UML (UNIFIED MODELING LANGUAGE)
http://jms32.eresmas.net/tacticos/UML/UMLIndex.html

Mapa General

Referencia Bibliográfica

- El Lenguaje Unificado de Modelado. G. Booch, J Rumbaugh, I. Jacobson. Addison-Wesley
- El Lenguaje Unificado de Modelado. Manual de Referencia. J Rumbaugh, I. Jacobson, G. Booch. Addison-Wesley
- UML, gota a gota. Fowler, Martín Addison Wesley Longman de México,SA de CV
- OMG UML v1.3 Specification.

BOSQUEJO DEL PROCESO DE DESARROLLO

RESUMEN

Resumen muy completo sobre los pasos a seguir en el desarrollo de un proyecto, desde la concepción del mismo hasta su finalización.

No se habla para nada de código de programa

Tiene un carácter introductorio de los conceptos, que se desarrollan en los siguientes temas, que viene a ser partes específicas de la teoría general que aquí se comenta

Indica las diversas etapas de un proyecto y la forma de calcular el tiempo total de ejecución.

Índice de Contenidos

Introducción

UML es un lenguaje para modelar, no un método. Esta en evolución pero El resultado se llamará *Rational Objectory Process*. Se intenta lograr una estructura de procesos, algo que atrape los elementos comunes pero que al mismo tiempo permita la flexibilidad de emplear técnicas apropiadas para su proyecto.

Es importante analizar primero el proceso, para poder ver cómo funciona un desarrollo orientado a objetos.

Panorámica del proceso

La Figura 2-1 muestra la secuencia al nivel más alto del proceso de desarrollo.

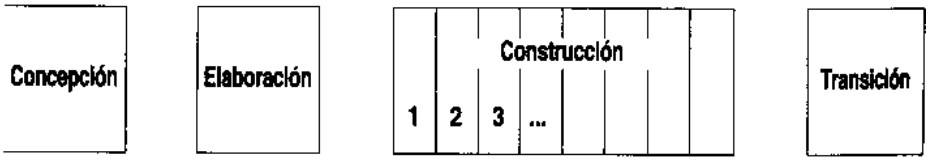

Figura 2-1: *Proceso de desarrollo del bosquejo*

El proceso de desarrollo es un proceso iterativo y gradual, en el sentido de que el software no se libera de un solo gran golpe al final del proyecto, sino que, al contrario, se desarrolla y se libera por partes. La etapa de **construcción** consta de muchas**iteraciones,** donde cada iteración construye software de calidad para producción, probado e integrado, que cumple un subconjunto de los requerimientos del proyecto. La entrega puede ser externa, destinada a los primeros usuarios, o puramente interna. Cada iteración contiene todas las etapas usuales del ciclo de vida: análisis, diseño, implementación y experimentación.

En principio, se puede comenzar por el inicio: seleccione cierta funcionalidad y constrúyala, escoja otra más, y así sucesivamente. Sin embargo, es importante, dedicar cierto tiempo a la planificación.

Las dos primeras etapas son las de concepción y elaboración. Durante la **concepción,**se establece la razón de ser del proyecto y se determina su alcance. Es aquí cuando se obtiene el compromiso del patrocinador del proyecto para proseguir. En la **elaboración,**se reúnen requerimientos más detallados, se hacen análisis y diseños de alto nivel, a fin de establecer una arquitectura base, y se crea el plan de construcción.

Incluso con este tipo de proceso iterativo, hay trabajos que deben quedar para el final, la etapa de **transición.** Entre ellos están las pruebas beta, y el entrenamiento del usuario.

Los proyectos varían en virtud de la cantidad de **ceremonia** que llevan consigo. Los proyectos de alto ceremonial tienen muchas entregas formales en papel, reuniones formales, autorizaciones formales. Los proyectos de bajo ceremonial pueden tener una etapa de concepción que consista en una plática de una hora con el patrocinador del proyecto y un plan asentado en una hoja de cálculo. Por supuesto, cuanto más grande sea el proyecto, más ceremonia se necesitará. Los pasos fundamentales de las etapas también se llevan a cabo, pero de modo muy diferente.

Hay que tratar de mantener el ceremonial al mínimo.

He presentado las iteraciones en la fase de construcción, pero no en las demás fases. De hecho, se pueden tener iteraciones en todas las fases y, con frecuencia, es buena idea tenerlas en las grandes fases. No obstante, la construcción es la fase clave donde se debe iterar.

Ésta es la perspectiva de alto nivel. Ahora nos sumergiremos en los detalles, de modo que tengamos la suficiente información para ver dónde encajan, dentro del panorama global, las técnicas que estudiaremos más adelante. Al hacerlo, hablaré un poco sobre dichas técnicas y cuándo usarlas. Podrá encontrar esto algo confuso si no está familiarizado con las técnicas. De ser éste el caso, pase por alto estas partes y vuelva a ellas después.

Concepción

La concepción puede adoptar muchas formas. Para algunos proyectos será, quizá, una charla frente a la cafetera automática: "Piensa cómo podemos poner nuestro catálogo de servicios en la Web." Para proyectos mayores, podría ser un amplio estudio de viabilidad que puede tardar meses.

Durante la etapa de concepción, se definirá la situación económica del proyecto: cuánto costará aproximadamente y cuánto rendirá. También se necesitará tener una idea del alcance del proyecto. Tal vez haga falta cierto trabajo de análisis inicial para tener una idea de la magnitud del proyecto.

No hay que dar demasiada importancia a la concepción. La concepción debe consistir en trabajar durante algunos días para determinar si vale la pena dedicar algunos meses de mayor investigación durante la elaboración.

Elaboración

En este momento tenemos luz verde para iniciar un proyecto. En esta etapa, lo normal es que sólo tenga unas ideas vagas de los requerimientos. Por ejemplo, podrá decir lo siguiente:

> *Vamos a construir la próxima generación del sistema de apoyo al cliente de la Watts Galore Utility Company. Tenemos la intención de construir un sistema más flexible, que esté más orientado al cliente, mediante tecnología orientada a objetos; en específico, uno de apoyo a las cuentas consolidadas de los clientes.*

En realidad, el documento de requerimientos será más extenso que éste, pero en la práctica no dirá mucho más.

A estas alturas, usted debemos comprender mejor el problema.

- ¿Qué es lo que va a construir en realidad?
- ¿Cómo lo va a construir?
- ¿Qué tecnología empleará?

Al tomar decisiones durante esta etapa acerca de dichas cuestiones, lo primero y más importante que debemos considerar son los riesgos del proyecto. ¿Cuáles son los factores que pueden pararlo? Cuanto mayor sea el riesgo, habrá que prestarle más atención.

Los riesgos se pueden clasificar, desde un punto de vista práctico, en cuatro categorías:

1. **Riesgos de requerimientos.** ¿Cuáles son los requerimientos del sistema? El gran peligro es que se construya el sistema erróneo, un sistema que no haga lo que quiere el cliente. Durante la etapa de elaboración, deberá comprender bien los requerimientos y sus prioridades relativas.
2. **Riesgos tecnológicos.** ¿Cuáles son los riesgos tecnológicos con los que tendremos que enfrentarnos?: Plantéese las siguientes preguntas:
    1. Va a usar objetos. ¿Tiene ya la suficiente experiencia en el trabajo de diseño orientado a objetos (diseño OO)?
    2. Se le ha aconsejado que use Java y la Web. ¿Funciona esta tecnología? ¿Puede proporcionar las funciones que los usuarios necesitan a través de un Navegador (browser) explorador de Web conectado a una base de datos?
3. **Riesgos de habilidades.** ¿Puede conseguir la asesoría y los expertos que necesita?
4. **Riesgos políticos.** ¿Existen fuerzas políticas que se puedan interponer en su camino y afectar seriamente el proyecto?

En su caso, podrá haber más riesgos, pero los que se incluyen en estas categorías casi siempre están presentes.

Manejo de los riesgos de requerimientos

Los requerimientos son importantes Y es donde las técnicas del UML son especialmente provechosas. El punto de partida son los casos de uso. Éstos, por lo tanto, son los motores de todo el proceso de desarrollo.

Los casos de uso se estudian en detalle en el tema siguiente, y a continuación sólo los describiré brevemente.

Un caso de uso es una interacción típica entre el usuario y el sistema con el fin de lograr cierto objetivo. Imagínese el procesador de texto con el que estoy trabajando. Un caso de uso sería "poner en negritas el texto seleccionado", otro, "crear el índice de un documento".

Como podrá apreciar en estos ejemplos, el tamaño de los casos de uso puede variar considerablemente. La clave es que cada uno indica una función que el usuario puede entender y, por tanto, tiene un valor para él. Un desarrollador puede responder de manera más concreta.

> *Me tornará dos meses hacer el índice de funciones que usted necesita. También tengo un caso de uso para manejar la revisión ortográfica. Sólo tengo tiempo de hacer uno. ¿Cuál necesita primero? Si quiere texto en negritas, lo puedo hacer en una semana, y puedo encargarme, al mismo tiempo, de las cursivas.*

Los casos de uso son la base para la comunicación entre los patrocinadores y los desarrolladores durante la planificación del proyecto.

Una de las cosas más importantes en la etapa de elaboración es descubrir de los casos de uso potenciales del sistema en construcción. Por supuesto, en la práctica no va a descubrirlos todos. Sin embargo, querrá encontrar la mayor cantidad posible, en especial los más importantes. Es por esta razón que, durante la etapa de elaboración, deberá programar entrevistas con los usuarios, con el fin de recopilar los casos de uso.

No hay necesidad de detallar los casos de uso. Normalmente bastan uno o dos párrafos de texto descriptivo. El texto debe ser lo bastante específico para que los usuarios comprendan la idea básica y para que los desarrolladores tengan una idea general de lo que abarcan.

Los casos de uso no son, sin embargo, todo el panorama. Otra tarea importante es elaborar el esqueleto del modelo conceptual del dominio. Dentro de las cabezas de uno o varios usuarios es donde se encuentra el panorama del funcionamiento del negocio. Por ejemplo:

> *Nuestros clientes pueden tener varios sitios y nosotros les suministramos diversos servicios a estos sitios. En la actualidad, a cada cliente se le entrega un recibo por todos los servicios proporcionados en un sitio. Queremos que al cliente se le facturen todos los servicios de todos los sitios. A esto lo llamamos facturación consolidada.*

Este pasaje contiene las palabras "*cliente*", "*sitio*" y "*servicio*". ¿Qué significan estos términos? ¿Cómo se relacionan entre ellos? Un modelo conceptual del dominio comienza a contestar estas preguntas y, al mismo tiempo, establece los fundamentos para el modelo de objetos con el, que se representarán los objetos del sistema, posteriormente, durante el proceso. Empleo el término **modelo de dominio** para describir cualquier modelo cuyo sujeto primario sea el mundo al que da apoyo el sistema de cómputo y cualquiera que sea la etapa del proceso de desarrollo en que se encuentre.

En Objectory usted se vale de distintos modelos para captar diversos aspectos del desarrollo. Los modelos de dominio y los casos de uso capturan los requerimientos funcionales; los modelos de análisis abarcan las implicaciones de estos requerimientos para una aplicación particular; los modelos de diseño agregan la infraestructura interna que hace que funcione la aplicación. El modelo de dominio de Objectory casi se termina de construir antes de que usted encuentre casos de uso; su propósito es explorar el vocabulario del dominio en términos comprensibles para los expertos del dominio.

Una vez que usted cuenta con un modelo de dominio y un modelo de casos de uso, desarrolla un **modelo de diseño,** el cual reconoce tanto la información en los objetos del dominio como el comportamiento de los casos de uso. El modelo de diseño agrega clases que se encargan de llevar a cabo el trabajo y que proporcionan además una arquitectura reutilizable, que servirá de ayuda para extensiones futuras. En los proyectos mayores, usted puede desarrollar un modelo de análisis intermedio con el que se pueden explorar las consecuencias de los requerimientos externos antes de tomar decisiones sobre el diseño.

Objectory no requiere que se construya todo el sistema a manera de "cascada". Es importante dejar correctas las clases de dominio clave y los casos de uso clave para después construir una arquitectura de sistema reutilizable que pueda manejar ampliaciones posteriores. Luego, se pueden agregar de manera progresiva casos de uso, los cuales se pueden implementar en el modelo de diseño como parte de un proceso iterativo de desarrollo. No se debe construir el sistema completo de un solo golpe.

Encuentro especialmente valiosas dos técnicas de UML para la construcción de modelos de dominio.

- Los diagramas de clases, cuando se dibujan desde una perspectiva conceptualperspectiva conceptual, son excelentes para capturar el lenguaje del negocio. Estos diagramas le pueden servir a usted para

establecer los conceptos de que se valen los expertos del negocio al pensar en él, y para plantear cómo estos expertos vinculan los conceptos entre sí.
- Los diagramas de actividades complementan los diagramas de clases describiendo el flujo del trabajo del negocio; es decir, los pasos que siguen los empleados para llevar a cabo sus labores. El aspecto crucial de los diagramas de actividad es que fomentan la búsqueda de procesos paralelos, lo cual es importante en la eliminación de secuencias innecesarias en los procesos del negocio.

A algunas personas les gusta apoyarse en los diagramas de interacción para investigar cómo interactúan diversas actividades en la empresa. Al considerar como unidad tanto a los trabajadores como a las actividades, logran comprender con mayor facilidad el proceso.

Los diagramas de interacción son más útiles durante este último paso. Además, los diagramas de interacción no promueven los procesos paralelos de la manera en que lo hacen los diagramas de actividad. Usted puede emplear los diagramas de actividad con carriles para encargarse tanto de las personas como del paralelismo, pero este procedimiento hace más complicados los diagramas (también puede usar diagramas de estado en conjunto con el flujo de trabajo.)

Una vez cubiertas la mayoría de las áreas importantes, hay que consolidar los diversos diagramas en un solo modelo de dominio consistente.

Intente desarrollar un solo modelo de área que maneje todos los requerimientos expresados en los modelos discretos anteriores. Este modelo puede entonces servir como punto de partida para la formación de clases y para un diseño de clases más profundo en la etapa de construcción. Si el modelo resulta muy grande, mediante paquetes divido el modelo en partes. Llevo a cabo la consolidación de los diagramas de clase y de actividad y, tal vez, dibujo un par de diagramas de estado para las clases que tengan ciclos de vida interesantes.

Este primer modelo de dominio es un esqueleto, no un modelo de alto nivel. El término "modelo de alto nivel" significa que faltan muchos detalles. Con frecuencia se comete este error en varias situaciones, por ejemplo, "no mostrar los atributos en estos modelos". El resultado son modelos sin sustancia. Es fácil entender por qué los desarrolladores se mofan de tales esfuerzos

No es conveniente construir un modelo detallado, porque le tomará demasiado tiempo y morirá de parálisis analítica. La clave está en encontrar y concentrarse en los detalles importantes. La mayor parte de los detalles se cubrirán durante el desarrollo iterativo. El esqueleto es el fundamento del resto del modelo. Es detallado, pero representa sólo una parte pequeña de la historia.

El modelado de dominios también es dirigido por los casos de uso a medida que se descubren. Conforme aparecen los casos de uso, el equipo de modelado debe estudiados para determinar si contienen alguna cosa que pudiera influir en el modelo de dominio. De ser así, es preciso investigar más a fondo; en caso contrario, los casos de uso se deben hacer a un lado, por el momento.

El equipo que construye el modelo de dominio debe ser un grupo pequeño (de dos a cuatro personas) que incluya desarrolladores y expertos en el dominio. El equipo viable más pequeño será de un desarrollador y un experto en el dominio. El experto en el dominio (y de preferencia también el desarrollador) debe estar entrenado en el manejo de los diagramas del UML apropiados para el modelado conceptual.

El equipo deberá trabajar intensamente durante el periodo de elaboración hasta concluir el modelo. Durante este periodo, el líder deberá asegurar que el equipo no se empantane en los detalles ni que opere a un nivel tan alto que pierda contacto con la realidad. Una vez que entienden lo que están haciendo, el estancamiento es el mayor peligro. Una fecha límite inflexible funciona de maravilla para lograr que las mentes se concentren.

Como parte de la comprensión de los requerimientos, se debe construir un prototipo de las partes intrincadas de los casos de uso. La de los prototipos es una técnica valiosa para entender mejor cómo funcionan las situaciones más dinámicas. A veces siento que entiendo bien la situación a partir de los diagramas, pero en otras ocasiones descubro que necesito un prototipo para apreciar adecuadamente lo que pasa. En general no hago un prototipo de todo el panorama, sino que, por el contrario, uso el modelo general del dominio para resaltar las áreas que necesitan prototipos.

Manejo de los riesgos tecnológicos

Lo más importante que hay que hacer al abordar los riegos tecnológicos es construir prototipos que prueben las partes tecnológicas con las que uno piensa trabajar.

Por ejemplo, digamos que usted está trabajando en C ++ y con una base de datos relacional. He aquí los pasos que deberá seguir:

1. Conseguir los compiladores de C ++ y las demás herramientas.
2. Construir una parte sencilla de una de las primeras versiones del modelo de dominio. Vea qué tal funcionan para usted las herramientas.
3. Construir la base de datos y conectada al código C ++.
4. Probar diversas herramientas. Vea cuáles son las más fáciles de manejar y las más apropiadas para el trabajo. Adquiera familiaridad con las herramientas que escoja.

No olvide que los riesgos tecnológicos mayores son inherentes a la manera en que se integran los componentes de un diseño, en lugar de hallarse en los componentes mismos. Usted puede conocer bien C++ y las bases de datos correspondientes, pero integrados no es tan fácil. Por eso es tan importante obtener todos los componentes con los que se pretende trabajar e integrados en esta etapa temprana del proceso.

También en esta etapa deberá ocuparse de cualquier decisión de diseño arquitectónico. Estas decisiones por lo general toman la forma de ideas acerca de lo que son los componentes y sobre la manera como se construirán. Esto es importante, en particular si se contempla un sistema distribuido.

Como parte de este ejercicio, concéntrese en las áreas que parezca que más adelante van a ser más difíciles de cambiar. Trate de llevar a cabo su diseño de tal forma que le permita cambiar los elementos del diseño en forma relativamente fácil. Pregúntese lo siguiente:

- ¿Qué sucederá si no trabaja una pieza de la tecnología?
- ¿Qué ocurrirá si no podemos conectar dos piezas del rompecabezas?
- ¿Cuál es la probabilidad de que algo vaya mal? ¿Qué haremos, si sucede esto?

Del mismo modo que en el modelo de dominio, usted deberá analizar los casos de uso a medida que aparezcan, a fin de determinar si contienen algo que pueda echar por tierra su diseño. Si abriga el temor de que contengan un "gusano púrpura", profundice en su investigación.

Durante este proceso, utilizará técnicas UML para esbozar sus ideas y documentar lo que está probando. En este momento, no intente tener una visión detallada; todo lo que necesita son bosquejos breves y, por tanto, eso es lo que debe usar.

- Los diagramas de clase y los diagramas de interacción son útiles para mostrar la manera en que se comunican los componentes.
- Los diagramas de paquetes pueden mostrar, en esta etapa, un cuadro de alto nivel de los componentes.

- Los diagramas de emplazamiento o *deployment diagrams*, pueden proveer una visión panorámica de la distribución de las piezas.

Manejo de los riesgos de habilidad

Suelo asistir con regularidad a conferencias y escucho ponencias sobre casos prácticos pronunciadas por gente que acaba de llevar a cabo un proyecto orientado a objetos. A la pregunta de" ¿cuál fue su mayor problema?" responden invariablemente con frases del estilo de "deberíamos haber recibido mayor entrenamiento".

Nunca deja de asombrarme que las compañías se embarquen en importantes proyectos OO con poca experiencia y sin idea sobre la manera de ganar más. Se preocupan por los costos del entrenamiento, pero pagan hasta el último centavo cuando el proyecto se alarga.

El entrenamiento es una forma de evitar errores, dado que los instructores ya los han cometido. Cometer errores consume tiempo y el tiempo cuesta. Así, se acaba pagando lo mismo en una u otra forma, pero la falta de entrenamiento provoca que el proyecto tarde más.

No soy un fanático de los cursos de entrenamiento formales. He impartido muchos de ellos y diseñado algunos otros también. Sigo sin convencerme de su eficacia en la enseñanza de las habilidades de la orientación a objetos. Dan a las personas un panorama de lo que necesitan saber, pero en realidad no logran transmitirles las habilidades indispensables para construir un proyecto real. Un breve curso de entrenamiento puede ser útil, pero sólo constituye el principio.

Si decide tomar un curso de entrenamiento, preste mucha atención a la persona que se lo impartirá, el instructor. Vale la pena pagar una buena suma extra por un instructor capaz y dedicado, porque así aprenderá mucho más durante el proceso. También es recomendable recibir el entrenamiento por partes, al momento que se necesiten. Por otra parte, si no pone en práctica de inmediato lo aprendido en el curso, pronto lo olvidará.

La mejor manera de adquirir las habilidades de la OO es a través del método de **tutoría,** mediante el cual un desarrollador experimentado colabora con usted en su proyecto durante un extenso periodo. El tutor le muestra cómo hacer las cosas, observa lo que usted hace y le da consejos, al tiempo que le ayudará con pequeños entrenamientos.

El tutor trabajará en los aspectos concretos de su proyecto y sabrá qué partes de su experiencia deben ponerse en práctica en el momento adecuado. En las etapas iniciales, el tutor es parte del equipo, y le ayudará a usted a encontrar soluciones. Con el paso del tiempo, usted se volverá cada vez más capaz y el tutor se dedicará cada vez más a revisar, en lugar de hacer. Por mi parte, mi meta como tutor es volverme innecesario.

Se pueden encontrar tutores para áreas específicas o para todo el proyecto, y desempeñarán su papel de tiempo completo o bien de medio tiempo. Muchos de ellos prefieren trabajar una semana al mes en cada proyecto; en cambio, otros consideran que una semana es poco tiempo. Busque a un tutor con experiencia y con la habilidad de transmitida. Él puede ser el factor más importante para el éxito de su proyecto; no olvide que vale la pena pagar por la calidad.

Si no puede conseguir un tutor, considere una revisión del proyecto cada dos meses, más o menos. Dentro de este plan, un tutor experimentado dedicará varios días a revisar los diversos aspectos del diseño. Durante este periodo, el revisor podrá detectar las áreas críticas, sugerir otras ideas y proponer técnicas útiles que el equipo no haya tenido en cuenta. Aunque esto no le da todos los beneficios de un buen tutor, puede ser valioso para señalar aspectos importantes susceptibles de mejorarse.

También podrá complementar sus habilidades mediante la lectura. Trate de leer un buen libro técnico, cuando menos cada dos meses. Mejor aún, intente leerlo como parte de un grupo dedicado a la lectura. Encuentre un par de personas que deseen leer el mismo libro. Acuerden leer un par de capítulos a la semana e inviertan un par de horas para analizados en conjunto. Con esta forma de lectura usted entenderá mejor la obra que leyéndola solo. Si usted es gerente, promueva esta forma de lectura. Consiga una habitación para el grupo; proporcione a su equipo dinero para comprar libros técnicos, y asigne el tiempo necesario para el grupo de lectura.

La comunidad de usuarios de patrones (patterns) ha descubierto que los grupos de lectura son particularmente valiosos. Han surgido varios grupos de lectura de patrones. Para más información sobre ellos, consulte la página base de patrones (<http://st-www.cs.uiuc.edu/users/pattems/pattems.html>).

A medida que avance en la elaboración de su proyecto, manténgase alerta en aquellas áreas en las que no tiene todavía habilidades ni experiencia suficientes. Planee adquirir la experiencia en el momento en que la necesite.

Manejo de los riesgos políticos

No puedo ofrecerle aquí ningún consejo importante, pues no soy un hábil político corporativo. Sin embargo, le recomiendo encarecida mente que busque alguien que lo sea.

Base arquitectónica

Un importante resultado de la elaboración es que se cuenta con una **base arquitectónica** para el sistema. Esta arquitectura se compone de:

- La lista de casos de uso, que le dice cuáles son los requerimientos.
- El modelo del dominio, que contiene lo que usted ha entendido sobre el negocio y sirve de punto de partida para las clases clave del dominio.
- La plataforma tecnológica, que describe las partes clave de la tecnología de implementación y la manera como se acoplan.

Esta arquitectura es el cimiento del desarrollo y funciona como anteproyecto de las etapas posteriores. Los detalles de la arquitectura cambiarán inevitablemente, sin que tenga que sufrir, sin embargo, muchos cambios importantes.

Sin embargo, la importancia de una arquitectura estable varía con la tecnología. En Smalltalk es posible efectuar cambios arquitectónicos significativos con mucha más facilidad, gracias a la rapidez con que suceden los ciclos edición-ejecución y a que no se requiere de una especificación estricta de los tipos de datos. Esto permite que la arquitectura sea muy evolutiva. En C++, es más importante tener una arquitectura estable subyacente a la construcción.

¿Cuándo se termina la elaboración?

Mi regla empírica es que la elaboración consume una quinta parte de la duración total del proyecto. Dos circunstancias son indicadores clave que señalan que se ha completado la elaboración:

- Los desarrolladores pueden tener la confianza necesaria para dar estimaciones, con un margen de hasta una semana-persona de esfuerzo, sobre el tiempo que tardará la construcción de cada caso de uso.
- Se han identificado todos los riesgos significativos y se han entendido los principales, al grado de que ya se sabe cómo tratados.

La planificación

La esencia de un plan es establecer una serie de iteraciones para la construcción y asignar los casos de uso a las iteraciones.

El plan se termina cuando todos los casos de uso han sido asignados a una iteración y cuando se ha identificado la fecha de inicio de todas las iteraciones. El plan no entra en mayores detalles.

El primer paso es clasificar los casos de uso por categoría.

- Los usuarios deberán indicar el nivel de prioridad de cada caso de uso. Por mi parte, suelo emplear tres niveles.
  - "Es indispensable tener esta función en cualquier sistema."
  - "Puedo vivir sin esta función por breve tiempo."
  - "Es una función importante, pero puedo sobrevivir sin ella durante un rato."
- Los desarrolladores deberán considerar el **riesgo arquitectónico** asociado con cada caso de uso, el cual consiste en que, si el caso de uso se deja de lado hasta muy avanzado el proyecto, el trabajo anterior se verá muy comprometido, lo que resultará en una gran cantidad de rediseño. Aquí, nuevamente, tiendo a establecer tres categorías de clasificación: alto riesgo, riesgo posible pero no probable y poco riesgo.
- Los desarrolladores deben evaluar la seguridad que tengan en la estimación del esfuerzo requerido para cada caso de uso. A esto lo llamo **riesgo de calendario.** También aquí considero valioso el uso de tres niveles:
  - "Estoy bastante seguro de saber cuánto tiempo tardará."
  - "Puedo estimar el tiempo sólo hasta el mes-persona más próximo."
  - "No tengo la menor idea de cuánto tiempo tarde."

Realizado lo anterior, habrá que estimar el tiempo de duración que tomará cada caso de uso, hasta la semana-persona más próxima. Al efectuar esta estimación, suponga que hay que efectuar análisis, diseño, codificación, pruebas individuales, integración y documentación. Suponga también que dispone de un desarrollador que trabaja de tiempo completo en el proyecto (más tarde agregaremos un factor de error).

Nótese que considero que quienes deben hacer las estimaciones son los desarrolladores, y no los gerentes. Como complemento de esta observación, asegúrese de que quien haga la estimación sea el desarrollador que posea el mayor conocimiento del caso de uso dado.

Una vez efectuadas las estimaciones, se puede decidir si se está listo para hacer el plan. Vea los casos de uso con mayor riesgo de calendario. Si gran parte del proyecto está atado a estos casos de uso o si estos casos tienen muchos riesgos arquitectónicos, entonces será necesaria una mayor elaboración.

El siguiente paso es la determinación de la longitud de iteración. Se busca una longitud de iteración fija para todo el proyecto, de modo que se pueda lograr un ritmo regular de entrega de iteraciones. Cada iteración debe ser lo suficientemente larga para realizar varios casos de uso. En el caso de **Smalltalk**, puede ser de un mínimo de dos a tres semanas, por ejemplo; para C++, puede ser de hasta seis a ocho semanas.

Ahora se puede considerar el esfuerzo para cada iteración.

Un buen punto de inicio es suponer que los desarrolladores operarán con un promedio de eficiencia del 50%; es decir, la mitad de su tiempo se dedicará al desarrollo de los casos de uso. Multiplique la longitud de la iteración por el número de desarrolladores y divida entre dos. El resultado será el esfuerzo de desarrollo por cada iteración. Por ejemplo, dados ocho desarrolladores y una longitud de iteración de tres semanas, tendrá 12 semanas-des arrollador «8*3) / 2) de esfuerzo por iteración.

Sume el tiempo de todos los casos de uso, divida entre el esfuerzo por iteración y sume uno por si hace falta. El resultado es su primera estimación de la cantidad de iteraciones que necesitará para su proyecto.

El siguiente paso es la asignación de los casos de uso a las iteraciones.

Los casos de uso de alta prioridad, riesgo arquitectónico y / o riesgos de calendario deben tratarse antes que los demás. ¡No deje los riesgos para el final! Tal vez necesite dividir los casos de uso más grandes y probablemente tenga que revisar las estimaciones de los casos de uso de acuerdo con el orden en el que está realizando las cosas. Tal vez tenga menos trabajo por hacer que el esfuerzo que requiere la iteración, pero nunca debe programar la ejecución de más de lo que le permita su esfuerzo.

Para la transición, asigne del 10% al 35% del tiempo de construcción a la afinación y el empaquetado para entrega (aplique una cifra mayor sino tiene experiencia en afinación y empaquetado en el ambiente actual).

Después, agregue un factor de contingencia: del 10% al 20% del tiempo de construcción, dependiendo de lo arriesgada que parezca ser la situación. Agregue este factor al final de la etapa de transición. Se debe planificarla entrega sin indicar el tiempo de contingencia (esto es, dentro de su fecha límite interna) pero comprométase a entregar al final del tiempo de contingencia.

Después de seguir todas estas instrucciones generales, usted deberá contar con un plan que muestre los casos de uso que se realizarán durante cada iteración. Este plan simboliza el compromiso entre los desarrolladores y los usuarios; un buen nombre para este plan es el de calendario de compromisos. Este itinerario no es inflexible; de hecho, todo el mundo debe esperar que el calendario de compromisos cambie a medida que avanza el proyecto. Sin embargo, debido a que se trata de un compromiso entre desarrolladores y usuarios, los cambios se deben hacer en conjunto.

Como podrá apreciar por todo lo anterior, los casos de uso son el fundamento de la planificación del proyecto, razón por la cual UML pone tanto énfasis en ellos.

Construcción

La construcción confecciona el sistema a lo largo de una serie de iteraciones. Cada iteración es un mini proyecto. Se hace el análisis, diseño, codificación, pruebas e integración de los casos de uso asignados a cada iteración. Ésta termina con una demostración al usuario y haciendo pruebas del sistema con el fin de confirmar que se han construido correctamente los casos de uso.

El propósito de este proceso es reducir el riesgo. Los riesgos surgen con frecuencia debido a que las cuestiones difíciles se posponen para el final del proyecto. He visto proyectos en los que la prueba y la integración se dejan para el final. Las pruebas y la integración son tareas mayores y generalmente tardan más de lo que se cree.

