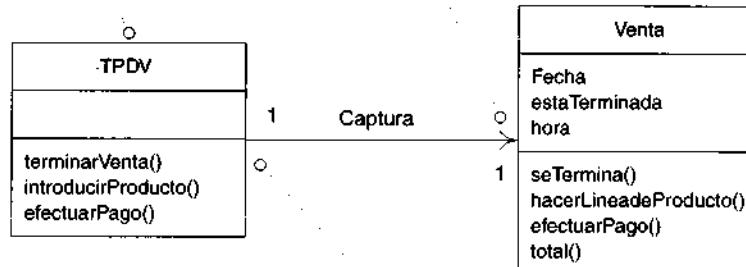
Figura 21.7 Incorporación de información sobre los tipos.

21.8.9 Incorporación de asociaciones y de navegabilidad

Se da el nombre de papel al fin de una asociación; en los diagramas de clases orientados al diseño podemos adornar el papel con una flecha de navegabilidad. La **navegabilidad** es una propiedad del papel e indica la posibilidad de navegar unidireccionalmente en una asociación, desde los objetos fuente hasta la clase destino. La navegabilidad significa visibilidad; generalmente la visibilidad de los atributos. Un ejemplo de ella se incluye en la figura 21.8.

La clase TPDV probablemente tenga un atributo que apunta a un objeto Venta.

La flecha de navegabilidad indica que los objetos TPDV están conectados unidireccionalmente con los objetos Venta.



La ausencia de la flecha de navegabilidad indica que no existe conexión de Venta a TPDV.

Figura 21.8 Descripción gráfica de la navegabilidad, o sea de la visibilidad de los atributos.

La interpretación usual de una asociación con una flecha de navegabilidad es la visibilidad de los atributos, desde la clase fuente hasta la clase destino. Cuando se implementa un lenguaje de programación orientado a objetos, suele traducirse como si la clase fuente tuviera un atributo que se refiere a una instancia de la clase destino. Por ejemplo, la clase TPDV definirá un atributo que referencia una instancia *Venta*.

En los diagramas de clase orientados al diseño, la mayoría de las asociaciones deberían completarse con las flechas necesarias de navegación.

En un diagrama de este tipo, las asociaciones se escogen con un riguroso criterio que esté orientado al software y que se rija por la necesidad de conocer: ¿cuáles asociaciones se requieren para satisfacer con los diagramas de interacción la visibilidad y las necesidades constantes de memoria? Esto contrasta con las asociaciones en el modelo conceptual, que pueden justificarse por la intención de mejorar la comprensión del dominio del problema. Una vez más capturamos una distinción entre las metas de los diagramas de clase orientados al diseño y el modelo conceptual: uno es analítico y el otro una descripción de los componentes del software.

La visibilidad y las asociaciones requeridas entre las clases se indican con los diagramas de interacción. A continuación se mencionan tres situaciones comunes que revelan la necesidad de definir una asociación con una flecha de navegabilidad de A a B.

- A envía un mensaje a B.
- A crea una instancia B.
- A necesita mantener una conexión con B.

Por ejemplo, en el diagrama de colaboración de la figura 21.9 —comenzando con el mensaje *crear* dirigido a una *Tienda* y a partir del contexto más amplio de otros diagramas de este tipo—, podemos observar que la *Tienda* probablemente tenga una conexión permanente con las instancias que genera. También es razonable que *CatalogodeProductos* necesite una conexión permanente con la colección de *EspecificacionesdeProducto* que genere. De hecho, el creador de otro objeto suele requerir ese tipo de conexión con él.

Por tanto, las conexiones en cuestión estarán presentes como asociaciones en el diagrama de clases.

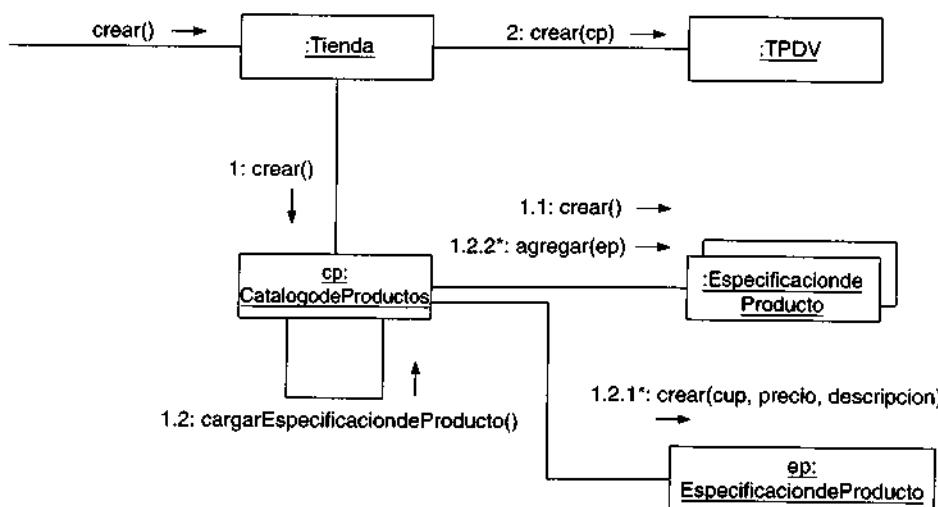


Figura 21.9 La navegabilidad se identifica a partir de los diagramas de colaboración.

Basándose en el criterio anterior de asociaciones y de navegabilidad, el análisis de los diagramas de colaboración obtenidos para la aplicación de punto de venta producirá un diagrama de clases (figura 21.10) con las siguientes asociaciones. (Por razones de claridad, se oculta la información exhaustiva sobre los tipos.)

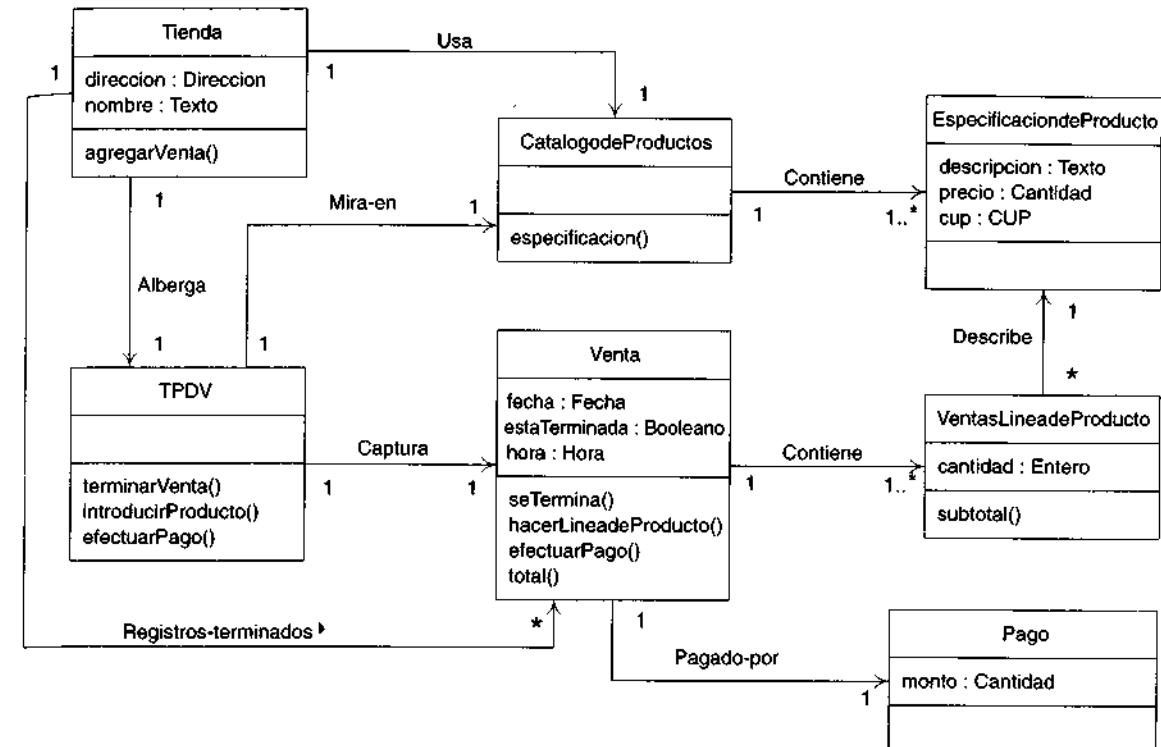


Figura 21.10 Asociaciones con símbolos de navegabilidad.

Nótese que éste no es exactamente el mismo conjunto de asociaciones que el generado para los diagramas de clases en el modelo de investigación. Por ejemplo, no existía la asociación *Mira-en* entre *TPDV* y *CatalogodeProductos* en el modelo conceptual: en ese momento no se descubrió que fuera una relación importante ni duradera. Pero durante la preparación de los diagramas de colaboración se decidió que el objeto de software *TPDV* debería tener una conexión permanente con un software *CatalogodeProductos* a fin de consultar *EspecificacionesdeProducto*.

21.8.10 Agregación de las relaciones de dependencia

El lenguaje UML incluye una **relación general de dependencia**, la cual indica que un elemento (de cualquier tipo como clases, casos de uso y otros) conoce la existencia de otro. Se denota con una línea punteada y con flecha. En el diagrama de clases, la relación de dependencia sirve para describir la visibilidad entre atributos que no se relaciona con ellos; en otras palabras, la visibilidad del parámetro global o declarada localmente. En cambio, la visibilidad simple de atributos se indica con una flecha normal de asociación y con una flecha de navegabilidad.

Por ejemplo, el objeto de software *TPDV* recibe un objeto devuelto del tipo *EspecificaciondeProducto* proveniente del mensaje de especificación que envió a un *CatalogodeProductos*. Así, *TPDV* tiene una visibilidad de corto plazo localmente declarada en *EspecificacionesdeProducto*. Y *Venta* recibe una *EspecificaciondeProducto* como parámetro en el mensaje *hacerLineadeProducto* tiene la visibilidad del parámetro en uno. Estas visibilidades no relacionadas con atributos pueden denotarse con la línea punteada y con flecha para indicar una relación de dependencia (figura 21.11). (La curvatura de las líneas de dependencia carece de importancia; en este ejemplo se usó por razones gráficas.)

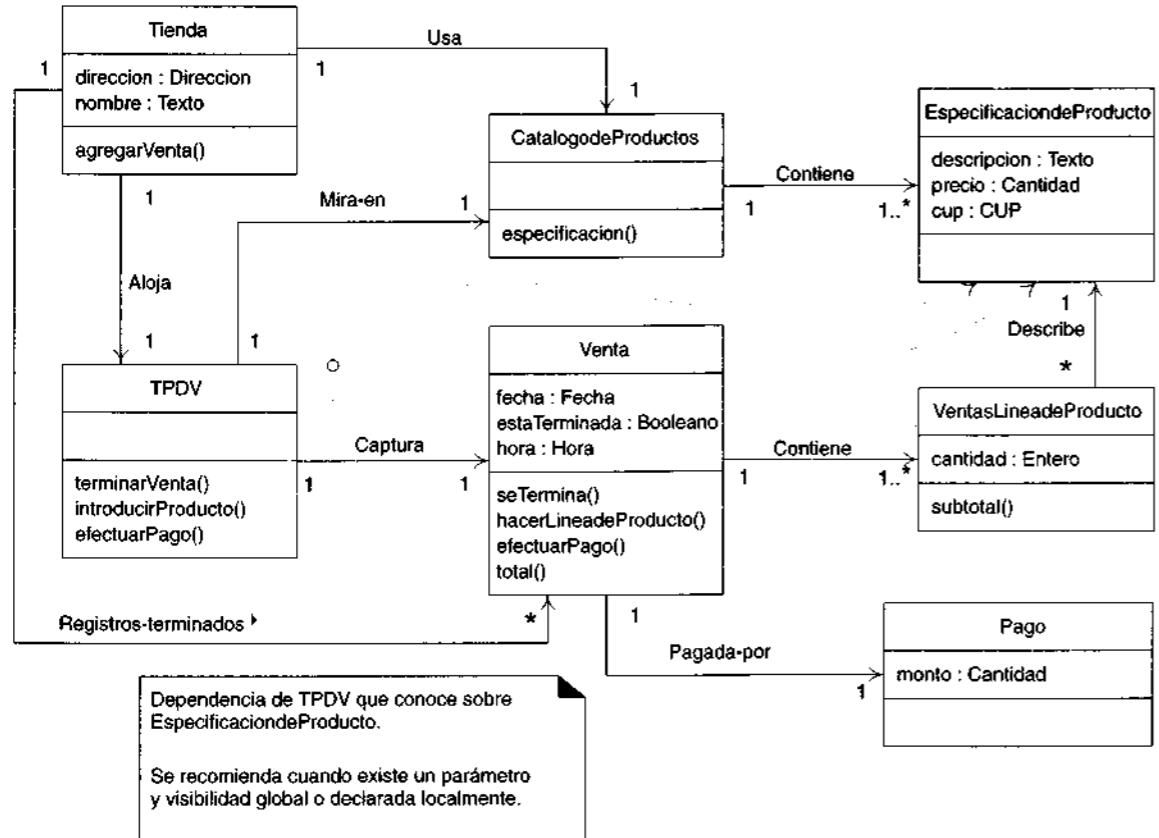


Figura 21.11 Relaciones de dependencia que indican una visibilidad no relacionada con atributos.

21.9 Notación de los detalles de los miembros

El lenguaje UML ofrece una rica notación para describir las características de los miembros de la clase e interfaz; por ejemplo, la visibilidad y los valores iniciales. Se supone que los atributos son privados por omisión. Un ejemplo de ello se da en la figura 21.12.

Nombre de la clase	java.awt.Font
<p>atributo</p> <p>atributo : tipo</p> <p>atributo : tipo = valor inicial</p> <p><u>atributodeClase</u></p> <p>/atributoDerivado</p> <p>...</p>	<p><u>plain</u> : Integer = 0</p> <p><u>bold</u> : Integer = 1</p> <p><u>name</u> : String</p> <p><u>style</u> : Integer = 0</p> <p>...</p>
<p>metodo1()</p> <p>metodo2(lista de parametros)</p> <p>: tipo de retorno</p> <p><i>metodoAbstracto()</i></p> <p>+metodoPublico()</p> <p>-metodoPrivado()</p> <p>#metodoProtegido()</p> <p><u>metododeClase()</u></p> <p>...</p>	<p>+<u>getFont(name : String) : Font</u></p> <p>+<u>getName() : String</u></p> <p>...</p> <p><i>java.awt.Toolkit</i></p> <p>...</p> <p>#<u>createButton(target : Button) : ButtonPeer</u></p> <p>...</p> <p>+<u>getColorModel() : ColorModel</u></p> <p>...</p>

Figura 21.12 Notación usada para los detalles de los miembros de la clase.

La iteración actual del diagrama de clases del diseño para el punto de venta (figura 21.13) no abarca muchos detalles interesantes referentes a los miembros; todos los atributos son privados y públicos todos los métodos.

<p>TPDV</p> <p>+terminarVenta() +introducirProducto() +efectuarPago()</p>	<p>CatalogodeProductos</p> <p>+especificacion()</p>	<p>EspecificaciondeProducto</p> <p>descripcion precio cup</p>
<p>Tienda</p> <p>direccion nombre</p> <p>+agregarVenta()</p>	<p>Venta</p> <p>fecha estaTerminada hora</p> <p>+seTermina() +hacerLineadeProducto() +efectuarPago() +total()</p>	<p>VentasLineadeProducto</p> <p>cantidad</p> <p>+subtotal()</p>
		<p>Pago</p> <p>monito</p>

Figura 21.13 Detalles de los miembros en el diagrama de clases para el punto de venta.

21.10 Modelos muestra

Los diagramas de clase del diseño constituyen el modelo de clases, o sea el de las clases de software y las interfaces.

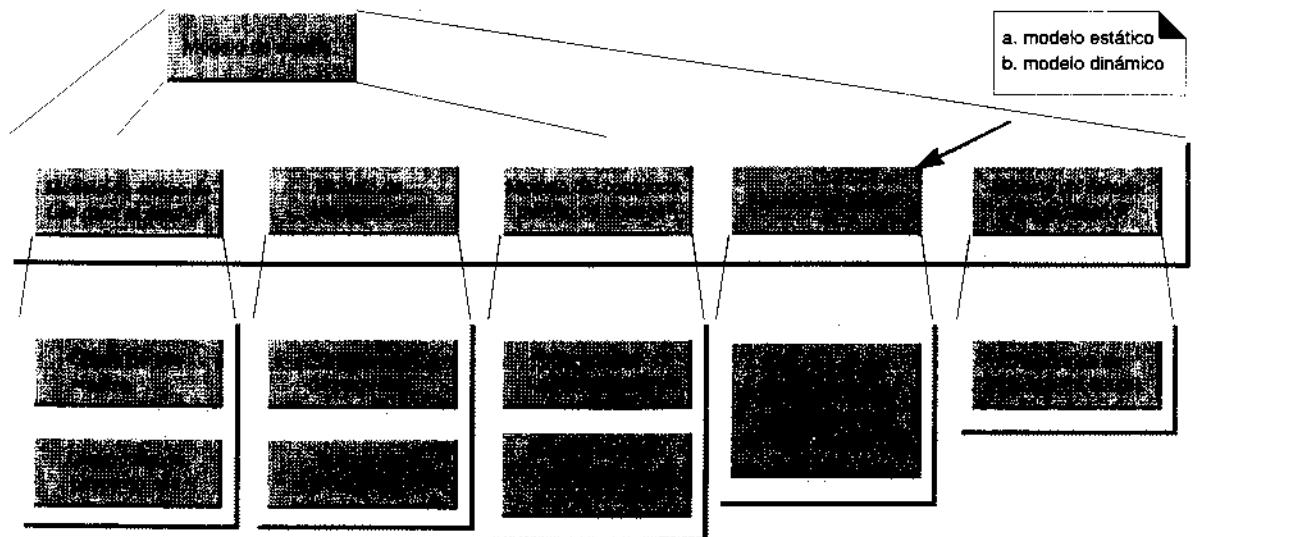


Figura 21.14 El modelo de diseño.

21.11 Resumen

Para la aplicación de la terminal situada en el punto de venta hemos preparado un diagrama de clases del diseño como parte del modelo del diseño que aparece en la figura 21.14. Ilustra la definición de las clases como componentes de software. Se trata de un tipo de diagramas sencillos y muy informativos, cualidades convenientes en la notación. Debido a tales cualidades, cuando se trabaja con un sistema orientado a objetos ya existente o cuando se diseña un nuevo sistema, los desarrolladores tienden a recurrir más a estos diagramas que a cualquier otro.

ALGUNOS ASPECTOS DEL DISEÑO DE SISTEMAS

Objetivos

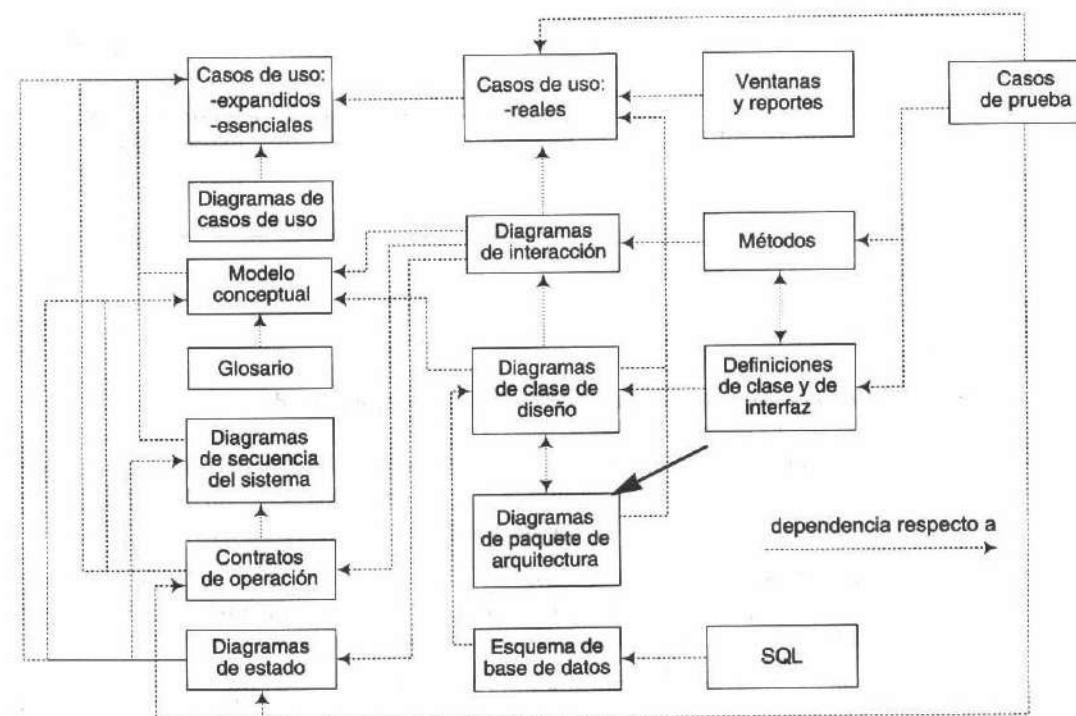
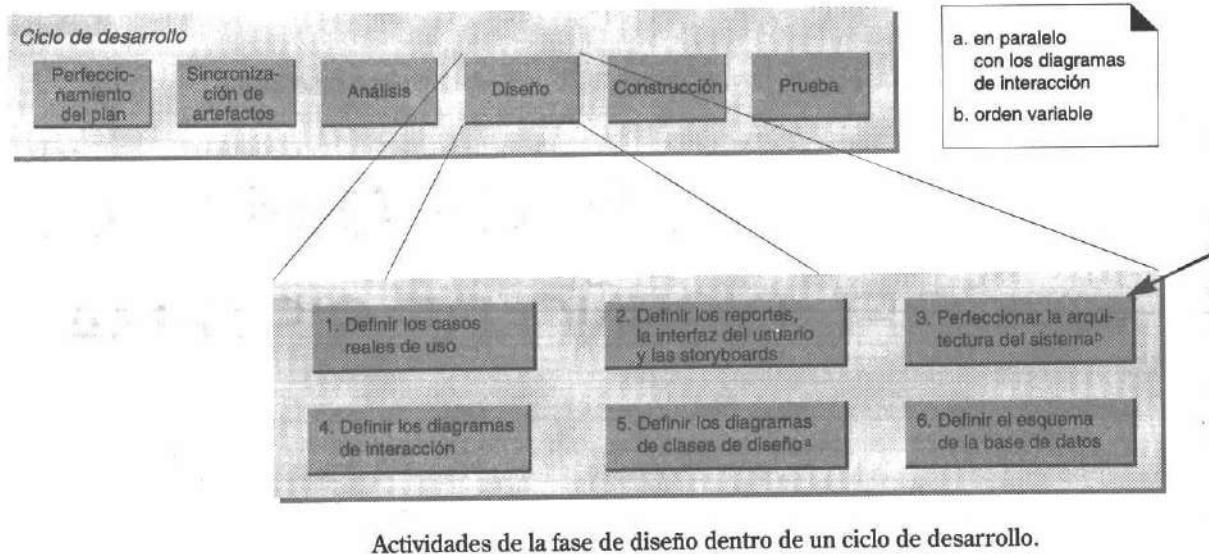
- Diseñar la arquitectura de un sistema a partir de las capas y de las particiones.
- Describir gráficamente el diseño arquitectónico mediante los diagramas de paquetes del UML.
- Aplicar los patrones Fachada, Separación de Modelo-Vista y Publicar-Suscribir, así como la notificación de eventos, para dar soporte a varias metas arquitectónicas.

22.1 Introducción

En el estudio anterior de casos nos centramos en los objetos del dominio del problema (entre ellos, *Venta*), porque así se definen los conceptos y el comportamiento básicos de un sistema. Pero un sistema se compone de muchos subsistemas, uno de los cuales son los objetos del dominio. Un sistema ordinario de información ha de conectarse a la interfaz del usuario y a un mecanismo de almacenamiento persistente.

En este capítulo examinaremos brevemente la arquitectura más amplia de un sistema, la comunicación y el acoplamiento entre subsistemas y la notación del UML—diagramas de paquetes—que nos permiten describir los subsistemas.

Los sistemas de información con una interfaz para el usuario (que entre otras cosas incluye software de aplicación como los procesadores de palabras) son la categoría más común de aplicación; de ahí que comencemos analizando las cuestiones más usuales de la arquitectura que se relacionan con ellos. Muchos de los conceptos aquí expuestos podrán utilizarse con otros tipos de aplicaciones.



Dependencias de los artefactos durante la fase de construcción.

22.2 Arquitectura clásica de tres capas

Una arquitectura común de los sistemas de información que abarca una interfaz para el usuario y el almacenamiento persistente de datos se conoce con el nombre de **arquitectura de tres capas** [Gartner95]; podemos observarla en la figura 22.1. He aquí una descripción clásica de las capas verticales:

1. **Presentación:** ventanas, reportes, etcétera.
2. **Lógica de aplicaciones:** tareas y reglas que rigen el proceso.
3. **Almacenamiento:** mecanismo de almacenamiento persistente.

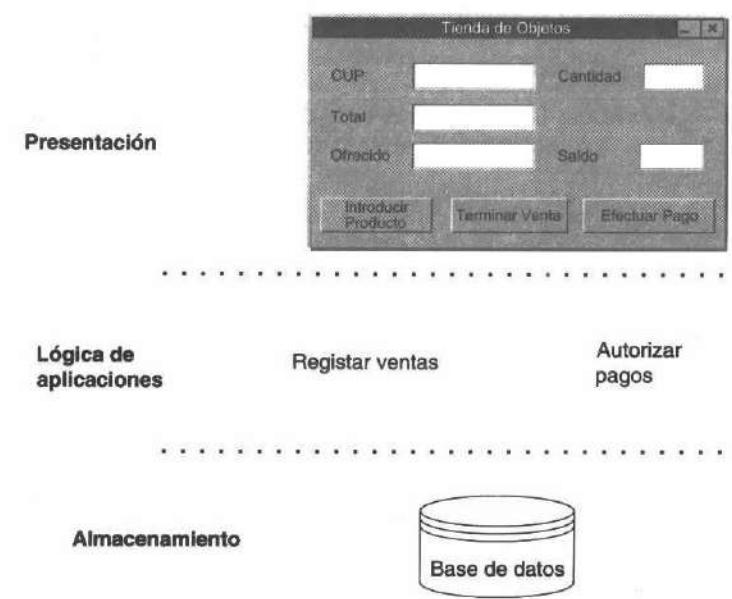


Figura 22.1 Vista clásica de una arquitectura de tres capas.

La calidad tan especial de la arquitectura de tres capas consiste en aislar la lógica de la aplicación y en convertirla en una capa intermedia bien definida y lógica del software. En la capa de presentación se realiza relativamente poco procesamiento de la aplicación; las ventanas envían a la capa intermedia peticiones de trabajo. Y éste se comunica con la capa de almacenamiento del extremo posterior.

Esta arquitectura contrasta con el diseño de **dos capas**, donde —por ejemplo— colocamos la lógica de aplicaciones dentro de las definiciones de ventana, que leen y escriben directamente en una base de datos; no hay una capa intermedia que separe la lógica. Una de sus desventajas es la imposibilidad de representar la lógica en componentes aislados, lo cual impide reutilizar el software. No es posible distribuir la lógica de aplicaciones en una computadora diferente.

22.3 Arquitecturas multicapas orientadas a objetos

Una arquitectura multicapas que se adecue a los sistemas de información orientados a objetos incluye la división de las responsabilidades que encontramos en la arquitectura clásica de tres capas. Las responsabilidades se asignan a los objetos de software.

22.3.1 Descomposición de la capa de la lógica de las aplicaciones

En un diseño orientado a objetos, la capa de la lógica de aplicaciones se divide en otras menos densas. Esta arquitectura, organizada a partir de las clases de software, se muestra en la figura 22.2. La capa de la lógica de aplicaciones está constituida por las siguientes capas:

- **Objetos del dominio:** clases que representan los conceptos del dominio; por ejemplo, una venta.
- **Servicios:** los objetos servicio de las funciones como la interacción de bases de datos, los reportes, las comunicaciones y la seguridad.

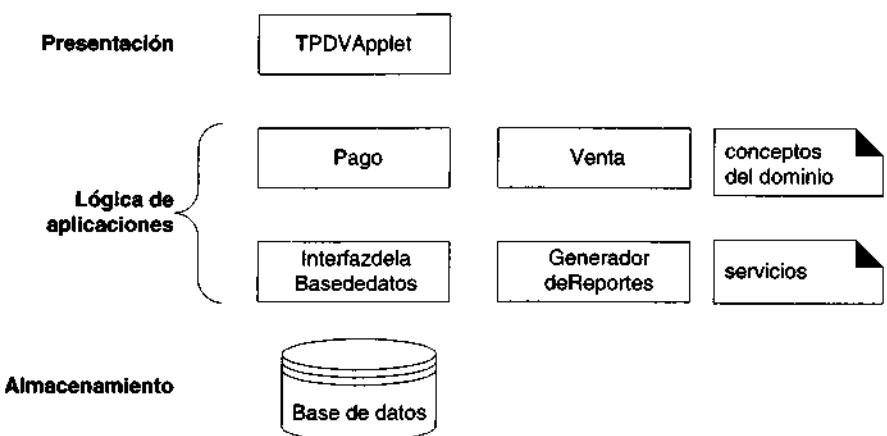


Figura 22.2 Descomposición de la capa de lógica de aplicaciones en estratos o subcapas más detalladas.

22.3.2 Más allá de las tres capas: arquitectura multicapas

Cuando vemos la arquitectura desde la perspectiva de una descomposición más detallada que se observa en la figura 22.2, podemos prescindir de la designación “arquitectura de tres capas” y hablar en cambio de **arquitecturas multicapas**; en ellas está implícita la capa intermedia de la lógica de aplicaciones.

Es posible agregar más capas y descomponer ulteriormente las ya existentes. Así, la capa *Servicios* puede dividirse en servicios de alto y bajo nivel (por ejemplo, generación de reportes frente a entrada/salida).

22.3.3 Despliegue

Una arquitectura *lógica* de tres capas puede desplegarse o implementarse *físicamente* en varias configuraciones, entre las cuales se encuentran:

1. Capas de la lógica de presentación y de aplicaciones en la computadora del cliente, en su almacenamiento o en su servidor.
2. La presentación en la computadora del cliente, la lógica de aplicaciones en un servidor de la aplicación y el almacenamiento en un servidor de datos independiente.

Dado el mayor uso de los lenguajes y tecnologías que soportan fácilmente el cómputo distribuido, Java entre otros, también el despliegue de los subsistemas se irá realizando de manera distribuida cada vez más.

22.3.4 Motivos para utilizar la arquitectura multicapas

Entre los motivos por los cuales se recurre a la arquitectura multicapas se cuentan los siguientes:

- Aislamiento de la lógica de aplicaciones en componentes independientes susceptibles de reutilizarse después en otros sistemas.
- Distribución de las capas en varios nodos físicos de cómputo y en varios procesos. Esto puede mejorar el desempeño, la coordinación y el compartir la información en un sistema de cliente-servidor.
- Asignación de los diseñadores para que construyan determinadas capas; por ejemplo, un equipo que trabaje exclusivamente en la capa de presentación. Y así se brinda soporte a los conocimientos especializados en las habilidades de desarrollo y también a la capacidad de realizar actividades simultáneas en equipo.

22.4 Cómo mostrar la arquitectura con paquetes de UML

El lenguaje UML ofrece el mecanismo **paquete** que permite describir los grupos de elementos o subsistemas. Un paquete es un conjunto de cualquier tipo de elementos de un modelo: clases, casos de uso, diagramas de colaboración u otros paquetes (los anidados). Un sistema puede examinarse íntegramente dentro del ámbito de un solo paquete de alto nivel: el paquete *Sistema*. El paquete define un espacio de un nombre anidado, de modo que los elementos del mismo nombre pueden duplicarse dentro de varios paquetes.

22.4.1 Notación de los paquetes de UML

Un paquete se muestra gráficamente como una carpeta con etiquetas (figura 22.3). Los paquetes subordinados se incluyen en su interior. El nombre del paquete se encuentra dentro de la etiqueta si el paquete describe sus elementos; en caso contrario estará en el centro de la misma carpeta.

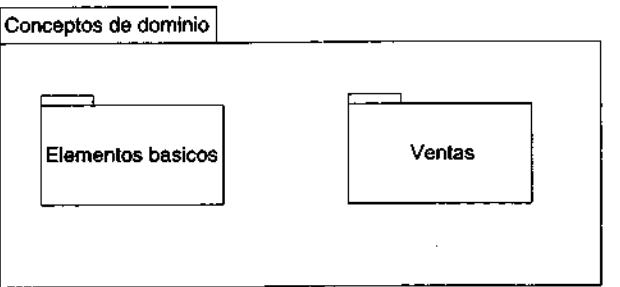


Figura 22.3 Paquetes en el lenguaje UML.

22.4.2 Diagramas de los paquetes descriptivos de la arquitectura

Los paquetes nos permiten describir la arquitectura de un sistema con la notación UML. En la figura 22.4 se incluye una vista alterna de la figura 22.2, que representa los agrupamientos lógicos mediante los diagramas de un paquete de UML. A este tipo de diagrama podemos llamarlo **diagrama de paquetes de la arquitectura**.

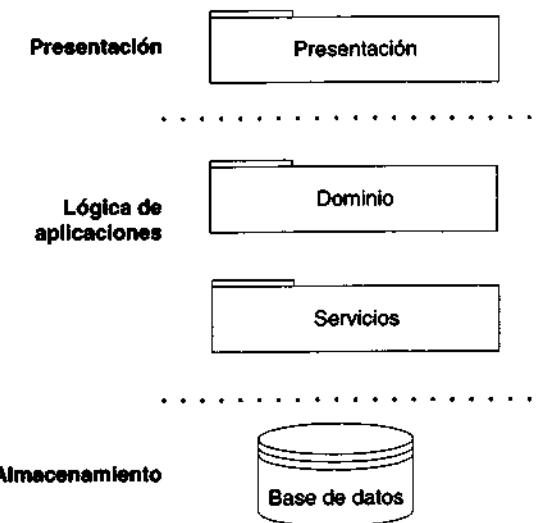


Figura 22.4 Unidades arquitectónicas expresadas en términos de los paquetes de UML.

22.4.3 Ejemplo de paquetes y de dependencias

La figura 22.5 contiene una descomposición más detallada de algunos paquetes comunes en la arquitectura de un sistema de información, así como las dependencias entre ellos.

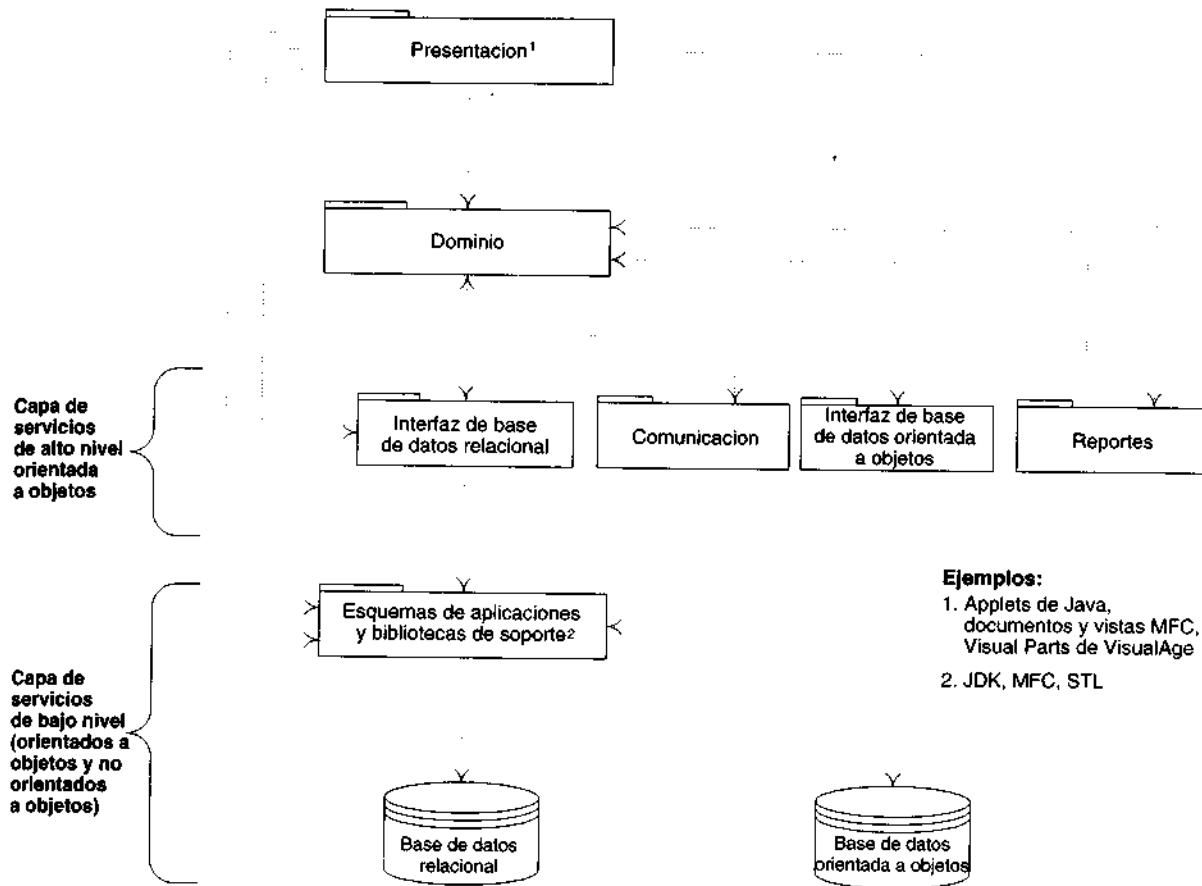


Figura 22.5 Arquitectura detallada.

Comentarios sobre los paquetes

- Los paquetes de la interfaz de bases de datos relacionales y de las orientadas a objetos ofrecen mecanismos para comunicarse con ellas. El proveedor suministrará una interfaz de base de datos orientada a objetos. Pero es necesario adaptar una interfaz de base de datos relacional o bien comprarle a otro proveedor un producto que desempeñe esa función.
- Entre los paquetes de servicios de alto nivel orientados a objetos podemos citar los siguientes: reportes, interfaces de bases de datos, seguridad y comunicaciones entre procesos preparados todos ellos con la tecnología de objetos. Normalmente son los diseñadores de aplicaciones quienes los escriben. En cambio, los servicios de bajo nivel ofrecen las funciones básicas, como la manipulación de ventanas y de archivos, y suelen hacerlo como bibliotecas estándar de lenguajes o adquirirse de otro proveedor.

- Las bibliotecas de soporte y de esquemas de aplicaciones casi siempre brindan soporte para crear ventanas, para definir los coordinadores de aplicaciones, acceder a bases de datos y a archivos, para la comunicación entre procesos, etcétera.

Comentarios sobre la dependencia

- Las relaciones de dependencia (línea punteada con flecha) indican si el paquete conoce el acoplamiento con otro. La *ausencia* de dependencia del paquete A con el B significa que los componentes de A *no* tienen referencias a ninguna clase, componentes, interfaz, método o servicios de B.
- Nótese que el paquete de dominio no tiene dependencia (acoplamiento) con las capas de presentación. Esto ejemplifica el principio de *Separación de Modelo-Vista* que comentaremos más adelante.

22.5 Identificación de los paquetes

Agrupe los elementos en un paquete aplicando la siguiente directriz:

Agrupe los elementos para ofrecer en un paquete un servicio común (o una familia de servicios relacionados), con un nivel relativamente alto de acoplamiento y colaboración.

En cierto nivel de abstracción, se verá el paquete como muy cohesivo (tiene responsabilidades estrechamente relacionadas).

En cambio, habrá relativamente bajo acoplamiento y colaboración entre los elementos de los paquetes.

22.6 Estratos y particiones

Podemos caracterizar una arquitectura multicapas como compuesta de estratos (subcapas) y particiones (figura 22.6).

Los **estratos** de una arquitectura representan las capas verticales, mientras que las **particiones** representan la división horizontal de subsistemas relativamente paralelos de un estrato. Por ejemplo, el estrato *Servicios* puede dividirse en particiones como *Seguridad* y *Reportes*.

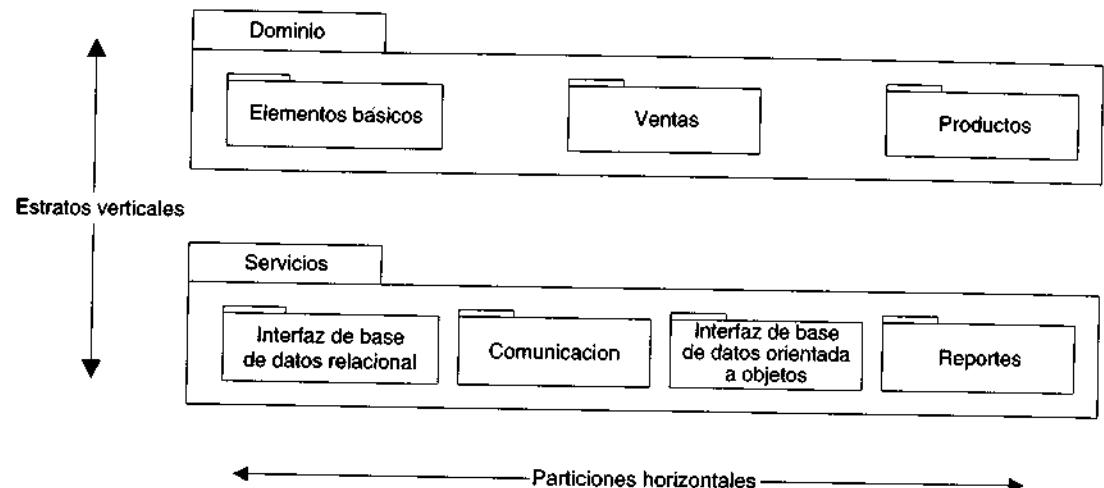


Figura 22.6 Estratos (subcapas) y particiones.

22.6.1 Arquitectura multicapas "laxa"

En las arquitecturas multicapas orientadas a objetos, los estratos *no* están acopladas en un sentido tan limitado como un protocolo de redes que se base en el modelo OSI de 7 capas. El modelo de protocolo comprende la restricción de que los elementos de la capa N sólo podrán acceder a los servicios de la capa o estrato inmediatamente inferior N-1. Aunque tal limitación es posible, la arquitectura suele ser de "capas laxas" o de "capas transparentes" [BMRSS96], donde los elementos de una capa o estrato se comunican con otras. La figura 22.5 presenta un acoplamiento ordinario entre estratos y particiones. Por ejemplo, el estrato de presentación (o de la coordinación de aplicaciones) normalmente está acoplada al de dominio y servicios.

22.7 Visibilidad entre las clases de paquetes

¿Cómo deberían relacionarse entre sí los componentes de un paquete? La visibilidad respecto a las clases en varios paquetes suele conformar el siguiente patrón:

- Acceso a los paquetes del dominio:** otros paquetes, generalmente el de presentación, tienen acceso a la visibilidad de *muchas* clases que representan los conceptos del dominio.
- Acceso a los paquetes de servicios:** otros paquetes, generalmente los de dominio y de presentación, tienen acceso a la visibilidad sólo respecto a *una* clase o unas cuantas en determinado paquete de servicios (casi siempre una clase *Fachada* como veremos luego).

- **Acceso a los paquetes de presentación:** ningún otro paquete tendrá acceso directo a la visibilidad de la capa de presentación. La comunicación es indirecta (mediante el patrón **Observador**, como veremos luego), si es que existe.

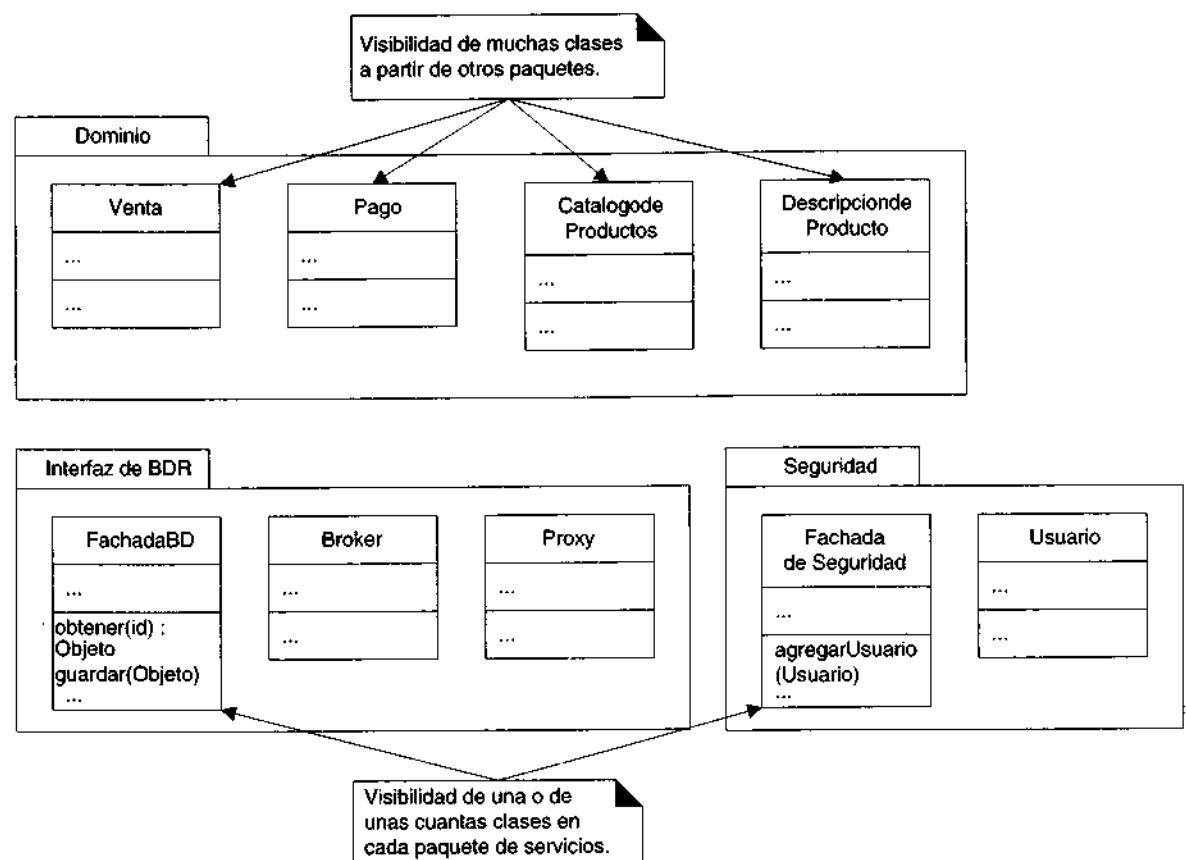


Figura 22.7 Visibilidad entre paquetes.

22.8 Interfaz de los paquetes de servicios: el patrón Fachada

¿Cómo deberían los componentes coordinarse con las clases de un paquete de servicios —por ejemplo, el paquete *InterfazBasededatosRelacional*, que ofrece servicios para realizar transformaciones entre los objetos y los renglones de las tablas? Cuando se define una clase con una interfaz común a un grupo de componentes o a un conjunto heterogéneo de interfaces, a esa clase le damos el nombre de **Fachada** [GHJV95]. Los elementos heterogéneos pueden ser las clases de un paquete, un conjunto de funciones, un esquema o un subsistema (local o remoto).

Fachada

Contexto/problema

Se requiere una interfaz común y unificada para un conjunto heterogéneo de interfaces, un subsistema por ejemplo. ¿Qué hacer?

Solución

Definir una clase individual que unifique la interfaz y le asigne la responsabilidad de colaborar con el subsistema.

El patrón **Fachada** a menudo sirve para suministrar una interfaz pública a un paquete de servicios. Por ejemplo, si se dispone de un paquete de *InterfazBasededatosRelacional* con muchas clases internas, podemos definir una clase como *FachadaBD* que ofrezca la interfaz pública común con los servicios del paquete (figura 22.7). Las clases de otros paquetes envían mensajes únicamente a una instancia de ella y no se acoplan a otras clases del paquete. *FachadaBD* colabora con otras clases (privadas) del paquete para prestar servicios. Este diseño repercute en un bajo acoplamiento.

22.9 Sin visibilidad directa respecto a las ventanas: el patrón de Separación Modelo-Vista

¿Qué tipo de visibilidad deberían tener otros paquetes respecto a la capa de presentación? ¿Cómo deberían comunicarse con las ventanas las clases no relacionadas con ellas? Por lo regular conviene que no haya un acoplamiento directo de otros componentes con los objetos ventana porque éstas se encuentran relacionadas con una aplicación en particular, mientras que (en teoría) podemos reutilizar en nuevas aplicaciones los componentes sin acceso a ellas o unirlos a una nueva interfaz. El principio que se aplica en este caso es el patrón Separación Modelo-Vista.

Dentro de este contexto, **modelo** es sinónimo de capa del dominio de objetos. **Vista** es sinónimo de objetos presentación, como ventanas, applets y reportes.

El patrón de **Separación Modelo-Vista**¹ (conocido también con el nombre de **Separación Dominio-Presentación**) establece que los objetos modelo (dominio) no deberían conocer *directamente* los objetos vista (presentación) ni estar directamente acoplados a ellos.

¹ Se describe totalmente como un patrón en [BMRSS96] con el nombre de *Modelo-Vista-Controlador*. En realidad, la parte del patrón correspondiente a *Controlador* (manejador de entrada de bajo nivel) es un anacronismo, ya que las responsabilidades del controlador suelen quedar incluidas dentro de las vistas y ser proporcionadas por el sistema operativo; los controladores se incluyeron hace años en Smalltalk-80 antes del advenimiento de los modernos sistemas operativos y los servicios gráficos de las ventanas. El patrón “Separación de Modelo-Vista” describe mejor el propósito y la arquitectura moderna.

Los métodos de una clase de modelo no deberían contener instrucciones que envíen mensajes directos a un objeto vista; este tipo de clase no debería conocer las interfaces del usuario ni relacionarse con ellas mediante códigos.

Una clase de modelos no tendrá visibilidad directa respecto a una clase vista. Pero, como veremos, se admite la visibilidad indirecta.

Una recomendación más de este patrón es que las clases del dominio (vista) encapsulen la información y el comportamiento relacionados con la lógica de aplicaciones.

Las clases de ventanas (vistas) son relativamente sencillas; se encargan de la entrada y de la salida, sin que conserven los datos ni ofrezcan directamente la funcionalidad de las aplicaciones.

Separación Modelo-Vista

Contexto/problema

Conviene desacoplar los objetos dominio (modelo) y las ventanas (vistas), a fin de brindar soporte a un mayor reuso de los objetos dominio y reducir al mínimo el impacto que los cambios de la interfaz tienen en ellos. ¿Qué hacer?

Solución

Definir las clases de dominio (modelo) para que no tengan acoplamiento ni visibilidad directa respecto a las clases ventana (vista) y para que los datos de la aplicación y de la funcionalidad se conserven en las clases de dominio, no en las de ventana.

Supongamos, por ejemplo, que (en Java) existe un objeto *TPDVApplet* que muestra información proveniente de un objeto *Venta* y que envía mensajes de eventos del sistema a un objeto *TPDV*.

El objeto *TPDVApplet* puede tener visibilidad respecto a los objetos dominio y enviarles mensajes; por ejemplo un objeto *TPDV* (el controlador de los eventos del sistema) y *Venta*.

Pero los objetos *TPDV* y *Venta* no conocerán la existencia de la ventana; ninguno de ellos contiene afirmaciones relacionadas con ventanas ni con la lógica de presentación.

Por tanto, no conviene que *TPDV* envíe un mensaje *presentarMensaje* a la ventana, como se observa en la figura 22.8.

Capa de presentación (vista)
(por ejemplo, *TPDVApplet*)

1:introducirProducto
(cup, cant)

Capa del dominio (modelo)

:TPDV

1: presentarMensaje(mens)

Bien.
Mensajes de la capa de vista a la de presentación.
Soporta la separación modelo-vista.

Mal.
No es conveniente enviar mensajes ni realizar el acoplamiento de la capa de modelo a la de vista.

Figura 22.8 Comparación entre la conformidad y la violación del patrón Separación Modelo-Vista.

En síntesis, los objetos de la capa de dominio no tienen comunicación directa con la de presentación.

En cambio, es aceptable que las vistas tengan visibilidad respecto a los modelos; por ejemplo, un objeto *TPDVApplet* puede enviar mensajes al objeto *TPDV*.

22.9.1 Motivos del patrón de Separación Modelo-Vista

Entre las razones por las cuales se emplea el patrón de separación Modelo-Vista, podemos citar las siguientes:

- Dar soporte a las definiciones de modelos cohesivos que se centran en los procesos de dominio más que en las interfaces de computadora.
- Permitir desarrollar independientemente el modelo y las capas de interfaz para el usuario.
- Reducir al mínimo el impacto que los cambios de requerimientos de la interfaz tienen en la capa de dominio.

- Permitir conectar fácilmente otras vistas a la capa actual de dominio, sin que esto la afecte.
- Permitir vistas simultáneas y múltiples del mismo objeto modelo; por ejemplo, la vista de la estadística de ventas en un diagrama tabular.
- Permitir ejecutar la capa del modelo independiente de la capa de interfaz para el usuario; por ejemplo, en un sistema de procesamiento de mensajes o de modo lotes.
- Permitir transportar fácilmente la capa de modelo a otro esquema de interfaz para el usuario.

22.9.2 El patrón de Separación Modelo-Vista y la comunicación indirecta

¿Cómo obtienen las ventanas información para mostrarla? Generalmente basta que envíen mensajes a los objetos de dominio, solicitando la información que después despliegan en dispositivos: un modelo de **escrutinio** o de **jalar desde arriba** de las actualizaciones de la presentación visual.

No obstante, en ocasiones un modelo de escrutinio resulta insuficiente: los objetos de dominio necesitan comunicarse (indirectamente) con las ventanas para lograr una actualización permanente y de tiempo real de la presentación visual, a medida que vaya cambiando el estado de la información de los objetos del dominio. En tal caso suelen ocurrir las siguientes situaciones:

- Aplicaciones del monitoreo; por ejemplo, administración de las redes de telecomunicaciones.
- Aplicaciones de simulación que requieren visualización; por ejemplo, modelos aerodinámicos.

En las situaciones anteriores, se requiere un modelo de **empujar desde abajo** de la actualización de la presentación visual. Dada la restricción del patrón de Separación Modelo-Vista, esto hace necesaria la comunicación *indirecta* entre otros objetos y ventanas: empujar hacia arriba la notificación para actualizar desde abajo, aunque en forma indirecta.

22.10 La comunicación indirecta en un sistema

El patrón de Separación Modelo-Vista es uno de muchos ejemplos en que debe darse una comunicación indirecta entre los elementos del sistema. En los lenguajes comunes orientados a objetos, un mensaje entre un objeto emisor y un objeto receptor exige que el emisor tenga visibilidad directa del receptor. En consecuencia, hacen falta otros mecanismos además del envío directo de mensajes, si conviene desacoplar al emisor y al receptor o si se necesita brindar soporte a la transmisión de una "señal".

En esta sección examinaremos las variaciones de la comunicación indirecta y la manera en que dan soporte a las metas arquitectónicas, entre ellas el bajo acoplamiento y la Separación Modelo-Vista.

22.10.1 El patrón Publicar-Suscribir

La solución de la comunicación indirecta que se requiere de los objetos de dominio a las ventanas es el patrón **Observador**, llamado también **Publicar-Suscribir**. Este método se explica en [GHJV95] con el nombre de **Observador** y en [BMRSS96] como **Editor-Suscriptor**.

Publicar-Suscribir

Contexto/problema

Un cambio de estado (un evento) ocurre dentro de un Editor del evento y otros objetos dependen de él o están interesados en él (suscriptores del evento). Pero el Editor no debería tener conocimiento directo de sus Suscriptores. ¿Qué hacer?

Solución

Defina un sistema de notificación de eventos para que el Editor pueda notificarlos indirectamente a los Suscriptores.

Por ejemplo, podemos definir una clase *AdministradordeEventos* que conserve los mapos entre eventos y suscriptores (es a la vez una *Fabricación Pura* y una *Indirección* en el sentido de los patrones GRASP). El editor publica un Evento enviando un mensaje *señaldeEventos* al *AdministradordeEventos*. Cuando se publica, éste encuentra todos los suscriptores a quienes interesa el evento y les envía una notificación, en teoría a través de un mensaje parametrizado, u objeto *respuesta* que dan al hacer la suscripción (figura 22.9). El evento se representa como una cadena simple; lo más probable es que se trate de una instancia de una clase *Evento*. En un sistema de eventos creado en casa, normalmente habrá una sola instancia de la clase *AdministradordeEventos*, a la cual se accede globalmente por medio del patrón Singleton (que se explica más adelante).

El diseño de *AdministradordeEventos* es un ejemplo de conceptos simple e independiente del lenguaje; deben utilizarse las herramientas propias del lenguaje cuando se disponga de ellas. Así, Java especifica un modelo del evento delegación que soporta las metas de publicar-suscribir. Los objetos suscriptor implementan una interfaz *Oyente de Eventos* y se suscribe a los eventos al registrarse con los objetos emisor de los eventos (los editores). Los mismos objetos emisor conservan una lista de todos los oyentes registrados y les dan una notificación siempre que ocurre un evento de interés para ellos. El acoplamiento se reduce al definir los suscriptores con una interfaz de Java y no con una clase.

Un diseño clásico de Publicar-Suscribir hace que el objeto editor tenga visibilidad directa de su colección de suscriptores a quienes interesa que se les notifiquen los cambios del editor. Esto tiene la desventaja de impactar el acoplamiento y la implementación de los editores. En cambio, una clase indirecta de *AdministradordeEventos* que cumpla esta función con los suscriptores ofrece la ventaja de reducir al mínimo el acoplamiento y las responsabilidades de los editores aunque puede ocasionar problemas de desempeño, pues es un cuello de botella de todos los eventos.

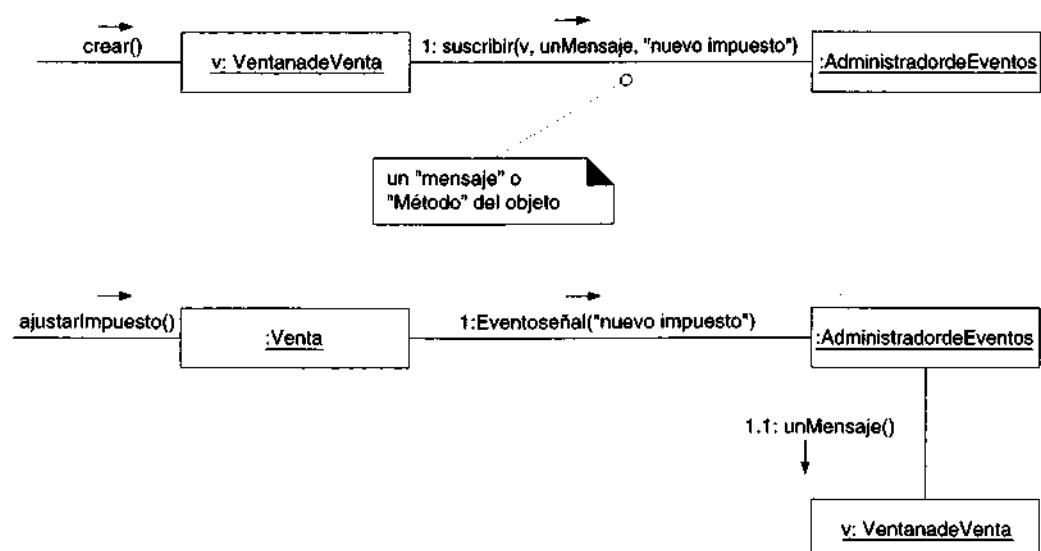


Figura 22.9 Publicar-Suscribir con notificación de los eventos.

22.10.2 Respuestas

El diseño de la figura 22.9 exige transmitir un mensaje parametrizado y un receptor potencial al `:AdministradordeEventos`. Podemos encapsular el mensaje y el receptor dentro de una clase de *Respuesta*.¹ Después al `:AdministradordeEventos` se le transmite una instancia *Respuesta*, que ejecuta al señalar un evento enviándole a la *Respuesta* un mensaje *ejecutar*.²

El uso de los objetos *Respuesta* tiene la ventaja de ocultar los detalles del receptor y del mensaje, además de que simplifica las responsabilidades del usuario de la *Respuesta*.

22.10.3 Sistemas de notificación de eventos

La arquitectura de Publicar-Suscribir proporciona un mecanismo de propósito general para la notificación de eventos y la comunicación indirecta en un sistema. Se origina así un nuevo método de diseñar los sistemas orientados a objetos: un diseño basado en la notificación de eventos. He aquí sus cualidades distintivas:

¹ Por ejemplo, utilizar el API Reflection de Java o una instancia de clase interna de este lenguaje.

² En Java, el diseño presenta variaciones que no requieren una clase explícita de *Respuesta*. Esto es posible con interfaces, con las clases internas anónimas y también con el modelo de eventos de delegación.

- No se requiere el acoplamiento directo entre emisores y receptores.
- Un evento individual puede transmitirse a cualquier número de suscriptores.
- Puede generalizarse la reacción ante un evento en los objetos *Respuesta*.
- Es relativamente fácil lograr la concurrencia con sólo ejecutar cada *Respuesta* en su propio subproceso múltiple (thread).

Además de las ventajas de un menor acoplamiento, de la transmisión y de una concurrencia simple, algunas veces un sistema basado en la notificación de eventos alcanza mayor eficiencia cuando las largas cadenas de mensajes pueden compactarse en un nivel de indirección.

Finalmente, un diseño de sistemas que se base principalmente en notificar asincrónicamente los eventos y en transmitírselos a los suscriptores va adquiriendo cada vez más la forma de un diseño de estado-máquina. En un capítulo posterior trataremos el tema de la modelación del estado de los objetos.

22.11 Coordinadores de las aplicaciones

El coordinador de aplicaciones es una clase que tiene la responsabilidad de mediar entre la capa de interfaz (ventanas, por ejemplo) y la del dominio. Tiene las siguientes responsabilidades fundamentales:

- Mapear la información entre los objetos del dominio y la interfaz. Por ejemplo, consolidar y transformar la información proveniente de uno o varios objetos del dominio.
- Responder a los eventos procedentes de la interfaz.
- Abrir ventanas que muestren la información proveniente de los objetos del dominio.
- Administrar las transacciones; por ejemplo, realizar las operaciones "commit" y "rollback".

Si la arquitectura tiene por objeto soportar vistas o ventanas múltiples que despliegan la misma información, los coordinadores de aplicación brindarán además soporte a las siguientes responsabilidades de vistas múltiples:

- Soportar la capacidad de que varias ventanas ("ventanas dependientes") muestren simultáneamente la información procedente de una instancia coordinador de aplicaciones.
- Notificarles a las ventanas dependientes cuando cambie la información y cuando las ventanas necesitan refrescarse (repintarse) con nueva información.

Algunos esquemas de aplicaciones incluyen el soporte de alguna modalidad de coordinador. Por ejemplo, en la aplicación MFC los coordinadores son *Documentos* en la arquitectura Documento-Vista; son subclases de la clase *CDocument*.¹

¹ En Smalltalk de VisualWorks, un *ApplicationModel* desempeña el papel de un coordinador de aplicaciones. En Visual Smalltalk, esto lo hace un *ViewManager* o *ApplicationCoordinator*.

En la figura 22.10 encontrará el lector un ejemplo concreto de la relación existente entre varios componentes y la arquitectura Documento-Vista de MFC. El atributo visibilidad se indica gráficamente en los enlaces del diagrama de colaboración. Adviéntase que —en apoyo del patrón de Separación Modelo-Vista—, los objetos de dominio *TPDV* y *Venta* no tienen visibilidad de *DocumentodeVenta* ni *VistadeVenta*.

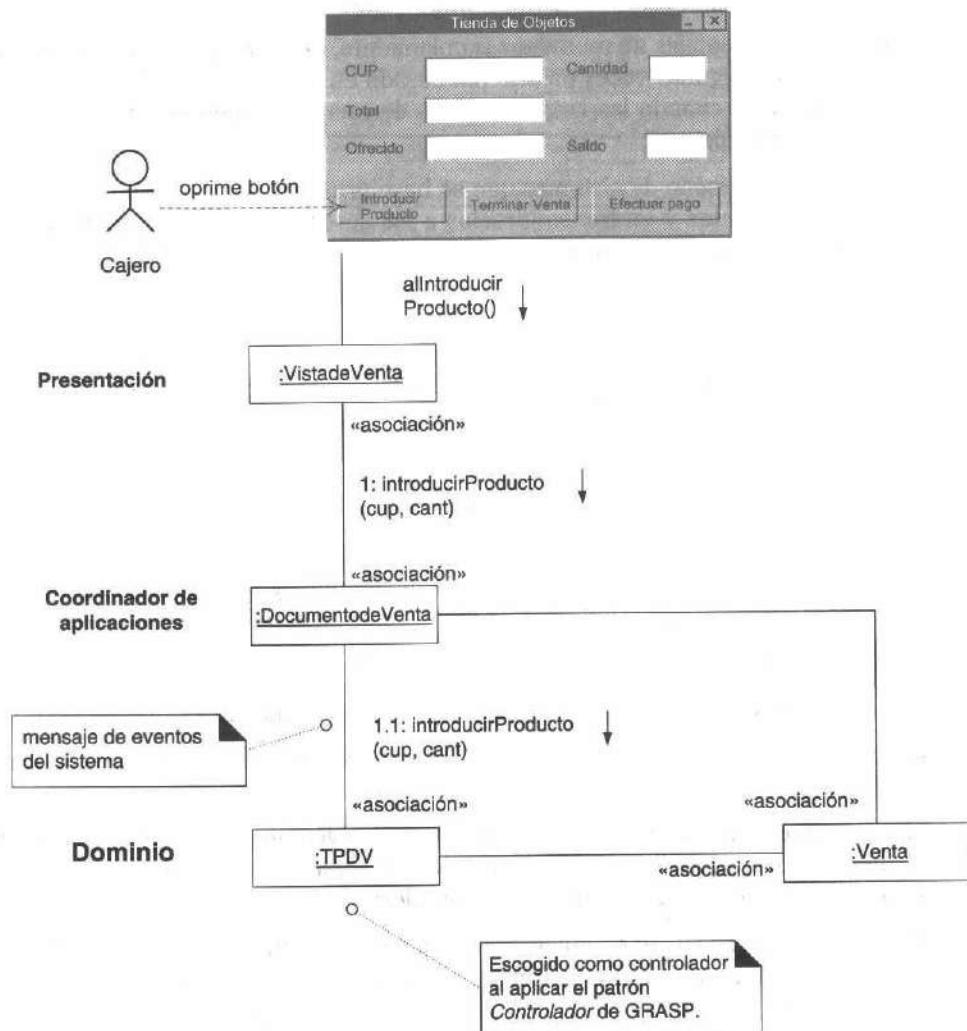


Figura 22.10 Colaboración y visibilidad entre objetos en varias capas de la arquitectura de Documento-Vista de Clases de la Microsoft Foundation (MFC).

El *DocumentodeVenta* (el coordinador de aplicaciones) tiene visibilidad de *TPDV* como el controlador escogido (en el sentido de los patrones GRASP) para los mensajes de operación del sistema y también tiene visibilidad de la *Venta* para la cual se va a mostrar información en la ventana *VistadeVenta*. El *DocumentodeVenta* estableció el atributo de la visibilidad de la *Venta* al solicitársela a *TPDV*, el creador de ella.

Considerese el siguiente ejemplo de flujo de control:

1. El cajero oprime un botón de la ventana.
2. El evento se envía como mensaje *allIntroducirProducto* a la instancia *VistadeVenta*, que representa la ventana exhibida.
3. La instancia *VistadeVenta* envía a *DocumentodeVenta* el evento como el mensaje *introducirProducto*.

DocumentodeVenta envía el mensaje *introducirProducto* (el mensaje de eventos del sistema) al controlador *TPDV* en la capa del dominio.