Fred Brooks estimaba, allá en la época del OS / 360, que la mitad del tiempo de un proyecto se iba en pruebas (con la inevitable depuración). Las pruebas y la integración son más difíciles cuando se dejan para el final, y son más desmoralizadoras.

Todo este esfuerzo va acompañado de un gran riesgo. Con el desarrollo iterativo, se lleva a cabo todo el proceso para cada iteración, lo que produce el hábito de lidiar con todos los aspectos cada vez.

Cuanto más envejezco, más agresivo me vuelvo en lo que respecta a las pruebas. Me gusta la regla empírica de Kent Beck que afirma que un desarrollador debería escribir cuando menos la misma cantidad de código de pruebas que de producción. El proceso de pruebas debe ser continuo. No se debe escribir ningún código hasta

saber cómo probado. Una vez escrito, haga sus pruebas hasta que éstas funcionen, no se podrá considerar que se haya terminado de escribir el código.

Una vez escrito, el código de pruebas debe conservarse para siempre. Prepare su código de pruebas de forma que pueda ejecutarlas todas con un comando simple o mediante la pulsación de un botón GUI. El código debe responder con "OK" o con una lista de fallas. Además, las pruebas deben revisar sus propios resultados. No hay mayor pérdida de tiempo que tener que investigar el significado de un número enviado a la salida por una prueba.

Divida las pruebas en pruebas individuales y de sistema. Las pruebas individuales deberán ser escritas por los desarrolladores. Habrán de organizarse con base en paquetes y codificarse de modo que prueben todas las clases de las interfaces. Las pruebas de sistemas deberán ser desarrolladas por un equipo pequeño independiente cuya única tarea sea probar. Este equipo deberá tener una visión completa del sistema y sentir un placer especial en descubrir fallas. (Los bigotes retorcidos y las risas siniestras, aunque opcionales, son deseables.)

Las iteraciones dentro de la construcción son tanto incrementales como iterativas.

- Funcionalmente, las iteraciones son incrementales. Cada iteración se construye sobre los casos de uso desarrollados en las iteraciones anteriores.
- Son iterativas en términos del código de base. Cada iteración implicará la reescritura de algún código ya existente con el fin de hacerlo más flexible. **La reestructuración de factores** es una técnica muy útil para la iteración del código. Es buena idea saber la cantidad de código desechado con cada iteración. Desconfíe si se descarta menos del 10% del código en cada ocasión.

La integración debe ser un proceso continuo. Para los principiantes, la integración total es parte del fin de cada iteración. Sin embargo, puede y debe ocurrir con mayor frecuencia. El desarrollador deberá integrar después de cada pieza de trabajo importante. La secuencia completa de pruebas individuales deberá ejecutarse con cada integración, a fin de garantizar pruebas regresivas completas.

Desarrollo y planificación iterativos

También se puede hacer, con cada iteración, una planificación más detallada. Una de las partes fundamentales de todo programa es enfrentar cosas que no van de acuerdo con lo planeado. Reconozcámoslo, siempre sucede así.

La característica clave del desarrollo iterativo es que los tiempos están limitados; no se permite retrasar ninguna fecha. En cambio, los casos de uso se pueden trasladar a una iteración posterior mediante negociaciones y acuerdos con el patrocinador. De lo que se trata es de habituarse a cumplir con las fechas, evitando la mala costumbre de retrasadas.

Sin embargo, nótese que si se están posponiendo demasiados casos de uso, es momento de rehacer el plan, incluyendo la reestimación de los niveles de esfuerzo de los casos de uso. En esta etapa, el desarrollador ya deberá tener una mejor idea de cuánto tardarán las cosas.

Empleo del UML en la construcción

Todas las técnicas del UML son útiles durante esta etapa. Debido a que me referiré a técnicas sobre las que no he hablado aún, siéntase con libertad para saltarse esta sección y volver a ella después.

Cuando se contemple la adición de un caso de uso específico, utilice, en primer lugar, el caso de uso para determinar su alcance. Un diagrama conceptual de clases diagrama conceptual de clases puede ser útil para

esbozar algunos conceptos del caso de uso y ver cómo encajan en el software que ya se ha construido. Si el caso de uso contiene elementos importantes del flujo de trabajo, se pueden ver mediante un diagrama de actividades

La ventaja de estas técnicas durante esta etapa es que se pueden usar en conjunto con el experto del dominio. Como dice Brad Kain: el análisis sólo se hace cuando el experto del dominio se encuentra en la sala (de otra forma, se trata de seudo análisis).

He descubierto que, para pasar al diseño, un diagrama de clases correspondiente a laperspectiva de especificaciones puede ser útil para proyectar con mayor detalle las clases. Los diagramas de interacción son valiosos para mostrar la manera en que interactuarán las clases para implementar el caso de uso.

Se pueden dibujar directamente los diagramas de clases y los diagramas de interacción o emplear las tarjetas CRC para indagar su comportamiento y después documentados mediante diagramas, si se desea. Cualquiera que sea el enfoque escogido, me parece importante prestar mucha atención a las responsabilidades en esta etapa del trabajo.

Considero que los diagramas UML son valiosos para entender de manera general el sistema. Sin embargo, debo subrayar que no creo en el hecho de diagramar de manera detallada todo el sistema. Citando a Ward Cunningham (1996):

> *Los memorandos cuidadosamente escritos y seleccionados pueden sustituir con toda facilidad a la tradicional documentación detallada del diseño. Esta última es sobresaliente en pocas ocasiones, excepto en algunos puntos aislados. Destaque estos puntos... y olvídese del resto.*

Restrinja la documentación a aquellas áreas en las que es útil. Si descubre que la documentación no le está ayudando, es señal de que algo anda mal.

Yo me apoyo en un diagrama de paquetes como mapa lógico del sistema. Este diagrama me ayuda a comprender las piezas lógicas del sistema y a ver las dependencias (y mantenerlas bajo control).

Me gusta utilizar herramientas que me ayuden a identificar las dependencias y asegurarme de no pasarlas por alto. Incluso herramientas sencillas, como los scripts scripts de Perl, pueden ser de utilidad. Java, con su manejo explícito de paquetes, es una gran ayuda. También puede ser de utilidad en esta etapa un diagrama de emplazamiento, el cual muestra el cuadro físico de alto nivel.

En cada paquete me gusta ver un diagrama de clases de perspectiva de especificaciones. No muestro todas las operaciones sobre todas las clases, sino sólo las asociaciones y los atributos y operaciones clave que me ayudan a comprender lo que hay allí.

Este diagrama de clases actúa como índice gráfico de materias. Muchas veces ayuda a mantener un glosario de clases que contiene definiciones breves de cada clase, con frecuencia por medio de declaraciones de responsabilidades. También es una buena idea mantener las declaraciones de responsabilidades en el código, en forma de comentarios, los cuales se pueden extraer con alguna herramienta adecuada.

Si una clase tiene un comportamiento de ciclo de vida complicado, lo describo dibujando un diagrama de estado . Sólo lo hago si el comportamiento es lo bastante complejo, lo cual no sucede con mucha frecuencia. Lo que es más común son las interacciones complicadas entre clases, para las que genero un diagrama de interacciones.

Mi forma favorita de documentación tiene tres elementos básicos:

1. Una o dos páginas que describen unas cuantas clases en un diagrama de clases.
2. Unos cuantos diagramas de interacción que muestren cómo colaboran las clases.
3. Cierto texto para integrar los diagramas.

Con frecuencia incluyo algún código importante, escrito en estilo de programa literario. Si está implicado un algoritmo particularmente complejo, consideraré la posibilidad de utilizar un diagrama de actividades , pero sólo si me ayuda a entender mejor que el propio código. En esos casos, utilizo un diagrama de clases correspondiente a la perspectiva de especificación o a la perspectiva de implementación, o tal vez ambos; todo depende de lo que esté tratando de comunicar.

En Objectory, se deberán dibujar diagramas de interacción para cada caso de uso que se identifique. Estos diagramas de interacción deberán cubrir todos los escenarios. No se necesita un diagrama individual para cada escenario, pero asegúrese de que la lógica de todos ellos se capte en el diagrama de interacciones que abarque los casos de uso asociados.

Si descubro conceptos que aparecen continuamente, capto las ideas básicas mediante **patrones**

Los patrones son útiles dentro del alcance de un proyecto y también para la comunicación de buenas ideas fuera del proyecto. De hecho, considero que los patrones son particularmente valiosos también para comunicación entre proyectos.

Transición

De lo que se trata en el desarrollo iterativo es de hacer todo el proceso de desarrollo consistentemente, de tal modo que el equipo del desarrollo se acostumbre a entregar código terminado. Pero hay varias cosas que no deben hacerse demasiado pronto. Un ejemplo primordial es la optimización.

La optimización reduce la claridad y la capacidad de ampliación del sistema con el objeto de mejorar el desempeño. Ésta es una concesión necesaria; a fin de cuentas, un sistema debe ser lo suficientemente rápido para satisfacer las necesidades de los usuarios, pero la optimización demasiado temprana hace más complicado el desarrollo. Así que esto es algo que ha de dejarse para el final.

Durante la transición, no se hacen desarrollos para añadir funciones nuevas (a menos que sean pequeñas y absolutamente indispensables). Ciertamente, sí hay desarrollo para depuración.

Un buen ejemplo de una fase de transición es el tiempo entre la liberación beta y la liberación definitiva del producto.

Cuándo se debe usar el desarrollo iterativo

El desarrollo iterativo únicamente se debe utilizar en aquellos proyectos que se quiere que tengan éxito.

Tal vez esto suene un tanto absurdo, pero a medida que envejezco, me he vuelto más intransigente en cuanto al manejo del desarrollo iterativo. Bien realizado, es una técnica esencial, que puede servir para exponer pronto los riesgos y lograr un mejor control sobre el desarrollo. No es lo mismo que no contar con alguna administración (aunque, para ser sinceros, debo señalar que algunos lo han usado de ese modo). Debe planificarse bien. Pero se trata de un enfoque sólido y todos los libros sobre el desarrollo de la OO lo recomiendan, por buenas razones.

Para mayor información

Entre mis fuentes favoritas están Goldberg y Rubin (1995), Booch (1994), McConnell (1996) y Graham (1993).

- Goldberg y Rubin hablan mucho sobre los principios generales y cubren mucho terreno.
- El libro de Booch es más directo. Cuenta lo que hace su autor y da muchos consejos.
- McConnell también presenta muchos consejos, en general ajustados al tipo de proceso descrito en este capítulo.
- Si desea una metodología completamente definida, paso a paso, del desarrollo iterativo, el mejor material que he consultado sobre estos temas se encuentra en la obra de Graham.

Rational Software tiene una descripción no publicada de la versión actual del proceso Objectory como producto. Entre los libros publicados, la descripción más cercana es la de Jacobson (1994 y 1995). También debe dar una ojeada a los artículos sobre patrones contenidos en Coplien y Schmidt (1995), así como el artículo sobre "episodios" de Ward Cum Üngham (1996).

Kent Beck está trabajando en un libro sobre patrones de administración de proyectos. Cuando se publique, sin duda será un recurso excelente. De hecho, muchas ideas en este capítulo surgieron en conversaciones con él y con Ward Cunningham, así como en conversaciones telefónicas con Ivar Jacobson.

Los casos de uso

Resumen

Un caso de uso viene a ser cada uno de los requerimientos del sistema, cada una de las "cosas" que queremos que haga nuestro programa.

Es una de los diagramas fundamentales que debe redactarse durante la fase de elaboración

Índice de Contenidos

Introducción

Los casos de uso son un fenómeno interesante. Durante mucho tiempo, tanto en el desarrollo orientado a objetos como en el tradicional, las personas se auxiliaban de escenarios típicos que les ayudaban a comprender los requerimientos. Estos escenarios, sin embargo, se trataban de modo muy informal; siempre se construían, pero pocas veces se documentaban. Ivar Jacobson es muy conocido por haber cambiado esto con su método Objectory y su primer libro sobre el tema.

Jacobson elevó la visibilidad del caso de uso (su nombre para un escenario) a tal punto que lo convirtió en un elemento primario de la planificación y el desarrollo de proyectos. Desde la publicación de su libro (1994), la comunidad de los objetos ha adoptado los casos de uso en un grado notable. El ejercicio de mi profesión ciertamente ha mejorado gracias a que comencé a emplear los casos de uso de este modo.

Veamos, pues, ¿qué es un caso de uso?

Un **caso de uso** es, en esencia, una interacción típica entre un usuario y un sistema de cómputo. Considérese el procesador de palabras con el que escribo estas líneas que usted lee. Dos casos de uso típicos serían "pon una parte del texto en negritas" y "crea un índice". Por medio de estos ejemplos, se puede uno dar una idea de ciertas propiedades de los casos de uso.

- El caso de uso capta alguna función visible para el usuario.
- El caso de uso puede ser pequeño o grande.
- El caso de uso logra un objetivo discreto para el usuario

En su forma más simple, el caso de uso se obtiene hablando con los usuarios habituales y analizando con ellos las distintas cosas que deseen hacer con el sistema. Se debe abordar cada cosa discreta que quieran, darle un nombre y escribir un texto descriptivo breve (no más de unos cuantos párrafos).

Durante la elaboración, esto es todo lo que necesitará para empezar. No trate de tener todos los detalles justo desde el principio; los podrá obtener cuando los necesite. Sin embargo, si considera que un caso de uso dado tiene ramificaciones arquitectónicas de importancia, necesitará más detalles a la mano. La mayoría de los casos de uso se pueden detallar durante la iteración dada, a medida que se construyen.

Objetivos del usuario e interacciones con el sistema

Un tema importante que me he encontrado en los casos de uso es la diferencia entre lo que llamo objetivos del usuario e interacciones con el sistema. Por ejemplo, considere la funcionalidad de la style sheet hoja de estilos con que cuentan la mayor parte de los procesadores de texto.

Las **interacciones con el sistema** permiten decir que los casos de uso incluirán cosas como: "*define estilo*", "*cambia estilo*", y "*mueve un estilo de un documento a otro*". Sin embargo, estos casos de uso reflejan más bien cosas que el usuario hace con el sistema, en lugar de los verdaderos objetivos que el usuario trata de conseguir. Los verdaderos **objetivos del usuario** se describirían con términos como "*garantizar el formateo consistente de un documento*" y "*hacer que el formato de un documento sea igual que el de otro*".

Esta dicotomía entre objetivo del usuario e interacción con el sistema no se presenta en todas las situaciones. Por ejemplo, el proceso de indización de un documento es muy parecido si se le considera como objetivo del usuario o como interacción con el sistema. No obstante, cuando los objetivos del usuario y las interacciones del sistema difieren, es muy importante tener clara la diferencia.

Ambos estilos de casos de uso tienen sus aplicaciones. Los casos de uso de interacción con el sistema sirven más para fines de planificación; conviene no perder de vista los objetivos del usuario, con el fin de poder considerar formas alternas para el cumplimiento de tales objetivos. Si se llega demasiado pronto a la interacción con el sistema, recurriendo a la primera alternativa obvia, se pasarán por alto otras maneras creativas de cumplir con mayor eficacia los objetivos del usuario. En todos los casos, es una buena idea preguntarse" ¿por qué hicimos esto?" Esta pregunta generalmente conduce a una mejor comprensión del objetivo del usuario.

En mi trabajo, me centro primero en los objetivos del usuario y después me ocupo de encontrar casos de uso que los cumplan. Hacia el final del periodo de elaboración, espero tener por lo menos un conjunto de casos de

uso de interacción con el sistema por cada objetivo de usuario que he identificado (como mínimo, para las metas del usuario que pretendo manejar en la primera entrega).

Diagramas de casos de uso

Jacobson (1994), además de introducir los casos de uso como elementos primarios del desarrollo del software, también diseñó un diagrama para la representación gráfica de los casos de uso. **El diagrama de casos de uso** es ya también parte del UML.

La Figura 3-1 muestra algunos de los casos de uso de un sistema de financiamiento.

Comenzaré el análisis de los elementos de este diagrama hablando sobre los actores.

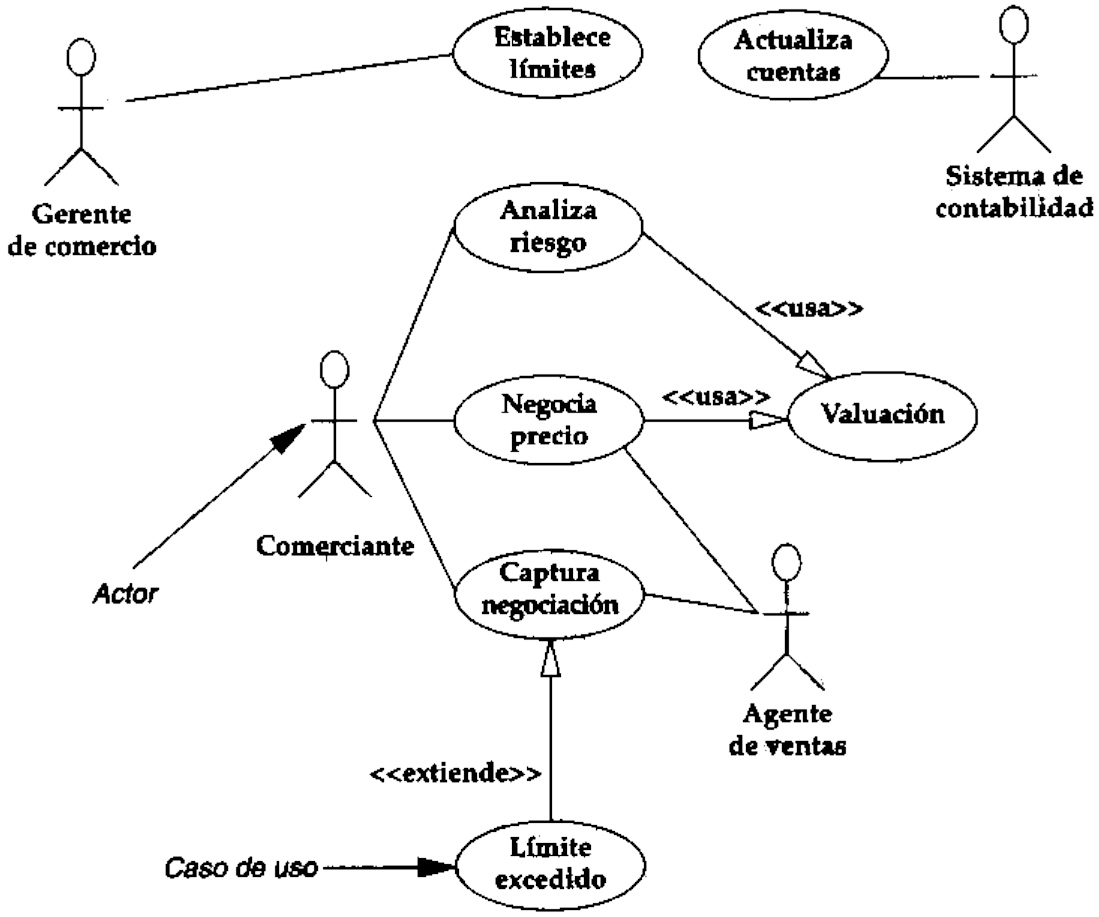

Figura 3-1: *Diagrama de casos de uso*

Figura 3-1 Diagrama de Uso

Actores

Empleamos el término **actor** para llamar así al usuario, cuando desempeña ese papel con respecta al sistema. Hay cuatro actores en la Figura 3-1: el gerente de comercial, el comerciante, el agente de ventas y el sistema de contabilidad.

En la mencionada organización, probablemente habrá diversos comerciantes. Además, un usuario puede desempeñar varios papeles. Por ejemplo, un comerciante de edad madura podría desempeñar el papel de gerente de comercio y además ser un comerciante normal. Por otra parte, un comerciante puede ser también agente de ventas. Cuando se trata con actores, conviene pensar en los papeles, no en las personas ni en los títulos de sus puestas.

Los actores llevan a cabo casos de uso. Un mismo actor puede realizar muchos casos de uso; A la inversa, un caso de uso puede ser realizado Por varios actores.

En la práctica, los actores son muy útiles cuando se trata de definir los casos de uso. Al enfrentarse con un sistema grande, puede ser difícil obtener una lista de casos de uso. Es más fácil en tales situaciones definir la lista de los actores y después tratar de determinar los casos de uso de cada actor.

Obsérvese que no es necesario que los actores sean seres humanos, a pesar de que los actores estén representados por figuras humanas en el diagrama del caso de uso. El actor puede ser también un sistema externo que necesite cierta información del sistema actual. En la figura 3-1 se puede apreciar la necesidad de actualizar las cifras del sistema de contabilidad.

El tema de las interacciones con sistemas externos produce mucha confusión y variaciones de estilo entre los usuarios de los diagramas de casos de uso. .

1. Algunos sienten que todas las interacciones con sistemas remotas deben aparecer en el diagrama. Por ejemplo, si se necesita acceder a Reuters con el fin de cotizar un contrato, se deberá mostrar el vínculo entre el caso Negocia precio y Reuters.
2. Algunas personas consideran que sólo se deben mostrar los casos de uso con interacción externa, cuando quien inicia el contacto es otro sistema. Según esta regla, sólo se mostraría el caso de uso del sistema de contabilidad si dicho sistema invocara algún proceso que le dijera al sistema fuente que la hiciera.
3. Algunas personas consideran que sólo se deben mostrar los actores del sistema cuando ellas sean las que necesiten el caso de uso. Por tanto, si el sistema genera cada noche un archivo que después es recogido por el sistema de contabilidad, entonces éste es el actor que corresponde, debido a que es quien necesita el archivo generado.
4. Otros más sienten que constituye un enfoque equivocado considerar que un sistema es un actor. Por el contrario, consideran que un actor es un usuario que desea algo del sistema (Por ejemplo, un archivo en particular). En el caso de nuestro ejemplo de sistema, los actores serían los auditores internos de la compañía.

Tomando en cuenta todos los factores, me inclino por la opción 3.

Todos los casos de uso tratan sobre funcionalidad requerida externamente. Si el sistema de contabilidad necesita un archivo, entonces ése es un requerimiento que debe satisfacerse.

El acceso a Reuters es importante, pero no una necesidad del usuario. Si sigue la opción 4, se termina por analizar el sistema de contabilidad, cosa que probablemente usted no quisiera hacer. Dicho lo anterior, siempre se deben cuestionar los casos de uso con los actores del sistema, descubrir los objetivos reales del usuario y considerar formas alternas para lograrlos.

Cuando trabaje con actores y casos de uso, no se preocupe mucho por la relación exacta entre ellos. Lo que le interesa casi siempre son los casos de uso; los actores son sólo un modo de llegar a ellos. Siempre y cuando obtenga todos los casos de uso, no se preocupe por los detalles acerca de los actores.

Sin embargo, una situación en la que los actores sí desempeñan un papel es en la configuración del sistema para varios tipos de usuarios. Si el sistema tiene casos de uso que corresponden a funciones de usuario de alto nivel,

usted puede emplear los vínculos entre casos de uso y actores para hacer los perfiles de los usuarios. Cada usuario tendría una lista asociada de nombres de actores con la que se determinarían los casos de uso que puede ejecutar cada uno.

Otra buena razón para rastrear a los actores es la necesidad de saber cuáles son los casos de uso que quiere cada uno. Esto puede ser importante cuando usted esté evaluando las necesidades que compiten entre ellos. La comprensión de los actores puede servir para negociar entre demandas de desarrollo que compiten entre sí. También pueden ser útiles para especificar una política de seguridad.

Algunos casos de uso no tienen vínculos claros con actores específicos. Considérese una compañía de servicios públicos. Por supuesto, uno de sus casos de uso es "envío de factura". No es tan fácil, sin embargo, identificar un actor asociado. No existe un papel de usuario en particular que solicite una factura. La factura se envía al cliente, pero el cliente no protestaría si esto no se llevara a cabo. Lo que más se parece a un actor es el Departamento de facturación, en el sentido de que obtiene un valor del caso de uso. Pero la facturación generalmente no interviene en la ejecución del caso de uso.

Los actores pueden tener varios papeles con respecto a un caso de uso. Pueden ser los que obtienen un valor del caso de uso, o tal vez sólo participen en él. Dependiendo de cómo se utilice la relación entre los actores será la importancia de los papeles que desempeñen los actores. Por mi parte, suelo preocuparme más por controlar el desarrollo del sistema. Así, en general, siento un mayor interés por quién quiere que se construya un caso de uso (por lo general, quienes obtienen un valor del caso de uso).

La clave es tener en cuenta que algunos casos de uso no saltarán a la vista como resultado de ponerse á pensar en los casos de uso de cada actor. Si esto sucede, no se preocupe demasiado. Lo importante es comprender los casos de uso y los objetivos del usuario que satisfacen.

Una buena fuente para identificar los casos de uso son los eventos externos. Piense en todos los eventos del mundo exterior ante los cuales usted quiera reaccionar. Un evento dado puede provocar una reacción en el sistema en la que no intervenga el usuario, o puede causar una reacción que provenga principalmente de los usuarios. La identificación de los eventos ante los que se necesitará reaccionar será de ayuda en la identificación de los casos de uso.

Uses y extends

Además de los vínculos entre los actores y los casos de uso, hay otros dos tipos de vínculos en la Figura 3-1. Éstos representan las relaciones de uses (usa) y extends (extiende) entre los casos de uso. Con frecuencia, tales relaciones son fuente de confusión para quienes mezclan los significados de ambos conceptos, de modo que tómese el tiempo necesario para comprenderlos.

Se usa la relación **extends** cuando se tiene un caso de uso que es similar a otro, pero que hace un poco más.

En nuestro ejemplo, el caso de uso es Captura negociación. Éste es un caso en que todo sucede sin contratiempos. Sin embargo, hay situaciones que pueden estropear la captura de una negociación. Una de ellas surge cuando se excede algún límite, por ejemplo, la cantidad máxima que la organización de comercio ha establecido para un cliente en particular. En este ejemplo, dado que no efectuamos el comportamiento habitual asociado con dicho caso de uso, efectuamos una variación.

Podríamos poner esta variación dentro del caso de uso Captura negociación. Sin embargo, esto llenaría dicho caso de uso de una gran cantidad de lógica especial, la cual oscurecería el flujo "normal".

Otro modo de abordar la variación es poner la conducta normal en un caso y la conducta inusual en cualquier otro lado. A continuación describiremos la esencia de la relación extends.

1. Primero obtenga el caso de uso simple y normal.
2. En cada paso de ese caso de uso, pregúntese "¿Qué puede fallar aquí?", "¿Cómo podría funcionar esto de modo diferente?"
3. Dibuje todas las variaciones como extensiones del caso de uso dado. Con frecuencia habrá un buen número de extensiones. Si se listan por separado serán mucho más fáciles de entender.

Puede suceder que se tenga qué dividir un caso de uso tanto en la etapa de elaboración como en la de construcción. Durante la elaboración, suelo dividir los casos de uso que se estén volviendo muy complicados.

En la etapa de construcción del proyecto (Enlace con un H2 del capítulo 2), realizo la división cuando no puedo construir un caso de uso completo en una sola iteración. Divido los casos de uso complejos en un caso normal y unas cuantas extensiones, y luego construyo el caso normal en una sola iteración y las extensiones como partes de una o varias iteraciones posteriores. (Por supuesto, aquí sucederán cambios en el plan de compromisos, lo que deberá negociarse con los usuarios.)

Las relaciones **uses** ocurren cuando se tiene una porción de comportamiento que es similar en más de un caso de uso y no se quiere copiar la descripción de tal conducta. Por ejemplo, tanto Analiza riesgo como Negociación precio requieren valuar la negociación. La descripción de la valuación de la negociación requiere de bastante escritura, y yo odio las operaciones de copiar y pegar. Por tanto, elaboro un caso de uso separado, Negociación valor, para esta situación y me refiero a él desde los casos de uso originales.

Adviértanse las diferencias entre extends y uses. Ambos implican la **factorización** de comportamientos comunes de varios casos de uso, dejando un solo caso de uso común que es empleado, o extendido, por otros varios casos de uso. No obstante, la intención es la que cambia.

En sus vínculos con los actores, estos dos tipos de relación implican asuntos diferentes. Tratándose de la relación extends, los actores tienen que ver con los casos de uso que se están extendiendo. Se supone que un actor dado se encargará tanto del caso de uso base como de todas las extensiones. En cuanto a las relaciones uses, es frecuente que no haya un actor asociado con el caso de uso común. Incluso si lo hay, no se considera que esté llevando a cabo los demás casos de uso.

Aplique las siguientes reglas.

- Utilice extends cuando describa una variación de conducta normal.
- Emplee uses para repetir cuando se trate de uno o varios casos de uso y desee evitar repeticiones.

Es probable que oiga usted el término **escenario** en conexión con los casos de uso. Cabe aclarar que este término se emplea de manera inconsistente. Algunas veces, escenario se usa como sinónimo de caso de uso. En el contexto del UML, la palabra escenario se refiere a una sola ruta a través de un caso de uso, una ruta que muestra una particular combinación de condiciones dentro de dicho caso de uso. Por ejemplo, para ordenar algunas mercancías, tendremos un solo caso de uso con varios escenarios asociados: uno en el cual todo va bien; otro donde no hay suficientes mercancías; otro en el que nuestro crédito es rechazado, y así por el estilo.

Conforme vaya realizando sus tareas de modelado, encontrará modelos que expresen la manera de hacer sus casos de uso, ya sea entre el software o entre la gente. Es evidente que hay más de una manera de llevar a cabo un caso de uso. En la jerga del UML, decimos que un caso de uso puede tener muchas **realizaciones.**

Con frecuencia se realizan varios bosquejos sobre las distintas formas de resolver un caso de uso, con el fin de discutirlas y determinar con cuál va a trabajar. Si hace esto, recuerde guardar información acerca de las

realizaciones descartadas, incluyendo las notas de por qué se descartaron. No quisiera contarles cuántas horas he desperdiciado en discusiones del estilo de "yo sé que existe una razón por la que no hicimos eso, pero..."

Cuándo emplear casos de uso

No puedo imaginar en este momento una situación en la que no emplearía los casos de uso. Son una herramienta esencial para la captura de requerimientos, la planificación, o el control de proyectos iterativos. La captura de los casos de uso es una de las tareas principales durante la fase de elaboración Enlace con cap2 un H2); de hecho, es lo primero que se debe hacer.

La mayoría de los casos de uso se generarán durante esa fase del proyecto, pero irá descubriendo otros a medida que avance. Esté siempre pendiente de ellos. Todo caso de uso es un requerimiento potencial y hasta que no haya usted capturado un requerimiento, no podrá planear cómo manejarlo en el proyecto.

Algunos prefieren listar y analizar los casos de uso primero, y luego llevar a cabo un poco de modelado. Yo a veces lo hago, pero también he encontrado que el modelado conceptual con los usuarios ayuda a descubrir los casos de uso. Mi recomendación es que se prueben ambas maneras y se escoja la que mejor sirva.

Los diseñadores emplean casos de uso con distintos grados de granularidad. Por ejemplo, Ivar Jacobson dice que en un proyecto de lo personas-año, él esperaría unos 20 casos de uso (sin contar con las relaciones use y extend). En un proyecto reciente de la misma magnitud, tuve más de 100 casos de uso. Prefiero los casos de uso con menor granularidad, pues con ellos es más fácil trabajar sin perder de vista el calendario de compromisos. Sin embargo, demasiados casos de uso pueden acabar siendo abrumadores. Por el momento no creo que haya una respuesta correcta, de modo que le recomiendo que sea flexible y trabaje con lo que le resulte más cómodo.

Para mayor información

Me parece que el mundo sigue a la espera de un libro realmente bueno sobre casos de uso. Por supuesto, el primer libro de Jacobson (1994) es una fuente valiosa, ya que constituye en realidad el libro con el que comenzó todo. El siguiente libro de Jacobson (1995) es útil por su énfasis en los casos de uso de procesos de negocios (que, según opiniones encontradas, deben usarse todo el tiempo). Ian Graham (1993) también incluye algunos buenos consejos (él emplea el término "script" en lugar del de "caso de uso"). Además, le recomiendo que consulte los textos de casos de uso en el sitio de Web de Alistair Cockburn:

DIAGRAMAS DE CLASE – FUNDAMENTOS

Resumen

El diagrama de clase describe los tipos de objetos que hay en el sistema y las diversas clases de relaciones estáticas que existen entre ellos. Hay dos tipos principales de relaciones estáticas:

- **Asociaciones** (por ejemplo, un cliente puede rentar diversas videocintas).
- **Subtipos** (una enfermera es un tipo de persona).

Los diagramas de clase también muestran los atributos y operaciones de una clase y las restricciones a que se ven sujetos, según la forma en que se conecten los objetos.

Índice de Contenidos

Introducción

La técnica del **diagrama de clase** se ha vuelto medular en los métodos orientados a objetos. Virtualmente, todos los métodos han incluido alguna variación de esta técnica.

El diagrama de clase, además de ser de uso extendido, también está sujeto a la más amplia gama de conceptos de modelado. Aunque los elementos básicos son necesarios para todos, los conceptos avanzados se usan con mucha menor frecuencia. Por eso, he dividido mi estudio de los diagramas de clase en dos partes; los fundamentos (en el presente capítulo) y los conceptos avanzados los conceptos avanzados (véase el capítulo siguiente).

El diagrama de clase describe los tipos de objetos que hay en el sistema y las diversas clases de relaciones estáticas que existen entre ellos. Hay dos tipos principales de relaciones estáticas:

- **Asociaciones** (por ejemplo, un cliente puede alquilar diversas videocintas).
- **Subtipos** (una enfermera es un tipo de persona).

Los diagramas de clase también muestran los atributos y operaciones de una clase y las restricciones a que se ven sujetos, según la forma en que se conecten los objetos.

Los diversos métodos OO utilizan terminologías diferentes (y con frecuencia antagónicas) para estos conceptos. Se trata de algo sumamente frustrante pero inevitable, dado que los lenguajes OO son tan desconsiderados como los métodos. Es en esta área que el UML aportará algunos de sus mayores beneficios, al simplificar estos diferentes diagramas.

La figura 4-1 muestra un diagrama de clase típico.

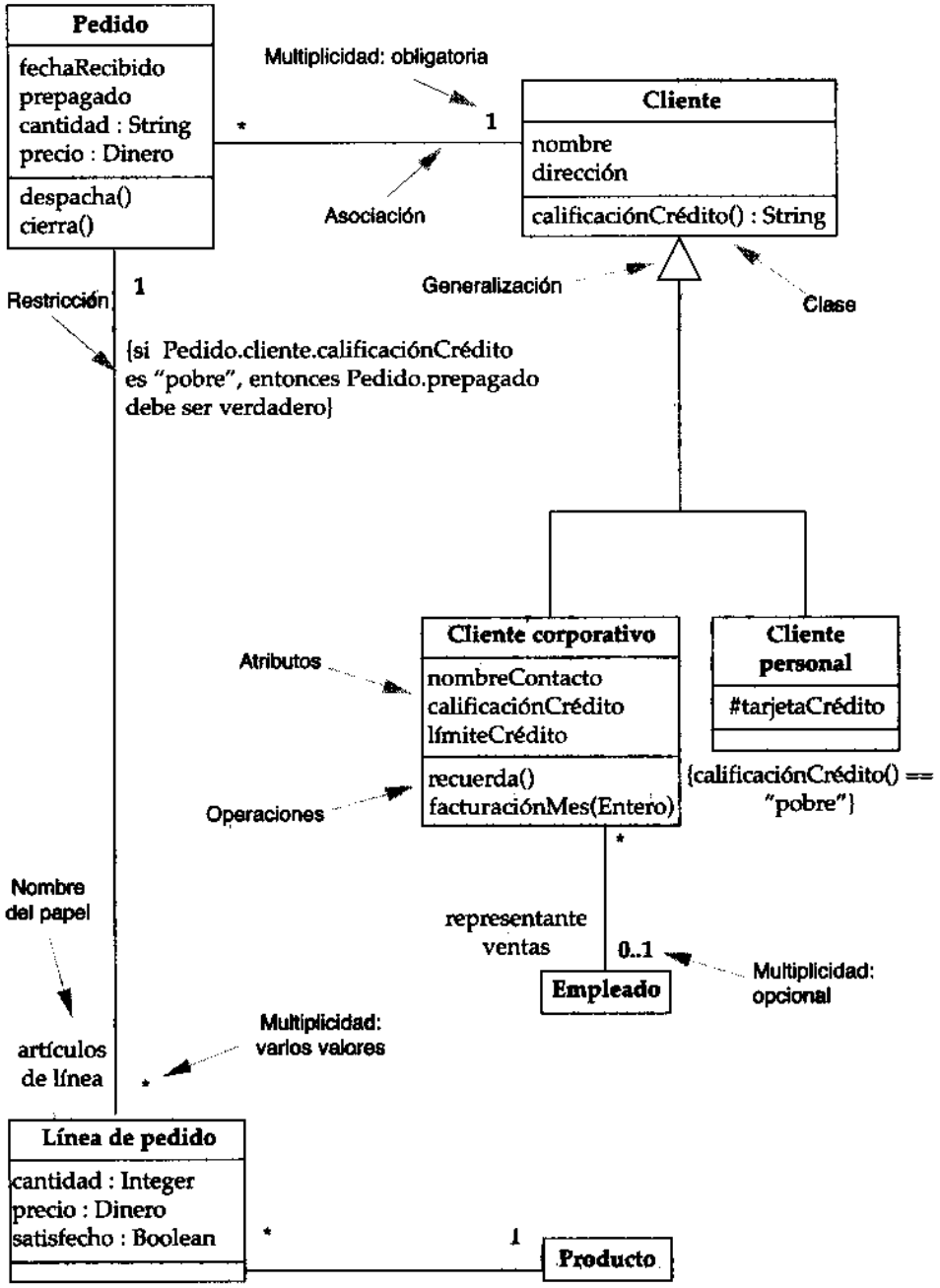

Figura 4-1: *Diagrama de clase*

Figura 4-1 Diagrama de clase

Perspectivas

Antes de empezar a describir los diagramas de clase, quisiera señalar una importante sutileza sobre el modo en que se usan. Tal sutileza generalmente no está documentada, pero tiene sus repercusiones en el modo en que debe interpretarse un diagrama, ya que se refiere a lo que se va a describir con un modelo.

Siguiendo la recomendación de Steve Cook y John Daniels (1994), considero que hay tres perspectivas que se pueden manejar al dibujar diagramas de clase (o, de hecho, cualquier modelo, aunque esta división se advierte de modo especial en relación con los diagramas de clase).

- **Conceptual.** Si se adopta la perspectiva conceptual, se dibuja un diagrama que represente los conceptos del dominio que se está estudiando. Estos conceptos se relacionan de manera natural con las clases que los implementan, pero con frecuencia no hay una correlación directa. De hecho, los modelos conceptuales se deben dibujar sin importar (o casi) el software con que se implementarán, por lo cual se pueden considerar como independientes del lenguaje. (Cook y Daniels llaman perspectiva esencial a esto; por mi parte, empleo el término" conceptual", pues se ha usado durante mucho tiempo.)
- **Especificación.** Ahora estamos viendo el software, pero lo que observamos son las interfaces del software, no su implementación, por tanto, en realidad vemos los tipos, no las clases. El desarrollo orientado a objetos pone un gran énfasis en la diferencia entre interfaz e implementación, pero esto con frecuencia se pasa por alto en la práctica, ya que el concepto de clase en un lenguaje OO combina tanto la interfaz como la implementación. Así, a menudo se hace referencia a las interfaces como tipos y a la implementación de esas interfaces como clases. Influidos por este manejo del lenguaje, la mayor parte de los métodos han seguido este camino. Esto está cambiando (Java y CORBA tendrán aquí cierta influencia), pero no con suficiente rapidez. Un tipo representa una interfaz que puede tener muchas implementaciones distintas, debido por ejemplo, al ambiente de implementación, características de desempeño y proveedor. La distinción puede ser muy importante en diversas técnicas de diseño basadas en la delegación.
- **Implementación.** Dentro de esta concepción, realmente tenemos clases y exponemos por completo la implementación. Ésta es, probablemente, la perspectiva más empleada, pero en muchos sentidos es mejor adoptar la perspectiva de especificación.

La comprensión de la perspectiva es crucial tanto para dibujar como para leer los diagramas de clase. Desdichadamente, las divisiones entre las perspectivas no son tajantes y la mayoría de los modeladores no se preocupa por definidas con claridad cuando las dibujan.

Cuando usted dibuje un diagrama, hágalo desde el punto de vista de una sola perspectiva clara. Cuando lea un diagrama, asegúrese de saber desde qué perspectiva se dibujó. Dicho conocimiento es esencial si se quiere interpretar correctamente el diagrama.

La perspectiva no es parte del UML formal, pero considero que es extremadamente valiosa al modelar y revisar los modelos. El UML se puede utilizar con las tres perspectivas. Mediante el etiquetado de las clases con un estereotipo , se puede dar una indicación clara de la perspectiva. Las clases se marcan con «clase de implementación» para mostrar la perspectiva de implementación y con «tipo» para el caso de la perspectiva de especificación y la conceptual. La mayor parte de los usuarios de los métodos OO adopta una perspectiva de implementación, lo cual es de lamentar, porque las otras perspectivas con frecuencia son más útiles.

La tabla 4-1 lista cuatro términos del UML que aparecen en la Figura 4-1 y sus términos correspondientes en otras metodologías bien establecidas.

Tabla 4-1: **Terminología de diagramas de clase**

UML	Clase	Asociación	Generalización	Agregación
Booch	Clase	Usa	Hereda	Contiene
Coad	Clase y objeto	Conexión de instancias	Espec-gen	Parte-todo
Jacobson	Objeto	Asociación por reconocimiento	Hereda	Consiste en
Odell	Tipo de objeto	Relación	Subtipo	Composición
Rumbaugh	Clase	Asociación	Generalización	Agregación
Shlaer/ Mellor	Objeto	Relación	Subtipo	N/A

Asociaciones

La figura 4-1 muestra un modelo de clases simple que no sorprenderá a nadie que haya trabajado con un proceso de pedidos. Describiré sus partes y hablaré sobre las maneras de interpretadas, desde las distintas perspectivas.

Comenzaré con las asociaciones. **Las asociaciones** representan relaciones entre instancias de clases (una persona trabaja para una empresa; una empresa tiene cierta cantidad de oficinas).

Desde la **perspectiva conceptual,** las asociaciones representan relaciones conceptuales entre clases. El diagrama indica que un Pedido debe venir de un solo Cliente y que un Cliente puede hacer varios Pedidos en un periodo de tiempo. Cada uno de estos Pedidos tiene varias instancias de Línea de pedido, cada una de las cuales se refiere a un solo Producto.

Cada asociación tiene dos **papeles;** cada papel es una dirección en la asociación. De este modo, la asociación entre Cliente y Pedido contiene dos papeles: uno del Cliente al Pedido; el segundo del Pedido al Cliente.

Se puede nombrar explícitamente un papel mediante una etiqueta. En este caso, el papel en sentido Pedido a Línea de orden se llama Artículos de línea. Si no hay etiqueta, el papel se puede nombrar de acuerdo con la etiqueta de la clase; de esta forma, el papel de Pedido a Cliente se llamará Cliente (en este libro, me refiero a la clase de donde parte el papel como **origen** y a la clase a dónde va el papel como **destino.** Esto significa que hay un papel de Cliente cuyo origen es Pedido y cuyo destino es Cliente).

Un papel tiene también una **multiplicidad,** la cual es una indicación de la cantidad de objetos que participarán en la relación dada. En la Figura 4-1, la [*] entre Cliente y Pedido indica que el primero puede tener muchas ordenes asociadas a él; el [1] indica que un Pedido viene de un solo Cliente.

En general, la multiplicidad indica los límites inferior y superior de los objetos participantes. El [*] representa de hecho el intervalo [0.."infinito"]: El Cliente no necesita haber colocado un Pedido, y no hay un tope superior (por lo menos en teoría) para la cantidad de pedidos que puede colocar. El [1] equivale a [1..1]: cada Pedido debe haber sido solicitado por un solo Cliente.

En la práctica, las multiplicidades más comunes son [1], [*], y [0..1] (se puede no tener ninguno o bien tener uno). Para una multiplicidad más general, se puede tener un solo número (por ejemplo, [11] jugadores en un equipo de fútbol), un intervalo (por ejemplo, [2..4] para los participantes de un juego de canasta) o combinaciones discretas de números e intervalos (por ejemplo, [2, 4] para las puertas de un automóvil).

La Figura 4-2 muestra las notaciones de cardinalidad del UML y de los principales métodos pre-UML.

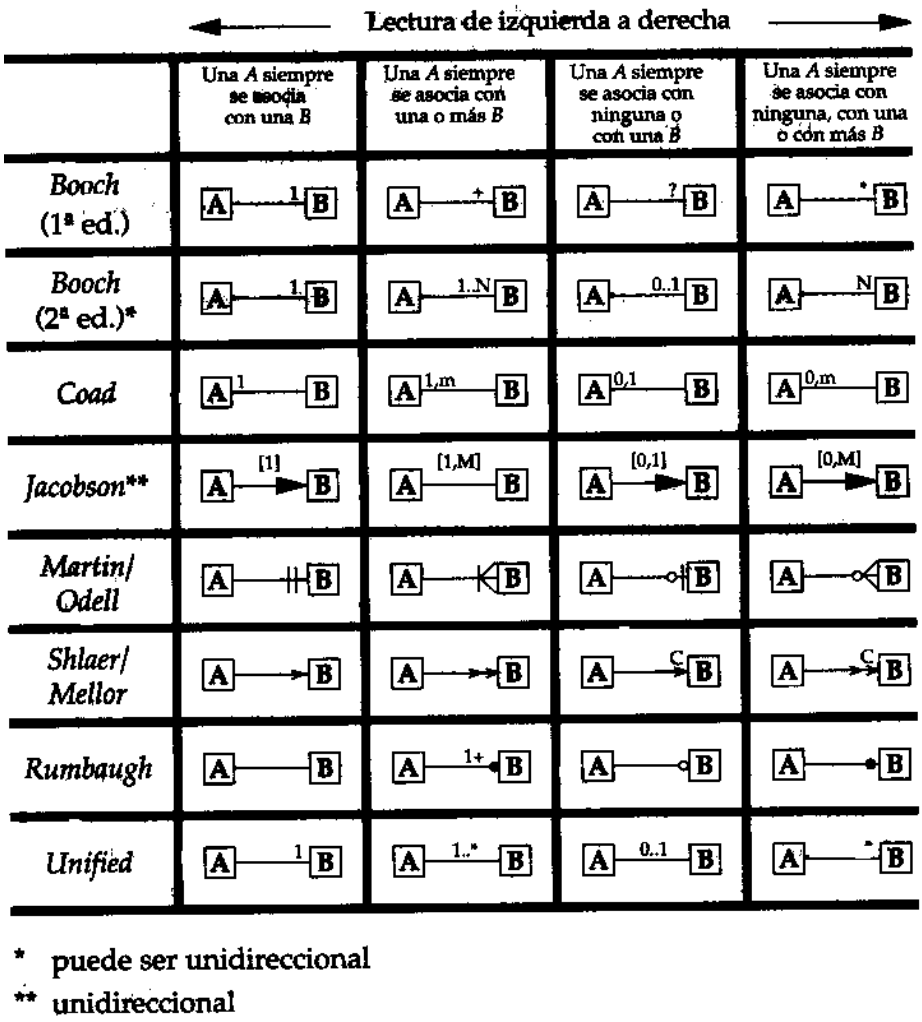

Figura 4-2: *Notaciones de cardinalidad*

Dentro de la **perspectiva de la especificación,** las asociaciones representan responsabilidades.

La Figura 4-1 significa que hay uno o más métodos asociados con Cliente que me proporcionarán los pedidos que ha colocado un Cliente dado. Del mismo modo, existen métodos en Pedido que me permitirán saber qué Cliente colocó tal Pedido y cuáles son los Artículos de línea que contiene un Pedido.

Si existen convenciones estándar para nombrar **métodos de consulta,** probablemente sus nombres se puedan deducir del diagrama. Por ejemplo, puedo tener una convención que diga que las **relaciones de un solo valor** se implementan con un método que devuelve el objeto relacionado y que las relaciones de varios valores se implementan mediante una enumeración (iterador) en un **conjunto de objetos relacionados.**

Al trabajar con una convención de nombres como ésta en Java, por ejemplo, puedo deducir la siguiente interfaz para una clase de Pedido:

```
class Pedido {
  public Cliente cliente();
  // Enumeración de líneas de Pedido
  public Enumeration lineaspedido() ;
```

Las convenciones de programación, por supuesto, variarán de acuerdo con el lugar y no indicarán todos los métodos, pero le serán a usted de gran utilidad para encontrar el camino.

La Figura 4-1 implica también cierta responsabilidad al poner al día la relación. Debe haber algún modo de relacionar el Pedido con el Cliente. De nuevo, no se muestran los detalles; podría ser que se especifique el Cliente en el constructor del Pedido. O, tal vez, exista un método agregarPedido asociado al Cliente. Esto se puede hacer más explícito añadiendo operaciones al cuadro de clase (tal y como veremos más adelante).

Sin embargo, estas responsabilidades no implican una estructura de datos. A partir de un diagrama a nivel de especificación, no puedo hacer suposición alguna sobre la estructura de datos de las clases. No puedo, ni debo poder decir si la clase Pedido contiene un apuntador a Cliente, o si la clase Pedido cumple su responsabilidad ejecutando algún código de selección que pregunte a cada Cliente si se refiere a un Pedido dado. El diagrama sólo indica la interfaz, y nada más.

Si éste fuera un **modelo de implementación,** ahora daríamos a entender con ello que hay apuntadores en ambos sentidos entre las clases relacionadas. El diagrama diría entonces que Pedido tiene un campo que es un conjunto de apuntadores hacia Línea de pedido y también un apuntador a Cliente. En Java, podríamos deducir algo como lo que exponemos a continuación:

```
class Pedido {
  private Cliente _cliente;
  private Vector _lineaspedido;
class Cliente {
  private Vector _pedidos;
```

En este caso, no podemos inferir nada sobre la interfaz a partir de las asociaciones. Esta información nos la darían las operaciones sobre la clase.

Ahora véase la Figura 4-3. Es básicamente la misma que la Figura 4-1, a excepción de que he añadido un par de flechas a las líneas de asociación. Estas líneas indican**navegabilidad.**

En un modelo de especificación, esto indicaría que un Pedido tiene la responsabilidad de decir a qué Cliente corresponde, pero un Cliente no tiene la capacidad correspondiente para decir cuáles son los pedidos que tiene. En lugar de responsabilidades simétricas, aquí tenemos responsabilidades de un solo lado de la línea. En un diagrama de implementación se indicaría qué Pedido contiene un apuntador a Cliente pero Cliente no apuntaría a Pedido.

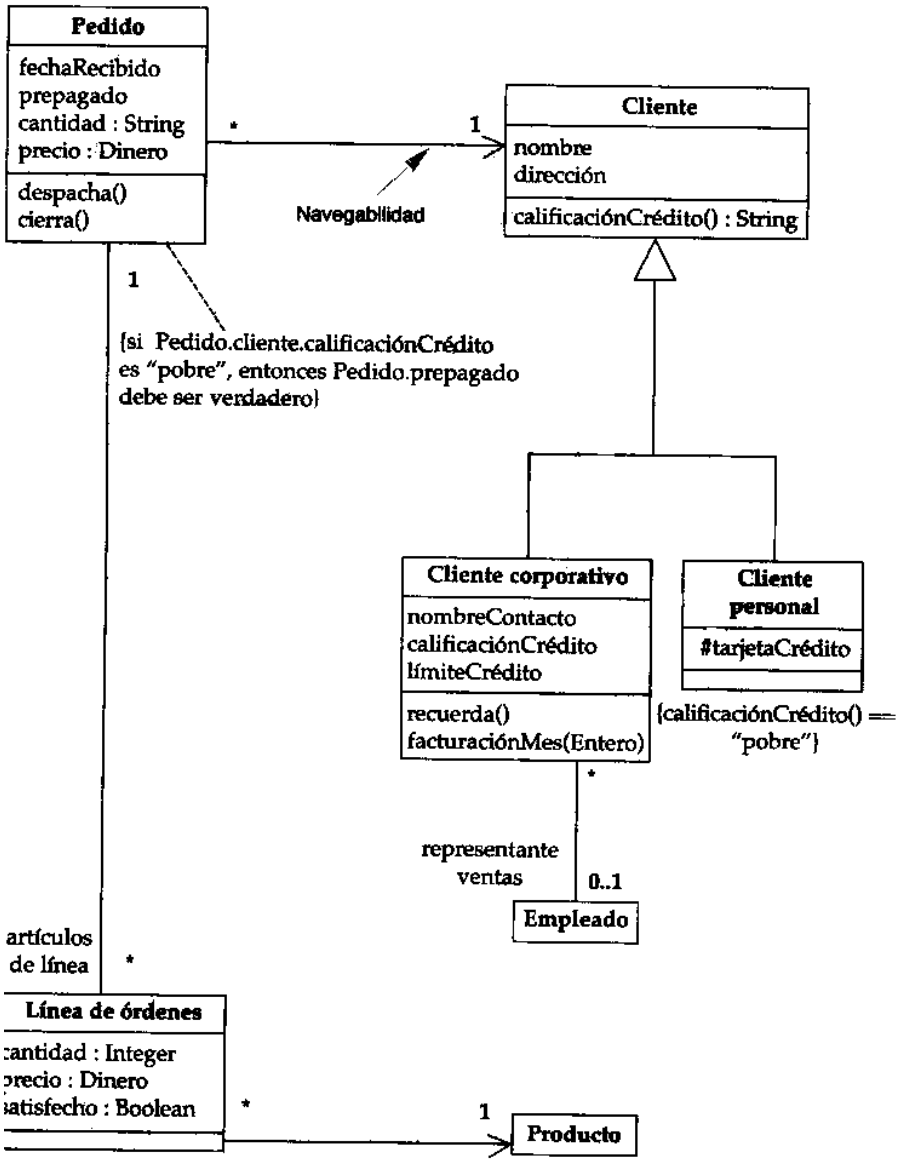

Figura 4-3: *Diagrama de clase con navegabilidades*

Como se podrá apreciar, la navegabilidad es una parte importante de los diagramas de implementación y especificación. Sin embargo, no pienso que la navegabilidad tenga utilidad en los diagramas conceptuales.

Frecuentemente verá diagramas conceptuales que primero no tendrán navegabilidades. Posteriormente, se agregarán las navegabilidades como parte del cambio a perspectivas de especificación y de implementación. Nótese también, que las navegabilidades posiblemente serán diferentes entre la especificación y la implementación.

Si existe una navegabilidad en una sola dirección, a la asociación se le llama **asociación unidireccional.** Una **asociación bidireccional** contiene navegabilidades en ambas direcciones. El UML dice que las asociaciones sin flechas significan que la navegabilidad es desconocida o que la asociación es bidireccional. El proyecto debe definirse en el sentido de uno u otro de los significados.

Las asociaciones bidireccionales incluyen una restricción adicional, que consiste en que ambos papeles son inversos entre ellos. Esto es igual al concepto de funciones inversas en las matemáticas. En el contexto de

la Figura 4-3, esto significa que cada Artículo de línea asociado con un Pedido se debe asociar con el Pedido original. De modo similar, si se toma una Línea de pedido y se busca en los artículos de línea el Pedido asociado, se deberá ver la Línea de pedido original en el conjunto. Esta cualidad se mantiene verdadera en las tres perspectivas.

Hay varias maneras de nombrar las asociaciones. A los modeladores de datos tradicionales les gusta denominar a una asociación con un verbo, de modo que se pueda emplear la relación en una oración. La mayoría de los modeladores de objetos prefieren valerse de sustantivos para denominar uno u otro papel, pues esta forma corresponde mejor a las responsabilidades y operaciones.

Algunos les dan nombres a todas las asociaciones. Yo prefiero nombrar las asociaciones sólo cuando mejora la comprensión. He visto demasiadas asociaciones con nombres como "tiene" o "se relaciona con". Si no existe un nombre para el papel, considero que su nombre es el de la clase señalada, como indiqué antes.

Atributos

Los **atributos** son muy parecidos a las asociaciones.

Desde un nivel conceptual, el atributo de nombre de un Cliente indica que los Clientes tienen nombres. Desde el nivel de especificación, este atributo indica que un objeto Cliente puede decir su nombre y tiene algún modo de establecer un nombre. En el nivel de implementación, un Cliente tiene un **campo** (también llamado variable de instancia o miembro de datos) para su nombre.

Dependiendo del detalle del diagrama, la notación de un atributo puede mostrar el nombre, el tipo y el valor predeterminado de un atributo (la sintaxis del UML *es visibilidad nombre: tipo = valor por omisión*, donde la *visibilidad* es igual que la de las Operacioneslas cuales se describirán en la sección siguiente).

Por lo tanto, ¿cuál es la diferencia entre un atributo y una asociación?

Desde la perspectiva conceptual, no hay diferencia; un atributo sólo lleva consigo otro tipo de notación, la cual puede servir si se estima conveniente. Los atributos siempre son de valor único. Por lo general, los diagramas no indican si los atributos son opcionales u obligatorios (aunque, hablando estrictamente, deberían indicado).

La diferencia se da en los niveles de especificación y de implementación. Los atributos implican navegabilidad sólo del tipo al atributo. Es más, queda implícito que el tipo contiene únicamente su propia copia del objeto de atributo, lo que implica que cualquier tipo que se emplee como atributo tendrá un valor más que una semántica de referencia.

Hablaré sobre el valor y los tipos referenciales más adelante. Por el momento, lo mejor es considerar que los atributos son clases simples y pequeñas, tales como cadenas, fechas, objetos de moneda y valores no objeto, como int y real.

Operaciones

Las **operaciones** son los procesos que una clase sabe llevar a cabo. Evidentemente, corresponden a los métodos sobre una clase. En el nivel de especificación, las operaciones corresponden a los métodos públicos sobre un tipo. En general, no se muestran aquellas operaciones que simplemente manipulan atributos, ya que por lo común, se pueden inferir. Sin embargo, tal vez sea necesario indicar si un **atributo dado es de sólo lectura** o está **inmutable congelado** (esto es, que su valor nunca cambia). En el modelo de implementación, se podrían mostrar también las operaciones privadas y protegidas.

La sintaxis del UML completa para las operaciones es

visibilidad nombre (lista-de-parámetros) : expresión-tipo-de-dato-a-regresar {cadena-de-propiedades} .

donde

- *visibilidad* es + (public), # (protected), o- (private)
- *nombre* es una cadena de caracteres (string)
- *lista-de-parámetros* contiene argumentos (opcionales) cuya sintaxis, es la misma que la de los atributos
- *expresión-tipo-de-dato-a-regresar* es una especificación opcional dependiente del lenguaje
- cadena-de-propiedades indica valores de propiedad que se aplican a la operación dada

Un ejemplo de operación sería: + *úItimaCantidadDe (valorTipoFenómeno) : Cantidad*

Dentro de los modelos conceptuales, las operaciones no deben tratar de especificar la interfaz de una clase. En lugar de ello, deberán indicar las principales responsabilidades de dicha clase, usando tal vez un par de palabras que sinteticen una **responsabilidad de CRC.**

Con frecuencia encuentro útil hacer la distinción entre aquellas operaciones que cambian el estado de una clase y aquellas que no lo hacen. Una **consulta** es una operación que obtiene un valor de una clase sin que cambie el estado observable de tal clase. El**estado observable de un objeto** es el estado que se puede determinar a partir de sus consultas asociadas.

Considérese un objeto de Cuenta que calcula su balance a partir de una lista de entradas. Para mejorar el desempeño, Cuenta puede poner en un campo caché el resultado del cálculo del balance, para consultas futuras. Por tanto, si el caché está vacío, la primera vez que se llama a la operación "balance", pondrá el resultado en el campo caché. La operación "balance" cambia así el estado real del objeto Cuenta, pero no su estado observable, pues todas las consultas devuelven el mismo valor, esté o no lleno el campo caché. Las operaciones que sí cambian el estado observable de un objeto se llaman **modificadores.**

Considero útil tener perfectamente clara la diferencia entre consultas y modificadores. Las consultas se pueden ejecutar en cualquier orden, pero la secuencia de los modificadores es más importante. Yo tengo como política evitar que los modificadores devuelvan valores, con el fin de mantenerlos separados.

Otros términos que algunas veces se presentan son: métodos de obtención y métodos de establecimiento. El **método de obtención** (*getting*) es un método que devuelve el valor de un campo (sin hacer nada más). El **método de establecimiento** (*setting*) pone un valor en un campo (y nada más). Desde afuera, el cliente no debe poder saber si una consulta es un método de obtención ni si un modificador es un método de establecimiento. El conocimiento sobre los métodos de obtención y establecimiento está contenido completamente dentro de la clase.

Otra distinción es la que se da entre operación y método. Una **operación** es algo que se invoca sobre un objeto (la llamada de procedimiento), mientras que un **método** es el cuerpo del procedimiento. Los dos son diferentes cuando se tiene polimorfismo. Si se tiene un supertipo con tres subtipos, cada uno de los cuales suplanta la operación "*foo*" del supertipo, entonces lo que hay es una operación y cuatro métodos que la implementan.

Es muy común que operación y método se empleen indistintamente, pero hay veces en que es necesario precisar la diferencia. En algunas ocasiones, la gente distingue entre una y otro mediante los términos llamada a un método, o declaración de método (en lugar de operación), y cuerpo del método.

Los lenguajes tienen sus propias convenciones para los nombres. En C ++, las operaciones se llaman funciones miembro; En Smalltalk, operaciones de métodos. C ++ también emplea el término miembros de una clase para denominar las operaciones y métodos de una clase.

Generalización

Un ejemplo característico de **generalización** comprende al personal y las Corporaciones clientes de una empresa. Unos y otros tienen diferencias, pero también muchas similitudes. Las similitudes se pueden colocar en una clase general Cliente (el**supertipo**) con los subtipos siguientes: Cliente personal y Cliente corporativo.

Este fenómeno también está sujeto a diversas interpretaciones, según el nivel de modelado. Por ejemplo, conceptualmente, podemos decir que el Cliente corporativo es un **subtipo** de Cliente, si todas las instancias de Cliente corporativo, también son, por definición, instancias de Cliente. Entonces, un Cliente corporativo es un tipo especial de Cliente. El concepto clave es que todo lo que digamos de un Cliente (asociaciones, atributos, operaciones) también vale para un Cliente corporativo.

En un modelo de especificación, la generalización significa que la **interfaz** del subtipo debe incluir todos los elementos de la interfaz del supertipo. Se dice, entonces, que la interfaz del subtipo se conforma con la interfaz del supertipo.