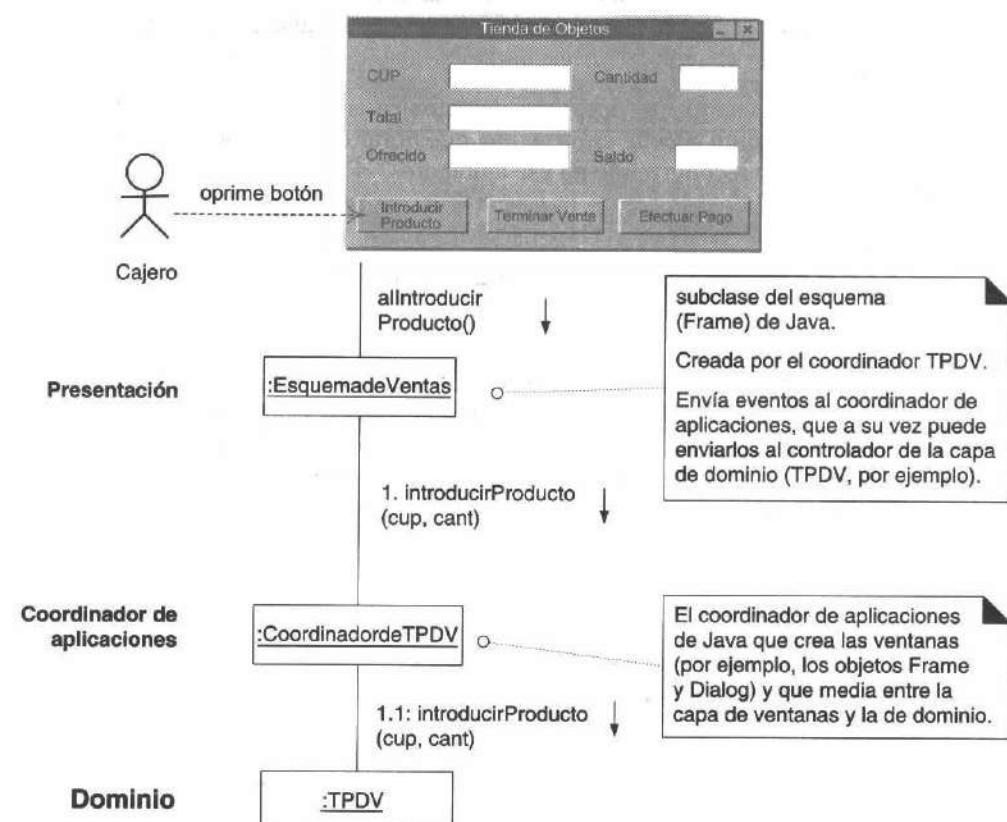


Figura 22.11 Agregado de un coordinador de aplicaciones a una aplicación de Java.

22.11.1 Los coordinadores de aplicaciones y los objetos ventana

Cuando los coordinadores de aplicaciones no cuentan con un soporte explícito de esquema, es común que la clase ventana o vista asuman algunas de las responsabilidades de ellos. Por ejemplo, JDK de Java no ofrece una coordinación in-

dividual de aplicaciones a partir de la presentación. Si se utiliza un applet de Java, casi siempre cumple con las responsabilidades de coordinar la presentación y la aplicación.

22.11.2 Coordinadores de las aplicaciones diseñados en casa

Aun cuando los coordinadores de aplicaciones carezcan del soporte de un esquema, podemos incorporarlos para llevar a cabo la administración de transacciones, notificar muchas ventanas dependientes y mapear la información.

Por ejemplo, podemos diseñar una aplicación (no un applet) de modo que la clase *CoordinadordeTPDV*, que representa al coordinador de aplicaciones, medie entre ventanas (por ejemplo, un *Esquema*) y la capa del dominio (figura 22.11).

Si no se incluye el soporte arquitectónico anterior para los coordinadores de aplicaciones, como sucede en Java, hay que obrar con cautela al incorporarlos, pues son un elemento incongruente en ese ambiente.

22.12 Almacenamiento y persistencia

Si se emplea una base de datos de objetos para guardar los objetos capa de dominio, los diseñadores no tendrán que hacer mucho esfuerzo para interactuar con ella: el proveedor les proporciona una interfaz. Pero si utiliza una base de otro tipo (por ejemplo, una base de datos relacional), se requerirá un subsistema (figura 22.12) que genere un mapeo entre los objetos y los renglones de las tablas. En el capítulo 38 veremos cómo se diseña.

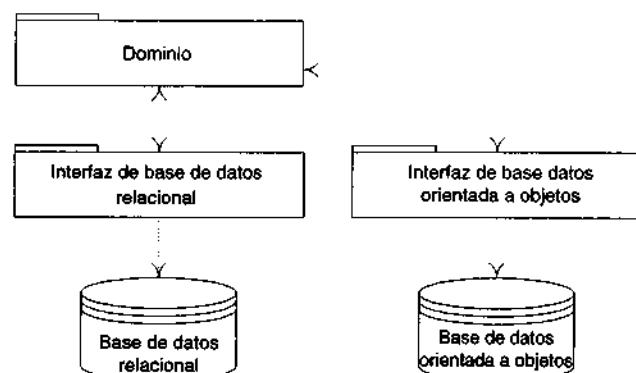


Figura 22.12 Subsistemas de persistencia.

22.13 Modelos muestra

Los diagramas de paquetes arquitectónicos son una parte del Modelo de Arquitectura: el de capas y subsistemas del sistema y del despliegue de los componentes para los procesadores.

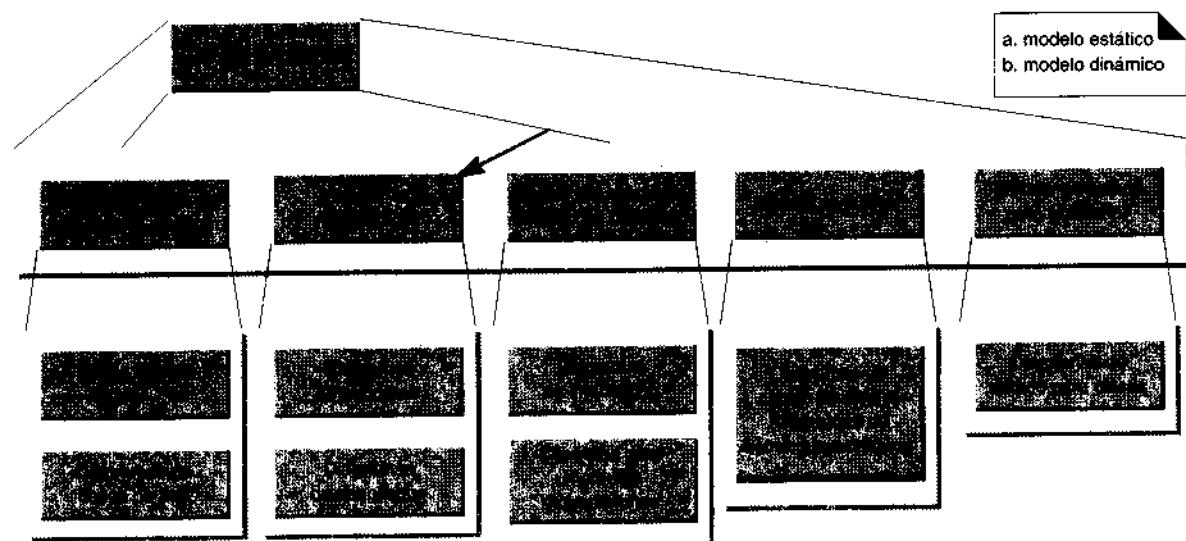


Figura 22.13 El modelo de diseño.

**PARTE V FASE DE
CONSTRUCCIÓN (1)**

MAPEO DE LOS DISEÑOS PARA CODIFICACIÓN

Objetivos

- Mapear los artefactos del diseño para realizar la codificación en un lenguaje orientado a objetos.

23.1 Introducción

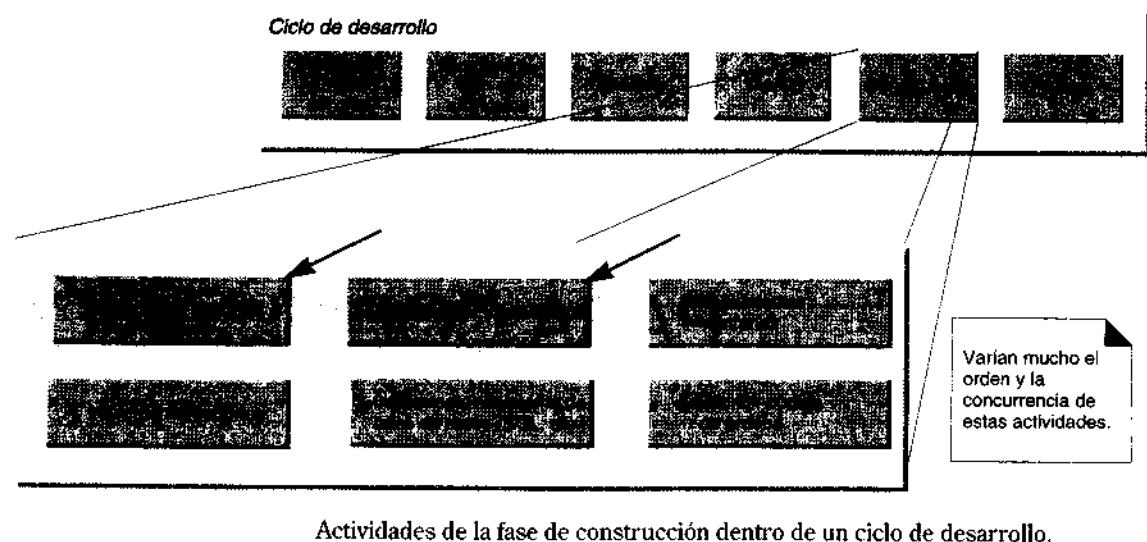
Una vez concluidos los diagramas de clases del diseño y destinados al ciclo de desarrollo actual en la aplicación punto de venta, dispondremos de suficientes detalles para generar un código que utilizaremos en la capa del dominio de los objetos.

Los artefactos del UML creados en la fase de diseño —los diagramas de colaboración y los de clases del diseño— servirán de entrada en el proceso de generación del código.

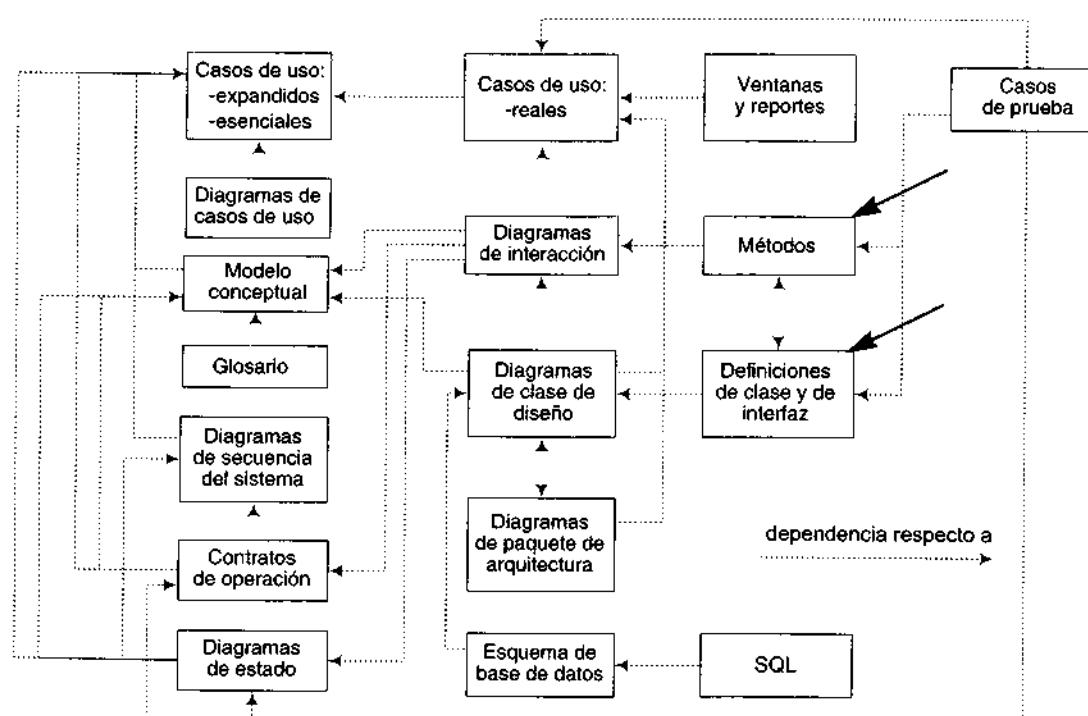
23.2 La programación y el proceso de desarrollo

Si se quiere reducir el riesgo y aumentar la probabilidad de conseguir una aplicación adecuada, el desarrollo debería basarse en un suficiente modelado del análisis y diseño antes de iniciar la codificación.¹ Ello no significa que durante la programación no tengan cabida los prototipos ni el diseño: las modernas herramientas del desarrollo ofrecen un excelente ambiente para examinar rápidamente métodos alternos, y normalmente vale la pena dedicar poco o mucho tiempo al diseño por la codificación.

¹ “Software” es una palabra que no hace mucho honor a su significado (“blando”). La modificación y el examen de alternativas son más difíciles y costosos durante la fase de programación que durante la de análisis y diseño.



Actividades de la fase de construcción dentro de un ciclo de desarrollo.



Dependencias de los artefactos durante la fase de construcción.

Pero la parte esencial de la aplicación —por ejemplo, el modelo conceptual básico, las capas arquitectónicas, las principales asignaciones de responsabilidades y las interacciones más importantes de los objetos— se determina más satisfactoriamente en una investigación formal y en el proceso de diseño que “apresurándose a codificar”. Esto último origina sistemas que son más difíciles de entender, ampliar y de darles mantenimiento, sin que brinden soporte a un proceso susceptible de repetirse eficazmente.

Dicho lo anterior, a menudo lo más conveniente es omitir la fase de diseño para realizar un poco de programación exploratoria a fin de descubrir un diseño funcional y luego regresar a la fase formal de diseño.

La creación de código en un lenguaje orientado a objetos —Java y Smalltalk, entre otros— no forma parte del análisis ni del diseño orientados a objetos; es meta final en sí misma. Los artefactos producidos en la fase de diseño suministran suficiente información necesaria para generar el código y, como veremos luego, se trata de un proceso de traducción relativamente simple.

Una ventaja del análisis, del diseño y de la programación orientados a objetos —cuando se emplean junto con un proceso de desarrollo como el que hemos recomendado— es que ofrece una guía completa de principio a fin para realizar la codificación a partir de los requerimientos. Los artefactos que se introducen en otros posteriores en una forma rastreable y útil culminarán finalmente en una aplicación funcional. Con ello no queremos decir que el camino sea fácil ni que podamos seguirlo mecánicamente, pues existen demasiadas variables. Pero la guía constituye un punto de partida para experimentar e intercambiar opiniones.

23.2.1 Creatividad y cambio durante la fase de construcción

La toma de decisiones y el trabajo creativo se realizaron en gran medida durante las fases de análisis y de diseño. En la siguiente exposición veremos que la generación del código —en nuestro ejemplo— es un proceso de traducción bastante mecánico.

No obstante, en general la fase de programación no es un paso fácil de la generación del código, todo lo contrario. En realidad los resultados obtenidos durante el diseño son un primer paso incompleto; en la programación y en las pruebas se efectuará multitud de cambios y se descubrirán y resolverán problemas detallados.

Los artefactos del diseño, cuando están bien hechos, producen un núcleo flexible que crece con elegancia y fuerza para atender los nuevos problemas que surjan en la programación.

En consecuencia, espere y planeé para afrontar el cambio y la desviación respecto al diseño durante la fase de construcción y de pruebas.

23.2.2 Cambios de código y el proceso iterativo

Una de las ventajas del proceso de desarrollo iterativo e incremental es la posibilidad de introducir los resultados de un ciclo anterior al iniciar el siguiente. Así pues, los resultados del análisis y diseño subsecuentes se perfeccionan sin cesar y aprovechan

el trabajo de la implementación precedente. Por ejemplo, cuando el ciclo N se desvía de su diseño (lo cual ocurrirá inevitablemente), el diseño final basado en la implementación puede utilizarse en los modelos de análisis y diseño del ciclo N+1.

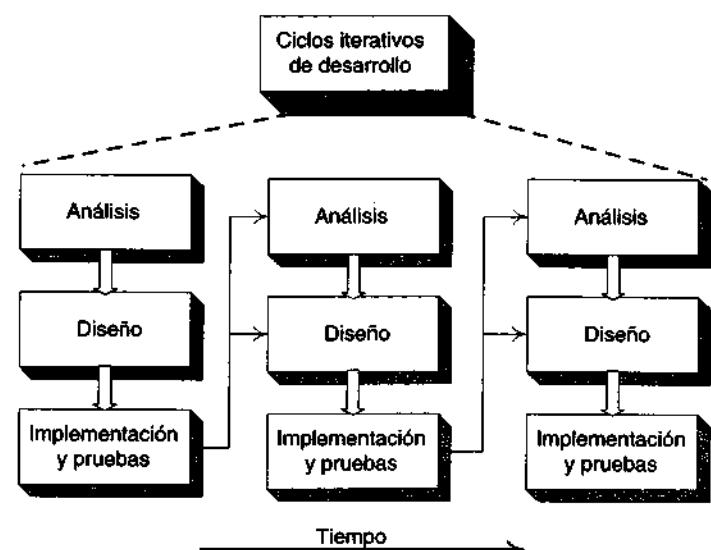


Figura 23.1 La implementación en un ciclo de desarrollo influye en el diseño ulterior.

Por ello, una actividad inicial de un ciclo de desarrollo consiste en sincronizar los artefactos; los diagramas del ciclo N no coincidirán con el código final de este ciclo, de modo que han de ser sincronizados antes de extenderlos mediante otros resultados del análisis y del diseño.

23.2.3 Cambios de código, las herramientas CASE y la ingeniería inversa

Es conveniente que los diagramas generados en la fase de diseño sean actualizados semi-automáticamente para que incluyan los cambios en la siguiente fase de codificación. En teoría, esto debería hacerse con una herramienta de CASE (*Computer-aided software engineering*, ingeniería de software asistida por computadora), la cual puede leer el código fuente (Java, entre otros) y producir automáticamente —por ejemplo— los diagramas de clases y de colaboración. Éste es un aspecto de la **ingeniería inversa**, o sea la actividad consistente en generar modelos lógicos partiendo de un código fuente ejecutable.

23.3 Mapeo de diseños para codificación

Para implementar un lenguaje de programación orientado a objetos se requiere escribir un código fuente para:

- definiciones de clase
 - definiciones de métodos

En las siguientes secciones explicaremos su creación en Java (un caso muy representativo); en el capítulo 24 veremos casos concretos de una implementación completa.

23.4 Creación de las definiciones de clase a partir de los diagramas de clases del diseño

Los diagramas de clases del diseño describen por lo menos el nombre de las clases, las superclases, las etiquetas de los métodos y los atributos simples de una clase. Esta información es suficiente para formular una definición básica de clase en un lenguaje orientado a objetos. Más adelante hablaremos de la incorporación de información sobre la interfaz y sobre el espacio destinado al nombre, entre otros detalles.

23.4.1 Definición de una clase con métodos y con atributos simples

Como se aprecia en la figura 23.2, un diagrama de clases del diseño facilita mapear las definiciones básicas de los atributos (simples variables de instancias en Java) y las etiquetas de métodos en la definición de *VentasLineadeProducto* de Java.

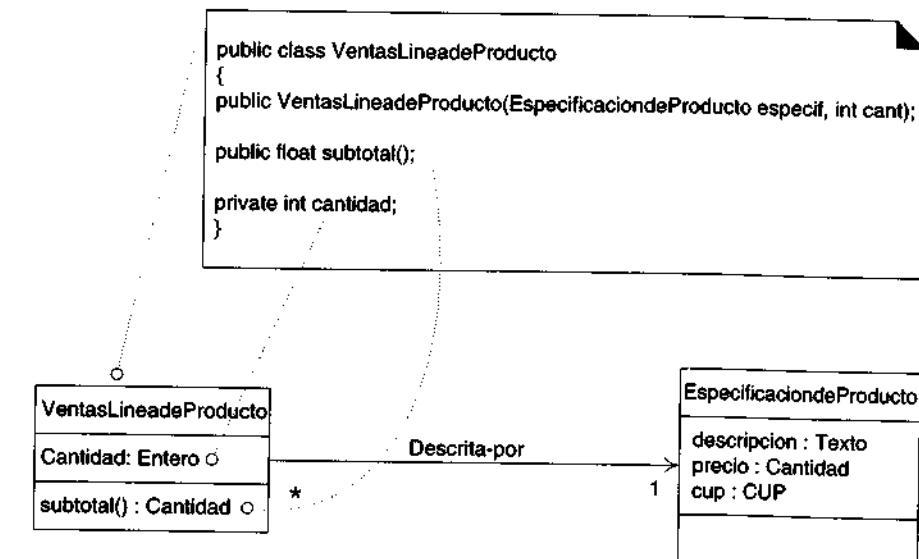


Figura 23.2 VentasLineadeProducto en Java.

Nótese que el constructor de Java `VentasLineadeProducto(...)` ha sido agregado. Deriva del hecho de que se le envía un mensaje `crear(especif, cant)` a `VentasLineadeProducto`.

en el diagrama de colaboración *introducirProducto*. En este lenguaje ello indica que se requiere un constructor que soporte los parámetros. El método *crear* a menudo se excluye en el diagrama de clase por ser muy común y por sus múltiples interpretaciones, según el lenguaje objetivo.

Observe también que transformamos de *Cantidad* a simple punto *flotante* el tipo de retorno del método *subtotal*. Suponga que, en el trabajo inicial de codificación, el diseñador no quiere dedicarle tiempo a la clase *Cantidad* y que pospondrá esta actividad.

23.4.2 Adición de los atributos de referencia

El atributo de referencia es aquel que remite a otro objeto complejo, no a un tipo primitivo como String, Number, etcétera.

Los atributos de referencia de una clase están indicados por las asociaciones y la navegabilidad en un diagrama de clases.

Por ejemplo, una clase *VentasLineadeProducto* tiene una asociación a *EspecificaciondeProducto*, con navegabilidad respecto a esta última. Se acostumbra interpretarla como atributo de referencia en la clase *VentasLineadeProducto* que remite a la instancia *EspecificaciondeProducto* (figura 23.3).

En Java, lo anterior significa que denota una variable de instancia que se refiere a la instancia *EspecificaciondeProducto*.

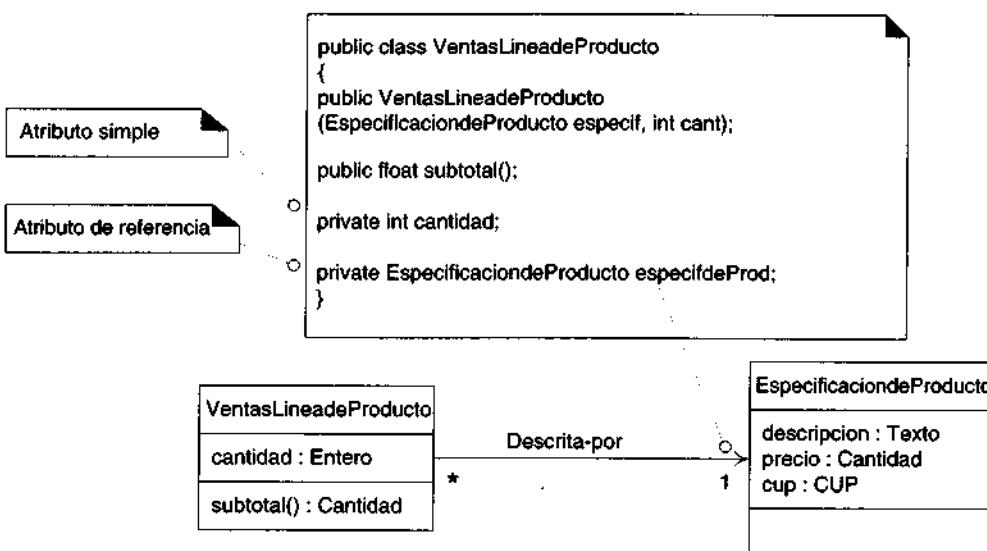


Figura 23.3 Agregación de atributos de referencia.

Nótese que a menudo los atributos de referencia de una clase están implícitos, no explícitos, en un diagrama de clases orientado al diseño.

Por ejemplo, aunque hemos incorporado una variable de instancia a la definición de *VentasLineadeProducto* en Java para apuntar a *EspecificaciondeProducto*, no se declara explícitamente como atributo en la sección dedicada a ellos dentro de la casilla que representa la clave. Hay una visibilidad *sugerida* de atributos —indicada por la asociación y la navegabilidad— que se define explícitamente como un atributo durante la fase de generación del código.

23.4.3 Atributos de referencia y nombres de los papeles

La siguiente iteración analizará el concepto de los nombres de papeles en los diagramas de estructura estática. A cada fin o extremo de una asociación se le da el nombre de papel. En pocas palabras, el **nombre de papel** es aquel que identifica el papel y muchas veces contiene algún contexto semántico respecto a la naturaleza del papel.

Si el nombre de un papel se encuentra en un diagrama de clase, utilícelo como criterio para elegir el nombre del atributo de referencia durante la generación del código, según se señala en la figura 23.4.

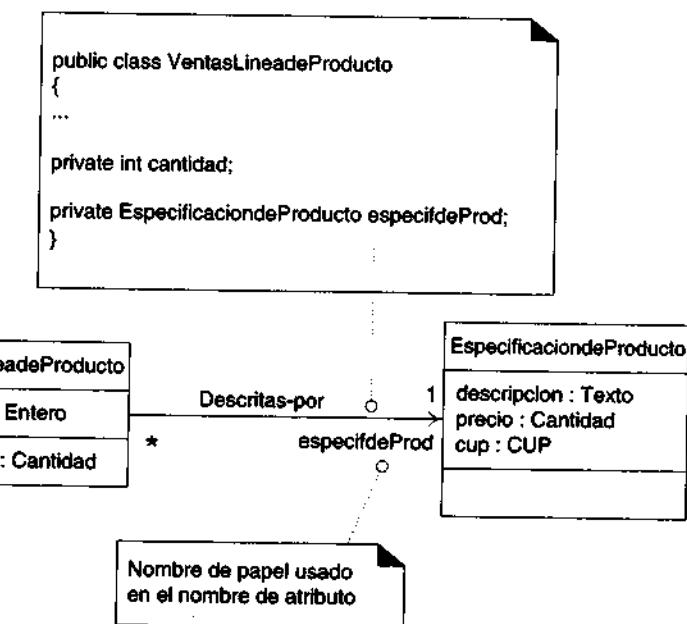


Figura 23.4 Con los nombres de papeles pueden generarse nombres de variables de instancias.

23.5 Creación de métodos a partir de los diagramas de colaboración

Un diagrama de colaboración muestra los mensajes que se envían en respuesta a una llamada al método. La secuencia de los mensajes se traduce a una serie de enunciados en la definición del método.

El diagrama de colaboración *introducirProducto* de la figura 23.5 nos servirá para dar un ejemplo concreto de la definición del método *introducirProducto* en Java.

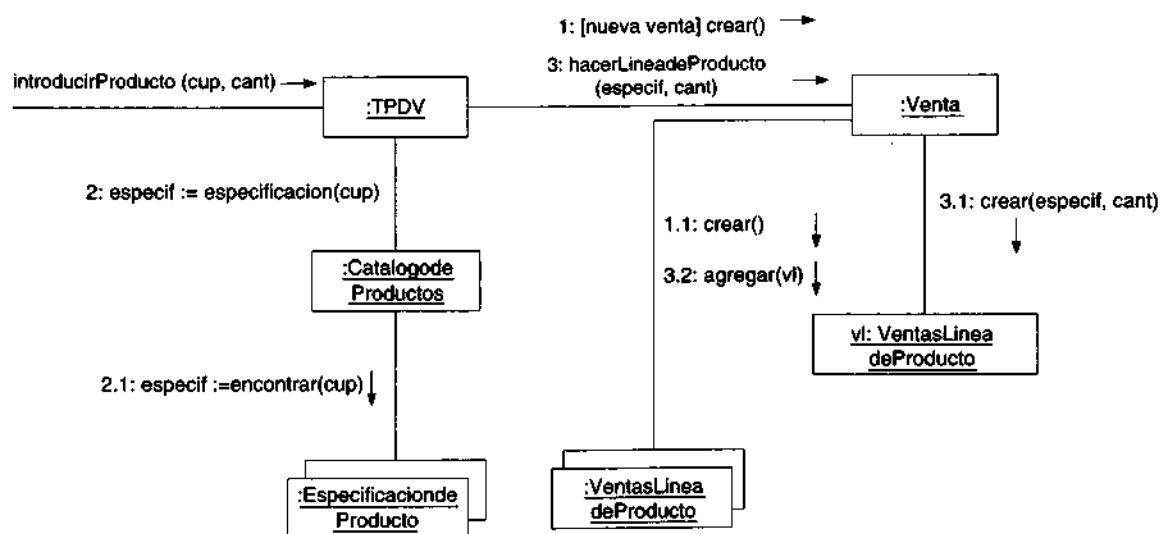


Figura 23.5 El diagrama de colaboración *introducirProducto*.

En este ejemplo utilizaremos la clase *TPDV*. En la figura 23.6 se incluye una definición de Java.

23.5.1 El método *TPDV--introducirProducto*

El mensaje *introducirProducto* se envía a una instancia *TPDV* y, por tanto, el método *introducirProducto* se define en la clase *TPDV*.

```
public void introducirProducto(int cup, int cant)
```

Mensaje 1: conforme al diagrama de colaboración, en respuesta al mensaje *introducirProducto* el primer enunciado tiene por objeto crear condicionalmente una nueva *Venta*.¹

```
if ( esNuevaVenta() ) { Venta = new Venta(); }
```

¹ Por razones de brevedad, se prescinde de los métodos de acceso aunque generalmente se recomienda utilizarlos.

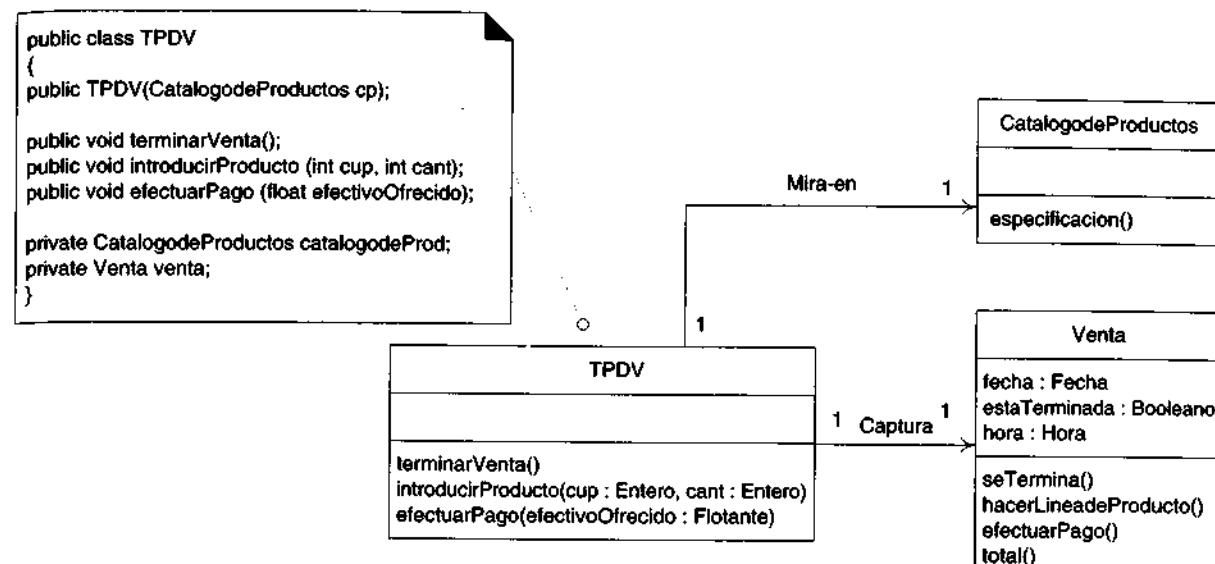


Figura 23.6 La clase *TPDV*.

Mensaje 2: Segundo, enviamos un mensaje *especificación* a *CatalogodeProductos* para recuperar una instancia *EspecificaciondeProducto*.

```
EspecificaciondeProducto especif =
catalogodeProd.especificacion(cup);
```

Mensaje 3: Tercero, enviamos a la *Venta* el mensaje *hacerLineadeProducto*.

```
venta.hacerLineadeProducto (especif, cant);
```

En resumen, en el método de Java los mensajes en secuencia dentro de un método se mapean en una proposición como se indica en el diagrama de colaboración.

El método íntegro *introducirProducto* y su relación con el diagrama de colaboración se muestran en la figura 23.7.

23.5.2 El método *TPDV--esNuevaVenta*

La definición del método *TPDV--introducirProducto* dio origen a un nuevo método en la clase *TPDV*: *esNuevaVenta*. Éste es un ejemplo pequeño de cómo, durante la fase de codificación, aparecen cambios en el diseño. Posiblemente este método hubiera sido descubierto durante la fase inicial de la solución, pero lo importante es que inevitablemente se producirán cambios mientras se efectúa la programación.

En un primer intento, este método de ayuda aplicará una prueba basándose en que la variable de la instancia *ventas* sea o no *nula*.

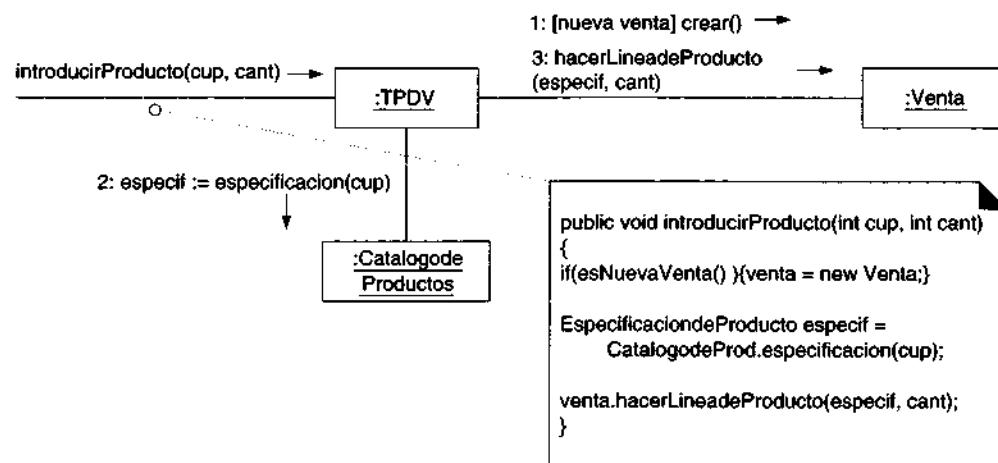


Figura 23.7 El método introducirProducto.

Por tanto,

```

private boolean esNuevaVenta()
{
    return (venta == null);
}
    
```

¿Por qué no simplemente codificamos esta prueba y la incorporamos al método *introducirProducto*? No lo hacemos porque se basa en una decisión de diseño sobre la representación de la información. En términos generales, lo más conveniente es incluir en métodos de ayudante las expresiones que dependen de este tipo de decisiones; se hace eso para reducir al mínimo el cambio en caso de que se modifique la representación. Además, para un lector del código la prueba *esNuevaVenta* resulta más informativa en lo tocante al propósito semántico. Considere la relativa claridad semántica de las siguientes expresiones alternas:

```

if ( esNuevaVenta() )...
    // a diferencia de

if (venta == null)...
    
```

El lector al reflexionar se dará cuenta de que esta prueba no es adecuada en el caso general. Por ejemplo, ¿qué sucedería si se terminó una venta y está a punto de empezar otra? De ser así, el atributo *venta* no será *nulo*; apuntará a la última venta. En consecuencia, se necesita alguna prueba adicional para determinar si se trata de una nueva venta. Para resolver este problema supongamos que, si la venta actual está en el estado *terminada*, podrá empezar otra venta. Puede efectuarse fácilmente un cambio, si en una iteración posterior esta consideración resulta ser una regla inapropiada de negocios.

Por tanto,

```

private boolean esNuevaVenta()
{
    return ( venta == null ) || sale.isComplete();
}
    
```

Este cambio extemporáneo del significado de “una nueva venta” nos permite apreciar una vez más la ventaja de encapsular su prueba dentro de un método de ayudante: el método no requiere más que un solo cambio.

23.6 Actualizaciones de las definiciones de clases

A partir de estas decisiones de codificaciones, habrá que agregar un nuevo método a la definición de la clase *TPDV*: *esNuevaVenta*. Deberíamos actualizar el diagrama de clases del diseño que describe la clase *TPDV* para que refleje este cambio de código. Si estamos utilizando una herramienta de CASE con las capacidades de la ingeniería inversa, tal vez no sea difícil regenerar los diagramas basándonos para ello en el estado del código. En caso contrario se requerirá el mantenimiento manual. Aplicar esta técnica a los diagramas es de gran utilidad; pero como rara vez se lleva a cabo de manera adecuada y resulta muy costoso, se justifica entonces invertir en una herramienta de CASE capaz de aplicar esta tecnología para generar los diagramas de UML.

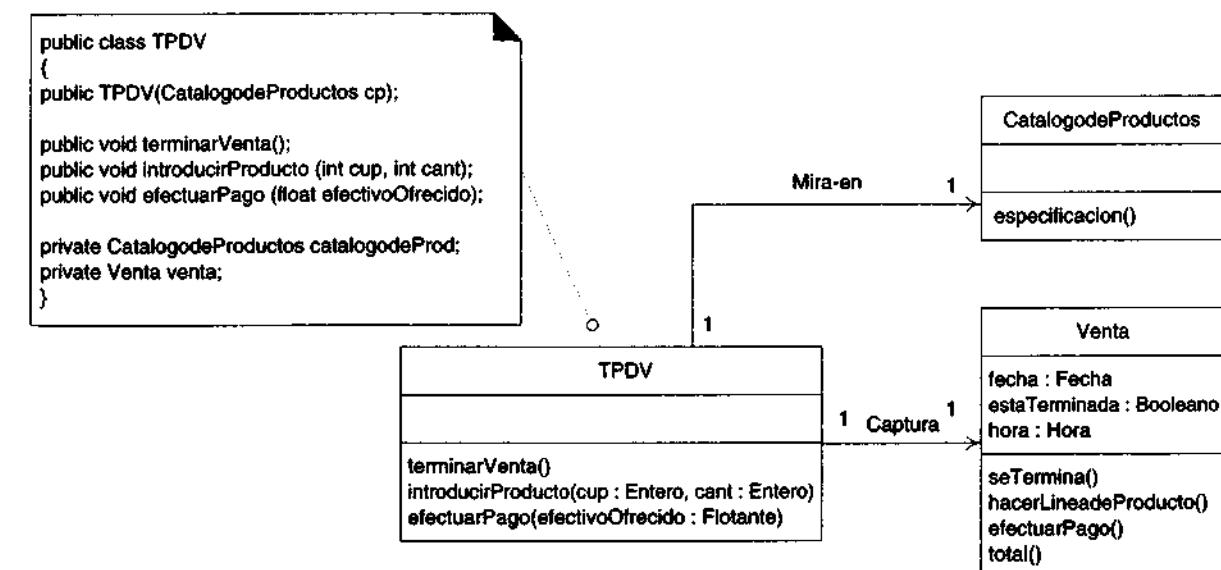


Figura 23.8 Definición actualizada de la clase TPDV.

23.7 Las clases de contenedor/colección en código

A menudo un objeto debe conservar la visibilidad de un grupo de otros objetos; la necesidad de ello se pone en evidencia generalmente con el valor de multiplicidad que presenta un diagrama de clase: puede ser mayor que uno. Por ejemplo, una *Venta* ha de conservar la visibilidad de un grupo de instancias *VentasLineadeProducto*, como se observa en la figura 23.9.

En los lenguajes de programación orientados a objetos estas relaciones a menudo se implementan introduciendo un contenedor o colección intermedios. La clase del lado "uno" define un atributo de referencia que apunta a una instancia contenedor/colección, el cual contiene instancias de la clase del lado "muchos".

Por ejemplo, el JDK de Java incluye clases de contenedor como (*Hashtable*) y *Vector*. Por medio de *Vector* la clase *Venta* puede definir un atributo que conserva una lista ordenada de las instancias *VentasLineadeProducto*.

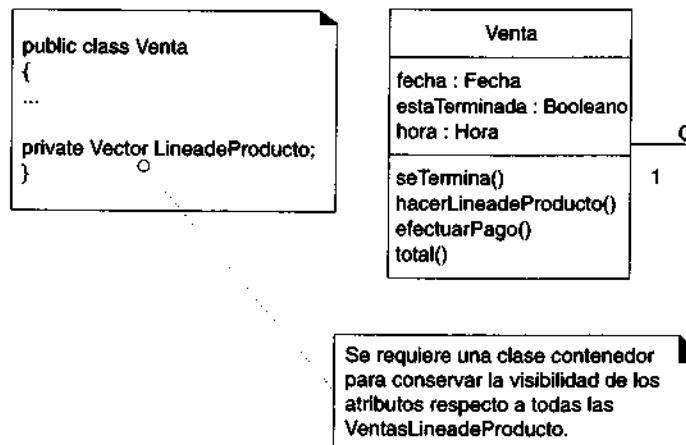


Figura 23.9 Agregación de un contenedor.

Los requerimientos influyen al elegir una clase de contenedor; la consulta basada en claves exige emplear una *Hashtable*, una lista ordenada creciente requiere un *Vector* y así sucesivamente.

23.8 Manejo de las excepciones y de los errores

Hasta ahora no hemos hablado del manejo de errores en la obtención de una solución. Lo hicimos intencionalmente con el propósito de concentrarnos en las cuestiones fundamentales de asignar responsabilidades y en el diseño orientado a objetos. Pero en una aplicación concreta conviene ocuparse del manejo de errores en la fase de diseño.

Por ejemplo, podemos comentar los contratos anexándoles una breve explicación de las situaciones ordinarias de errores y el plan general de solución.

El lenguaje UML no cuenta con una notación especial para indicar las excepciones. Más bien, se indican con la notación de los diagramas de colaboración basados en los mensajes. Un diagrama de colaboración puede comenzar con un mensaje que representa una excepción ocurrida.

23.9 Definición del método Venta-->hacerLineadeProducto

Un último ejemplo: el método *hacerLineadeProducto* también puede escribirse observando atentamente el diagrama de colaboración *introducirProducto*. En la figura 23.10 se incluye una versión abreviada de este diagrama, con el método correspondiente de Java.

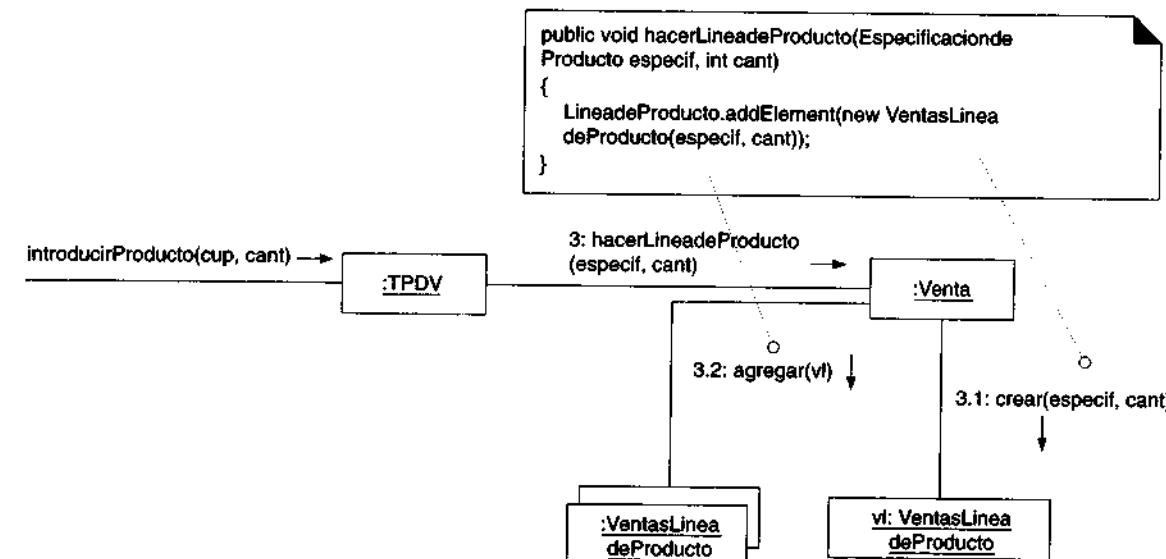


Figura 23.10 Método Venta-->hacerLineadeProducto.

Este método presenta otro ejemplo de modificación del código respecto al diagrama de colaboración: el mensaje genérico *agregar* ha sido traducido al mensaje específico de Java *addElement*. En este tipo de diagramas también podemos emplear los mensajes propios del lenguaje o de la biblioteca, como *addElement*.

23.10 Orden de la implementación

Es necesario implementar las clases (y, en teoría, probarlas totalmente de manera individual), de la menos acoplada a la más acoplada (figura 23.11). Por ejemplo, las posi-

bles primeras clases para implementar son *Pago* o *EspecificacioneProducto*. Vienen después las clases que dependen únicamente de las implementaciones anteriores: *CatalogodeProductos* o *VentasLineadeProducto*.

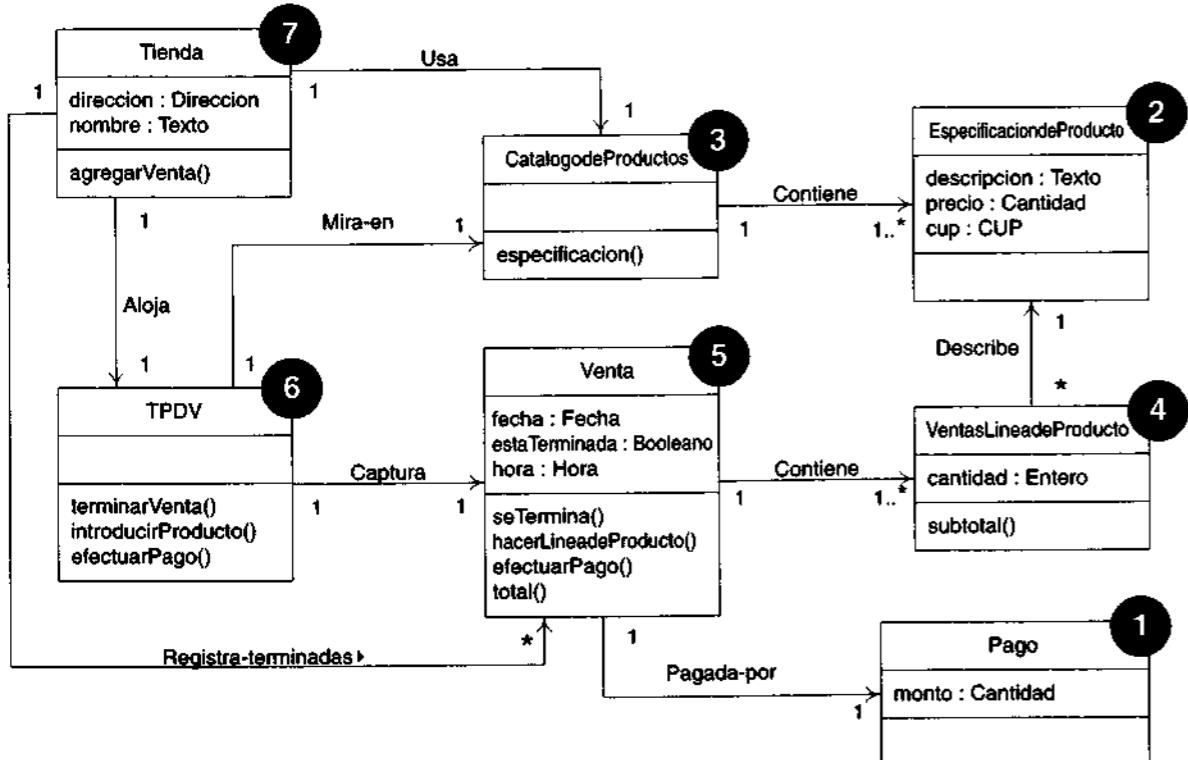


Figura 23.11 Posible orden de implementación de clases.

23.11 Resumen del mapeo del diseño a la codificación

Es muy sencillo el proceso de traducir a definiciones de clases los diagramas del diseño y también el proceso de traducir a métodos los diagramas de colaboración. En la fase de programación todavía pueden tomarse muchas decisiones, efectuarse muchos cambios de diseño y realizarse una exploración muy extensa; pero, en teoría, la arquitectura global y las decisiones más importantes ya se establecieron totalmente antes que comience la fase de codificación.

SOLUCIÓN EN PROGRAMA DE JAVA

24.1 Introducción a la solución convertida en programa

En este capítulo incluimos ejemplos de la solución convertida en programa de la capa de objetos del dominio para el primer ciclo de desarrollo de la aplicación punto de venta. La generación de código proviene principalmente de los diagramas de clases del diseño y de los diagramas de colaboración que se definieron en la fase de diseño; la generación de código se basa en los principios del mapeo de diseño a codificación que expusimos en el capítulo anterior.

El punto principal de este listado es el siguiente: la traducción relativamente directa de los artefactos del diseño a la creación de código. El código define un caso sencillo de prueba: no pretende exemplificar un programa robusto y totalmente desarrollado de Java con sincronización, manejo de excepciones u otras características.

Clase Pago

```

package tpdv;

public class Pago {
    private float monto;
    public Pago( float efectivoOfrecido ) {
        this.monto = efectivoOfrecido;
    }
    public float obtenerMonto() { return monto; }
}
  
```

Clase CatalogodeProductos

```

package tpdv;
  
```

```

import java.util.*;
public class CatalogodeProductos
{
private Hashtable EspecificacionesdeProducto
= new Hashtable();
public CatalogodeProductos()
{
    EspecificacionesdeProducto ep =
        New EspecificaciondeProducto( 100, 1,"producto 1" );
    EspecificacionesdeProducto.put(new Integer( 100 ), ep );
    Ep = new EspecificaciondeProducto(200, 1, "producto 2");
    EspecificacionesdeProducto.put(new Integer( 200 ), ep );
}

public EspecificaciondeProducto obtenerEspecificacion
( int cup )
{
    return (EspecificaciondeProducto)
        EspecificacionesdeProducto.get( new Integer ( cup ) );
}
}

```

Clase TPDV

```

package tpdv;
import java.util.*;
class TPDV
{
private CatalogodeProdutos catalogodeProductos;
private Venta venta;

public TPDV( CatalogodeProductos catalogo )
{
    catalogodeProductos = catalogo;
}

public void terminarVenta()
{
    venta.seTermina();
}

public void introducirProducto( int cup, int cantidad )
{
    if( esNuevaVenta() )
    {
        venta = new Venta();
    }
    EspecificaciondeProducto especif =

```

```

        catalogodeProductos.especificacion( cup );
        venta.hacerLineadeProducto( especif, cantidad );
    }

    public void efectuarPago( float efectivoOfrecido )
    {
        venta.efectuarPago( efectivoOfrecido );
    }

    private boolean esNuevaVenta()
    {
        return ( venta == null ) ||( venta.estaTerminada() );
    }
}

```

Clase EspecificaciondeProducto

```

Package tpdv;

public class EspecificaciondeProducto
{
private int      cup          = 0;
private float    precio       = 0;
private String   descripcion = "";

public EspecificaciondeProducto
( int cup, float precio, String descripcion)
{
    this.cup      = cup;
    this.precio   = precio;
    this.descripcion = descripcion;
}

public int obtenerCUP() { return cup; }

public float obtenerPrecio() { return precio; }

public String obtenerDescripcion() { return descripcion; }
}

```

Clase Venta

```

package tpdv;
import java.util.*;

class Venta
{
private Vector LineadeProducto = new Vector();
private Date fecha = new Date();
private boolean estaTerminada = false;
}

```

```

private Pago pago;

public float obtenerSaldo()
{
    return pago.obtenerMonto() - total();
}

public void seTermina() { estaTerminada = true; }

public boolean estaTerminada() { return estaTerminada; }

public void hacerLineadeProducto
(EspecificaciondeProducto especif, int cantidad)
{
    LineadeProducto.addElement(
        new VentaLineadeProducto( especif, cantidad ) );
}

public float total()
{
    float total = 0;
    enumeration e = LineadeProducto.elements();
    while( e.hasMoreElements() )
    {
        total += ( (ventasLineadeProducto)
            e.nextElement() ).subtotal();
    }
    return total;
}

public void efectuarPago( float efectivoOfrecido )
{
    pago = new Pago( efectivoOfrecido );
}

```

Clase VentasLineadeProducto

```

package tpdv;

class VentasLineadeProducto
{
private int cantidad;
private EspecificaciondeProducto especifdeProd;

public VentasLineadeProducto
(EspecificaciondeProducto especif, int cantidad)
{
    this.especifdeProd = especif;
    this.cantidad = cantidad;
}

```

```

public float subtotal()
{
    return cantidad * especifdeProd.obtenerPrecio();
}
}

```

Clase Tienda

```

package tpdv;

class Tienda
{
private CatalogodeProductos catalogodeProductos
    = new CatalogodeProductos();
private TPDV tpdv = new TPDV( CatalogodeProductos );

public TPDV obtenerTPDV() { return tpdv; }
}

```

**PARTE VI FASE
DE ANÁLISIS (2)**

ELECCIÓN DE LOS REQUERIMIENTOS DEL CICLO DE DESARROLLO 2

25.1 Requerimientos del ciclo de desarrollo 2

En el segundo ciclo de desarrollo de la aplicación del punto de venta revisamos los casos de uso *Comprar Productos* e *Iniciar*, pero les agregaremos una mayor funcionalidad para que se aproximen al caso de uso final y completo. Se acostumbra utilizar muchas veces un mismo caso de uso en varios ciclos de desarrollo y ampliarlo después para alcanzar la funcionalidad que se requiere.

El primer ciclo llevó a cabo muchas simplificaciones y por eso el problema no resultó excesivamente complejo. Una vez más —y por la misma razón—, consideraremos un grado relativamente pequeño de funcionalidad complementaria.

La funcionalidad complementaria consiste en que se dará soporte a los pagos en efectivo, con tarjeta de crédito y con cheque.

En un proyecto de desarrollo esto no será una decisión unánime de mayor funcionalidad en el segundo ciclo: otra posibilidad es actualizar de manera automática el inventario o un caso de uso totalmente diferente. Pero esta opción ofrece multitud de oportunidades valiosas de aprendizaje.

25.2 Suposiciones y simplificaciones

Las siguientes suposiciones y simplificaciones se adoptan en los dos casos de uso que hemos venido examinando.

Comprar Productos

- No se actualiza el inventario.
- Es una tienda independiente, no parte de una organización más grande.
- Captura manual de los códigos universales de producto (CUP); sin lector de código de barras.
- Sin cálculos de impuestos.
- Sin cupones.
- Sin políticas especiales de precios.
- El cajero no tiene que darse de alta en una cuenta.
- No se llevan registros de cada cliente ni de sus hábitos de compra.
- No se lleva un control de la gaveta de la caja.
- En el recibo aparecen el nombre y la dirección de la tienda, la fecha y la hora de la venta.
- En el recibo no aparecen la identificación del cajero ni la de TPDV.
- Las ventas terminadas se anotan en el registro histórico o bitácora.
- Sólo un pago, de un tipo, se utiliza en una venta.
- Los pagos se realizan en su totalidad; no se aceptan pagos parciales ni pagos en abonos.

Se autorizan los pagos con cheque y con tarjeta de crédito.

Se utiliza un servicio diferente de autorización de crédito con cada tipo de crédito (Visa, MasterCard u otras tarjetas de crédito).

El mismo servicio de autorización de crédito se emplea con todos los cheques.

La terminal punto de venta se encarga de comunicarse con el servicio de autorización de crédito; el lector de las tarjetas de crédito es un dispositivo tonto que se limita a enviar a la terminal la información de la tarjeta.

La comunicación con un servicio externo se lleva a cabo a través de un módem. Debe marcarse un número telefónico cada vez que se emplee.

Generalmente un banco da los servicios de autorización de crédito.

Iniciar

- Los pagos con cheque y con tarjeta de crédito deben cubrir el monto exacto de la venta.
- Suponga que la fecha y la hora son las correctas.
- Se requiere un mínimo de inicialización para brindar soporte al caso de uso *Comprar Productos*.
- No se almacena información de inicialización en la base de datos.

CÓMO RELACIONAR CASOS MÚLTIPLES DE USO

Objetivos

- Relacionar los casos de uso con las asociaciones de aplicaciones y de extensión.

26.1 Introducción

Podemos expresar los procesos relacionados con diversas formas de pago en la aplicación del punto de venta como casos independientes y relacionarlos con otros casos.

Su finalidad es preparar un diagrama actualizado que muestre estos casos de uso adicionales y sus relaciones.

El lenguaje UML tiene una notación especial para indicar las relaciones de los casos de uso; en este capítulo vamos a estudiarla y a describir un nuevo diagrama para el sistema.

26.2 Cuándo crear casos de uso independientes

En la explicación anterior sobre los casos de uso, los procesos de pago se anotaron en las subsecciones del caso *Comprar Productos*. También pudimos haberlos dividido en casos aislados. ¿Cuándo debería un paso importante o una actividad ramificada de un caso de uso escribirse como una parte de una subsección y no como un caso de uso aparte?

Escriba los pasos principales o las actividades ramificadas de un caso de uso como caso independiente cuando:

- Se duplican en otros casos.
- Son complejos y largos, y su separación facilita colocarlos en unidades manejables y comprensibles.

En la aplicación del punto de venta, los métodos de pago son las actividades ramificadas que pueden estar presentes también en otros casos como *Intercambiar Productos* o *Abonar a un Plan Apartado*. Por tanto, se justifica tratarlos como casos independientes y asignarles los siguientes nombres:

- *Pago en efectivo*, *Pago con tarjeta de crédito*, *Pago con cheque*.

26.3 Diagramas de casos de uso con las relaciones *usa*

Si un caso de uso inicia o contiene el comportamiento de otro, se dice que *usa* el segundo caso y que ambos se encuentran en una relación *usa*.

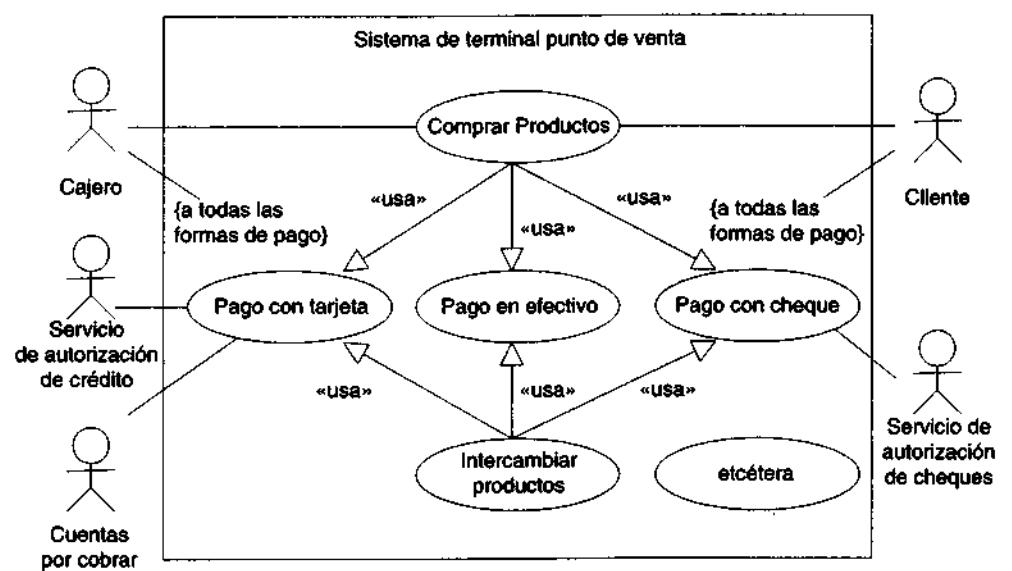


Figura 26.1 Conexión de los casos de uso por medio de la relación *usa*.

26.3.1 Ejemplo

Supongamos que los diferentes procesos de pago en el proceso *Comprar Productos* se describen en los casos de uso separados *Pago en efectivo*, *Pago con tarjeta de crédito* y *Pago con cheque*. El caso de uso *Comprar Productos* dará inicio a uno de ellos y, por lo mismo, estará en una relación *usa* con ellos.

En el lenguaje UML, la relación *usa* se indica (como se advierte en la figura 26.1) con una línea de generalización (línea continua con una flecha hueca) acompañada de la etiqueta «*usa*».

26.4 Documentos de casos de uso con las relaciones *usa*

Cuando los casos están en una relación *usa*, sus documentos han de expresar con palabras esa asociación. El caso que utiliza el comportamiento de otro debería indicar la conexión por medio de la palabra clave *iniciar* en el texto del caso de uso, del modo siguiente:

Caso de uso: Comprar Productos

...

El cliente escoge el método de pago:

- a. Si el pago se realiza en efectivo, iniciar *Pago en efectivo*.
- b. Si el pago se realiza con tarjeta de crédito, iniciar *Pago con tarjeta de crédito*.
- c. Si el pago se efectúa con cheque, iniciar *Pago con cheque*.

...

26.4.1 Ejemplo

Si los diferentes procesos de pago del proceso *Comprar Productos* se describen en los casos de uso aislados *Pago en efectivo*, *Pago con tarjeta de crédito* y *Pago con cheque*, el caso *Comprar Productos* iniciará uno de ellos y, por tanto, estará en una relación *usa* con ellos.

En este enfoque, los métodos de pago ya no se expresan en subsecciones del caso de uso *Comprar Productos*; son casos independientes. Nótese que podemos agregar una sección más de *Casos de uso relacionados* que resuma los casos afines, los cuales serán útiles si el caso es extenso y si no se capta fácilmente cuáles otros intervienen.

26.4.2 Comprar Productos

Caso de uso:	Comprar productos
Actores:	Cliente (iniciador), Cajero.
Resumen:	Un cliente llega a la caja registradora con productos que desea comprar. El Cajero registra los productos y recibe el pago. Al terminar la transacción, el Cliente se marcha con los productos adquiridos.

Curso normal de los eventos

Acción de los actores	Respuesta del sistema
1. Este caso comienza cuando un Cliente llega a la caja registradora TPDV con productos que desea comprar.	
2. El Cajero registra los productos. Si hay más de un producto de la misma línea, también captura la cantidad.	3. Determina el precio del producto y agrega la información correspondiente a la transacción actual de ventas. Despliega la descripción y el precio del producto en cuestión.
4. Al terminar de capturar el producto, el Cajero indica a TPDV que se concluyó la venta.	5. Calcula y muestra el total de la venta.
6. El Cajero le indica el total al Cliente.	
7. El Cliente escoge la forma de pago:	
a. Si paga en efectivo, iniciar <i>Pago en efectivo</i> .	
b. Si paga con tarjeta de crédito, iniciar <i>Pago con tarjeta de crédito</i> .	
c. Si paga con cheque, iniciar <i>Pago con cheque</i> .	
8. Registra la venta terminada.	
9. Imprime un recibo.	

Curso normal de los eventos

Acción de los actores	Respuesta del sistema
10. El Cajero entrega el recibo al Cliente.	
11. El Cliente se marcha con los productos comprados.	

Cursos alternos

- Sección 2: invalidar el identificador capturado del producto. Desplegar el error.
- Sección 7: el Cliente no pudo pagar. Cancelar la transacción de venta.

Casos de uso relacionados

- usa *Pago en efectivo*
- usa *Pago con tarjeta de crédito*
- usa *Pago con cheque*

26.4.3 Pago en efectivo

Caso de uso:	Pago en efectivo
Actores:	Cliente (iniciador), Cajero.
Resumen:	Un Cliente paga en efectivo una compra en una terminal situada en el punto de venta.

Curso normal de los eventos

Acción de los actores	Respuesta del sistema
1. Este caso de uso comienza cuando un Cliente decide pagar en efectivo, una vez que le comunican el total de la venta.	
2. El Cliente entrega un pago en efectivo —el “efectivo ofrecido”—, posiblemente mayor que el total de la venta.	

Curso normal de los eventos

Acción de los actores	Respuesta del sistema
3. El Cajero registra el efectivo ofrecido.	4. Muestra el cambio o vuelto que se le regresará al Cliente.
5. El Cajero deposita el efectivo recibido y extrae el dinero que devolverá. El Cajero entrega al Cliente el vuelto de la transacción.	

Cursos alternos

- Sección 2: el cliente no tiene suficiente efectivo. Puede cancelarse la venta o iniciarse otra forma de pago.
- Sección 3: la gaveta de la caja registradora no contiene bastante efectivo para devolver al Cliente su cambio. El Cajero pide más efectivo al supervisor o propone al Cliente otra forma de pago.

26.4.4 Pago con tarjeta de crédito

Caso de uso:	Pago con tarjeta de crédito, forma esencial.
Actores:	Cliente (iniciador), Cajero, Servicio de autorización de crédito, Cuentas por cobrar.
Resumen:	Un Cliente paga una compra con tarjeta de crédito en una terminal situada en el punto de venta. El pago es validado por un servicio externo de autorización y se registra en un sistema de cuentas por cobrar.

Curso normal de los eventos

Acción de los actores	Respuesta del sistema
1. Este caso de uso comienza cuando un Cliente decide pagar con tarjeta de crédito, una vez que le han comunicado el total de la venta.	
2. El Cliente da la información crediticia que se requiere en esta forma de pago.	3. Genera una solicitud de pago con tarjeta de crédito y la envía al Servicio externo de autorización de crédito.
4. El Servicio de autorización de crédito autoriza el pago.	5. Recibe una respuesta aprobatoria del pago por parte del Servicio de autorización de crédito (SAC).
	6. Registra el pago con tarjeta de crédito y la respuesta aprobatoria en el sistema de Cuentas por cobrar. (El servicio de autorización de crédito le debe dinero a la Tienda y, por tanto, cuentas por cobrar debe darle seguimiento.)
	7. Muestra el mensaje con que se concede la autorización.

Cursos alternos

- Sección 4: el Servicio de autorización de crédito rechaza la solicitud de crédito. Proponerle al Cliente otra forma de pago.

26.4.5 Pago con cheque

Caso de uso:	Pago con cheque, forma esencial.
Actores:	Cliente (iniciador), Cajero, Servicio de autorización de cheques.
Resumen:	Un Cliente paga con cheque una compra en una terminal situada en el punto de venta. El pago es validado por un servicio externo de autorización de cheques.

Curso normal de los eventos**Acción de los actores**

1. Este caso de uso comienza cuando un Cliente decide pagar con cheque, una vez que le han comunicado el total de la venta.
2. El Cliente extiende un cheque y presenta una identificación.
3. El Cajero registra la información relativa a la identificación y solicita la autorización del pago con cheque.
4. Genera una solicitud de pago con cheque y la envía al Servicio externo de autorización de cheques.
5. El Servicio de autorización de cheques autoriza el pago.
6. Recibe la respuesta aprobatoria del pago por parte del Servicio de autorización de cheques.
7. Muestra el mensaje de que se obtuvo la autorización.

Respuesta del sistema**Cursos alternos**

- Sección 5: solicitud de pago con cheque rechazada por el Servicio de autorización. Sugerirle al Cliente otra forma de pago.

EXTENSIÓN DEL MODELO CONCEPTUAL

Objetivos

- Identificar nuevos conceptos para incluirlos en el segundo ciclo de desarrollo.

27.1 Nuevos conceptos en el sistema del punto de venta

Igual que en el primer ciclo de desarrollo, podemos desarrollar poco a poco el modelo conceptual considerando los casos actuales de uso en el segundo ciclo. Primero repasamos la *lista de categorías de conceptos* y luego aplicamos la identificación de frases nominales o sustantivos a partir de los casos de uso (con la advertencia habitual de que este segundo método exige mucha sensatez y prudencia). Como vimos en capítulos anteriores, un procedimiento muy eficaz para crear un modelo conceptual robusto y rico consiste en estudiar el trabajo que otros autores han dedicado al tema, sobre todo los que se valen del estudio de casos, entre ellos [Fowler96]. En este libro no vamos a examinar la multitud de cuestiones tan sutiles concernientes a la construcción de modelos que analizan en sus obras.

Nuestras estrategias fundamentales no darán por resultado el modelo conceptual más útil. En este capítulo elaboraremos un modelo provisional, que nos sirva simplemente para descubrir los conceptos iniciales. En capítulos posteriores redondearemos los detalles a partir de otros conceptos, atributos y asociaciones con el propósito de desarrollar un mejor modelo.

27.1.1 *Lista de las categorías de los conceptos*

La siguiente tabla tan sólo contiene los conceptos nuevos en el problema del sistema del punto de venta.

Categoría	Ejemplos
objetos físicos o tangibles	TarjetadeCredito, Cheque
especificaciones, diseños o descripciones de cosas	
lugares	
transacciones	PagoenEfectivo, PagoconTarjetadeCredito, PagoconCheque
transacciones con líneas de producto	
papeles de las personas	
contenedores de otras cosas	
cosas dentro de un contenedor	
otros sistemas de cómputo o electromecánicos externos a nuestro sistema	ServiciodeAutorizaciondeCredito, ServiciodeAutorizaciondeCheques
conceptos sustantivos abstractos	
organizaciones	ServiciodeAutorizaciondeCredito, ServiciodeAutorizaciondeCheques
eventos	
reglas y políticas	
catálogos	
registros de finanzas, de trabajo, de contratos, de asuntos legales	CuentasporeCobrar
instrumentos y servicios financieros	
manuales, libros	

27.1.2 Identificación de las frases nominales en los casos de uso

La identificación de las frases nominales o nombres sustantivos, repetimos, no puede aplicarse mecánicamente para descubrir los conceptos pertinentes que incluiremos en

un modelo conceptual. Es necesario recurrir al sentido común y a las abstracciones idóneas, pues el lenguaje natural es ambiguo y los conceptos relevantes no siempre se expresan de un modo explícito o claro en el texto. No obstante, es un procedimiento práctico en la construcción de modelos conceptuales por su gran simplicidad.

En teoría, recomendamos emplear los casos *reales* de uso —y no los casos de uso esenciales— en la identificación de conceptos, porque contienen especificaciones más detalladas y concretas.

Pago en efectivo

No hay nuevos conceptos, pues los que se utilizan ya se explicaron en la iteración 1. Pese a ello, podemos especificar el nombre del pago como *PagoenEfectivo*.

Pago con tarjeta de crédito

1. Este caso de uso comienza cuando un Cliente decide pagar con tarjeta de crédito, una vez que le han comunicado el total de la venta.
2. El Cliente desliza su **tarjeta de crédito** en un lector de tarjetas para efectuar el **pago con tarjeta de crédito**.
3. Genera una **solicitud de pago con tarjeta de crédito** y la envía a un **Servicio externo de autorización de crédito**, a través de un módem conectado a la TPDV. Requiere que se marque al servicio, que se transmita un registro de solicitud y que se espere el registro de respuesta.
4. Recibe una **respuesta aprobatoria de crédito** por parte del Servicio de autorización de crédito (SAC). Se codifica la respuesta en un registro de respuesta y se recibe a través del módem.
5. Registra en el sistema **Cuentas por cobrar** el pago con tarjeta y la información de la respuesta aprobatoria. (El Servicio externo de autorización de crédito le debe dinero a la Tienda y por eso Cuentas por cobrar debe darle seguimiento.)
6. Muestra el mensaje de autorización del crédito.

Pago con cheque

1. Este caso de uso comienza cuando un Cliente decide pagar con cheque, una vez que le han comunicado el total de la venta.
2. El Cliente extiende un cheque y se lo entrega al Cajero junto con su licencia de conducir.
3. El Cajero anota en el cheque el número de la licencia de conducir, lo teclea y lo introduce en el campo texto del *Número LC* en la ventana; después oprime el botón *Autorización de Cheques* para solicitar la autorización del pago con cheque.
4. Genera una solicitud de pago con cheque y la envía a un Servicio externo de autorización de cheques a través de un módem conectado a la TPDV. Se requiere telefonear al servicio, transmitir un registro de solicitud, y esperar el registro de respuesta.
5. Recibe una respuesta aprobatoria del cheque por parte del Servicio de autorización de cheques. Se codifica la respuesta en un registro de respuesta y se recibe a través del módem.
6. Muestra el mensaje de autorización del pago.

27.1.3 Transacciones con el servicio de autorización

La identificación de frases nominales en los casos de uso revela conceptos como *SolicituddePagoconTarjeta* y *RespuestaAprobatoriadePagoconTarjeta*. Es posible ver estos conceptos como tipos de transacciones con los servicios externos; en términos generales, conviene identificarlas porque la actividad y los procesos suelen girar en torno a ellas.

Estas transacciones no necesariamente deben representar registros de computadora ni bits que se desplacen sobre una línea. Representan la abstracción de la transacción independientemente del medio con que se ejecute. Así, una solicitud de pago con tarjeta de crédito puede ser ejecutada por personas que se comunican con teléfono, por dos computadoras que se transmiten registros o mensajes y por otros medios.

27.1.4 El modelo conceptual TPDV: bosquejo 1

En la figura 27.1 se incluye el modelo conceptual preliminar que se basa en la identificación de conceptos mediante la lista de comprobación y la identificación de sustantivos. No es tan útil ni tan completo como lo hubiéramos deseado; por eso, en los capítulos subsecuentes vamos a examinar detenidamente otras listas concernientes a la construcción de modelos.

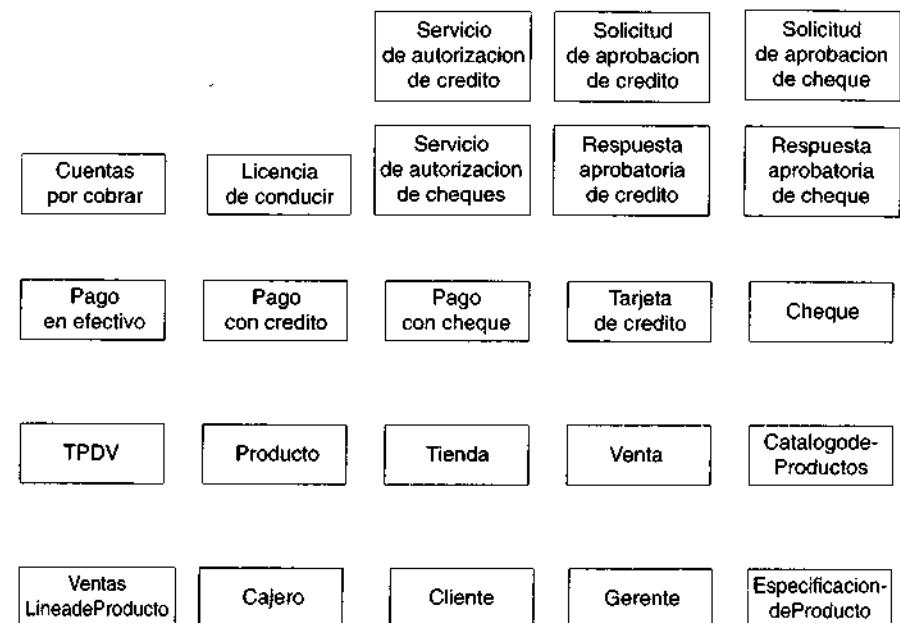


Figura 27.1 Modelo conceptual preliminar para el dominio de la terminal punto de venta.

GENERALIZACIÓN

Objetivos

- Crear jerarquías de generalización-especialización.
- Identificar cuándo vale la pena mostrar un subtipo.
- Aplicar las reglas de 100% y de Es-un para validar los subtipos.

28.1 Generalización

Los conceptos de *PagoenEfectivo*, *PagoconTarjeta* y *PagoconCheque* se parecen mucho. En este caso es posible (y útil)¹ organizarlos (figura 28.1) en una **jerarquía de tipo generalización-especialización** (o simplemente en una **jerarquía de tipo**) en la cual el **supertipo** *Pago* represente un concepto más general y los **subtipos**, conceptos más especializados.

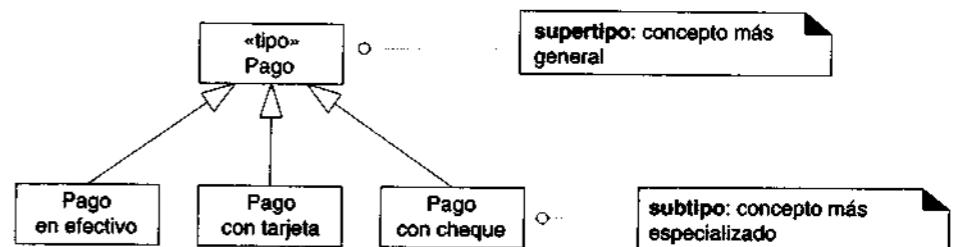


Figura 28.1 Jerarquía de generalizaciones-especificaciones.

¹ Más adelante, en este mismo capítulo, investigaremos las razones por las cuales se definen las jerarquías de tipos.

La **generalización** es la actividad consistente en identificar los aspectos comunes de los conceptos y en definir las relaciones entre el supertipo (concepto general) y el subtipo (concepto especializado). Es una forma de efectuar clasificaciones taxonómicas entre los conceptos que después se exemplifican en las jerarquías de tipos.

Identificar los supertipos y los subtipos es una labor útil en un modelo conceptual, porque su presencia nos permite entender los conceptos en términos más generales, más complejos y más abstractos. Favorece además la simplificación de la expresión, mejora la comprensión y reduce la información redundante. Y aunque por ahora nos concentraremos en un modelo conceptual y no en una implementación de software, obtenemos mejores programas de cómputo al implementar después los supertipos y los subtipos como clases.

Por tanto:

Identifique los supertipos y subtipos del dominio que se relacionan con la investigación en curso y describalos gráficamente en el modelo conceptual.

28.1.1 Notación de UML

En el lenguaje UML, la relación de generalización entre los elementos se indica con una punta de flecha grande y hueca que señala el elemento más general partiendo del más especializado. Puede emplearse un estilo de flechas blancas separadas o de una flecha compartida.

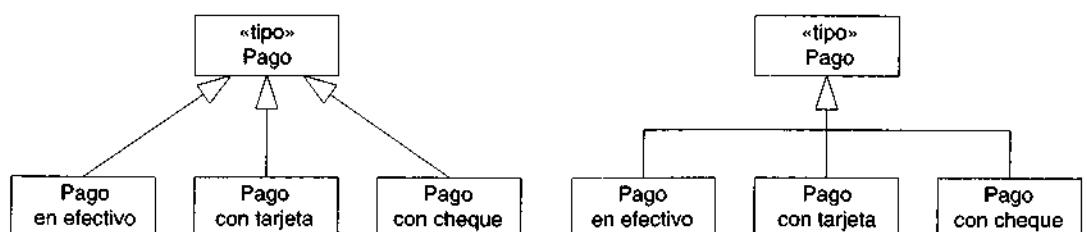


Figura 28.2 Jerarquía de tipos con notación de flechas separadas y con notación de una flecha compartida.

28.2 Definición de supertipos y de subtipos

Por ser de gran utilidad identificar los supertipos y los subtipos, conviene conocer con toda claridad y precisión la generalización, los supertipos y los subtipos a partir de la definición y de los conjuntos de tipos.¹ En la siguiente sección estudiamos los tipos.

¹ En otras palabras, la denotación y connotación de un tipo. Esta explicación se inspira en [MO95].

28.2.1 Generalización y la definición de tipos

¿Qué relación existe entre un supertipo y un subtipo?

La definición de un supertipo es más general o amplia que la de un subtipo.

Consideremos, por ejemplo, el supertipo *Pago* y sus subtipos (*PagoenEfectivo* y otros). Supongamos que, según su definición, *Pago* representa la transacción de transferir dinero (no necesariamente efectivo) de una entidad a otra en una compra y que en todos los pagos se refiere cierta cantidad de dinero. El modelo correspondiente a esta transacción aparece en la figura 28.3.

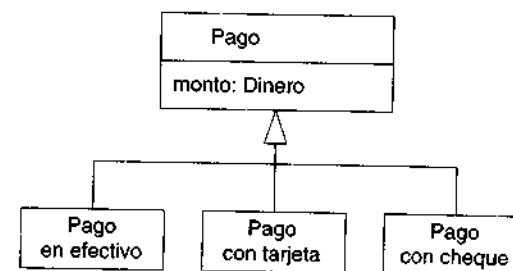


Figura 28.3 Jerarquía de los tipos de pago.

Un *PagoconTarjeta* es una transferencia de dinero que se realiza a través de una institución de crédito y que ha de ser autorizada. Mi definición de *Pago* abarca la de *PagoconTarjeta* y es más general que ella.

28.2.2 Generalización y los conjuntos de tipos

Los subtipos y los supertipos están relacionados por su pertenencia a un conjunto.

Todos los miembros de un conjunto de subtipo pertenecen al respectivo conjunto de supertipo.

Por ejemplo, en lo tocante a la pertenencia a un conjunto, todas las instancias del conjunto *PagoconTarjeta* también son miembros del conjunto de *Pago*. Esto se muestra en la figura 28.4 como un diagrama de Venn.

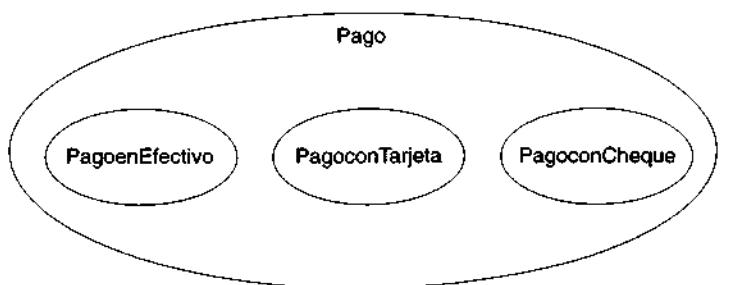


Figura 28.4 Diagrama de Venn para las relaciones de un conjunto.

28.2.3 Conformidad con la definición de subtipos

Cuando se crea una jerarquía de tipos, en torno a los supertipos se hacen afirmaciones que se aplican también a los subtipos. Así, la figura 28.5 señala que todos los *Pagos* tienen un *monto* y que se asocian a una *Venta*.

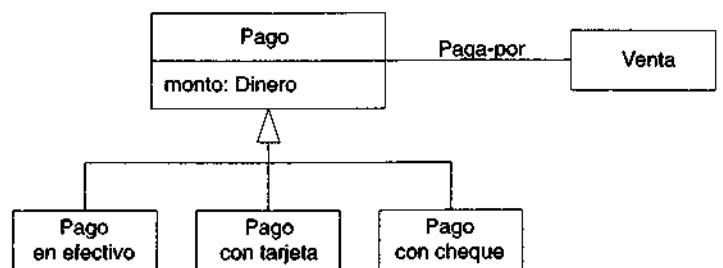


Figura 28.5 Conformidad con los subtipos.

Todos los subtipos de *Pago* han de conformarse y tener un monto y un pago de una *Venta*. En términos generales, a esta regla de conformidad con una definición de un supertipo se le da el nombre de *Regla de 100%*:

Regla del 100%

El 100% de la definición del supertipo debería aplicarse también al subtipo. Éste ha de conformarse con el supertipo al 100% de sus:

- atributos
- asociaciones

28.2.4 Conformidad con el conjunto de subtipos

Un subtipo debería ser miembro del conjunto del supertipo. Así, *PagoconTarjeta* debería pertenecer al conjunto de *Pagos*.

De un modo informal, la afirmación anterior expresa la idea de que el subtipo es *una clase de supertipo*. *PagoconTarjeta* es *una clase de Pago*. En un lenguaje más técnico, *es-una-clase-de* que se designa como *es-un*.

Este tipo de conformidad es la *Regla es-un*:

Regla es-un

Todos los miembros de un conjunto de subtipos han de pertenecer a su conjunto de supertipos.

En el lenguaje natural, esto casi siempre podemos probarlo informalmente formulando el enunciado

El subtipo es un supertipo

Por ejemplo, el enunciado el *PagoconTarjeta es un Pago* tiene sentido y expresa la idea de conformidad con la pertenencia a un conjunto.

28.2.5 ¿Qué es un subtipo correcto?

Basándose en la explicación que acabamos de ofrecer, aplique las siguientes pruebas¹ para definir un subtipo correcto cuando construya un modelo conceptual:

Un subtipo potencial deberá conformarse con:

- La regla del 100% (conformidad con la definición).
- La regla es-un (conformidad con la pertenencia a un conjunto).

28.3 Cuándo definir un subtipo

Hemos expuesto las reglas que garantizan la corrección de un subtipo (la regla del 100% y la regla es-un). Pero, ¿cuándo deberíamos tomarnos la molestia de definir un subtipo? Ante todo, una definición: una **partición de tipo** es una división del tipo en subtipos disjuntos [MO95].

¹ Los nombres de estas reglas se eligieron (en inglés) por su utilidad mnemotécnica más que por su precisión.

La pregunta también puede plantearse así: "¿cuándo conviene mostrar una partición de tipos?"

Por ejemplo, en el dominio del punto de venta, *Cliente* puede partirse (o dividirse en subtipos) correctamente en *ClienteVarón* y en *ClienteMujer*. ¿Pero es relevante o útil incluir esto en nuestro modelo (figura 28.6)?

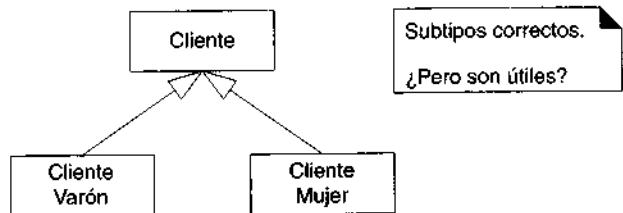


Figura 28.6 Particiones de tipos legales, ¿pero son útiles en nuestro dominio?

Esta partición no es útil en nuestro dominio; en la siguiente sección explicamos el porqué.

28.3.1 Motivos para partir un tipo en subtipos

A continuación se enumeran algunos motivos muy convincentes para partir un tipo en subtipos.

Se crea un subtipo de un supertipo cuando:

1. El subtipo tiene otros atributos más de interés.
2. El subtipo tiene otras asociaciones más de interés.
3. Se opera sobre el concepto del subtipo, se maneja, se reacciona ante él o se manipula de modo diferente a como se haría con el supertipo u otros subtipos, todo ello en aspectos que resultan relevantes.
4. El concepto de subtipo representa una cosa animada (un robot o un animal, por ejemplo) que se comporta de manera distinta al supertipo o a otro subtipo en aspectos que resultan relevantes.

Conforme a los criterios anteriores, no es obligatorio partir *Cliente* en *ClienteVarón* y *ClienteMujer* porque no poseen atributos ni asociaciones adicionales, no se opera sobre ellos (ni se les trata) en forma diferente y tampoco muestran un comportamiento distinto que resulte de interés.¹

¹ Los hábitos de compra de los hombres son distintos a los de las mujeres. Pero tales diferencias no son importantes para los requerimientos de nuestro caso de uso, lo cual constituye el criterio que rige nuestra investigación.

A continuación se incluyen algunos ejemplos de partición de tipos en el dominio de los pagos y en otras áreas, aplicando los siguientes criterios:

Justificación del subtipo	Ejemplos
El subtipo tiene otros atributos más de interés.	Pagos: no aplicable.
Biblioteca: <i>Libro</i> , subtipo de <i>RecursoPrestable</i> , tiene un atributo de <i>ISBN</i> .	
El subtipo tiene otras asociaciones más de interés.	Pagos: <i>PagoconTarjeta</i> , subtipo de <i>Pago</i> , se asocia a <i>TarjetadeCredito</i> .
Biblioteca: <i>Video</i> , subtipo de <i>RecursoPrestable</i> , se asocia a <i>Director</i> .	
Se opera sobre el concepto del subtipo, se maneja, se reacciona ante él o se manipula en forma diferente a como se haría con el supertipo u otros subtipos, todo ello en aspectos que resultan de interés.	Pagos: <i>PagoconTarjeta</i> , subtipo de <i>Pago</i> , se maneja en forma distinta a otros tipos de pago en lo tocante a la forma de autorizarse.
Biblioteca: <i>Software</i> , subtipo de <i>RecursoPrestable</i> , requiere un depósito para que se preste.	
El concepto de subtipo representa un ser animado (un robot o un animal, por ejemplo) cuyo comportamiento difiere del de otro supertipo o subtipos en aspectos que resultan de interés.	Pagos: no aplicable.
Biblioteca: no aplicable.	
Investigación de mercado: <i>HumanoVaron</i> , subtipo de <i>Humano</i> , no se comporta igual que <i>HumanoMujer</i> respecto a los hábitos de compra.	

28.4 Cuándo definir un supertipo

Se acostumbra convertir la generalización en un supertipo común cuando se descubren aspectos comunes entre los subtipos potenciales.

A continuación se enumeran algunos motivos que hacen generalizar y definir un supertipo.

Se crea un supertipo en una relación de generalización con subtipos cuando:

- Los subtipos potenciales representan variaciones de un concepto semejante.
- Los subtipos se conforman a las reglas del 100% y de es-un.
- Todos los subtipos con un mismo atributo pueden factorizarse y expresarse en el supertipo.
- Todos los subtipos tienen la misma asociación que puede factorizarse y relacionarse con el supertipo.

En las siguientes secciones los puntos anteriores se aclaran por medio de ejemplos.

28.5 Jerarquías de los tipos del punto de ventas

28.5.1 Tipos de pago

Conforme a los criterios precedentes con que se parte el tipo *Pago*, conviene elaborar una jerarquía de tipos de varias clases de pagos. El motivo de crear supertipos y subtipos se indica en la figura 28.7.

28.5.2 Tipos del servicio de autorización

Los servicios de autorización de créditos y cheques son variantes de un concepto similar y presentan atributos comunes de interés. Se obtiene así la jerarquía de tipos de la figura 28.8.

El supertipo se justifica por los atributos y a las asociaciones comunes.

A cada subtipo de pago se le da un tratamiento distinto.

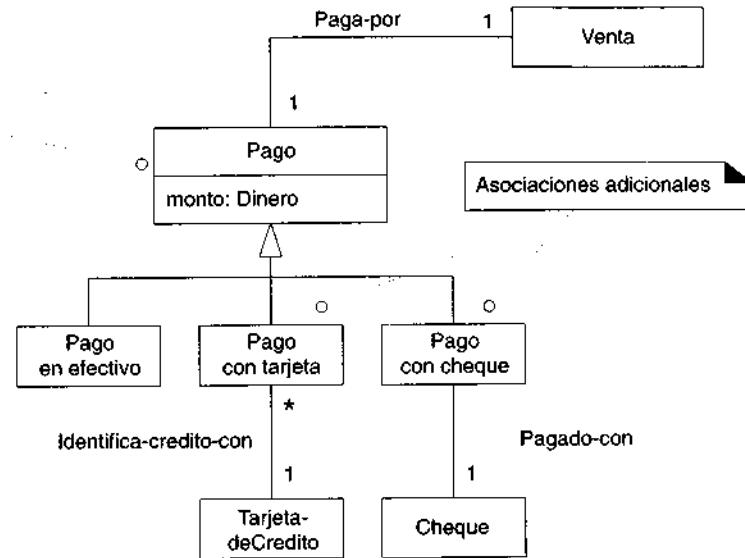


Figura 28.7 Justificación de los subtipos de *Pago*.

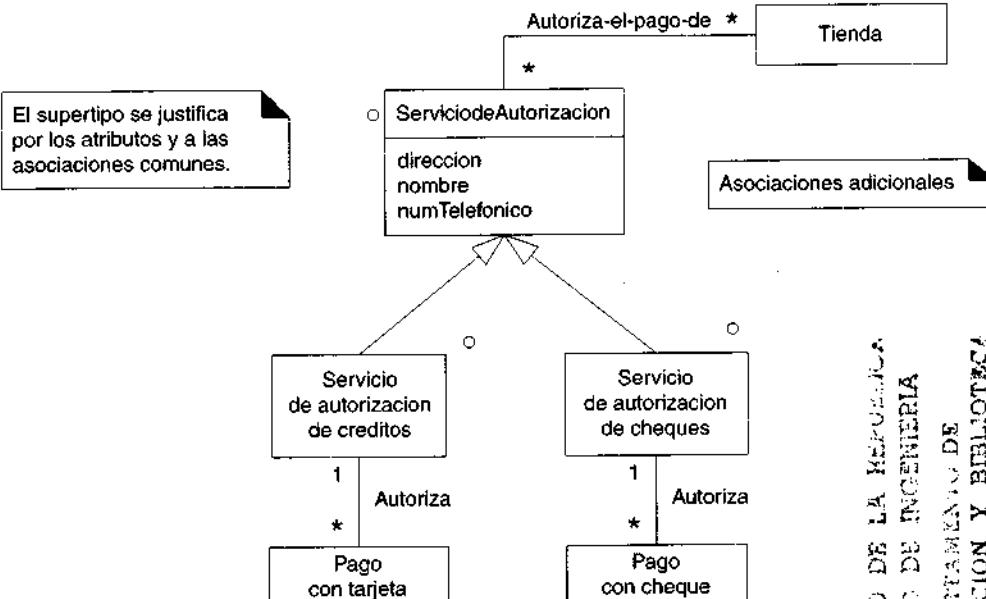


Figura 28.8 Justificación de la jerarquía *ServicioDeAutorizacion*.

28.5.3 Tipos de transacciones de autorización

El modelado de varios tipos de transacciones (solicitudes y respuestas) de los servicios de autorización presenta un caso interesante. Por lo regular las transacciones con los servicios externos sirven para mostrar un modelo conceptual, porque la actividad y los procesos suelen girar en torno a ellas. Se trata, pues, de conceptos importantes en extremo.

¿Debería el modelador dar ejemplos de *todas* las variantes de una transacción con algún servicio externo? No podemos ofrecer una respuesta válida para todos los casos. Según mencionamos con anterioridad, los modelos conceptuales no son intrínsecamente correctos ni erróneos, sino de utilidad variable. Esto debemos tenerlo presente, porque cada tipo de transacción se relaciona con distintos conceptos, procesos y reglas de negocios.¹

Un segundo punto de mucho interés es el grado de generalización cuya utilidad respalde su inclusión en el modelo. Para facilitar nuestra explicación supongamos que toda transacción tiene fecha y hora. Estos atributos comunes, junto con el deseo de formular una generalización fundamental para esta familia de conceptos relacionados, justifica la creación de la *TransacciondeAutorizaciondePago*.

¿Pero conviene generalizar una respuesta e incorporarla a *RespuestadeAutorizaciondeCredito* y *RespuestadeAutorizaciondePagoconCheque*, como se advierte en la figura 28.9, o basta para mostrar una menor generalización, como se indica en la figura 28.10?

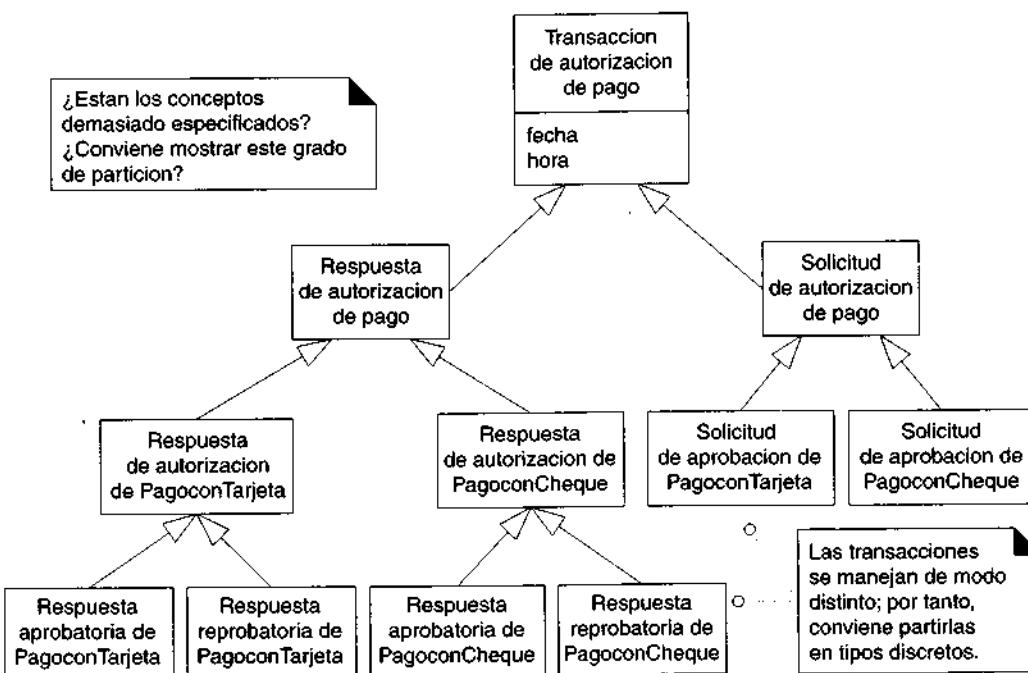


Figura 28.9 Una jerarquía posible de tipos para las transacciones con un servicio externo.

¹ En los modelos conceptuales de telecomunicaciones, también recomendamos identificar cada tipo de mensaje de cambio o intercambio.

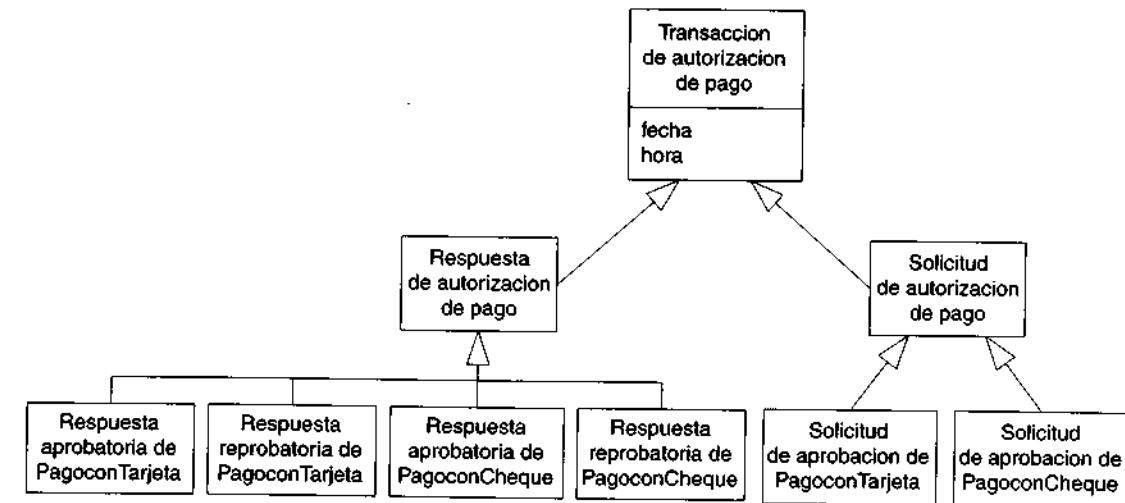


Figura 28.10 Jerarquía alterna de tipos de transacción.

El tipo de jerarquía de la figura 28.10 es suficientemente útil en lo tocante a la generalización, porque las generalizaciones complementarias no aportan un valor evidente. La jerarquía de la figura 28.9 expresa una granularidad más fina de generalización, la cual no mejora significativamente nuestro conocimiento de los conceptos ni de las reglas de negocios; pero sí hace más intrincado el modelo, además de que no conviene una mayor complejidad salvo que aporte otros beneficios.

28.6 Tipos abstractos

En la presente sección definiremos la designación del tipo abstracto. Se recomienda identificar esta clase de tipos en un modelo conceptual porque limitan de cuáles tipos es posible dar instancias concretas, aclarando con ello las reglas del dominio del problema.

Si todos los miembros de un tipo T han de serlo también de un subtipo, al tipo T se le da el nombre de tipo abstracto.

Supongamos, por ejemplo, que todas las instancias *Pago* han de ser más específicamente una instancia del subtipo *PagoconTarjeta*, *PagoenEfectivo* o *PagoconCheque*. Esto se muestra gráficamente con el diagrama de Venn de la figura 28.11 (b). Dado que todo miembro *Pago* pertenece también a un subtipo, *Pago* será un tipo abstracto por definición.

En cambio, si puede haber instancias *Pago* que no sean miembro de un subtipo, no será un tipo abstracto según se aprecia en la figura 28.11 (a).

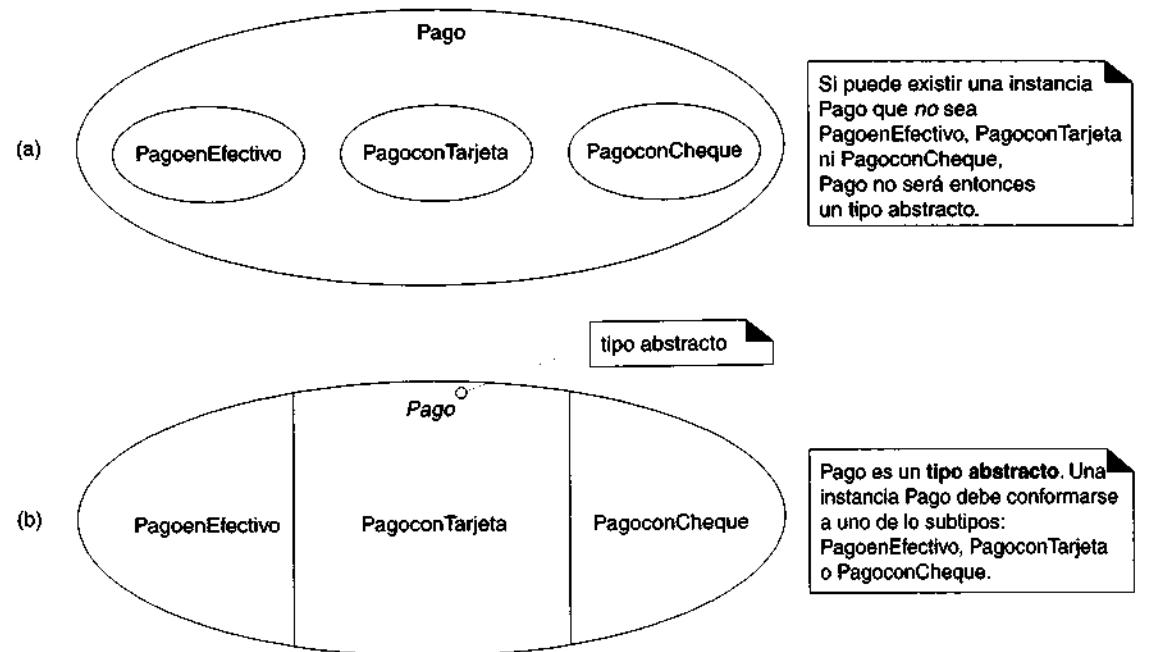
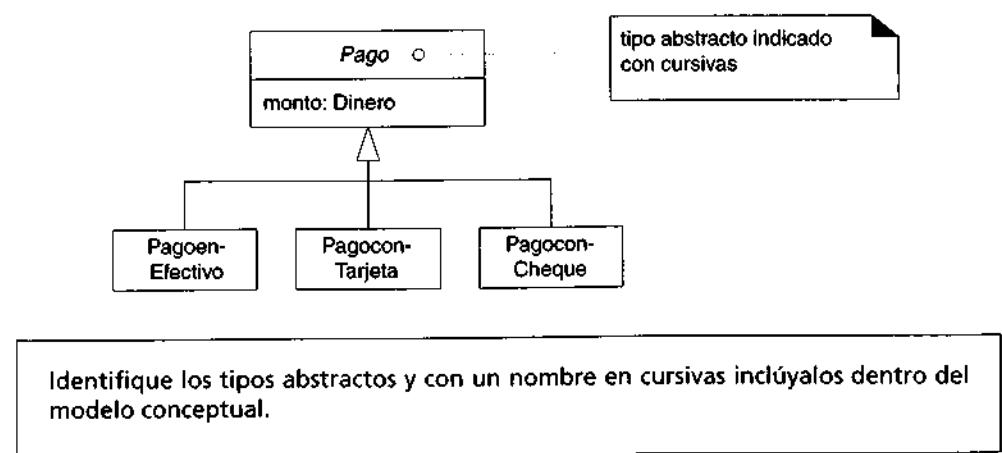


Figura 28.11 Tipos abstractos.

En el dominio del punto de venta, todo *Pago* es en realidad miembro de un subtipo. La figura 28.11 (b) es la descripción exacta de los pagos y, por tanto, *Pago* es un tipo abstracto.

28.6.1 Notación de tipos abstractos en el lenguaje UML

El UML cuenta con una notación que indica los tipos abstractos, y el nombre de tipo se imprime en cursivas.



28.6.2 Clases y métodos abstractos

Si un tipo abstracto se implementa en el software como una clase durante la fase de diseño, solemos representarlo con una **clase abstracta**, lo cual significa que no podemos crear instancias para ella. Un **método abstracto** es aquel que se declara en una clase abstracta, pero que no se implementa en ella; en el lenguaje UML se escribe también en cursivas.

28.7 Construcción de modelos con estados cambiantes

Suponga que un pago puede estar en estado autorizado o no autorizado, y que podemos presentar esos dos estados en el modelo conceptual (tal vez no sea así, pero lo supondremos en nuestra exposición). Como se observa en la figura 28.12, una forma de construir el modelo respectivo consiste en definir los subtipos de *Pago*: *PagoNoAutorizado* y *PagoAutorizado*. Pero adviértase que un pago no permanece en uno de estos estados; ordinariamente se da una transición del estado no autorizado al autorizado. De aquí deducimos la siguiente directriz:

No modele los estados de un concepto X como subtipos de X. Por el contrario:

1. Defina una jerarquía de estados y asóciela a X.
2. No incluya los estados de un concepto en el modelo conceptual; inclúyelos en los diagramas de estado.

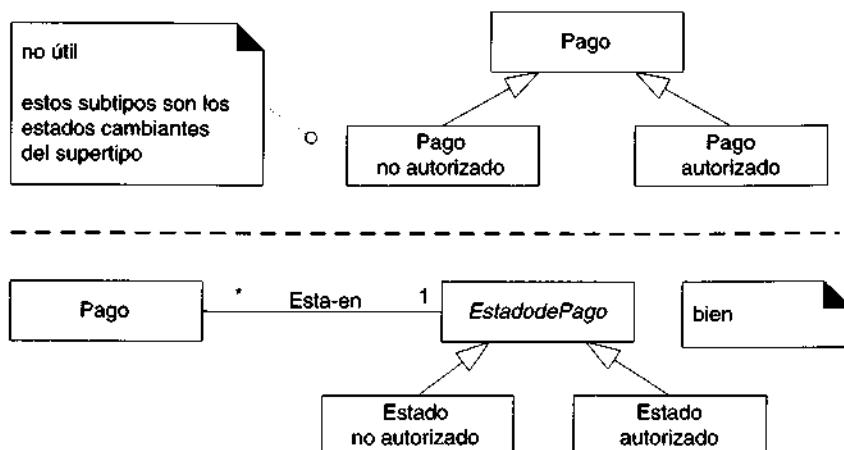


Figura 28.12 Modelado de los estados cambiantes.

28.8 Jerarquías de clases y herencia

En nuestra exposición sobre las jerarquías de tipos no hemos mencionado la *herencia*, porque nos hemos centrado en un modelo conceptual de entidades del mundo real, no en artefactos de software. Una clase es una implementación de un concepto o de un tipo en software, y en un lenguaje de programación orientado a objetos una subclase hereda las definiciones de atributos y de operaciones de sus superclases, gracias a la creación de **jerarquías de clases**. La *herencia* es un mecanismo de software que implementa la conformidad de los subtipos con sus definiciones de supertipo. Por tanto, la herencia no interviene en la explicación de un modelo conceptual, pero sí lo hace cuando efectuamos la transición a la fase de diseño.

Las jerarquías de tipos aquí generadas pueden reflejarse o no en nuestra solución. Así, es posible reducir o ampliar la jerarquía de los tipos de transacciones relacionadas con los servicios de autorización, integrándolas a las jerarquías de clases de software, según las características del lenguaje y otros factores. Por ejemplo, las clases de plantillas de C++ podrían servirnos para reducir el número de clases por desarrollar.

PAQUETES: ORGANIZACIÓN DE LOS ELEMENTOS

Objetivos

- Organizar los elementos en paquetes.

29.1 Introducción

El modelo conceptual del segundo ciclo de desarrollo para la aplicación del punto de venta comienza a tener un tamaño excesivo; advertimos entonces la necesidad de dividir sus elementos en subconjuntos más pequeños.

Organizar los elementos en paquetes ofrece la ventaja de separar los elementos detallados en abstracciones más amplias, lo cual brinda soporte a una vista de nivel superior y permite contemplar el modelo en agrupamientos más simples.

En este capítulo examinaremos cómo se utilizan los paquetes para particionar el modelo conceptual.

29.1.1 Capas y particiones

Según señalamos antes, podemos concebir la arquitectura global del sistema como si estuviera compuesta de particiones verticales y horizontales. Los paquetes del modelo conceptual, en caso de usarse en el diseño, pueden considerarse *particiones* de la capa de objetos del dominio.

29.2 Notación de los paquetes en el UML

Un paquete se muestra gráficamente como una carpeta etiquetada (figura 29.1). En su interior podemos incluir los paquetes subordinados. El nombre del paquete se halla dentro de la etiqueta si el paquete describe sus elementos; de lo contrario aparecerá en el centro de ella.

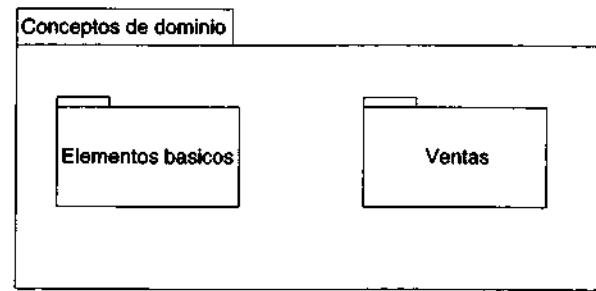


Figura 29.1 Un paquete representado con el lenguaje UML.

29.2.1 Propiedad y referencias

Un elemento —un tipo o una clase, por ejemplo— es *propiedad* del paquete dentro del cual está definido, pero puede *referenciarse* en otros. En tal caso su nombre se especifica con el del paquete que emplea el formato del nombre de la trayectoria *NombredelPaquete::NombredelElemento* (figura 29.2). El tipo o clase incluidos en un paquete extraño pueden modificarse con nuevas asociaciones, pero por lo demás han de permanecer inalterados.

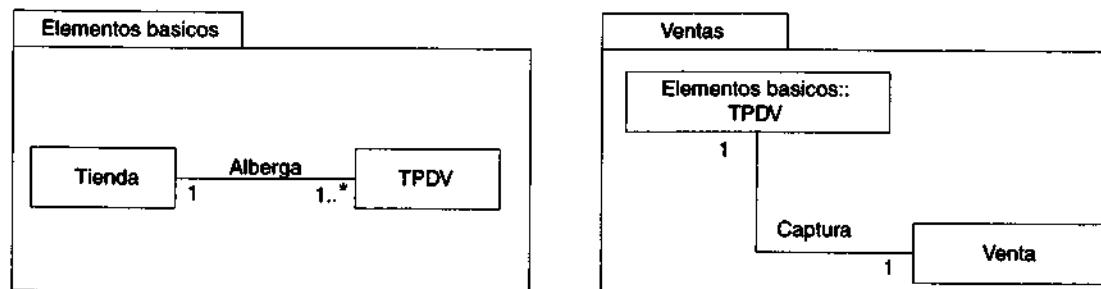


Figura 29.2 Tipo referenciado en un paquete.

29.2.2 Dependencias de los paquetes

Si de alguna manera un elemento de un modelo depende de otro, la dependencia puede mostrarse con una relación de subordinación, que indica una línea con flecha.

La dependencia de un paquete significa que sus elementos conocen en alguna forma los del paquete objetivo y están acoplados a ellos. Por ejemplo, existirá dependencia si un paquete referencia un elemento que es propiedad de otro. Así, el paquete *Ventas* tiene dependencia respecto al de *Elementos Básicos* (figura 29.3).

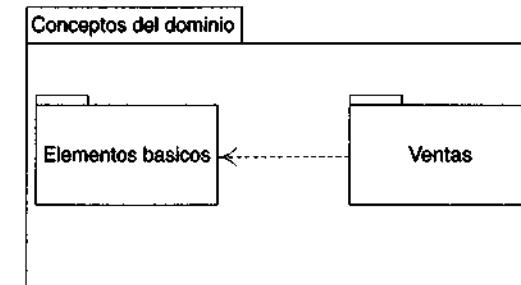


Figura 29.3 Una dependencia de paquete.

29.2.3 Indicación de paquetes sin diagrama

A veces no conviene dibujar un diagrama de paquete, pero sí indicar el paquete del cual forman parte los elementos.

En tal caso se incluye una nota de restricción (nota con una esquina doblada) en el diagrama, como se aprecia en la figura 29.4.

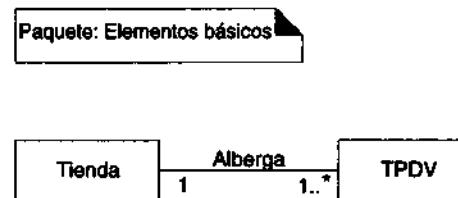


Figura 29.4 Presentación gráfica de la propiedad de un paquete con restricción.

29.3 Cómo partir el modelo conceptual

En esta sección nos centraremos en la partición del modelo conceptual; en el capítulo 22 explicaremos la organización de la arquitectura del software en paquetes.

¿Cómo debemos organizar en paquetes los tipos de un modelo conceptual?

La respuesta es sencilla: basta aplicar las siguientes directrices generales:

Para partir el modelo conceptual en paquetes, reúna los elementos que:

- se encuentren en la misma área o tema, o sea que estén estrechamente relacionados por un concepto o propósito
- se encuentren en la misma jerarquía de tipos
- participen en los mismos casos de uso
- presenten una asociación muy íntima.

Conviene que todos los elementos relacionados con el modelo conceptual estén contenidos en un paquete denominado *Conceptos del Dominio* y que los conceptos comunes compartidos ampliamente se definan con un nombre empaquetado como *Elementos Básicos* o *Conceptos Comunes*, cuando no haya un paquete significativo donde incluirlos.

29.4 Paquetes del modelo conceptual del punto de venta

Con base en los criterios anteriores, en la figura 29.5 se ofrece la organización de un paquete para el dominio del punto de venta.

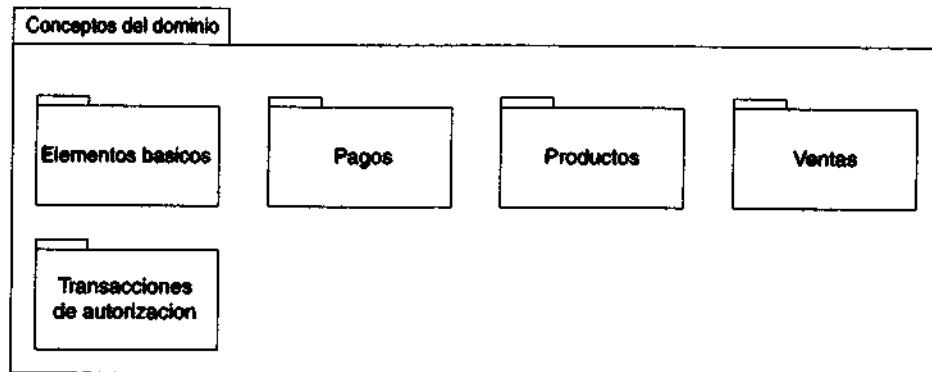


Figura 29.5 Paquetes de conceptos del dominio.

Los diagramas de paquete de las figuras 29.6 y 29.7 describen gráficamente la propiedad de los tipos individuales. En los siguientes capítulos vamos a examinar los detalles de las asociaciones y de los atributos.

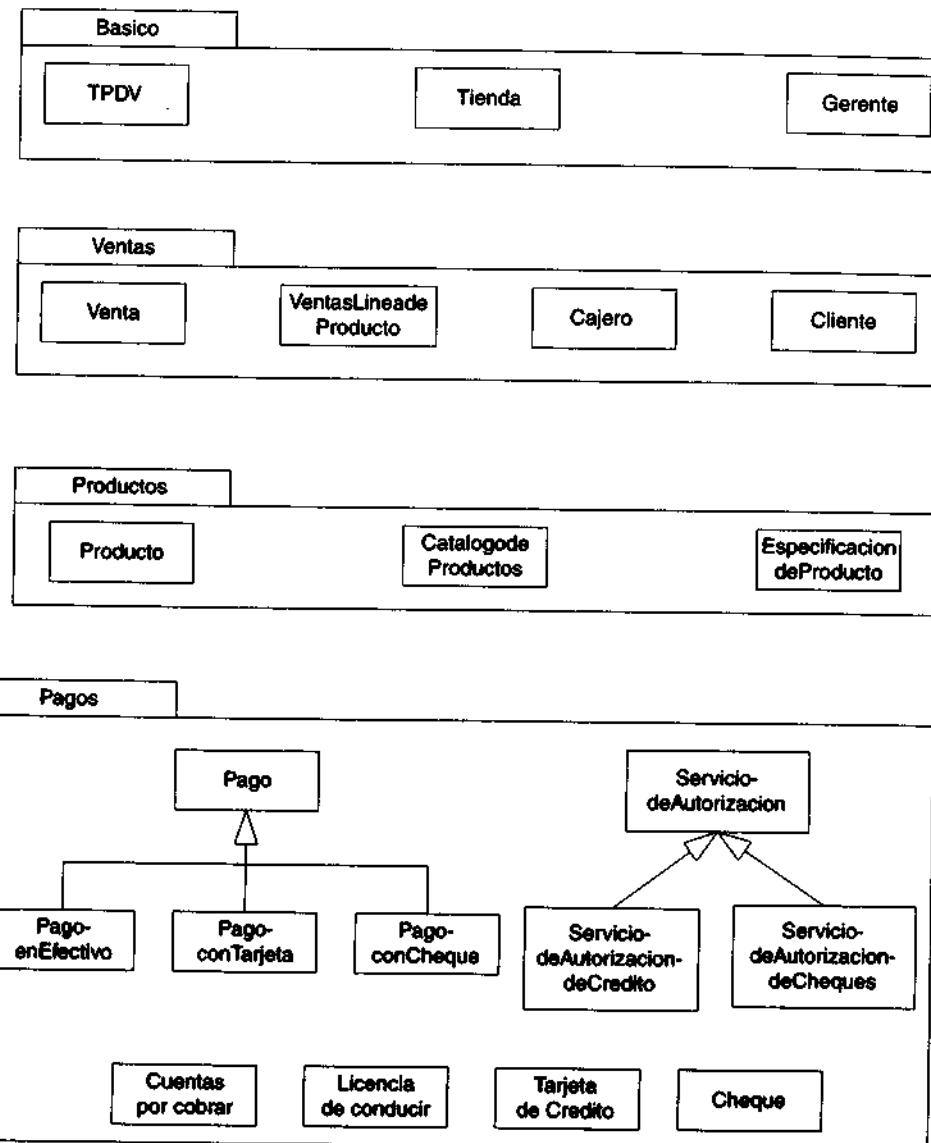


Figura 29.6 Paquetes del dominio.

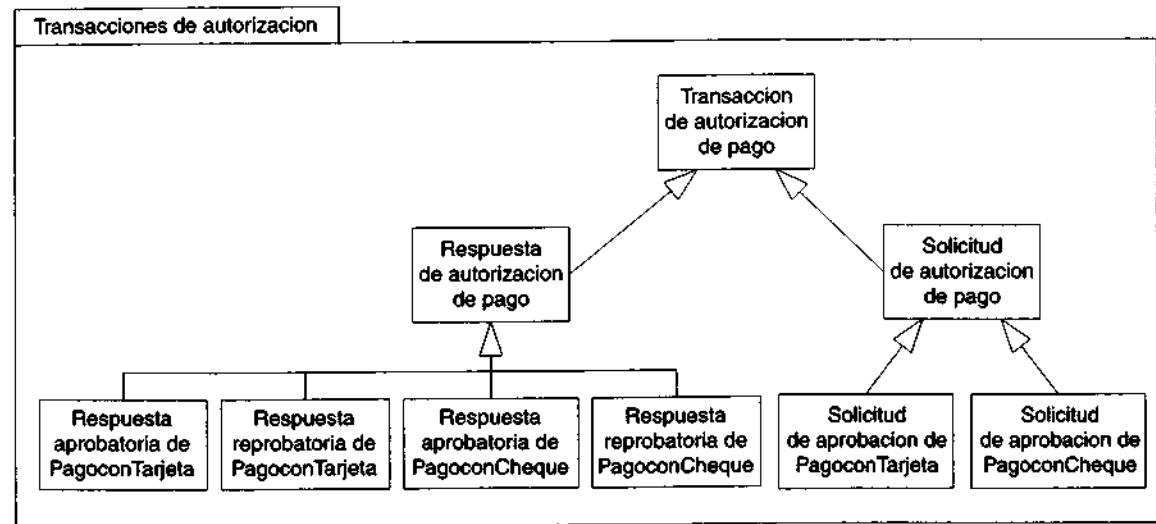


Figura 29.7 Más paquetes del dominio.

REFINAMIENTO DEL MODELO CONCEPTUAL

Objetivos

- Agregar tipos asociativos al modelo conceptual.
- Incorporar las relaciones de agregación.
- Decidir cómo construir los modelos de papeles (funciones que se desempeñan).

30.1 Introducción

En este capítulo vamos a estudiar más ideas y otra notación con que se construyen los modelos conceptuales y vamos a aplicarlos para perfeccionar los aspectos del modelo del punto de venta. En el capítulo siguiente consolidaremos los resultados aquí obtenidos.

30.2 Tipos asociativos

Los siguientes requerimientos del dominio preparan el terreno para los tipos asociativos:

- Los servicios de autorización asignan a las tiendas una identificación comercial que les permite identificarlas durante la comunicación.
- Una solicitud de autorización de pago hecha por una tienda a un servicio de autorización exige incluir la identificación comercial que la identifica ante él.
- Además, una tienda tiene una identificación comercial para cada servicio.

¿En qué parte del modelo conceptual debería residir el atributo de identificación (ID)? Es incorrecto colocar *idComercial* en la *Tienda* porque una *Tienda* puede tener más de un valor de *idComercial*. Lo mismo sucede cuando lo ponemos en *Servicio de Autorización* (figura 30.1).

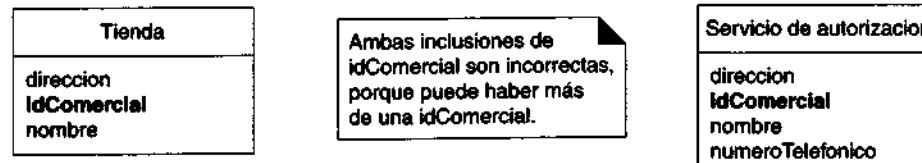


Figura 30.1 Uso incorrecto de un atributo.

De lo que acabamos de decir extraemos el siguiente principio de la construcción de modelos:

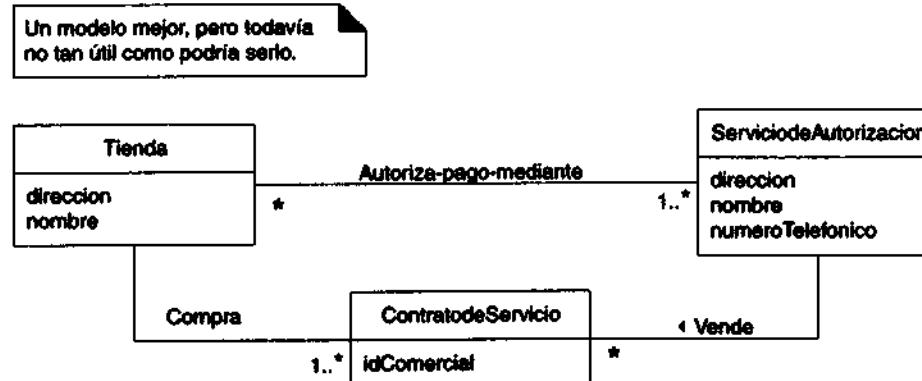
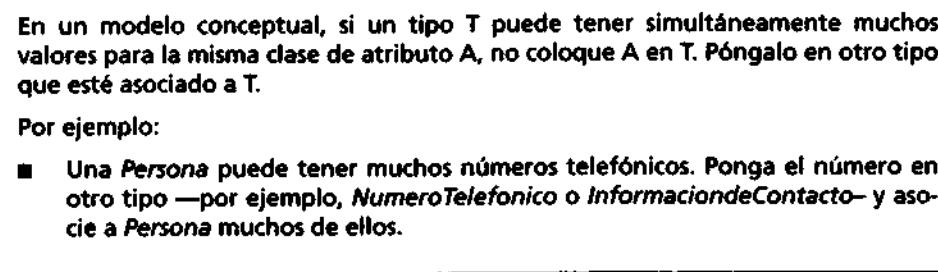


Figura 30.2 Primer intento de construir el modelo del problema de *idComercial*.

El principio anterior establece que se requiere algo como el modelo de la figura 30.2. En el mundo de los negocios, ¿qué concepto registra formalmente la información referente a los servicios que una compañía ofrece a un cliente?: un *Contrato* o una *Cuenta*.

El hecho de que tanto *Tienda* como *Servicio de Autorización* se relacionen con el *ContratodeServicio* es una indicación dependiente de la relación que existe entre los dos. Podemos considerar la *idComercial* como un atributo relacionado con la asociación entre la *Tienda* y el *Servicio de Autorización*.

Esto desemboca en el concepto de **tipo asociativo** en que podemos agregar características a la asociación.

Podemos modelar el *ContratodeServicio* como un tipo asociativo relacionado con la asociación que existe entre *Tienda* y *Servicio de Autorización*.

En el UML, esto se denota con una línea punteada que de la asociación se dirige al tipo asociativo (figura 30.3).

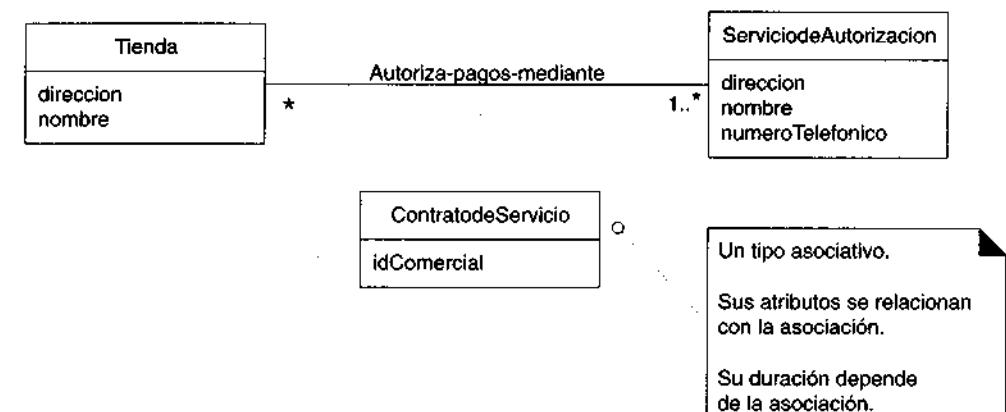


Figura 30.3 Un tipo asociativo.

La figura 30.3 nos transmite visualmente la idea de que un *ContratodeServicio* y sus atributos se relacionan con la asociación existente entre *Tienda* y *Servicio de Autorización*, y de que la duración del *ContratodeServicio* depende de la relación.

30.2.1 Directrices

A continuación ofrecemos algunas normas para agregar los tipos asociativos.

Indicaciones de que un tipo asociativo puede ser útil en un modelo conceptual:

- Un atributo está relacionado con una asociación.
- Las instancias del tipo asociativo presentan una dependencia de toda la vida respecto a la asociación.
- Hay asociaciones de muchos a muchos entre los dos conceptos, y la información se relaciona con la propia asociación.
- Sólo existe una instancia del tipo asociativo entre dos objetos que participan en la asociación.

La presencia de la asociación de muchos a muchos es una indicación común de que un tipo asociativo útil se halla en alguna parte del trasfondo; cuando vea uno, piense en la conveniencia de utilizar un tipo asociativo.

He aquí otros ejemplos de tipos asociativos:

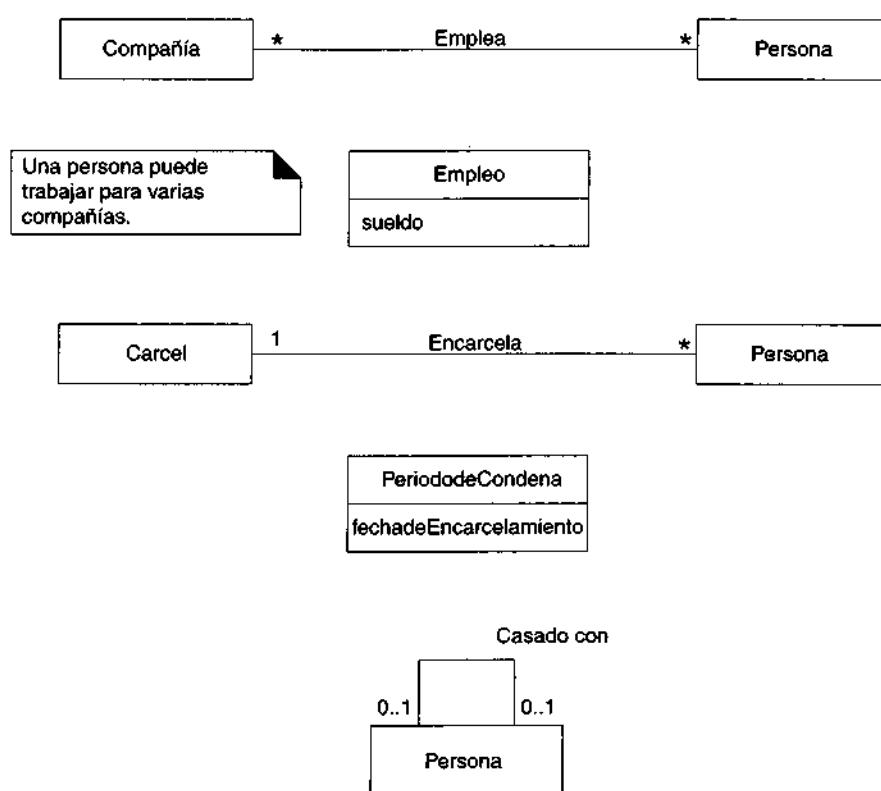


Figura 30.4 Tipos asociativos.

30.3 Agregación y composición

La **agregación** es una clase de asociación con que se modelan las relaciones de parte-todo entre las cosas. Al todo se le llama generalmente **compuesto**; en cambio, las partes no tienen un nombre estándar: suele designárseles simplemente como *parte* o *componente*.

Así, los ensambles físicos se organizan en relaciones de agregación; por ejemplo, una *Mano* agrega o conjunta *Dedos*.

30.3.1 Agregación en el UML

La agregación se muestra en el UML con un símbolo de diamante en blanco o sombreado (en negro) en el extremo correspondiente al compuesto en una asociación de parte-todo (figura 30.5).

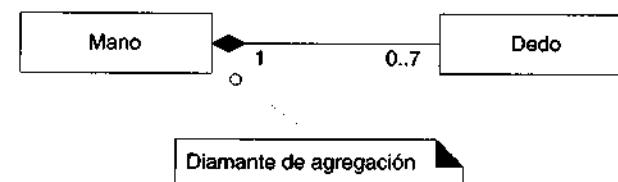


Figura 30.5 Notación de la agregación.

La agregación es una propiedad de un papel de asociación.¹

El nombre de la asociación frecuentemente se excluye en las relaciones de agregación, porque casi siempre se le considera como un *Tiene-partes*. Sin embargo, puede emplearse para proporcionar más detalles semánticos.

30.3.2 Agregación de compuestos: el diamante sombreado

La **agregación compuesta**, o simplemente **composición**, significa que la multiplicidad en el extremo correspondiente al compuesto puede ser al máximo una; la denotamos con un diamante sombreado. Indica que únicamente el compuesto posee la parte y que se encuentra en una jerarquía de partes con estructura de árbol; es la modalidad más frecuente de agregación que encontramos en los modelos.

Por ejemplo, un dedo es parte de no más de una mano y, por lo mismo, el diamante de agregación aparece sombreado para indicar la agregación compuesta (figura 30.6).

¹ Recuerde que cada extremo de una asociación es un papel y que un papel posee varias propiedades: multiplicidad, nombre, navegabilidad y esAgregado.

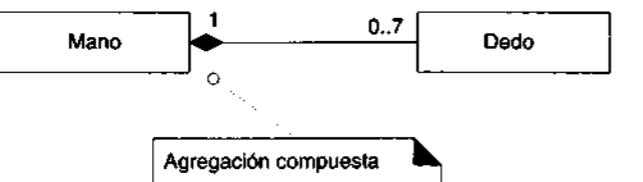


Figura 30.6 Agregación compuesta.

30.3.3 Agregación compartida: diamante en blanco

La **agregación compartida** significa que la multiplicidad en el extremo del compuesto puede ser más de una; se denota con un diamante en blanco. Indica que la parte puede estar en muchas instancias compuestas. La agregación compartida rara vez existe en agregados físicos, pues más bien la encontramos en conceptos no físicos.

Por ejemplo, podemos considerar un paquete de UML para agregar sus elementos. Pero podemos referenciar un elemento en más de un paquete (es propiedad de alguien y se referencia en otros); éste es un ejemplo de la agregación compartida (figura 30.7).

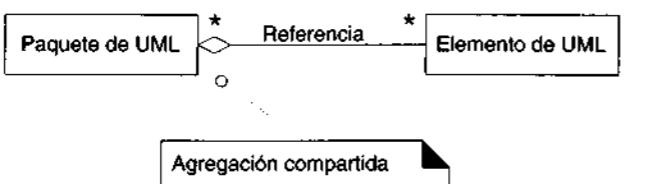


Figura 30.7 Agregación compartida.

30.3.4 Cómo identificar una agregación

En algunos casos la presencia de la agregación es patente: generalmente en los ensambles físicos. Pero otras veces no es tan evidente.

Al agregar: en caso de duda, no incluya el elemento.

A continuación se dan algunas directrices que señalan cuándo mostrar la agregación:

Estudie la conveniencia de mostrar la agregación si:

- La duración de la parte es dependiente de la que tiene el compuesto: la parte muestra una dependencia de crear-eliminar respecto al todo.
- Existe un evidente ensamblado físico o lógico de parte-todo.
- Algunas propiedades del compuesto se difunden hacia las partes, entre ellas su ubicación.
- Las operaciones aplicadas al compuesto se propagan a las partes: destrucción, movimiento, registro.

Si excluimos el hecho de que algo sea un ensamblado evidente de partes, la siguiente indicación más útil es la dependencia de crear-eliminar que la parte presenta respecto al todo.

30.3.5 Ventaja de mostrar la agregación

No es indispensable ni identificar ni mostrar la agregación; nada impide que excluyamos ambas actividades en un modelo conceptual. Recomendamos identificar y mostrar la agregación porque se obtienen los siguientes beneficios, algunos de los cuales se relacionan con la fase de solución del software más que con la de construcción del modelo conceptual. A esto se debe que excluirla en este último no sea muy importante.

- Aclara las restricciones del dominio concernientes a la existencia elegible de la parte independiente del todo. En la agregación compuesta, la parte no puede existir fuera de la duración del todo.
 - Durante la fase de solución, ello repercute en las dependencias crear-eliminar que presentan las clases de software de parte-todo.
- Ayuda a identificar al creador (el compuesto) por medio del patrón Creador de GRASP.
- Las operaciones —como copiar o eliminar— que se aplican al todo deberían propagarse a las partes.
- El identificar un todo en relación con una parte da soporte al encapsulamiento. El patrón No hables con extraños de GRASP sirve para ocultar las partes dentro del todo.

30.3.6 Agregación en el modelo del punto de venta

En el dominio del punto de venta, la instancia *VentasLineadeProducto* puede considerarse parte de un compuesto *Venta*; en una transacción general, la línea de productos

se ve como parte de una transacción agregada (figura 30.8). Además de la conformidad con ese patrón, se observa una dependencia crear-eliminar en la línea de productos de la Venta: su duración no rebasa la de la Venta.

En virtud de una justificación semejante, la instancia *CatalogodeProductos* es un agregado de *EspecificacionesdeProducto*.

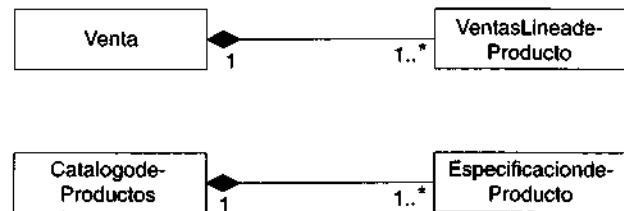


Figura 30.8 Agregación en la aplicación del punto de venta.

Ninguna otra relación es una combinación obligatoria que indique la semántica de parte-todo y una dependencia crear-eliminar; “en caso de duda, no se incluirá”.

30.4 Nombres de los papeles de la asociación

Cada extremo de una asociación es un papel y posee varias propiedades como:

- nombre
- multiplicidad

El nombre del papel identifica un extremo de una asociación y, en teoría, describe la función que los objetos desempeñan en la asociación.

La figura 30.9 contiene ejemplos de nombres de papeles.

No se requiere un nombre explícito de los papeles, aunque resulta de gran utilidad cuando el papel del objeto no es claro. Suele comenzar con una letra minúscula. Si no aparece explícitamente, suponga que el nombre por omisión es igual al nombre del tipo relacionado aunque deberá indicarse con letra minúscula.

Según señalamos ya en este capítulo, los papeles que aparecen en los diagramas de diseño pueden interpretarse como el criterio para asignar nombre a los atributos cuando se genera el código.

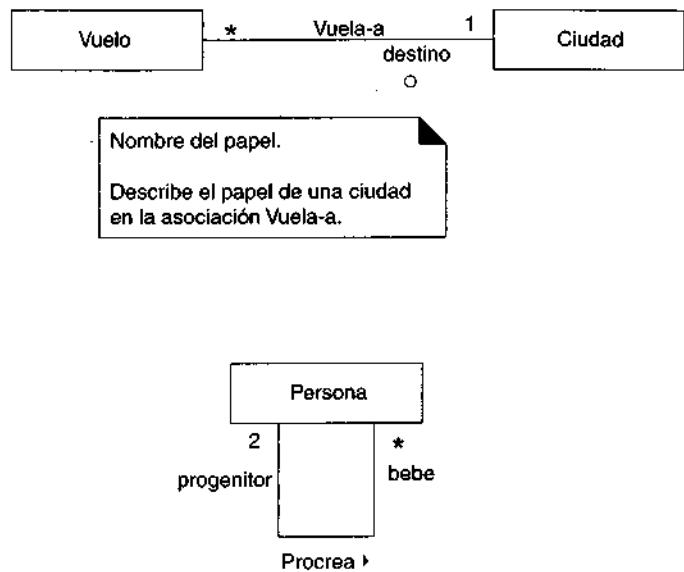


Figura 30.9 Nombres de papeles.