Otra manera de considerar esto involucra el principio de la **sustituibilidad.** Yo debo poder sustituir a un Cliente corporativo con cualquier código que requiera un Cliente y todo debe operar sin problemas. En esencia, esto significa que, si escribo un código suponiendo que tengo un Cliente, puedo entonces usar con toda libertad cualquier subtipo de Cliente. El Cliente corporativo puede responder a ciertos comandos de manera diferente a otro Cliente (por el principio del polimorfismo), pero el invocador no debe preocuparse por la diferencia.

La generalización desde la perspectiva de implementación se asocia con la herencia en los lenguajes de programación. La subclase hereda todos los métodos y campos de la superclase y puede suplantarlos.

El punto clave aquí es la diferencia entre generalización desde la perspectiva de la especificación (subtipificación o herencia de interfaces) y desde la perspectiva de la implementación (subclasificación o herencia de implementaciones). La **subclasificación**(formación de subclases) es una manera de implementar la subtipificación. La**subtipificación** (formación de subtipos) también se puede implementar mediante delegación.

En el caso de estas dos formas de generalización, se deberá asegurar siempre que también tenga validez la **generalización conceptual.** He encontrado que, si no se hace lo anterior, se puede uno ver en problemas, debido a que la generalización no es estable cuando hay necesidad de efectuar cambios posteriores.

En algunas ocasiones aparecen casos en los que un subtipo tiene la misma interfaz que el supertipo, pero el subtipo implementa sus operaciones de modo diferente. Si éste es el caso, se podría decidir no mostrar el subtipo en el diagrama de perspectiva de especificación. Así lo hago con frecuencia si para los usuarios de la clase es interesante la existencia de los tipos, pero no lo hago cuando los subtipos varían únicamente debido a razones internas de implementación.

Reglas de restricción

Una buena parte de lo que se hace cuando se dibuja un diagrama de clase es indicar las condiciones limitantes o restricciones.

La Figura 4-3 indica que un Pedido únicamente puede ser colocado por un solo Cliente. El diagrama implica también que Artículos de línea se considera por separado: digamos 40 adminículos de color marrón, 40 adminículos azules y 40 adminículos rojos, y no 40 artefactos rojos, azules y marrones. El diagrama dice, además, que los Clientes corporativos tienen límites de crédito, pero que los Clientes personales no los tienen.

Los artificios básicos de la asociación, el atributo y la generalización, colaboran de manera significativa con la especificación de condiciones importantes, pero no pueden indicarlas todas. Estas restricciones aún necesitan ser captadas; el diagrama de clase es un lugar adecuado para hacerlo.

El UML no define una sintaxis estricta para describir las restricciones, aparte de ponerlas entre llaves ({ }). Yo prefiero escribir en lenguaje informal, para simplificar su lectura. Puede usarse también un enunciado más formal, como un cálculo de predicados o alguna derivada. Otra alternativa es escribir un fragmento de código de programación.

Idealmente, las reglas se deberían implementar como afirmaciones en el lenguaje de programación. Éstas se corresponden con el concepto de invariantes del **Diseño por contrato** En lo personal, soy partidario de crear un método *revisalnvariante* e invocarlo desde algún código de depuración que ayude a comprobar los invariantes.

Cuándo emplear los diagramas de clases

Los diagramas de clase son la columna vertebral de casi todos los métodos OO, por lo cual usted descubrirá que se emplean todo el tiempo. Este capítulo ha abarcado los conceptos básicos; el capítulo siguiente, analizará muchos de los conceptos más avanzados.

El problema con los diagramas de clase es que son tan ricos que pueden resultar abrumadores. A continuación, damos algunos consejos breves

- No trate de usar todas las anotaciones a su disposición. Comience con el material sencillo de este capítulo: clases, asociaciones, atributos y generalización. Introduzca otras anotaciones más avanzadas sólo cuando las necesite.
- Ajuste la perspectiva desde la cual se dibujan los modelos a la etapa del proyecto.
    - Si se está en la etapa del análisis, dibuje modelos conceptuales.
    - Cuando se trabaje con software, céntrese en los modelos de especificación.
    - Dibuje modelos de implementación sólo cuando se esté ilustrando una técnica de implementación en particular.
- No dibuje modelos para todo; por el contrario, céntrese en las áreas clave. Es mejor usar y mantener al día unos cuantos diseños que tener muchos modelos olvidados y obsoletos.

El principal peligro con los diagramas de clase es que se puede quedar empantanado muy pronto en los detalles de la implementación. Para contrarrestar esto, céntrese en las perspectivas conceptual y de especificación. Si cae en estos problemas, encontrará muy útiles las tarjetas de CRC

Para mayor información

Mi consejo sobre otras fuentes depende de si usted prefiere una perspectiva de implementación o conceptual. Para la perspectiva de implementación, es recomendable Booch (1994); Para una perspectiva conceptual son recomendables los libros de "fundamentos" de Martin y Adell (1994). Una vez que haya leído su primera selección, lea la otra, pues ambas perspectivas son importantes. Después, cualquier buen libro sobre OO le dará algunos puntos de vista interesantes. Tengo una particular predilección por el de Cook y Daniels (1994), debido a su manejo de las perspectivas y a la formalidad que introducen los autores.

DIAGRAMAS DE CLASE - CONCEPTOS AVANZADOS

Resumen
En este capítulo se tratan los conceptos avanzados de los diagramas de clases.
Los conceptos básicos que se emplean en más del 90% de los casos, están descritos en el capítulo anterior.
En este capítulo se presentan las características avanzadas de la programación con objetos

Índice de Contenidos

Introducción

Los conceptos descritos en el capítulo anterior corresponden a las notaciones fundamentales de los diagramas de clase. Éstos son los que usted necesita comprender y dominar primero, pues significarán el 90% del esfuerzo que se invertirá en la construcción de los diagramas de clase.

La técnica de diagramas de clase, sin embargo, ha generado decenas de nuevas notaciones para otros conceptos. En realidad, éstos no se emplean con frecuencia, pero hay ocasiones en las que resultan muy útiles.

A continuación las explicaré una por una y señalaré algunas de las consideraciones sobre su uso. No obstante, recuerde que son opcionales y que hay mucha gente que ha obtenido grandes ventajas de los diagramas de clase sin haber utilizado estas características adicionales.

Tal vez encuentre el presente capítulo un poco denso. Sin embargo, tengo una buena noticia al respecto, y es que este capítulo puede usted saltado completo, sin ningún problema, en la primera lectura que haga del libro, y volver a él más tarde.

Los estereotipos

La idea de los **estereotipos** fue acuñada por Rebecca Wirfs-Brock. El concepto ha sido adoptado con entusiasmo por los inventores del UML, aunque de un modo que en realidad no significa lo mismo. Sin embargo, ambas ideas son valiosas.

La idea original de un estereotipo se refería a una clasificación de alto nivel de un objeto que diera alguna indicación del tipo de objeto que era. Un ejemplo es la diferencia entre un "controlador" y un "coordinador".

Con frecuencia se encuentran diseños OO en los que una clase parece hacer todo el trabajo, muchas veces por medio de un gran método "hazlo" (*doIt*), mientras que las demás clases no hacen otra cosa más que encapsular datos. Éste es un diseño pobre, pues significa que el controlador es muy complejo y difícil de manejar.

Para mejorar esta situación, se traslada el comportamiento del controlador a los objetos de datos relativamente tontos, de modo que éstos se vuelven más inteligentes y adquieren responsabilidades de más alto nivel. Así, el controlador se convierte en un coordinador. El coordinador es el encargado de disparar tareas en una secuencia particular, pero otros objetos son los que saben cómo desempeñadas.

La esencia del estereotipo consiste en que sugiere ciertas responsabilidades generales de una clase. El UML ha adoptado este concepto y lo ha convertido en un mecanismo general de extensión del lenguaje mismo.

En su trabajo original (1994), Jacobson clasifica todas las clases de un sistema en tres categorías: objetos de interfaz, objetos de control y objetos de entidad (sus objetos de control, cuando están bien diseñados, son parecidos a los coordinadores de Wirfs-Brock). Jacobson sugirió reglas para la comunicación entre estos tipos de clases y les dio a cada una de ellas un icono diferente. Esta distinción no es una parte medular del UML. Al contrario, este tipo de clases constituye en realidad estereotipos de clases; de hecho, son estereotipos, en el sentido que Wirfs-Brock le da al término.

Los estereotipos generalmente se indican en el texto entre comillas francesas («objeto de control»), pero también se pueden mostrar definiendo un icono para el estereotipo. De lo que se trata es de que, si no se está empleando el enfoque de Jacobson, se pueden olvidar los estereotipos. Pero si quiere trabajar con dicho enfoque, se pueden definir los estereotipos y las reglas para su uso.

Muchas extensiones del núcleo de UML se pueden describir como una colección de estereotipos. En los diagramas de clase, pueden ser estereotipos de clases, asociaciones o generalizaciones. Puede pensarse en los estereotipos como subtipos de los tipos Clase, Asociación y Generalización en el metamodelo.

He observado que la gente que trabaja con el UML tiende a confundir las restricciones, es decir, las condiciones limitantes con los estereotipos. Si se marca una **clase como abstracta,** ¿se trata de una restricción o de un estereotipo? Los documentos oficiales actuales llaman a esto restricción, pero hay que estar conscientes de que se hace un uso confuso de una y otro. Esto no es de sorprender, ya que los subtipos son con frecuencia más limitados que los supertipos.

Clasificación múltiple y dinámica

La **clasificación** se refiere a la relación entre un objeto y su tipo.

La mayor parte de los métodos hacen ciertas suposiciones sobre este tipo de relación (premisas que están presentes también en los principales lenguajes de programación OO). Estas premisas fueron cuestionadas por Jim Odell, quien las consideró demasiado restrictivas para el modelado conceptual. Las suposiciones son para

una clasificación simple y estática de los objetos; Odell sugiere emplear una clasificación múltiple y dinámica de los objetos en los modelos conceptuales.

En la **clasificación simple,** un objeto pertenece a un solo tipo, que puede haber heredado de supertipos. En la **clasificación múltiple,** un objeto puede ser descrito por varios tipos que no están conectados necesariamente por medio de herencia.

Obsérvese que la clasificación múltiple es diferente de la **herencia múltiple.** La herencia múltiple plantea que un tipo puede tener muchos supertipos, pero que se debe definir un solo tipo por cada objeto. La clasificación múltiple permite varios tipos para un objeto sin definir un tipo específico para tal fin.

Como ejemplo, considere una persona subtipificada como hombre o mujer, doctor o enfermera, paciente o no (véase la Figura 5-1). La clasificación múltiple permite asignar cualquiera de estos tipos a un objeto en cualquier combinación posible, sin necesidad de que se definan tipos para todas las combinaciones legales.

Si se emplea la clasificación múltiple, habrá de asegurarse que haya quedado claro cuáles son las combinaciones legales. Esto se hace etiquetando una línea de generalización con un discriminador, el cual es una indicación de la base de la subtipificación. Varios subtipos pueden compartir el mismo discriminador. Todos los subtipos con el mismo discriminador están desunidos, es decir, cualquier instancia del supertipo puede ser una instancia de sólo uno de los subtipos, dentro del discriminador. Una buena convención es que todas las subclases que emplean un discriminador desemboquen en un triángulo, como se muestra en la Figura 5-1. Se pueden tener, como alternativa, varias flechas con la misma etiqueta.

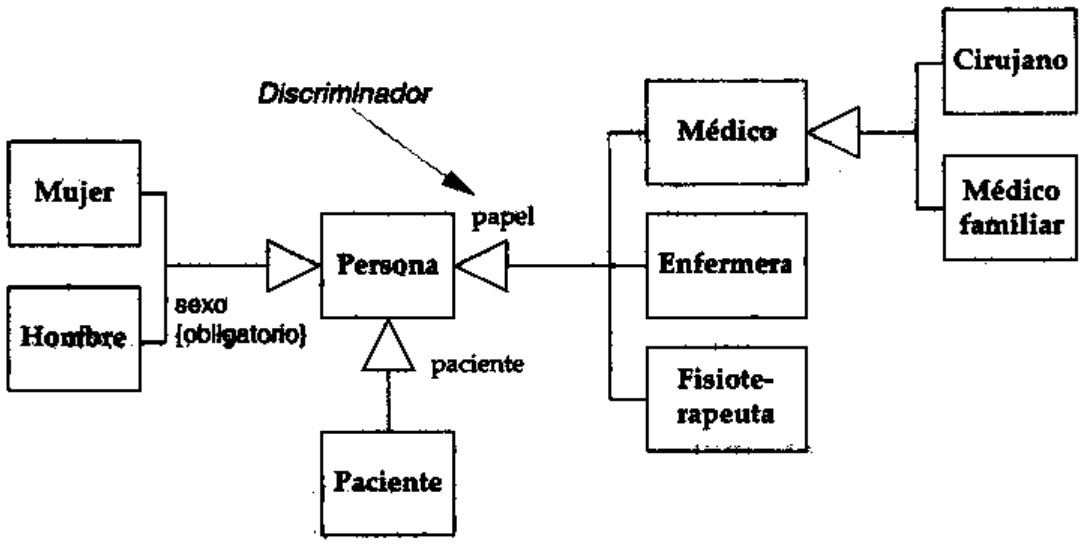

Figura 5-1: *Clasificación múltiple*

Una restricción útil (pero no estándar en el UML) consiste en marcar al discriminador como {obligatorio}. Esto significa que cualquier instancia de la superclase debe ser una instancia de una de las subclases del grupo (por tanto, la superclase es abstracta).

Para ilustrar, obsérvense las siguientes combinaciones legales de subtipos en el diagrama: (Mujer, Paciente, Enfermera); (Hombre, Fisioterapeuta); (Mujer, Paciente); y (Mujer, Médico, Cirujano). Obsérvense también que las combinaciones como (Paciente, Médico) y (Hombre, Médico, Enfermera) son ilegales: la primera, porque no incluye un tipo del discriminador Sexo {obligatorio}; la última, porque contiene dos tipos del discriminador "Papel". La clasificación simple, por definición, corresponde a un solo discriminador no etiquetado.

Otra pregunta que cabe plantearse es si un objeto puede cambiar de tipo. Un buen ejemplo para contestada es una cuenta bancaria. Cuando se sobregira la cuenta, cambia sustancialmente su conducta; específicamente se suplantan varias operaciones (incluyendo la de "retiro" y la de "cierre").

La **clasificación dinámica** permite a los objetos cambiar de tipo dentro de la estructura de subtipificación; la **clasificación estática,** no. En la clasificación estática se hace la distinción entre tipos y estados; en la clasificación dinámica se combinan estos dos conceptos.

¿Debería usarse la clasificación múltiple y dinámica? Considero que es útil para el modelado conceptual. Se puede hacer con modelado por especificaciones, pero conviene sentirse cómodo con las técnicas necesarias para realizado. El truco consiste en implementado de manera que parezca lo mismo que subclasificar desde la interfaz, de tal suerte que el usuario de una clase no sepa qué implementación se está usando. Sin embargo, como en la mayoría de estas cosas, la elección depende de las circunstancias, así que se deberá aplicar lo que se juzgue mejor. La transformación de una interfaz múltiple y dinámica a una implementación estática única bien puede acarrear más problemas de los que resolvería.

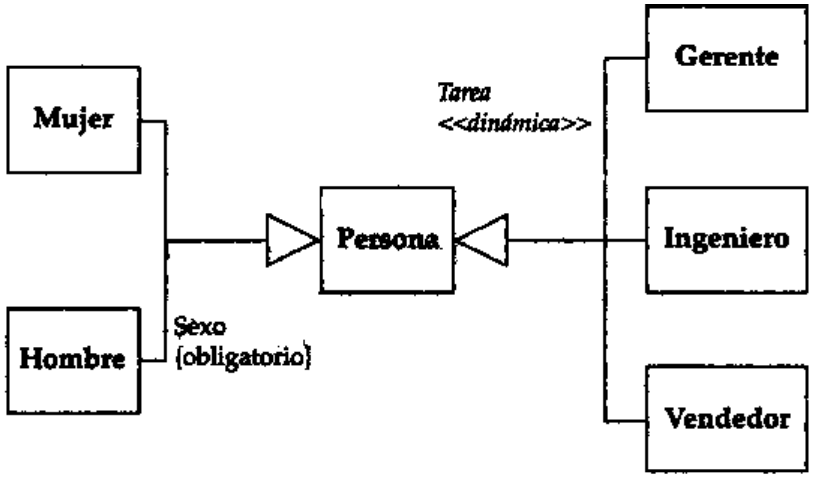

Figura 5-2: *Clasificación dinámica*

La figura 5-2 muestra un ejemplo del empleo de la clasificación dinámica para el trabajo de una persona, que, por supuesto, puede cambiar. Esto puede ser lo apropiado, pero los subtipos necesitarían un comportamiento adicional, en lugar de ser sólo etiquetas. En estos casos, muchas veces vale la pena crear una clase separada para el trabajo y vincular a la persona con ella mediante una asociación.

Agregación y composición

Una de las cosas más aterradoras del modelado es la **agregación.** Ésta es fácil de explicar con pocas palabras: la agregación es la relación de componentes. Es como decir que un automóvil tiene como componentes motor y ruedas. Esto suena muy bien, pero la parte difícil es determinar cuál es la diferencia entre agregación y asociación.

Peter Coad ejemplificó la agregación como la relación entre una organización y sus empleados; Jim Rumbaugh afirmó que una compañía no es la agregación de sus empleados. Cuándo los gurúes no se ponen de acuerdo, ¿qué debemos hacer? El problema es que los metodólogos no tienen una definición aceptada por todos de la diferencia entre asociación y agregación.

De hecho, muy pocos se sirven de cualquier definición formal. La cuestión práctica que nos interesa aquí es que todos manejan un concepto levemente diferente, de modo que hay que tener precaución con este concepto, y,

generalmente, prefiero evitado, a menos que el equipo del proyecto se ponga de acuerdo en un significado riguroso y útil.

El UML ofrece, además de la agregación simple, una variedad más poderosa, la cual se denomina composición. Con la **composición,** el objeto parte puede pertenecer a un todo único; es más, se espera, por lo general, que las partes vivan y mueran con el todo. Cualquier borrado del todo se extiende en cascada a las partes.

El borrado en cascada se considera con frecuencia como una parte definitoria de la agregación, pero está implícita en cualquier papel con una multiplicidad de [1..1]; si realmente se quiere eliminar a un Cliente, por ejemplo, habrá que aplicar este borrado en cascada a Pedido (y, por tanto, a Línea de pedido).

La Figura 5-3 muestra ejemplos de estos artificios. Las composiciones a Punto indican que cualquier instancia de Punto puede estar en un Polígono o en un Círculo, pero no en ambos. Sin embargo, una instancia de Estilo puede ser compartida por muchos Polígonos y Círculos. Es más, esto implica que el borrado de un Polígono provocaría el borrado de sus Puntos asociados, pero no del Estilo asociado.

Esta restricción (un Punto puede solamente aparecer en un solo Polígono o Círculo a la vez) no se puede expresar mediante las multiplicidades, únicamente. También implica que el punto es un objeto de valor(Vease mas adelante en este mismo capitulo). Se puede añadir una multiplicidad a un lado compuesto de la asociación, pero yo no me tomo la molestia. El rombo negro dice todo lo que hay que decir.

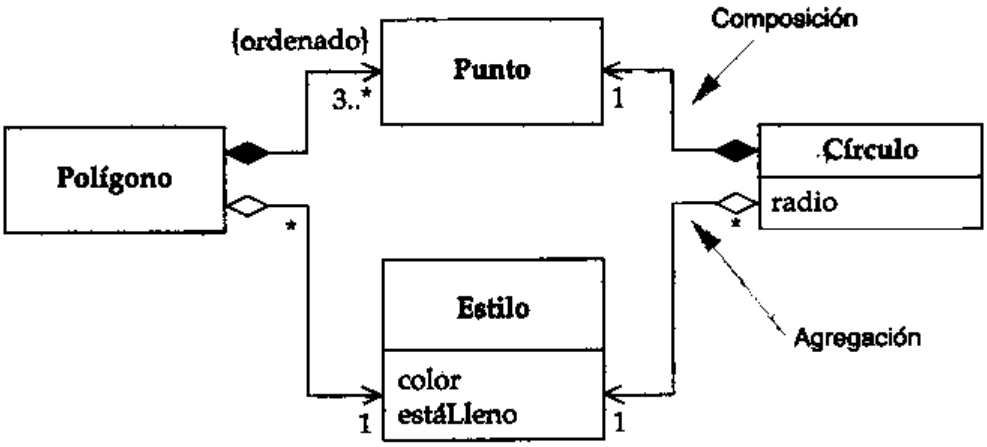

Figura 5-3: *Agregación y composición*

La figura 5-4 muestra otra notación para la composición. En este caso, se pone el componente dentro del todo. El nombre de la clase del componente no está en negritas y se escribe en la forma nombrepapel: Clase nombre. Además, se pone la multiplicidad en la esquina superior derecha.

Las diferentes notaciones de composición operan en situaciones diferentes. Hay un par más, aunque la variedad de notaciones de composición ofrecida por el UML es más bien abrumadora. Obsérvese que estas variaciones sólo pueden servir para la composición, no para la agregación.

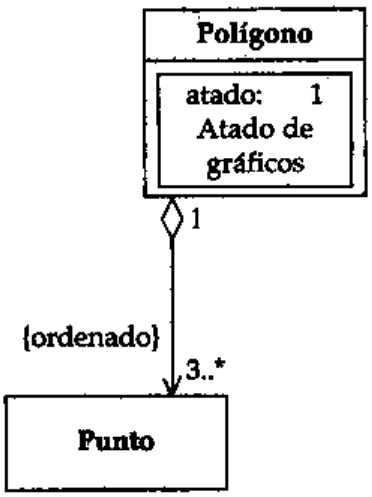

Figura 5-4: *Notación alterna para la composición*

Asociaciones y Atributos Derivados

Las **asociaciones derivadas** y los **atributos derivados** son aquellos que se pueden calcular a partir de otras asociaciones y atributos, respectivamente, de un diagrama de clase. Por ejemplo, un atributo de edad de una Persona se puede derivar si se conoce su fecha de nacimiento.

Cada perspectiva contribuye con su propia interpretación de las características derivadas de los diagramas de clase. La más crítica de éstas se relaciona con la perspectiva de especificación. Desde este ángulo, es importante comprender que las características derivadas indican una restricción entre valores y no una declaración de lo que se calcula y lo que se almacena.

La figura 5-5 muestra una estructura jerárquica de cuentas dibujada desde una perspectiva de especificación. El modelo utiliza el patrón Compuesto.

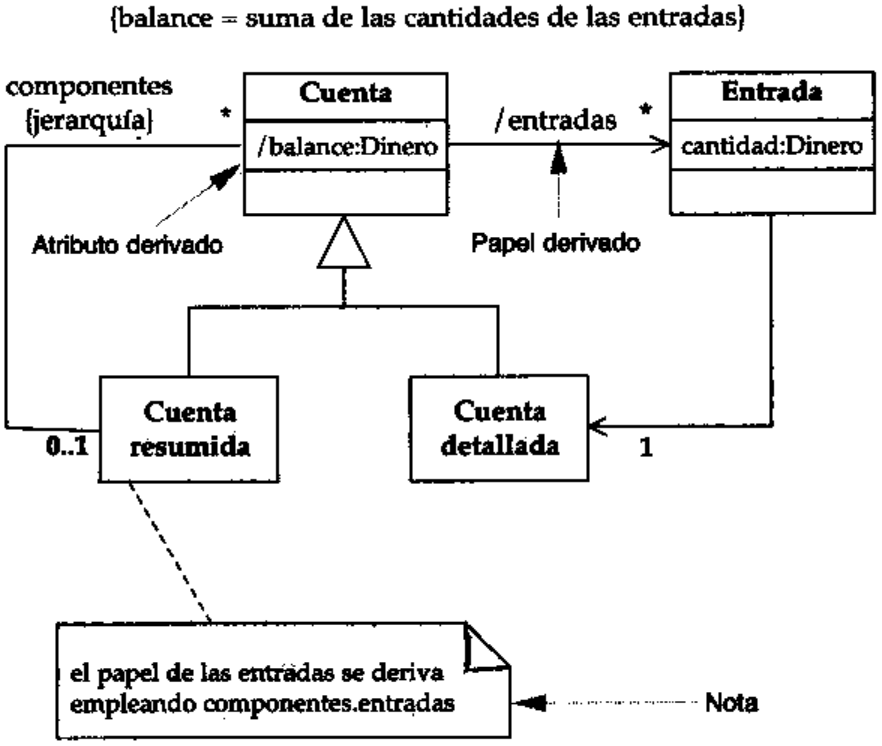

Figura 5-5: *Asociaciones y atributos derivados*

Obsérvese lo siguiente:

- Los objetos de entrada están conectados a Cuentas detalladas.
- El balance de una Cuenta se calcula como la suma de las cantidades de Entrada.
- Las entradas de una Cuenta resumida son las entradas de sus componentes, determinados de manera recurrente.

Debido a que la Figura 5-5 ilustra un modelo de especificación, no indica que las clases Cuentas no contienen campos para hacer balances; bien podría estar presente tal caché, pero está oculto para los clientes de la clase Cuenta.

Figura 5-6: *Clase de periodo de tiempo*

Puedo ilustrar cómo los elementos derivados indican restricciones de una clase titulada Periodo de tiempo (véase la Figura 5-6). Si éste es un diagrama de especificación, entonces, aunque sugiera que inicio y fin están almacenados y duración se calcula, el programador puede, de hecho, implementar esta clase de cualquier

modo que mantenga dicha conducta externa. Por ejemplo, es perfectamente aceptable almacenar inicio y duración y calcular fin.

En los diagramas de implementación, los valores derivados son valiosos para anotar campos que se emplean como cachés por razones de desempeño. Al marcarlos y registrar la derivación del caché, es más fácil ver explícitamente lo que hace éste. Con frecuencia, refuerzo esto en el código indicando la palabra "caché" en tal campo (por ejemplo, cachéBalance).

En los diagramas conceptuales, utilizo marcadores derivados para que me recuerden dónde existen tales derivaciones y confirmar, con los expertos del dominio, que existen las derivaciones. Entonces, éstas se correlacionan con su empleo en los diagramas de especificación.

En los mundos del OMT y de Odell, las asociaciones derivadas se indican con una diagonal en la línea de asociación. Este uso no es parte del UML, aunque, no obstante, confieso que lo utilizo. Me parece más claro, en particular cuando no le doy nombre a la asociación.

Interfaces y clases abstractas

Una de las grandes cualidades del desarrollo orientado a objetos es que se pueden variar las interfaces de las clases, independientemente de la implementación. Gran parte del poder del desarrollo de objetos surge de esta propiedad. Muy pocos, sin embargo, hacen un buen uso de ella.

Los lenguajes de programación usan un solo artificio, la clase, que contiene tanto la interfaz como la implementación. Cuando se subclasifica, se heredan ambas. Se recurre muy poco a utilizar la interfaz como un artificio separado, lo cual es una verdadera lástima.

Una interfaz pura (como la de Java) es una clase sin implementación y, por tanto, tiene declaraciones de operaciones, pero no cuerpos de método ni campos. Con frecuencia, las interfaces se declaran por medio de clases abstractas. Dichas clases pueden proporcionar alguna realización, pero con frecuencia se usan principalmente para declarar una interfaz. La cuestión es que la subclasificación (o cualquier otro mecanismo) proporcionará la implementación, pero los clientes sólo verán la interfaz y no la realización.

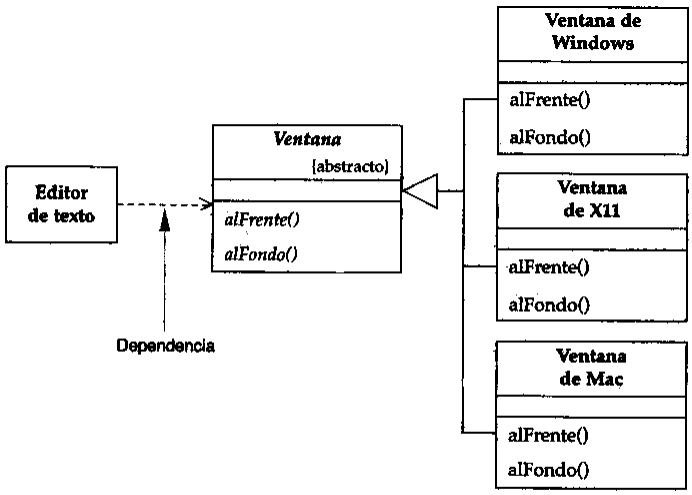

Figura 5-7: *Ventana como clase abstracta*

El editor de texto representado en la Figura 5-7 es un ejemplo típico de lo anterior. Para permitir que el editor sea independiente de la plataforma, definimos una clase abstracta Ventana, independiente de la plataforma. Esta

clase no tiene operaciones; sólo define una interfaz para el editor de texto. Las subclases específicas de la plataforma se pueden emplear como se desee.

Si se tiene una clase o método abstractos, la convención en el UML es poner con cursivas el nombre del elemento abstracto. La restricción {abstracto} también puede utilizarse (y también en sustitución). Por mi parte, manejo {abstracto} en los pizarrones, porque no puedo escribir el texto en cursivas. Sin embargo, teniendo una herramienta de diagramación, prefiero la elegancia de las cursivas.

Pero la subclasificación no es la única manera de hacer lo anterior. Java ofrece una interfaz específica y el compilador revisa que la clase ofrezca una implementación para todas las operaciones definidas para dicha interfaz.

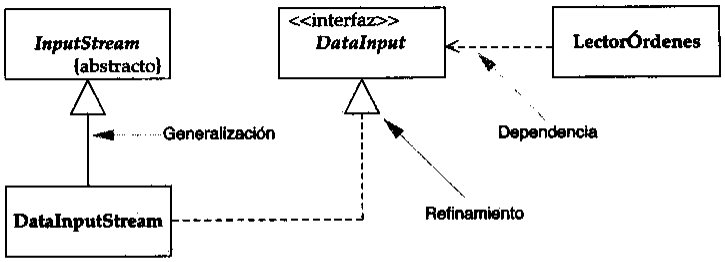

Figura 5-8: *Interfaces y clase abstracta: un ejemplo de Java*

En la Figura 5-8, vemos InputStream, DataInput y DataInputStream (definidos en el archivo java.io estándar). InputStream es una clase abstracta; DataInput es una interfaz.

Alguna clase cliente, digamos, LectorÓrdenes necesita utilizar la funcionalidad de DataInput. La clase de DataInputStream implementa las interfaces DataInput e InputStream y es una subclase de esta última.

El vínculo entre DataInputStream y DataInput es una relación de refinamiento. El**refinamiento** es un término general que se emplea en el UML para indicar un mayor nivel de detalle. Puede usarse para la implementación de interfaces o algún otro fin (véanse los libros de los tres amigos para mayores detalles). El refinamiento deliberadamente es similar a la generalización.

En un modelo de especificación, tanto la subclasificación como el refinamiento se representarían como una subtipificación; la diferencia entre el refinamiento y la generalización es válida sólo en los modelos de implementación. Algunos modeladores prefieren usar «tipo» en lugar de «interfaz». En mi opinión, soy partidario de «interfaz», pues así queda explícito el papel de la interfaz.