30.5 Los papeles como conceptos y los papeles en las asociaciones

En un modelo conceptual, un papel del mundo real —sobre todo un papel humano— puede modelarse en varias formas, como concepto independiente, o expresarse como un papel en una asociación.¹

Por ejemplo, el papel de un cajero y el de gerente pueden expresarse al menos en las formas que se describen gráficamente en la figura 30.10.

A la primera forma podemos darle el nombre de “papeles en las asociaciones”; al segundo el de “papeles como conceptos”. Ambas modalidades ofrecen ventajas.

Los papeles en las asociaciones son interesantes por constituir un medio bastante preciso de expresar el concepto de que la misma instancia de una persona adopta múltiples papeles (y cambia dinámicamente) en diversas asociaciones. Así, yo —una persona— puedo asumir simultáneamente o en sucesión el papel de maestro, usuario de la tecnología de objetos, padre u otros papeles.

¹ Para no entrar en detalles, prescindimos de otras soluciones excelentes como las que se comentan en [Fowler96].

Por otra parte, los papeles como conceptos ofrecen facilidad y flexibilidad en la incorporación de atributos y asociaciones especiales, así como una semántica complementaria. Más aún, la implementación de papeles como clases aisladas resulta más fácil por las limitaciones de los actuales lenguajes orientados a objetos: es difícil transformar dinámicamente una instancia de una clase en otra o agregar dinámicamente comportamiento y atributos conforme el papel de una persona cambia.¹

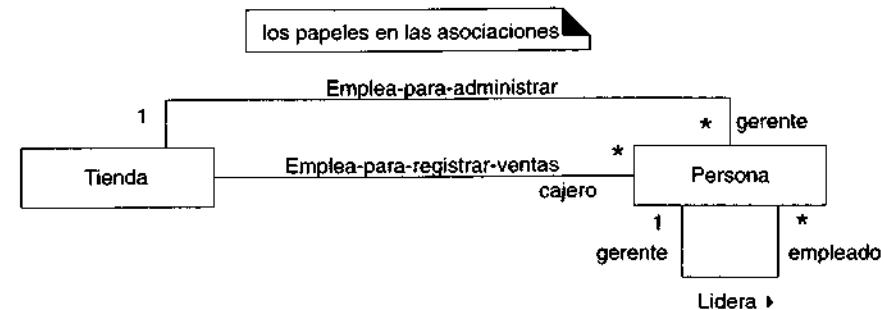


Figura 30.10 Dos formas de modelar los papeles humanos.

30.6 Elementos derivados

Un elemento derivado puede determinarse a partir de otros. Los atributos y las asociaciones son los más comunes de ellos.

Por ejemplo, podemos衍生 la información relativa a una *Venta total* y a *VentasLineadeProducto* (figura 30.11). En el UML esto se indica anteponiéndole la diagonal “/” al nombre del elemento.

¹ Lo que se necesita es un lenguaje coloquial que brinde apoyo a los papeles múltiples y de cambio dinámico. Una buena solución parece ser la tecnología de Java basada en agentes.

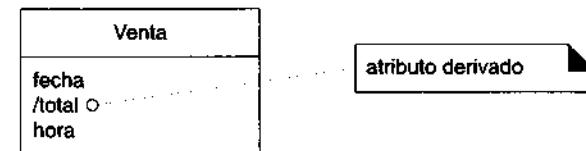


Figura 30.11 Atributo derivado.

¿Cuándo deberíamos mostrar los elementos derivados?

No muestre en un diagrama los elementos derivados, pues aumentan la complejidad sin aportar nueva información. Pero agregue un elemento derivado cuando ocupe un lugar prominente en la terminología del dominio y cuando su exclusión dificulte la comprensión.

Un ejemplo más: una *cantidad de VentaLineadeProducto* es en realidad derivable del número de instancias de *Productos* asociados a la línea (figura 30.12).

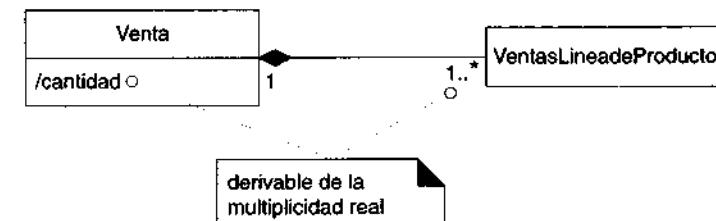


Figura 30.12 Atributo derivado relacionado con la multiplicidad.

30.7 Asociaciones calificadas

Podemos usar un **calificador** en una asociación; su función consiste en distinguir un conjunto de objetos situado en el extremo de la asociación que se basa en el valor del calificador. Se da el nombre de **asociación calificada** a la que posee un calificador.

Por ejemplo, podemos distinguir *EspecificacionesdeProducto* en un *CatalogodeProductos* por su código universal de producto (CUP), como se observa en la figura 30.13 (b). En contraste con la figura 30.13(a) frente a (b), con la calificación se reduce la multiplicidad en el extremo del calificador, generalmente de muchos a uno. La descripción de un calificador en un modelo conceptual señala cómo, en el dominio, los miembros de un tipo se diferencian de otro. En el modelo conceptual no deberían servir para expresar las decisiones de diseño concernientes a las claves de consulta, lo cual sí conviene hacer en los diagramas posteriores de diseño.

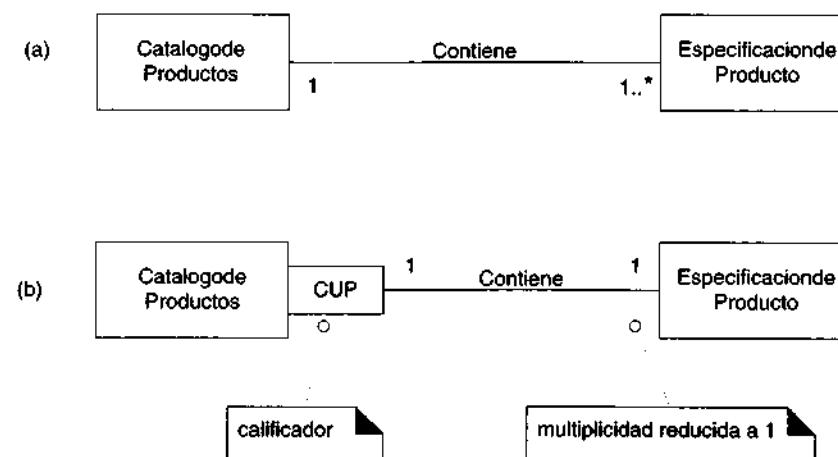


Figura 30.13 Asociación calificada.

Por lo regular los calificadores aportan información nueva y útil, existiendo además el riesgo de caer en la trampa del “pensamiento orientado al diseño”. Pero cuando se utilizan correctamente, pueden aumentar nuestro conocimiento del dominio. Las asociaciones calificadas entre *CatalogodeProductos* y *EspecificacioneProducto* constituyen un ejemplo razonable de un calificador que añade valor al modelo.

30.8 Asociaciones recursivas o reflexivas

Un concepto puede tener una asociación consigo mismo; a esto se le llama **asociación recursiva o reflexiva**.¹

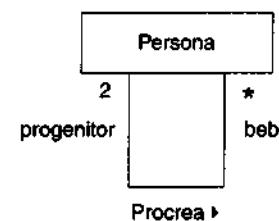


Figura 30.14 Asociación recursiva.

¹ [MO95] limita aún más la definición de las asociaciones reflexivas.

MODELO CONCEPTUAL: RESUMEN

Objetivos

- Consolidar la información del capítulo anterior referente al modelo conceptual y presentar gráficamente los resultados del modelo del punto de venta.

31.1 Introducción

En este capítulo vamos a reafirmar y explicar visualmente las ideas y la notación de UML que expusimos en los cuatro anteriores. Los resultados se presentan en un modelo conceptual íntegro para el segundo ciclo de desarrollo de la aplicación del punto de venta.

Un modelo conceptual no es intrínsecamente correcto ni erróneo, sino que ofrece una utilidad variable. En consecuencia, contiene decisiones que pueden ponerse en tela de juicio o modificarse por buenas razones, sobre todo al momento de elegir las asociaciones. Pero nos da una buena descripción de los conceptos y de sus relaciones importantes, lo bastante útil para sentar las bases de investigaciones posteriores y del trabajo en la fase de solución. Los estudios subsecuentes seguramente detectarán deficiencias y omisiones en el modelo actual, cosa por lo demás previsible. El modelo se actualizará conforme se vayan obteniendo mejoras.

31.2 Paquete de los conceptos del dominio

En el capítulo 29 se analizaron los motivos del paquete del dominio de alto nivel que se incluye en la figura 31.1. Aquí no se muestran las dependencias del paquete.

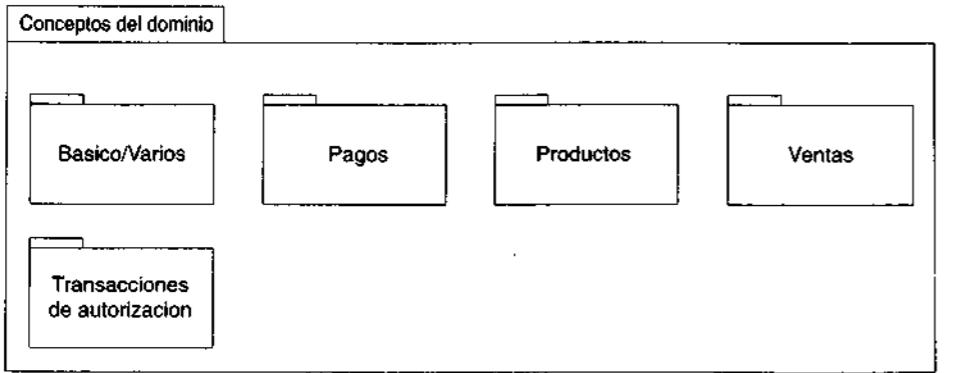


Figura 31.1 Paquete de conceptos del dominio de alto nivel.

31.3 Paquete básico/varios

Cuando se desee poseer conceptos compartidos por muchos o conceptos que carecen de un sitio bien conocido, conviene servirse de un paquete básico/varios (figura 31.2). En referencias ulteriores, designaremos a este paquete simplemente como *Básico*.

El paquete no contiene nuevos conceptos ni asociaciones propias de la iteración 2.

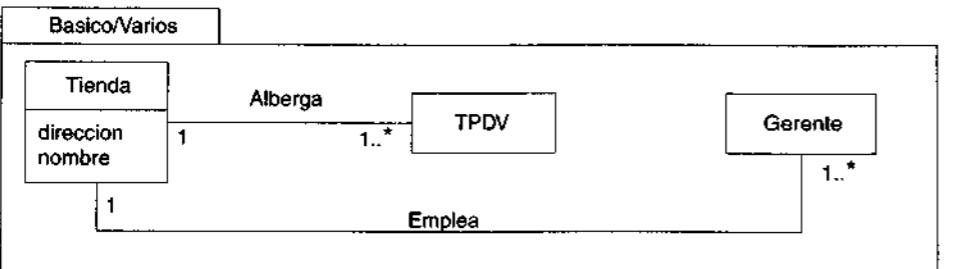


Figura 31.2 Paquete básico.

31.4 Pagos

Igual que en la iteración 1, las nuevas asociaciones se deben principalmente al criterio de la necesidad de conocer. Por ejemplo, es preciso recordar la relación entre un *Pago-con Tarjeta* y *Tarjeta de Credito*. En cambio, algunas asociaciones se agregan para mejorar la comprensión, como *LicenciadoConducir Identifica Cliente* (figura 31.3).

Nótese que *RespuestadeAutorizaciondePago* se expresa como una clase asociativa. Una respuesta surge de la asociación entre un pago y su servicio de autorización.

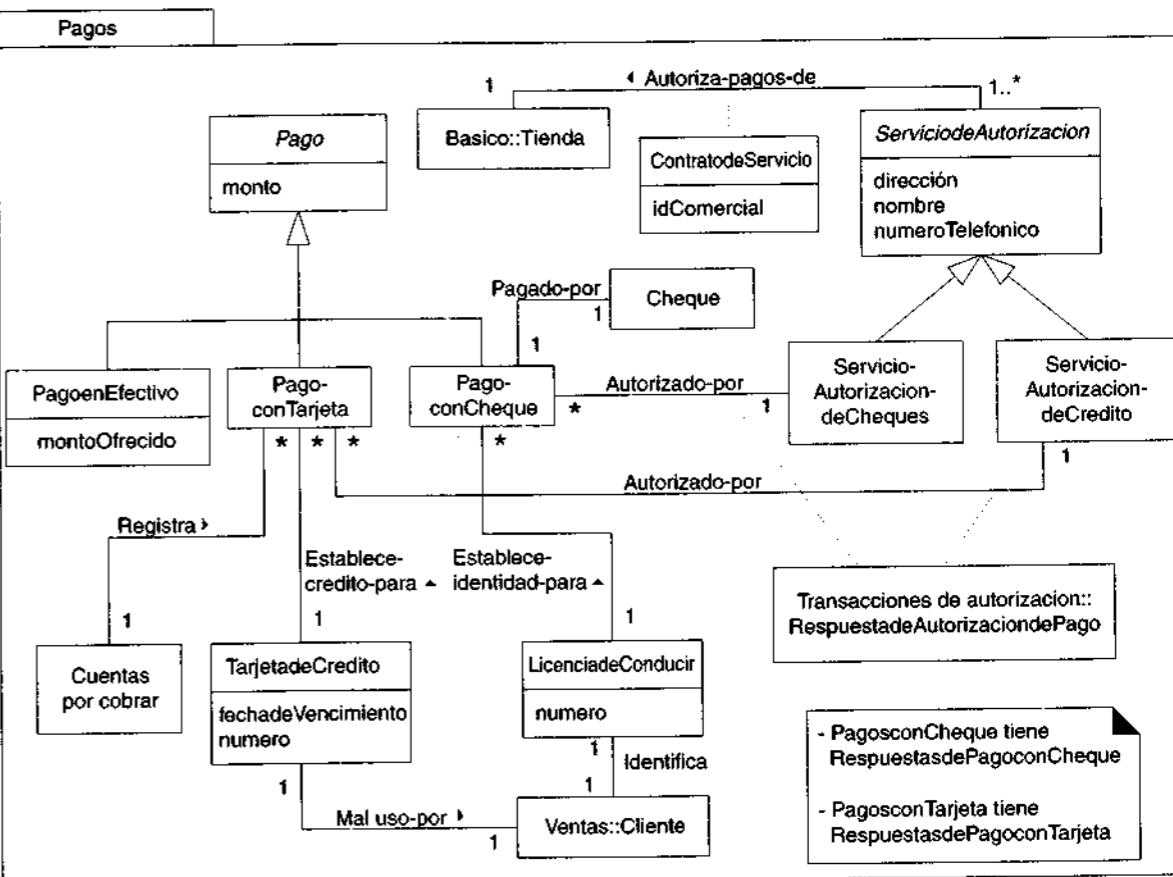


Figura 31.3 Paquete de pagos.

31.5 Productos

A excepción de la agregación compuesta, este paquete no contiene nuevos conceptos ni asociaciones propias de la iteración 2 (figura 31.4).

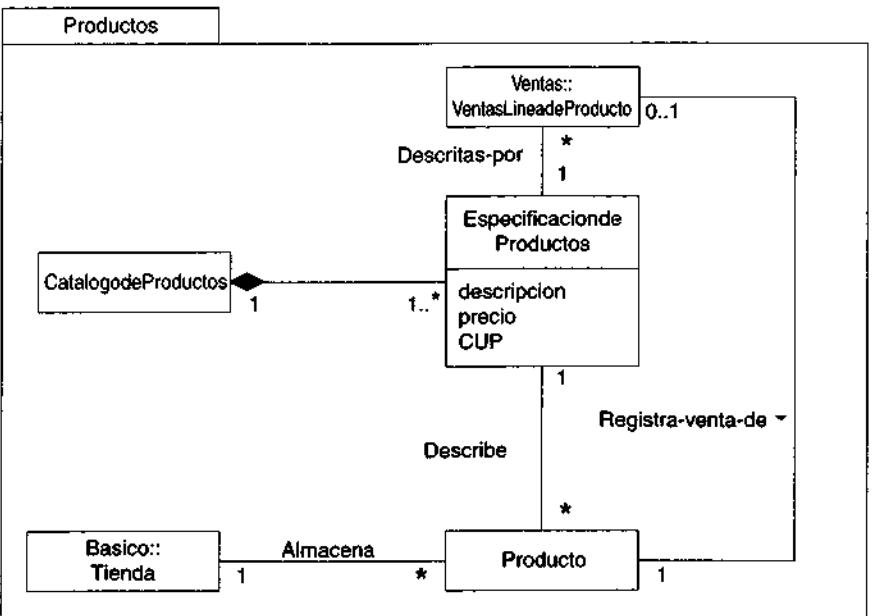


Figura 31.4 Paquete de productos.

31.6 Ventas

Con excepción de la agregación compuesta y los atributos derivados, este paquete no incluye nuevos conceptos ni asociaciones propias de la iteración 2 (figura 31.5).

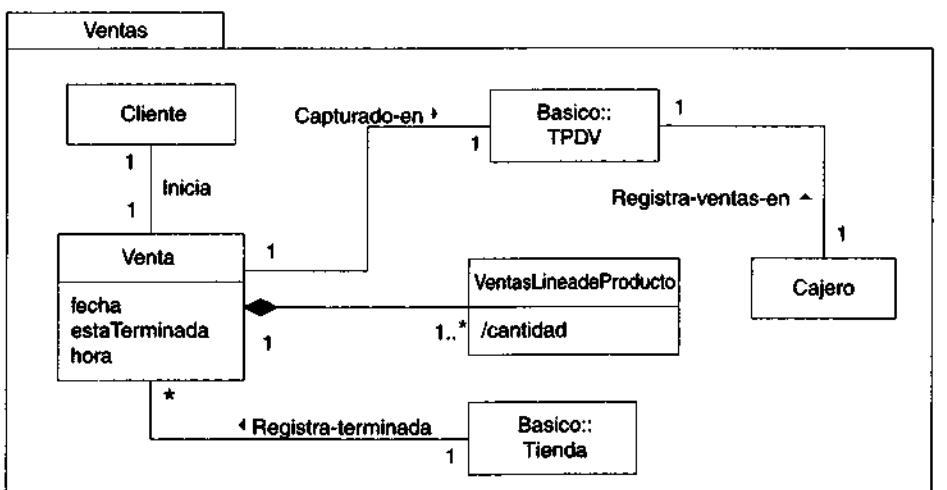


Figura 31.5 Paquete de ventas.

31.7 Transacciones de autorización

Aunque se recomienda asignar nombres significativos a las asociaciones, en algunas circunstancias tal vez ello no sea necesario, especialmente si para la audiencia es evidente el propósito de la asociación. Un caso muy ilustrativo es el constituido por las asociaciones entre los pagos y las transacciones correspondientes. No se especificaron sus nombres, pues supongo que la audiencia al leer el diagrama de tipos de la figura 31.6 entenderá que las transacciones se refieren al pago; si hubiera incorporado sus nombres, simplemente habría atiborrado el diagrama.

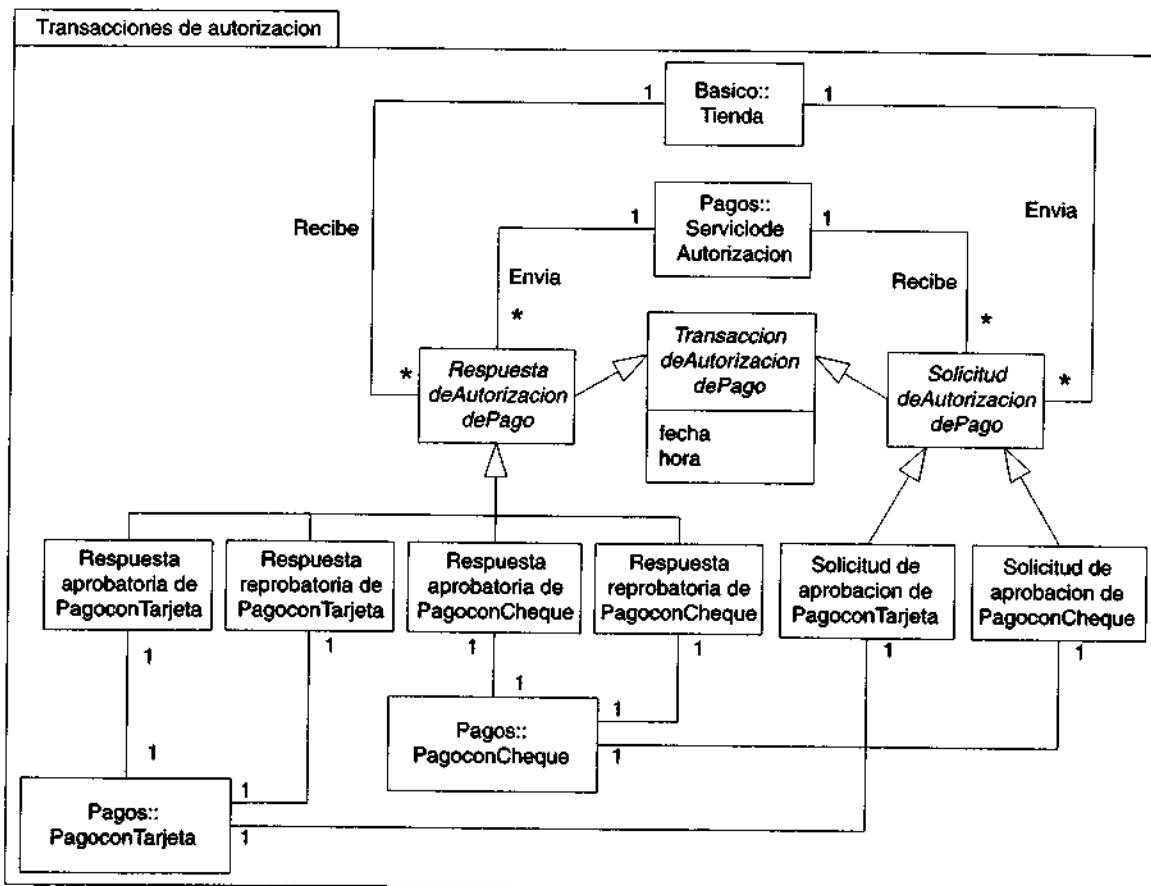


Figura 31.6 Paquete de transacciones de autorización

¿Resulta demasiado detallado el diagrama anterior porque contiene demasiadas especializaciones? No podemos contestar esta pregunta de manera tajante. El criterio definitivo es la utilidad. Aunque no es incorrecto, ¿de veras contribuye de alguna manera a conocer mejor el dominio? La respuesta a esta pregunta debería influir el número de especializaciones que incorporaremos al modelo conceptual.

COMPORTAMIENTO DE LOS SISTEMAS

Objetivos

- Definir los diagramas de secuencia del sistema y los contratos de su operación en el segundo ciclo de desarrollo.

32.1 Diagramas de secuencia del sistema

En el segundo ciclo de desarrollo, los diagramas de secuencia del sistema (DSS) han de soportar la ramificación de los tipos de pago que se producen en ella. Se agrupan las secuencias comunes en diagramas independientes, como se observa en los siguientes ejemplos.

32.1.1 Inicio común de Comprar Productos

El diagrama de secuencia del sistema de la parte inicial del caso de uso abarca los eventos *introducirProducto* y *terminarVenta*; estos elementos son comunes sin importar el método de pago.

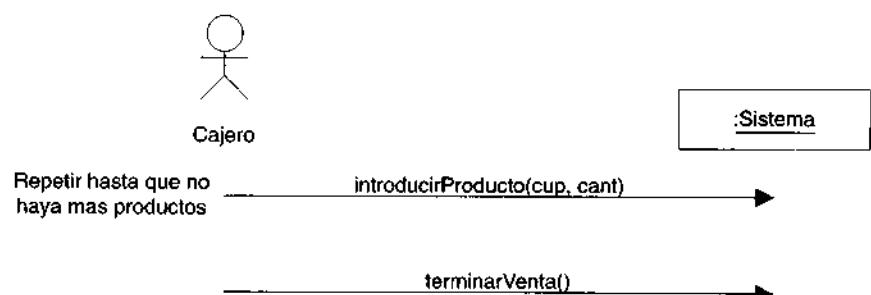


Figura 32.1 Inicio común de los diagramas de secuencia del sistema.

32.1.2 Pago con tarjeta

Este diagrama de secuencia comienza tras un inicio común, cuando se recurre al pago con tarjeta.

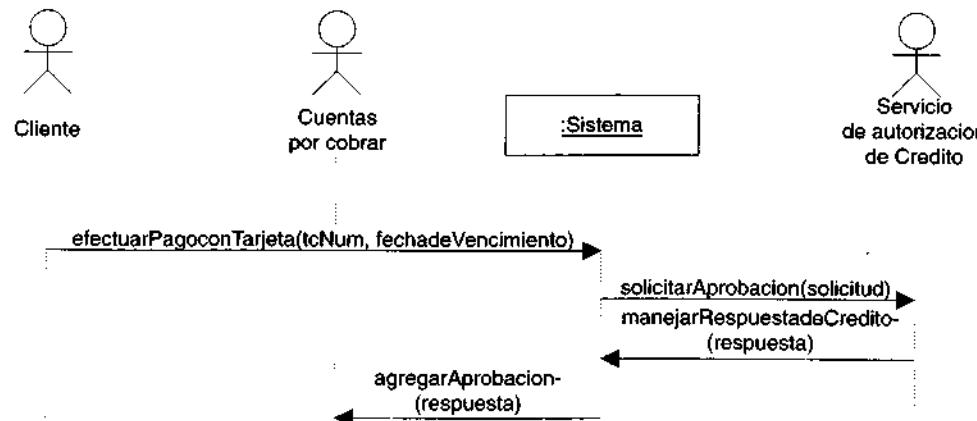


Figura 32.2 Diagramas de secuencia del sistema del pago con tarjeta de crédito.

32.1.3 Pago con cheque

Este diagrama de secuencia empieza tras un inicio común, cuando se recurre al pago con cheque.

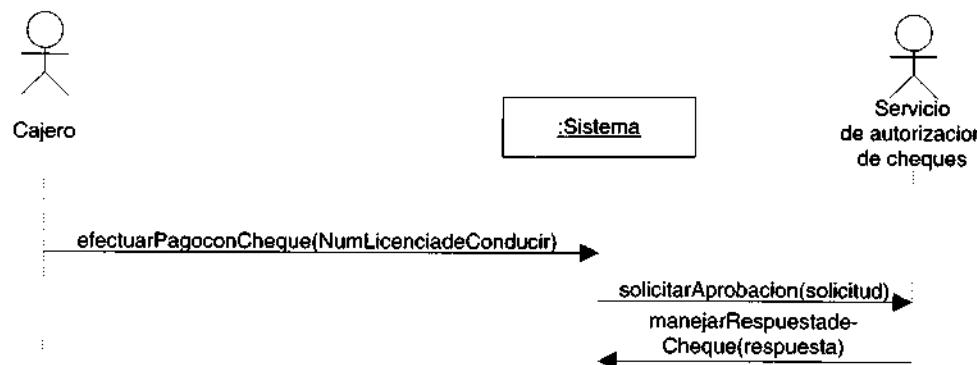


Figura 32.3 Diagramas de secuencia del sistema del pago con cheque.

32.2 Nuevos eventos del sistema

En este ciclo de desarrollo, los nuevos eventos del sistema de la terminal punto de venta son:

- efectuarPagoconTarjeta
- manejarRespuestadeCredito
- efectuarPagoconCheque
- manejarRespuestaconCheque

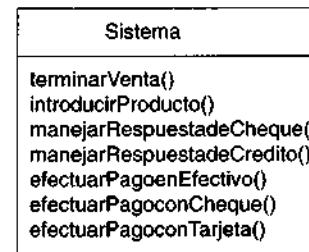
32.2.1 Cambio de nombre de EfectuarPago

En la primera iteración, el evento y la operación del sistema del pago en efectivo eran *EfectuarPago*. Ahora que los pagos son de distintos tipos, le cambiaremos el nombre y lo llamaremos:

- efectuarPagoenEfectivo

32.3 Contratos

Una operación del sistema se define para cada uno de sus eventos.



Hay que escribir contratos para las nuevas operaciones del sistema. Se trata de la mejor conjectura posible respecto a los cambios de estado que ocurrirán en respuesta a un evento del sistema, pero aun así han de considerarse tentativos: durante la fase de diseño probablemente resulten útiles pero incompletos.

En las siguientes secciones describiremos los contratos de operación para la aplicación del punto de venta.

Contrato	
Nombre:	efectuarPagoconTarjeta (tcNum : número fechadeVencimiento : fecha).
Responsabilidades:	Crear y solicitar autorización de un pago con tarjeta de crédito.
Tipo o clase:	Sistema (tipo).
Referencias cruzadas:	
Notas:	Es necesario transformar la solicitud en un registro plano.

Salida: Una solicitud de pago con tarjeta se envía al servicio de autorización de crédito.

Precondiciones: Se terminó la venta actual.

Poscondiciones:

- Se creó un *PagoconTarjeta pgo*.
- Se asoció *pgo* con la *Venta* actual.
- Se creó una *TarjetadeCredito tc*; *tc.numero* = *tcNum*, *tc.fechaVencimiento* = *fechadeVencimiento*.
- Se asoció *tc* a *pgo*.
- Se creó una *SolicitudPagoconTarjeta spt*.
- Se asoció *pgo* a *spt*.
- Se asoció *spt* a *ServiciodeAutorizaciondeCredito*.

Contrato

Nombre: *manejarResultadodeCredito*
(*respuesta* : *RespuestaPagoconTarjeta*).

Responsabilidades: Contestar la respuesta del servicio de autorización de crédito. Si la respuesta es aprobatoria, se concluye la venta y se registra el pago en cuentas por cobrar.

Tipo o clase: Sistema (tipo).

Referencias cruzadas:

Notas: *respuesta* es en realidad un registro que ha de ser transformado en una *RespuestaAprobatoriadePagoconTarjeta* o en una *RespuestaReprobatoriadePagoconTarjeta*.

Salida: Si se aprueba la solicitud, la respuesta aprobatoria se envía a cuentas por cobrar.

Precondiciones: La solicitud de pago con tarjeta se envió al servicio de autorización de crédito.

Poscondiciones:

■ Si la respuesta fue aprobatoria:

- Se creó una *aprobación RespuestaAprobatoriadePagoconTarjeta*.
- Se asoció *aprobación* a *CuentasporeCobrar*.
- Se asoció la *Venta* a la *Tienda* para incorporarla al registro histórico de ventas terminadas.

■ En caso contrario, si la respuesta fue reprobatoria:

- Se creó una *reprobatoria RespuestareprobatoriadePagoconTarjeta*.

Contrato

Nombre: *efectuarPagoconCheque*
(*NumLicenciadeConducir* : *numero*).

Responsabilidades: Crear y solicitar autorización de pago con cheque.

Tipo o clase: Sistema (tipo).

Referencias cruzadas:

Notas: Es necesario transformar la solicitud en un registro plano.

Salida:

Precondiciones: La venta actual está terminada.

Poscondiciones:

■ Se creó un *pgo PagoconCheque*.

■ Se asoció *pgo* a la *Venta* actual.

■ Se creó una *lc LicenciadeConducir* y *lc.número* = *numLicenciadeConducir*.

■ Se asoció *lc* a *pgo*.

■ Se creó una *spch SolicituddePagoconCheque*.

■ Se asoció *pgo* a *spch*.

■ Se asoció *spch* a *ServiciodeAutorizaciondeCheques*.

Contrato	
Nombre:	manejarRespuestadeCheque (respuesta : RespuestaPag .conCheque).
Responsabilidades:	Contestar la respuesta de autorización del servicio de autorización de cheques. Si la respuesta es aprobatoria, se concluye la venta.
Tipo o clase:	Sistema (tipo).
Referencias cruzadas:	
Notas:	respuesta es en realidad un registro que es necesario transformar en una <i>RespuestaAprobatoriadePagoconCheque</i> o en una <i>RespuestaReprobatoriadePagoconCheque</i> .
Salida:	
Precondiciones:	La solicitud de pago con cheque se envió al servicio de autorización de cheques.
Poscondiciones:	
<ul style="list-style-type: none"> ■ Si la respuesta fue aprobatoria: <ul style="list-style-type: none"> □ Se creó una <i>aprobación RespuestaAprobatoriadePagoconCheque</i>. □ Se asoció la <i>Venta</i> a la <i>Tienda</i>, para incorporarla al registro histórico de ventas terminadas. ■ En caso contrario, si la respuesta fue negativa: <ul style="list-style-type: none"> □ Se creó una <i>reprobatoria respuestaReprobatoriadePagoconCheque</i>. 	

MODELADO DEL COMPORTAMIENTO EN LOS DIAGRAMAS DE ESTADO

Objetivos

- Elaborar diagramas de estado para los conceptos y los casos de uso.

33.1 Introducción

El lenguaje UML cuenta con una notación para los diagramas de estado que describen gráficamente los eventos y los estados de los objetos. En este capítulo expondremos las características más importantes de la notación, pero hay otras que no mencionaremos. Ponemos de relieve la utilización de los diagramas de estado para indicar los eventos del sistema en los casos de uso, aunque bien podrían aplicarse a cualquier tipo.

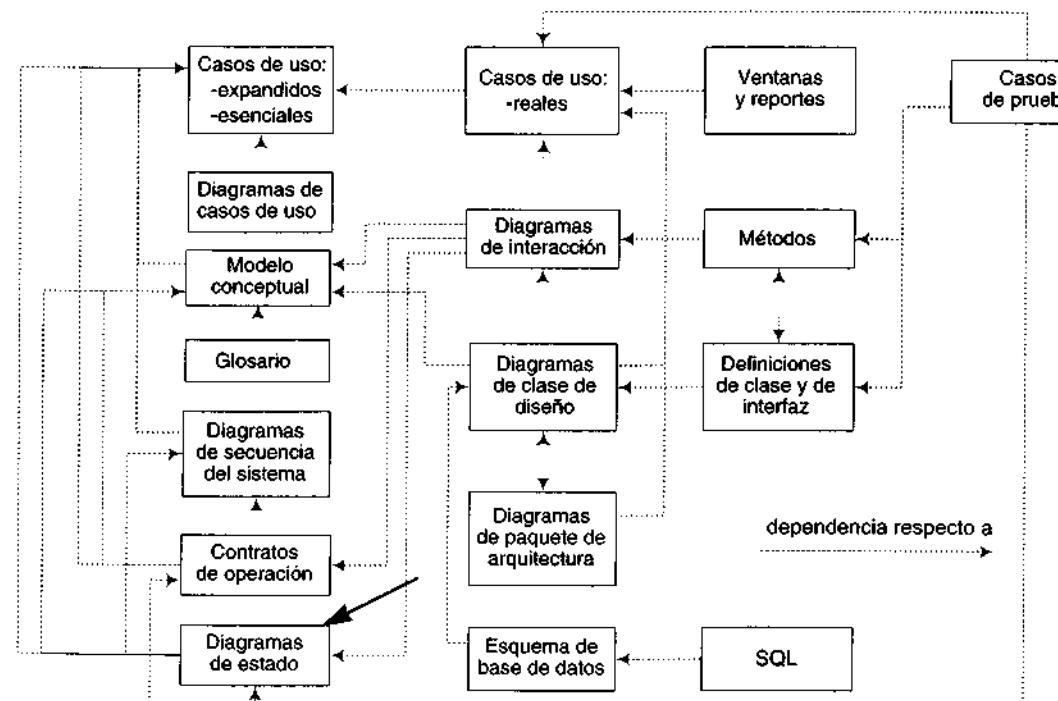
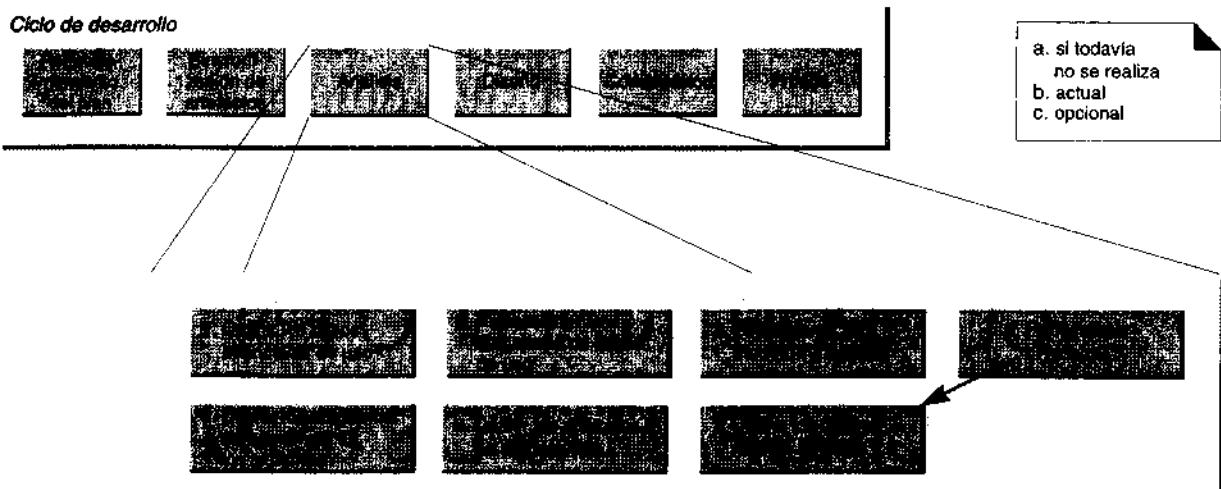
33.2 Eventos, estados y transiciones

Un **evento** es un acontecimiento importante o digno de señalar. Por ejemplo:

- levantar el auricular telefónico.

Un **estado** es la condición de un objeto en un momento determinado: el tiempo que transcurre entre eventos. Por ejemplo:

- un teléfono se encuentra en estado “ocioso” una vez que el auricular es puesto en su sitio y mientras no lo levantemos.



Dependencias de los artefactos durante la fase de construcción.

La **transición** es una relación entre dos estados; indica que, cuando ocurre un evento, el objeto pasa del estado anterior al siguiente. Por ejemplo:

- cuando ocurre el evento “levantar el auricular”, el teléfono realiza la transición del estado “ocioso” al estado “activo”.

33.3 Diagramas de estado

Un diagrama de estado del UML (figura 33.1) describe visualmente los estados y eventos más interesantes de un objeto, así como su comportamiento ante un evento. Las transiciones se muestran con flechas que llevan el nombre del evento respectivo. Los estados se colocan en óvalos. Se acostumbra incluir un seudooestado inicial que cumple automáticamente la transición a otro estado en el momento de crear una instancia.

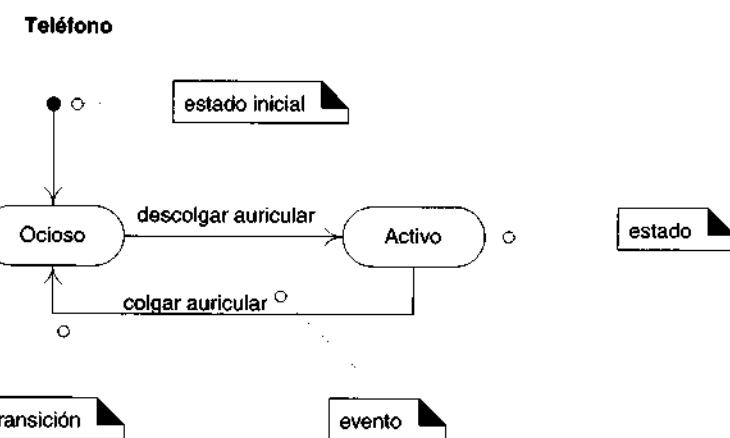


Figura 33.1 Diagrama de estado para un teléfono.

Un diagrama de estado presenta el ciclo de vida de un objeto: los eventos que le ocurren, sus transiciones y los estados que median entre esos eventos. No es necesario que muestre todos los eventos posibles; si tiene lugar un evento que no está representado en el diagrama, se excluye siempre y cuando no sea relevante en el diagrama de estado. Podemos, pues, crear un diagrama que describa el ciclo de vida de un objeto en niveles arbitrariamente simples o complejos, según las necesidades del momento.

33.3.1 Áreas de un diagrama de estado

Un diagrama de estado puede aplicarse a varios elementos del UML, a saber:

- clases de software
- tipos (conceptos)
- casos de uso

Podemos representar un “sistema” entero con un tipo, concepto o conjunto de sistemas en un dominio del problema que abarque los sistemas distribuidos; de ahí la posibilidad de que cuente con su propio diagrama de estado.

33.4 Diagramas de estado para los casos de uso

Una aplicación útil de este tipo de diagramas consiste en describir la secuencia permitida de los eventos externos del sistema que reconoce y maneja un sistema dentro del contexto de un caso de uso. Por ejemplo:

- En el caso *Comprar Productos* de la aplicación del punto de venta, no está permitido efectuar la operación *efectuarPagoconTarjeta* mientras no haya ocurrido el evento *terminarVenta*.
- En el caso *Procesar Documento* con un procesador de palabras, no está permitido efectuar la operación *Guardar en Archivo* mientras no haya ocurrido el evento *Archivo-Nuevo* o *Archivo-Abierto*.

Un diagrama de estado que describe los eventos globales del sistema y su secuencia en un caso de uso es una clase de **diagrama de estado para casos de uso**. El diagrama de la figura 33.2 constituye una versión simplificada de los eventos del sistema del caso de uso *Comprar Productos* en la aplicación punto de venta. Indica que no es correcto generar un evento *efectuarPago* si el evento *terminarVenta* no ha hecho que el sistema realice la transición al estado *EnEsperadelPago*.

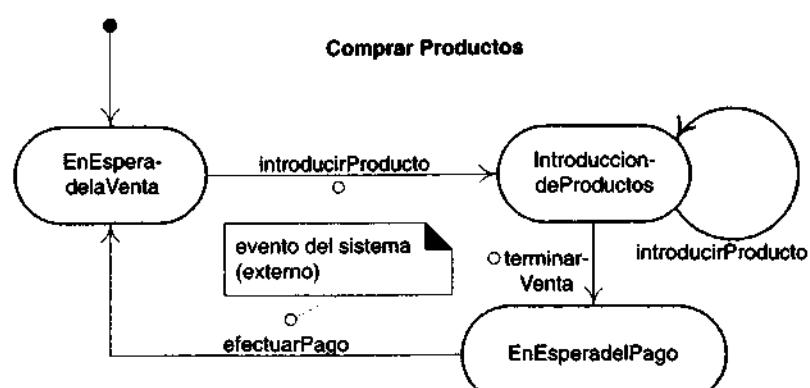


Figura 33.2 Diagrama de estado del caso de uso *Comprar Productos*.

33.4.1 Utilidad de los diagramas de estado para los casos de uso

El número de eventos de un sistema y su orden correcto en el caso *Comprar productos* tienen (hasta hoy) apariencia trivial; por eso, no parece obligatorio emplear un diagrama de estado para señalar la secuencia correcta. Pero en un caso complejo con multitud de eventos de sistema —cuando se utiliza un procesador de palabras, por ejemplo— conviene recurrir a un diagrama de estado que describa el orden legal de los eventos externos.

He aquí como hacerlo: durante las fases de diseño e implementación, hay que preparar e implementar un diseño que garantice que no ocurran eventos fuera de la secuencia, pues de lo contrario podría sobrevenir una condición de error. Por ejemplo, no debe permitirse que TPDV reciba un pago cuando aún no se concluye una venta; habrá que escribir un código que garantice esto.

Con un conjunto de diagramas de estado para los casos de uso, el diseñador podrá desarrollar metódicamente un diseño que garantice el orden correcto de eventos del sistema. A continuación se enumeran algunas soluciones posibles del diseño:

- pruebas condicionales rigurosamente codificadas para los eventos fuera de orden
- uso del patrón *Estado* que se explica en un capítulo posterior
- inactivación de elementos (widgets) en las ventanas activas para prohibir los eventos ilegales (método recomendable)
- un intérprete de una máquina de estados, el cual ejecute una tabla de estados que represente un diagrama de estado para los casos de uso

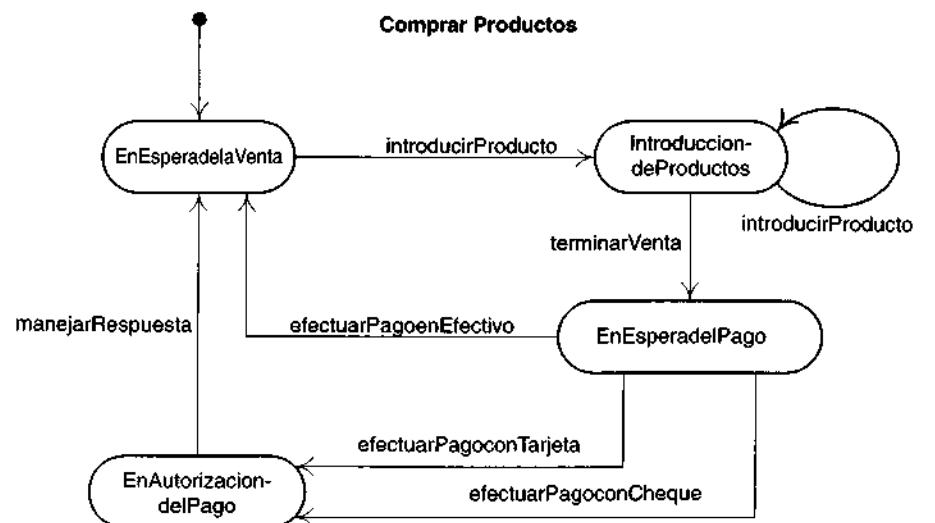
En un dominio con muchos eventos del sistema, la concisión y el rigor al emplear los diagramas de estado de casos de uso le ayudan al diseñador a cerciorarse de que no omite nada.

33.5 Diagramas de estado del sistema

Una variante de este tipo de diagramas es el **diagrama de estado del sistema**, que muestra las transiciones de los eventos del sistema a lo largo de todos los casos de uso. Es la unión de todos los diagramas de estado y resulta útil mientras el número total de los eventos sistémicos sea lo bastante pequeño para que sigan siendo manejables.

33.6 Diagramas de estado de los casos de uso para la aplicación del punto de venta

33.6.1 Comprar Productos



33.6.2 Iniciar

El diagrama de estado para el caso de uso *Iniciar* no es interesante y por ello no lo incluimos. En la práctica, este tipo de diagramas no es necesario si no hay un ordenamiento significativo de los eventos del sistema.

33.7 Tipos que requieren diagramas de estado

Además de los diagramas de estado para los casos de uso o el sistema global, podemos crear diagramas prácticamente para cualquier tipo o clase.

33.7.1 Tipos independientes y dependientes del estado

Si un objeto siempre reacciona igual ante un evento, se le considera **independiente del estado** (o sin modo) respecto a dicho evento. Por ejemplo, si un objeto recibe un mensaje y si su método de respuesta siempre hace lo mismo, el método generalmente no tendrá una lógica condicional. El objeto es independiente del estado respecto a ese mensaje. Si un tipo siempre reacciona de la misma manera ante todos los eventos de interés,

será un tipo **independiente del estado**. En cambio, los **tipos dependientes del estado** reaccionan de manera distinta ante los eventos según el estado de estos últimos.

Prepare diagramas de estado para los tipos dependientes del estado cuyo comportamiento sea complejo.

En términos generales, los sistemas de información de las empresas tienen pocos tipos interesantes dependientes del estado. Por el contrario, los dominios de control de procesos y de telecomunicaciones a menudo tienen muchos objetos dependientes del estado.

33.7.2 Tipos y clases comunes dependientes del estado

Damos a continuación una lista de clases o tipos comunes que suelen depender del estado y para los cuales posiblemente convenga elaborar un diagrama de estado:

- **Casos de uso (procesos).**
 - Visto como tipo, el caso de uso *Comprar Productos* reacciona de modo distinto ante el evento *terminarVenta* según que una venta esté realizándose o no.
- **Sistemas.** Un tipo que representa la aplicación o sistema íntegros.
 - El “*sistema del punto-de-venta*”.
- **Ventanas.**
 - La acción de Editar-Pegar sólo es válida cuando hay algo en el “portapapeles” para pegar.
- **Coordinadores de aplicaciones.**
 - “Applets” en Java.
 - “Documents” en el esquema de aplicación Document-View de MFC C++ de Microsoft.
 - “ApplicationsModels” en el esquema de aplicación de Smalltalk de VisualWorks.
 - “Visual Parts” en Smalltalk de VisualAge.
- **Controladores.** Una clase que no administra aplicaciones ni ventanas y que se encarga de manejar los eventos del sistema, como se explicó en el patrón Controlador de GRASP.
 - La clase *TPDV*, que maneja los eventos *introducirProducto* y *terminarVenta* del sistema.
- **Transacciones.** La forma en que una transacción reacciona ante un evento a menudo depende de su estado actual a lo largo de todo su ciclo de vida.
 - Si una *Venta* recibió un mensaje *hacerLineadeProducto* después del evento *terminarVenta*, debería presentar una condición de error o ser omitida.

- **Dispositivos.**
 - TPDV, televisor, lámpara, módem; reaccionan de modo distinto ante un evento particular, según su estado actual.
- **Mutadores.** Tipos que cambian su tipo o su papel.
 - Una persona que cambia papeles: de civil a militar.

33.8 Otros diagramas de estado para la aplicación Punto de Venta

Los controladores (en el sentido de GRASP) son los candidatos comunes entre las sugerencias anteriores de los tipos que pueden depender del estado. Por ejemplo, el controlador que hemos usado hasta ahora en nuestra aplicación ha sido la clase *TPDV*. Nótese que esta clase, como se aprecia en la figura 33.3, tiene una prueba condicional en el método *IntroducirProducto* para determinar si se trata de una nueva venta, creando una instancia *Venta* si lo es.

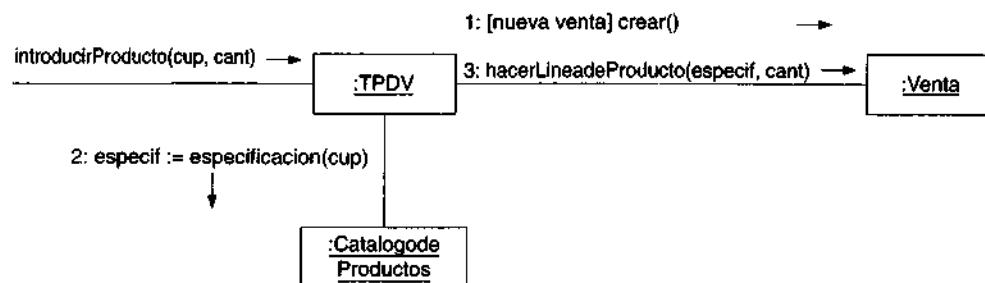


Figura 33.3 El método *introducirProducto* tiene un paso condicional, lo cual significa que *TPDV* depende del estado.

Una instancia *TPDV* reacciona de manera diferente ante el mensaje *IntroducirProducto* (que surge a consecuencia del evento *IntroducirProducto* del sistema), según su estado. En la figura 33.4 se incluye un diagrama de estado que explica gráficamente esto.

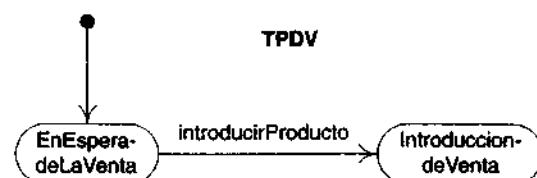


Figura 33.4 Diagrama de estado de *TPDV*.

33.9 Ejemplificación de eventos externos y de intervalos

33.9.1 Tipos de eventos

Conviene clasificar los eventos de la siguiente manera:

- **Evento externo:** llamado también evento del sistema, se debe a algún factor (un actor, por ejemplo) situado fuera de la frontera del sistema. Los diagramas de secuencia describen visualmente esta clase de eventos. Algunos eventos externos muy importantes provocan la invocación de las operaciones del sistema para responder a ellos.
 - Cuando un cajero oprime el botón "introducir producto" en la terminal punto de venta, significa que ha ocurrido un evento externo.
- **Evento interno:** se debe a un factor interno de nuestro sistema. Por lo que respecta al software, un evento interno tiene lugar cuando se invoca una operación a través de un mensaje o señal que partió de otro objeto interno. Los mensajes en los diagramas de colaboración indican la presencia de eventos internos.
 - Cuando una *Venta* recibe un mensaje *hacerLineadeProducto*, significa que ha ocurrido un evento interno.
- **Evento temporal:** se debe a la ocurrencia de una fecha u hora específicas o bien al transcurso del tiempo. En lo tocante al software, un reloj de tiempo real o de tiempo simulado son los que conducen este tipo de eventos.
 - Suponga que, una vez realizada una operación *terminarVenta*, debe realizarse una operación *efectuarPago* en un plazo de cinco minutos, pues de lo contrario se depurará automáticamente la venta actual.

33.9.2 Diagramas de estado para eventos internos

Un diagrama de estado puede mostrar eventos *internos* que suelen representar los mensajes recibidos de otros objetos. Los diagramas de colaboración también contienen los mensajes y sus reacciones (a partir de otros mensajes); cabe preguntar entonces: ¿por qué utilizar un diagrama de estado para describir los eventos internos y el diseño de objetos? El enfoque del diseño orientado a objetos establece que los objetos colaboran a través de mensajes para llevar a cabo las tareas; este enfoque lo explica directamente el diagrama de colaboración de UML. Resulta un poco incongruente usar un diagrama de estado para mostrar un diseño de intercambio de mensajes y de interacción entre los objetos.¹

¹ El lector a quien le interesen publicaciones sobre el análisis y diseño orientados a objetos encontrará libros de texto y en revistas ejemplos de complejos diagramas de estado dedicados a los eventos internos y a la reacción del objeto ante ellos. En lo esencial, los diseñadores han remplazado el paradigma de la interacción y colaboración entre objetos a través de mensajes por el paradigma de objetos como máquinas de estado. Además se han servido de los diagramas de estado para diseñar el comportamiento de objetos en vez de recurrir a los diagramas de colaboración. Desde un punto de vista abstracto, ambas perspectivas son equivalentes.

Por eso tengo mis reservas y no me atrevo a recomendar, para el diseño creativo orientado a objetos, los diagramas de estado que muestren eventos internos.¹ Sin embargo, puede servirnos para resumir los resultados de un diseño, una vez que lo hayamos concluido.

En cambio, como señalamos antes al explicar los diagramas de estado para los casos de uso, un diagrama de estados para eventos *externos* puede ser una herramienta sucinta y de mucha utilidad.

Prefiera los diagramas de estado para describir eventos externos y temporales, así como su reacción frente a ellos, en vez de servirse de ellos para diseñar el comportamiento de los objetos a partir de eventos internos.

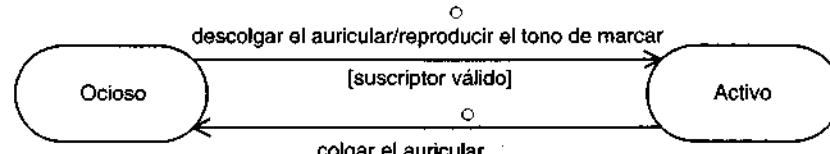
33.10 Notación complementaria de los diagramas de estado

La notación de los diagramas de estado en el lenguaje UML contiene muchas características que no utilizaremos en esta introducción. Tres de las más importantes son:

- acciones de transición
- condiciones protectoras de las transiciones
- estados anidados

33.10.1 Acciones y protecciones de las transiciones

acción de transición



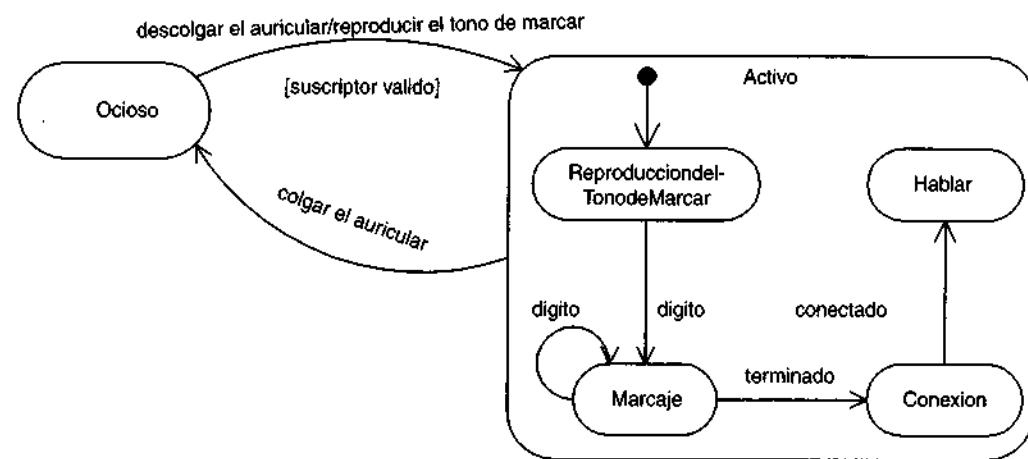
condición de protección

Una transición puede hacer que una acción se active. En una implementación de software, esto puede representar la invocación de un método de clases del diagrama de estado.

¹ Los diagramas de estado sirven, entre otras cosas, para mostrar el diseño del objeto basándose en los eventos internos; esta aplicación es razonable cuando hay que producir un código con un generador de código que se basará en el diagrama de estado o cuando se use un intérprete de la máquina de estado para que ejecute el software.

Una transición también puede tener una protección condicional, o prueba booleana. Sólo la tomamos si pasa la prueba.

33.10.2 Estados anidados



**PARTE VII FASE
DE DISEÑO (2)**

GRASP: MÁS PATRONES PARA ASIGNAR RESPONSABILIDADES

Objetivos

- Aprender a aplicar el resto de los patrones GRASP.

34.1 GRASP: Patrones generales de software para asignar responsabilidades

En páginas anteriores estudiamos la aplicación del primero de los cinco patrones GRASP:

- Experto, Creador, Alta Cohesión, Bajo Acoplamiento, Controlador

Los últimos cuatro patrones GRASP son:

- Polimorfismo
- Fabricación Pura
- Indirección
- No Hables con Extraños

En este capítulo examinaremos el resto de los patrones, los cuales constituyen los principios fundamentales con que se asignan responsabilidades a los objetos. En el siguiente capítulo trataremos de otros patrones de mucha utilidad y luego los aplicaremos al desarrollo de la segunda iteración en la aplicación terminal punto de venta.

34.2 Polimorfismo

Solución Cuando por el tipo varían las alternativas o comportamientos afines, las responsabilidades del comportamiento se asignarán —mediante operaciones polimórficas— a los tipos en que el comportamiento presenta variantes.¹

Corolario: no realizar pruebas con el tipo de un objeto ni utilizar lógica condicional para plantear diversas alternativas basadas en el tipo.

Problema ¿Cómo manejar las alternativas basadas en el tipo? ¿De qué manera crear componentes de software conectables?

Alternativas basadas en el tipo. La variación condicional es un tema esencial en la programación. Si se diseña un programa mediante la lógica condicional if-then-else o con el estatuto case, habrá que modificar la lógica del case cuando surja una variante. Este procedimiento dificulta extender un programa con otras variantes, porque los cambios tienden a requerirse en varios lugares donde exista la lógica condicional.

Componentes de software conectables. Viendo los componentes en las relaciones cliente-servidor, ¿cómo podemos remplazar un componente servidor sin afectar al cliente?

Ejemplo En la aplicación del punto de venta, ¿quién debería encargarse de autorizar las diversas clases de pagos?

El comportamiento de autorización varía con el tipo de pago —en efectivo,² con tarjeta de crédito o con cheque—; por eso, conforme al polimorfismo deberíamos asignar esa responsabilidad a cada tipo de pago, implementado con una operación polimórfica *autorizar* (figura 34.1). Cada una de estas operaciones se *instrumentará* de modo diferente: *PagoconTarjeta* se comunicará con un servicio de autorización de crédito y así sucesivamente.

¹ El polimorfismo tiene varios sentidos. Dentro de este contexto significa “asignar el mismo nombre a servicios en varios objetos” [Coad95], cuando los servicios se parecen o están relacionados entre ellos. Los tipos de objetos suelen estar relacionados en una jerarquía con una superclase común, aunque ello no sea estrictamente necesario (sobre todo en lenguajes de ligado dinámico, como Smalltalk, o en los que dan soporte a interfaces, como Java).

² Algunas terminales situadas en el punto de venta están provistas de un dispositivo que autentifica los billetes, pues determinan si son falsos o no. Por ejemplo, ésta es una práctica muy generalizada en algunos países europeos.

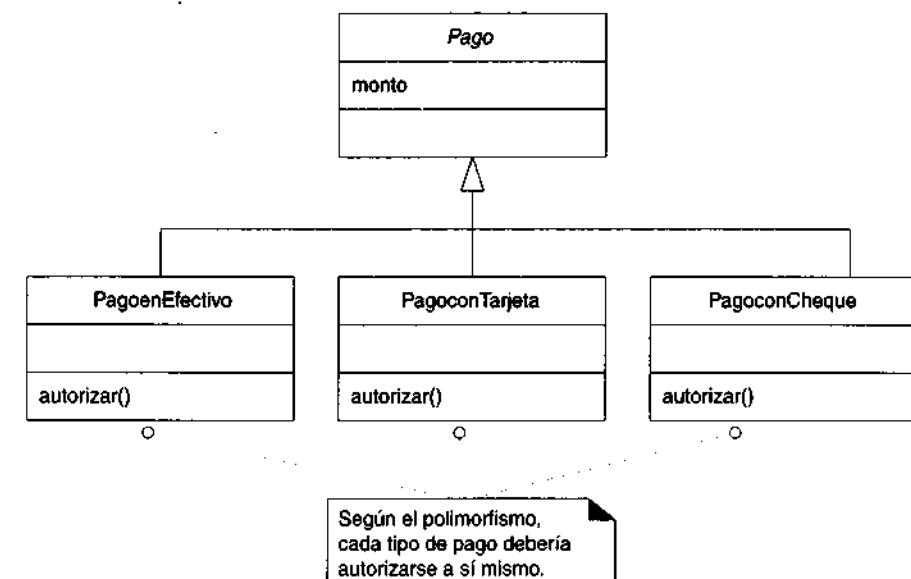


Figura 34.1 Polimorfismo en la autorización de pagos.

Explicación Como el patrón Experto, el uso del patrón Polimorfismo está acorde al espíritu del patrón “Lo hago yo mismo” [Coad95]. En vez de operar externamente sobre un pago para autorizarlo, el pago se autoriza a sí mismo; esto constituye la esencia de la orientación a objetos.

Si podemos caracterizar Experto como el patrón fundamental *táctico*, Polimorfismo será el más importante patrón *estratégico* en el diseño orientado a objetos. Es un principio fundamental en que se fundan las estrategias globales, o planes de ataque, al diseñar cómo organizar un sistema para que se encargue del trabajo. Un diseño basado en la asignación de responsabilidades mediante el polimorfismo puede ser extendido fácilmente para que realice nuevas variantes. Por ejemplo, agregar una nueva clase *PagoconTarjetaDebito* su operación polimórfica *autorizar* afectará poco el diseño actual dada la forma como se maneja la autorización.

Cuando vemos los objetos en las relaciones de cliente-servidor, los objetos del cliente requieren poca o nula modificación al introducir un nuevo objeto servidor, a condición de que el nuevo servidor soporte las operaciones polimórficas que espera el cliente.

Beneficios ■ Es fácil agregar las futuras extensiones que requieren las variaciones imprevistas.

También conocido como o semejante a “Lo hago yo mismo”, “Elección de mensaje”, “No pregantes ¿qué clase es?”

34.3 Fabricación Pura

Solución Asignar un conjunto altamente cohesivo de responsabilidades a una clase artificial que no representa nada en el dominio del problema: una cosa inventada para dar soporte a una alta cohesión, un bajo acoplamiento y reutilización.

Esa clase es una *fabricación* de la imaginación. En teoría, las responsabilidades que se asignan brindan soporte a una alta cohesión y a bajo acoplamiento, de modo que el diseño de la fabricación sea muy limpio, o *puro*. De ahí el nombre: fabricación pura.

Finalmente, una fabricación pura supone exigir algo, y esto lo hacemos cuando estamos desesperados.

Problema ¿A quién asignar la responsabilidad cuando uno está desesperado y no quiere violar los patrones Alta Cohesión y Bajo Acoplamiento?

Los diseños orientados a objetos se caracterizan por implementar como clases de software las representaciones de conceptos en el dominio de un problema del mundo real; por ejemplo, una clase *Venta* y *Cliente*. Pese a ello, se dan muchas situaciones donde el asignar responsabilidades exclusivamente a las clases del dominio origina problemas por una mala cohesión o acoplamiento o bien por un escaso potencial de reutilización.

Ejemplo Supongamos, por ejemplo, que se necesita soporte para guardar las instancias *Venta* en una base de datos relacional. En virtud del patrón Experto, en cierto modo se justifica asignar esta responsabilidad a la clase *Venta*. Pero reflexionemos sobre las siguientes implicaciones:

- La tarea requiere un número relativamente amplio de operaciones de soporte orientadas a la base de datos, ninguna de las cuales se relaciona con el concepto de vender; por tanto, la clase *Venta* reduce su cohesión.
- La clase *Venta* ha de ser acoplada a la interfaz de la base de datos relacional (interfaz que suele proporcionar el proveedor de las herramientas de desarrollo); de ahí que mejore su acoplamiento. Y éste ni siquiera se realiza con otro objeto del dominio, sino con una interfaz idiosincrásica de la base de datos.
- Guardar los objetos en una base de datos relacional es una tarea muy general en que debemos brindar soporte a muchas clases. Asignar estas responsabilidades a la clase *Venta* indica que habrá poca reutilización o mucha duplicación en otras clases que cumplen la misma función.

En conclusión, aunque en virtud del patrón Experto *Venta* es un candidato lógico para guardarse a sí misma en una base de datos, da origen a un diseño de baja cohesión, alto acoplamiento y bajo potencial de reutilización, precisamente el tipo de situación desesperada que exige inventar algo.

Una solución razonable consiste en crear una clase nueva que se encargue tan sólo de guardar los objetos en algún tipo de almacenamiento persistente: una base de datos

relacional; lo llamaremos *Agente de Almacenamiento Persistente*.¹ Esta clase es una Fabricación Pura, una mera abstracción de la imaginación.



Agente de Almacenamiento Persistente
guardar()

Sirve para resolver los siguientes problemas de diseño:

- La Venta conserva su buen diseño, con alta cohesión y bajo acoplamiento.
- La clase *Agente de Almacenamiento Persistente* es también relativamente cohesiva, pues su único propósito es guardar los objetos en un medio de almacenamiento persistente.
- La clase *Agente de Almacenamiento Persistente* es un objeto extremadamente genérico y reutilizable.

Crear una fabricación pura en este ejemplo es precisamente la situación donde se justifica su uso: hay que eliminar un diseño deficiente, con mala cohesión y acoplamiento y cambiar por un buen diseño que ofrezca un mayor potencial de reutilización.

Nótese que, igual que con los patrones GRASP, aquí se pone de relieve a quién deberán asignarse las responsabilidades. En nuestro ejemplo las transferimos de la clase *Venta* a la Fabricación Pura.

Explicación Para diseñar una Fabricación Pura debe buscarse ante todo un gran potencial de reutilización, asegurándose para ello de que sus responsabilidades sean pequeñas y cohesivas. Estas clases tienden a tener un conjunto de responsabilidades de *granularidad fina*.

Una fabricación pura suele partirse atendiendo a su funcionalidad y, por lo mismo, es una especie de objeto de función central.

Generalmente se considera que la fabricación es parte de la capa de servicios orientada a objetos de alto nivel en una arquitectura.

Muchos patrones actuales del diseño orientado a objetos constituyen ejemplos de Fabricación Pura: Adaptador, Observador, Visitante y otros más [GHJV95].

Beneficios

- Se brinda soporte a una Alta Cohesión porque las responsabilidades se dividen en una clase de granularidad fina que se centra exclusivamente en un conjunto muy específico de tareas afines.
- Puede aumentar el potencial de reutilización debido a la presencia de las clases de Fabricación Pura de granularidad fina, cuyas responsabilidades pueden utilizarse en otras aplicaciones.