El vínculo entre LectorÓrdenes y DataInput es una dependencia. Muestra que LectorÓrdenes usa la interfaz de DataInput con algún fin. En esencia, una dependenciaindica que, si la interfaz de DataInput cambia, también tiene que cambiar el LectorÓrdenes. Uno de los objetivos del desarrollo es el de mantener a un mínimo las dependencias, de modo que los efectos de los cambios también sean mínimos.

La Figura 5-9 muestra una notación alterna más compacta. En ella, las interfaces están representadas por pequeños círculos llamados con frecuencia paletas (*lollipops*) que surgen de las clases que las implementan.

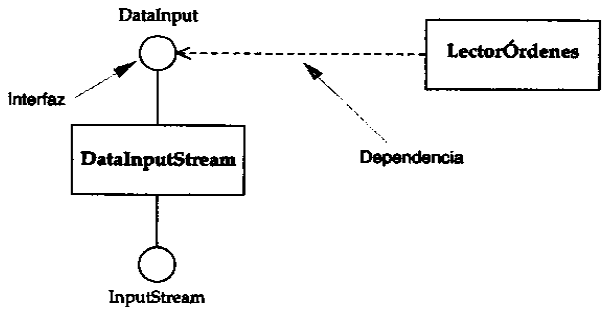

Figura 5-9: *Notación de paletas para representar interfaces*

No existe diferencia alguna entre el refinamiento de una interfaz y la subclasificación de una clase abstracta.

Las clases abstractas y las interfaces son similares, aunque existe una diferencia entre ellas. Ambas le permiten definir una interfaz y diferir su realización para más tarde. Por otra parte, la clase abstracta permite añadir la implementación de algunos de los métodos; en cambio, una interfaz obliga a diferir la definición de todos los métodos.

Objetos de referencia y objetos de valor

Una de las cosas comunes que se dicen de los objetos es que tienen identidad. Esto es cierto, pero el asunto no es tan simple como parece. En la práctica, usted se percata de que la identidad es importante para los objetos de referencia, pero no tanto para los objetos de valor.

Los **objetos de referencia** son cosas como Cliente. La identidad aquí es muy importante, porque usualmente se quiere que un solo objeto de software designe a un cliente del mundo real. Cualquier objeto que se refiera al objeto Cliente lo hará por medio de una referencia o un apuntador; todos los objetos que se refieran a este Cliente se referirán al mismo objeto de software. De esta manera, los cambios en un Cliente estarán disponibles para todos los usuarios del Cliente.

Si se tienen dos referencias de un Cliente y se desea saber si son iguales, por lo común se comparan sus identidades. Las copias probablemente no estén permitidas y, si lo están, tenderán a hacerse en raras ocasiones, tal vez para propósitos de respaldo o para su réplica en una red. Si se hacen copias, habrá que determinar la manera de sincronizar los cambios.

Los **objetos de valor** son cosas como Fecha. Con frecuencia, se tienen varios objetos de valor que representan al mismo objeto en la realidad. Por ejemplo, es normal tener a cientos de objetos que designen el 1 de enero de 1997. Todas éstas son copias intercambiables. A menudo se crean y destruyen fechas nuevas.

Si se tienen dos fechas y se quiere saber si son iguales, no se ven sus identidades; se ven los valores que representan. Esto, por lo general, significa que hay que escribir un operador de prueba de igualdad, el cual, hablando de fechas, efectuará pruebas sobre año, mes y día (o cualquiera que sea la representación interna). Usualmente, cada objeto que se refiera al 1° de enero de 1997 tendrá su propio objeto dedicado, pero en ocasiones se pueden tener fechas combinadas.

Los objetos de valor deben ser **objetos de inmutables.** En otras palabras, no se debe poder tomar un objeto con fecha del 1 de enero de 1997 y cambiarlo al 2 de enero de 1997. En cambio, se deberá crear un nuevo objeto 2 de enero de 1997 y vinculado al primer objeto. La razón de esto es que si la fecha fuera compartida, se actualizaría la fecha de otro objeto de manera impredecible.

En C ++ esto no constituye un problema, pues se hace un gran esfuerzo por no compartir fechas; el compartir objetos de valor conduce a problemas de administración de memoria. En lugar de eso, se puede suplantar la asignación para hacer copias. En los ambientes de memoria administrada, tales como Java, esto es más importante, en especial porque, en Java, las fechas no son inmutables. Como regla empírica, no cambie un objeto de valor.

En el pasado, la diferencia entre los objetos de referencia y los objetos de valor era más clara. Los objetos de valor eran los valores interconstruidos del sistema de tipos. En la actualidad, se puede extender el sistema de tipos con clases propias del usuario, por lo que esta cuestión requiere de mayor análisis. En el UML, los atributos se usan generalmente para objetos de valor y las asociaciones para objetos de referencia. También se puede manejar composición para objetos de valor.

No considero que la diferencia entre objetos de referencia y objetos de valor sea de utilidad en los modelos conceptuales. No hay diferencia entre las dos construcciones desde la perspectiva conceptual. Si represento un vínculo con un objeto de valor mediante una asociación, marco como [*] la multiplicidad del papel desde el valor a su usuario, a menos que exista una regla de unicidad (como, digamos, un número de secuencia).

Colecciones para relaciones de valor múltiple

Una **relación de valor múltiple** (*multi-valued role*) es aquella cuyo límite más alto de multiplicidad es mayor que 1 (por ejemplo, *). La convención usual es que las relaciones de valor múltiple se consideran como conjuntos. No hay un ordenamiento para los objetos destino y ningún objeto destino aparece más de una vez en la relación. Sin embargo, se pueden cambiar estas premisas al conectar una restricción.

La restricción {ordenado} implica que hay un ordenamiento para los objetos destino (esto es, los objetos destino forman una lista). Los objetos destino pueden aparecer sólo una vez en la lista.

Yo utilizo la restricción {bolsa} para indicar que los objetos destino pueden aparecer más de una vez, pero no hay un ordenamiento (si quisiera un ordenamiento y varias apariciones, indicaría {ordenado bolsa}, aunque aún no he tenido necesidad de hacerlo). También manejo la restricción {jerarquía} para indicar que los objetos destino forman una jerarquía.

Congelado

Congelado (*frozen*) es una restricción que, según la define el UML, se aplica a lospapeles pero también puede aplicarse a los atributos y a las clases.

En un papel o en un atributo, congelado indica que sus valores no pueden cambiar durante la vida del objeto origen. El valor debe establecerse en el momento de la creación del objeto y nunca puede cambiar. El valor inicial puede ser nulo. Ciertamente, si éste es el valor en el momento de la construcción del objeto, así seguirá, mientras el objeto viva. Esto implica que debe haber un argumento para este valor en un constructor y que no debe haber ninguna operación que lo actualice.

Cuando se aplica a una clase, congelado indica que todas las funciones y atributos asociados con esa clase están congelados.

Congelado no es igual que la restricción sólo-lectura. La restricción sólo-lectura implica que un valor no puede cambiarse directamente, sino que puede cambiar debido a la modificación de algún otro valor. Por ejemplo, si una persona tiene una fecha de nacimiento y una edad, entonces la edad puede ser de sólo lectura, pero no

congelado. Marco "congelamiento" mediante la restricción {congelado} y los valores de sólo lectura con {sólo lectura}.

Si usted tiene la intención de "congelar" algo, tenga en cuenta que las personas cometen errores. En el software modelamos lo que conocemos sobre el mundo, no cómo es el mundo. Si modeláramos cómo es el mundo, el atributo de "fecha de nacimiento" de un objeto Persona sería inmutable, pero, en la mayoría de los casos, querríamos cambiarlo si nos percatamos de que un registro previo es incorrecto.

Clasificación y generalización

Con frecuencia, oigo a las personas referirse a la subtipificación como la relación de "es un". Le recomiendo muy encarecidamente tener cuidado con esta manera de pensar. El problema es que la frase "es un" puede significar cuestiones diferentes.

Considere las siguientes frases:

1. Shep es un Border Collie.
2. Un Border Collie es un Perro.
3. Los Perros son Animales.
4. Un Border Collie es una Raza.
5. El Perro es una Especie.

A continuación, intente combinar las frases anteriores. Si se combinan las frases 1 y 2, se obtiene "Shep es un Perro"; la combinación de las frases 2 y 3 da "los Border Collie son Animales". Y 1 más 2 más 3 da "Shep es un Animal" . Hasta aquí no hay problemas. Ahora, inténtese 1 con 4: "Shep es una Raza". La combinación de 2 con 5 da "un Border Collie es una Especie." Estas combinaciones ya no son tan buenas.

¿Por qué se pueden combinar algunas de estas frases y otras no? La razón es que algunas son **clasificaciones** (el objeto Shep es una instancia del tipo Border Collie) y otras son **generalizaciones** (el tipo Border Collie es un subtipo del tipo Perro). La generalización es transitiva, pero la clasificación no. Puedo combinar una clasificación seguida por una generalización, pero no puedo llevar a cabo la combinación a la inversa.

Puntualizo lo anterior para que desconfíe de la frase "es un". Su uso puede conducir al empleo inapropiado de la subclasificación y a la confusión de responsabilidades. Las frases "los Perros son clases de Animales" y "cada instancia de un Border Collie es una instancia de un Perro" serían, en este caso, mejores pruebas para subtipos.

Asociaciones calificadas

Una **asociación calificada** es el equivalente en UML de un concepto de programación que se conoce de diferentes modos: arreglos asociativos, mapas y diccionarios.

Figura 5-10: *Asociación calificada*

La Figura 5-10 muestra un modo de representar la asociación entre las clases Pedido y Línea de pedido que emplea un calificador. El calificador dice que, en conexión con un Pedido, puede haber una Línea de pedido para cada instancia del Producto.

Conceptualmente, este ejemplo indica que no se puede tener dos instancias de Línea de pedido para el mismo Producto dentro de un Pedido. Desde la perspectiva de especificación, esta asociación calificada implicaría una interfaz semejante a

```
class Pedido {
  public Lineapedido articuloLinea (Producto unProducto)¡
  public void agregaArticuloLinea (Numero cantidad,
          Producto paraProducto);
```

Por tanto, todos los accesos a un Artículo de línea dado requieren como argumento la identidad de un Producto. Una multiplicidad de [1] indicaría que debe haber un Artículo de línea para cada Producto; [*] indicaría que se pueden tener varias Líneas de pedido por Producto, pero que el acceso a los Artículos de línea sigue estando indizado por Producto.

Desde una perspectiva de implementación, lo anterior sugiere el empleo de un arreglo asociativo o de una estructura de datos similar para contener las líneas del pedido.

```
Class Pedido {
   private Dictionary _articulosLinea;
```

En el modelado conceptual, yo utilizo el artificio calificador únicamente para mostrar restricciones del tipo "una sola Línea de pedido por Producto en el Pedido." En el caso de los modelos de especificación, me sirve para mostrar una interfaz de búsqueda indizada. Me siento cómodo manejando esto y asociación no calificada al mismo tiempo, si se trata de una interfaz adecuada.

Utilizo los calificadores dentro de los modelos de realización para mostrar los usos de un arreglo asociativo o de una estructura de datos similar.

Clase de asociación

Las clases de asociación permiten añadir atributos, operaciones y otras características a las asociaciones, como se muestra en la Figura 5-11.

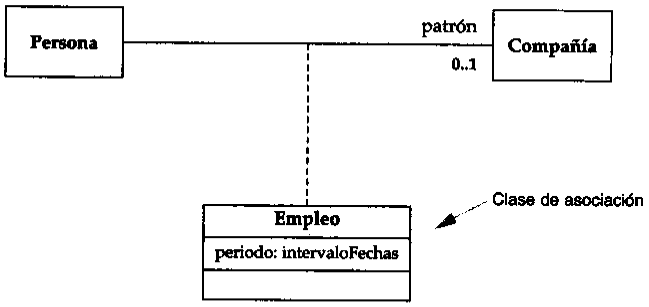

Figura 5-11: *Clase de asociación*

El diagrama nos permite apreciar que una Persona puede trabajar para una sola Compañía. Necesitamos conservar la información sobre el periodo de tiempo que trabaja cada empleado para cada Compañía.

Para lograrlo, añadimos un atributo de intervaloFechas a la asociación. Podríamos agregar este atributo a la clase Persona, pero, en realidad, es un hecho sobre la relación entre una Persona y una Compañía, la cual cambiará si también lo hiciera el patrón de la persona.

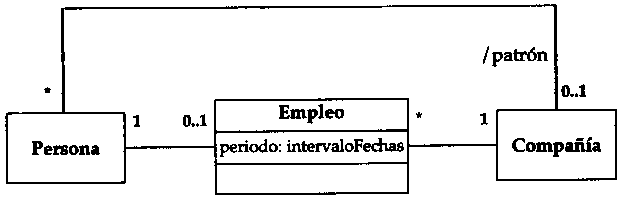

Figura 5-12: *Promoción de una clase de asociación a una clase completa*

La Figura 5-12 muestra otra manera de representar esta información: convertirEmpleo en una clase completa por derecho propio (obsérvese cómo las multiplicidades se han movido en consecuencia). En este caso, cada una de las clases de la asociación original tiene una función de un solo valor con relación a la clase Empleo. El papel del "patrón" es ahora derivado, aunque no sea necesario mostrar esto.

¿Qué beneficio se logra con la clase de asociación que compense la notación extra que hay que recordar? La clase de asociación añade una restricción nueva, en el sentido de que sólo puede haber una instancia de la clase de asociación entre cualesquiera de dos objetos participantes. Me parece que hace falta un ejemplo.

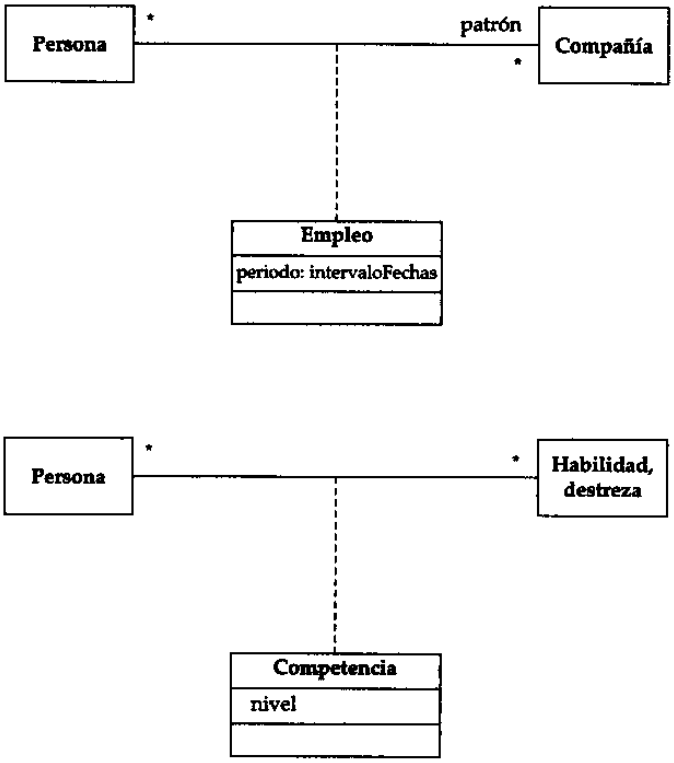

Figura 5-13: *Sutilezas de la clase de asociación*

Observe por un momento los dos diagramas que aparecen en la Figura 5-13. Ambos tienen formas muy parecidas. Sin embargo, podríamos imaginamos a una Persona trabajando para la misma Compañía en diferentes periodos de tiempo (es decir, la persona deja la compañía y después vuelve). Esto significa que una Persona puede tener más de una asociación de Empleo con la misma Compañía a través del tiempo.

En lo que se refiere a las clases Persona y Oficio, sería difícil apreciar por qué una persona tendría más de una Competencia en el mismo oficio; de hecho, probablemente se consideraría como un error.

En el UML sólo es legal el segundo caso. Se puede tener solamente una Competencia para cada combinación de Persona y Oficio. El diagrama superior de la Figura 5-13 no permitiría a una Persona tener más de un Empleo en la misma Compañía. Si es necesario permitir esto, habrá que convertir a Empleo en una clase completa, a la manera de la Figura 5-12

En el pasado, los modeladores tenían varias premisas sobre el significado de una clase de asociación en estas circunstancias. Algunos suponían que sólo podían tenerse combinaciones únicas (como es el caso de la competencia); mientras que otros más no suponían tal restricción. Muchos no lo tomaban en consideración en lo absoluto y podrían asumir la restricción en algunos lugares y en otros no. Así pues, cuando utilice el UML, recuerde que la restricción siempre está allí.

Con frecuencia, se encuentra este tipo de artificio con información histórica, como en el caso de Empleo anterior. Aquí, un patrón útil es el de Mapa Histórico. Podemos usado definiendo un estereotipo «historia» (véase la figura 5-14).

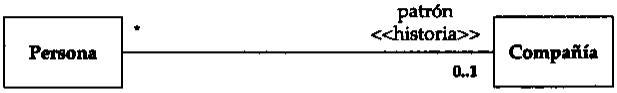

Figura 5-14: *Estereotipo de historia para las asociaciones*

El modelo indica que una Persona únicamente puede trabajar para una sola Compañía en un momento dado. Sin embargo, a través de un tiempo determinado, una Persona puede trabajar para varias Compañías. Esto sugiere una interfaz semejante a las líneas siguientes

```
class Persona {
   // obtiene el patrón actual
   Compañía patrón();
   // patrón en una fecha dada
   Compañía patrón(Date);
   void cambiaPatron(Compañía nuevopatron, Date cambiaFecha);
   void dejaPatron (Date cambiaFecha):
```

El estereotipo «historia» no forma parte del UML, pero lo menciono aquí por dos razones. Primera, es un concepto que he encontrado útil en varias ocasiones durante mi ejercicio como modelador. Segundo, muestra cómo se pueden usar los estereotipos para ampliar el UML.

Clase con parámetro

Algunos lenguajes, particularmente C ++, tienen el concepto de **clase con parámetro**(también conocida como **plantilla).**

La utilidad más obvia de este concepto es en el trabajo con colecciones en un lenguaje con especificación estricta de tipos de datos. De este modo, se puede definir la conducta de los conjuntos en general por medio de la definición de una clase de plantillas Conjunto.

```
class Conjunto <T> {
   void insert (T nuevoElemento) ;
   void remove (T unElemento);
```

Hecho esto, se podrá emplear la definición general para hacer clases de conjuntos para elementos más específicos.

```
Conjunto <Empleado> conjuntoEmpleados;
```

Una clase con parámetro en UML se declara mediante la notación que aparece en la Figura 5-15.

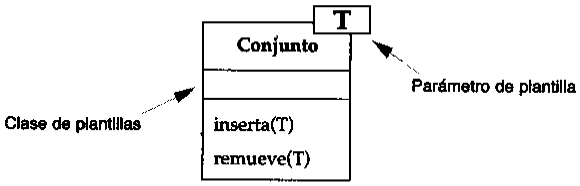

Figura 5-15: *Clase con parámetro*

La T en el diagrama es un marcador para el parámetro de tipo (se podrá tener más de uno). En un lenguaje sin tipos, como Smalltalk, esta cuestión no surge, por lo que el concepto carece de utilidad.

Al uso de una clase con parámetro, tal como Conjunto<Empleado> mostrado antes, se le llama **elemento enlazado** (*bound*).

Se puede mostrar un elemento enlazado de dos maneras. La primera refleja la sintaxis C++ (véase la figura 5-16).

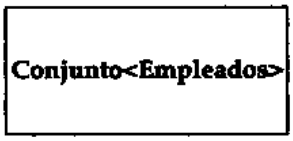

Figura 5-16: *Elemento enlazado (versión 1)*

La anotación alternativa (véase la figura 5-17) retuerza el vínculo con la plantilla y permite renombrar el elemento enlazado.

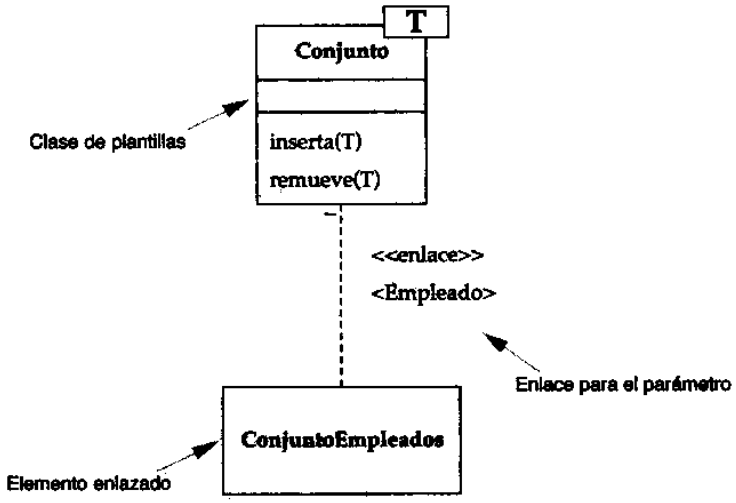

Figura 5-17: *Elemento enlazado (versión 2)*

El estereotipo «vínculo» es un estereotipo en la relación de refinamiento. Esta relación indica que el ConjuntoEmpleados se conformará con la interfaz de Conjunto. En términos de especificación, el ConjuntoEmpleados es un subtipo de Conjunto. Esto se ajusta al otro modo de implementar las colecciones específicas de tipos, que consiste en declarar todos los subtipos apropiados.

Sin embargo, usar un elemento enlazado no es lo mismo que subtipificar. No se permite añadir características al elemento enlazado: es especificado en su totalidad por su plantilla; sólo se está añadiendo información de tipo restrictivo. Si se desea agregar características, habrá que crear un subtipo.

Aunque el empleo clásico de las clases con parámetros es con colecciones, hay muchas otras maneras de utilizarlas en C++

Las clases con parámetros le permiten a usted usar **tipificación derivada.** Cuando se escribe el cuerpo de la plantilla, es posible invocar operaciones sobre el parámetro. Posteriormente, cuando se declara un elemento enlazado, el compilador trata de asegurarse de que el parámetro que se ha proporcionado maneje las operaciones requeridas por la plantilla.

Éste es un mecanismo de tipificación derivada, pues no hay necesidad de definir un tipo para el parámetro; el compilador determina si el enlace es viable viendo a la fuente de la plantilla. Esta propiedad es medular para las clases con parámetros en el STL de C ++; estas clases pueden servir, además, para otros trucos interesantes.

El empleo de clases con parámetros tiene repercusiones. Por ejemplo, pueden causar una considerable inflación del código en C ++. Por mi parte, pocas veces incluyo clases con parámetros en el modelado conceptual, ante todo porque principalmente se usan para las colecciones, las cuales se dan a entender mediante asociaciones. Sólo uso clases con parámetros en el modelado de especificación y de implementación, si son manejadas por el lenguaje con el que estoy trabajando.

La visibilidad

Debo confesar que esta sección me produce cierta perturbación.

La **visibilidad** es uno de esos temas que, en principio, es simple, pero tiene sutilezas complejas. La idea básica es que cualquier clase tiene elementos públicos y privados. Los elementos públicos pueden ser usados por

cualquier otra clase; los elementos privados sólo pueden ser usados por la clase propietaria. Sin embargo, cada lenguaje tiene sus propias reglas. Aunque todos los lenguajes utilizan términos como " **público** ", "**privado** " y " **protegido** " (*public, private, protected*), su significado varía según el lenguaje. Estas diferencias son pequeñas, pero generan confusión, en especial para quienes trabajamos con más de un lenguaje.

El UML trata de resolver este problema, sin complicarse demasiado. En esencia, en el UML se puede etiquetar cualquier atributo u operación con un indicador de visibilidad. Se puede usar el marcador que se quiera y su significado depende del lenguaje. Sin embargo, el UML proporciona tres abreviaciones de visibilidad (un tanto difíciles de recordar): + (público), - (privado), y # (protegido).

Siento la tentación de dejar aquí el tema pero, por desgracia, cada uno dibuja la visibilidad de una manera específica. Por tanto, para entender en realidad algunas de las diferencias comunes que existen entre los modelos es necesario comprender los enfoques que adoptan los diversos lenguajes, con respecto a la visibilidad. Así pues, aspiremos profundo y sumerjámonos en el fango.

Iniciamos con C++, ya que es la base del empleo regular del UML.

- Un miembro público es visible en todo el programa y puede ser llamado por cualquier objeto dentro del sistema.
- Un miembro privado sólo puede ser usado por la clase que lo define.
- Un miembro protegido sólo puede ser usado por (a) la clase que lo define, o (b) una subclase de esa clase.

Considérese una clase Cliente que tiene una subclase Cliente personal. Considérese también el objeto Martín, que es una instancia de Cliente personal. Martín puede usar cualquier miembro público de cualquier objeto del sistema. También puede usar cualquier miembro privado de la clase Cliente personal. Martín no puede usar ningún miembro privado definido en Cliente; sin embargo, sí puede usar los miembros protegidos de Cliente y los usuarios protegidos de Cliente personal.

Java es similar a C++ en el hecho de que ofrece libre acceso a los miembros de otros objetos de su misma clase. Java también añade un nuevo nivel de visibilidad al que llama paquete (package). Un miembro con visibilidad de paquete sólo puede ser accedido por instancias de otras clases dentro del mismo paquete.

Siguiendo con nuestro tema, para asegurar que las cosas no sean tan sencillas, Java redefine levemente la visibilidad protegida. En Java, a un miembro protegido puede accederse por subclases, pero también por cualquier otra clase que esté en el mismo paquete que la clase propietaria. Esto quiere decir que, en Java, protegido es más público que paquete.

Java también permite que las clases sean marcadas como públicas o paquetes. Los miembros públicos de una clase pública pueden ser usados por cualquier clase que importe el paquete al que pertenezca la clase. Una clase de paquete puede ser usada sólo por otras clases del mismo paquete.

C ++ añade el toque final. Un método o clase C ++ puede ser convertido en amigo de una clase. Un **amigo** (*friend*) tiene completo acceso a todos los miembros de una clase. De aquí viene la frase: "en C++, los amigos se tocan entre ellos las partes privadas".

Cuando esté manejando visibilidad, emplee las reglas del lenguaje con que trabaja. Cuando vea un modelo UML que provenga de otra parte, desconfíe del significado de los marcadores de visibilidad y tenga presente la manera en que pueden cambiar tales significados de un lenguaje a otro.

En general, encuentro que las visibilidades cambian a medida que se trabaja en el código, así que no las tome demasiado en cuenta, desde un principio.

Características del alcance de clase

Se pueden definir operaciones o atributos en el nivel de alcance de clase.

Esto significa que éstos son característicos de la clase, en lugar de ser de cada objeto. Estas operaciones o atributos corresponden a los miembros estáticos en C ++ y Java y a los métodos y variables de clase en Smalltalk. Estas características se muestran de la misma manera que cualquier otra operación o atributo, a excepción de que se subrayan.

DIAGRAMAS DE ITERACCIÓN

Resumen

Los diagramas de interacción son modelos que describen la manera en que colaboran grupos de objetos para cierto comportamiento.

Habitualmente, un diagrama de interacción capta el comportamiento de un solo caso de uso. El diagrama muestra cierto número de objetos y los mensajes que se pasan entre estos objetos dentro del caso de uso.

Índice de Contenidos

Introducción

Los diagramas de interacción son modelos que describen la manera en que colaboran grupos de objetos para cierto comportamiento.

Habitualmente, un diagrama de interacción capta el comportamiento de un solo caso de uso (Enlace Cap 03). El diagrama muestra cierto número de ejemplos de objetos y los mensajes que se pasan entre estos objetos dentro del caso de uso.

Ilustraré este enfoque mediante un caso de uso simple que exhibe el comportamiento siguiente:

- La ventana Entrada de pedido envía un mensaje "prepara" a Pedido.

- El Pedido envía entonces un mensaje "prepara" a cada Línea de pedido dentro del Pedido.
- Cada Línea de pedido revisa el Artículo de inventario correspondiente.
  -
  - Si esta revisión devuelve "verdadero", la Línea de pedido descuenta la cantidad apropiada de Artículo de inventario del almacén.
  - Si no sucede así, quiere decir que la cantidad del Artículo de inventario ha caído más abajo del nivel de reposición (en original reorden) y entonces dicho Artículo de inventario solicita una nueva entrega.

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

Diagramas de secuencia

En un **diagrama de secuencia,** un objeto se muestra como caja en la parte superior de una línea vertical punteada (véase la Figura 6-1.

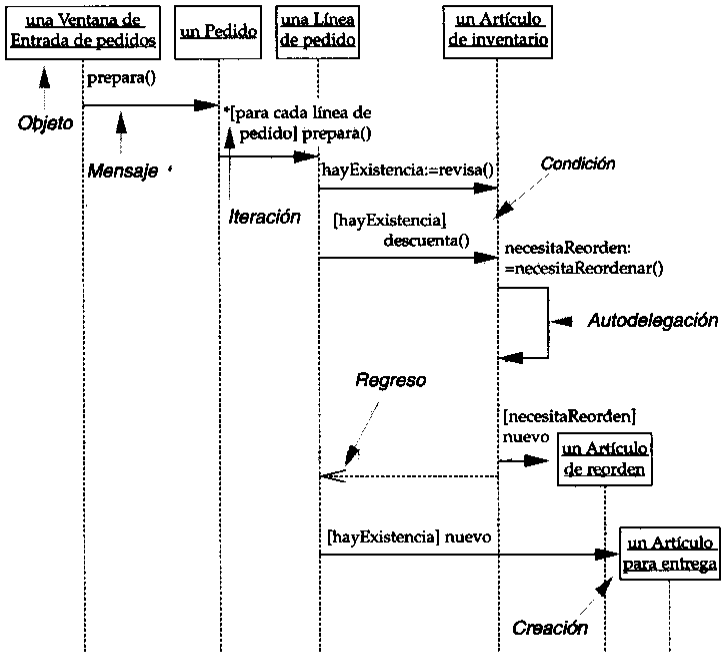

Figura 6-1: *Diagrama de secuencia*

Esta línea vertical se llama **línea de vida del objeto.** La línea de vida representa la vida del objeto durante la interacción. Esta forma fue popularizada inicialmente por Jacobson.

Cada mensaje se representa mediante una flecha entre las líneas de vida de dos objetos. El orden en el que se dan estos mensajes transcurre de arriba hacia abajo. Cada mensaje es etiquetado por lo menos con el nombre del mensaje; pueden incluirse también los argumentos y alguna información de control, y se puede mostrar la**autodelegación,** que es un mensaje que un objeto se envía a sí mismo, regresando la flecha de mensaje de vuelta a la misma línea de vida.

Dos partes de la información de control son valiosas. Primero, hay una **condición,** que indica cuándo se envía un mensaje (por ejemplo, [*necesitaReorden*])(Se necesita reponer el almacen con articulos). El mensaje se envía sólo si la condición es verdadera. El segundo marcador de control útil es el **marcador de iteración,** que muestra que un mensaje se envía muchas veces a varios objetos receptores, como sucedería cuando se itera

sobre una colección. La base de la iteración se puede mostrar entre corchetes (como en *[para cada línea de pedido]).

Como se puede apreciar, Figura 6-1 es muy simple y tiene un atractivo visual inmediato; en ello radica su gran fuerza.

Una de las cuestiones más difíciles de comprender en un programa orientado a objetos es el flujo de control general. Un buen diseño tiene muchísimos pequeños métodos en diferentes clases, y a veces resulta muy complicado determinar la secuencia global de comportamiento. Podrá acabar leyendo el código, tratando de encontrar dónde está el programa. Esto ocurre así, en especial, para todos aquellos que comienzan a trabajar con objetos. Los diagramas de secuencia le ayudan a ver la secuencia.