Problemas posibles Puede perderse el espíritu de los buenos diseños orientados a objetos que se centran en los objetos y no en las funciones, pues las clases de Fabricación Pura casi siempre

¹ En un esquema de persistencia real, definitivamente se requerirá más de una clase de Fabricación Pura para elaborar un buen diseño.

se dividen atendiendo a su funcionalidad; dicho con otras palabras, se confeccionan clases destinadas a conjuntos de funciones. Si se abusa de ello, la creación de clases de Fabricación Pura, originará un diseño centrado en procesos o funciones que se implementa en un lenguaje orientado a objetos.

- Patrones relacionados**
- Bajo Acoplamiento.
 - Alta Cohesión.
 - Una Fabricación Pura normalmente asume las responsabilidades de la clase del dominio a la cual se le asignarían basándose en el patrón Experto.
 - Muchos patrones actuales de diseño orientados a objetos son ejemplos de Fabricación Pura: Adaptador, Observador, Visitante y otros más [GHJV95].

34.4 Indirección

Solución Se asigna la responsabilidad a un objeto intermedio para que medie entre otros componentes o servicios, y éstos no terminen directamente acoplados.

El intermediario crea una *indirección* entre el resto de los componentes o servicios.

Problema ¿A quién se asignarán las responsabilidades a fin de evitar el acoplamiento directo? ¿De qué manera se desacoplarán los objetos de modo que se obtenga un Bajo Acoplamiento y se conserve un alto potencial de reutilización?

Ejemplos Agente de Almacenamiento Persistente

El ejemplo de Fabricación Pura para desacoplar la *Venta* y los servicios de la base de datos relacional introduciendo la clase *Agente de Almacenamiento Persistente* es también un ejemplo de asignar responsabilidades para apoyar la Indirección. El *Agente de Almacenamiento Persistente* sirve de intermediario entre la *Venta* y la base de datos.

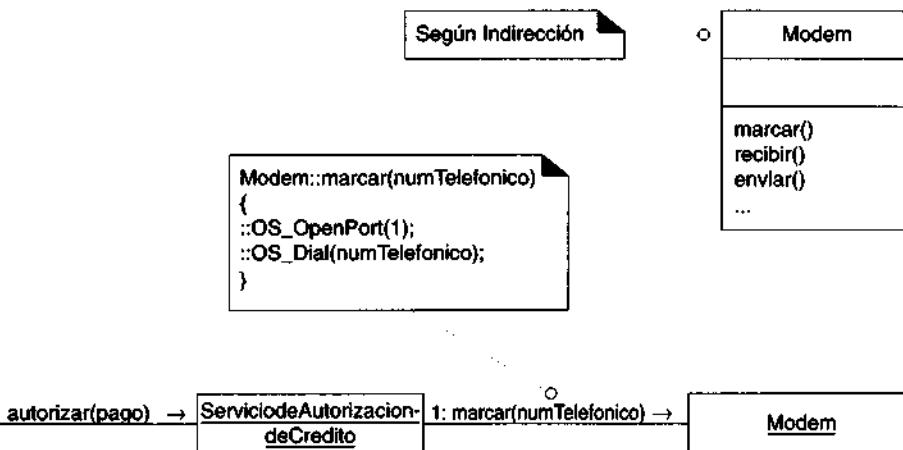
Módem

Suponga que:

- Una aplicación de terminal situada en el punto de venta necesita manipular un módem para transmitir solicitudes de pago con tarjeta de crédito.
- El sistema operativo tiene una llamada de función API de bajo nivel para hacer esto.
- Una clase denominada *Servicio de Autorización de Crédito* asume la responsabilidad de hablar con el módem.

Si el *Servicio de Autorización de Crédito* invoca directamente las llamadas de la función API de bajo nivel, estará estrechamente acoplado a las peculiaridades de la API del sistema operativo en cuestión. Requerirá modificación si la clase necesita ser trasladada a otro sistema operativo (para usarla en la misma aplicación o en otra).

Se agrega una clase intermedia *Modem* entre el *Servicio de Autorización de Crédito* y la API del módem. Es responsable de traducir los requerimientos abstractos del módem a la API y crear una Indirección entre el *Servicio de Autorización de Crédito* y la API del módem. (También se le da el nombre de agente [*proxy*] del dispositivo a una clase como *Modem* que representa a un dispositivo electromecánico y se conecta con él.)



34.4.1 Publicar-Suscribir u Observador

El patrón Publicar-Suscribir u Observador [GHJV95] también constituye un ejemplo del patrón Indirección. Los objetos se suscriben a eventos ante un *Administrador de Eventos*; otros publican eventos para el *Administrador de Eventos*, que los notifica a los suscriptores. A través de la indirección del *Administrador de Eventos* se desacoplan los editores y los suscriptores.

Explicación “La mayoría de los problemas de la computación puede resolverlos otro nivel de indirección”, reza un adagio muy acorde a los diseños orientados a objetos.¹

Del mismo modo que muchos patrones actuales de diseño son especializaciones de la Fabricación Pura, también muchos son especialización de Indirección; por ejemplo, Adaptador, Fachada y Observador [GHJV95]. Además, muchas clases de Fabricación Pura se generan a causa del patrón Indirección. El motivo de la Indirección casi siempre es el Bajo Acoplamiento; se agrega un intermediario con el fin de desacoplar otros componentes o servicios.

¹ Si es que algún adagio puede considerarse antiguo en las ciencias computacionales. No recuerdo ahora la fuente (Parnas, tal vez).

Beneficios ■ Bajo acoplamiento.

- Patrones** ■ Bajo Acoplamiento.
■ Mediador [GHJV95].
■ Muchos intermediarios de Indirección son situaciones de Fabricación Pura.

34.5 No Hables con Extraños

Solución Se asigna la responsabilidad a un objeto directo del cliente para que colabore con un objeto indirecto, de modo que el cliente no necesite saber nada del objeto indirecto.

El patrón, también conocido con el nombre de ley de Demeter [Lieberherr88], impone restricciones a los objetos a los cuales deberíamos enviar mensajes dentro de un método. El patrón establece que en un método, los mensajes sólo deberían ser enviados a los siguientes objetos:

1. El objeto *this* (o *self*).
2. Un parámetro del método.
3. Un atributo de *self*.
4. Un elemento de una colección que sea atributo de *self*.
5. Un objeto creado en el interior del método.

Con esto se busca no acoplar un cliente al conocimiento de objetos indirectos ni a las representaciones internas de objetos directos. Los objetos directos son "conocidos" del cliente, los objetos indirectos son "extraños", y un cliente debería tener sólo conocidos, no extraños.

Cumplir con las restricciones anteriores significa que los objetos directos pueden requerir nuevas operaciones que actúen como intermediarios, a fin de que el cliente pueda evitar "hablar con extraños".

Problema ¿A quién asignar las responsabilidades para evitar conocer la estructura de los objetos indirectos?

Si un objeto conoce las conexiones internas y las estructuras de otros, entonces presentará alto acoplamiento. Si un objeto cliente tiene que usar un servicio u obtener información a partir de un objeto indirecto, ¿cómo podrá hacerlo sin acoplarse al conocimiento de la estructura interna de su servidor directo o de los objetos indirectos?

Ejemplo En una aplicación del punto de venta, suponga que una instancia *TPDV* posee un atributo referente a una *Venta*, la cual cuenta con un atributo referente a un *Pago*.

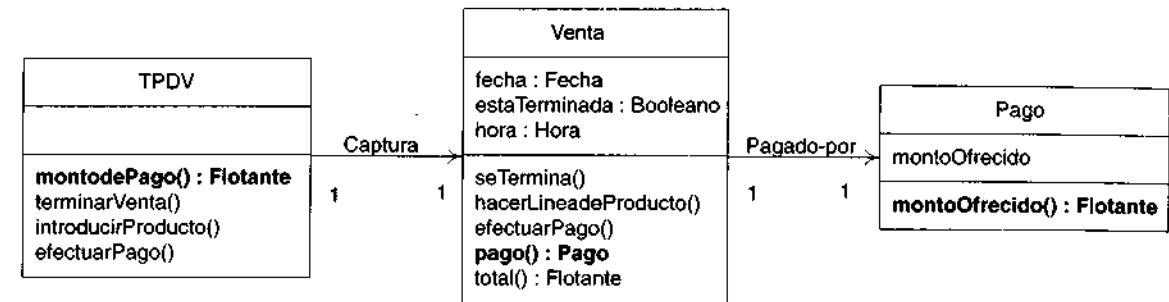


Figura 34.2 Diagrama parcial de clases del diseño.

Y además suponga que:

- Las instancias *TPDV* soportan la operación *montodePago*, que devuelve el actual monto ofrecido como pago.
- Las instancias *Venta* soportan la operación *pago*, la cual devuelve la instancia *Pago* asociada a la *Venta*.

Una forma de devolver el monto del pago es:

```

TPDV::MontodePago()
{
    pag: e_venta->Pago();
    // viola "No Hables con extraños"
    return pag->montoOfrecido();
}
  
```

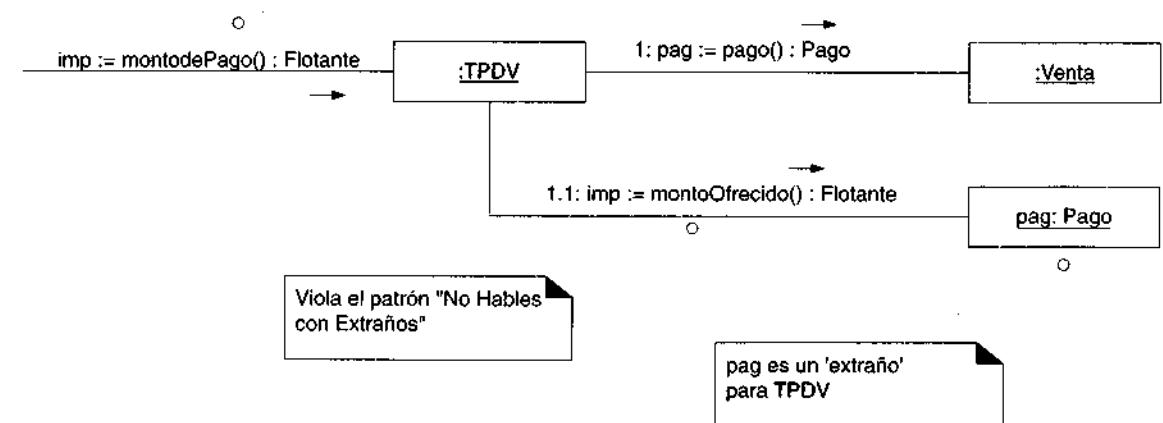


Figura 34.3 Violación del patrón "No Hables con Extraños".

La solución de la figura 34.3 constituye una violación del principio No Hables con Extraños, porque la instancia *TPDV* envía un mensaje a un objeto indirecto: el *Pago* no es uno de los cinco candidatos “conocidos”.

La solución, como lo indica el patrón, consiste en agregar la responsabilidad al objeto directo —la *Venta*, en este caso— para que devuelva a *TPDV* el monto del pago. A esto se le llama **promoción de la interfaz**, que es la solución general que da soporte al principio general. Por tanto, se agrega una operación *montodePago*, de modo que la instancia *TPDV* no tenga que hablar con un extraño (figura 34.4).

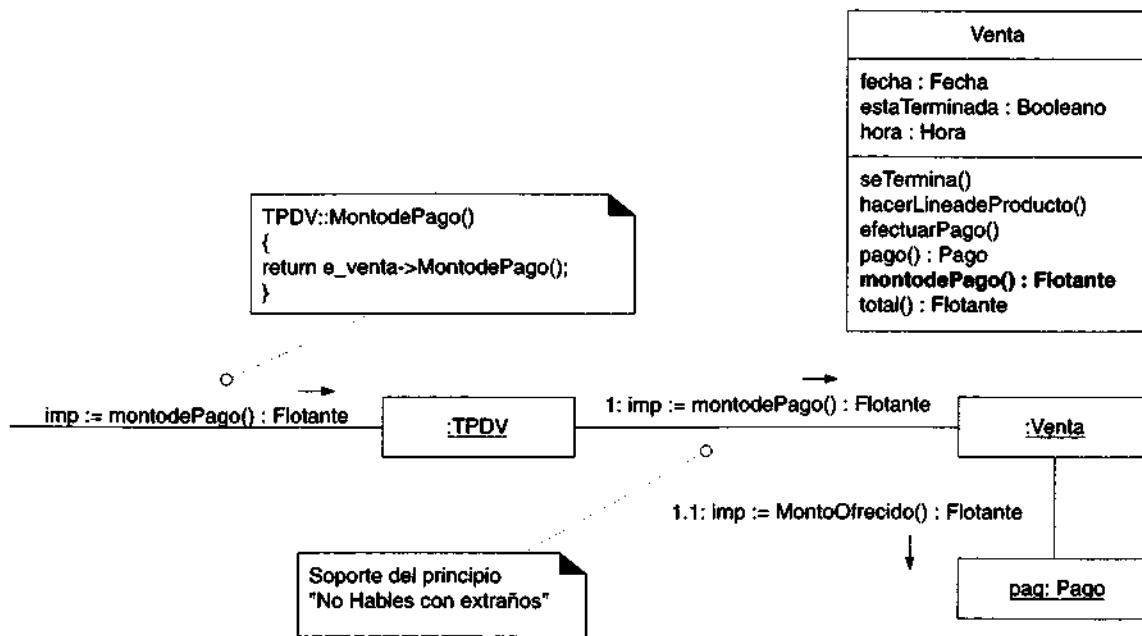


Figura 34.4 Soporte del principio “No Hables con Extraños”.

Explicación No Hables con Extraños se refiere a no obtener una visibilidad temporal frente a objetos indirectos, que son de conocimiento de otros objetos pero no del cliente. La desventaja de conseguir visibilidad ante extraños es la siguiente: la solución se acopla entonces a la estructura interna de otros objetos. Ello origina un alto acoplamiento, que hace el diseño menos robusto y más propenso a requerir un cambio si se alteran las relaciones estructurales indirectas.

Supongamos, por ejemplo, que se usa la solución de la figura 34.3 y que luego se modifica la definición de clase relativa a *Venta*, de modo que ya no conserve su referencia a *Pago*. En tal caso, el método *TPDV-montodePago* queda invalidado y requiere mantenimiento. En este ejemplo pequeño no parece existir un problema difícil, pero sí constituye un problema en un sistema con cientos —si no es que miles— de clases que son desarrolladas simultáneamente por muchos diseñadores de software. Un sistema se torna frágil cuando se diseña una solución que dependa del conocimiento de las relaciones estructurales indirectas.

En contraste con la solución de la figura 34.3, si se recurre a la de la figura 34.4, en los métodos *TPDV* no influirá ningún cambio de las representaciones ni de las conexiones internas de la *Venta*. Lo único que *TPDV* sabe es que una *Venta* puede contestar su pago total, pero sin saber cómo se consigue.

La violación de este principio es común en Smalltalk, lenguaje donde se cuenta con muchos métodos públicos para acceder a las conexiones estructurales privadas de un objeto. Un código como el siguiente aparece con mucha frecuencia:

```

miMetodoIncorrectodeObtenerPrecio
  "saltar de X a A a B a C y a D
  para regresar el precio de D"

mixDirecta obtenerA obtenerB obtenerC obtenerD
obtenerprecio
  
```

El código se desplaza por las conexiones estructurales saltando de un objeto al siguiente. Reflexione sobre la fragilidad del código: está estrechamente acoplado al conocimiento de muchas relaciones estructurales indirectas.

34.5.1 Violación de la ley

Lo primero que debemos conocer sobre las leyes del software es que están hechas para ser quebrantadas. Por eso, aunque la de No Hables con Extraños (ley de Demeter) parezca un excelente consejo, se dan situaciones donde conviene prescindir de ella. Una situación común donde esto ocurre se presenta cuando hay algún tipo de “agente” o “servidor de objetos” (generalmente una Fabricación Pura) encargado de devolver otros objetos, basándose para ello en una consulta mediante el valor de una clave. Se juzga aceptable obtener visibilidad ante un objeto *X* a través de un agente y luego enviarle directamente mensajes a *X*, aun cuando con ello se viole la ley No Hables con Extraños.

Beneficios ■ Bajo acoplamiento

Patrones ■ Bajo Acoplamiento, Indirección, Cadena de responsabilidades [GHJV95] conexos

DISEÑO CON MÁS PATRONES

Objetivos

- Aplicar los patrones GRASP y de la Pandilla de los Cuatro al diseñar la aplicación del punto de venta.

35.1 Introducción

En este capítulo vamos a estudiar la creación de diagramas de colaboración en la segunda iteración de la aplicación del punto de venta. Procuraremos ante todo mostrar la manera de aplicar los patrones GRASP y algunos de gran utilidad que han publicado otros autores. Demostraremos con ejemplos que el diseño orientado a objetos y la asignación de responsabilidades pueden explicarse y aprenderse utilizando los patrones: un vocabulario de principios y métodos susceptibles de combinarse para diseñar objetos.

Repetimos una vez más: la asignación de responsabilidades a los objetos y el diseño de colaboración entre ellos constituyen la esencia de este tipo de diseño. El trabajo de análisis que hemos efectuado hasta ahora se proponía dar soporte a la fase de diseño orientado a objetos.

Aunque los contratos para las operaciones de un sistema no se mencionan explícitamente aquí, las poscondiciones estipuladas en ellos se cumplen en las siguientes colaboraciones de los objetos. En la práctica, se revisan los contratos al ir desarrollando el diseño.

35.1.1 *Los patrones de la Pandilla de los Cuatro*

Los otros patrones que describiremos en este capítulo están tomados de *Design Patterns* [GHJV95], obra pionera en que se presentan 23 patrones de gran utilidad

durante la fase de diseño orientado a objetos. Dado que el libro fue escrito por cuatro autores, a los patrones se les conoce hoy con el nombre de "Gang of Four" (Pandilla de los Cuatro) o por la abreviatura en inglés "GoF".¹

En este capítulo vamos a ofrecer una introducción a algunos de ellos; en capítulos subsecuentes explicaremos otros, entre ellos el Método de Plantilla (Template Method) y Observador. Recomendamos al lector estudiar detenidamente la obra *Design Patterns*.

35.2 Estado (Pandilla de los Cuatro)

Como vimos en el capítulo anterior al hablar de los diagramas de estado, la reacción de la clase *TPDV* ante el mensaje *IntroducirProducto* dependerá de su estado. Esto se explica visualmente en los diagramas de estado de las figuras 35.1 y 35.2.

En vez de servirse de una prueba condicional, un método alterno consiste en utilizar el patrón Estado de la Pandilla de los Cuatro. En términos generales, nos sirve para eliminar las pruebas condicionales que provienen de las dependencias de estado.

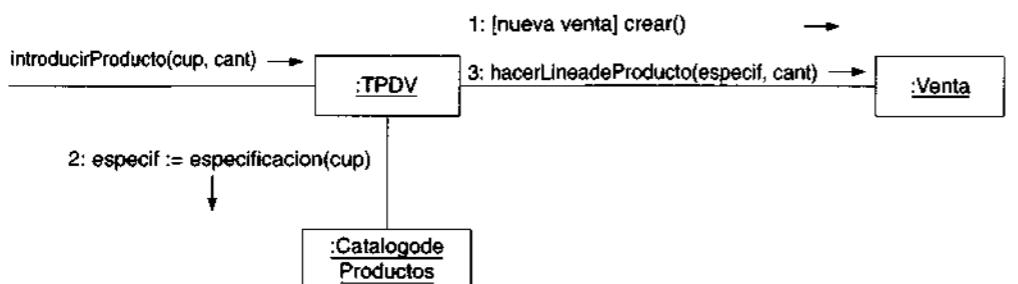


Figura 35.1 Diagrama de colaboración parcial de *introducirProducto* que muestra la reacción dependiente del modo.

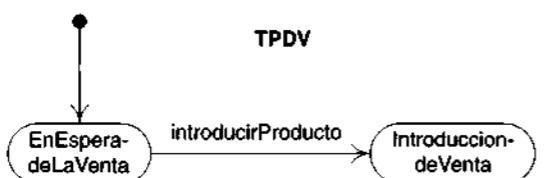


Figura 35.2 Diagrama de estado de *TPDV*.

Estado

Contexto/Problema

El comportamiento de un objeto depende de su estado. La lógica condicional no es adecuada a causa de su complejidad, escalabilidad o duplicación.

Solución

- Crear una clase para cada estado que influya en el comportamiento del objeto dependiente del estado (el objeto "contexto").
- Con base en el polimorfismo, asignar métodos a cada clase de estado para manejar el comportamiento del objeto contexto.
- Cuando el objeto contexto reciba un mensaje dependiente del estado, el mensaje será enviado al objeto estado.

Como se aprecia en la figura 35.3, se define una jerarquía de clases que representa los estados de la clase *TPDV*. Cada uno contiene un método *introducirProducto* porque su comportamiento depende del estado. Una instancia *TPDV* posee visibilidad de atributos ante su instancia actual *TPDV-Estado*.

Como se indica en la figura 35.4, cuando la instancia *TPDV* recibe el mensaje *IntroducirProducto*, lo envía a su estado y éste lo maneja mediante el patrón Polimorfismo.

Nótese que el objeto contexto se transmite junto con el objeto estado; con ello el contexto recupera la visibilidad de los parámetros y entonces puede recibir mensajes; por ejemplo, el que cambia su estado.

Comentario sobre la notación

En la figura 35.4 observe lo siguiente: se indica que el mensaje *introducirProducto* es enviado a una instancia de la superclase *TPDVEstado*, aun cuando en realidad se trata de la instancia de una de las subclases.

Para cada caso polimórfico de las subclases, se dibuja un nuevo diagrama de colaboración, comenzando con el mensaje polimórfico.

Recomendamos utilizar este estilo cuando se muestre un mensaje polimórfico.

¹ Con una alusión velada a la política de China.

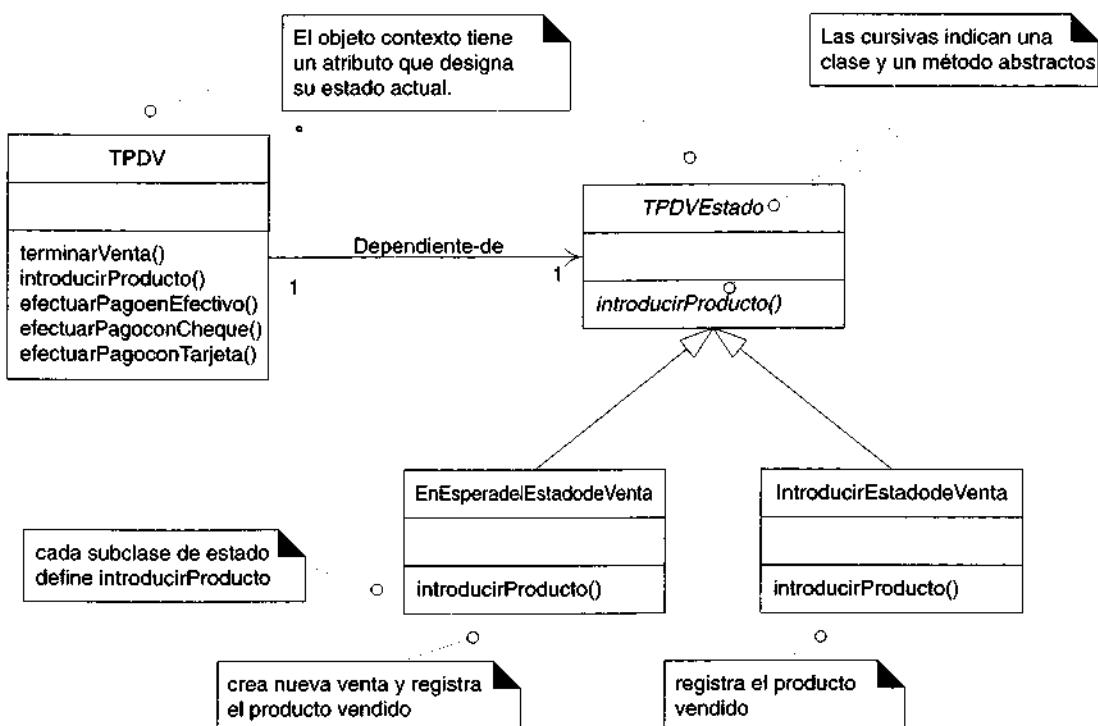


Figura 35.3 Clases de estado utilizadas en el patrón Estado.

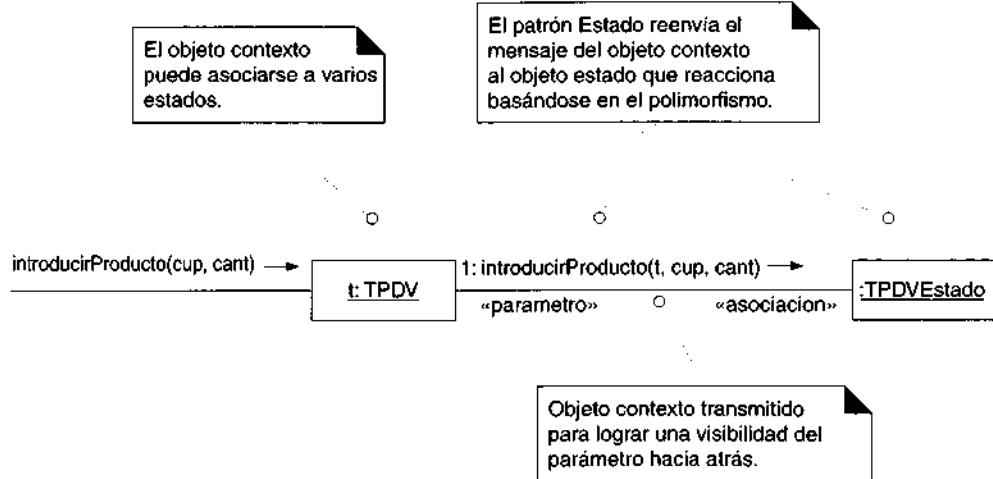


Figura 35.4 El patrón Estado comienza con el objeto contexto enviando un mensaje al objeto estado asociado.

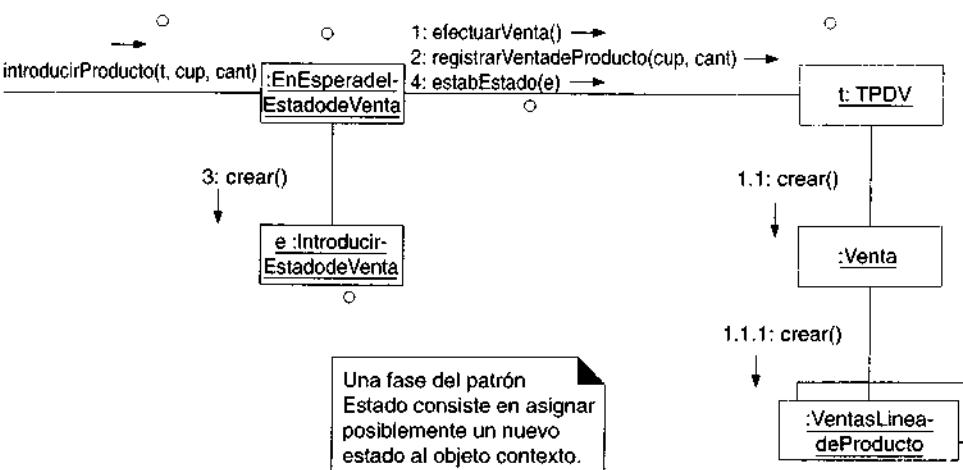
Las figuras 35.5 y 35.6 empiezan con el mensaje *introducirProducto* enviado a una instancia de *EnEsperadelEstadodeVenta* e *IntroducirEstadodeVenta*, respectivamente.

En la figura 35.5 el método *introducirProducto* inicia la creación de una nueva venta antes de registrar el artículo vendido. Además —y esto es una parte común del patrón Estado—, el estado actual le asigna un nuevo estado al objeto contexto.

En nuestro ejemplo, se creó la instancia de un nuevo estado, pero también se acostumbra emplear el patrón **Singleton** de la Pandilla de los Cuatro, que describiremos más adelante, para asignar una instancia de estado Singleton en vez de crearla. Otro patrón afín que puede usarse con los objetos Estado es **Flyweight** de la Pandilla de los Cuatro, que se parece a **Singleton**. Es un patrón de gran utilidad cuando podemos compartir una instancia entre muchos objetos, porque la instancia no alberga estado alguno sino que sólo posee comportamiento. Por ejemplo, las instancias *TPDVEstado* no conservan información; el objeto contexto se transmite como parámetro cuando se necesita. En este caso, en el sistema no se requiere más que una instancia de las clases concretas (*EnEsperadelEstadodeVenta* y otras).

El patrón Estado envía un mensaje del objeto contexto a un objeto estado, el cual reacciona basándose en patrón Polimorfismo. El objeto estado puede dar la vuelta y volver a colaborar con el objeto contexto.

El objeto contexto puede asociarse con varios estados.

Figura 35.5 Un caso polimórfico del patrón estado: *EnEsperadelEstadodeVenta*.

La figura 35.6 muestra el caso polimórfico de *introducirProducto* para *IntroducirEstadodeVenta*. En este caso no es necesario crear una nueva venta, sino que basta registrar el artículo vendido. Adviértase que esta vez el estado no cambia; un método de estado no causará necesariamente un cambio de estado.



Figura 35.6 Un segundo caso polimórfico del patrón Estado: IntroducirEstadoVenta.

Finalmente, el diagrama de colaboración `registrarVentadeProductos` se muestra en la figura 35.7.

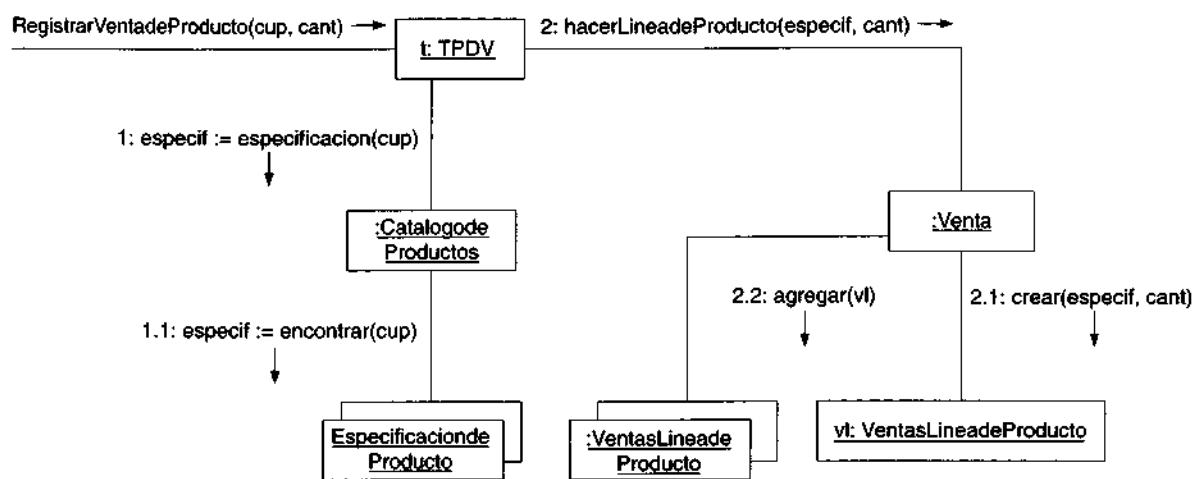


Figura 35.7 El método `registrarVentadeProducto` de la clase `TPDV`.

35.2.1 Estado: conclusión

El patrón Estado es muy útil cuando el comportamiento de un objeto depende de su estado. Con él se elimina lógica condicional en los métodos del objeto contexto y se obtiene un mecanismo elegante para extender el comportamiento de dicho objeto sin modificarlo.

Si un sistema presenta muchos estados, este patrón puede ser idóneo por la proliferación de clases. Otra alternativa consiste en definir un intérprete de la máquina de estado que funcione contra un grupo de reglas de transición de estados.

35.3 Polimorfismo (GRASP)

Como vimos en el capítulo anterior al tratar de los patrones GRASP, el hecho de que un pago se autorice a sí mismo corresponde al espíritu de los diseños orientados a objetos: el patrón “Lo hago yo mismo” [Coad95]. Además, como existen muchos tipos de pago, se requiere el patrón Polimorfismo para que cada tipo de pago se autorice a sí mismo.

Por tanto, como se observa en la figura 35.8, cada subclase `Pago` cuenta con su propio método `Autorizar`.

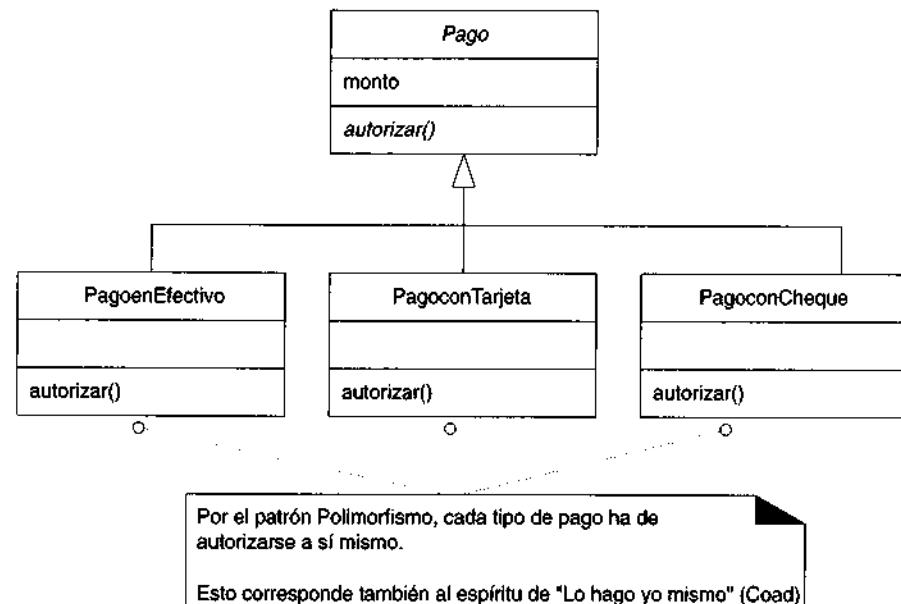


Figura 35.8 Jerarquía de pagos con múltiples métodos `Autorizar`.

Por ejemplo, como se muestra en las figuras 35.9 y 35.10, una `Venta` instancia un `PagoconTarjeta` o `PagoconCheque` y le pide que se autorice ella misma.

Nótese también la creación de los objetos de software `TarjetadeCredito`, `LicenciadeConducir` y `Cheque`. Un primer impulso sería registrar simplemente en sus respectivas clases de pago la información que aportan y eliminar tales clases de granularidad fina. Sin embargo, por lo regular una estrategia más provechosa consiste en usarlas; a menudo terminan ofreciendo un comportamiento de gran utilidad y siendo muy reutilizables. Por ejemplo, `TarjetadeCredito` es un Experto natural al indicarnos su tipo de institución de crédito (Visa, MasterCard y otros más). Este comportamiento resultará necesario en nuestra aplicación.

35.4 Singleton

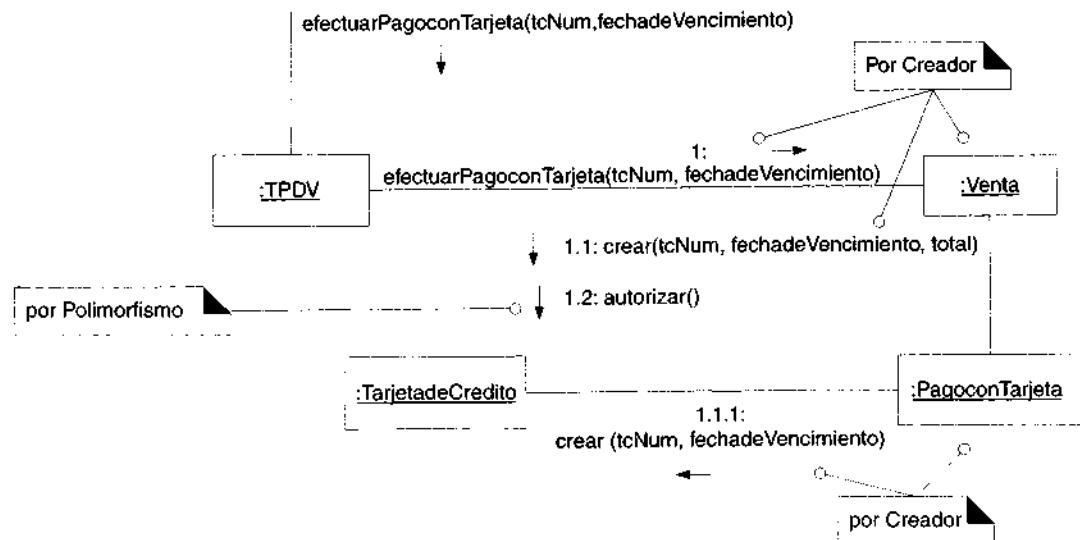


Figura 35.9 Creación de un `PagoconTarjeta`.

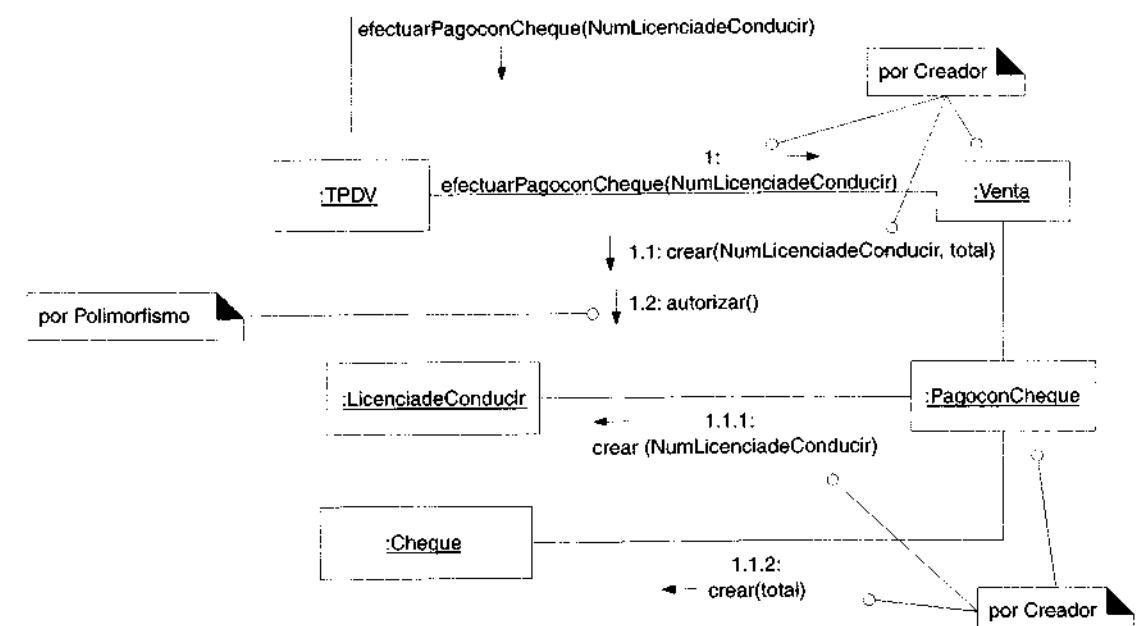


Figura 35.10 Creación de un `PagoconCheque`.

Cuando un `PagoconTarjeta` recibe un mensaje `Autorizar`, necesita enviar un mensaje a la `Tienda` para determinar con cuál servicio de autorización de crédito se comunicará (Bank of Foo para Visa, Bank of Bar para MasterCard, etc.). ¿Por qué preguntarle a `Tienda`? Porque es un Experto natural en la información concerniente a los servicios de autorización.

Pero surge aquí un problema de visibilidad: la instancia recién creada `PagoconTarjeta` no tiene acceso a la instancia `Tienda`. Una solución consiste en transmitirla hacia abajo (como parámetro) desde `TPDV` (que posee la visibilidad de atributo ante ella) y asignarla a un atributo de `PagoconTarjeta`, para que también tenga visibilidad de atributo frente a ella. Lo anterior es aceptable pero poco práctico; otra opción la constituye el patrón Singleton.

A veces conviene soportar la visibilidad global o un solo punto de acceso a una instancia individual de una clase y no a alguna otra forma de visibilidad. Y resulta que esto ocurre en el caso de la instancia `Tienda`.

Singleton

Contexto/Problema

Se permite exactamente una instancia de una clase: se trata de un "singleton" (solitario). Los objetos necesitan un solo punto de acceso.

Solución

Definir un método de clase o una función que no sea miembro (en C++) y que devuelva el "singleton".

Por ejemplo, como se aprecia en la figura 35.11, suponga que el patrón Singleton se necesita en una `Tienda`. Una solución consiste en definir un atributo de clase `instancia` en la clase `Tienda` que se refiera a una instancia `Tienda` y en definir un método de clase `Instancia()` que retorne el atributo de la clase.¹

Nótese que, en el lenguaje UML los miembros de una clase se indican con el símbolo "\$".

Como se muestra en la figura 35.12, con esta interfaz `PagoconTarjeta` pide a la clase `Tienda` visibilidad ante su única instancia y luego ante la instancia de `ServiciodeAutorizaciondeCredito` basándose en una `TarjetadeCredito`.

Puesto que la visibilidad ante las clases tiene un ámbito (relativamente) global, todo objeto puede llamar directamente el método clase que devuelva el elemento considerado Singleton.

¹ Datos y métodos estáticos en el lenguaje Java.

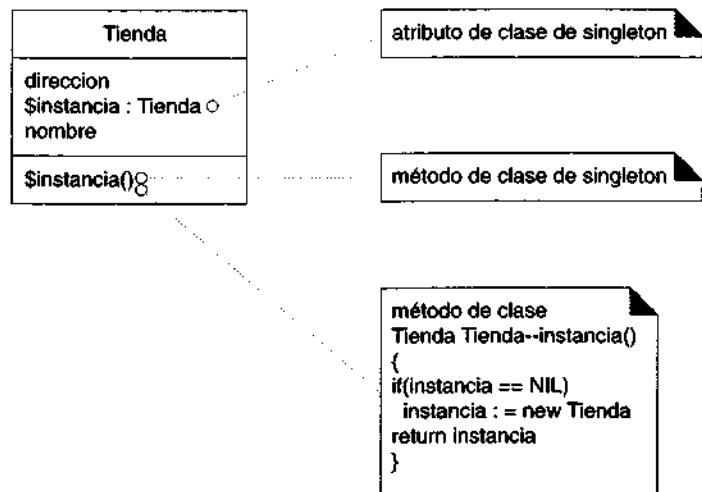


Figura 35.11 Soporte del patrón Singleton en la clase Tienda.

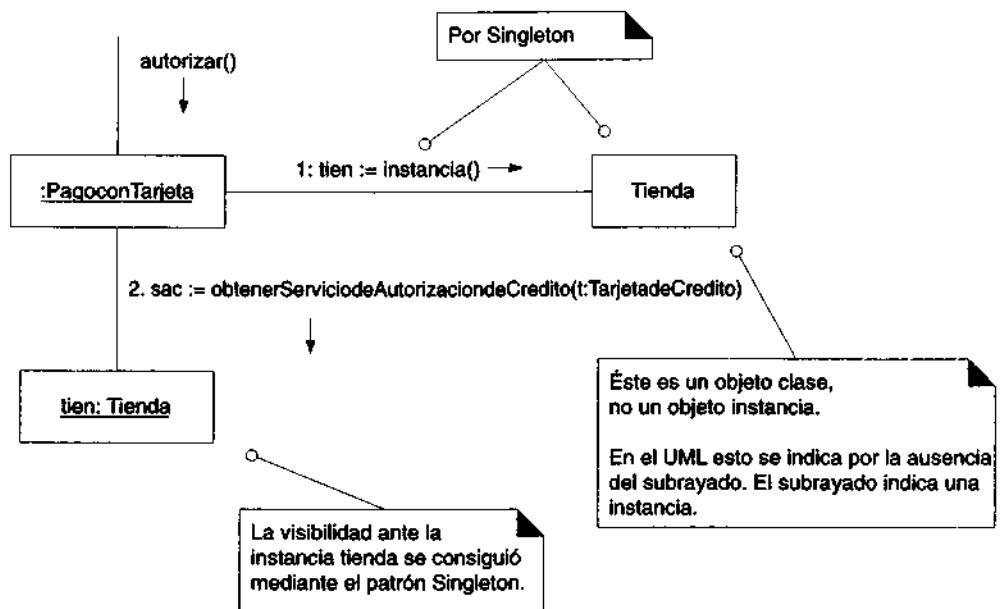


Figura 35.12 Uso del patrón Singleton.

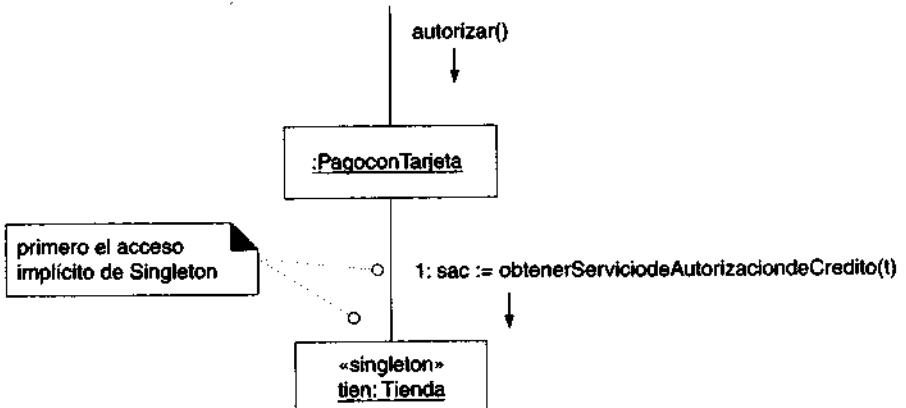
35.4.1 Código muestra

En un lenguaje de programación orientado a objetos, para emplear el patrón Singleton hay que enviar un mensaje a la clase y obtener visibilidad ante la instancia, la cual podrá entonces enviar mensajes.

C++	<code>Tienda::instancia() ->obtenerSAC(una-Tarjeta);</code>
Java	<code>Tienda.instancia().obtenerSAC(una-Tarjeta);</code>
Smalltalk	<code>Tienda instancia obtenerSAC: una-Tarjeta.</code>

35.4.2 Abreviatura del patrón Singleton en el UML

Una notación de UML que implica —pero que no necesariamente muestra— el acceso de singleton consiste en estereotipar la instancia con «singleton».

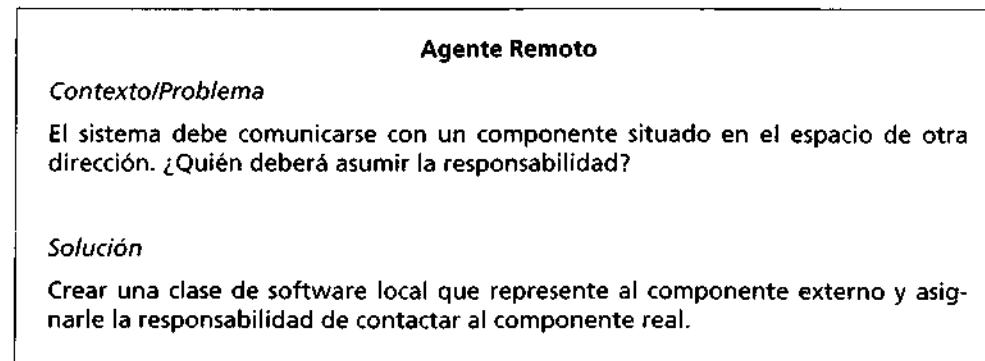


35.4.3 Implementación

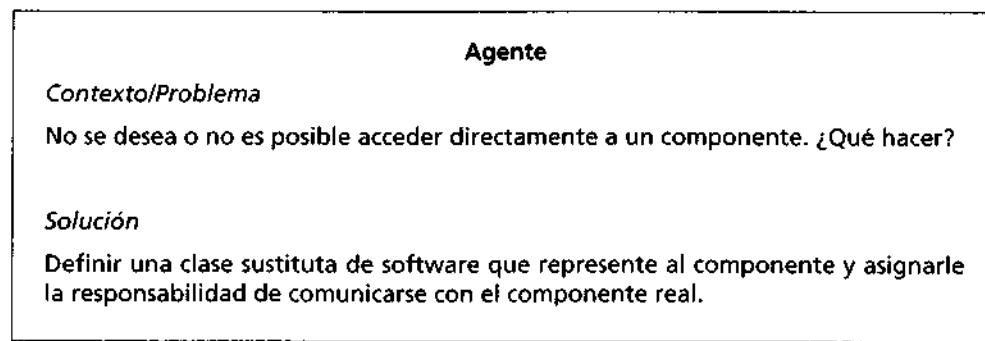
- Si se requiere seguridad continua, el método Singleton necesita alguna clase de protección.
- Puntos alternos de acceso:
 - un método de clase en una clase “utilería”
 - (C++) una función que no es miembro

35.5 Agente Remoto y Agente (Pandilla de los Cuatro)

En la sección anterior aludimos a la necesidad de acceder a una instancia de *Servicio de Autorización de Crédito* sin explicar por qué. Recordemos que el sistema debe comunicarse con un servicio externo. En tal situación, el patrón Agente Remoto de la Pandilla de los Cuatro sugiere crear una clase de software local que represente el servicio externo y asignarle la responsabilidad de contactar el servicio real. Por tanto, podríamos usar un objeto local *Servicio de Autorización de Crédito* para hablar con el mundo externo.



El Agente Remoto es un caso especial del patrón general Agente (Proxy) de la Pandilla de los Cuatro y sugiere utilizar un sustituto del componente (objeto, servidor, controlador de dispositivo, DLL, etc.) en algunos contextos.



En nuestro sistema hemos de determinar cuál servicio de autorización emplear y luego, con base en el patrón Agente Remoto, pedirle que se comunique con el servicio real.

Como se aprecia en la figura 35.13, puesto que una *Tienda* es un Experto natural en el conocimiento de sus servicios, pago le pedirá uno de ellos. La clave del servicio corresponde al tipo de compañía de crédito; por eso se transmite *Tarjeta de Crédito* y se consulta su tipo (*Tarjeta de Crédito* es un patrón Experto en el conocimiento de su tipo). Y entonces podrá accederse al servicio apropiado.

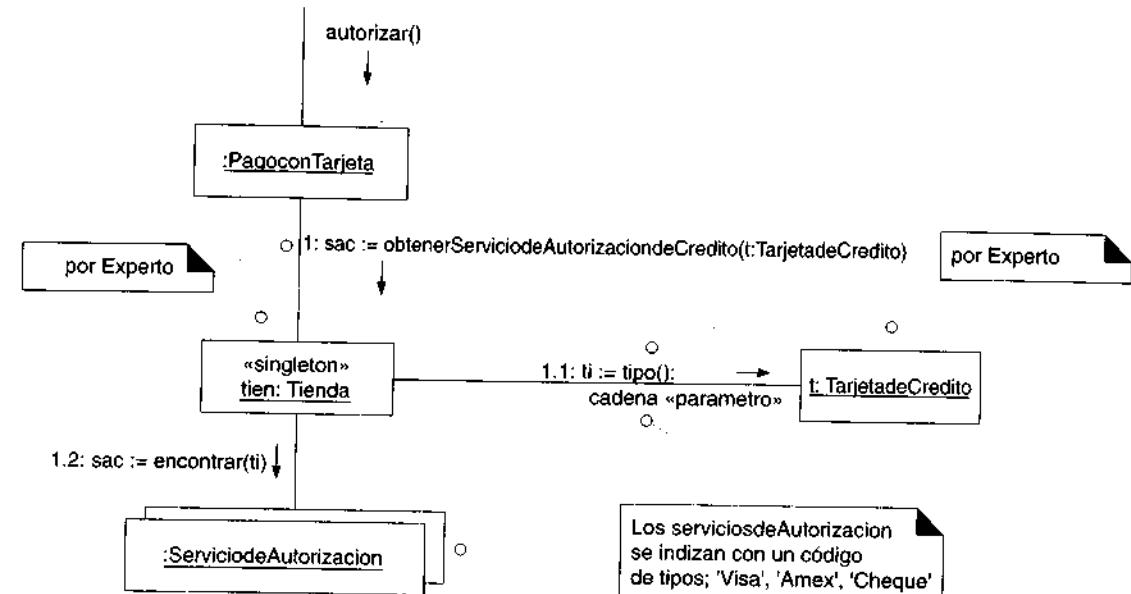


Figura 35.13 Localización de un servicio.

Una vez localizado el Agente Remoto adecuado, se le confiere la responsabilidad de terminar la autorización, según se indica en la figura 35.14.

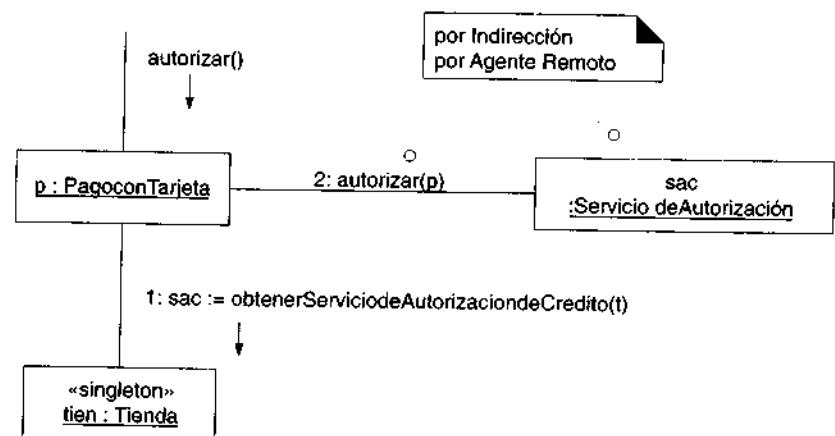


Figura 35.14 Utilización del patrón Agente Remoto.

El Agente Remoto *Servicio de Autorización de Crédito* debe transmitir un mensaje de solicitud y recibir una respuesta, lo cual se hace en la figura 35.15.

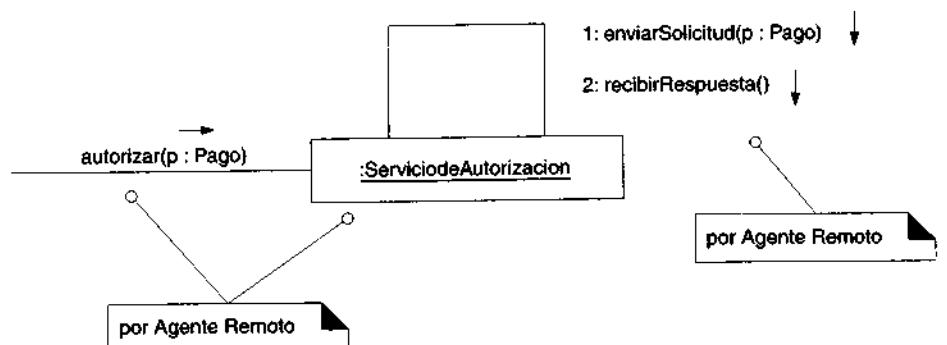


Figura 35.15 Acciones del patrón Agente Remoto.

35.5.1 Problemas

- El lector quizás se haya percatado de que este diseño viola el principio No Hables con Extraños, cuando *PagoconTarjeta* alcanza visibilidad ante el *Servicio de Autorización de Credito* y le envía el mensaje *Autorizar*. Cuando un objeto es un registro de otra clase de objetos (como lo es *Tienda* de *Servicio de Autorización de Credito*), frecuentemente se viola este principio. Una alternativa consiste en darle a la *Tienda* una interfaz *Autorizar(unPago)* para soportar No Hables con Extraños.
- Agente Remoto no es más que uno de tantos agentes. Favor de consultar los detalles en [GHJV95] y en [BMRSS96].
- El Agente Remoto es un simple método de manejar la comunicación. Otras formas basadas en patrones se explican en [BMRSS96].

35.6 El patrón Fachada y el Agente Dispositivo (Pandilla de los Cuatro)

35.6.1 Integración

Nuestro sistema debe utilizar un módem para marcar y conectarse con el servicio externo. Supongamos que el sistema operativo en cuestión ofrece una interfaz basada en funciones para emplear un módem (por ejemplo, *OS_DIAL(...)*). Estas funciones no orientadas a objetos pueden integrarse o quedar "envueltas" dentro de una clase que las agrupe, proporcionando a las funciones una interfaz con el método. En términos generales, a esto se le llama **integración**; puede aplicarse para crear una interfaz orientada a objetos con cualquier otra que no se apegue a este paradigma.

Por ejemplo, podemos definir una clase *Modem* que integre las llamadas.

35.6.2 Fachada

Se le da el nombre de Fachada —uno de los patrones de la Pandilla de los Cuatro— a la clase definida que ofrece una interfaz común con un conjunto heterogéneo de interfaces, entre ellas la clase *Modem*. Las interfaces heterogéneas pueden ser un conjunto de funciones, un esquema, un grupo de otras clases o un subsistema (local o remoto).

Fachada

Contexto/Problema

Se requiere una interfaz común unificada con un conjunto heterogéneo de interfaces, como la de un subsistema. ¿Qué debe hacerse?

Solución

Definir una sola clase que unifique las interfaces y asignarle la responsabilidad de colaborar con el subsistema.

35.6.3 Agente Dispositivo

Además de ser una Fachada, un *Modem* es además una clase de Agente del módem físico real: un Agente Dispositivo.

Agente Dispositivo

Contexto/Problema

Se requiere interactuar con un dispositivo electromecánico. ¿Qué hacer?

Solución

Definir una clase que represente al dispositivo y asignarle la responsabilidad de interactuar con él.

35.6.4 Indirección

Observe que Fachada, Agente Remoto y Agente Dispositivo, como tantos otros agentes, constituyen una variante del patrón básico Indirección de GRASP.

35.6.5 Adecuación y serialización

En el diseño orientado a objetos, cuando nos comunicamos con el mundo externo a través de un mecanismo no orientado a objetos, una técnica común consiste en crear objetos que representen los mensajes dirigidos al interior y al exterior para transformarlos luego entre objetos y cadenas. La conversión de un objeto en una representación de cadena recibe el nombre de **serialización**, para la cual algunos lenguajes (Java entre ellos) cuentan con un soporte integrado. Si queremos enviar lo que consideramos un mensaje orientado a objetos con parámetros a través de un medio de comunicación no orientado a objetos (por ejemplo, un “socket”), generalmente debemos transformar el mensaje y los parámetros (mediante la serialización) en un flujo de bytes adecuado para la transmisión y para el servidor receptor. A este proceso se le da el nombre de **adecuación**. Nuestra responsabilidad en la serialización y en la adecuación depende del lenguaje y del mecanismo de comunicación. Si utilizamos Java y su mecanismo de invocación remota del método (RMI), tan sólo habremos de asegurarnos que la serialización produzca una disposición conveniente de cadenas para los parámetros. Cuando no se cuenta con un mecanismo integrado de comunicación distribuida como la invocación remota del método, casi siempre el patrón Agente Remoto se encarga de adecuar y **desadecuar** (transformar en comandos y en objetos las cadenas que retornan).

35.6.6 Utilización del Agente Remoto y del Agente Dispositivo

En conclusión, la clase *Modem* es a la vez un Agente Dispositivo y una Fachada. El Agente Remoto *Servicio de Autorización* crea un objeto que representa al mensaje de salida, lo serializa convirtiéndolo en cadena y luego colabora con el *Modem* para enviarlo afuera. Esto se muestra en la figura 35.16.

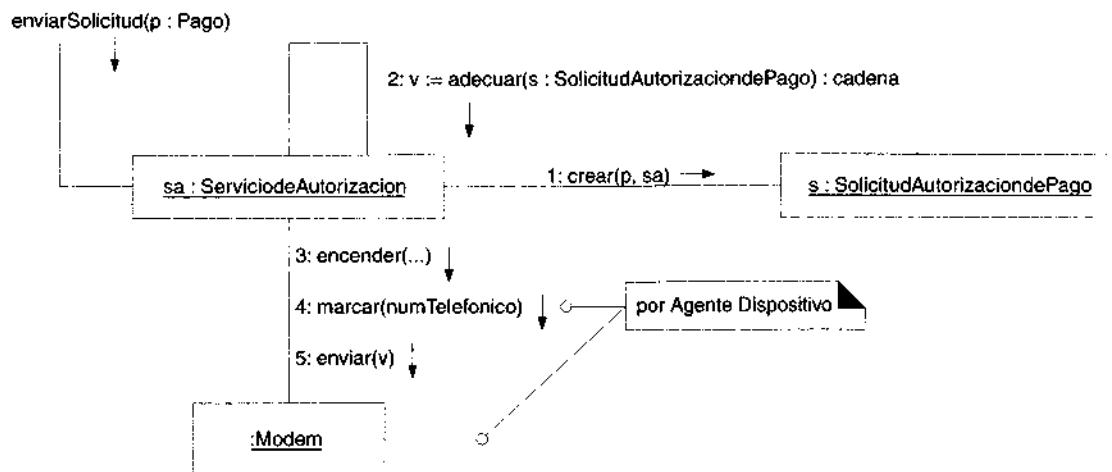


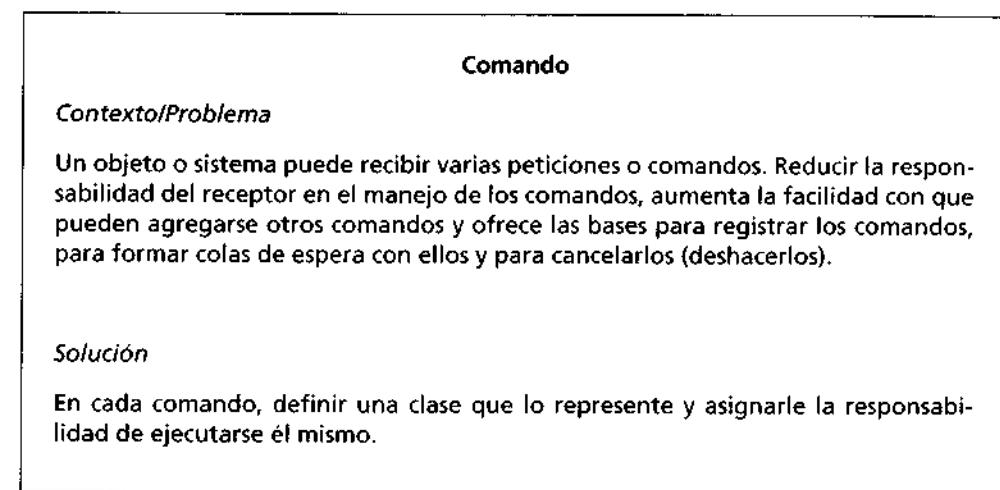
Figura 35.16 Utilización de un Agente Dispositivo.

35.7 El patrón Comando (Pandilla de los Cuatro)

Una vez que el *Servicio de Autorización* envía la solicitud, se dirige también a sí mismo un mensaje *recibirRespuesta* y espera la contestación.¹

Cuando la recibe, este mensaje con formato de cadena se desadecua e introduce en una instancia *RespuestaAprobatoria de Pago con Tarjeta* o en una instancia *RespuestaReprobatoria de Pago con Tarjeta*, según el código de aprobación que se utilice.

¿Cómo debería efectuarse la ejecución? El patrón Comando de la Pandilla de los Cuatro indica que estos objetos de respuesta representan una clase de solicitud de comando o acción y que han de ejecutarse a sí mismos, basándose para ello en el patrón Polimorfismo.



Como se aprecia en la figura 35.17, cada respuesta tiene su propia clase de Comando, con un método *ejecutar* polimórfico.

Cuando se desadecua una respuesta y se la incorpora a la instancia *RespuestaAprobatoria de Pago con Tarjeta* o a una instancia *RespuestaReprobatoria de Pago con Tarjeta*, se le envía un mensaje *ejecutar*, según vemos en la figura 35.18.

Nótese que esa figura sólo muestra la creación de una superclase abstracta *Respuesta de Autorización de Pago*, aun cuando se produjo una instancia de una subclase.

Los casos polimórficos *ejecutar* se incluyen en un nuevo diagrama de colaboración, según se advierte en las figuras 35.19 y 35.20.

¹ También es probable una espera asincrónica en un hilo (thread), pero rebasa el ámbito de nuestra investigación.

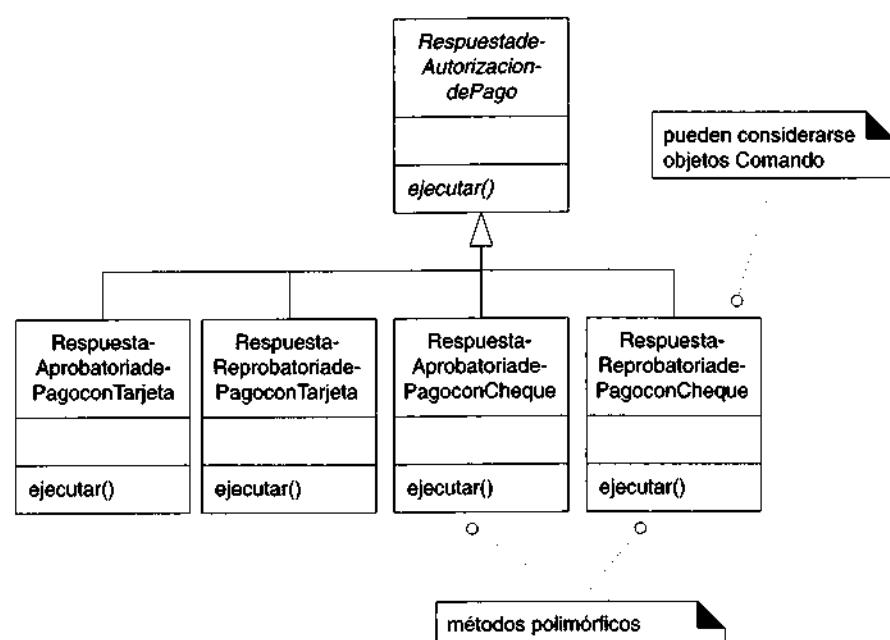


Figura 35.17 Clases de comandos.

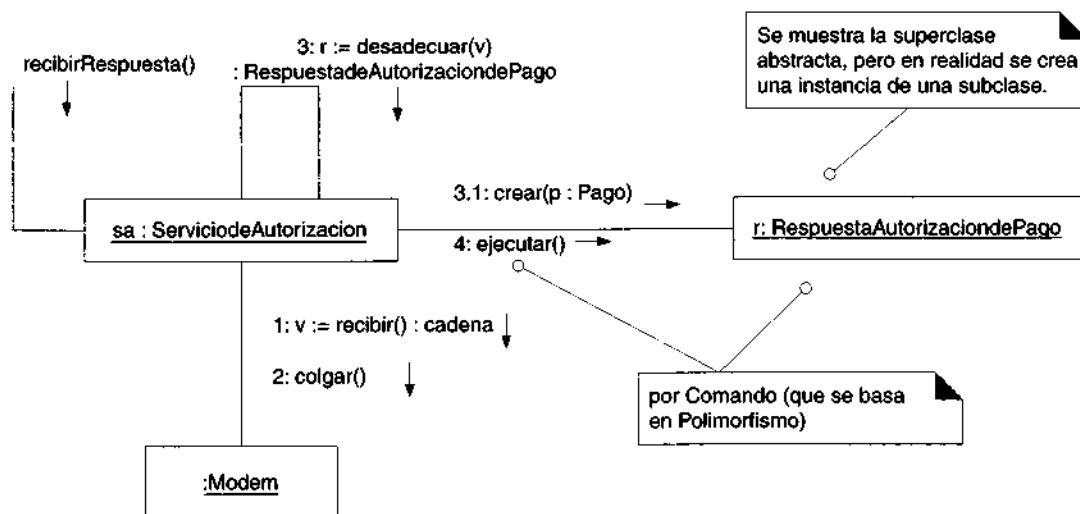


Figura 35.18 Aplicación del patrón Comando.

35.7.1 RespuestaAprobatoriadePagoconTarjeta: ejecutar

En virtud de los patrones Comando y Polimorfismo, la *RespuestaAprobatoriadePagoconTarjeta* recibe un mensaje *ejecutar*. En caso de aprobación, debe registrarse la respuesta en el sistema de cuentas por cobrar. Como se observa en la figura 35.9, esto lo hace por medio de los patrones Singleton, No Hables con Extraños y Agente Remoto.¹

Finalmente, hay que transformar el estado de pago en estado autorizado. (Nótese que a la respuesta se le confirió visibilidad de atributo ante el pago al momento de producir la respuesta.)