Este diagrama incluye un **regreso,** el cual indica el regreso de un mensaje, no un nuevo mensaje. Los regresos difieren de los mensajes normales en que la línea es punteada.

Los diagramas POSA (Buschmann et al., 1996), en los que se basa una gran parte de las anotaciones de la notación de gráficas de secuencia de UML, emplean ampliamente los regresos. Yo no lo hago así. He observado que los regresos saturan el diagrama y tienden a oscurecer el flujo. Todos los regresos se hallan implícitos por el modo como se secuencian los mensajes. Sólo utilizo regresos cuando aumentan la claridad del diagrama.

Mi consejo es mostrar los regresos cuando ayudan a aumentar la claridad. La única razón por la que usé un regreso en la Figura 6-1 fue para demostrar la notación; si se elimina el regreso, a mi juicio el diagrama continúa tan claro como antes.

En el UML 1.0 los regresos se indicaban mediante puntas de flecha tipo pluma, en lugar de líneas punteadas. Las he encontrado tan difíciles de implementar que de todas maneras he utilizado las líneas punteadas, de modo que me alegra ver el cambio.

Esto lo menciono, debido a que quisiera ofrecer aquí un pequeño consejo: evite ir en contra de la notación del UML. Esta notación se volverá cabalmente comprendida, y hacer algo fuera de la norma dañará la comunicación del diseñador con otros diseñadores. Sin embargo, si hay algo que esté causando una gran confusión, yo haría algo fuera de la norma. Después de todo, el propósito principal del diagrama es la comunicación. Si es que llega a romper las reglas del UML, hágalo muy de vez en cuando y defina con claridad lo que ha hecho.

Procesos concurrentes y activaciones

Los diagramas de secuencia son valiosos también para los procesos concurrentes.
En la Figura 6-2 vemos algunos objetos que revisan una transacción bancaria.

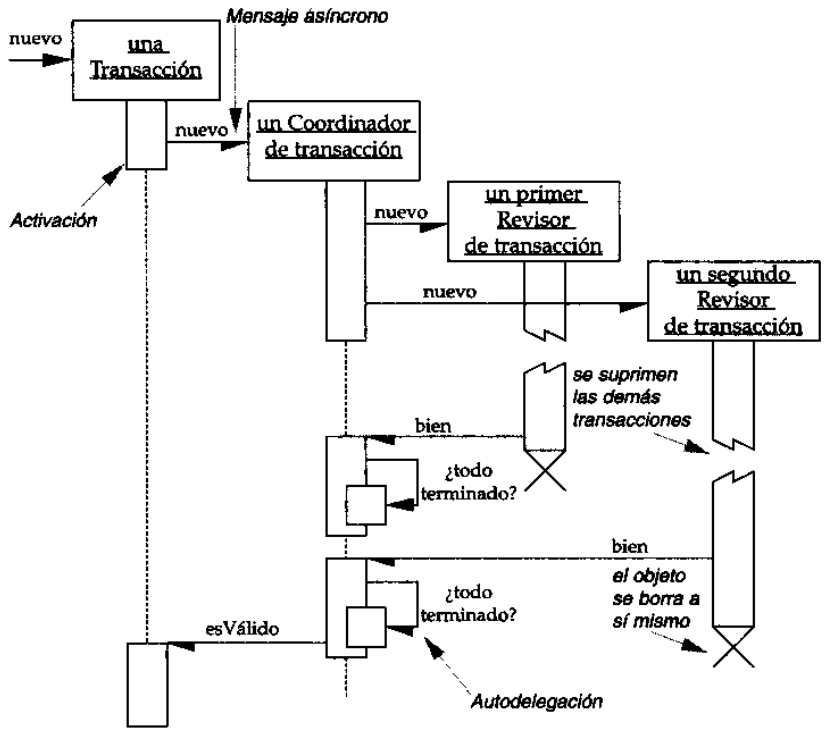

Figura 6-2: *Procesos y activaciones concurrentes*

Cuando se crea una Transacción, ésta crea un Coordinador de transacción que coordina el trámite de la Transacción. Este coordinador crea una cantidad (en el presente caso, dos) de objetos Revisor de transacción, cada uno de los cuales es responsable de una revisión en particular. Este proceso permitirá añadir con facilidad otros procesos de revisión, porque cada registro es llamado asincrónicamente y opera en paralelo.

Cuando termina un Revisor de transacción, se lo notifica al Coordinador de transacción. El coordinador comprueba si todos los revisores respondieron; de no ser así, no hace nada. Si todos han respondido indicando terminaciones exitosas, como en el presente caso, entonces el coordinador notifica a la Transacción que todo está bien.

La Figura 6-2 introduce varios elementos nuevos en los diagramas de secuencia. En primer lugar, se ven las **activaciones,** que aparecen explícitamente cuando está activo un método, ya sea porque está efectuando operaciones o porque se encuentra esperando la devolución de una subrutina. Muchos diseñadores utilizan las activaciones todo el tiempo. A mi juicio, éstas no aportan mucho a la ejecución de procedimientos. Por tanto, sólo las uso en situaciones concurrentes.

Las medias cabezas de flecha indican **mensajes asíncronos.** Un mensaje asíncrono no bloquea al invocador, por lo cual puede continuar con su proceso. Un mensaje asíncrono puede hacer alguna de estas tres cosas:

1. Crear un nuevo proceso, en cuyo caso se vincula con el principio de una activación.
2. Crear un nuevo objeto.
3. Comunicarse con un proceso que ya está operando.

El **borrado** (*deletion*) de un objeto se muestra con una X grande. Los objetos pueden auto destruirse (como se muestra en la Figura 6-2), o pueden ser destruidos mediante otro mensaje (véase la Figura 6-3).

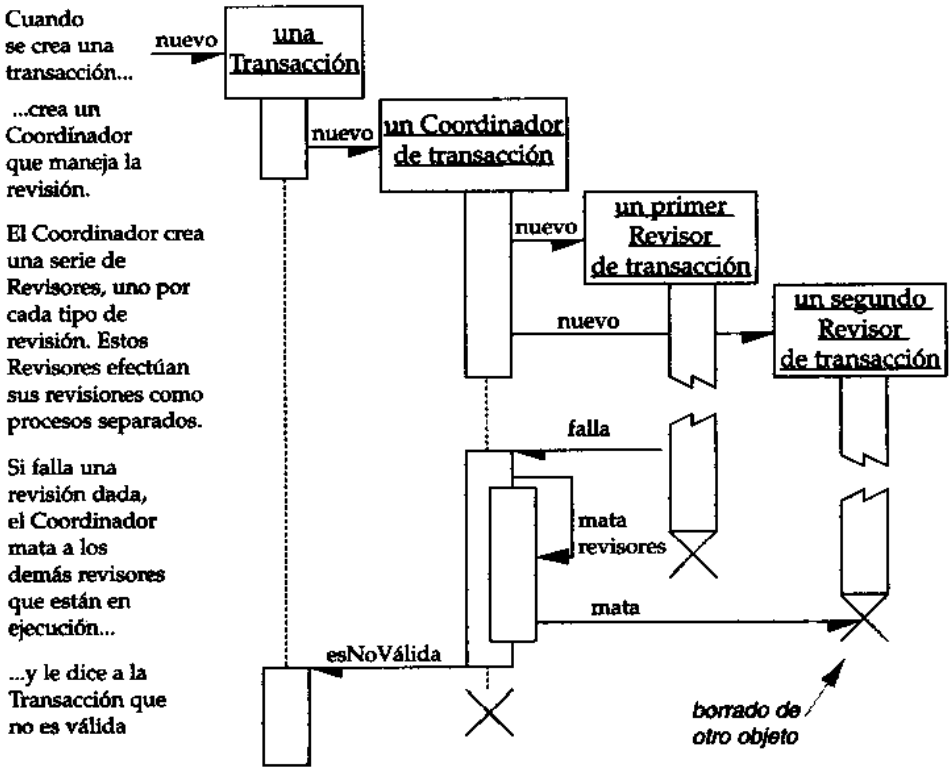

Figura 6-3: *Diagrama de secuencia: revisión de fallas*

Texto de la imagen

- Cuando se crea una nueva transacción
- Crea un Coordinador que maneja la revisión.
- El Coordinador crea una serie de Revisores, uno por cada tipo de revisión. Estos Revisores efectúan sus revisiones como procesos separados.
- Si falla una revisión dada, el Coordinador mata a los demás revisores que están en ejecución...
- y le dice a la Transacción que no es válida

Se pueden mostrar con más claridad las consecuencias de la autodelegación cuando se tienen activaciones. Sin ellas, o sin la notación de apilamiento que se usa aquí, es difícil decir dónde ocurrirán más llamadas tras una autodelegación: en el método invocador o en el invocado. Las activaciones de apilamientos dejan en claro este aspecto. Descubro que, en algunas ocasiones, ésta es una razón para utilizar activaciones en una interacción de procedimiento, aun cuando por lo común no uso las activaciones en estos casos.

Las Figuras 6-2 y 6-3 muestran dos de las situaciones del caso de uso de "revisión de transacciones". He dibujado cada situación por separado. Hay técnicas para combinar la lógica condicional en un solo diagrama, pero prefiero no usadas, ya que complican demasiado el diagrama.

En la Figura 6-3 me he servido de una técnica muy útil: he insertado descripciones textuales de lo que sucede en el lado izquierdo del diagrama de secuencia. Ello implica la alineación de cada bloque de texto con el mensaje apropiado dentro del diagrama. Esto ayuda a comprender el diagrama (a cambio de un poco de trabajo extra), y lo hago con aquellos documentos que conservaré, pero no con bosquejos desde cero.

Diagramas de colaboración

La segunda forma del diagrama de interacción es el **diagrama de colaboración.**

En los diagramas de colaboración, los objetos ejemplo se muestran como iconos. Las flechas indican, como en los diagramas de secuencia, los mensajes enviados dentro del caso de uso dado. Sin embargo, en esta ocasión la secuencia se indica numerando los mensajes.

El numerar los mensajes dificulta más ver la secuencia que poner las líneas verticales en la página. Por otra parte, la disposición espacial del diagrama permite mostrar otras cosas mejor. Se puede mostrar cómo se vinculan entre ellos los objetos y emplear la disposición para sobre poner paquetes u otra información.

Se puede usar alguno de los diversos esquemas de numeración para los diagramas de colaboración. El más simple se ilustra en la Figura 6-4. Otro enfoque comprende un esquema de numeración decimal, el cual aparece en la Figura 6-5.

En el pasado, el esquema más utilizado era el de la numeración simple. El UML utiliza el esquema decimal, pues hace evidente qué operación llama a otra y cuál es esa otra, aunque pueda ser más difícil apreciar la secuencia general.

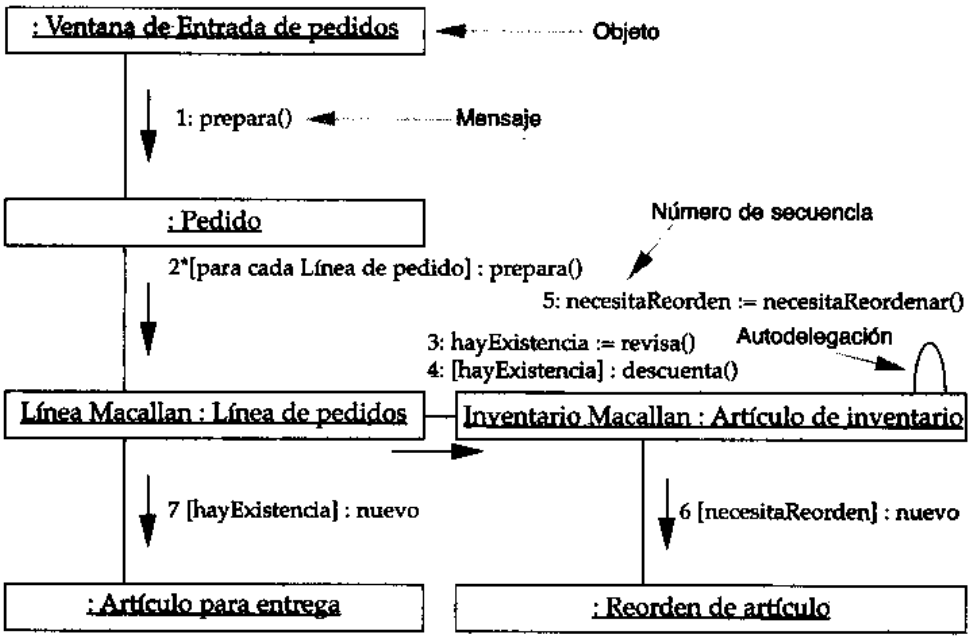

Figura 6-4: *Diagrama de colaboración con numeración simple*

Sin importar el tipo de esquema de enumeración que se utilice, se puede añadir el mismo tipo de control de información que se mostraría en un diagrama de secuencia.

En las Figuras 6-4 y 6-5 se pueden apreciar las distintas formas del esquema de nombrado de objetos en UML. Tal esquema adopta la forma de nombreObjeto : NombreClase, donde se puede omitir el nombre del objeto o el de la clase. Obsérvese que, si se omite el nombre del objeto, hay que conservar los dos puntos (:), para que quede claro que es el nombre de la clase y no el del objeto. Así, el nombre "Línea Macallan : Línea de pedidos" indica una instancia de Línea de pedidos llamada Línea Macallan (debo decir que éste es un pedido que yo apreciaría de modo particular). Acostumbro a nombrar objetos con el estilo de Smalltalk que empleé en los diagramas de secuencia (este esquema es válido en UML, pues "unObjeto" es un nombre perfectamente correcto para un objeto).

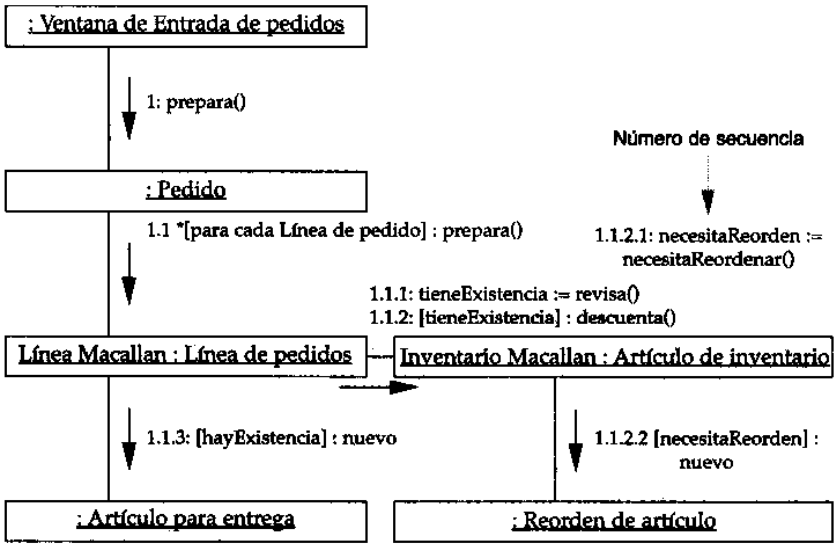

Figura 6-5: *Diagrama de colaboración con numeración decimal*

Comparación de los diagramas de secuencia y de colaboración

Los diferentes desarrolladores tienen distintas preferencias cuando se trata de seleccionar la forma de diagrama de interacción que emplearán. Por lo general, yo prefiero el diagrama de secuencia, debido a que me gusta el énfasis que pone en la secuencia; es fácil apreciar el orden en el que ocurren las cosas. Otros, en cambio, prefieren el diagrama de colaboración, porque pueden usar la distribución para indicar cómo se conectan estáticamente los objetos.

Una de las características principales de ambos tipos de diagrama de interacción es su simplicidad. Se pueden ver con facilidad los mensajes mirando tan sólo el diagrama. Sin embargo, si trata de representar algo más que un simple proceso secuencial, sin mucho comportamiento condicional o de ciclos, la técnica comienza a fallar.

El comportamiento condicional

¿Cuál es el mejor modo de mostrar el **comportamiento condicional**?

Existen dos escuelas de pensamiento. Una consiste en usar diagramas separados para cada situación. La otra consiste en emplear condiciones en los mensajes, a fin de indicar el comportamiento.

Yo prefiero la primera. Los diagramas de interacción están en su mejor forma cuando su comportamiento es simple. Rápidamente pierden su claridad con un comportamiento más complejo. Si quiero captar una conducta complicada en un diagrama simple, prefiero utilizar un diagrama de actividades.

El UML ofrece mucha sintaxis adicional para los diagramas de secuencia, la cual se basa en los patrones del equipo de Siemens (Buschmann et aL, 1996). No entraré aquí en los detalles, ante todo debido a que no me satisface la complejidad que provoca. Para mí, la belleza de los diagramas de interacción reside en su simplicidad y muchas de las notaciones adicionales la echan a perder en su intento por ser completos desde el punto de vista computacional. Le aconsejo que no se precipite utilizando las formas más complejas de los diagramas de interacción, pues quizá encuentre que los más simples tienen un valor superior.

Cuándo utilizar los diagramas de interacción

Se deberán usar los diagramas de interacción cuando se desee ver el comportamiento de varios objetos en un caso de uso. Son buenos para mostrar la colaboración entre los objetos; sin embargo, no sirven tan bien para la definición precisa del comportamiento.

Si desea ver el comportamiento de un solo objeto a través de muchos casos de uso, use un diagrama de estado de transición. Si quiere ver el comportamiento a través de muchos casos de uso o muchos procesos, considere un diagrama de actividad.

DIAGRAMAS DE PAQUETES

Resumen

Explica como usar los diagramas de paquetes, que permiten mostrar las clases que contiene el paquete y sus relaciones., o por el contrario, las relaciones de los paquetes entre si y las dependencias del sistema

Índice de Contenidos

Introducción

Una de las preguntas más antiguas en los métodos de software es: ¿cómo se puede fragmentar un sistema grande en sistemas más pequeños? Preguntamos esto porque, en la medida en que los sistemas se hacen más grandes, se vuelve más difícil comprenderlos, así como entender sus cambios.

Los métodos estructurados se valieron de la **descomposición funcional,** en la cual el sistema en su conjunto se correlacionaba como función y se dividía en subfunciones, que a su vez se dividían en otras subfunciones, y así sucesivamente. Las funciones eran como los casos de uso en un sistema orientado a objetos, en el que las funciones representaban algo que hacía el sistema como un todo.

Eran los días en que el proceso y los datos estaban separados. De tal modo que, además de una descomposición funcional, también había una estructura de datos. Esta última ocupaba el segundo lugar, aunque ciertas técnicas de ingeniería de información agrupaban los registros de datos en áreas temáticas y producían matrices que mostraban la interrelación entre las funciones y los registros de datos.

Es desde este punto de vista que podemos apreciar el gran cambio que han significado los objetos. Ha desaparecido la separación entre el proceso y los datos, y la descomposición funcional, pero la vieja pregunta sigue en pie. Una idea es agrupar las clases en unidades de nivel más alto. Esta idea aparece, aplicada de manera

muy libre, en muchos métodos orientados a objetos. En el UML, a este mecanismo de agrupamiento se le llama **paquete.**

La idea de un paquete se puede aplicar a cualquier elemento de un modelo, no sólo a las clases. Sin cierta heurística que agrupe las clases, el agrupamiento se vuelve arbitrario. El que yo he encontrado más útil, que también es el que recibe mayor énfasis en el UML, es la de pendencia. Empleo el término **diagrama de paquetes** para indicar un diagrama que muestra los paquetes de clases y las dependencias entre ellos.

Hablando estrictamente, los paquetes y las dependencias son elementos de un diagrama de clases, por lo cual un diagrama de paquetes es sólo una forma de un diagrama de clases. En la práctica dibujo estos diagramas por diferentes razones, así que me gusta manejar nombres diferentes.

Existe una (*dependency*) **dependencia** entre dos elementos si los cambios a la definición de un elemento pueden causar cambios al otro. En las clases, la dependencia existe por varias razones: una clase envía un mensaje a otra; una clase tiene a otra como parte de sus datos; una clase menciona a otra como parámetro para una operación. Si una clase cambia su interfaz, entonces los mensajes que envía pueden dejar de ser válidos.

En forma ideal, sólo los cambios a una interfaz de clase deberían afectar a otra clase. El arte del diseño en gran escala implica minimizar las dependencias, de modo tal que se reduzcan los efectos del cambio y se requiera de menos esfuerzo para cambiar el sistema.

En la Figura 7-1 tenemos las clases de dominio que modelan el negocio, las cuales se agrupan en dos paquetes: Pedidos y Clientes. Ambos paquetes son parte de un paquete que abarca todo el dominio. La aplicación de Captura de pedidos tiene dependencias con los dos paquetes del dominio. La Interfaz de Usuario (IU) para Captura de pedidos tiene dependencias con la Aplicación Captura de pedidos y con AWT (un juego de herramientas GUI de Java).

Existe una dependencia entre dos paquetes si existe algún tipo de dependencia entre dos clases cualquiera en los paquetes. Por ejemplo, si cualquier clase en el paquete Lista de correo depende de cualquier clase del paquete Clientes, entonces se da una dependencia entre sus paquetes correspondientes.

Existe una similitud obvia entre dependencia de paquetes y dependencias de compilación. Pero también, de hecho, hay una diferencia vital: con los paquetes, las dependencias no son transitivas.

Un ejemplo de **relación transitiva** es aquella en la que Jim tiene una barba más larga que Grady y éste, una más larga que Ivar, por lo que se deduce que Jim tiene una barba más larga que Ivar. Otros ejemplos incluyen relaciones como "está al norte de" y "es más alto que". Por otra parte, "es un amigo de" no constituye una relación transitiva.

Para apreciar por qué es importante esto para las dependencias, obsérvese de nuevoFigura 7-1 . El cambio a una clase del paquete Pedidos no indica que el paquete IU Captura de pedidos deba ser cambiado. Indica tan sólo que debe revisarse el paquete de aplicación Captura de pedidos para ver si cambia. Sólo si se altera la interfaz del paquete de aplicación Captura de pedidos hay necesidad de cambiar el paquete IU Captura de pedidos. Si esto es así, la aplicación Captura de pedidos está protegiendo a la IU Captura de pedidos de cambios a los pedidos.

Este comportamiento representa el propósito clásico de una arquitectura en capas. De hecho, ésta es la semántica del comportamiento de las "*imports*" de Java, pero no la del comportamiento de los "*include*" de C/C ++, ni la de los prerrequisitos de Envy. El *include*de C/C ++ es transitivo, lo que significa que la IU Captura de pedido sería dependiente del paquete de Pedidos. Una dependencia transitiva hace difícil limitar el alcance de los cambios mediante la compilación.

¿Qué significa trazar una dependencia con un paquete que contenga subpaquetes? Los diseñadores se sirven de convenciones diferentes.

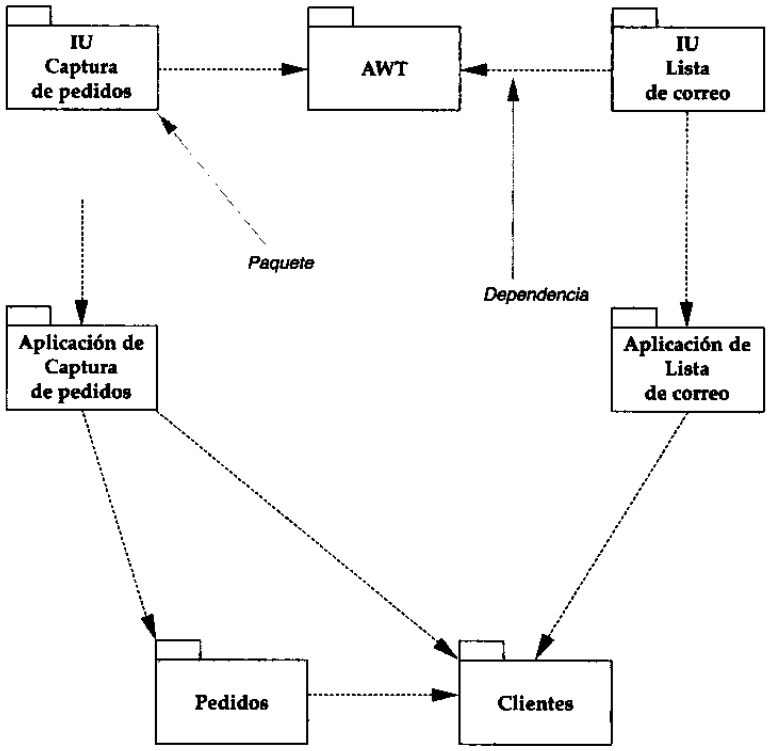

Figura 7-1: *Diagrama de paquetes*

Algunos suponen que dibujar una dependencia hacia un paquete "contenedor" da visibilidad al contenido de todos los paquetes contenidos y a sus respectivos contenidos. Otros consideran que sólo se ven las clases dentro del paquete contenedor, pero no las clases dentro de los paquetes anidados (o sea, la visión es opaca).

Debe declararse explícitamente la convención que se está utilizando en el proyecto o dejarla planteada de manera muy clara colocando estereotipos en los paquetes. Sugiero usar el estereotipo «transparente» para indicar que se pueden ver los paquetes anidados, y usar el estereotipo «opaco» para indicar que no se pueden ver. Aquí, mi convención es que los paquetes sean transparentes.

¿Qué es lo que se ve, si se tiene una dependencia hacia un paquete? En esencia, se ven todas las clases públicas del paquete y todos sus métodos públicos. Bajo el esquema de visibilidad de C++, esto puede causar un problema, debido a que tal vez se quiera tener una clase que contenga métodos que pueden ser vistos por otros objetos dentro del mismo paquete, pero no por objetos que pertenezcan a otros paquetes.

Ésta es la razón por la cual Java tiene la visibilidad de paquete. Por supuesto, esto le simplifica las cosas a Java. Dentro de C++ se pueden marcar clases y operaciones con visibilidad de paquetes. Aun cuando esta convención no es aplicada por el compilador, sigue siendo útil para el diseño.

Una técnica eficaz aquí es reducir aún más la interfaz del paquete exportando sólo un pequeño subconjunto de las operaciones asociadas con las clases del paquete. Esto se puede hacer dando a todas las clases visibilidad de paquete, de tal modo que sólo puedan ser vistas por otras clases del mismo paquete y añadiendo otras clases públicas para el comportamiento público. Estas clases adicionales, llamadas **Fachadas**(*Facades*) delegan operaciones públicas a sus compañeras más tímidas dentro del paquete.

Los paquetes no dan respuestas sobre la manera de reducir las dependencias en el sistema, pero sí ayudan a ver cuáles son las dependencias, y sólo se puede efectuar el trabajo para reducidas, cuando es posible verlas. Los diagramas de paquetes son, desde mi punto de vista, una herramienta clave para mantener el control sobre la estructura global de un sistema.

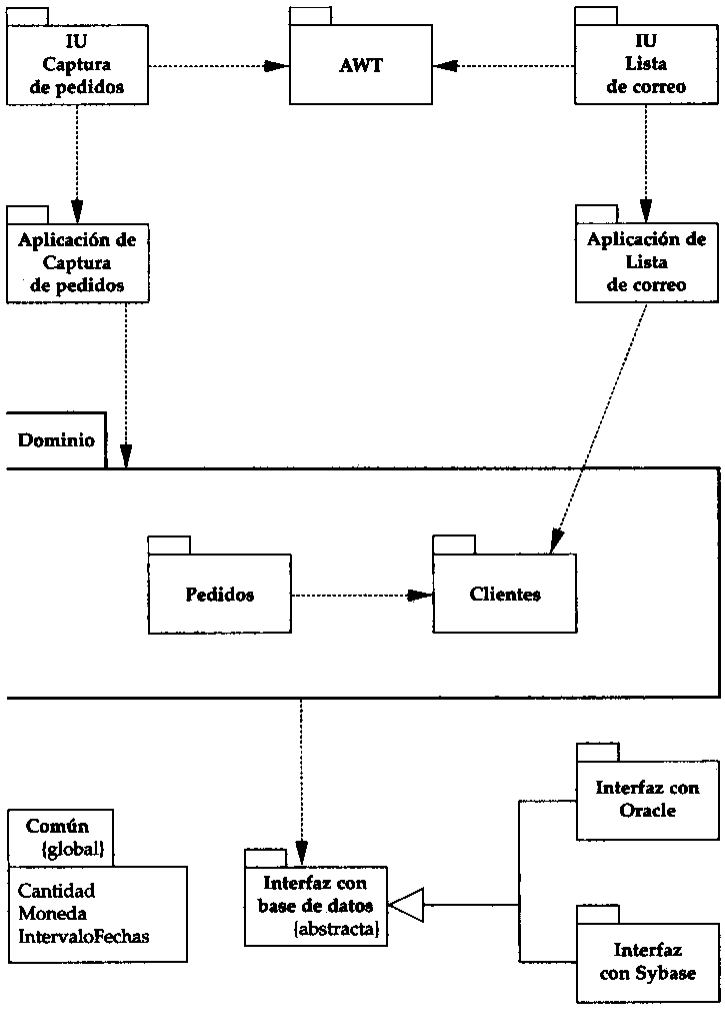

Figura 7-2: *Diagrama de paquetes avanzado*

La Figura 7-2 es un diagrama de paquetes más complejo que contiene artificios adicionales.

En primer lugar, vemos que he añadido un paquete de Dominio que contiene paquetes de pedidos y clientes. Esto es de utilidad, pues significa que puedo trazar dependencias desde y hacia el paquete general, en lugar de trazar muchas dependencias separadas.

Cuando se muestra el contenido de un paquete, se pone el nombre del paquete en una "etiqueta" y el contenido dentro del cuadro principal. Estos contenidos pueden ser una lista de clases (tal y como sucede en el paquete Común), otro diagrama de paquetes (como en Dominio) o un diagrama de clases (que no se muestra, pero el concepto ya debe ser obvio).

La mayor parte de las veces, considero suficiente listar las clases clave, pero en algunas ocasiones es útil otro diagrama. En el presente caso he mostrado que, mientras la aplicación Captura de órdenes tiene una dependencia con todo el paquete Dominio, la aplicación Lista de correo sólo depende del paquete Clientes.

Estrictamente hablando, el mero listado de clases no es UML puro (se deben mostrar los iconos de clase), pero éste constituye una de las áreas en que me inclinaría a las reglas.

La Figura 7-2 muestra el paquete Común marcado como {global}. Esto significa que todos los paquetes del sistema tienen una dependencia hacia él. Por supuesto, se debe emplear este artificio con moderación, pero las clases comunes (como Dinero) se emplean en todas partes.

Con los paquetes se puede aplicar la generalización. Esto significa que el paquete específico debe conformarse a la interfaz del paquete general. Esto es comparable con la perspectiva de especificación de la subtipificación en los diagramas de clases (véase el capítulo 4). Por tanto, de acuerdo con la Figura 7-2 , el agente de la base de datos puede usar la Interfaz con Oracle o la Interfaz con Sybase. Cuando se usa la generalización de este modo, el paquete general puede marcarse como {abstracto} para mostrar que sólo define una interfaz implementada por un paquete más específico.