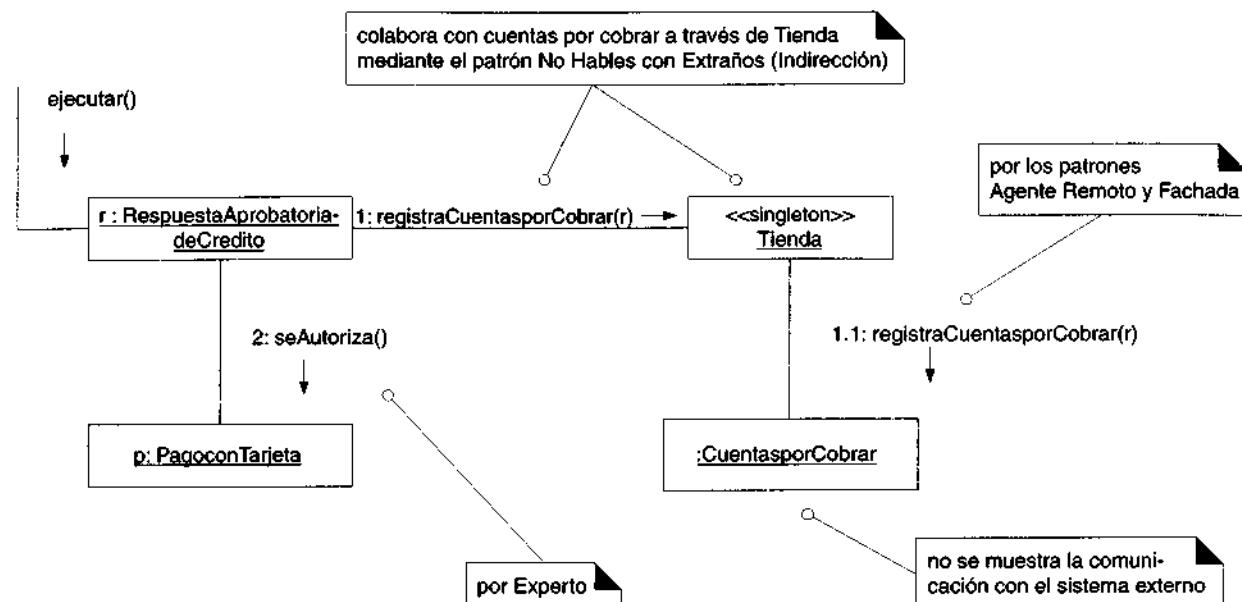


Figura 35.19 Mensaje ejecutar para RespuestaAprobatoriadeCredito.

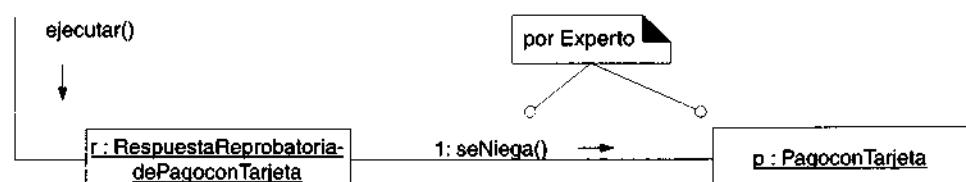


Figura 35.20 Mensaje ejecutar para RespuestaReprobatoriadePagoconTarieta.

¹ Observe la abstracción posible cuando se comunica un diseño al utilizar los nombres de los patrones.

35.7.2 *RespuestaReprobatoria de Pago con Tarjeta: ejecutar*

En virtud de los patrones Comando y Polimorfismo, la instancia *RespuestaReprobatoria de Pago con Tarjeta* recibe un mensaje *ejecutar*. En caso de una respuesta reprobatoria o negativa, ésta ha de cambiar el estado del pago que se rechaza (figura 35.20).

35.8 Conclusión

En este capítulo hemos estudiado varios patrones de la Pandilla de los Cuatro y también hemos hecho aplicaciones de los patrones GRASP.

La principal lección que extraemos de nuestra exposición es la siguiente: durante el diseño orientado a objetos, las responsabilidades pueden asignarse partiendo de la utilización de patrones. Éstos ofrecen un conjunto explicable de técnicas especializadas con las cuales podemos construir sistemas bien diseñados que se orienten a objetos.

PARTE VIII

TEMAS ESPECIALES

OTRA NOTACIÓN DE UML

36.1 Introducción

En este capítulo ofrecemos una breve reseña de una parte de la notación del lenguaje UML que todavía no hemos explicado. Recomendamos leer los comentarios del interior de los diagramas para obtener mejor detalle.

36.2 Notación general

36.2.1 *Notas y restricciones*

Una nota es un comentario del diagrama. No ejerce influencia semántica sobre los elementos.

diseñar esto cuidadosamente

Una restricción se anexa a un elemento. Ejerce influencia semántica sobre él.

```
Persona--edad()
{
    return hoy - fechaCumpleaños
}
```

segundo formato
de las restricciones

{hoy – fechaCumpleaños}

Persona
fechaCumpleaños : Fecha
edad()

Figura 36.1 Notas y restricciones.

36.2.2 Dependencia

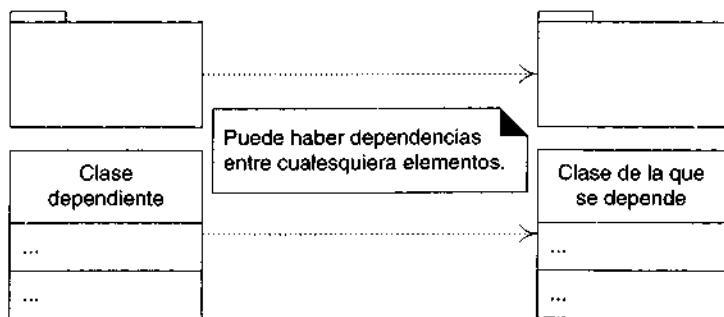


Figura 36.2 Dependencias.

36.2.3 Estereotipos y especificación de las propiedades

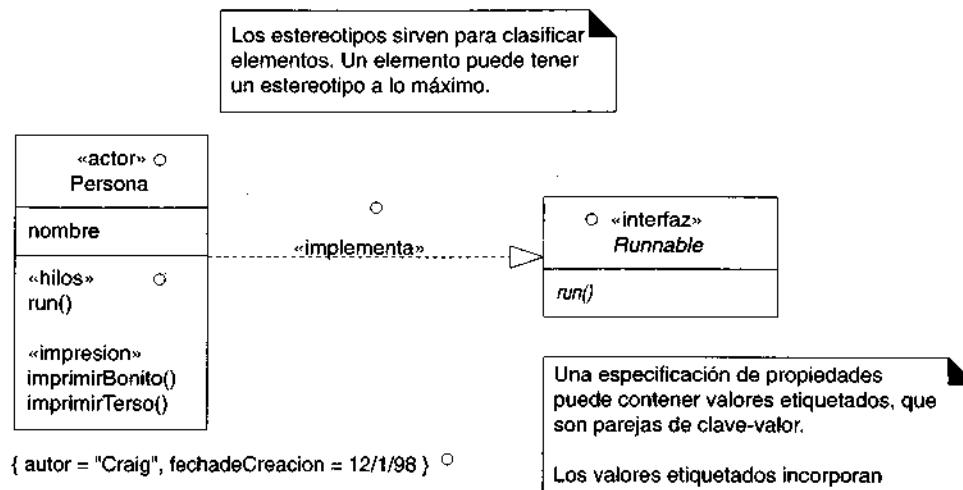


Figura 36.3 Estereotipos y propiedades.

36.3 Interfaces

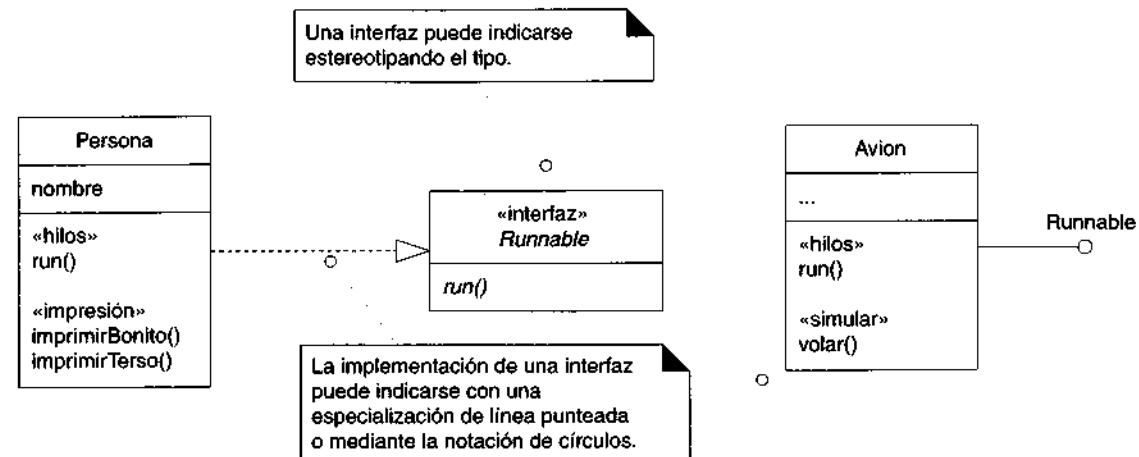


Figura 36.4 Interfaces.

36.4 Diagramas de implementación

36.4.1 Diagramas de componentes

Los diagramas de componentes muestran las dependencias del compilador y del "runtime" entre los componentes del software; por ejemplo, los archivos del código fuente y los DLL.

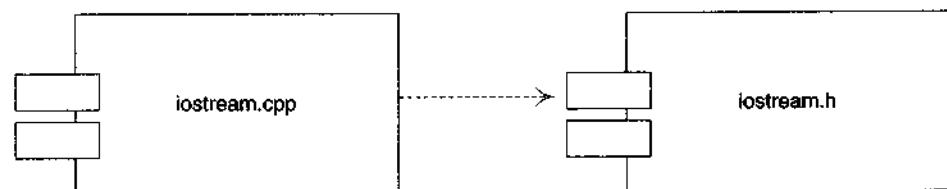


Figura 36.5 Componentes del software.

36.4.2 Diagramas de despliegue

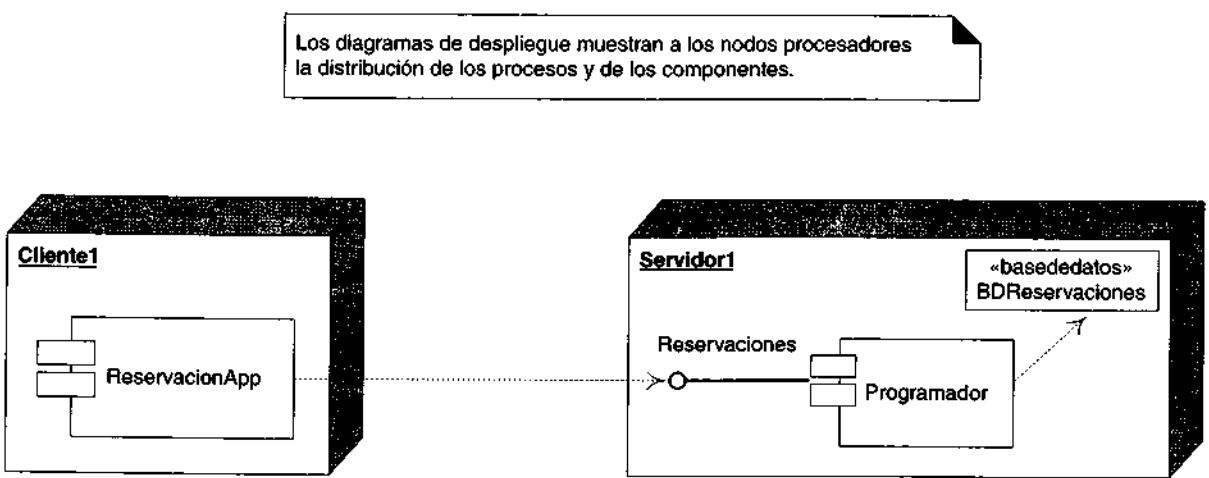


Figura 36.6 Diagrama de despliegue.

36.5 Mensajes asincrónicos en los diagramas de colaboración

El lenguaje UML cuenta con una notación que muestra los mensajes asincrónicos y los nuevos hilos de control. Por ejemplo, en Java una instancia *Avion* que implementa la interfaz *Ejecutable* (el método *ejecutar*) puede crear un *Hilo* y enviarle un mensaje iniciar (asíncrono). El mensaje ejecutar será entonces devuelto a la instancia *Avion*. El método *ejecutar* es el cuerpo del hilo y *Avion* es un “objeto activo” que ejecuta su propio hilo.

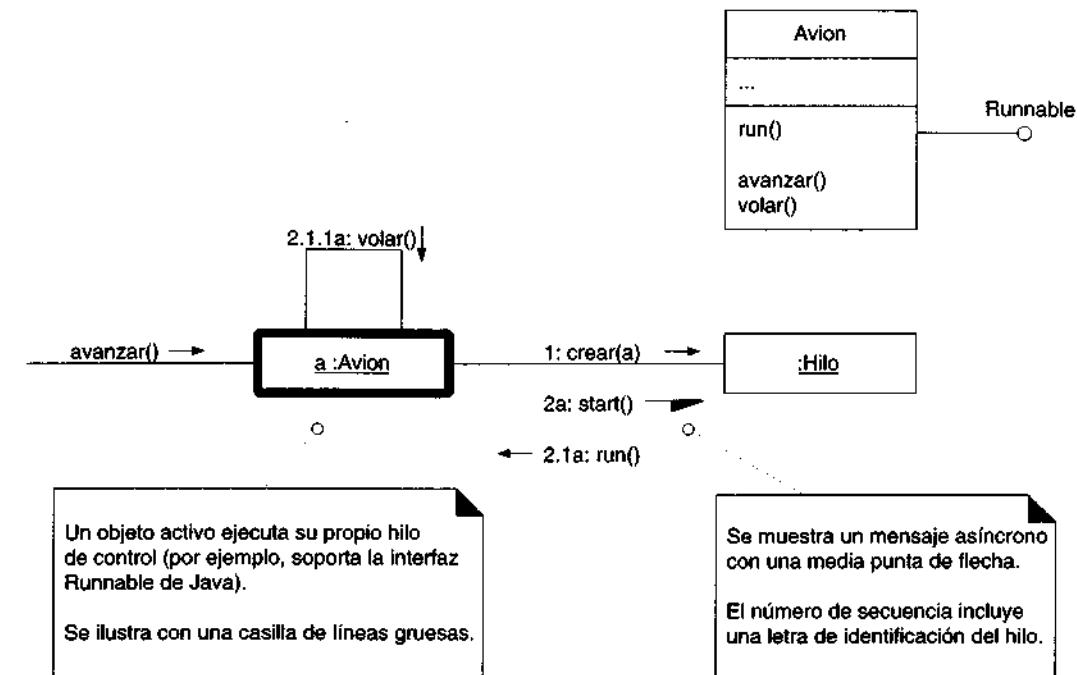


Figura 36.7 Concurrencia y los mensajes asíncronos.

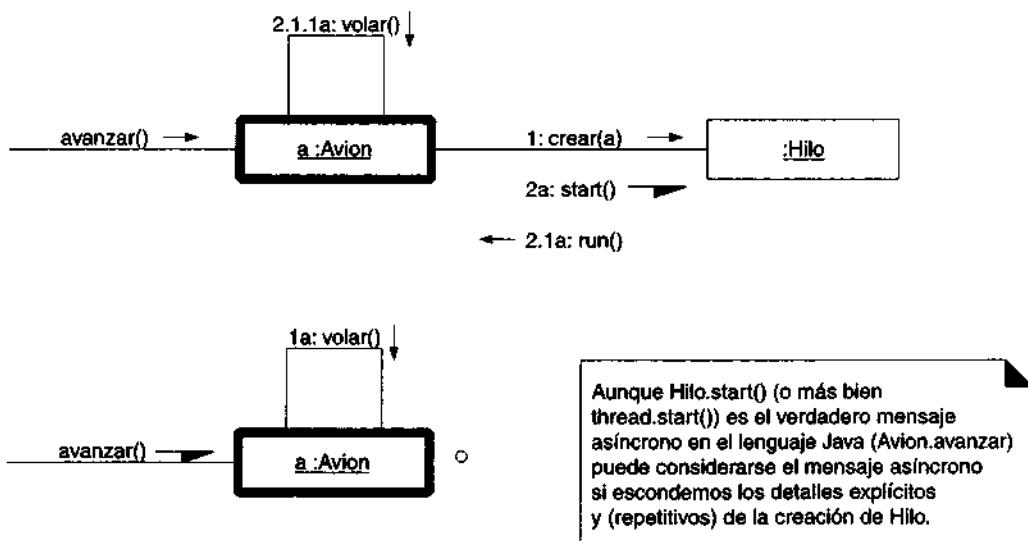


Figura 36.8 Creación implícita de hilos y llamada asíncrona en Java.

36.6 Interfaces de paquetes

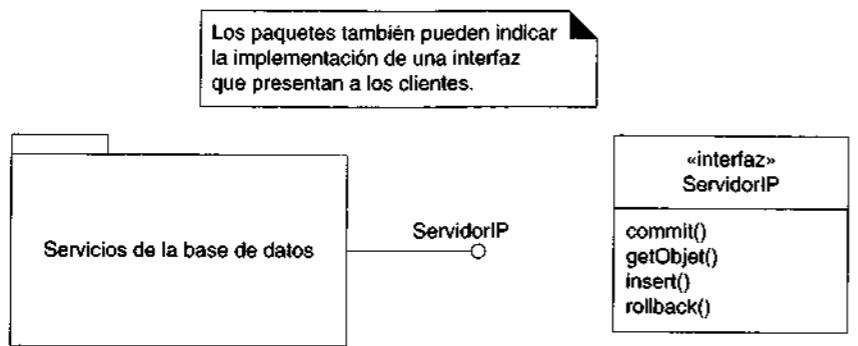


Figura 36.9 Interfaz de un paquete.

PROBLEMAS DEL PROCESO DE DESARROLLO

Objetivos

- Seguir un proceso de desarrollo iterativo, incremental y orientado a los casos de uso.
- Organizar el trabajo a partir de los ciclos de desarrollo.
- Programar un ciclo de desarrollo dentro de un plazo perentorio (o time-box).
- Organizar el desarrollo en equipos paralelos a lo largo de las líneas arquitectónicas.
- Expresar requerimientos técnicos no evidentes como casos de desarrollo de sistemas y calendarizarlos.

37.1 Introducción

En este capítulo estudiaremos algunos problemas fundamentales relacionados con el desarrollo de los sistemas. Vamos a tratar más a fondo algunos temas ya expuestos, repetiremos algunos puntos y luego profundizaremos ideas ya explicadas en capítulos anteriores. Los procesos del desarrollo de software son un tema muy extenso; no sólo abarcan los pasos esenciales de un proyecto, sino también su dirección y muchísimas cuestiones afines. En este capítulo nos limitaremos a algunos de los puntos más importantes.

37.2 ¿Por qué molestarnos?

Con frecuencia oímos verdaderas historias de terror acerca de los fracasos de proyectos de software que han costado millones de dólares: sistemas fiscales, sistemas de defensa y otros. La conclusión salta a la vista: el desarrollo de software es un negocio riesgoso. Un estudio reveló que 31% de los proyectos no se concluyen, que 53% de ellos rebasa casi en 200% el costo estimado, que en 1995 las compañías estadounidenses y las dependencias gubernamentales invirtieron 81 millones de dólares en proyectos de software que fueron cancelados [Standish94]. Simplemente no están obteniéndose los resultados deseados.

Se requiere tiempo y, por tanto, dinero para seguir un proceso de desarrollo y realizar el análisis y el diseño orientados a objetos. ¿Por qué molestarnos? ¿Por qué no iniciar de inmediato la programación? He aquí algunas razones por las cuales debemos tomarnos la molestia de hacer esas dos cosas:

1. Como indican las estadísticas anteriores, los proyectos de software son una actividad riesgosa; de ahí que el motivo primordial sea aminorar el riesgo, esto es, aumentar la probabilidad de crear un sistema eficiente. Y una estrategia para conseguirlo consiste en examinar rigurosamente los requerimientos y elaborar un diseño formal.
2. Conviene crear un proceso que puedan reproducir los individuos, y sobre todo los equipos. No es sustentable el desarrollo que se funda en heroicos esfuerzos individuales.
3. Es más barato y fácil efectuar cambios durante las actividades de análisis y diseño que en la fase de construcción: el software es “más duro” de lo que pensamos.
4. Podemos atenuar y manejar la abrumadora complejidad con sólo modelar los sistemas y hacer abstracciones para detectar los detalles esenciales; así se evita una excesiva complejidad.
5. Conviene crear sistemas robustos, susceptibles de mantenimiento y capaces de soportar una mayor reutilización del software. Por lo regular estas metas no se cumplen sin un riguroso examen y diseño efectuados antes de la programación.

A todo lo anterior le da soporte una aplicación metódica del análisis y del diseño. Por desgracia, muchas compañías no le dan suficiente importancia y son renuentes a invertir recursos en este tipo de actividades, optando generalmente por iniciar cuanto antes el proceso de codificación.

37.3 Directrices de un proceso eficiente

Los siguientes principios forman parte de un proceso recomendable de desarrollo [Booch96]:

- desarrollo iterativo e incremental
- desarrollo orientado a los casos de uso
- desarrollo que comienza por definir la arquitectura (“centrado en la arquitectura”)

37.4 Desarrollo iterativo e incremental

37.4.1 Desventajas de un ciclo de vida en cascada

La característica distintiva de un ciclo de desarrollo en cascada es que contiene un *solo* paso de análisis, diseño y construcción; en un único paso se realiza *todo* el análisis, luego *todo* el diseño, luego la codificación y finalmente todas las pruebas. He aquí algunas de sus deficiencias:

- carga frontal de gran complejidad: excesiva complejidad
- retraso de la retroalimentación
- congelamiento temprano de las especificaciones, mientras cambia el ambiente de negocios

37.4.2 Un ciclo de vida iterativo

En contraste con el ciclo de vida en cascada, el ciclo iterativo se funda en el perfeccionamiento gradual de un sistema a través de *múltiples* ciclos de análisis, diseño y construcción, según se aprecia en la figura 37.1.

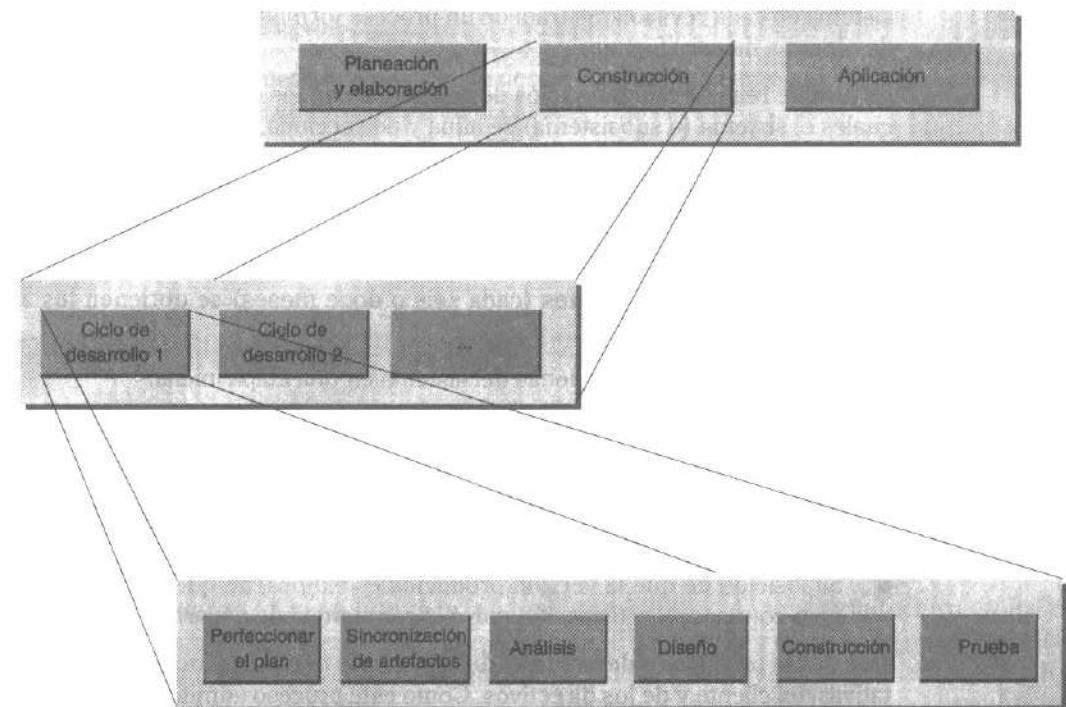


Figura 37.1 Ciclo de vida iterativo.

En cada ciclo se trata un conjunto relativamente pequeño de requerimientos y el sistema crece incrementalmente conforme van concluyéndose los ciclos. Por eso a este proceso se le califica de iterativo e incremental. Aporta los siguientes beneficios:

- La complejidad nunca resulta abrumadora porque en un ciclo sólo se abordan unidades cuya complejidad es manejable. Se evitan así la "parálisis del análisis" y la "parálisis del diseño".
- Se genera retroalimentación en las fases iniciales, porque la implementación ocurre rápidamente en un pequeño subconjunto del sistema. Esta retroalimentación le suministra información al análisis de ciclos subsecuentes, pudiendo además mejorarlo. Llega a los diseñadores a través de los resultados de su fase de implementación y pruebas o a través de la comunidad que usa una versión ya liberada del sistema.
- El equipo de desarrollo podrá ir re-aplicando gradualmente su creciente dominio de las herramientas: no es necesario que desde un principio se sirva de las mejores (y más complejas) características del lenguaje/herramienta ni tampoco que se sirva exclusivamente de técnicas no dominadas para dividir el sistema.
- Los requerimientos son ajustables para que respondan a las necesidades cambiantes de la compañía, a medida que avanza el proyecto.

El desarrollo iterativo no significa hacer una pausa, alcanzar una meta importante y luego volver a detenerse. He aquí una definición más exacta del **desarrollo iterativo**: es un proceso *planeado* que consiste en revisar varias veces un área, mejorando el sistema en cada revisión. Se trata de un proceso formalmente planeado y programado, de ninguna manera fortuito.

Es posible realizar muchos ciclos de desarrollo con los mismos requerimientos, en los cuales el sistema (o subsistema) se afina y perfecciona, y también hacerlo con nuevos requerimientos (que es el caso más común).

El **desarrollo incremental** consiste en agregarle funcionalidad a un sistema en los ciclos productivos; con cada uno va aumentando la funcionalidad. Una creación incremental se compone de varios ciclos de desarrollo iterativo. Con producciones incrementales bastante frecuentes (cada seis o doce meses) se obtienen los siguientes beneficios:

- Integración global y pruebas del sistema en una etapa inicial.
- Retroalimentación temprana al diseñador por parte de la comunidad de usuarios finales, utilizando el sistema generado.
- Retroalimentación al cliente y a la comunidad de usuarios finales respecto a la habilidad y confiabilidad del equipo encargado del desarrollo.
- Suposición de que la versión producida es exitosa, de que desde un principio genera confianza y satisfacción entre la comunidad de usuarios.

Una de las principales desventajas del desarrollo iterativo e incremental son las expectativas del cliente y de los directivos. Como este proceso admite demostraciones en las primeras fases, el cliente y los directivos tenderán a pensar que el sistema "casi está terminado": el conocido problema de "funciona la interfaz para el usuario = sistema

terminado". Se requiere de un manejo riguroso de las expectativas y una comunicación frecuente y clara por parte del equipo de desarrollo, si se quiere atenuar esta reacción, desgraciadamente inevitable.

37.5 Desarrollo orientado a los casos de uso

El proceso de desarrollo debería evidenciar el influjo de los casos de uso y organizarse en torno a ellos. Por ejemplo:

- Los requerimientos se organizan y se redactan a partir de los casos de uso.
- En las estimaciones influyen directa e indirectamente los casos: su número, su complejidad, los servicios de soporte que requieren, etcétera.
- Los programas se organizan a partir de ciclos iterativos, los cuales se basan en la realización de los casos de uso.
- A partir de los casos de uso se escogen los requerimientos que deberían cumplirse en un ciclo iterativo particular.
 - Por ejemplo, en la iteración 1 nos ocuparemos del caso A, en la iteración 2 nos ocuparemos de una versión simplificada del caso B y en la iteración 3 nos ocuparemos del resto del caso B e íntegramente de los casos C y D.
- Las actividades de un ciclo de desarrollo procuran ante todo llevar a cabo el caso o casos que se consideran en el ciclo.
- Los conceptos y las clases de software pueden identificarse atendiendo a los casos de uso.
- Se escriben casos de prueba para validar los que se llevan a cabo.

37.6 Énfasis inicial en la arquitectura

Dentro del contexto de este capítulo, con el término **arquitectura** designamos la estructura de alto nivel de los subsistemas y de los componentes, así como de sus interfaces, conexiones e interacciones. La arquitectura se compone de un conjunto de esquemas, subsistemas, clases, asignaciones de responsabilidades y colaboraciones entre objetos que cumplen con las funciones del sistema.

Un proceso eficaz estimula la generación *temprana* de una arquitectura global del sistema: está centrado en la arquitectura. Esto significa que en los primeros ciclos se ponen de relieve la investigación y el desarrollo de esquemas arquitectónicos bien diseñados; al mismo tiempo se tiene una visión global de una arquitectura cohesiva a lo largo del proyecto. El diseño se ha construido en una forma sistemática y organizada.

Una arquitectura bien diseñada reúne las siguientes cualidades ideales:

- tiene subsistemas sustentados en arquitectura de capas
- ofrece bajo acoplamiento entre subsistemas
- es fácil de entender
- es robusta, flexible y se incrementa de modo adecuado
- tiene muchos componentes reutilizables
- se orienta a los casos de uso más importantes y riesgosos
- consta de interfaces bien definidas con el sistema y con los subsistemas

Durante el desarrollo temprano, deberá darse prioridad a las capas y a los subsistemas con mayor riesgo e incertidumbre. Pueden conformar la capa del dominio o alguna de las capas de servicios de alto nivel. No obstante, la importancia especial que se concede a la arquitectura significa que desde un principio se atenderá a la mayoría de las capas y de los subsistemas. Esto incluye sus componentes, sus interfaces e interacciones con otras capas.

37.7 Fases del desarrollo

37.7.1 Versiones de producción

En el macrónivel, un proyecto de desarrollo consta de una o más versiones incrementales de producción: una versión que se destina al uso. En el caso del software comercial, ello significa una versión que se vende; en el caso del software de una empresa, significa un uso operacional.

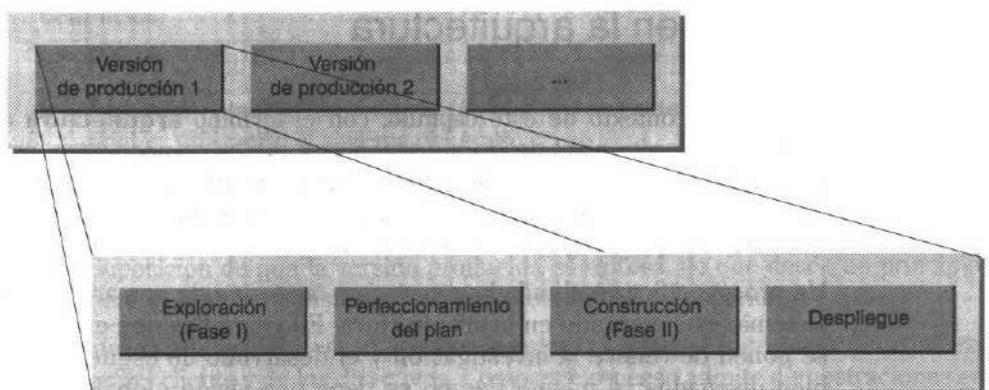


Figura 37.2 Desarrollo incremental del producto.

Una versión incremental de producción consta de las siguientes fases:

1. **Explorar:** se definen los requerimientos y se construye el sistema a través de varios ciclos de desarrollo hasta alcanzar un punto donde puede formularse un plan aceptable.
2. **Perfeccionar el plan:** se modifica el programa del proyecto, sus presupuestos y otros aspectos, basándose para ello en los resultados de la exploración.
3. **Construir:** es la fase principal de la terminación de desarrollo.
4. **Desplegar:** se traslada el sistema al uso de producción.

37.7.2 Principales pasos del desarrollo

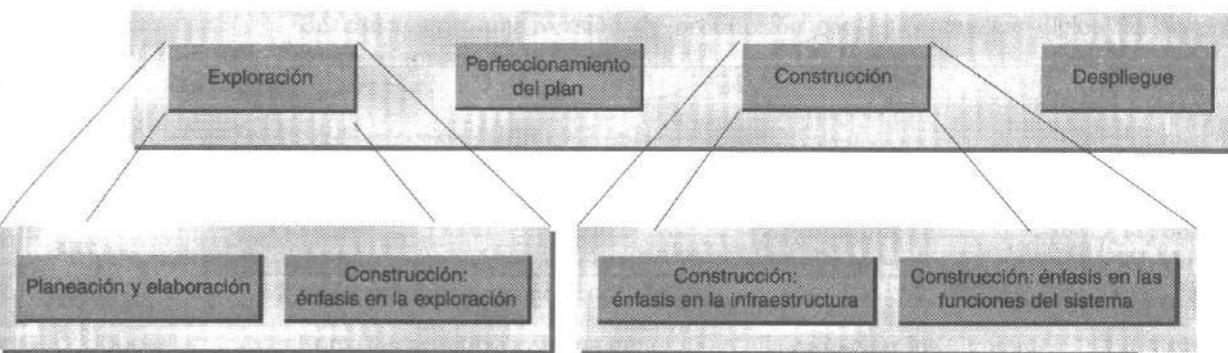


Figura 37.3 Principales pasos del desarrollo.

En una versión de producción, la fase Explorar (también llamada fase I) se propone entender los requerimientos y luego examinar la construcción de un sistema hasta lograr generar una estimación y un programa aceptables.¹ Esta fase está constituida por dos grandes etapas:

1. **Planeación y elaboración:** los requerimientos iniciales de obtención, definición, planeación y formulación del problema.
2. **Construcción: énfasis en la exploración.** Un periodo de uno a tres meses de los ciclos de desarrollo en que se llevan a cabo algunos casos de uso y trabajo de arquitectura de alto nivel.

La importante fase de construcción (llamada también fase II) tiene por objeto desarrollar íntegramente el sistema. Los ciclos de desarrollo de que consta pueden agruparse así:

1. **Construcción: énfasis en la infraestructura.** Son los ciclos iniciales de desarrollo cuyo fin es establecer y perfeccionar la arquitectura técnica global, como

¹ Son mera ficción las estimaciones y los programas de actividades obtenidos sin ciclos exploratorios de desarrollo que midan concretamente las dimensiones y dificultad del problema, así como la rapidez del equipo de desarrollo.

esquemas (frameworks), capas y subsistemas. Esta fase sí incluye realizar los casos del dominio (*Comprar Productos*, entre ellos), pero se procura ante todo sentar las bases arquitectónicas que soporten su realización.¹

- Construcción: énfasis en las funciones del sistema.** Son los ciclos restantes que subrayan el cumplimiento de las funciones de aplicación, durante los cuales queda por hacer poco trabajo de infraestructura.

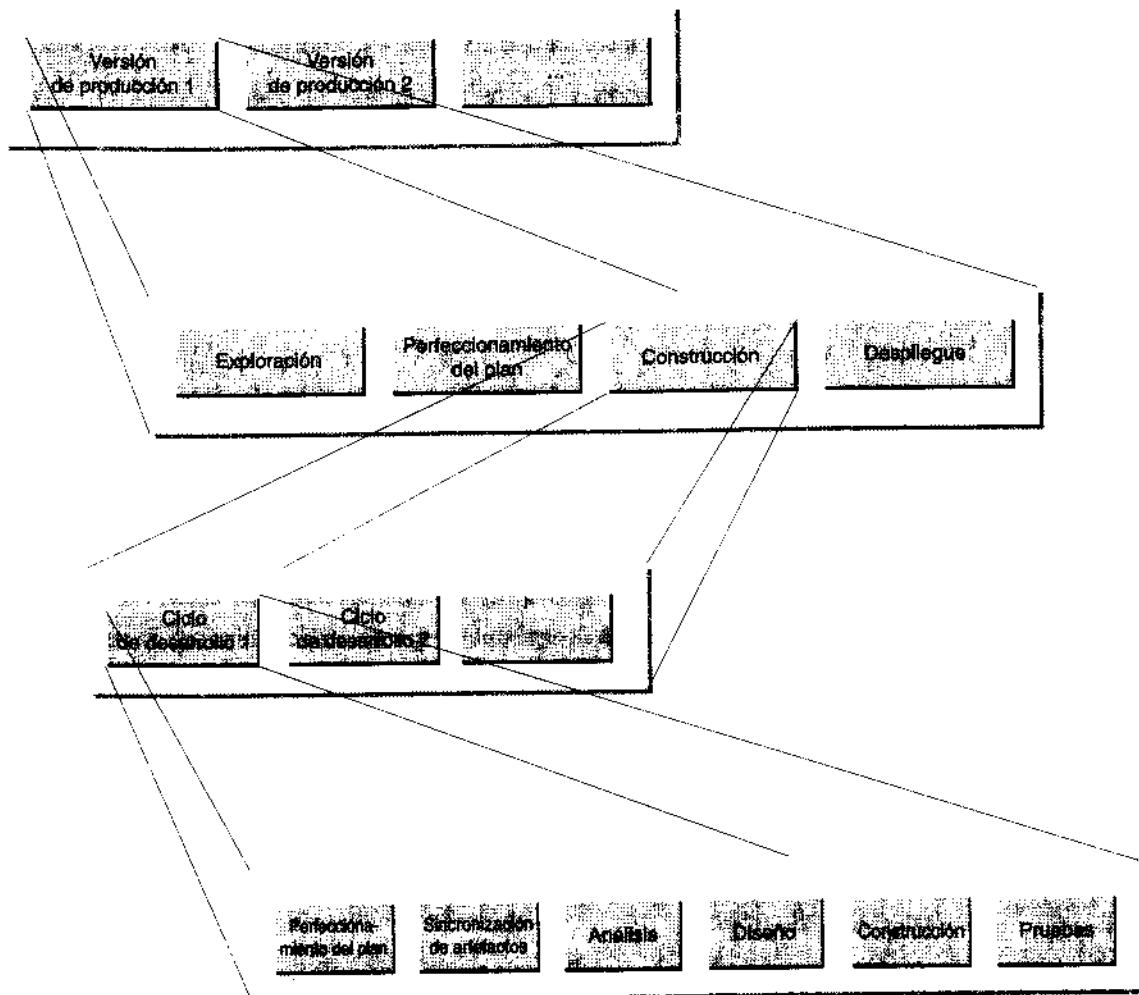


Figura 37.4 Ciclos de desarrollo iterativo.

¹ A veces se subestima mucho el tiempo necesario para crear la infraestructura y concluir esta fase.

37.7.3 Comparación entre la versión de producción y los ciclos de desarrollo

Un error muy frecuente en el desarrollo iterativo consiste en confundir una versión de producción con un ciclo de desarrollo.

Por ejemplo, desde el punto de vista del desarrollo iterativo no conviene tener un solo ciclo de desarrollo semestral y luego generar un sistema de producción al final de ese periodo.

Por el contrario, si hay que entregar un sistema de producción cada seis meses, ese lapso debería dividirse en —por ejemplo— seis ciclos mensuales: seis ciclos para analizar, diseñar, implementar y efectuar pruebas.

En conclusión, una versión de producción consta de *muchos* ciclos de desarrollo, como se aprecia en la figura 37.4.

37.7.4 La fase de planeación y elaboración

La fase de planeación y elaboración de un proyecto (figura 37.5) abarca la concepción inicial, la investigación de alternativas, la planeación y la especificación de requerimientos entre otras cosas.

Entre los artefactos que se generan en ella cabe citar los siguientes:

- *Plan*: programa del proyecto, recursos, presupuesto y otros elementos.
- *Informe preliminar de investigación*: motivos, opciones, necesidades de la empresa, etcétera.
- *Especificación de requerimientos*: declaración de los requerimientos.
- *Glosario*: un diccionario de términos (nombres de conceptos y otros vocablos) e información relacionada con ellos; por ejemplo, reglas y restricciones.
- *Prototipo*: un sistema de prototipos cuya finalidad es facilitar la comprensión del problema, de los problemas de alto riesgo y de los requerimientos.
- *Casos de uso*: descripciones textuales de los procesos del dominio.
- *Diagramas de los casos de uso*: descripción visual de todos los casos y de sus relaciones.
- *Modelo conceptual preliminar*: modelo inicial que ayuda a entender el vocabulario del dominio, sobre todo en su relación con los casos de uso y con la especificación de requerimientos.

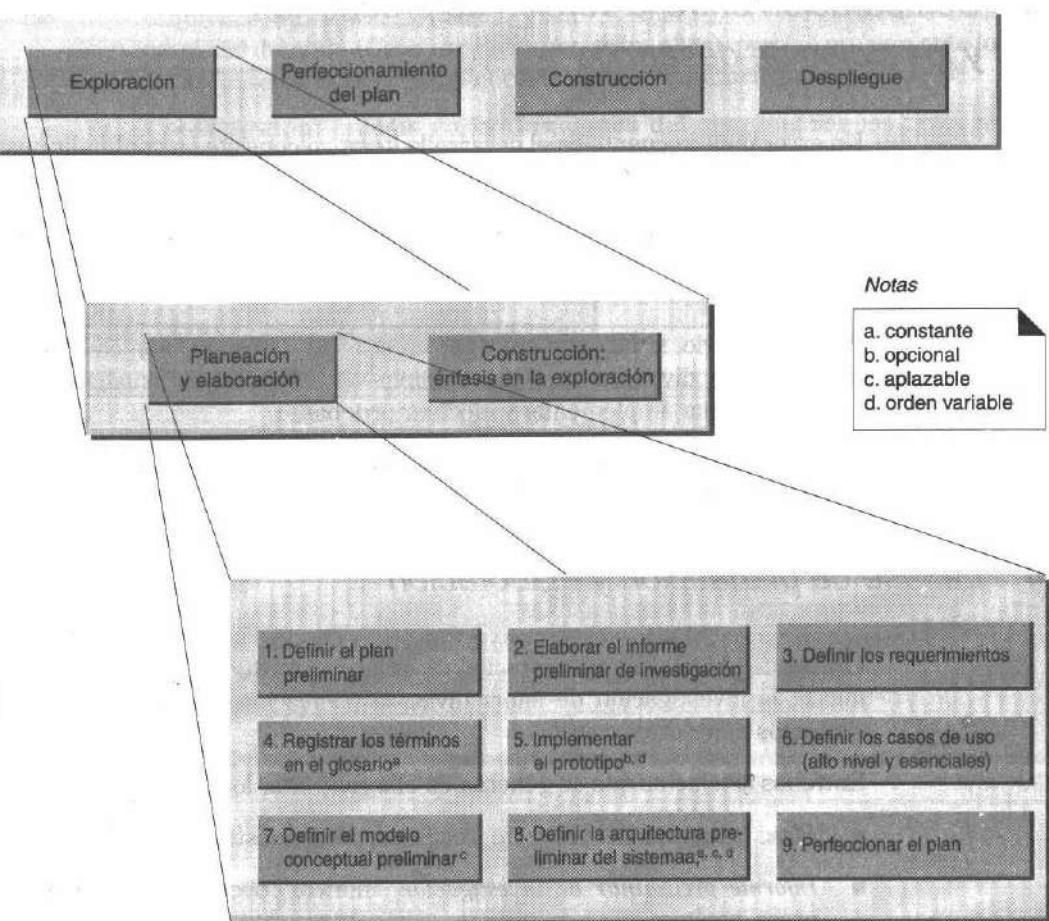


Figura 37.5 Fase de planeación y elaboración del desarrollo.

37.7.5 Fase de construcción: énfasis en la exploración

Esta fase, que aparece en la figura 37.6, incluye los ciclos de desarrollo de uno a tres meses. Se examinan los casos de uso de alto nivel que influyen profundamente en la arquitectura. Con ello se busca explorar lo suficiente como para recabar información, a partir de la cual idear un programa y una estimación confiables para el proyecto.

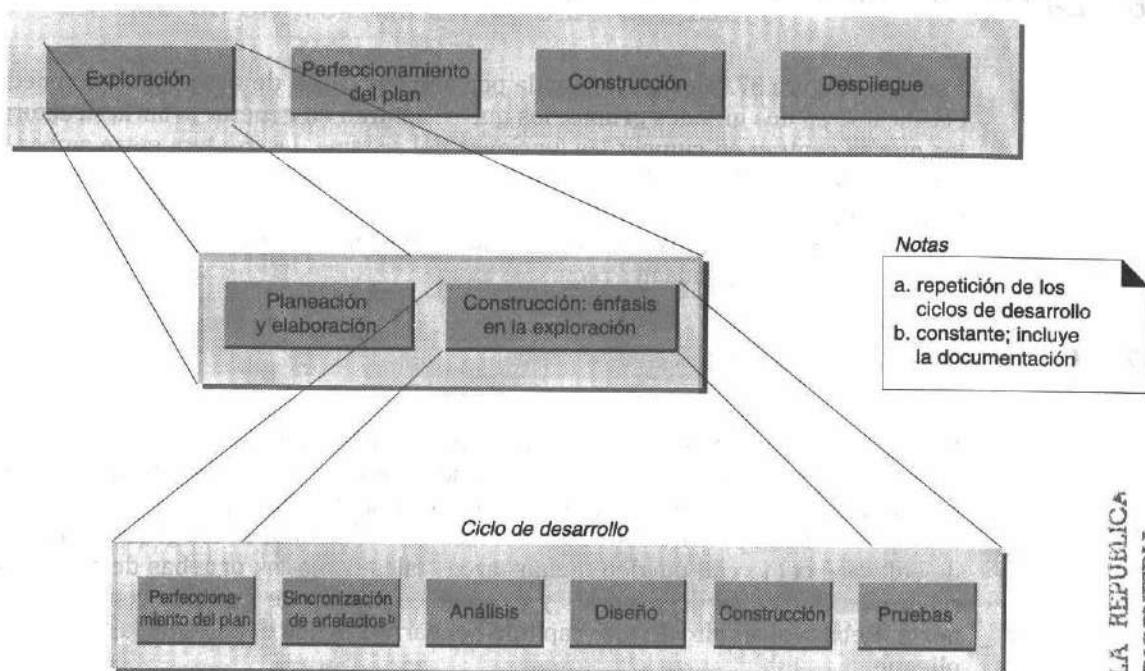


Figura 37.6 Actividades del ciclo de desarrollo.

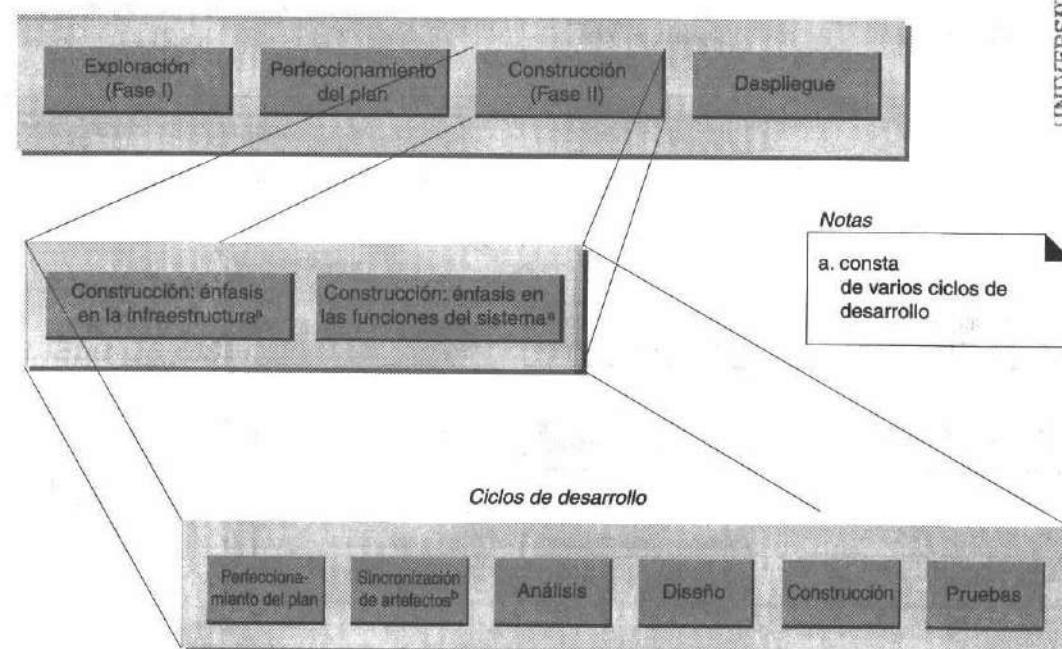


Figura 37.7 La fase de construcción consta de varios ciclos de desarrollo.

37.7.6 La fase primaria de construcción

Esta fase (figura 37.7) está constituida por muchos ciclos de desarrollo que pueden clasificarse en dos grandes grupos: los que se centran en generar la infraestructura y los que se centran en cumplir las funciones del sistema. La frontera entre ambos no está bien definida: durante los ciclos de desarrollo orientados a la infraestructura, uno o más equipos trabajarán en las funciones del sistema y a la inversa. Con esta precisión deseamos poner de relieve que, en muchos proyectos, el esfuerzo dedicado a sentar las bases requiere abundantes recursos.

37.7.7 La fase de despliegue

Esta fase (figura 37.8) consiste en introducir el sistema en la etapa de producción. Si se trata de software comercial, ello significa que se venderá a los clientes para que lo utilicen; si se trata de software para la empresa, significa que formará parte del uso operacional. Las actividades de esta fase varían mucho; en los grandes sistemas de software comercial pueden incluir tareas tales como las pruebas de aceptación realizadas por miles de usuarios y el establecimiento de grandes centros de soporte. Rebasa el ámbito de este capítulo ocuparnos de los detalles de la fase de despliegue.

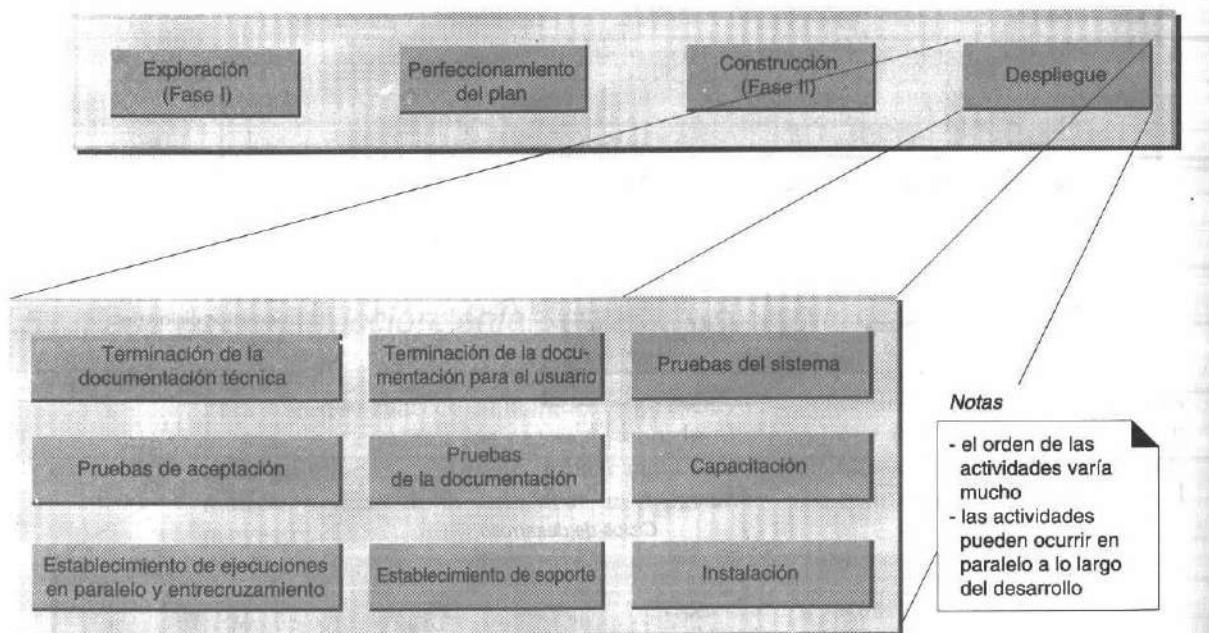


Figura 37.8 La fase de despliegue.

37.7.8 La fase de análisis de un ciclo de desarrollo

Esta fase (figura 37.9) se centra en una investigación del problema y en el análisis de los requerimientos. Ya la explicamos ampliamente en capítulos anteriores.

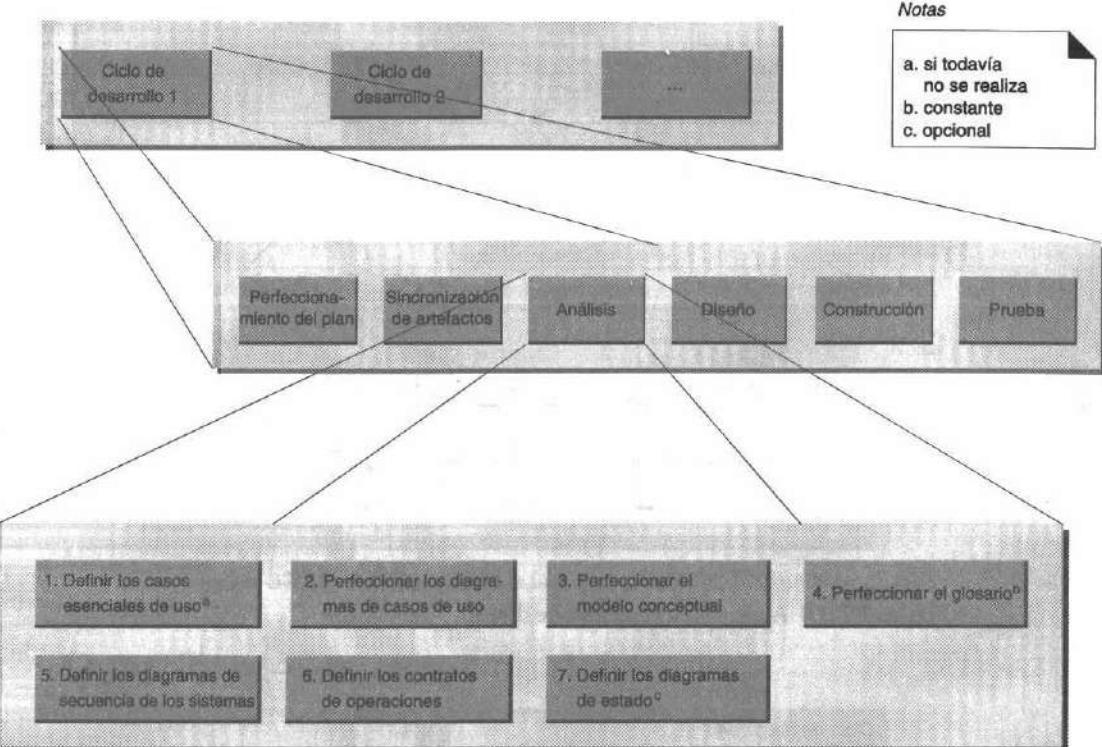


Figura 37.9 Actividades de la fase de análisis.

37.7.9 La fase de diseño de un ciclo de desarrollo

En esta fase se busca ante todo definir una solución lógica (figura 37.10). Ya la estudiamos detenidamente en capítulos anteriores.

37.7.10 La fase de construcción de un ciclo de desarrollo

Esta fase requiere implementar el diseño en software y en hardware (figura 37.11).

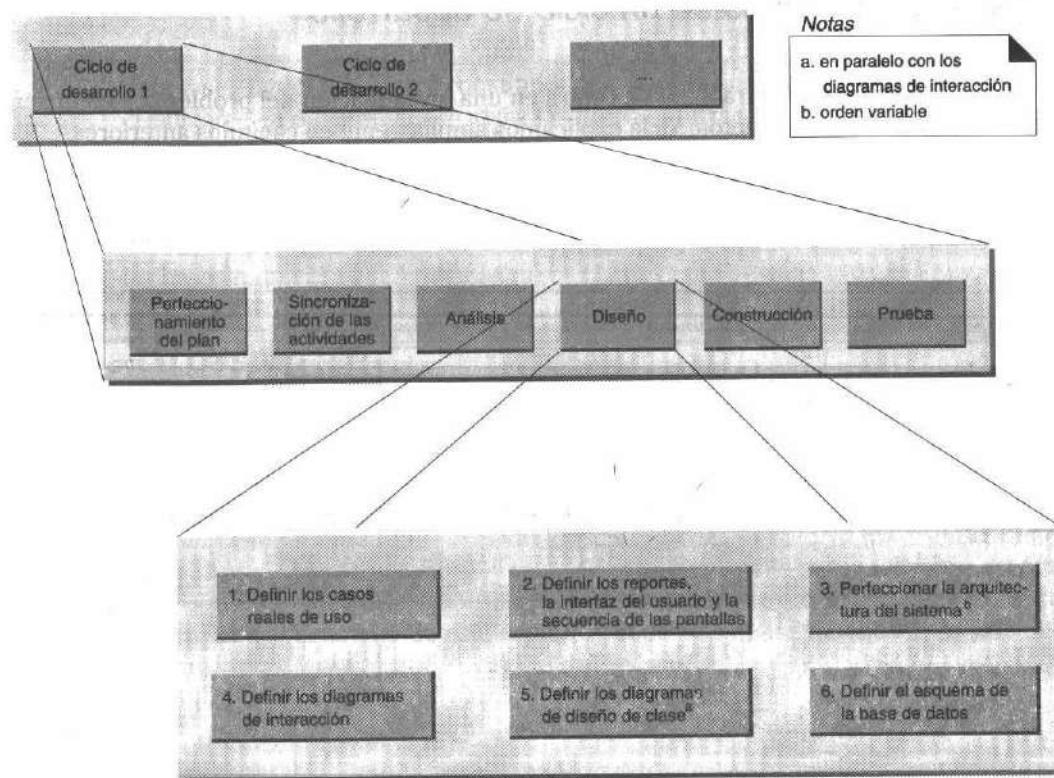


Figura 37.10 Actividades de la fase de diseño.

37.7.11 La fase de pruebas

Aunque esta fase se incluye como paso final dentro de un ciclo de desarrollo, también se recomienda como una actividad constante en la fase de construcción. No todas las pruebas mencionadas en la figura 37.12 son adecuadas en los ciclos de desarrollo. Por ejemplo, las pruebas de integración del sistema entero pueden efectuarse con menor frecuencia en cada uno.

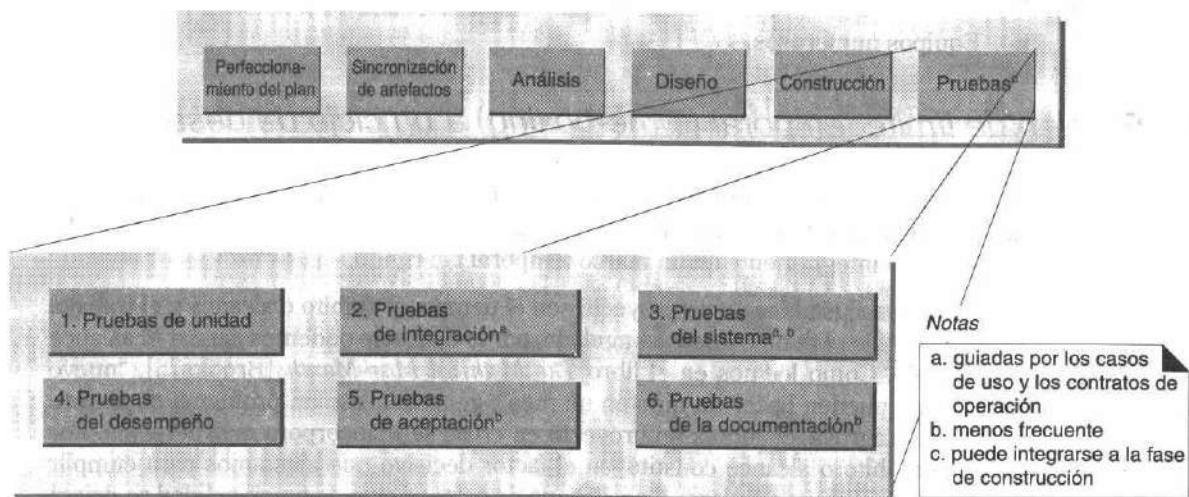


Figura 37.12 Actividades de la fase de realización de pruebas.

37.8 Duración de los ciclos de desarrollo

Con una duración menor de la necesaria será difícil realizar un conjunto significativo de requerimientos y actividades; con una duración mayor se correrá el riesgo de una excesiva complejidad y de insuficiente retroalimentación inicial.

Un ciclo de desarrollo debería durar entre dos semanas y dos meses.

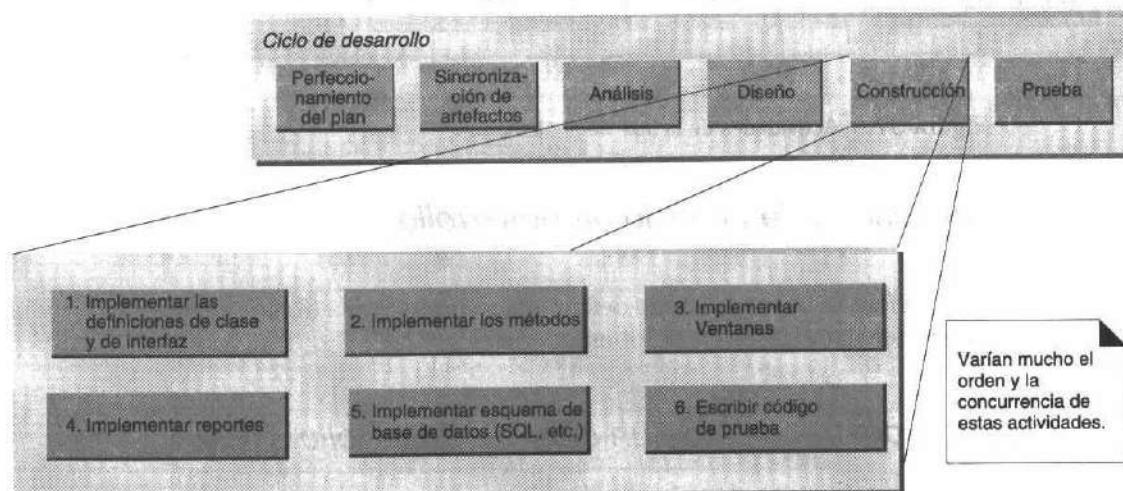


Figura 37.11 Actividades de la fase de construcción.

En igualdad de circunstancias, aconsejamos optar por un ciclo de dos a cuatro semanas. A continuación se mencionan algunos factores que pueden hacer que un ciclo se prolongue hasta durar dos meses:

- Ciclos iniciales de desarrollo en que empiezan a desarrollarse la infraestructura y los servicios básicos, y además se lleva a cabo una abundante investigación exploratoria. Advertencia: muchas veces se subestima el tiempo que se tardará en concluir la infraestructura.
- Introducción de tecnología o de métodos nuevos.

- Inaccesibilidad a los expertos en el dominio del tema.
- Importante número de procesos distribuidos o concurrentes o trabajo de comunicación.
- Personal novato (en la tecnología de objetos, en el dominio del problema o en el desarrollo iterativo).
- Equipos de desarrollo en paralelo.
- Equipos de desarrollo físicamente distribuidos que trabajan en paralelo.¹
- Equipos numerosos.

37.8.1 Fijación de límite temporal (*Time-boxing*) a un ciclo de desarrollo

Una estrategia de gran utilidad en los ciclos de desarrollo consiste en establecer un límite temporal, esto es, un periodo fijo, digamos cuatro semanas. El trabajo debe llevarse a cabo íntegramente en un marco temporal tan rígido.

Tres factores ajustables en un proyecto son el tiempo, el ámbito o alcance y el trabajo. En un plazo fijo el tiempo queda congelado, por lo cual sólo podemos ajustar el alcance y el trabajo. Como leemos en el libro *The Mythical Man-Month* [Brooks75], “nueve mujeres no pueden hacer un niño en un mes”; generalmente un problema no mejora sino que se agrava, cuando a un proyecto en crisis se le incorpora más personal. Por tanto, el ámbito o alcance constituyen el factor decisivo que ajustamos para cumplir con las restricciones del plazo; si se dispone de poco tiempo, la funcionalidad se dejará para un ciclo ulterior de desarrollo.

El equipo de desarrollo habrá de decidir cuáles requerimientos cumplir en el plazo; son ellos los que deben presentar los resultados.

37.9 Aspectos del ciclo de desarrollo

37.9.1 Equipos y ciclos de desarrollo en paralelo

Un proyecto extenso suele dividirse en actividades de desarrollo en paralelo, donde varios equipos trabajan de modo simultáneo (figura 37.13).

Una forma de organizar los equipos en líneas arquitectónicas es hacerlo por capas y subsistemas. Otra forma consiste en basarse en un conjunto de características, que pueden corresponder muy bien a la organización arquitectónica. Por ejemplo:

¹ Tener un equipo en otro piso del mismo edificio produce casi el mismo impacto que si estuviera en un sitio geográficamente distante.

- Equipo de la capa del dominio (o equipo del subsistema del dominio)
- Equipo de interfaz para el usuario
- Equipo de internacionalización
- Equipo de la capa de servicios de alto nivel (equipo de persistencia, equipo de presentación de reportes, etcétera)

Estos equipos pueden trabajar en ciclos de desarrollo en paralelo; todos concluirán un ciclo en la misma fecha.

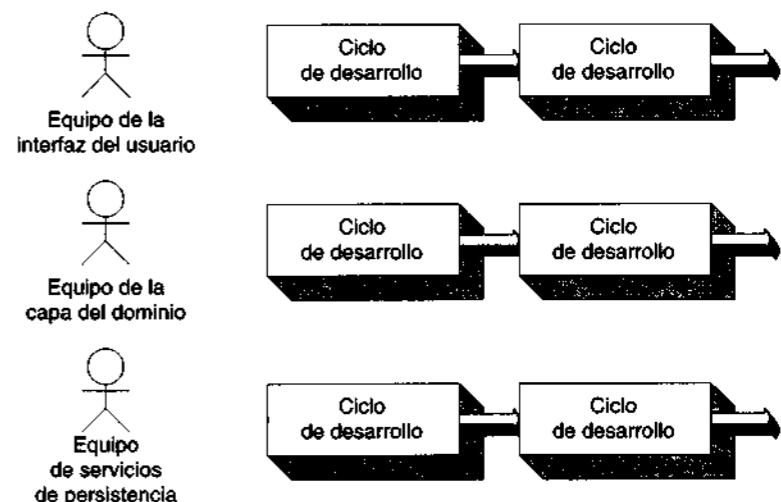


Figura 37.13 Equipos de desarrollo en paralelo.

En capas muy complejas donde el avance siempre es lento al inicio (entre ellas, los servicios de persistencia), se recomienda aumentar la duración del ciclo de desarrollo (figura 37.14). De ese modo, los equipos podrán trabajar en ciclos de duración variable. Todas ellas deberán ser múltiplos del ciclo más corto.

37.9.2 Requerimientos no evidentes y la arquitectura técnica

Algunos requerimientos pueden estar implícitos, o sea que no son explícitamente evidentes en la especificación de requerimientos ni en los casos de uso. Ello sucede sobre todo al diseñar una arquitectura técnica global y servicios de alto nivel, como los de persistencia.

Se requiere mucho trabajo para diseñar e implementar la arquitectura técnica global de un sistema: capas arquitectónicas, despliegue físico, cómputo distribuido, etc. Hay que identificar y programar este trabajo: en la siguiente sección se explica cómo hacerlo.

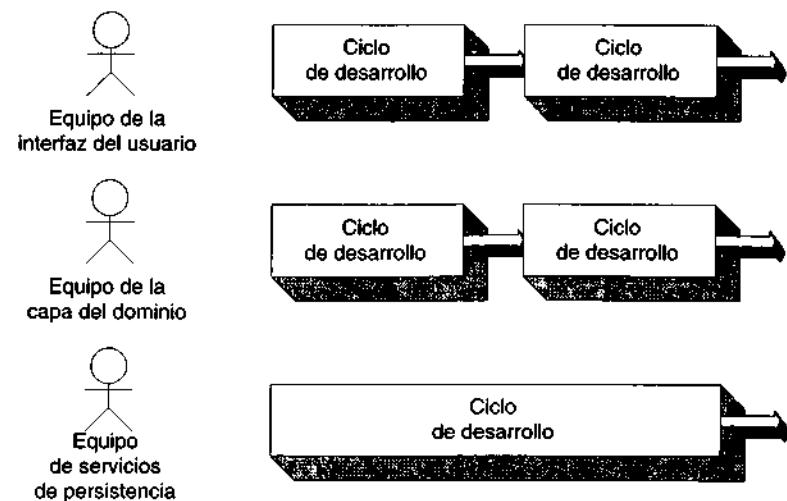


Figura 37.14 Ciclos de desarrollo de duración variable.

37.9.3 Casos de desarrollo de sistemas

Para aclarar los requerimientos técnicos no evidentes de la arquitectura, aconsejamos formular **casos de desarrollo de sistemas**, una especie de casos de uso que describen el trabajo de diseño y las metas funcionales de la actividad en cuestión.¹ Luego pueden programarse los casos como los que están orientados al dominio.