La generalización implica una dependencia del subtipo al supertipo (no se necesita mostrar la dependencia extra; basta con la generalización misma). El poner las clases abstractas en un paquete de supertipo es una buena forma de romper ciclos en la estructura de dependencias. En tal situación, los paquetes de interfaz con la base de datos son los responsables de cargar y guardar los objetos de dominio en una base de datos. Por lo tanto, necesitan saber sobre los objetos del dominio. Los objetos del dominio, sin embargo, deben disparar las operaciones de carga y guardado.

La generalización nos permite poner la interfaz disparadora necesaria (varias operaciones de carga y guardado) en el paquete de interfaz con la base de datos. Estas operaciones, posteriormente, son implementadas por clases dentro de los paquetes de subtipo. No necesitamos una dependencia entre el paquete de interfaz con la base de datos y el paquete de interfaz con Oracle, ya que al momento de ejecución será en realidad el paquete de subtipo el que va a ser llamado por el dominio. Pero el dominio sólo piensa que está tratando con el paquete de interfaz (más simple) con la base de datos. El polimorfismo es tan útil para los paquetes como lo es para las clases.

Como regla general, es buena idea quitar los ciclos de la estructura de dependencias. No estoy convencido de que se puedan quitar todos los ciclos, pero ciertamente se deberán minimizar. Si se tienen, habrá que tratar de contenerlos dentro de un paquete contenedor más grande. En la práctica he encontrado casos en que no me ha sido posible evitar ciclos entre paquetes de dominio, pero de todas maneras trato de eliminarlos de las interacciones entre el dominio y las interfaces externas. La generalización de paquetes es un elemento clave para hacerlo.

En un sistema ya existente, las dependencias se pueden deducir observando las clases. Ésta es una tarea muy útil que puede ser llevada a cabo por una herramienta. Encuentro esto muy útil, si trato de mejorar la estructura de un sistema ya existente. Un paso útil inicial es dividir las clases en paquetes y analizar las dependencias entre estos últimos. Después realizo un reordenamiento de factores para reducir las dependencias.

Cuándo utilizar los diagramas de paquetes

Los paquetes son una herramienta vital para los proyectos grandes. Úselos siempre que un diagrama de clases que abarque todo el sistema ya no sea legible en una hoja de papel tamaño carta (o A4).

Deberá mantener sus dependencias al mínimo, ya que ello reduce el acoplamiento. Sin embargo, la heurística de esto no está bien comprendida.

Los paquetes son especialmente útiles para pruebas. Aunque yo escribo algunas pruebas para verificar clase por clase, prefiero hacer mis pruebas unitarias en el nivel de paquete por paquete. Cada paquete deberá tener una o más clases de pruebas que verifiquen su comportamiento.

Para mayor información

La fuente original de paquetes era Grady Booch (1994); los llamaba categorías de clase. Su análisis, sin embargo, era muy breve. El mejor estudio que conozco sobre este tema es el de Robert Martin (1995), cuyo libro da varios ejemplos de utilización de Booch y C++, prestando gran atención a la minimización de las dependencias. También puede encontrarse información valiosa en Wirfs-Brock (1990), autora que se refiere a los paquetes como subsistemas.

DIAGRAMAS DE ESTADOS

Resumen

Los **diagramas de estados** son una técnica conocida para describir el comportamiento de un sistema. Describen todos los estados posibles en los que puede entrar un objeto particular y la manera en que cambia el estado del objeto, como resultado de los eventos que llegan a él. En la mayor parte de las técnicas OO, los diagramas de estados se dibujan para una sola clase, mostrando el comportamiento de un solo objeto durante todo su ciclo de vida.

Índice de Contenidos

Introducción

Los **diagramas de estados** son una técnica conocida para describir el comportamiento de un sistema. Describen todos los estados posibles en los que puede entrar un objeto particular y la manera en que cambia el estado del objeto, como resultado de los eventos que llegan a él. En la mayor parte de las técnicas OO, los diagramas de estados se dibujan para una sola clase, mostrando el comportamiento de un solo objeto durante todo su ciclo de vida.

Existen muchas formas de diagramas de estados, cada una con semántica ligeramente diferente. La más popular que se emplea en las técnicas de OO se basa en la tabla de estados de David Harel (Vol. 8). OMT fue quien la usó por primera vez para los métodos de OO y fue adoptada por Grady Booch en su segunda edición (1994).

La Figura 8-1 muestra un diagrama de estados de UML para un pedido del sistema de proceso de pedidos que presenté antes. El diagrama indica los diversos estados de un pedido.

Comenzamos en el punto de partida y mostramos una transición inicial al estado de Comprobación. Esta transición está etiquetada como "/obtener el primer artículo". La sintaxis de una etiqueta de transición tiene tres

partes, las cuales son optativas: *Evento {Guard Guardia} / Acción*. En este caso, sólo tenemos la acción "obtiene primer artículo." Una vez realizada tal acción, entramos al estado de Comprobación. Este estado tiene una actividad asociada con él, la cual se indica mediante una etiqueta con la sintaxis*hace/actividad*. En este caso, la actividad se llama" comprueba artículo".

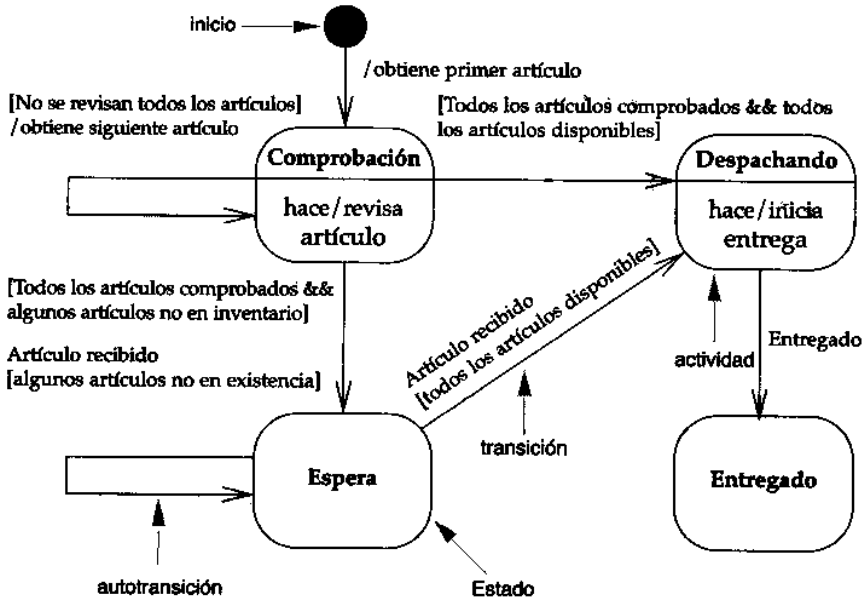

Figura 8-1: *Diagrama de estados*

Nótese que utilizo los términos "acción" para indicar la transición, y "actividad" para indicar el estado. Aunque ambos son procesos, implementados característicamente mediante algún método sobre Pedido, se tratan de manera diferente. Las **acciones** se asocian con las transiciones y se consideran como procesos que suceden con rapidez y no se pueden interrumpir. Las **actividades** se asocian con los estados y pueden tardar más. Una actividad puede ser interrumpida por algún evento.

Adviértase que la definición de "rápidamente" depende del tipo de sistema que se está produciendo. En un sistema de tiempo real, "Rápidamente" puede significar unas pocas instrucciones de código máquina; en los sistemas de información normales, "rápidamente" puede significar menos de unos cuantos segundos.

Cuando una transición no tiene evento alguno en su etiqueta, significa que la transición se da tan pronto como se completa cualquier actividad asociada con el estado dado. En este caso, ello significa tan pronto se termine la Comprobación. Del estado Comprobación se derivan tres transiciones. Las tres sólo tienen guardias en su etiqueta. Un **guardia** es una condición lógica que sólo devuelve "verdadero" o "falso." Una transición de guardia ocurre sólo si el guardia se resuelve como "verdadero".

Sólo se puede tomar una transición de un estado dado, por lo que tratamos de que los guardias sean mutuamente excluyentes para cualquier evento. En la Figura 8-1abarcamos tres condiciones:

1. Si no hemos comprobado todos los artículos, tomamos el siguiente artículo y regresamos al estado de Comprobación para comprobado.
2. Si hemos comprobado todos los artículos y todos están en existencia, hacemos la transición al estado de Despachando.
3. Si hemos comprobado todos los artículos pero no todos están en existencia, hacemos la transición al estado Espera.

Veamos, en primer lugar, el estado Espera. Como no hay actividades para este estado, el pedido se detiene en él esperando un evento. Ambas transiciones del estado Espera se etiquetan con el evento Artículo recibido. Esto significa que el pedido espera hasta que detecta este evento. Llegado ese momento, evalúa los guardias de las transiciones y hace la transición apropiada (ya sea a Despachando o de vuelta a Espera).

Dentro del estado Despachando, tenemos actividad que inicia una entrega. También hay una transición simple no guardada, disparada por el evento Entregado. Esto indica que la transición ocurrirá siempre que tenga lugar el evento. Sin embargo, nótese que la transición no sucede cuando termina la actividad; por el contrario, una vez terminada la actividad "iniciar entrega", el pedido se mantiene en el estado Despachando, hasta que ocurre el evento Entregado.

La última cuestión a considerar es la transición denominada" cancelado". Queremos tener la posibilidad de cancelar un pedido en cualquier momento, antes de que sea entregado. Podríamos hacerlo dibujando transiciones separadas desde cada uno de los estados, Comprobación, Espera y Despacho. Una alternativa útil es crear un superestado con los tres estados, y después dibujar una transición simple, a partir de él. Los subestados simplemente heredan todas las transiciones sobre el superestado.

Las Figura 8-2 y 8-3 muestran cómo estos enfoques reflejan el mismo comportamiento del sistema. La Figura 8-2 aparece más bien cargada, a pesar de que sólo tiene tres transiciones duplicadas. La Figura 8-3, en cambio, da un cuadro más claro y, si se necesitan posteriormente los cambios, será más difícil olvidar el evento cancelado.

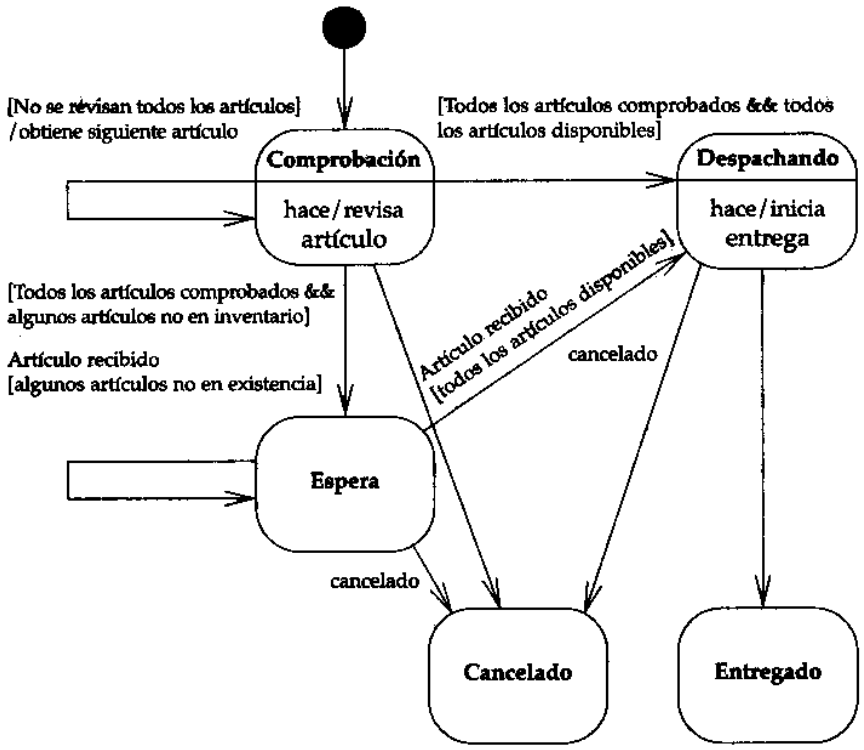

Figura 8-2: *Diagrama de estados sin superestados*

Figura 8-2: Diagrama de estados sin superestados

En los ejemplos actuales, he mostrado una actividad dentro de un estado, indicándola con texto en la forma de *hace/actividad*. También se pueden indicar otras cosas en un estado.

Si un estado responde a un evento con una acción que no produzca una transición, dicha condición se puede mostrar poniendo un texto de la forma*nombreEvento/nombreAcción* en el cuadro de estado.

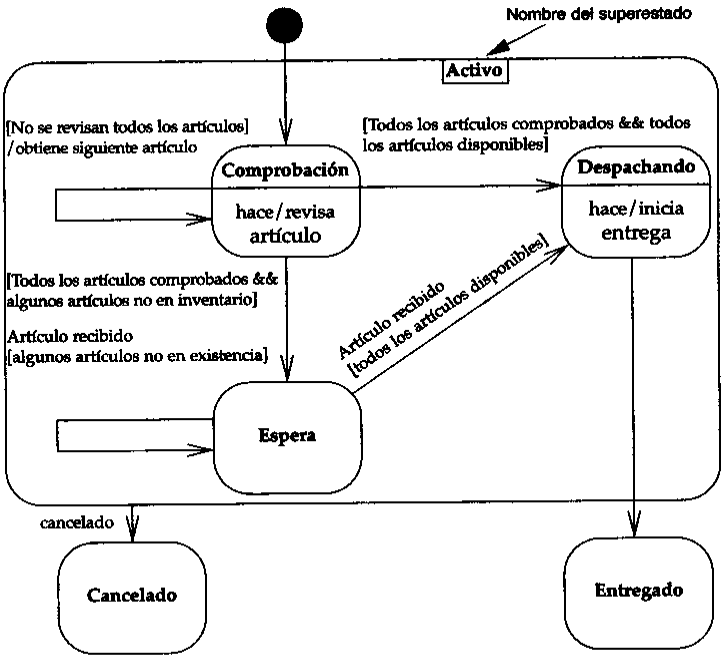

Figura 8-3: *Diagrama de estados con superestados*

Figura 8-3: Diagrama de estados con superestados

Existen también dos eventos especiales, entrada y salida. Cualquier acción que esté marcada como vinculada al evento **entrada** se ejecuta siempre que se entre al estado dado a través de una transición. La acción asociada con el evento **salida** se ejecuta siempre que se sale del estado por medio de una transición. Si se tiene una transición que vuelve al mismo estado (a esto se le llama **autotransición**) por medio de una acción, se ejecuta primero la acción de salida, luego la acción de transición y, por último, la acción de entrada. Si el estado tiene también una actividad asociada, ésta se ejecuta tras la acción de entrada.

Diagramas de estados concurrentes

Además de los estados de un pedido que se basan en la disponibilidad de los artículos, también existen estados que se basan en la autorización de pagos. Si vemos estos estados, podríamos ver un diagrama de estados como el de la figura 8-4.

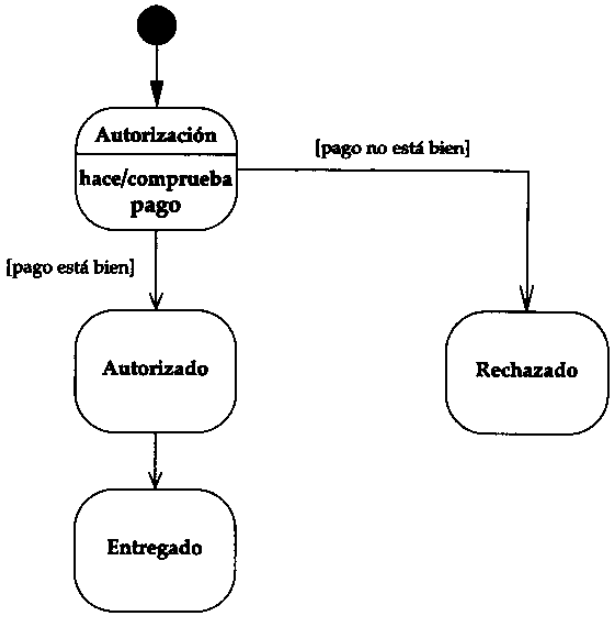

Figura 8-4: *Autorización de pagos*

Aquí comenzamos efectuando una autorización. La actividad de "comprobación de pago" termina señalando que el pago fue aprobado. Si el pago está bien, el pedido espera en el estado Autorizado hasta que sucede el evento de "entrega". De otro modo, el pedido entra al estado de Rechazado.

El objeto Orden presenta una combinación de los comportamientos que se muestran en las Figura 8-1 y 8-2. Los estados asociados y el estado Cancelado mencionados anteriormente pueden combinarse en un **diagrama de estados concurrentes**

(Véase la Figura 8-5).

Nótese que en la figura 8-5 he dejado fuera los detalles de los estados internos.

Las secciones concurrentes del diagrama de estados son lugares en los que, en cualquier punto, el pedido está en dos estados diferentes, uno por cada diagrama. Cuando el pedido deja los estados concurrentes, se encuentra en un solo estado. Se puede ver que un pedido se inicia tanto en el estado Comprobando como en el estado Autorizando. Si la actividad de "comprobación de pago" del estado Autorizando se completa inicialmente de modo exitoso, entonces el pedido estará en los estados Comprobando y Autorizado. Si sucede el evento" cancelar", entonces el pedido sólo estará en el estado Cancelado.

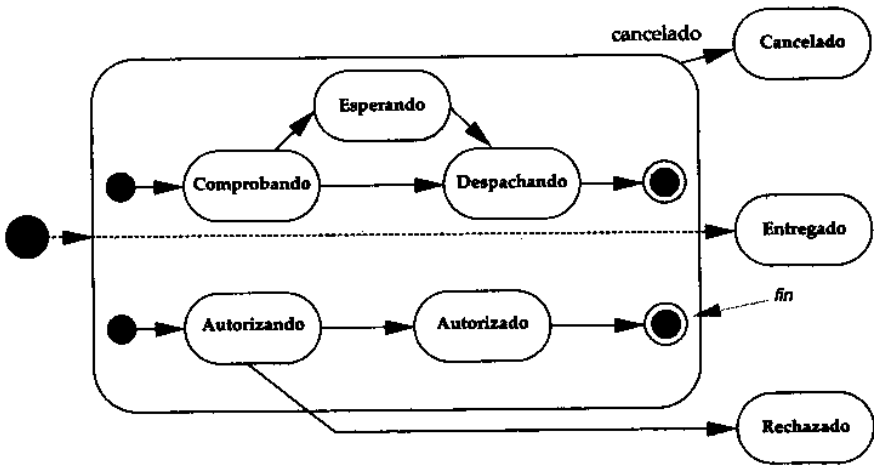

Figura 8-5: *Diagrama de estados concurrentes*

Los diagramas de estados concurrentes son útiles cuando un objeto dado tiene conjuntos de comportamientos independientes. Nótese, sin embargo, que no se debe permitir que sucedan demasiados conjuntos de comportamientos concurrentes en un solo objeto. Si se tienen varios diagramas de estados concurrentes complicados para un solo objeto, se deberá considerar la división del objeto en varios.

Cuándo utilizar los diagramas de estados

Los diagramas de estados son buenos para describir el comportamiento de un objeto a través de varios casos de uso. No son tan buenos para describir un comportamiento que involucra cierto número de objetos que colaboran entre ellos. Así pues, es útil combinar los diagramas de estados con otras técnicas. Por ejemplo, los diagramas de interacción(véase el capítulo 6) son buenos para la descripción del comportamiento de varios objetos en un mismo caso de uso. Por su parte, los diagramas de actividades (véase el capítulo 9) son buenos para mostrar la secuencia general de las acciones de varios objetos y casos de uso.

Hay quienes consideran que los diagramas de estado son naturales, pero muchos no los consideran así. Preste atención al modo en que los emplean quienes trabajan con ellos; podría ocurrir que su equipo no considere útiles los diagramas de estados, debido a su modo de trabajar. Esto no sería un gran problema; como siempre, deben combinarse las técnicas que sean de utilidad.

Si decide utilizar diagramas de estados, no trate de dibujar uno por cada clase del sistema. Aunque éste es el enfoque que emplean los detallistas ceremoniosos, casi siempre es un esfuerzo inútil. Utilice los diagramas de estados sólo para aquellas clases que presenten un comportamiento interesante, cuando la construcción de tales diagramas le ayude a comprender lo que sucede. Muchos consideran que los objetos de interfaz de usuario (IU) y de control tienen el tipo de comportamiento que es útil describir mediante diagramas de estados.

Para mayor información

Tanto Grady Booch (1994) como Jim Rumbaugh (1991) tienen material sobre los diagramas de estados, aunque no contiene mucha mayor información de la que hemos mencionado en este capítulo. Cook y Daniels (1994) son quienes han abordado con mayor detalle los diagramas de estados; yo recomiendo ampliamente su libro, si usted emplea con frecuencia los diagramas de estados. La semántica que definen es mucho más detallada que la de cualquier otro libro. Aunque tal semántica no corresponda enteramente con la del UML, los autores tratan de forma exhaustiva cuestiones que usted debe tener en cuenta, si va usar diagramas de estados.

DIAGRAMAS DE ACTIVIDADES

Resumen

Los diagramas de actividades son particularmente útiles para la descripción del comportamiento que tiene una gran cantidad de proceso paralelo. Permite seleccionar el orden en que se harán las cosas. Indica las reglas esenciales de secuenciación que tengo que seguir. Ésta es la diferencia clave entre un diagrama de actividades y un diagrama de flujo. Los diagramas de flujo se limitan normalmente a procesos secuenciales; los diagramas de actividades pueden manejar procesos paralelos. Los diagramas de actividades también son útiles para los programas concurrentes, ya que se pueden plantear gráficamente cuáles son los hilos y cuándo necesitan sincronizarse

Los diagramas de clases nos muestran el cuadro completo de clases interconectadas y los diagramas de actividades hacen lo mismo en lo que respecta al comportamiento

Índice de Contenidos

Introducción

Los diagramas de actividades constituyen una de las partes más imprevistas del UML.

El **diagrama de actividades** combina ideas de varias técnicas: el diagrama de eventos de Jim Odell, las técnicas de modelado de estados de SDL y las redes de Petri Estos diagramas son particularmente útiles en conexión con el flujo de trabajo y para la descripción del comportamiento que tiene una gran cantidad de proceso paralelo.

En la Figura 9-1, que proviene de la documentación del UML 1.0, el símbolo clave es la**actividad.** La interpretación de este término depende de la perspectiva desde la cual se dibuja el diagrama. En un diagrama conceptual, una actividad es cierta tarea que debe ser llevada a cabo, ya sea por un ser humano o por una computadora. En un diagrama de perspectiva de especificación o de perspectiva de implementación, una actividad es un método sobre una clase.

Cada actividad puede ser seguida por otra actividad. Esto simplemente es secuenciación. Por ejemplo, en la Figura 9-1, la actividad Poner el café en el filtro va seguida por la actividad Poner el filtro en la máquina. Hasta ahora, el diagrama de actividades parece un *flow-chart*> diagrama de flujo. Podemos investigar las diferencias observando la actividad Encuentra bebida.

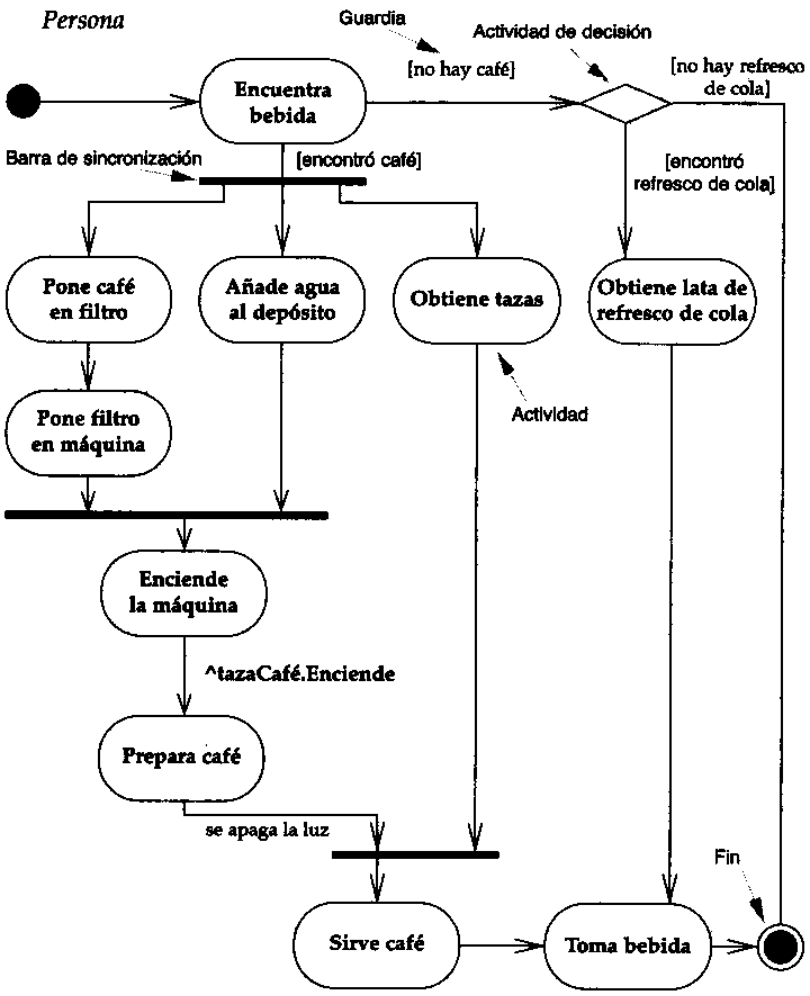

Figura 9-1: *Diagrama de actividades*

De Encuentra bebida salen dos disparadores. Cada disparador tiene un **guardia,** una expresión lógica que se evalúa como "verdadero" o "falso", del mismo modo que en undiagrama de estados. En el caso de la Figura 9-1, la persona seguirá la actividad Encuentra bebida considerando las opciones de café o refresco de cola.

Supongamos que podremos disfrutar de una taza de café de Colombia y seguiremos la ruta del café. Este disparador nos conduce a la **barra de sincronización,** a la cual están unidos otros tres disparadores que, a su vez, conducen a las actividades Pone café en filtro, Añade agua al depósito y Obtiene tazas.

El diagrama señala que estas actividades pueden suceder en paralelo, lo cual significa, en esencia, que su orden no es significativo. Se podría poner el café en el filtro primero, después añadir agua al depósito y, por último, obtener las tazas. También podríamos conseguir las tazas, después poner el café en el filtro... ya se entiende la idea.

También puedo llevar a cabo estas actividades en forma intercalada. Podría obtener una taza, añadir después un poco de agua al depósito, obtener otra taza, agregar otro poco más de agua y así sucesivamente. O también podría llevar a cabo estas operaciones simultáneamente: agregar el agua al depósito con una mano y con la otra alcanzar la taza. De acuerdo con el diagrama, cualquiera de estas formas de operar es correcta.

El diagrama de actividades me permite seleccionar el orden en que se harán las cosas. Esto es, simplemente me dice las reglas esenciales de secuenciación que tengo que seguir. Ésta es la diferencia clave entre un diagrama de actividades y un diagrama de flujo. Los diagramas de flujo se limitan normalmente a procesos secuenciales; los diagramas de actividades pueden manejar procesos paralelos.

Esta característica es importante para el modelado de negocios. Los negocios con frecuencia tienen procesos secuenciales innecesarios. Una técnica como ésta, que promueve el comportamiento paralelo, es valiosa en estas situaciones, porque auspicia que las personas se aparten de las secuencias innecesarias en su comportamiento y descubran oportunidades para hacer cosas en paralelo. Esto puede mejorar la eficiencia y capacidad de respuesta de los procesos del negocio.

Los diagramas de actividades también son útiles para los programas concurrentes, ya que se pueden plantear gráficamente cuáles son los hilos y cuándo necesitan sincronizarse.

Cuando se tiene un comportamiento paralelo, se impone la necesidad de sincronizar. No queremos prender la cafetera hasta haberle colocado el filtro y llenado de agua el depósito. Por eso, vemos a los disparadores de estas actividades saliendo juntos de una barra sincronizadora. Una barra de sincronización simple como ésta indica que el disparo de salida ocurre sólo cuando han sucedido ambos disparos de entrada. Como veremos después, estas barras pueden ser más complicadas.

Más adelante se dará otra sincronización: el café debe ser preparado y las tazas deben estar disponibles, antes de poder servir el café.

Vayamos ahora al otro carril.

En este caso, tenemos una decisión compuesta. La primera decisión es sobre el café. Esta decisión es la que determina los dos disparadores que salen de Encuentra bebida. Si no hay café, nos enfrentamos a una segunda decisión, basada ésta en refresco de cola.

Cuando tenemos decisiones como la anterior, señalamos la segunda decisión con un rombo de decisión. Esto nos permite describir decisiones anidadas, de las cuales podemos tener cualquier cantidad.

En la actividad Toma bebida convergen dos disparadores, lo que significa que se llevará a cabo en cualquiera de los dos casos. Por el momento, se puede considerar esto como un caso OR (lo hago, si sucede uno u otro disparador) y a la barra de sincronización como el caso AND (lo hago, si suceden ambos disparadores).

Diagramas de actividades para casos de uso

La Figura 9-1 describe un método sobre el tipo Persona. Los diagramas de actividades son útiles para describir métodos complicados. También pueden servir para otras cosas; por ejemplo, para describir un caso de uso .

Considérese un caso de uso para el proceso de pedidos.

> *Cuando recibimos un pedido, comprobamos cada artículo de línea del pedido para ver si lo hay en existencia. Si la respuesta es afirmativa, asignamos la mercancía al pedido. Si esta asignación hace bajar la cantidad de mercancía en existencia por debajo del nivel de reposición (reorden), se repone. Mientras hacemos esto, revisamos el pago para ver si está correcto. Si el pago está bien y hay mercancías en existencia, despachamos el pedido. Si el pago está correcto pero no hay las mercancías en existencia, dejamos en espera el pedido. Si el pago no está bien, cancelamos la orden.*

Véase la Figura 9-2 para una representación visual de este caso de uso.

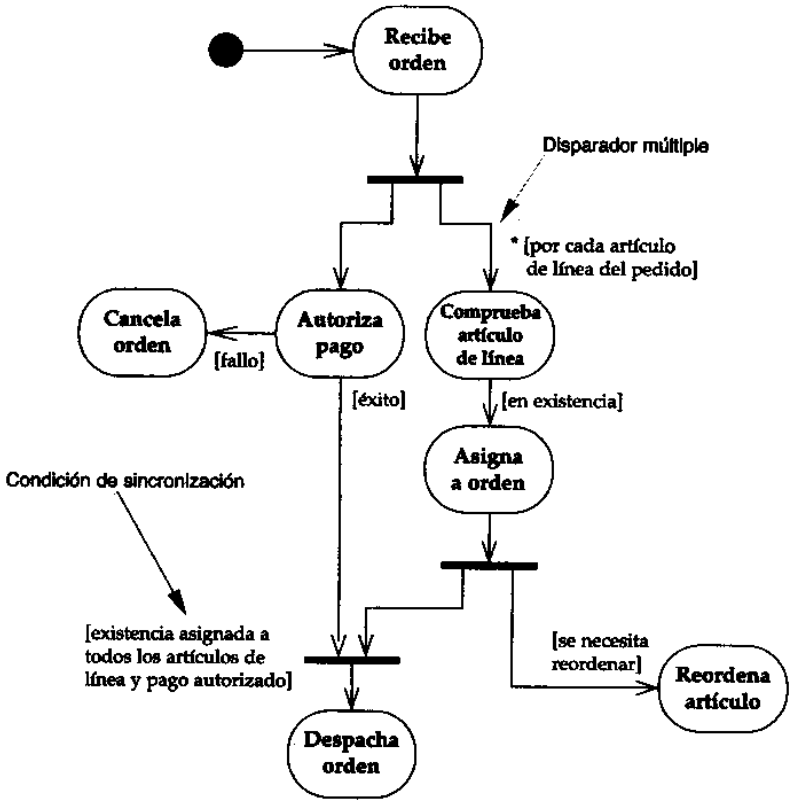

Figura 9-2: *Recepción de un pedido*

La figura introduce un nuevo artificio al diagrama de actividades. Véase el disparador de entrada asociado con la actividad Comprueba artículo de línea. Está marcado con un [*]. Éste es un marcador de multiplicidad (se trata del mismo marcador utilizado en losdiagramas de clases; que muestra que, cuando recibimos una orden, tenemos que llevar a cabo la actividad Comprueba artículo de línea para cada artículo de línea del pedido. Esto significa que la actividad Recibe orden va seguida por una invocación de la actividad Autoriza pago y varias invocaciones de la actividad Comprueba artículo de línea. Todas estas invocaciones suceden en paralelo.