He aquí algunos casos de desarrollo de sistemas:

- Diseñar las capas arquitectónicas y los subsistemas.
- Desarrollar un servicio de persistencia.

De hecho, a todas las actividades relacionadas con el desarrollo y el mantenimiento del software podemos considerarlas casos del desarrollo de sistemas y modelarlas como casos de uso normales de procesos de un dominio, desarrollándolos después diagramas de casos del desarrollo del sistema e indicando, las relaciones *usa*, etcétera (figura 37.5).

En consecuencia, podemos considerar un caso de desarrollo de sistemas, como un caso de uso, y asignarlo a los ciclos de desarrollo.

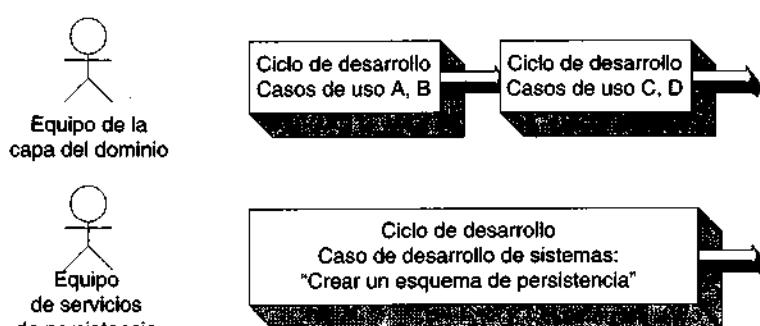


Figura 37.15 Asignación de un caso de desarrollo de sistemas a un ciclo de desarrollo.

37.9.4 Dependencias del desarrollo en paralelo

Cuando varios equipos trabajan en un proyecto, se dan dependencias entre ellos, a saber:

- La interfaz del usuario se basa en la capa de dominio.
- La capa de dominio se basa en las de servicios, como la persistencia.
- El equipo de pruebas (si existe) necesita que algo haya sido terminado y sea lo bastante estable para poder probarlo.

En esta situación el problema radica en que un equipo quiere utilizar el resultado proveniente de otro, aunque no esté terminado. Entre las estrategias para manejar estas dependencias, figuran las siguientes:

- *Talones*: las llamadas a servicios incompletos se implementan como talones de hacer nada o hacer lo mínimo posible. Al irse realizando los servicios, se remplazan los talones.
- *Ciclos escalonados de desarrollo*. Algunos equipos operan sobre un ciclo que se relaciona con el trabajo terminado de un ciclo anterior de otro equipo. Al equipo de la interfaz del usuario le resulta muy cómodo aprovechar los resultados de un ciclo anterior de desarrollo efectuado por un equipo de la capa del dominio (figura 37.16).

¹ Se relacionan con los bloques de procesos, de procedimientos y actividades en el modelo de casos de desarrollo propuesto por Jacobson [Jacobson92].

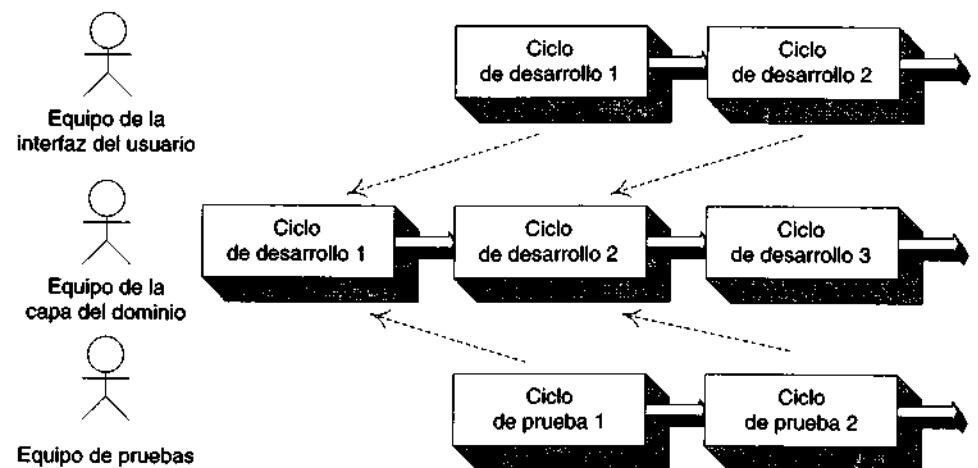


Figura 37.16 Ciclos escalonados dependientes de desarrollo y de pruebas.

37.10 Programación del desarrollo de las capas arquitectónicas

Las capas normales de un sistema son el dominio, el acceso a la base de datos (persistencia), los servicios de alto nivel y la interfaz del usuario. No existe una regla infalible sobre las capas donde hay que centrarse al iniciar el desarrollo, pero sí hay algunas cuestiones a considerar:

- Por lo regular la interfaz del usuario es la concretización visible del sistema. Es lo que ven personas ajenas al desarrollo (directivos y clientes) y que para ellas constituye el sistema. En el ser humano predomina el sentido de la vista. Casi siempre una interfaz atractiva le procurará satisfacción al cliente, aun cuando no sea muy funcional. De ahí la conveniencia de comenzar el proyecto con una interfaz atractiva para el usuario. La desventaja consiste en que los clientes, al verla, creen que el sistema está casi concluido, cuando en realidad apenas si acabamos de iniciarla.
- En un ciclo de desarrollo, si un equipo está desarrollando la capa del dominio y de los servicios, aconsejamos comenzar con la del dominio. Terminela hasta que se cumplan todas las funciones relacionadas con los casos de la iteración actual. Defina como talones para no hacer nada la capa que brinde soporte a los servicios de alto nivel (los de persistencia o de presentación de reportes, por ejemplo). Por último, implemente estos servicios etiquetados.
- Incorpore el soporte de persistencia (base de datos) al inicio del desarrollo, pero no inmediatamente.
 - Tal vez se requieran demasiados recursos para desarrollar los servicios de persistencia y el soporte a la base de datos. Si abordamos prematuramente esta tarea, tal vez ya no prestemos mucha atención a una capa de dominio bien diseñada. Por desgracia, en la tecnología actual

posponer la tarea suele favorecer el ajuste regresivo o modificaciones de diseño por la interacción de metadatos, el esquema de la base de datos, el diseño de jerarquía de clases, la administración de las transacciones, etcétera.

- Cuando se desarrollan los servicios de persistencia, se genera un esquema mínimo de base de datos y también el marco necesario de persistencia para continuar la iteración actual. Junto con la realización de los procesos de dominio, se mejorarán de modo incremental los servicios de persistencia a lo largo de varios ciclos de desarrollo.

ESQUEMAS, PATRONES Y PERSISTENCIA

Objetivos

- Definir lo que es un esquema (framework).
- Aplicar el patrón más fundamental de esquema de la Pandilla de los Cuatro: el método de plantilla.
- Aplicar otros usos de patrones en los esquemas de persistencia, entre ellos la Instanciación de Objetos Complejos.
- Aplicar otros patrones de la Pandilla de los Cuatro, especialmente Agente Virtual.

38.1 Introducción

En la generalidad de las aplicaciones es necesario guardar y recuperar la información en un mecanismo de almacenamiento persistente, una base de datos relacional por ejemplo. Este capítulo versa sobre un diseño orientado a objetos de un esquema o estructura para objetos persistentes que deben guardarse en un almacenamiento persistente.

No vamos a explicar el diseño de un esquema de persistencia de calidad industrial: el problema de los esquemas de persistencia nos servirá para exponer las cuestiones generales concernientes al diseño de esquemas, así como otros aspectos fundamentales de los servicios de persistencia. También explicaremos la manera de emplear el lenguaje UML para comunicar un diseño.

En este libro no se abordan todos los temas del diseño; las figuras contienen comentarios donde se profundizan aspectos que no podrían expresarse tan claramente por otros medios.

38.2 El problema: objetos persistentes

Supongamos que, en la aplicación del punto de venta, muchas instancias de *EspecificacioneProducto* residieran en algún mecanismo de almacenamiento persistente —una base de datos por ejemplo—, las cuales han de ser introducidas en la memoria local durante la aplicación.

En la mayoría de las aplicaciones hay que guardar datos y recuperarlos de un mecanismo de almacenamiento persistente, digamos una base de datos relacional u orientada a objetos. Los **objetos persistentes** son aquellos que requieren almacenamiento persistente, entre ellos las instancias *EspecificacioneProducto*.

El tema de este capítulo es la manera de diseñar servicios de objetos persistentes, porque es un problema común y por ser un instrumento que ayuda a aprender más patrones y los principios del diseño orientado a objetos.

38.2.1 Mecanismos de almacenamiento y los objetos persistentes

Bases de datos orientadas a objetos. Si con una de ellas se almacenan y recuperan objetos, no se necesitan más servicios específicos de persistencia ni de terceros. Éste es uno de los atractivos de su uso.

Bases de datos relacionales. Dado el predominio de este tipo de bases de datos, a menudo se requiere su uso en vez de otras bases más manejables orientadas a objetos. De ser así, surgen varios problemas a causa de la desigualdad entre las representaciones de datos orientadas a registros y las que se orientan a objetos; estos problemas los abordaremos más adelante. Se requieren servicios especiales de ambos tipos en las bases de datos relacionales.

Otros. Además de las bases relacionales de datos, a veces conviene guardar los objetos en otros mecanismos de almacenamiento: archivos planos, bases de datos jerárquicas, etc. Igual que con las bases de datos relacionales, existe divergencia representacional entre las formas orientadas a objetos y el resto de las formas en que se almacenan en esos mecanismos. Como en el caso de las bases de datos relacionales, se necesitan servicios especiales para lograr que funcionen con objetos.

38.3 La solución: un esquema de persistencia

Un **esquema o estructura de persistencia** es un conjunto reutilizable —y, generalmente, expansible— de clases que prestan servicios a los objetos persistentes. Suelen escribirse un esquema de persistencia para trabajar con bases de datos relacionales o un API común de servicios de datos orientados a registros; por ejemplo, ODBC de Microsoft. Por lo regular, un esquema de persistencia debe traducir los objetos a registros y guardarlos en una base de datos; después traduce los registros a objetos cuando los recupera de una base (figura 38.1).

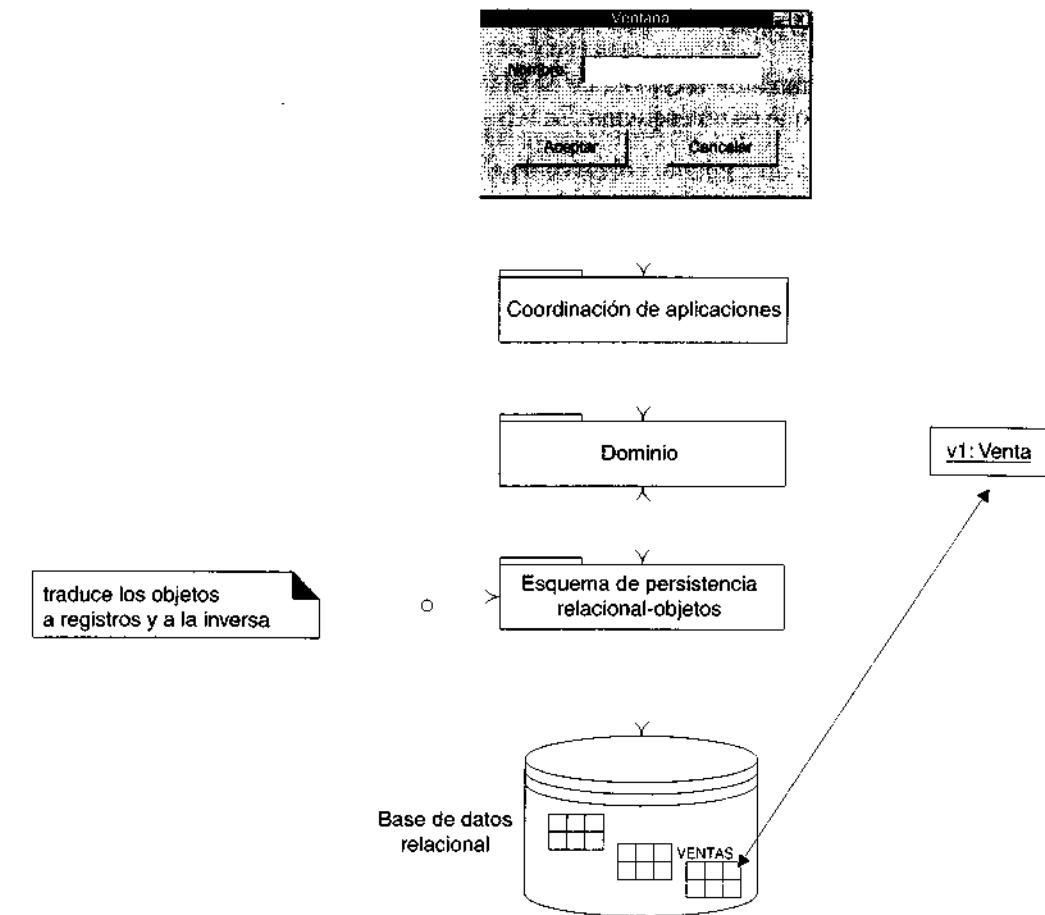


Figura 38.1 Esquema de persistencia relacional-objetos.

Si se emplea una base de datos orientada a objetos, no hace falta un esquema de persistencia. Pero si se utiliza otro mecanismo de almacenamiento, se recomienda servirse de ese esquema; por ejemplo, es de gran utilidad en las bases de datos relacionales y en los archivos planos. En este capítulo vamos a examinar el diseño de un esquema de persistencia y los patrones aplicables a él.

38.4 ¿Qué es un esquema (framework)?

A riesgo de simplificar demasiado el concepto, diremos que un esquema es un subsistema expandible de un conjunto de servicios afines; por ejemplo:

- Los esquemas de interfaz gráfica para el usuario (Foundation Classes de Microsoft, Model-View-Controller de Smalltalk-80).

- Los esquemas de persistencia (esto es, los servicios para realizar objetos persistentes).

El ejemplo de esquema de persistencia que damos en este capítulo explicará en concreto qué es un esquema. En términos generales, un **esquema**:

- Es un conjunto cohesivo de clases que colaboran para prestar servicios a la parte fundamental e invariable de un subsistema lógico.
- Contiene clases concretas y, especialmente, abstractas que definen las interfaces a las cuales se conformarán, las interacciones en que participarán y otras invariantes.
- En términos generales, se requiere (aunque no necesariamente) que el usuario defina subclases de las actuales del esquema para que utilice, adapte y amplíe los servicios que le ofrece.
- Posee clases abstractas que pueden incluir métodos abstractos y concretos.
- Se basa en el **Principio de Hollywood**: “*No nos llame, nosotros le llamamos*.” Ello significa que las clases definidas por el usuario (entre ellas, las clases nuevas) recibirán mensajes de las clases previamente definidas del esquema. Suelen ser manejadas al implementar los métodos abstractos de las superclases.

El siguiente ejemplo de esquema de persistencia demostrará los principios anteriores.

38.4.1 Los esquemas son muy reutilizables

Los esquemas presentan una gran reutilización, la cual es mucho mayor que las clases individuales. Por eso, si una empresa quiere (y quién no?) aumentar considerablemente su grado de reuso del software, habrá de dar prioridad a la creación de esquemas.

38.5 Requerimientos del esquema de persistencia

Deseamos contar con un esquema de persistencia que ofrezca servicios a los objetos persistentes. Llamémoslo EP (esquema de persistencia). EP es un esquema simplificado; en esta introducción no estudiaremos un esquema totalmente terminado de calidad industrial, aunque sí mencionaremos algunas áreas que necesitan desarrollo u otro diseño.

En particular, el esquema debería ofrecer las siguientes funciones:

- Guardar y recuperar objetos en un mecanismo de almacenamiento persistente.
- Transacciones del tipo *commit* y *rollback*.

El diseño deberá dar soporte a las siguientes cualidades:

- Poder extenderse para soportar cualquier mecanismo de almacenamiento, como bases de datos relacionales, archivos planos, etcétera.
- Requerir un mínimo de modificación del código actual.
- Facilidad de uso (para el programador).
- Ser muy transparente: existe en el trasfondo sin forzarla.

38.6 ¿Superclase ObjetoPersistente?

Una solución parcial muy común del diseño para obtener la persistencia de los objetos consiste en crear una superclase de utilería abstracta, el *ObjetoPersistente*, que heredan todos los objetos de persistencia (figura 38.2). Es una clase que normalmente define los atributos de persistencia, entre ellos un identificador único de objetos, y también los métodos para guardar en una base de datos.

Es un procedimiento aceptable, sólo que tiene la deficiencia de acoplar firmemente la clase existente con la del *ObjetoPersistente*; las clases del dominio terminan recibiendo la herencia de una clase de utilería.

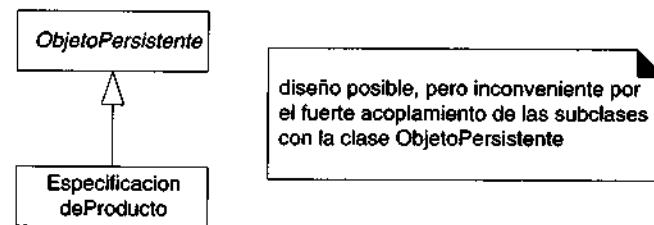


Figura 38.2 Problemas de la superclase *ObjetoPersistente*.

El diseño de esquema persistente examinará medios de evitar una superclase *ObjetoPersistente* y explicará las ventajas y desventajas de varios lenguajes orientados a objetos.

38.7 Ideas básicas

En secciones posteriores estudiaremos las siguientes ideas fundamentales:

- **Mapeo.** Debe haber cierto grado de mapeo entre una clase y su almacenamiento persistente (por ejemplo, la tabla de una base de datos) y entre los atributos del objeto y los campos (columnas) de un registro.
- **Identidad del objeto.** Los registros y los objetos poseen un solo identificador de objeto para relacionar fácilmente los registros con los objetos y para asegurarse de que no haya duplicados inapropiados.

- **Intermediario (broker) de base de datos.** Un agente de base de datos de fabricación pura se encarga de la materialización y de la desmaterialización.
- **Materialización y desmaterialización.** La materialización es el acto de transformar —de un almacenamiento persistente— una representación de datos no orientada a objetos (registros, por ejemplo) a objetos. Desmaterialización es la actividad opuesta (también conocida como pasivación).
- **Caché.** Los intermediarios (brokers) de bases de datos almacenan en un caché los objetos materializados.
- **Materialización lenta por demanda.** No todos los objetos se materializan inmediatamente; una instancia particular sólo se materializa por demanda cuando se necesita.
- **Referencias inteligentes.** La materialización lenta por demanda se implementa empleando una referencia inteligente denominada agente virtual.
- **Objetos complejos.** Se representan y materializan los objetos que poseen estructuras complejas, como las conexiones con muchos otros objetos.
- **Estado de la transacción del objeto.** Conviene conocer el estado de los objetos en cuanto a su relación con la transacción actual. Por ejemplo, es útil saber cuáles objetos han sido modificados (están *sucios*), con el fin de determinar si hay que volver a guardarlos y así actualizarlos en su almacenamiento persistente.
- **Operaciones de transacciones.** Son las operaciones de commit y rollback.
- **Búsqueda.** Son la localización y materialización de los objetos a partir de algunos criterios.

38.8 Mapeo: patrón *Representación de objetos como tablas*

¿Cómo mapear un objeto a un archivo o a un esquema de base de datos relacional?

El patrón **Representación de Objetos como Tablas** [Brown96] propone definir una tabla (si se emplea una base de datos relacional) para cada clase de objeto persistente. Los atributos de objetos que contienen tipos primitivos de datos (número, cadena, booleano y otros) se mapean en las columnas.

Si un objeto posee atributos exclusivamente de tipos primitivos de datos, el mapeo será simple. Pero, como veremos luego, las cosas no son tan sencillas porque los objetos pueden tener atributos que se refieran a otros objetos complejos, mientras que el modelo relacional exige que los valores sean atómicos (primera forma normal).

Venta
fecha
hora
seTermina()
...

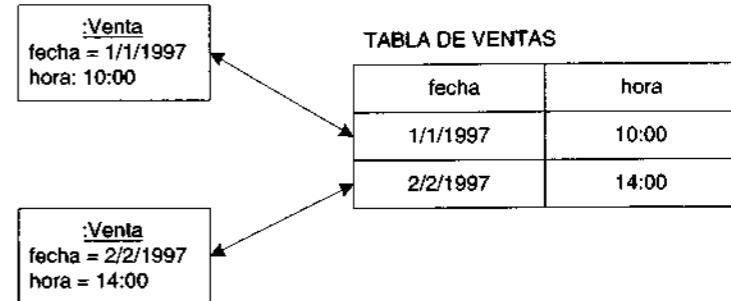


Figura 38.3 Mapeo de objetos y de tablas.

38.9 Identificación de objetos: el patrón *Identificador de objetos*

Conviene contar con un medio que relacione los objetos con los registros y de asegurarse de que la repetición de la materialización de un registro no culmine en la duplicación de objetos.¹

El patrón **Identificador de Objetos** (IDO) [Brown96] se propone asignar un IDO a cada registro y objeto (o al agente de un objeto).

Un identificador de objetos suele ser un valor alfanumérico; es único para un objeto específico. Una elección excelente es el Identificador Universalmente Único² (IUU) de 32 caracteres, que no se repite en el planeta Tierra en ninguna fecha ni hora. En reconocimiento de que nuestro planeta no representa al universo, se le cambió el nombre y se le puso otro correspondiente a la terminología de Microsoft: Identificador Globalmente Único (IGU), y el API Windows de Microsoft contiene una función para generarlo automáticamente.

Toda tabla de base de datos relacional tiene un IDO como clave primaria, y los objetos también contarán (directa o indirectamente) con un identificador.

Si todos los objetos se asocian a un IDO y si todas las tablas poseen una clave primaria IDO, los objetos podrán mapearse de modo singular en un renglón de alguna tabla.

¹ Más adelante examinaremos los problemas de duplicación.

² Según un algoritmo de la Open Software Foundation.

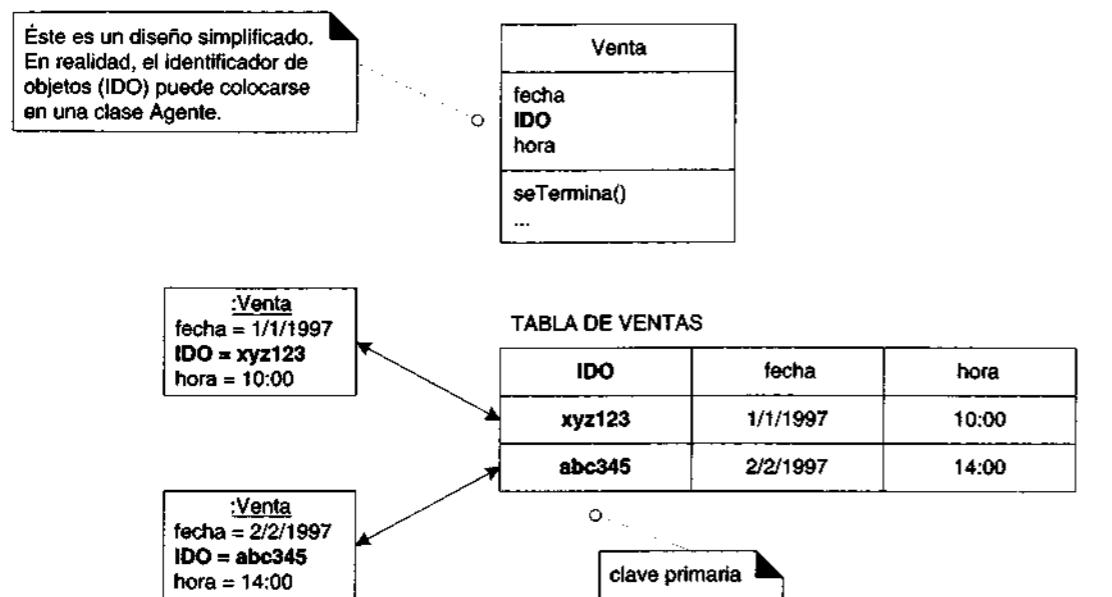


Figura 38.4 Los identificadores de objetos vinculan objetos y registros.

Ésta es una visión simplificada del diseño. En realidad, el IDO no ha sido colocado en el objeto persistente, aunque es posible hacerlo. Por el contrario, podemos ponerlo en un objeto Agente, según veremos más adelante. La elección del lenguaje influye en el diseño.

38.10 Intermediarios: el patrón *Intermediario de base de datos*

¿Quién debería asumir la responsabilidad de materializar y desmaterializar los objetos (por ejemplo, una *EspecificacioneProducto*) a partir de un almacenamiento persistente? El patrón Experto indica que debería hacerlo la clase de objeto persistente (*EspecificacioneProducto*), pero ésta tiene varios defectos, entre otros:

- Acoplar la clase de objeto persistente con el conocimiento de almacenamiento persistente (violación del patrón Bajo Acoplamiento).
- Las responsabilidades complejas en un área nueva y sin relación con las responsabilidades que asumía antes el objeto (violación del patrón Alta Cohesión).

La solución, como siempre que el acoplamiento es grande, está constituida por el patrón Indirección: intercalar algo, generalmente una clase de Fabricación Pura.

El patrón *Intermediario de base de datos* [Brown96] propone construir una clase que se encargue de materializar, de desmaterializar y guardar un objeto en un objeto caché. Podemos definir una clase de intermediarios para cada clase de objetos persisten-

tes.¹ Un Intermediario de base de datos es una clase de Fabricación Pura que soporta los patrones Alta Cohesión y Baja Cohesión por medio de Indirección. La figura 38.5 muestra visualmente que cada objeto persistente puede tener su propia clase de intermediario y que los mecanismos de almacenamiento pueden contar con varias clases de intermediarios.

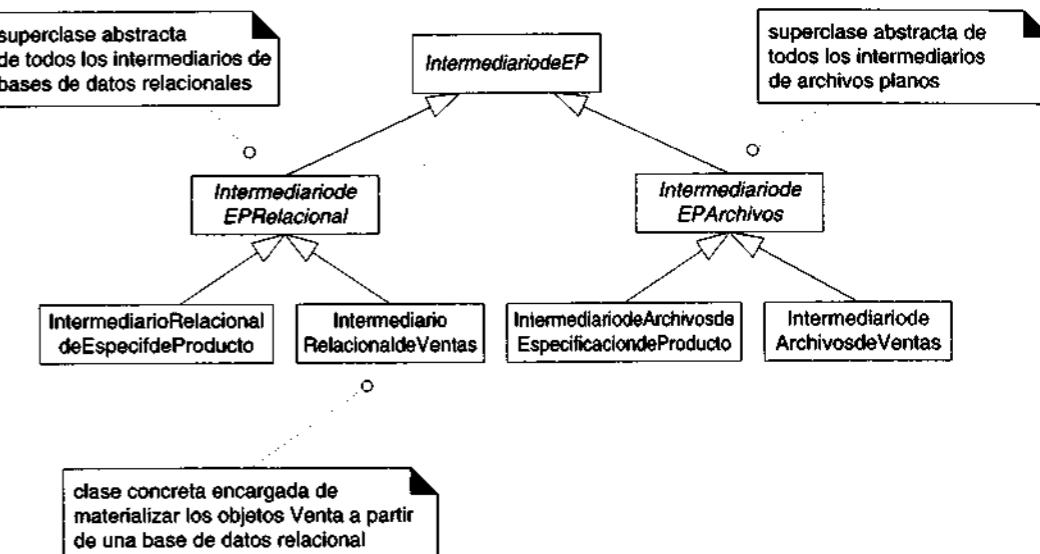


Figura 38.5 Jerarquía de intermediarios de la base de datos.

38.11 Diseño de esquemas: el patrón *Método de Plantilla*

En la siguiente sección describimos algunas de las características esenciales del diseño de Intermediarios de Bases de Datos, que forman parte importante del esquema de persistencia. Esas características se fundan en el patrón *Método de plantilla* de la Pandilla de los Cuatro [GHJV95].

Este patrón constituye la esencia del diseño de esquemas. Con él se define un método (el *Método de Plantilla*) en una superclase que a su vez define el esqueleto de un algoritmo, con sus partes variables e invariables. Este método llama a otros métodos, algunos de los cuales son operaciones que pueden ser omitidas en una subclase.

Por tanto, las subclases pueden desplazar los métodos variables a fin de agregar su comportamiento propio en algunos puntos de la variabilidad.

El patrón *Método de Plantilla* ejemplifica el principio de Hollywood: "No nos llame, nosotros lo llamamos". En la figura 38.6 la clase *ClaseConcreta* desplaza *operacionPrimitiva*. Se la llamará automáticamente cuando se invoque el heredado *metododePlantilla*.

¹ Más adelante mencionaremos otra alternativa que se funda en metadatos y en instancias parametrizadas.

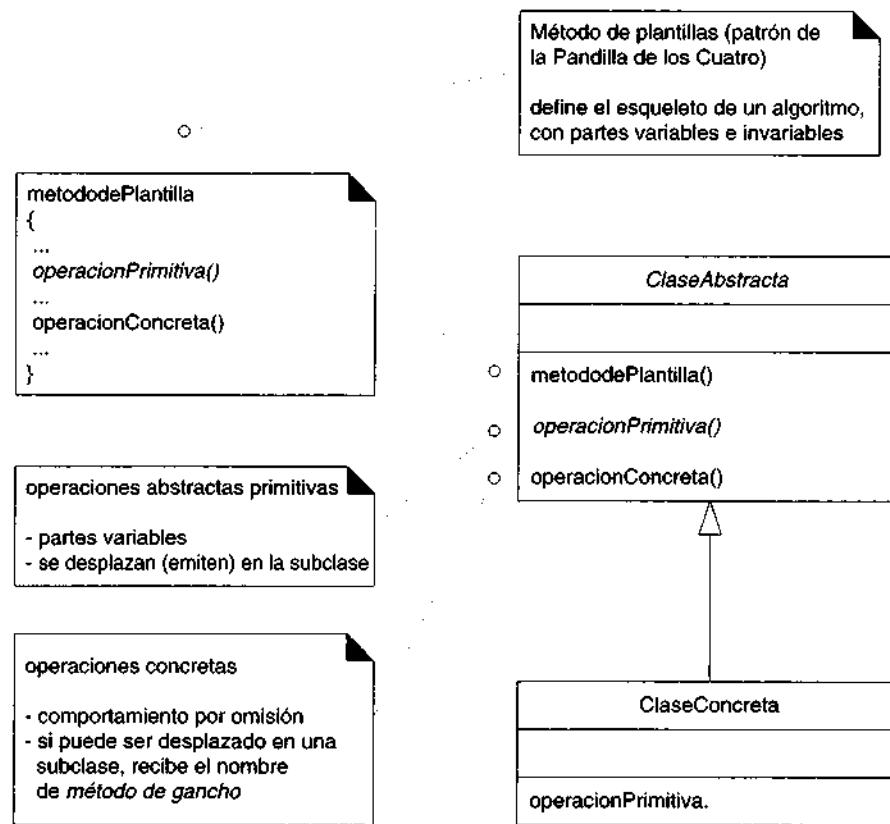


Figura 38.6 Patrón del método de plantillas.

38.12 Materialización: el patrón *Método de Plantillas*

En el esquema de persistencia, los objetos pueden recuperarse invocando el método `objetoCon(unIDO)` que se define en la clase *IntermediarioEP* (*Esquema de Persistencia*). Este método puede, a su vez, invocar a `materializarCon(unIDO)` para materializar el objeto. Como se observa en la figura 38.7, la lógica de la materialización suele requerir que se genere una instancia de la clase apropiada y que luego se desplacen los datos del registro hacia los atributos de la nueva instancia. Por ejemplo, el campo *hora* de un renglón de la tabla *VENTAS* puede copiarse en el atributo *hora* de la instancia asociada *Venta*.

La jerarquía de la clase *Intermediario de Bases de Datos* forma parte esencial del esquema de persistencia; el programador de la aplicación podrá agregar más clases para adoptarlas a nuevos tipos de mecanismos de almacenamiento persistente, a otras tablas o archivos dentro de un mecanismo de almacenamiento ya existente.

Por ejemplo, es posible incorporar una nueva subclase de *IntermediarioEP* e *IntermediarioEPArchivos* para materializar los objetos a partir de archivos planos. Y dentro

de *IntermediarioEPRelacional* es posible agregar la materialización para otras clases, entre ellas una clase *IntermediarioRelacionalVentas*.

Como se muestra en la figura 38.7, el diseño expansible se funda en el patrón *Método de Plantillas*.¹

El método público más importante en *IntermediarioEP* es `objetoCon(unIDO)`, método de plantilla que adopta un identificador como parámetro y luego devuelve el objeto de él. Si ya antes materializamos el objeto y actualmente está en un caché, tan sólo se devuelve. Pero si no estaba en caché, lo materializamos mediante el método `materializarCon(unIDO)`, el cual es una “operación primitiva” del método de plantilla `objetoCon`. Podemos incorporar más subclases que se materialicen en diversas formas.

Notese la naturaleza característica del método de plantillas `objetoCon`. Define partes variables e invariables del esquema. El patrón de consulta del caché y la materialización son invariables, pero los detalles de cómo se realiza la materialización pueden modificarse; se conserva como una operación primitiva que deben definir las subclases.

De modo análogo, `registroActualComoObjeto` es una operación elemental del método de plantilla *IntermediarioEPRelacional::materializarCon*. Un intermediario concreto para cada pareja clase/tabla define cómo transformar en un objeto el actual registro de la base de datos que ha sido leído. En este ejemplo, la transformación aparece bastante sencilla; más tarde examinaremos una lógica más compleja de la materialización.

La jerarquía de *IntermediarioEP* muestra las tres siguientes características clásicas del diseño de esquemas:

1. El uso de los métodos de plantillas en superclases abstractas que se definieron de antemano.
2. La incorporación de las subclases definidas por el programador.
3. La definición de los métodos de “operación primitiva” en las subclases para completar los métodos de plantilla heredados.

¹ Más adelante expondremos una solución alterna que se basa en metadatos.

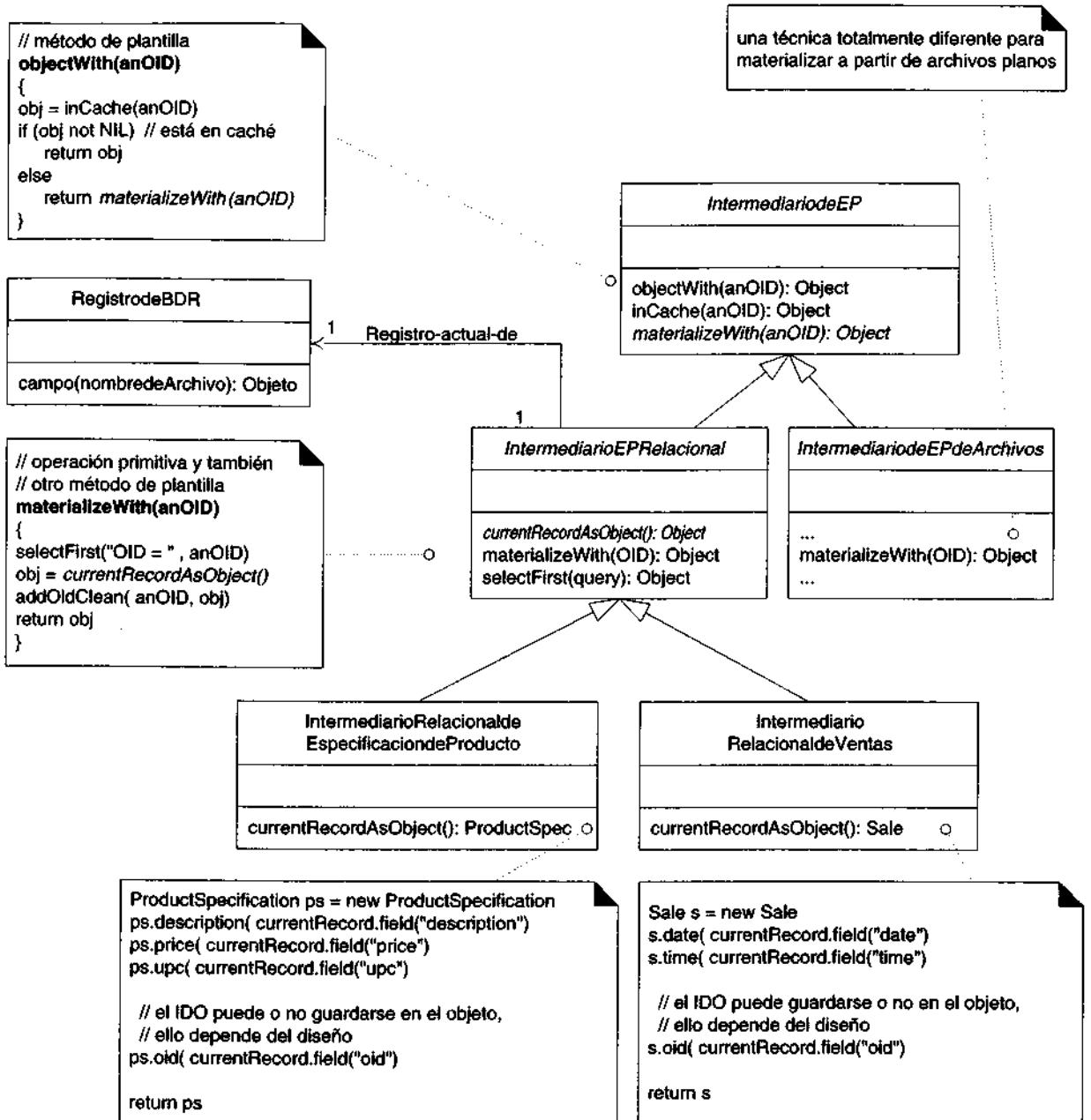


Figura 38.7 Patrón del método de plantilla en la jerarquía del Intermediario de Bases de Datos.

38.13 Objetos colocados en espacio caché: el patrón *Administración de Caché*

Conviene colocar en una caché local los objetos materializados, pues sólo así se podrá mejorar el desempeño (la materialización es relativamente lenta) y brindar soporte a las operaciones de la administración de transacciones, entre ellas la de commit (figura 38.8).

El patrón **Administración de Caché** [Brown96] propone asignar a los intermediarios de bases de datos la responsabilidad de dar mantenimiento a su caché. Si se utiliza un intermediario diferente con cada clase de objeto persistente, el intermediario habrá de darle mantenimiento a su propia caché.

Cuando se materializan los objetos, se colocan en la caché con su identificador como clave. Si después se le pide al intermediario un objeto, primero buscará la caché y con ello evitirá una materialización innecesaria.

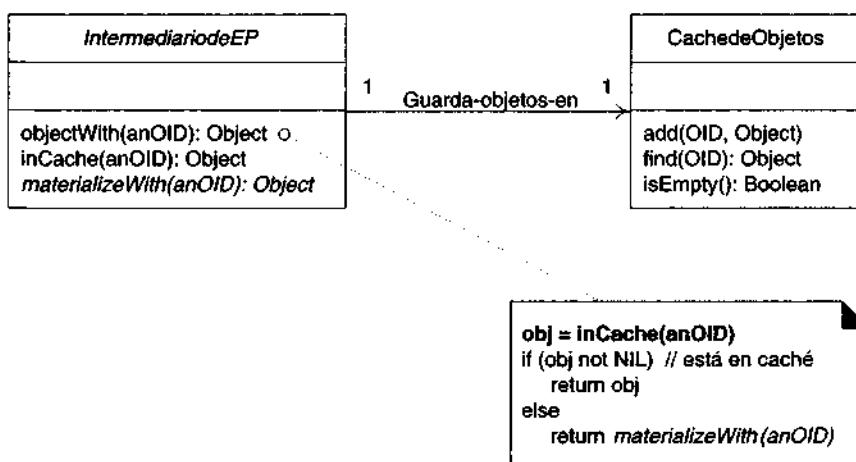


Figura 38.8 Los intermediarios conservan una caché de los objetos materializados.

38.13.1 Cachés para la administración de transacciones

Una variante de una caché individual consiste en conservar los objetos en varias cachés, según el estado que presenten dentro del contexto de la transacción actual (figura 38.9). En este método, el intermediario conserva hasta seis cachés.¹ Se sientan así las bases para determinar cómo realizar un commit y un rollback de las transacciones.

¹ En la generalidad de las aplicaciones se requieren menos de seis porque las reglas de transacción son las mismas en varias cachés. Por ejemplo, las cachés Nueva y Limpia y Nueva y Sucia se tratan igual tanto en un commit (insertar en base de datos) como en un rollback. Más adelante explicaremos una alternativa de las cachés múltiples que se basa en objetos estado.

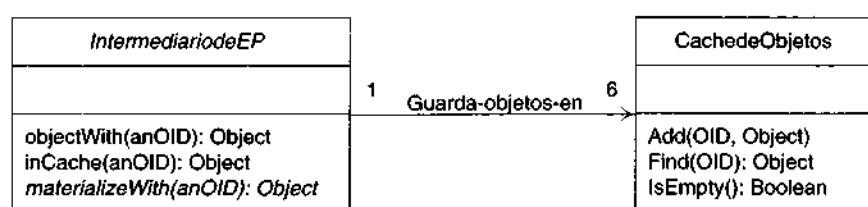


Figura 38.9 Cachés múltiples del intermediario.

Las seis cachés son:

- Caché Limpia y Nueva.** Objetos nuevos, sin modificaciones.
- Caché Limpia y Vieja.** Objetos viejos que se materializan a partir de una base de datos, sin modificaciones.
- Caché sucia y nueva.** Objetos nuevos, modificados.
- Caché sucia y vieja.** Objetos viejos que se materializaron de una base de datos y que fueron modificados.
- Caché Eliminar Nueva.** Objetos nuevos a eliminar.
- Caché Eliminar Vieja.** Objetos viejos que se materializaron a partir de una base de datos y que deben ser eliminados.

En secciones posteriores veremos cómo los objetos se dirigen hacia estas cachés y el razonamiento que se les aplica con las operaciones de commit y de rollback.

Un Agente Virtual es un caso especial del patrón Indirección. Como se advierte en la figura 38.10, un objeto cliente tiene una referencia al objeto Agente Virtual y no al sujeto real, pero el cliente trata al agente como si fuera el sujeto real. El Agente Virtual implementa la misma interfaz que ese sujeto; así que el objeto cliente ve al Agente Virtual como si fuera el sujeto real.

Un Agente Virtual es también una especialización de otro patrón de la Pandilla de los Cuatro: Puente, conocido también con el nombre de Manivela-Cuerpo [GHJV95]. El patrón Puente es también un caso especial del patrón Indirección, donde un objeto intermedio (la *Manivela*) recibe solicitudes de un cliente y las transmite a otro objeto (el *Cuerpo*) para que las atienda. El objeto Manivela suele ser muy sencillo en lo tocante a sus responsabilidades; en términos generales, se limita a transmitirle al Cuerpo las solicitudes. El patrón Puente permite al diseñador desacoplar la implementación en la interfaz.

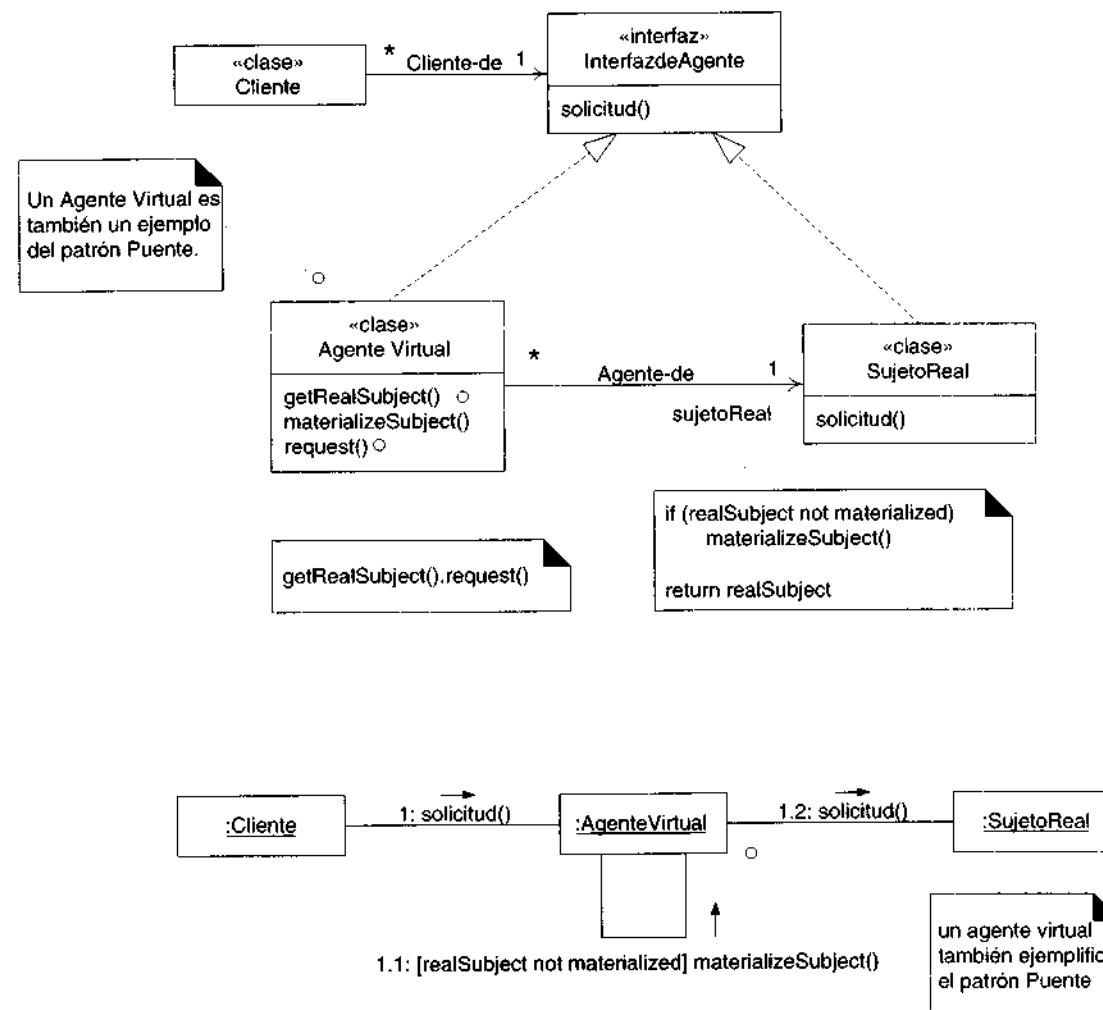


Figura 38.10 Los patrones Agente Virtual y Puente.

38.14 Referencias inteligentes: los patrones Agente Virtual y Puente

Como veremos luego con mayor detalle, en ocasiones conviene posponer la materialización de un objeto hasta que sea absolutamente necesaria. A esto se le llama **materialización lenta o por demanda**. Podemos implementarla por medio del patrón Agente Virtual de la Pandilla de los Cuatro, una de las muchas variantes del patrón Agente [GHJV95]. (En un capítulo anterior examinamos ya Agente Remoto y Agente Dispositivo.)

Un Agente Virtual es un agente que hace referencia inteligente a otro objeto (el *sujeto real*) que materializa este último al referenciarlo por primera vez; implementa, pues, una materialización por demanda. Es un objeto ligero que representa a un objeto “real”, el cual puede materializarse o no. Al Agente Virtual se le considera una referencia inteligente, porque un cliente la trata como una referencia regular al sujeto real.

En la figura 38.11 se da un ejemplo concreto del patrón Agente Virtual con *VentasLineadeProducto* y con *EspecificaciondeProducto*. El diseño se basa en la suposición de que los agentes conocen el identificador de objetos de su sujeto real y de que, cuando se requiere la materialización, el identificador sirve para localizar y recuperar el sujeto real.

Nótese que *VentasLineadeProducto* posee visibilidad de atributo ante una instancia *EspecificaciondeProducto*. Esta instancia de *VentasLineadeProducto* tal vez todavía no se materialice en la memoria. Cuando *VentasLineadeProducto* envía un mensaje *descripción* al agente, materializa la *EspecificaciondeProducto* real, utilizando para ello el identificador de objeto de la *EspecificaciondeProducto* real para identificar al sujeto real.

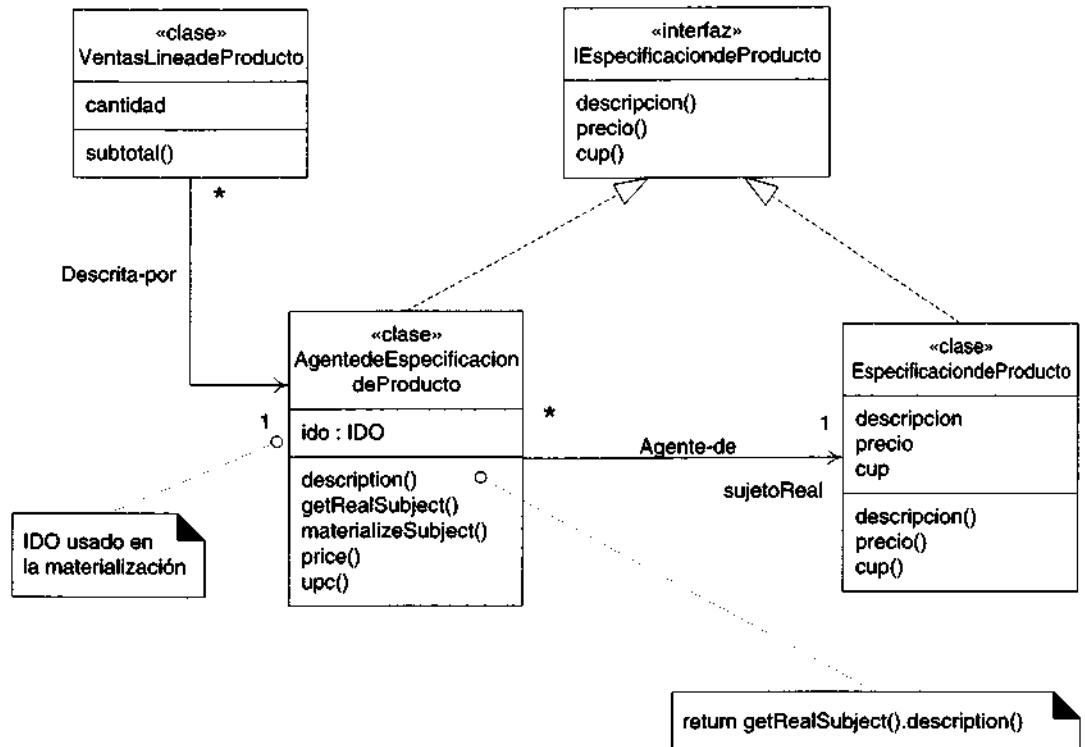


Figura 38.11 Agente Virtual de EspecificaciondeProducto.

38.14.1 Agente Virtual generalizado

Todas las clases de Agente Virtual han de efectuar la materialización por demanda, necesitan conocer su sujeto real y también pueden conocer el identificador de objetos; de ahí la posibilidad de crear una superclase abstracta generalizada *AgenteVirtual* que cumpla estas responsabilidades. Las subclases concretas tan sólo deben dar soporte a la interfaz del sujeto real que han de implementar. Este diseño se muestra visualmente en la figura 38.12.

38.14.2 No se requiere una superclase común ObjetoPersistente

Nótese que, en este diseño, no es necesario conservar el identificador de objetos en el objeto del dominio (por ejemplo, *EspecificaciondeProducto*). Gracias a ello podemos prescindir de los objetos del dominio para recibir la herencia de una superclase común como *ObjetoPersistente* (figura 38.13).

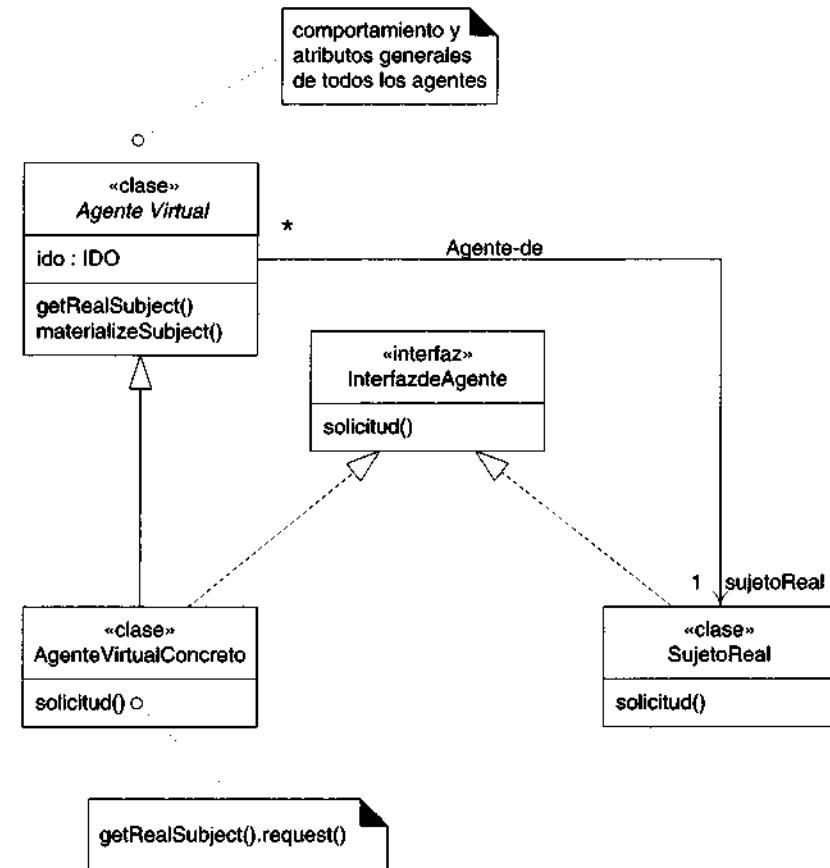


Figura 38.12 La clase generalizada Agente Virtual.

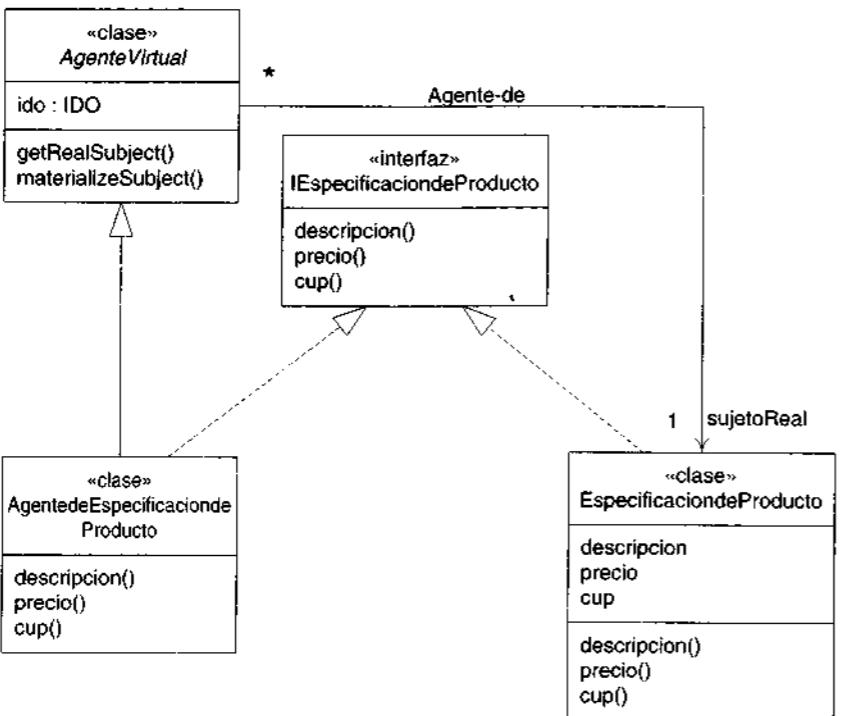


Figura 38.13 La clase AgenteedeEspecificaciondeProducto.

38.14.3 Implementación de un Agente Virtual

La implementación de un Agente Virtual varía según el lenguaje que se utilice. Los detalles de la implementación rebasan el ámbito de este capítulo; por ello nos limitamos a ofrecer una sinopsis:

Lenguaje	Implementación de Agente Virtual
C++	Definir una clase de apuntadores inteligentes con plantilla. No se define la clase abstracta AgenteVirtual ni las clases AgenteVirtual/Concreto, pero el apuntador inteligente realiza su comportamiento. No se requiere definir InterfazdeAgente.

Java	Se implementa la clase AgenteVirtual. Se implementan las clases AgenteVirtual/Concreto. Se definen las interfaces InterfazdeAgente.
Smalltalk	<ol style="list-style-type: none"> 1. Se implementa la clase AgenteVirtual. Se implementan las clases AgenteVirtual/Concreto. No se requieren InterfacesdeAgente debido al vínculo dinámico de Smalltalk. 2. Definir un Agente virtual cambiante (o Agente Fantasma), que utiliza #doesNotUnderstand: y #become: para cambiar al sujeto real. No se definen la clase abstracta AgenteVirtual ni las clases AgenteVirtual/Concreto. No se requiere la definición de InterfazdeAgente.

38.15 Agentes Virtuales e Intermediarios de Bases de Datos

Ahora que ya sentamos las bases de nuestra exposición, vamos a describir los detalles del esquema de persistencia. Un Agente Virtual puede colaborar con un Intermediario de Bases de Datos a fin de materializar un objeto, partiendo para ello del identificador de objetos conservado por el agente (figura 38.14).

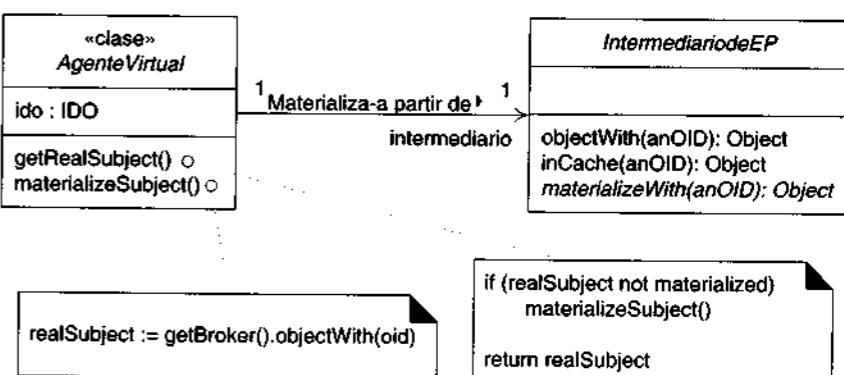


Figura 38.14 Colaboración entre Agente Virtual e intermediario de Base de Datos.

38.15.1 Conexión de Agente Virtual e Intermediario de base de datos: el patrón Método Fábrica

¿Cómo un Agente Virtual concreto sabe cuál Intermediario de Base de Datos habrá de utilizar? La respuesta consiste en aplicar el patrón Método Fábrica de la Pan-

dilla de los Cuatro [GHJV95]. El patrón **Método Fábrica** (figura 38.15) es un caso especial del patrón **Método de Plantilla** en que la operación primitiva (el **Método Fábrica**) se encarga de crear una instancia. Este método, además de su patrón más general **Método de Plantilla**, es muy común en el diseño de esquemas. El **Método Fábrica** *crearIntermediario* se sirve del patrón **Singleton** para devolver un **Intermediario** de Base de Datos, porque conviene tener una sola instancia de cada intermediario.

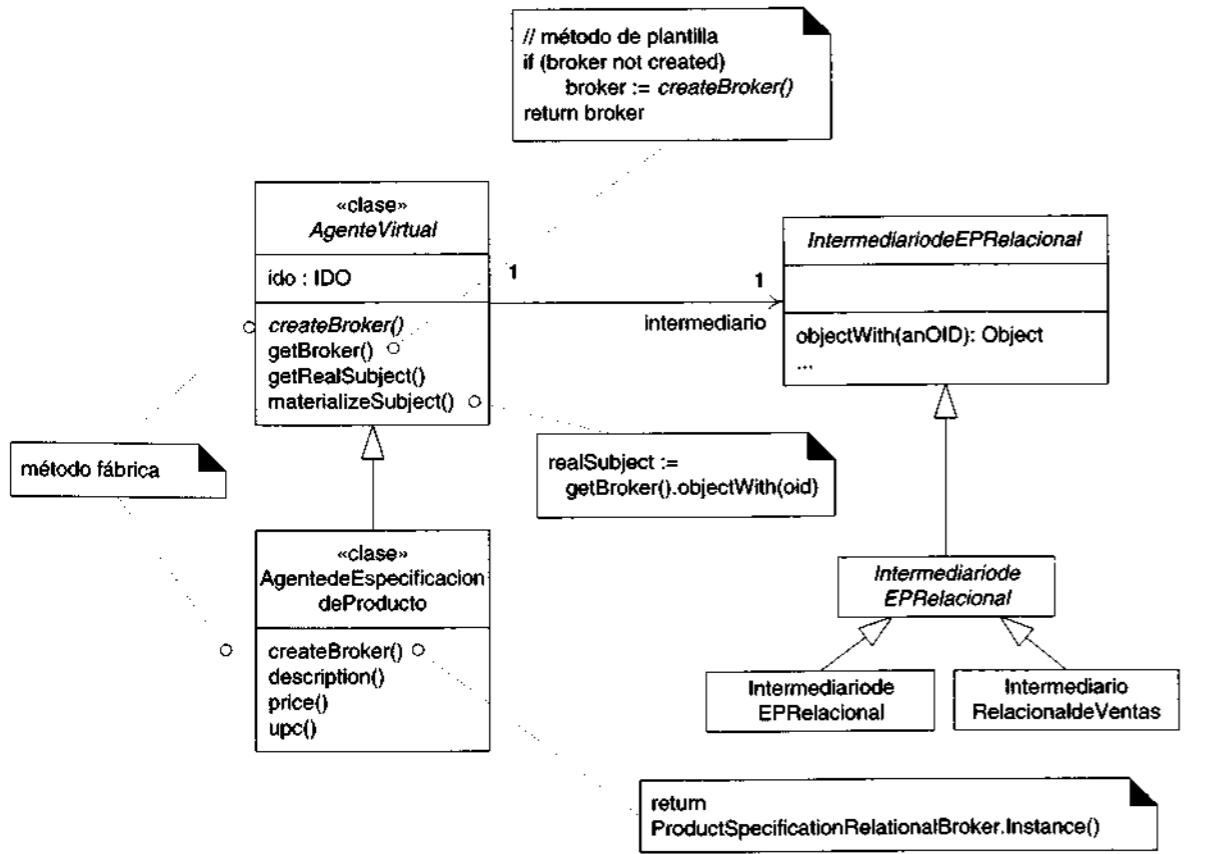


Figura 38.15 El patrón Método Fábrica.

Ahora podemos ver un tema que se repite varias veces en el diseño de esquemas: las clases *IntermediarioEP* y *AgenteVirtual* forman parte del esquema y comparten algunas cualidades con ellos. Por ejemplo:

- Las superclases abstractas pueden ser subclases para el usuario de los esquemas.
- Se definen los métodos de plantilla cuyas operaciones primitivas define el usuario de esquemas en una subclase.

38.15.2 Cómo convertir todo en un agente

Cuando se usan agentes virtuales, conviene que *toda* referencia a los objetos se efectúe a través de los objetos agente y no a través de las referencias directas. Ello significa que:

- Todas las definiciones de atributos se refieren a objetos agente a interfaces, no a objetos directos.
- Todos los parámetros se refieren a objetos agente o a interfaces.

Por ejemplo, si va a usarse el lenguaje C++ y si los agentes se implementan con apunadores inteligentes provistos de plantillas, todos los miembros de datos y las declaraciones de parámetros estarán destinados a los apunadores inteligentes, no a los objetos persistentes. En el caso de C++, esta técnica también puede servir para sopportar la administración automática de la memoria; los agentes están en condiciones de administrarla.

Si va a usarse el lenguaje Java, todas las declaraciones de los campos de instancias y de parámetros se referirán a las interfaces (cosa que aconsejamos en todos los casos por su flexibilidad).

Esta utilización constante de los agentes garantiza que la materialización por demanda y las operaciones de transacciones se lleven a cabo correctamente.

38.16 Cómo Representar las relaciones en tablas

En la siguiente sección explicaremos cómo materializar los objetos complejos: los que se conectan a otros en vez de limitarse a conservar atributos primitivos simples. A manera de preparación para esa exposición, es necesario examinar las formas en que podemos definir las relaciones de objetos en un almacenamiento persistente. Emplearemos tablas relacionales en nuestra exposición, aunque se dispone de esquemas para otros mecanismos de almacenamiento.

¿Cómo representar las relaciones de objetos en una tabla de una base de datos relacional? La respuesta se da en el patrón **Representación de las relaciones de objetos como Tablas** [Brown96] que propone lo siguiente:

- Asociaciones **uno a uno**.
 - Colocar una clave foránea del identificador de objetos en una o en las dos tablas que representan los objetos en la relación.
 - Crear una tabla asociativa que registre los identificadores de objetos de cada objeto en la relación.
- Asociaciones de **uno a muchos**, una colección por ejemplo.
 - Crear una tabla asociativa que registre los identificadores de cada objeto en la relación.

■ Asociaciones de muchos a muchos.

- Crear una tabla asociativa que registre todos los identificadores de objetos en la relación.

Utilizaremos siempre una tabla asociativa para simplificar nuestra exposición.

38.17 El patrón *Instanciación de Objetos Complejos*

38.17.1 Problema: materializar una jerarquía de composición

Los objetos Agente Virtual e Intermediario de Base de Datos nos dan los medios para lograr la materialización por demanda. Entonces nos preguntamos: ¿por qué molestarnos? Lo hacemos porque los objetos pueden contener conexiones con otros, formando así una jerarquía de composición. Supongamos, por ejemplo, que los objetos *Venta*, *VentasLineadeProducto* y *Especificacione de Producto* son persistentes. ¿Qué significa materializar una *Venta*? ¿Significa acaso que se materializa únicamente la *Venta* o también su *VentasLineadeProducto* y su *Especificacione de Producto* asociada? Si queremos materializar un objeto con una profunda jerarquía de composición, posiblemente también haya que materializar docenas o cientos de objetos relacionados. La materialización de una jerarquía íntegra de composición usa el espacio lento e inefficientemente.

38.17.2 Solución: materialización por demanda

En consecuencia, la solución es el patrón **Instanciación de Objetos Complejos** [Brown96], el cual propone aplazar la instancia (materialización) de los objetos dependiendo de los patrones de acceso y de los requerimientos del desempeño. El caso extremo de este patrón —que se incluye en el presente capítulo para simplificar la exposición— es una materialización total por demanda: diferir la materialización de los objetos hasta que sea necesario. En muchas aplicaciones se recomienda adoptar una solución intermedia, en que una jerarquía de composición se materializa uno o dos niveles de profundidad. Con un intermediario distinto para cada objeto persistente, es posible decidir, intermediario por intermediario, el grado de materialización de los objetos persistentes y de sus objetos asociados.

38.17.3 Ejemplo: materialización de una instancia *VentasLineadeProducto*

Para entender la complejidad con que funciona la instanciación de objetos dentro del contexto del esquema persistente, examinaremos el caso de materializar la instancia *VentasLineadeProducto*. Supongamos que ella y su información relacionada se guardan en tablas relacionales, como se indica en la figura 38.16.

VENTASLINEADEPRODUCTO

IDO	cantidad
vli1	1
vli2	2

ESPECIFDEPRODUCTOS

IDO	descripcion	precio	cup
p1	pañuelos desechables	1.50	111
p2	tempeh	2.25	222

VENTASLINEADEPRODUCTO-A-ESPECIFDEPRODUCTOS

VLI-IDO	EP-IDO
vli1	p1
vli2	p2

Figura 38.16 Tabla de VentasLineadeProducto y su información relacionada.

Supongamos además que sabemos que el identificador de objetos de *VentasLineadeProducto* es “vli1”.¹ Veamos lo que ocurre si ejecutamos el siguiente código:

```
//crear el agente
AgenteVentasLineadeProducto unVLI =
    new AgenteVentasLineadeProducto("vli1");

//causa materializacion de los objetos
int total = unVLI.subtotal();
```

La colaboración se muestra de manera gráfica a partir de la figura 38.17. Obsérvese que la proposición simple *unVLI.subtotal()* causa una importante materialización de objetos que no se observa directamente. Un paso decisivo se encuentra dentro del método *RegistroactualComoObjeto* de *IntermediarioRelacionaldeVentasLineadeProducto*. Una vez creada la instancia *VentasLineadeProducto*, el identificador de objetos de la *Especificacione de Productos* asociada se determina leyendo en la tabla asociativa.² En vez de materializar *Especificacione de Productos*, creamos un *Agente de Especificacione de Productos* y lo asociamos a *VentasLineadeProducto*.

¹ Más adelante veremos cómo se determina el IDO del objeto inicializado para efectuar la materialización.

² Se emplea un seudo SQL para indicar la lectura a partir de otra tabla. Varían mucho los detalles de cómo se realiza esa lectura.

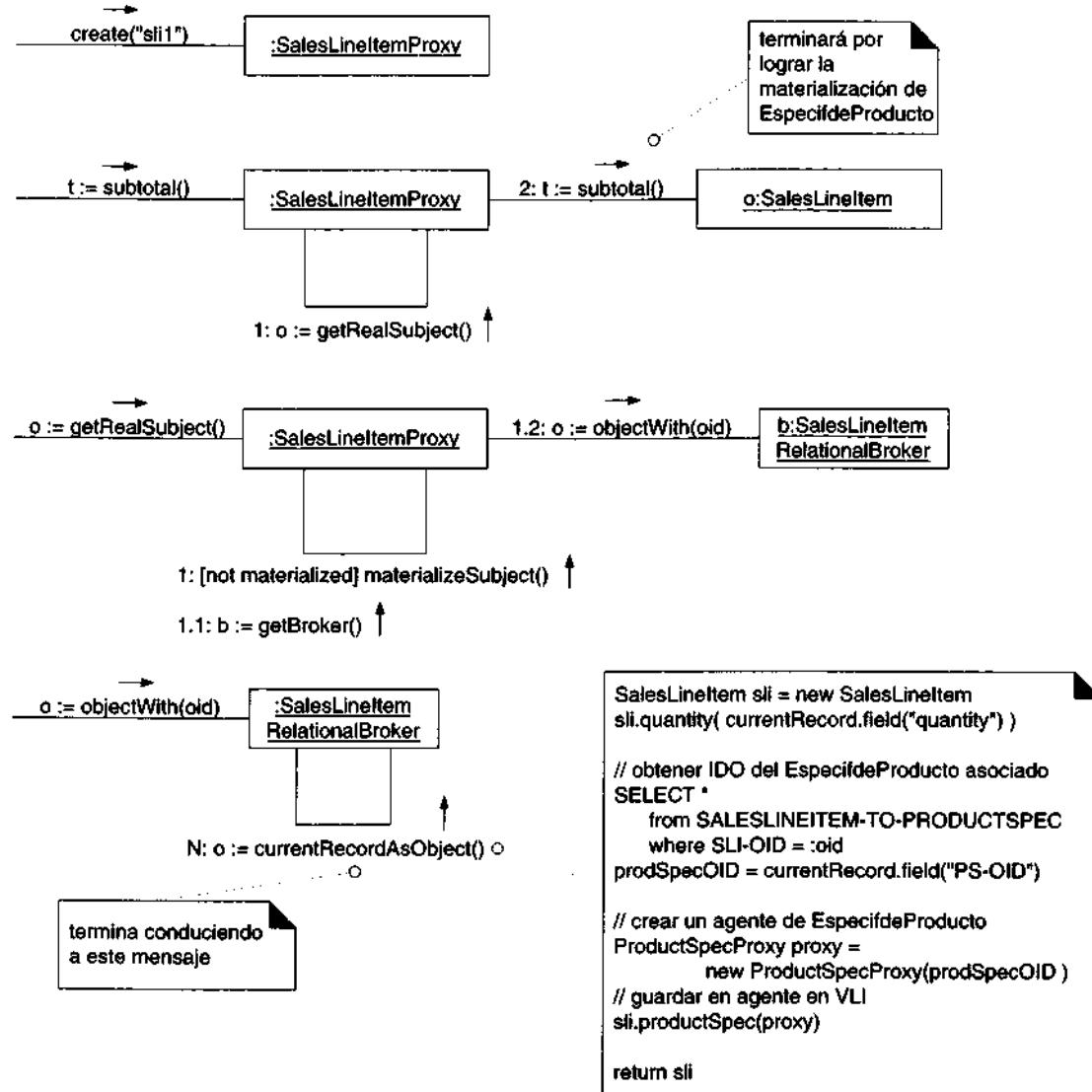


Figura 38.17 Materialización a partir de una instancia VentasLineadeProducto.

Así pues, *VentasLineadeProducto* referencia a un agente, no a la *EspecificaciondeProductos* real. Esta última no se materializará mientras no se le envíe el mensaje *precio*, como se advierte en la figura 38.18.

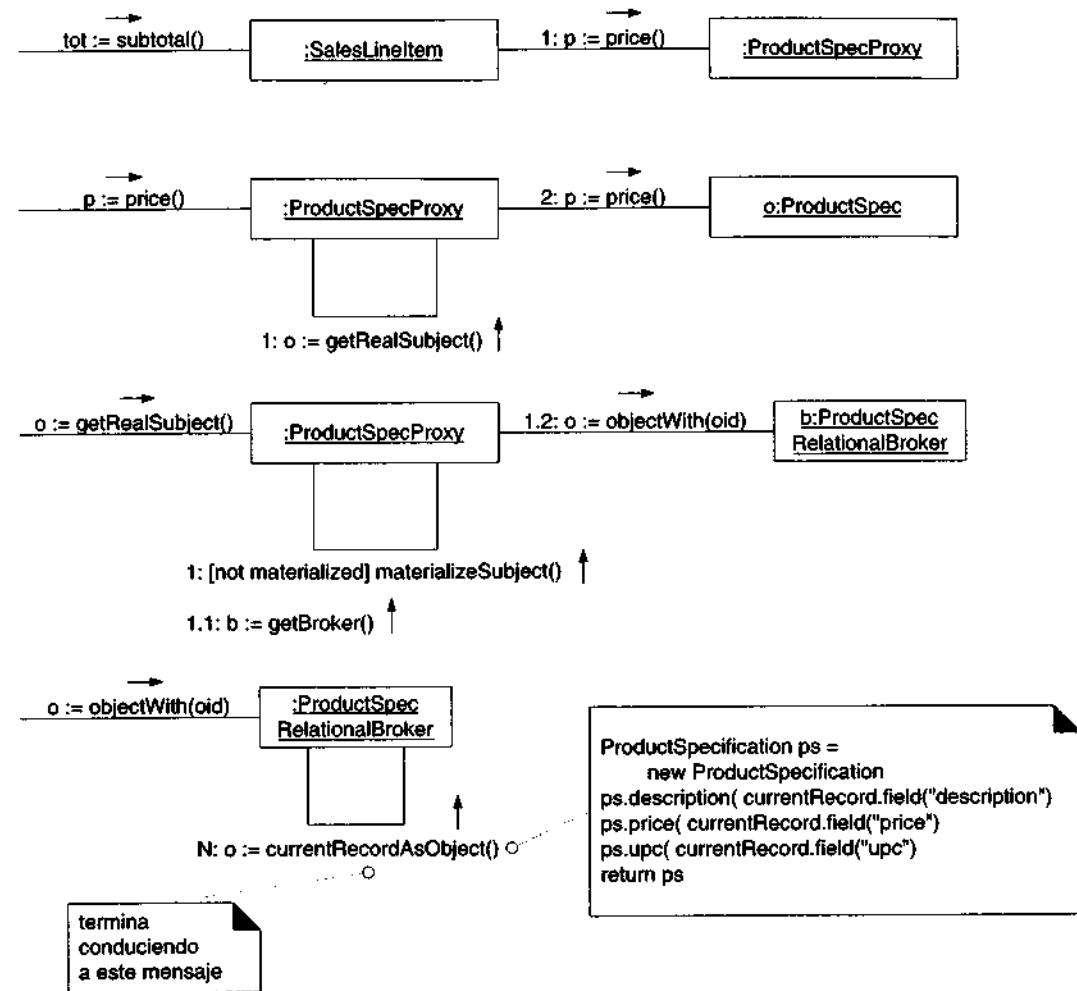


Figura 38.18 Materialización de una instancia EspecificaciondeProducto.

38.18 Operaciones de transacciones

Por lo general, una transacción acaba por ser guardada en una base de datos. Cuando se invoca una operación de *commit*, a los objetos se les trata de modo diferente según el estado de transacción que presenten. Por ejemplo, si hemos materializado un objeto viejo a partir de una base de datos pero sin modificarlos, no es necesario volver a introducirlo en la base. Por el contrario, si lo modificamos, habrá que actualizar el registro correspondiente de la base de datos.

38.18.1 Estado de transacción de los objetos

He aquí los posibles estados relevantes de la transacción:

1. **Limpio y nuevo.** Objetos nuevos, sin modificaciones.
2. **Limpio y viejo.** Objetos viejos materializados a partir de una base de datos, sin modificaciones.
3. **Sucio y nuevo.** Objetos nuevos, sin modificaciones.
4. **Sucio y viejo.** Objetos viejos materializados a partir de una base de datos, con modificaciones.
5. **Eliminar nuevo.** Objetos nuevos que deben ser eliminados.
6. **Eliminar viejo.** Objetos viejos que fueron materializados a partir de una base de datos y que deben ser eliminados.

Como vimos con anterioridad, un Intermediario de Base de Datos conservará cachés especiales para cada uno de estos estados y garantizará con ello que un objeto está en la caché apropiada.

Cuando un objeto se materializa por primera vez, su intermediario se encarga de colocarlo en la caché Limpio y Viejo. Cuando se modifica, se le cambia de sitio y se le pone en la caché Sucio y Viejo. Los objetos nuevos comienzan en la caché Limpio y Nuevo, pudiendo dirigirse después a la caché Sucio y Nuevo cuando sean modificados. Si se marca un objeto para eliminación, se le colocará en su respectiva caché Eliminar.

38.18.2 Cómo se ensucia un objeto

Un objeto se *ensucia* cuando modificamos uno de sus atributos. La forma más frecuente de señalar que se haga esto son los métodos mutador (establecedor). Por ejemplo:

```
// clase EspecificaciondeProducto
void price(float p)
{
    price = p;
    BrokerServer.instance().dirty(this);
}
```

A una Fabricación Pura *ServidordeIntermediario* del patrón Singleton y Fachada se le notificará que un objeto en particular está sucio. El *ServidordeIntermediario* encontrará el Intermediario apropiado de la Base de Datos para esta clase de objetos y le notificará que el objeto está sucio. El Intermediario entonces introduce el objeto en una caché sucia. Si era un objeto materializado, entrará en la caché Vieja y Sucia. Si era un objeto nuevo que todavía no se encuentre en la base de datos, se dirigirá hacia la Caché Nueva y Sucia.

¿Por qué enviar mensajes al objeto Fachada *ServidordeIntermediario* y no a cualquier otro? Porque así se conserva en un nivel mínimo el acoplamiento de los objetos del dominio. Él tan sólo sabe que una clase forma parte del esquema de persistencia: el *ServidordeIntermediario*. Con ello se da soporte al patrón Bajo Acoplamiento y se reduce al mínimo el impacto de los cambios.

38.18.3 Cómo eliminar

Si se desea eliminar un objeto, habrá que registrar explícitamente el hecho para que podamos realizar la modificación correspondiente en la base de datos después de una operación *commit*. Como en el caso de la señal *sucia*, esto se logra enviando un mensaje a *ServidordeIntermediario*. Por ejemplo:

```
// clase CatalogodeProductos
void removeProductSpec(ProductSpec p)
{
    // eliminar p en la colección de ProductSpecs
    productSpecs.remove(p);

    // notificarle al intermediario que p debe eliminarse
    // en la base de datos
    BrokerServer.instance().delete(p);
}
```

A un *ServidordeIntermediario* del patrón Fachada Singleton puede notificársele que un objeto está sucio. El *ServidordeIntermediario* encontrará al intermediario de bases de datos correspondiente de esta clase de objeto y le notificará que el objeto está sucio. El intermediario de la base de datos introducirá entonces el objeto en una caché sucia. Si era un objeto materializado, los introduciremos en la Caché Vieja y Sucia. Si era un objeto nuevo que todavía no estaba en la base de datos, se dirigirá hacia la Caché Nueva y Sucia.

38.18.4 La operación *commit*

Cuando se decidió instalar la transacción, se envió un mensaje *commit* a la Fachada *ServidordeIntermediario*. Aunque cualquier objeto puede dirigir este mensaje, se acostumbra que un Coordinador de Aplicaciones se encargue de ello.

```
BrokerServer.instance().commit();
```

El método *ServidordeAgente-commit* simplemente dirige un mensaje *commit* a cada intermediario.

```
void BrokerServer.commit()
{
    for each broker b
        b.commit()
}
```

En el tiempo de una transacción los objetos pueden ser modificados, creados y eliminados. Suponiendo que se encuentren en la caché del estado correspondiente de la transacción (Viejo y Sucio, por ejemplo), he aquí las reglas de lo que deberá hacer el mensaje *commit*:

- Caché Nueva y Limpia.
 - insertar en base de datos.
 - dirigirse a Caché Vieja y Limpia.

- Caché Vieja y Limpia.
 - ignorar; no han cambiado.
- Caché Nueva y Sucia.
 - insertar en base de datos.
 - dirigirse a Caché Vieja y Limpia.
- Caché Vieja y Sucia.
 - actualizar en base de datos.
 - dirigirse a Caché Vieja y Limpia.
- Caché Nueva Eliminada.
 - eliminar en caché.
- Caché Vieja Eliminada.
 - eliminada en la base de datos.
 - Eliminar en caché.

38.19 Búsqueda de objetos en el almacenamiento persistente

¿Cómo podemos recuperar un registro a partir de un almacenamiento persistente? La solución depende de las herramientas, de las bibliotecas y el ambiente operativo. Por ejemplo, en los ambientes del sistema operativo de Microsoft, podemos valernos de los servicios DAO dentro de MFC. Rebasa el ámbito de nuestra exposición ofrecer más detalles.

Dentro del esquema hemos definido el método *IntermediarioEPRelacional-SeleccionarPrimero(consulta)* para prestar el servicio de localizar el primer registro (renglón de una tabla) que reúna los criterios de la consulta. Su implementación depende del ambiente operativo.

Un Intermediario de Base de Datos necesita ofrecer dos patrones de búsqueda cuando se recuperen registros a partir de un almacenamiento persistente:

- Búsqueda mediante el identificador de objetos (el criterio más común de búsqueda).
- Búsqueda mediante criterios arbitrarios, como la clave primaria del dominio (por ejemplo, el número del seguro social).

Cuando se utiliza un esquema de persistencia, enfrentamos el problema de *preparar el escenario*: ¿con qué criterios debería recuperarse el objeto raíz en una jerarquía de composición? Los objetos no raíz pueden materializarse utilizando Agentes Virtuales y realizando la búsqueda con sus identificadores de objeto como clave de consulta. Pero normalmente no conoceremos el identificador del objeto raíz. Consideremos, por ejemplo, el caso de materializar una instancia *Venta*, su instancia relacionada *VentasLineadeProducto* y sus *EspecificacionesdeProducto* asociadas. Suponiendo que se haya materializado la *Venta* y que tuviera referencias a Agentes Virtuales para todas sus *VentasLineadeProducto*, las líneas de productos se materializarán por demanda, basándose en los valores del identificador de objetos en los agentes. De manera análoga, si se materializó una instancia *VentasLineadeProducto* y si tenía una referencia a un Agente Virtual para su *EspecificaciondeProducto*, también podemos materializarla por medio de su valor de identificador de objetos. Pero si en esta jerarquía de composición no se conoce el identificador del objeto raíz *Venta*, ¿cómo preparamos la escena e introduciremos en la memoria esta primera instancia *Venta*?

El problema anterior nos indica la necesidad de contar con capacidades de búsqueda orientadas al dominio, entre ellas la búsqueda de *Ventas* mediante su fecha y hora o alguna otra clave primaria adecuada. Así, en el caso de la *Venta*, la clase *IntermediarioRelacionaldeVentas* puede ofrecer además un método público para efectuar la materialización por fecha y hora, como se advierte en la figura 38.19.

38.18.5 La operación de rollback

Una vez que se haya decidido someter la transacción a un rollback, se envía un mensaje *rollback* a la Fachada *ServidordeIntermediario*. Aunque cualquier objeto puede hacerlo, se acostumbra que el Coordinador de Aplicaciones asuma esta responsabilidad.

```
BrokerServer.instance().rollback();
```

El método *ServidordeIntermediario-rollback* simplemente envía un mensaje de *rollback* a cada intermediario.

He aquí las reglas de lo que debe hacer una instalación:

- Caché Vieja y Limpia.
 - ignorar; no han cambiado.
- El resto de las cachés.
 - eliminar en la caché.

Una de las ventajas del uso constante de Agente Virtual con un objeto y con una referencia directa es el efecto que se obtiene en el caso de un rollback. Una vez vaciadas las cachés, todos los Agentes Virtuales se referirán a los objetos no materializados y no a los modificados en la memoria local. La siguiente referencia a un Agente Virtual hará que el objeto viejo sea rematerializado a partir del almacenamiento persistente; cualquier cambio del objeto se evaporó al vaciar la caché.

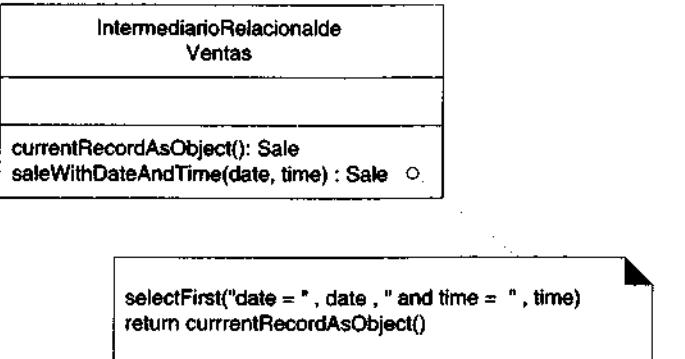


Figura 38.19 Capacidades de búsqueda orientadas al dominio.

38.20 Diseños alternos

38.20.1 Metadatos e intermediarios parametrizados

En contraste con la creación de un Intermediario de Base de Datos concreto e independiente para cada clase de objetos persistentes, podemos definir metadatos (datos acerca de datos) respecto al mapeo de clases y tablas, respecto al mapeo de los nombres de atributos y campos, etc. Los metadatos pueden conservarse en un objeto *MetadatosdeAlmacenamientoPersistente*. En este enfoque no es necesario elaborar una jerarquía extensa de intermediarios. Es posible, por ejemplo, que el *Intermediario-deEPRelacional* sea una clase concreta instanciada y parametrizada con metadatos para cada clase de objetos persistentes. No se requieren subclases de *Intermediario-deEPRelacional*.

El enfoque de metadatos es más robusto y flexible que el de formación de subclases adoptado en el actual esquema persistente; se recomienda en aplicaciones que contengan muchas clases de objetos persistentes y esquemas rápidamente cambiantes. Por ser más complejos de explicar e implementar, no lo estudiaremos a fondo. Si el lector desea más información, le recomendamos consultar el patrón Reflexión en [BMRSS96].

38.20.2 Objetos consulta

A diferencia de las consultas de cadenas simples que se ejemplifican en esta exposición (por ejemplo, “OID = 123”), es posible crear una clase *Consulta* cuyas instancias estén parametrizadas con expresiones booleanas. Este esquema tiene la ventaja de abstraer de cualquier lenguaje de manipulación de datos (SQL, por ejemplo) y de manipular las consultas y razonar con ellas (por ejemplo, compilarlas para darles una forma más eficiente).

38.20.3 Cambio de intermediarios y de intermediarios de bases de datos en la memoria

Es posible y conveniente crear un Intermediario en-la Memoria que no guarde objetos en un almacenamiento persistente externo cuando se envía la señal de *commit*. Esto es muy útil al iniciarse el desarrollo y las pruebas, pues se evita el impacto que en el desempeño tiene el utilizar un intermediario real y la complejidad de su implementación completa.

Los clientes de todas las subclases de *IntermediarioEP* emplean sólo unos cuantos métodos públicos, entre ellos *objetoCon(unIDO)*; podemos desconectar un intermediario y conectar otro, sin que ello afecte a los objetos cliente. Así, un diseñador puede utilizar un Intermediario en-la Memoria (figura 38.20) durante algún periodo y luego remplazarlo por un intermediario relacional o plano. El cambio puede efectuarse modificando el Método Fábrica *crearIntermediario* en el patrón Agente Virtual, que especifica cuál intermediario emplear.

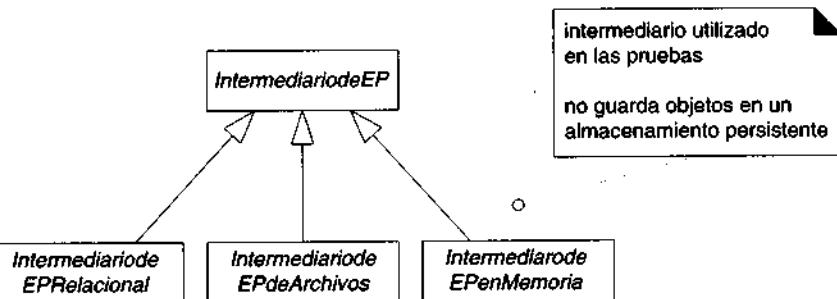


Figura 38.20 Intermediario en-la Memoria.

38.20.4 Comparación entre estados de transacción y cachés múltiples

Colocar un objeto en una de las cachés de intermediarios, digamos la caché *ViejaLimpia* o *ViejaSucia*, es una forma de recordar el estado de transacción del objeto. En el caso de un intermediario, se trata de un mecanismo eficiente. Pero podemos recurrir a un diseño alterno donde cada objeto se asocie a un objeto *EstadodeTransaccion* que indica si es viejo y limpio, viejo y sucio, etc. Todos los objetos pueden estar en una caché, y el estado de transacción del objeto se conoce mediante el objeto asociado de estado y no mediante su pertenencia a la caché.

¿Dónde debería recordarse la relación *EstadodeTransaccion*? Conviene no agregar directamente el conocimiento de la persistencia a las definiciones del objeto del dominio; de ahí que recomendemos las siguientes opciones:

- Si se usa una superclase común *ObjetoPersistente*, el estado es un atributo definido en esa clase.
- En un Mapa (*Dictionary* o *Hashtable*) que conserva el intermediario. La clave del Mapa es el objeto, el valor asociado Mapa es el *EstadodeTransaccion* del objeto.

- Como un atributo del *AgenteVirtual*. Este diseño presenta una complicación: si varios agentes se relacionan con el mismo objeto real, todos ellos habrán de permanecer sincronizados.

Además, cada clase *Estado de Transaccion* puede definir un método que especifique lo que ha de hacerse en un commit o rollback. Y como los objetos estado probablemente no conserven ninguna información, también pueden ser singlettons: sólo una instancia de cada clase se necesita en la aplicación.

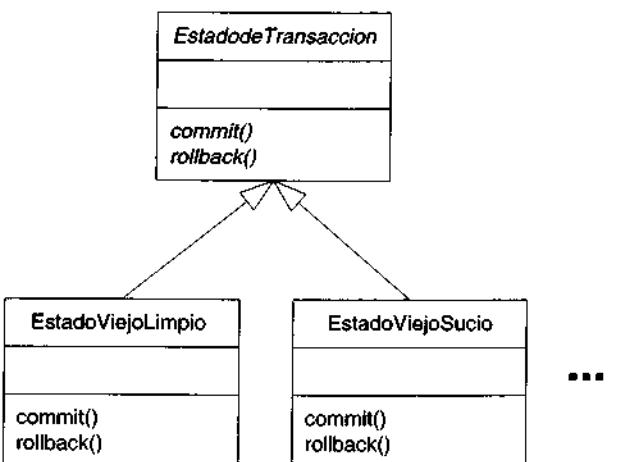


Figura 38.21 Estados de transacción de los objetos persistentes.

38.21 Cuestiones sin resolver

Hemos dado una brevíssima introducción a los problemas y a las soluciones de diseño en un esquema de persistencia. Ofrecemos una sucinta exposición de varios temas importantes:

- Desmaterialización de objetos. En resumen, los agentes concretos deben definir un método *objetoComoRegistro* que transforme un objeto en registro. Para desmaterializar las jerarquías de composición se requieren la colaboración entre varios intermediarios y la conservación de tablas asociativas (si se emplea una base de datos relacional).
- Apunadores hacia atrás. Un esquema de persistencia a veces tienta al diseñador para que a otros objetos les agregue apunadores hacia atrás que el objeto no requeriría por sí mismo.
- Materialización y desmaterialización de colecciones.
- Manejo de errores cuando fracasa una operación de base de datos.
- Estrategias de acceso a multiusuarios y de bloqueo.
- Seguridad: control del acceso a la base de datos.

APÉNDICE A. LECTURAS RECOMENDADAS

Análisis y diseño generales orientados a objetos

- Booch, G., 1994. *Object-Oriented Analysis and Design*. Redwood City, CA: Benjamin/Cummings.
- Coad, P. 1995. *Object Models: Strategies, Patterns and Applications*. Englewood Cliffs, NJ.: Prentice-Hall.
- Coleman, D., et al. 1994. *Object-Oriented Development: The Fusion Method*. Englewood Cliffs, NJ.: Prentice-Hall.
- Jacobson, I., et al. 1992. *Object-Oriented Software Engineering: A Use Case Driven Approach*. Reading, MA.: Addison-Wesley.
- Rumbaugh, J., et al. 1991. *Object-Oriented Modelling and Design*. Englewood Cliffs, NJ.: Prentice-Hall.

Construcción de modelos conceptuales

- Fowler, M. 1996. *Analysis Patterns: Reusable Object Models*. Reading, MA.: Addison-Wesley.
- Hay, D. 1996. *Data Model Patterns: Conventions of Thought*. NY, NY.: Dorset House.
- Martin, J. y Odell, J. 1995. *Object-Oriented Methods: A Foundation*. Englewood Cliffs, NJ.: Prentice-Hall.

Diseño orientado a objetos

- Riel, A. 1996. *Object-Oriented Design Heuristics*. Reading, MA.: Addison-Wesley.
- Wirfs-Brock, R., Wilkerson, B. y Wiener, L. 1990. *Designing Object-Oriented Software*. Englewood Cliffs, NJ.: Prentice-Hall.

Patrones

- Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P. y Stal, M. 1996. *A System of Patterns*. West Sussex, Inglaterra: Wiley.
- Coad, P. 1995. *Object Models: Strategies, Patterns and Applications*. Englewood Cliffs, NJ.: Prentice-Hall.
- Fowler, M. 1996. *Analysis Patterns: Reusable Object Models*. Reading, MA.: Addison-Wesley.
- Gamma, E., Helm, R., Johnson R. y Vlissides, J. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA.: Addison-Wesley.
- Riel, A. 1996. *Object-Oriented Design Heuristics*. Reading, MA.: Addison-Wesley.
- Varios editores. *Pattern Languages of Program Design*. Todos los volúmenes. Reading, MA.: Addison-Wesley.

Análisis de requerimientos

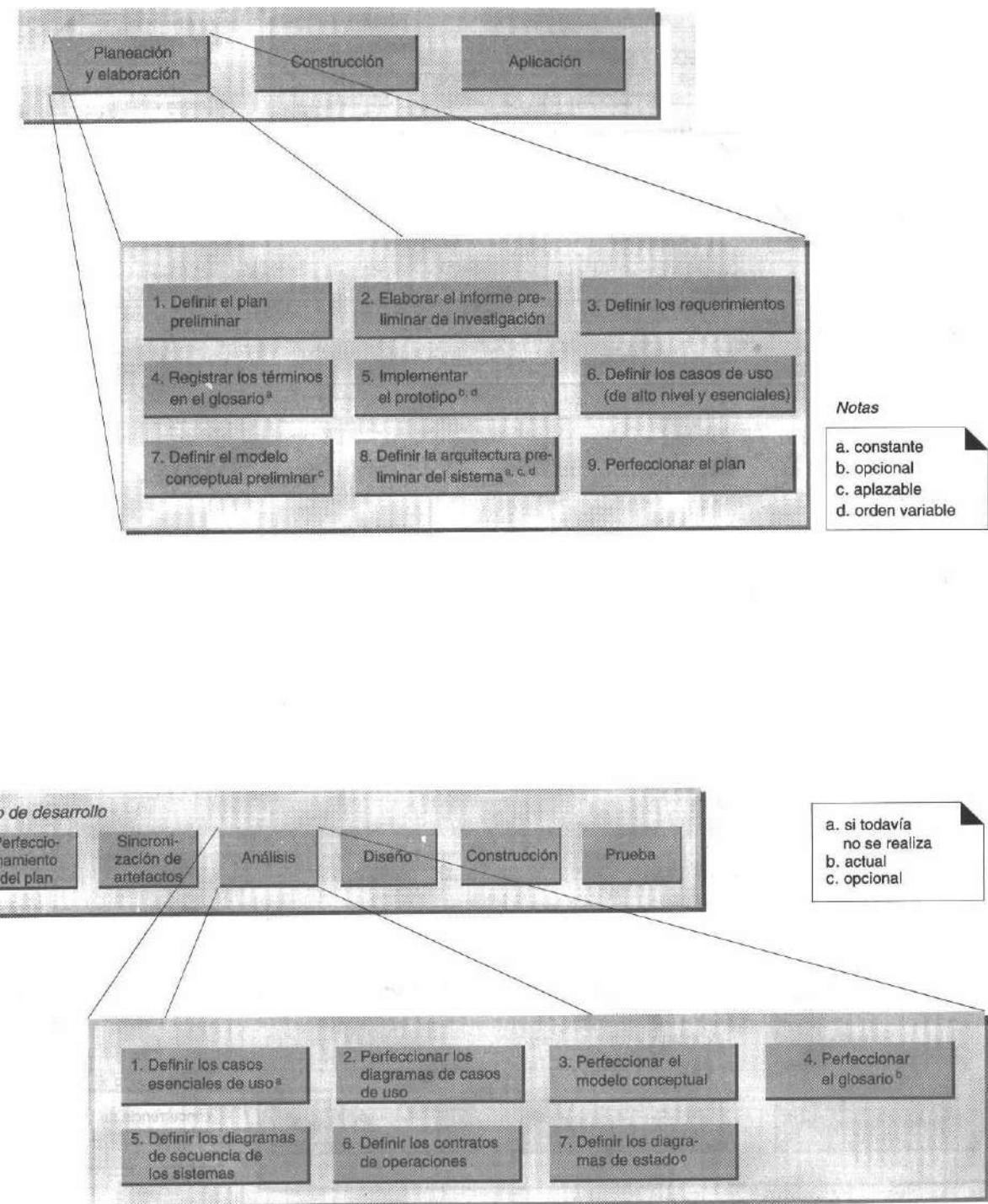
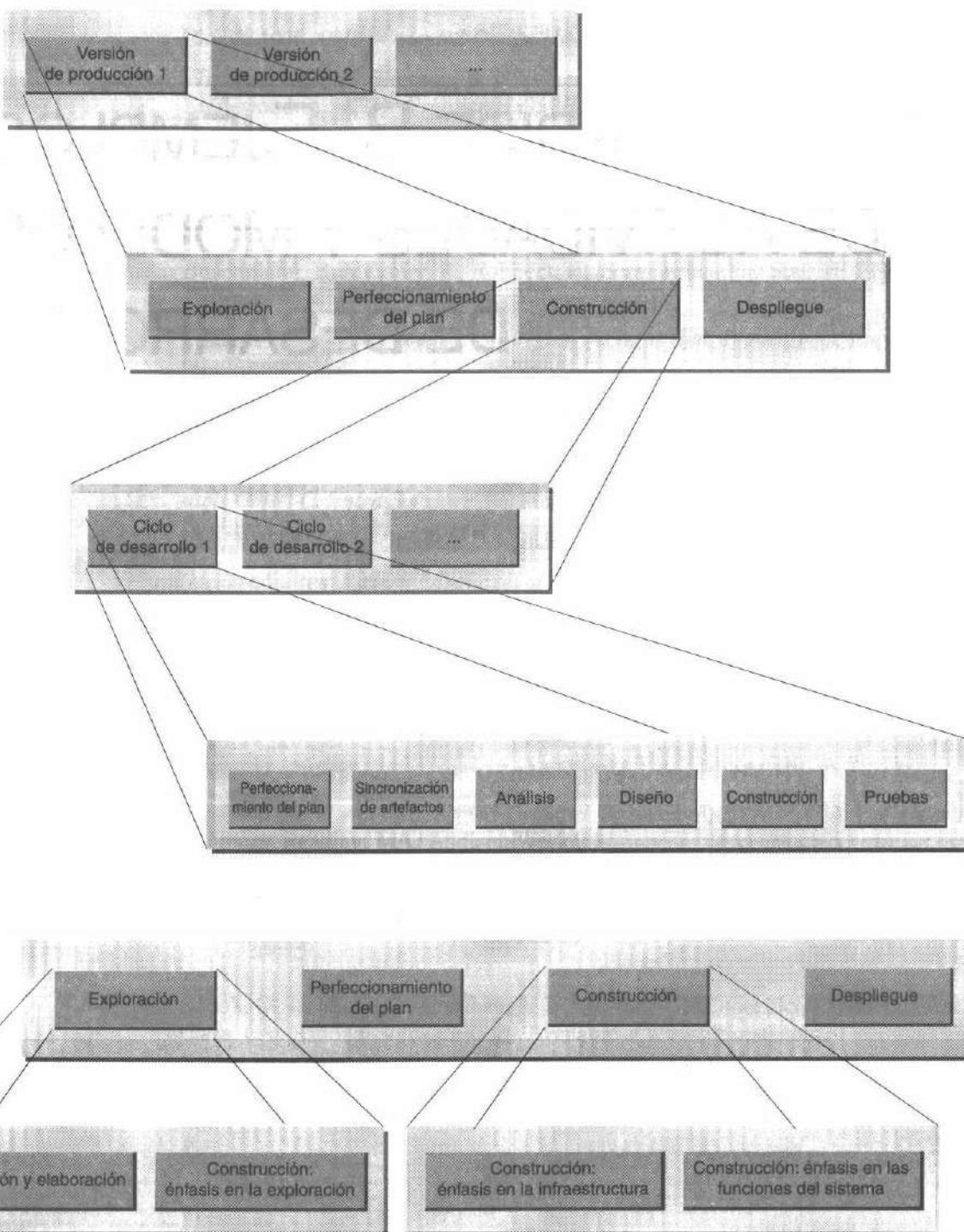
- Gause, D. y Weinberg, G. 1989. *Exploring Requirements*. NY, NY.: Dorset House.
- Jacobson, I., et al. 1992. *Object-Oriented Software Engineering: A Use Case Driven Approach*. Reading, MA.: Addison-Wesley.

El UML (Unified Modeling Language)

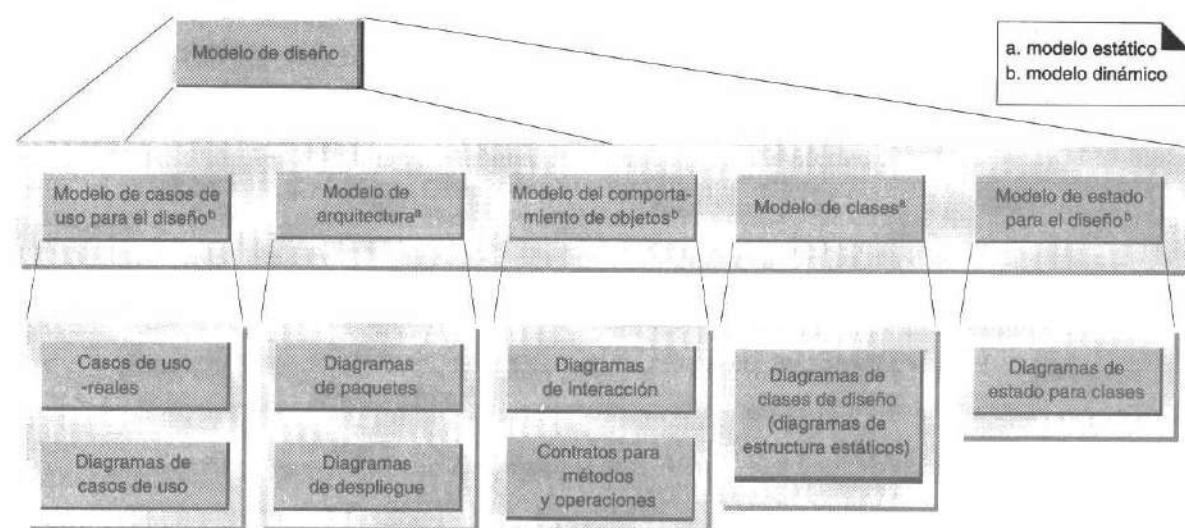
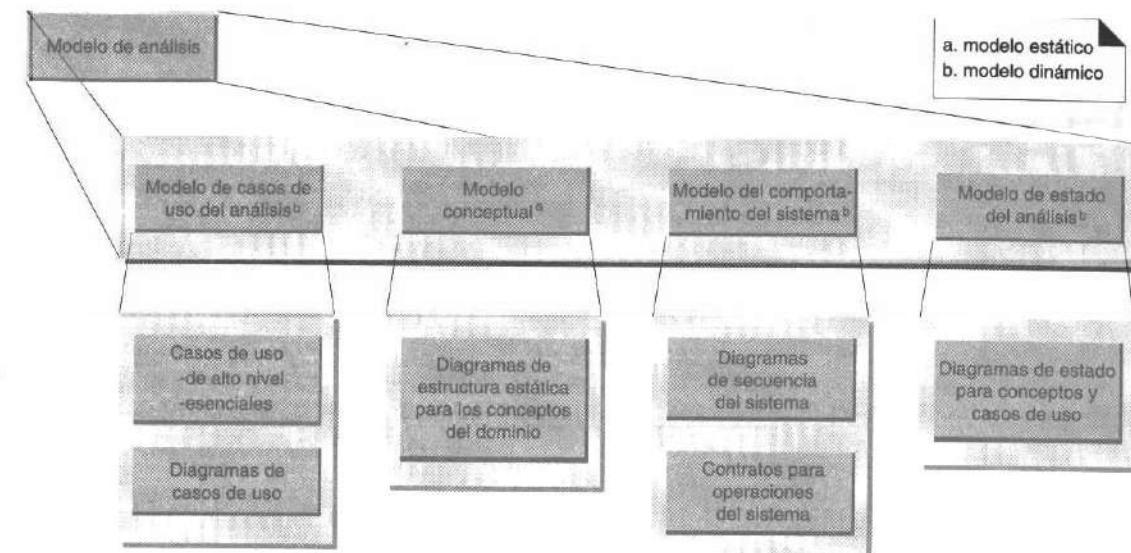
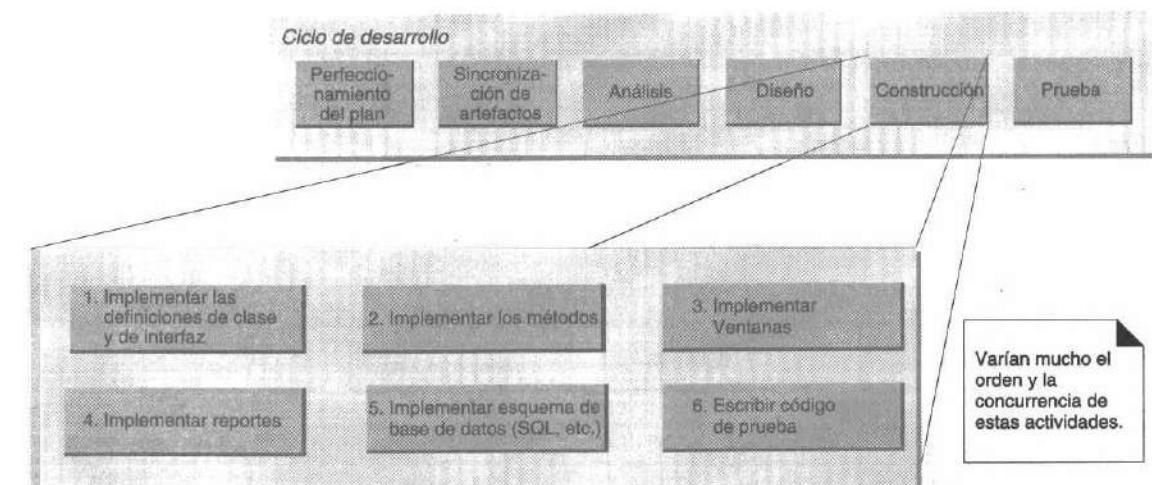
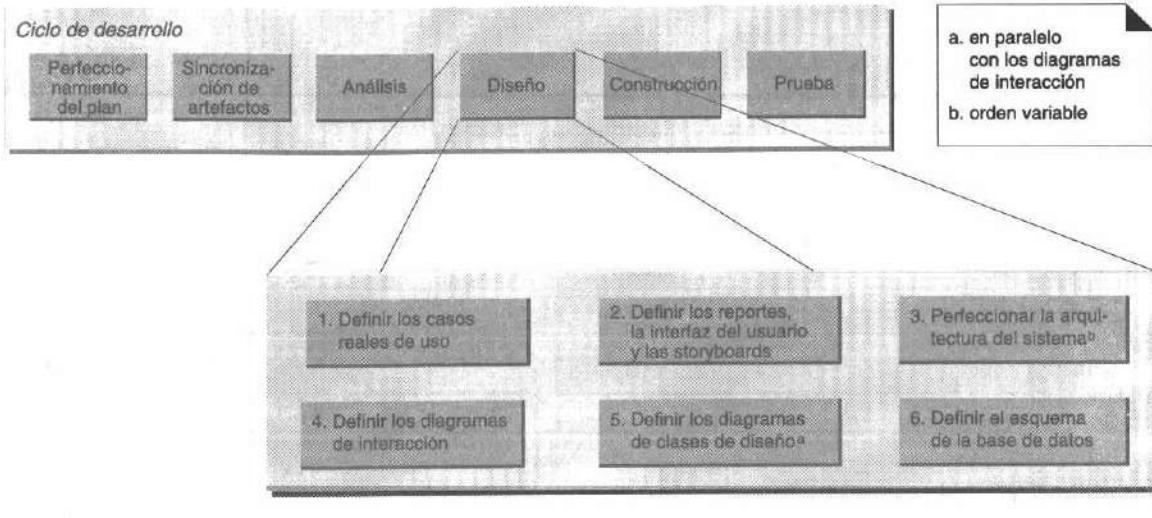
- Booch, G., Jacobson, I. y Rumbaugh, J. 1997. The UML specification documents. Santa Clara, CA.: Rational Software Corp. Available at www.rational.com.

APÉNDICE B. EJEMPLOS DE ACTIVIDADES Y MODELOS DE DESARROLLO

Ejemplo de actividades de desarrollo



Ejemplos de modelos



BIBLIOGRAFÍA

- Abbot83** Abbott, R. 1983. Program Design by Informal English Descriptions. *Communications of the ACM* vol. 26(11).
- AIS77** Alexander, C., Ishikawa, S. y Silverstein, M. 1977. *A Pattern Language-Towns-Building-Construction*. Oxford University Press.
- BC89** Beck, K. y Cunningham, W. 1989. A Laboratory for Object-oriented Thinking. *Proceedings of OOPSLA 89*. SIGPLAN Notices, vol. 24, núm. 10.
- Beck94** Beck, K. 1994. Patterns and Software Development. *Dr Dobbs Journal*. Feb., 1994.
- BJR97** Booch, G., Jacobson, I. y Rumbaugh, J. 1997. The UML specification documents. Santa Clara, CA.: Rational Software Corp. Véanse los documentos en www.rational.com.
- BMRSS96** Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P. y Stal, M. 1996. *A System of Patterns*. West Sussex, Inglaterra: Wiley.
- 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.
- Brooks75** Brooks, F., 1975. *The Mythical Man-Month*. Reading, MA: Addison-Wesley.
- Brown96** Brown, K. y Whitenack, B. 1996. Crossing Chasms. *Pattern Languages of Program Design* vol. 2. Reading, MA: Addison-Wesley.
- Coad95** Coad, P. 1995. *Object Models: Strategies Patterns and Applications*. Englewood Cliffs, NJ.: Prentice-Hall.

GLOSARIO

- Coleman94** Coleman, D., et al. 1994. *Object-Oriented Development: The Fusion Method*. Englewood Cliffs, NJ.: Prentice-Hall.
- Constantine97** Constantine, L. 1997. The Case for Essential Use Cases. *Object Magazine*. Mayo, 1997. NY, NY: SIGS Publications.
- Coplien95** Coplien, J. 1995. *The History of Patterns*. Véase <http://c2.com/cgi/wiki?HistoryOfPatterns>.
- Fowler96** Fowler, M. 1996. *Analysis Patterns: Reusable Object Models*. Reading, MA.: Addison-Wesley.
- Gartner95** Schulte, R., 1995. *Three-Tier Computing Architectures and Beyond*. Informe Publicado. GartnerGroup.
- GHJV95** Gamma, E., Helm, R., Johnson, R. y Vlissides, J. 1995. *Design Patterns*. Reading, MA.: Addison-Wesley.
- GW89** Gause, D. y Weinberg, G. 1989. *Exploring Requirements*. NY, NY.: Dorset House.
- Jacobson92** Jacobson, I., et al. 1992. *Object-Oriented Software Engineering: A Use Case Driven Approach*. Reading, MA.: Addison-Wesley.
- Lieberherr88** Lieberherr, K., Holland I. y Riel, A. 1988. Object-Oriented Programming: An Objective Sense of Style. *OOPSLA 88 Conference Proceedings*. NY, NY: ACM SIGPLAN.
- Mo95** Martin, J. y Odell, J. 1995. *Object-Oriented Methods: A Foundation*. Englewood Cliffs, NJ.: Prentice-Hall.
- Rumbaugh91** Rumbaugh, J., et al. 1991. *Object-Oriented Modelling and Design*. Englewood Cliffs, NJ.: Prentice-Hall.
- Rumbaugh97** Rumbaugh, J. 1997. Models Through the Development Process. *Journal of Object-Oriented Programming*, Mayo, 1997. NY NY: SIGS Publications.
- Standish94** Anónimo. 1994. *Charting the Seas of Information Technology: Chaos*. Informe publicado. The Standish Group.
- Wirfs-Brock93** Wirfs-Brock, R. 1993. Designing Scenarios: Making the Case for a Use Case Framework. *Smalltalk Report*, nov-dic 1993. NY, NY: SIGS Publications.

- abstracción** Acción de concentrar las cualidades esenciales o generales de cosas similares. También, las características esenciales resultantes de una cosa.
- acoplamiento** Dependencia entre elementos (generalmente tipos, clase y subsistemas), normalmente debida a la colaboración entre ellos para prestar un servicio.
- agregación** Propiedad de una asociación que representa una relación todo-parte y donde (normalmente) se le contiene por toda la vida.
- análisis** Investigación de un dominio, la cual da origen a modelos que describen sus características estáticas y dinámicas. Se centra en cuestiones de "qué" más que de "cómo".
- análisis orientados a objetos** Investigación del dominio o sistema de problemas a partir de los conceptos de dominio, como tipos de objetos, asociaciones y cambios de estado.
- arquitectura** Descripción de la organización y estructura de un sistema. Varios niveles de arquitecturas intervienen en la creación de sistemas de software, desde la arquitectura física del hardware hasta la arquitectura lógica de un esquema de aplicaciones.
- asociación** Descripción de un conjunto relacionado de enlaces o vínculos entre objetos de dos tipos.
- asociación calificada** Aquella cuya pertenencia parte o divide el valor de un calificador.
- asociación recursiva** Aquella en que la fuente y el destino son el mismo tipo de objeto.
- atributo** Característica o propiedad de tipo con un nombre asignado.
- atributo de clase** Característica o propiedad que es igual en todas las instancias de una clase. Esta información suele almacenarse en la definición de la clase.
- caso de uso** Descripción narrativa textual de la secuencia de eventos y acciones que ocurren cuando un usuario parte o divide en un diálogo con un sistema durante un proceso significativo.

clase	En el lenguaje UML, “descripción de un conjunto de objetos que comparten los mismos atributos, operaciones, métodos, relaciones y significado” [BJR97]. En este libro, a menudo se usa como sinónimo de una “clase de implementación” de UML: el mecanismo de software con que se definen e implementan los atributos y métodos de un tipo en particular.	diagrama de estado	Forma de una máquina de estado finito con que se describe el comportamiento dinámico de un tipo.
clase abstracta	Clase que puede emplearse exclusivamente como superclase de alguna otra clase; no es posible crear objetos de una clase abstracta, salvo como instancias de una subclase. Una clase abstracta suele servir para definir una interfaz común de varias subclases.	diseño	Proceso que se sirve de los productos del análisis para generar una especificación destinada a implementar un sistema. Descripción lógica de cómo funcionará un sistema.
clase concreta	Aquella que puede tener instancias.	diseño orientado a objetos	Especificación de una solución lógica de software a partir de objetos de software: clases, atributos, métodos y colaboraciones.
clase de contenedor	Aquella diseñada para alojar y manipular una colección de objetos.	dominio	Límite formal que define determinado tema o área de interés.
clasificación	La que define una relación entre un tipo y sus instancias. El mapeo taxonómico identifica la extensión de un tipo.	encapsulamiento	Mecanismo con que se ocultan los datos, la estructura interna y los detalles de la implementación de un objeto. La interacción con un objeto se realiza a través de una interfaz pública de las operaciones.
colaboración	Dos o más objetos que participan en la relación de cliente/servidor, con el propósito de prestar un servicio.	especificaciones de requerimientos	Documento que describe lo que hace un sistema de software: sus funciones y sus atributos. Generalmente escritas desde el punto de vista del usuario.
componente	Módulo discreto de software con una interfaz.	esquema	Conjunto de clases abstractas y concretas que colaboran y que pueden servir de plantilla para resolver una familia afín de problemas. Por lo regular se extiende formando subclases del comportamiento propio de la aplicación.
composición	La identificación de un tipo en que cada instancia está constituida por otros objetos.	estado	Condición de un objeto entre eventos.
concepto	Categoría de ideas o cosas. En este libro, con él se designan cosas del mundo real y no entidades de software. La intención de un concepto es la descripción de sus atributos, operaciones y significado. La extensión de un concepto es el conjunto de instancias u objetos muestra que pertenecen a él. A menudo se define como sinónimo de tipo.	evento	Ocurrencia notable.
conformidad	Relación entre dos tipos tal que, si el tipo X se conforma al tipo Y, los valores de aquél son también miembros de éste y satisfacen la definición del tipo Y.	extensión	Conjunto de objetos a los que se aplica un concepto. Los objetos de la extensión son ejemplos o instancias del concepto.
constructor	Método especial que se invoca siempre que la instancia de una clase se crea en el lenguaje C++ o en el lenguaje Java. El constructor a menudo lleva a cabo acciones de inicialización.	generalización	Actividad consistente en identificar aspectos comunes entre conceptos y en definir las relaciones entre el supertipo (concepto general) y el subtipo (concepto especializado). Es una forma de hacer clasificaciones taxonómicas entre conceptos que luego se explican con ejemplos en las jerarquías de tipos. Los subtipos se conforman a los supertipos en cuanto a la intención y la extensión.
contrato	Define las responsabilidades y poscondiciones que se aplican al uso de una operación o método. También designa el conjunto de las condiciones relacionadas con una interfaz.	herencia	Característica de los lenguajes de programación orientados a objetos, en virtud de la cual las clases pueden especializarse a partir de superclases más generales. La subclase adquiere automáticamente las definiciones de atributos y clases hechas a partir de las superclases.
delegación	Idea de que un objeto puede enviarle un mensaje a otro en respuesta a un mensaje. Por tanto, el primer objeto delega la responsabilidad al segundo.	identidad del objeto	Característica de que la existencia de un objeto es independiente de los valores asociados a él.
derivación	Proceso que consiste en definir una nueva clase por referencia a otra existente y agregar luego atributos y métodos. La clase existente es la superclase; a la nueva se le llama subclase o clase derivada.	IDO	Identificador del objeto.
escomposición funcional	Proceso de refinar la solución de un problema descomponiéndolo varias veces en pasos funcionales cada vez más pequeños.	instancia	Miembro individual de un tipo o de una clase.
		instanciación	Creación de una instancia de un tipo o de una clase.

intensión	Definición de un concepto.	patrón	Es la descripción etiquetada de un problema, de la solución, de cuándo aplicar la solución y la manera de hacerlo dentro de otros contextos.
interfaz	Conjunto de representaciones de operaciones públicas.	persistencia	Almacenamiento duradero del estado de un objeto.
jerarquía de clase	Descripción de las relaciones de herencia entre clases.	polimorfismo	Concepto según el cual dos o más tipos de objetos pueden responder a un mismo mensaje en formas diferentes, usando para ello operaciones polimórficas. También, capacidad de definir las operaciones polimórficas.
lenguaje de programación orientado a objetos	Aquel que soporta los conceptos de encapsulamiento, herencia y polimorfismo.	poscondición	Restricción que debe cumplirse una vez terminada una operación.
mensaje	Mecanismo en virtud del cual los objetos se comunican entre sí; generalmente una respuesta para ejecutar un método.	precondición	Restricción que debe cumplirse antes que se solicite una operación.
metamodelo	Modelo que define otros modelos. El metamodelo de UML define los tipos de elementos del lenguaje; por ejemplo Tipo y Operación.	privado	Mecanismo de ámbito con el cual se limita el acceso a los miembros de una clase para que otros objetos no puedan verlos. En condiciones normales se aplica a todos los atributos y a algunos métodos.
método	En el lenguaje UML, implementación o algoritmo específicos de la operación de una clase. Informalmente, procedimiento de software que puede ejecutarse en respuesta a un mensaje.	público	Mecanismo de ámbito con el cual se hace a los miembros accesibles a otros objetos. Normalmente se aplica a algunos métodos pero no a los atributos, porque los atributos públicos violan el encapsulamiento.
método de clase	Aquel que define el comportamiento de una clase en contraste con el de sus instancias.	receptor	Objeto al cual se envía un mensaje.
método de instancia	Aquel cuyo ámbito es una instancia. Se invoca enviando un mensaje a una instancia.	responsabilidad	Servicio o grupo de servicios ofrecidos por un tipo; una responsabilidad abarca uno o más propósitos u obligaciones de un tipo.
modelo	Descripción de las características estáticas, dinámicas o ambas de un tema, presentada en varias vistas (generalmente diagramáticas o textuales).	restricción	Limitación o condición que se impone a un elemento.
multiplicidad	Número de objetos a los que se permite participar en una asociación.	subclase	Especialización de otra clase (la superclase). Una subclase hereda los atributos y métodos de la superclase.
objeto	En el lenguaje UML, instancia de una clase que encapsula el estado y el comportamiento. Más informalmente, ejemplo de una cosa.	subtipo	Especialización de otro tipo (el supertipo) que se conforma a la connotación y denominación del supertipo.
objeto activo	Aquel que tiene su propia red de control.	superclase	Clase cuyos atributos y métodos hereda otra clase.
objeto persistente	Aquel que puede sobrevivir al proceso o el hilo que lo creó. Un objeto persistente existe mientras no sea eliminado explícitamente.	supertipo	En una relación de generalización-especialización, el tipo más general; objeto que tiene subtipos.
operación	En el lenguaje UML, “servicio que puede solicitarse a un objeto para realizar el comportamiento” [BJR97]. Una operación tiene una representación, especificada por su nombre y parámetros, y se la invoca a través de un mensaje. Un método es una implementación de una operación mediante un algoritmo específico.	tipo	En el lenguaje UML, descripción de un conjunto de objetos similares con atributos y operaciones, pero que no incluye métodos. Algunos autores definen el tipo y el concepto como sinónimos.
operación polimórfica	La misma operación implementada de modo distinto por dos o más tipos.	tipo abstracto	Tipo tal que todos los objetos que se conforman a él deben conformarse también a uno de sus subtipos.
papel	Fin nombrado de una asociación que indica su propósito.	tipo concreto	Aquel que puede tener instancias.

transición Relación entre estados que se cruzan, si el evento especificado ocurre y se cumple la condición de custodia.

transición de estado Cambio de estado en un objeto; algo que puede indicarse con un evento.

valores puros de datos Tipos de datos para los cuales la identidad única de instancia no es significativa; por ejemplo, números, booleanos y cadenas. En el lenguaje UML, también se conocen como Tipos de Datos.

variable de instancia Usado en Java y en Smalltalk, atributo de una instancia.

vínculo Conexión entre dos objetos; instancia de una asociación.

visibilidad Capacidad de ver o tener referencia a un objeto.

A

- acoplamiento, 200
- actor, 52
 - iniciador, 52
 - participante, 53
- adecuación, 420
- Agente (Proxy)
 - dispositivo, 400
 - remoto, 415
 - virtual, 468
- agregación, 198, 359
 - compartida, 360
 - compuesta, 359
- agregación de compuestos, 359
- Alta Cohesión, 203
- Alto nivel, caso de uso de, 56
- análisis, 6
 - análisis de requerimientos, 8
 - análisis del dominio orientado a objetos, 8
 - análisis estructurado, 90
 - análisis orientado a objetos, 6
 - análisis y diseño
 - definición de, 6
 - análisis y diseño estructurados, 14
 - análisis y diseño, modelos de, 29-30
 - análisis y diseño orientados a objetos, 6
 - analogía de, 7
 - definición de, 6
 - ejemplo del juego de dados, 10
 - principios básicos de, 3-16
 - y descomposición funcional, 14
- arquitectura, 437
 - arquitectura de tres capas, 273
 - arquitectura multicapas, 274
- artefacto, 30
 - relaciones, 31
- asignación de responsabilidades, 9
 - importancia de, 5
- asociación, 105
 - adicción de, 105-108
 - asignación de nombre, 111
 - calificada, 365
 - criterios de, útil, 106
 - de alta prioridad, 109
 - destacar la necesidad de conocer, 115
 - directrices de, 110
 - múltiple entre tipos, 112
 - multiplicidad de, 110
 - navegabilidad, 264
 - nivel de detalle, 109
 - nombres de papeles, 362
 - notación de UML, 106
 - obtención con lista de, 107
 - recursiva, 366

- reflexiva, 366
- vínculo, 173
- atributo, 120
 - adicción, 119-130
 - de referencia, 300
 - derivado, 128, 364
 - notación UML, 120
 - simple, 121
 - sin claves foráneas, 123
 - tipos no primitivos, 124
 - tipos válidos, 120
 - valor puro de datos, 121, 22
 - y cantidades, 125
- atributos sistemáticos, 42

B

Bajo acoplamiento, 200

C

- calificador, 365
- capa, 349
- caso de uso, 8, 10, 49
 - abstracto, 53
 - asignación de nombre, 61
 - clasificación, 76
 - clasificación y programación, 73-80
 - creación de, 47-71
 - creación de, real, 163-166
 - cuándo crear casos abstractos de uso, 321
 - de alto nivel, 26, 56
 - de ejemplo real, 165
 - diagrama de, 55
 - diagrama de estado, 382
 - ejemplo de relaciones de usos, 323
 - error común en, 53
 - esencial, 58
 - esencial comparado con real, 58
 - expandido, 27, 50, 56
 - formatos de, 55
 - funciones y rastreabilidad de sistemas, 55
 - identificación de, 53
 - Iniciar, 76
 - límites de sistemas, 56
 - primario, 58
 - proceso de, 54
 - programación, 75
 - real, 59, 163
 - relación, 321-328
 - relaciones de usos, 322
 - secundario, 58
 - versiones, 77

y proceso de desarrollo, 63
y procesos del dominio, 54
casos de desarrollo del sistema, 450
ciclo de desarrollo
 inicio del, 81-82
 programación de casos de uso, 75
 segundo ciclo, 317-319
clase, 102
 abstracta, 347
 notación UML, 257
código
 creación de, 295-308
 solución de, en Java, 309-313
cohesión, 203
 baja, 210
Comando, 421
Comando, patrón de, 214
comportamiento
 del sistema, 137, 147
comportamiento sistémico, 137, 147
 principios básicos del, 137
compuesto, 359
concepto(s), 89
 comparados con papel, 363
 comparados con tipos e interfaces, 101
 conceptos semejantes, 97
 error en la obtención de, 97
 especificación o descripción de, 99
extensión de, 89
intensión de, 89
notación UML, 87
obtención de, con identificación de sustantivos, 93
obtención de, con una lista de categorías, 91
símbolo de, 89
Véase también tipo
construcción, 6
contrato, 147
 descripción de secciones del, 148
 directrices y, 154
 ejemplo de, 147
 elaboración de, 145-158
 grado de terminación, 153
 poscondición del, 150
 precondiciones del, 153
 y descripción de algoritmos, 153
Controlador, 206
 aplicación, 223
 inflado, 210
coordinador de aplicaciones, 287
CRC, 215
Creador, 197, 225
 aplicación, 225-235

D

definición de clase
 a partir de un diagrama de diseño de clases, 299
dependencia
 notación UML, 428
desarrollo incremental, 436
desarrollo iterativo, 20, 436
diagrama
 de clases del diseño, 9, 12, 257

adicción de métodos, 260
creación de, 255-270
ejemplo de, 257
información de tipos, 263
notación de los miembros de, 268
presentación de la navegabilidad, 264
presentación de las relaciones de dependencia, 267
versus modelo conceptual, 259
y multiobjetos, 262
de colaboración, 9, 12, 169
colecciones, 181
componentes, 429
condicionales mutuamente excluyentes, 180
creación de, 172, 217-245
creación de instancias, 177
ejemplo de, 170
instancias, 173
iteración, 176
mensaje a objeto de clase, 182
mensajes, 174
mensajes condicionales, 180
multibjeto, 181
notación de, 167-183
número de secuencias, 179
parámetros de, 174
secuenciación de mensajes, 179
sintaxis de mensajes, 175
valor de retorno, 175
vínculos, 173
y otros artefactos, 218
de despliegue, 430
de estado, 381
 acciones de transición, 388
 caso de uso, 382
 condiciones de protección, 388
 diseño de, 379-389
 ejemplo de, 384
 estados anidados, 388
 para el caso de uso, 382
 ¿para qué tipos?, 384
 sistema, 383
de estado de casos de uso, 382
de estado sistémico, 383
de implementación, 429
de interacción, 169
de paquete, 276
 creación de, 349-354
de secuencia de sistemas, 137
 creación de, 135-144
 presentación del texto de casos de uso, 143
de secuencia de sistemas y otros artefactos, 140
de secuencias, 137, 169
diagrama de clase
 diseño de, 257
diagramas de colaboración
 creación de, 405-424
diccionario de modelos, 131
diseño, 6
 de arquitectura multicapas, 271-291
de mapeo, conversión de, en código, 295-308
 solución de Java al, 309-313
del sistema
 principios básicos, 271-291

orientado a objetos, 6
traducción de, a código, 295-308
traducción de, a Java, 309-313

E

especialización, 335
especificación de las propiedades
 notación UML, 428
Esquema, 458
 búsqueda de objetos, 483
 de persistencia, 456
 diseño de, 455-486
 diseños alternos de, 484
 ideas básicas, 459
 materialización, 464
 operaciones de transacción, 479
 patrón Administración de Caché, 467
 patrón Identificador de Objeto, 461
 patrón Intermediario de Base de Datos, 462
 patrón Representación de objetos como tablas, 460
 persistencia, 456
 representación de relaciones en tablas, 475
 requerimientos, 458
estado, 379
 construcción de modelos de, 347
estereotipo
 notación UML, 428
estudio de casos de punto de venta
 introducción, 35-38
evento, 379
 asignación de nombre al, 142
 externo, 387
 interno, 387
 sistémico, 138, 206
 temporal, 387
Experto, 193
 aplicación de, 227, 230, 231
extensión, 89

F

Fabricación Pura, 396
Fachada (Facade), 418
función sistemática, 42

G

Gang of Four Patterns (patrones de la pandilla de cuatro), 405
generalización, 335, 336
 aplicación de, 335-348
 conformidad y, 338
 notación de tipos abstractos, 346
 notación UML, 336
 partición, 340
 principios básicos, 335
 pruebas de validez de subtipos, 339
 y conjuntos de tipos, 337
 y tipos, 337

glosario, 131
 elaboración de, 131-133
GoF (Pandilla de Cuatro), patrones de aplicación de, 405-424, 455-486
GRASP
 aplicación, 217-245, 405-424
 introducción, 185-215
 patrones residuales, 393-404

H

herencia, 348

I

inadecuación, 420
independiente del estado, 384
Indirección, 399
ingeniería inversa, 298
instancia
 notación UML, 173
Integración, 418
intensión, 89
interfaz, 102
 notación UML, 429

J

jerarquía de clase, 348
jerarquía de tipos, 335

L

Lenguaje Unificado de Construcción de Modelos, UML, 4, 10
Véase también UML
Ley de Demeter, 401
Véase también No hables con extraños (Don't Talk to Strangers)
límite de sistemas, 141
Lo Hago Yo Mismo (Do It Myself), 196

M

mensaje
 asíncrono, 430
 notación de UML, 427-431
 notación UML, 174
método
 abstracto, 347
 a partir del diagrama de colaboración, 302
 Fábrica (Factory Method), 473
 Plantilla (Template Method), 463
modelo, 29
 conceptual, 8, 11, 87
 adicción de asociaciones, 105-118
 adicción de atributos, 119-130
 conceptos adicionales, 329-333
 conceptos similares, 97

construcción de modelos de lo irreal, 98
 construcción de modelos para los estados cambiantes, 347
 creación de, 85-103
 directrices de, 96
 estrategia del diseñador de mapas, 96
 introducción al, 87
 obtención de conceptos, 91
 organización de, en paquetes, 351
 vocabulario del dominio, 88
 y descomposición, 90
 y diagrama de diseño de clases, 259
 de análisis, 31
 de casos de uso del análisis, 71
 de diseño, 31
 de sistemas, 30
 dinámico, 30
 estático, 30
 sistemático, 30
 multiobjetos, 181
 multiplicidad, 110

N

navegabilidad, 264
 No Hables con Extraños (Don't Talk to Strangers), 401
 violaciones aceptables, 404
 nombre de papeles (roles), 301
 nota
 notación UML, 427
 notificación de evento, 286

O

objeto, 9
 de dominio inicial, 239
 de sistemas, 138, 206
 operación, 101
 persistente, 456

P

papel, 110
 comparado con concepto, 363

paquete, 275
 dependencias de, 350
 notación UML, 275, 350
 propiedad de, 350
 referencia de, 350

paquete de arquitectura, diagrama de, 276

partición, 349
 de tipos, 340

patrón, 4, 189
 Agente dispositivo (Device Proxy), 400, 418

Agente remoto, 415
 Agente virtual, 468
 Alta Cohesión, 203
 Bajo acoplamiento, 200
 Comando, 214, 421
 Controlador, 206
 Creador, 197
 de Estado, 158
 Experto, 193

Fabricación Pura, 396
 Fachada, 418
 GRASP, 190
 Indirección, 399
 Ley de Demeter, 401
 Lo Hago Yo Mismo (Do It Myself), 196, 394
 Método Fábrica, 473
 Método Plantilla, 463
 No Hables con Extraños (Don't Talk to Strangers), 401
 nombres, 190
 Polimorfismo, 394
 Publicar-Suscribir (Publish-Subscribe), 285
 Puente, 469
 Separación modelo-vista, 224, 281
 Singleton, 413
 patrones
 Pandilla de Cuatro (Gang of Four), 405
 de diseño
 Pandilla de Cuatro "Gang of Four", 405
 de GRASP
 Alta Cohesión, 203
 Bajo Acoplamiento, 200
 Controlador (Controller), 206
 Creador (Creator), 197
 Experto (Expert), 193
 Fabricación Pura (Pure Fabrication), 396
 Indirección, 399
 No Hables con Extraños (Don't Talk to Strangers), 401
 Polimorfismo, 394
 polimorfismo, 394
 aplicación, 411
 poscondición
 como metáfora, 151
 Principio de Hollywood, 458
 proceso de desarrollo, 4, 17-27
 de software, 17
 fase de construcción, 25
 macropasos, 20
 motivación, 434
 plan y fase de perfeccionamiento, 23
 problemas del, 433-435
 programación
 con la clasificación de casos de uso, 76
 orientada a objetos, 6
 versiones de casos de uso, 77
 Publicar-Suscribir, 285
 Puente, 469

R

reingeniería de procesos de negocios, 57
 relación de dependencia, 267
 relaciones de usos en casos de uso, 322
 requerimientos (condiciones), 41
 creación de, 39-46
 responsabilidad, 187
 de hacer, 187
 de conocer, 188
 responsabilidades
 patrones de, 189
 patrones GRASP, 190
 y diagramas de interacción, 188
 y métodos, 187

Respuestas, 286
 restricción
 notación en UML, 427

S

Separación de modelo-vista (Model-View Separation), 224, 281
 símbolo, 89
 Singleton, 413
 notación abreviada de UML, 415
 subsistema, 349
 subsistemas, identificación de, 278
 subtipo
 conformidad de, 338
 creación de, 339
 partición de, 340
 pruebas de validez para, 339
 supertipo
 creación, 342

T

tipo, 101
 abstracto, 345
 notación UML, 346

asociativo, 357
 notación UML, 87
 partición de, 340
Véase también concepto
 transición, 381

U

UML
 aspectos básicos de, 15
 nò proceso de desarrollo, 19
 notación adicional de, 427-431
 Unified Modeling Language, 4, 10
Véase también UML

V

valor puro de datos, 121, 122
 vínculo, 173
 visibilidad, 227, 248
 aspectos generales de la, 247-253
 atributo de, 249
 global, 252
 localmente declarada, 251
 parámetro, 250