Esto resalta la segunda fuente del paralelismo en un diagrama de actividades. Se pueden tener actividades paralelas a través de varias transiciones que salen de una barra sincronizadora; también se pueden tener actividades paralelas cuando se dispara la misma actividad por medio de un **disparador múltiple.** Siempre que se tenga un disparador múltiple, habrá de indicarse en el diagrama cuál es su base, como sucede con [por cada artículo de línea de pedido].

Cuando encuentre un disparador múltiple, generalmente verá una barra de sincronización más abajo del diagrama, que une los hilos paralelos. En este caso, vemos la barra antes de la actividad Despacha Orden. La barra de sincronización tiene aplicada a ella una condición. Cada vez que llega un disparador a la barra de sincronización, se prueba la condición. Si la condición es verdadera, sucede el disparo de salida (también se puede marcar otro [*] para indicar la unión de las líneas; yo prefiero no mostrar un segundo [*], pues vuelve muy confuso el diagrama, y considero que la condición de sincronización deja las cosas lo bastante claras).

Las barras de sincronización no etiquetadas operan de la misma forma. La ausencia de una condición significa que se está usando la condición predeterminada para las barras de sincronización. La condición predeterminada es que todos los disparadores ya han ocurrido. Por esa razón, no hay condiciones en las barras de la Figura 9-1

Un diagrama de actividades no tiene que tener un punto de terminación definido. El punto de terminación de un diagrama de actividades es el punto en el cual todas las actividades disparadas han sido operadas y no hay nada más por hacer. En la Figura 9-2, no tendría utilidad marcar un punto de terminación explícito.

La Figura 9-2 tiene también un callejón sin salida: la actividad Reordena artículo. Nada sucede después de realizada esta actividad. Estos callejones sin salida están bien en diagramas de actividades sin terminación, como éste. Algunas veces son obvios, como en este caso. En otras ocasiones no lo son tanto. Obsérvese la actividad Comprobación de artículo de línea; sólo tiene un disparador de salida, el cual tiene una condición. ¿Qué sucede si no hay existencia del artículo de línea? Nada, el hilo del proceso simplemente se detiene allí.

En nuestro ejemplo, no podemos despachar un pedido hasta recibir una orden que reabastezca la existencia. Éste podría ser un caso de uso separado.

> *Cuando llega un reabastecimiento, vemos los pedidos sobresalientes y decidimos cuáles podemos surtir con el material recibido y, entonces, asignamos lo correspondiente a sus respectivos pedidos. Con esto se podrían liberar dichas órdenes para ser despachadas. La mercancía restante la ponemos en el almacén.*

La Figura 9-3 es un diagrama de actividades que representa este caso de uso.

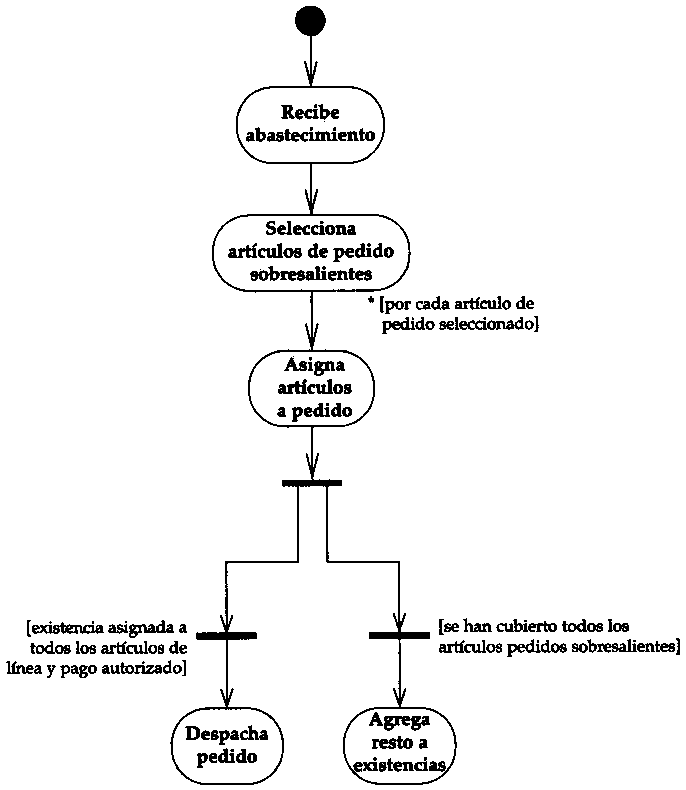

Figura 9-3: *Recepción de abastecimiento*

Este segundo caso de uso muestra cómo la orden puede quedar en espera de ser despachada hasta que suceda el reabastecimiento.

Cuando cada uno de los dos casos muestra parte del cuadro general, encuentro que es útil dibujar un diagrama combinado, como el de la Figura 9-4.

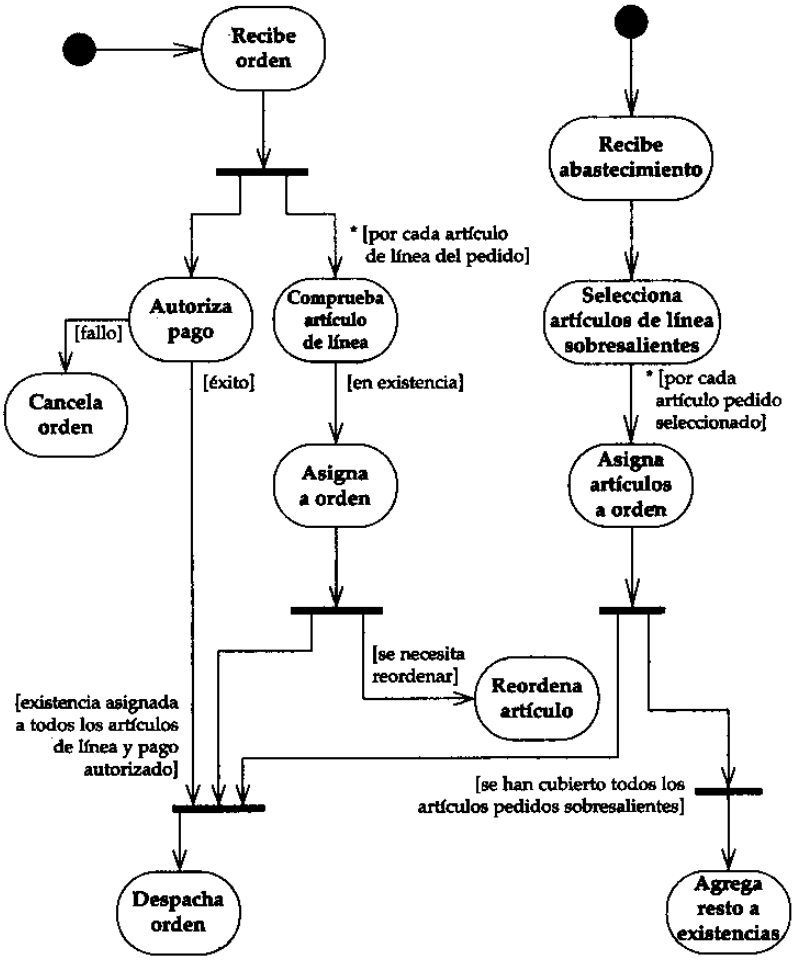

Figura 9-4: *Recibe orden y recibe existencias*

Este diagrama muestra los diagramas de actividades para los dos casos de uso superpuestos, de modo que se pueden ver cómo las acciones en un caso de uso afectan las acciones del otro. Un diagrama de actividades como éste tiene múltiples puntos de partida, lo cual está muy bien, pues el diagrama de actividades representa la manera en que el negocio reacciona ante varios eventos externos.

Considero particularmente útil esta capacidad de los diagramas de actividades de mostrar comportamientos que abarcan varios casos de uso. Los casos de uso nos dan rebanadas de información sobre un dominio visto desde afuera; cuando vemos el cuadro interno, necesitamos ver el conjunto. Los diagramas de clases; nos muestran el cuadro completo de clases interconectadas y los diagramas de actividades hacen lo mismo en lo que respecta al comportamiento.

Carriles

Los diagramas de actividades dicen qué sucede, pero no lo que hace cada quién. En la programación, esto significa que el diagrama no especifica qué clase es responsable de cada actividad.

En el modelado del dominio esto significa que el diagrama no indica cuáles son las personas o departamentos responsables de cada actividad. Una manera de superar esto es etiquetando cada actividad con la clase o la

persona responsable. Este procedimiento funciona, pero no ofrece la misma claridad que los diagramas de interacción en lo que se refiere a mostrar la comunicación entre los objetos.

Los **carriles** (*swirnlanes*) son una forma de subsanar esta deficiencia.

Para usar los carriles, se deben disponer los diagramas de actividades en zonas verticales separadas por líneas. Cada zona representa la responsabilidad de una clase en particular o, como en el caso de la Figura 9-5, de un departamento particular.

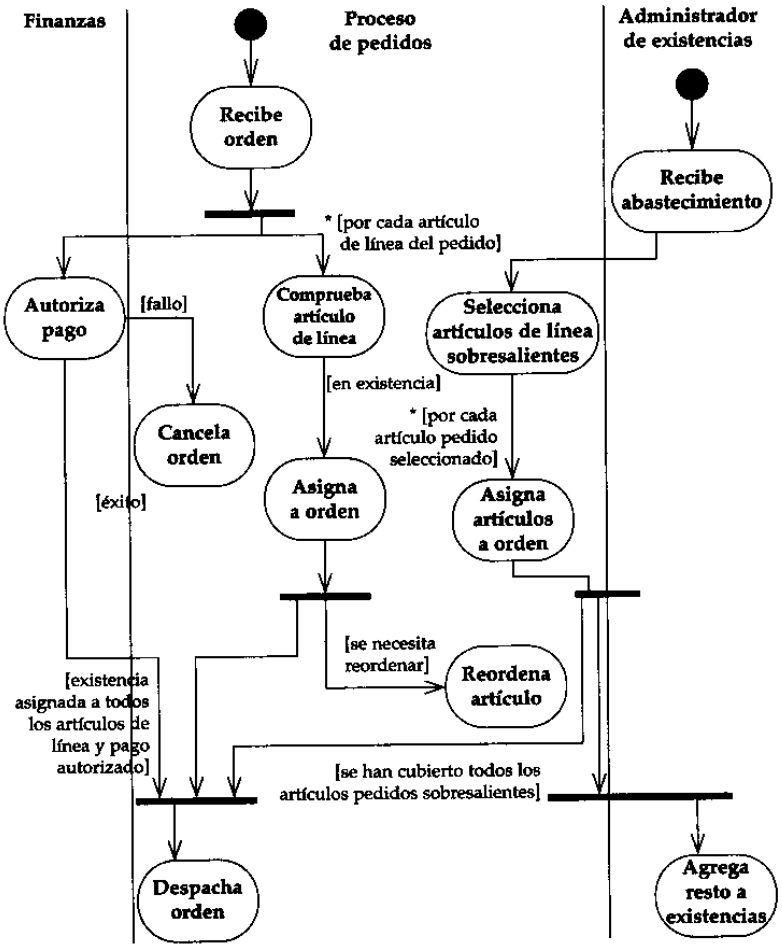

Figura 9-5: *Carriles*

Los carriles son buenos en el sentido de que combinan la representación lógica del diagrama de actividades con la representación de responsabilidades del diagrama de interacción. Sin embargo, puede ser difícil dibujados en un diagrama complejo. En ocasiones he dibujado zonas no lineales, que son mejores que nada (a veces es necesario evitar el impulso de tratar de decir demasiado en un diagrama).

Algunos se aseguran de asignar actividades a objetos cuando dibujan un diagrama de actividades. A otros les basta trabajar primero con el diagrama de actividades, a fin de lograr una idea general del comportamiento, y después asignar las actividades a los objetos. He presenciado casos en los que quienes hacen inmediatamente las asignaciones reclaman, molestos, a quienes las posponen; les acusan de hacer diagramas de flujo de datos y de no estar orientados a objetos.

Confieso que en ocasiones dibujo un diagrama de actividades, asignando sólo hasta después el comportamiento a los objetos. Considera muy útil comprender una cosa a la vez. Esto es particularmente cierto cuando hago modelado de negocios y estoy animando a un experto del dominio a pensar en nuevas formas de hacer las cosas.

Para mí, esto funciona. Otros prefieren asignar inmediatamente el comportamiento a los objetos. Usted debe hacer lo que le parezca más cómodo. Lo importante es asignar las actividades a las clases, antes de terminar. Con frecuencia, recurro a un diagrama de interacción .

Descomposición de una actividad

Una actividad puede ser descompuesta en una descripción posterior. Esta descripción puede ser un texto, un código u otro diagrama de actividades (véase la Figura 9-6.)

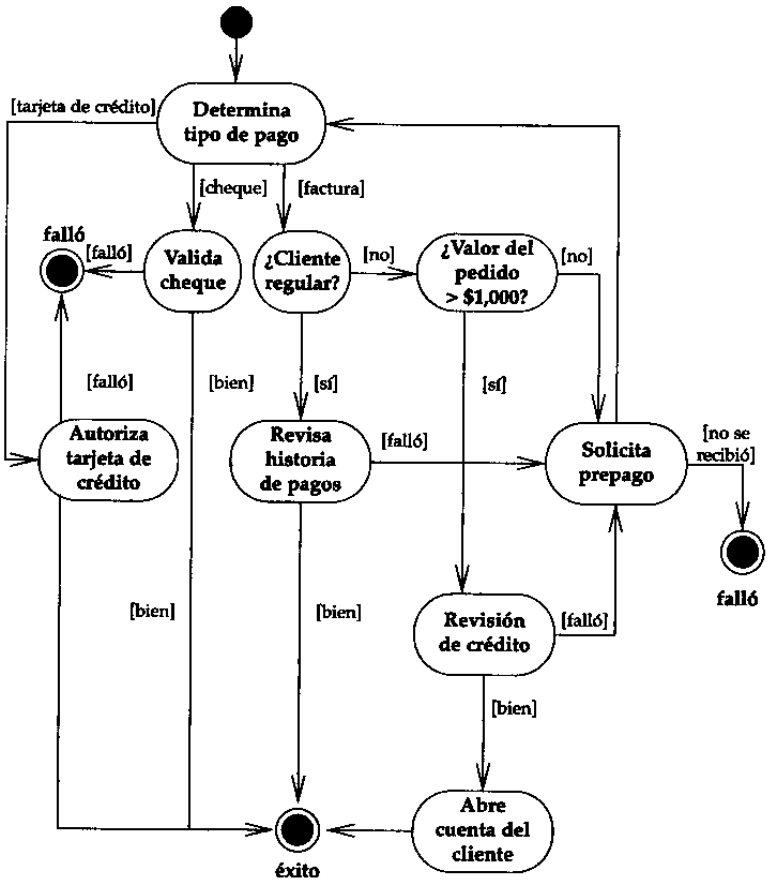

Figura 9-6: *Diagrama descompuesto de actividades*

Cuando se dibuja un diagrama de actividades como una descomposición de una actividad de más alto nivel, se debe proporcionar un solo punto de partida. Sin embargo, se pueden suministrar tantos puntos de terminación como disparadores de salida haya dentro de la actividad de alto nivel. Esto permite al diagrama subsidiario devolver un valor que determina los posteriores disparos. Por ejemplo, la Figura 9-6 muestra que la actividad Autoriza tarjeta de crédito devuelve" correcto" o "no-aprobado".

La Figura 9-6 se dibujó desde una perspectiva conceptual, pero no es muy difícil imaginar lo mismo dibujado como un retrato gráfico del código de programación, tal como un diagrama de flujo. Tengo tendencia a evitar esto, por la misma razón por la que no dibujo diagramas de flujo. Con frecuencia, es más fácil simplemente escribir el código. Si considera que aquí son útiles los diagramas, podría probar este procedimiento, en particular si quiere mostrar varios hilos.

Si se tiene que representar mucha lógica, es fácil que los diagramas de actividades se compliquen demasiado. En esta situación, una tabla de verdad puede ser una mejor representación.

Cuándo utilizar diagramas de actividades

Como la mayor parte de las técnicas de modelado de comportamiento, los diagramas de actividades tienen virtudes y defectos definidas, por lo que lo mejor es utilizarlos en combinación con otras técnicas.

La gran virtud de los diagramas de actividades reside en que manejan y promueven el comportamiento en paralelo. Esta cualidad hace de ellos una excelente herramienta para el modelado de flujo de trabajo y, en principio, para la programación multihilos. Su gran desventaja es que no dejan muy claros los vínculos entre acciones y objetos.

Usted puede definir a qué se refiere una relación mediante el procedimiento de etiquetar una actividad con un nombre de objeto o bien por medio de carriles (que dividen un diagrama de actividades con base en responsabilidades), pero este recurso no tiene la sencilla inmediatez de los diagramas de interacción . Por esa razón, algunos consideran que los diagramas de actividades no están orientados a objetos y que, por tanto, son malos. He encontrado que esta técnica puede ser una herramienta de gran utilidad, y no tengo por norma desechar de entre mis herramientas de trabajo cualquiera que sea útil.

Me gusta emplear los diagramas de actividades en las siguientes situaciones:

- **En el análisis de un caso de uso**. En esta etapa, no me interesa asignar acciones a los objetos: sólo necesito comprender qué acciones deben ocurrir y cuáles son las dependencias de comportamiento. Asigno los métodos a los objetos posteriormente, y muestro tales asignaciones mediante un diagrama de interacción.
- **En la comprensión del flujo de trabajo, a través de numerosos casos de uso**. Cuando los casos de uso interactúan entre ellos, los diagramas de actividades son una gran herramienta para representar y entender este comportamiento. En las situaciones dominadas por el flujo de trabajo, los considero una herramienta formidable.
- **Cuando se trata de aplicaciones multihilos**. No he usado los diagramas de actividades para este propósito, pero son muy adecuados para ello.

Por otra parte, no uso los diagramas de actividades en las siguientes situaciones:

- **Para tratar de ver cómo colaboran los objetos**. Un diagrama de interacción es más simple y da un panorama más claro de las colaboraciones.
- Para tratar de ver **cómo se comporta un objeto durante su periodo de vida**. Utilice un diagrama de estados para ese fin.

Para mayor información

Los diagramas de actividades se basan en diversos enfoques orientados al flujo de trabajo. Su antecedente más inmediato es el diagrama de eventos de Jim Odell, sobre el que usted podrá encontrar más información en el libro de "fundamentos" de Martín y Odell (1994).

DIAGRAMAS DE EMPLAZAMIENTO

Resumen

El **diagrama de emplazamiento** (*deployment diagram*) es aquel que muestra las relaciones físicas entre los componentes de software y de hardware en el sistema entregado. Así, el diagrama de emplazamiento es un buen sitio para mostrar cómo se enrutan y se mueven los componentes y los objetos, dentro de un sistema distribuido.

Índice de Contenidos

Introducción

El **diagrama de emplazamiento** (*deployment diagram*) es aquel que muestra las relaciones físicas entre los componentes de software y de hardware en el sistema entregado. Así, el diagrama de emplazamiento es un buen sitio para mostrar cómo se enrutan y se mueven los componentes y los objetos, dentro de un sistema distribuido.

Cada **nodo** de un diagrama de emplazamiento representa alguna clase de unidad de cómputo; en la mayoría de los casos se trata de una pieza de hardware. El hardware puede ser un dispositivo o un sensor simple, o puede tratarse de un mainframe.

La Figura 10-01 muestra una PC conectada a un servidor UNIX por medio de TCP /IP. Las **conexiones** entre nodos muestran las rutas de comunicación a través de las cuales interactuará el sistema.

Los **componentes** en un diagrama de emplazamiento representan módulos físicos de código. En mi experiencia, corresponden exactamente a los paquetes de un diagrama de paquetes (véase el capítulo 7), de tal modo que el diagrama de emplazamiento muestra dónde se ejecuta cada paquete en el sistema.

Las **dependencias** entre los componentes deben ser las mismas que las dependencias de paquetes. Estas dependencias muestran cómo se comunican los componentes con otros componentes. La dirección de una dependencia dada indica el conocimiento en la comunicación.

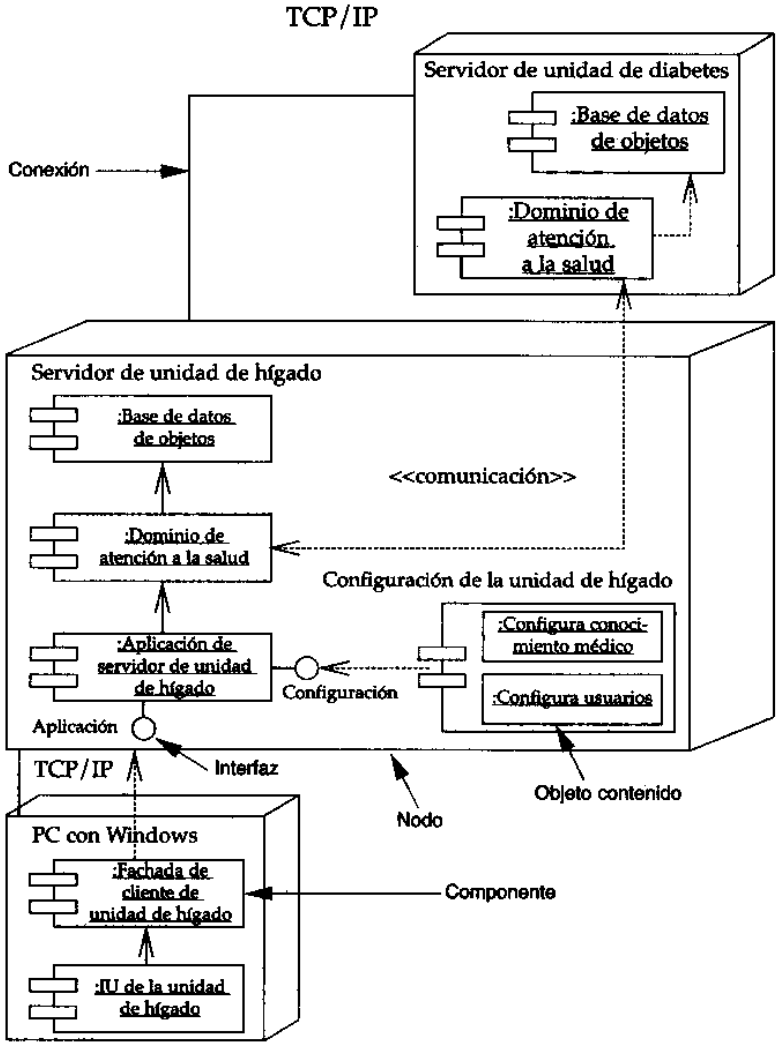

Figura 10-1: *Diagrama de emplazamiento*

Así, en el diagrama, la IU de la unidad de hígado depende de la Fachada de cliente de unidad de hígado, ya que llama a métodos específicos en la fachada. A pesar de que la comunicación es en ambas direcciones, en el sentido de que la Fachada devuelve datos, la Fachada no sabe quién la llama y, por tanto, no depende de la IU . En la comunicación entre ambos componentes del Dominio de atención a la salud, ambos saben que están hablando con otro componente de Dominio de atención a la salud, así que la dependencia de la comunicación es en dos sentidos.

Un componente puede tener más de una interfaz, en cuyo caso usted podrá ver cuáles componentes se comunican con cada interfaz. En la Figura 10-01, la PC contiene dos componentes: la IU y la fachada de la aplicación. La fachada de aplicación habla con la interfaz de la aplicación en el servidor. Un componente de configuración separado se ejecuta sólo en el servidor. La aplicación se comunica con su componente local del Dominio de atención a la salud, el cual, a su vez, puede comunicarse con otros componentes de Dominios de atención a la salud de la red.

La utilización de los componentes de diversos Dominios de atención a la salud está oculta para la aplicación. Cada componente del Dominio de atención a la salud tiene una base de datos local.

Cuando utilizar los diagramas de emplazamiento

En la práctica, no he visto que se use mucho este tipo de diagramas. La mayoría de la gente dibuja diagramas para mostrar este tipo de información, pero se trata de bocetos informales. En general, no tengo problemas con este tipo de diagramas, ya que cada sistema tiene sus propias características físicas que se querrán subrayar. A medida que se tiene que lidiar cada vez más con los sistemas distribuidos, estoy seguro de que se requerirá mayor formalidad, según se vaya entendiendo mejor cuáles son los asuntos que se deben resaltar en los diagramas de emplazamiento.

INFORMACIÓN COMPLEMENTARIA

REESTRUCTURACIÓN DE FACTORES

Resumen
Índice de Contenidos

Reestructuración de factores

¿Se ha topado alguna vez con el principio de la entropía de software? Este principio sugiere que los programas comienzan en un estado de buen diseño pero, a medida que se añade código para nuevas funciones, gradualmente van perdiendo su estructura, deformándose de tal modo que acaban como una masa de espagueti.

Parte de este hecho se debe a la escala. Se escribe un programa pequeño que hace bien un trabajo específico. Luego se pide mejorar el programa, por lo que se vuelve más complejo. Esto puede suceder incluso cuando se lleva el registro del diseño.

Una de las razones por las que surge la entropía en el software es que, cuando se añade una función al programa, se construye encima del programa existente, con frecuencia de una manera para la que no estaba diseñado. En tal situación, se puede rediseñar el programa existente para que maneje mejor los cambios, o bien adecuar esos cambios al código que se añada.

Aunque teóricamente es mejor rediseñar el programa, este procedimiento por lo general cuesta más trabajo, ya que cualquier reescritura de su programa generará nuevas fallas y problemas. Recuerde el viejo adagio de la ingeniería: "¡si no está descompuesto, no lo arregle!". Sin embargo, si no rediseña su programa, las adiciones serán más complejas de lo que deberían.

A la postre, esta complejidad extra tendrá un alto costo. Por tanto, existe un intercambio de una cosa por otra: el rediseño provoca molestias de corto plazo a cambio de una ganancia a largo plazo. Comprendiendo lo que es la presión sobre los calendarios de trabajo, la mayoría de las personas prefieren posponer las molestias para el futuro.

La reestructuración de factores es un término que describe técnicas con las que se reducen las molestias a corto plazo del rediseño. Cuando se reestructuran los factores, no cambia la funcionalidad del programa; lo que se hace es cambiar su estructura interna, a fin de simplificar su lectura y su modificación.

Los cambios por la reestructuración de factores usualmente se hacen a pasos pequeños: Renombrar un método; mover un campo de una clase a otra; consolidar dos :métodos similares en Una superclase. Cada paso es pequeño; sin embargo, un par de horas invertidas en estos pequeños pasos pueden hacer maravillas por el programa.

La reestructuración de factores se vuelve más sencilla con los siguientes principios:

- No reestructure un programa y agregue funcionalidad al mismo tiempo; separe claramente ambas acciones mientras trabaja. Puede alternadas en pasos cortos; por ejemplo media hora de reestructuración, una hora agregando una nueva función, y otra media hora Reestructurando el código recién añadido.
- Asegúrese de tener a la mano buenas pruebas antes de comenzar la reestructuración. Ejecute las pruebas con la mayor frecuencia posible. De este modo, sabrá pronto si sus cambios han echado a perder algo.
- Avance con pasos cortos y deliberados. Mueva un campo de una clase a otra. Fusione dos métodos similares, creando una superclase. La reestructuración implica con frecuencia hacer muchos cambios localizados cuyo resultado sea un cambio de mayor escala. Si sus pasos son cortos y los prueba de uno en uno, evitará depuraciones prolongadas.

Se deberá reestructurar en los casos siguientes:

- Cuando se añada alguna funcionalidad al programa Y se descubra que el código viejo estorba. En cuanto esto se vuelva un problema, suspenda la adición de la nueva función y reestructure el código viejo.
- Cuando se tengan dificultades para comprender el código. La reestructuración es un buen modo de ayudarle a comprender el código y de retener este entendimiento para el futuro.

Frecuentemente le sucederá que quiera reestructurar algún código escrito por otro. Cuando lo haga, llévelo a cabo con el autor del código. Es muy difícil escribir un código de modo que otros lo puedan comprender con facilidad. La mejor forma de reestructurar es trabajando junto con alguien que entienda el código. Así podrá combinar el conocimiento de él con la inexperiencia de usted.

Cuándo reestructurar

La reestructuración de factores es una técnica que no se aprovecha suficientemente. Apenas ha comenzado a ser reconocida, en especial en la comunidad de Smalltalk. Creo, sin embargo, que se trata de una técnica clave para mejorar el desarrollo del software en cualquier medio ambiente. Asegúrese de que entiende la manera de llevar a cabo la reestructuración de manera disciplinada. Una forma de logrado es que su tutor le enseñe las técnicas.

Para mayor información

Debido a su novedad, es poco lo que se ha escrito sobre la reestructuración. La tesis de doctorado de William Opdyke (1992) es probablemente el tratado más completo sobre el tema, aunque está orientado a las herramientas automáticas de reestructuración, más que a las técnicas que le puedan servir a la gente. Kent Beckes uno de los más destacados exponentes de la reestructuración; su libro sobre patrones (1996) incluye muchos de éstos que son medulares para la reestructuración. Véase también el artículo de Beck de 1997, que da una buena idea del proceso de reestructuración.

Si usa VisualWorks o Smalltalk de IBM, deberá bajar Refactory, una herramienta que maneja reestructuración (véase [http://st-www.cs.uiuc.edu/users/droberts/Refactory.html]). Esta herramienta fue desarrollada por Don

Roberts y John Brandt, quienes trabajan con Ralph Johnson en la Universidad de Illinois. Considero que esta herramienta es el desarrollo más importante en herramientas de codificación desde el Ambiente integrado de desarrollo (Integrated Development Environment).

Patrones

El UML le dice cómo expresar un diseño orientado a objetos. Los **patrones,** por el contrario, se refieren a los resultados del proceso: modelos ejemplo.

Muchas personas han comentado que los proyectos tienen problemas debido a que la gente que intervino no conocía diseños que a personas con más experiencia les son familiares. Los patrones describen maneras comunes de hacer las cosas. Son redactados por personas que detectan temas que se repiten en los diseños. Estas personas toman cada tema y lo describen de tal forma que otros lo puedan leer y sepan cómo aplicado.

Veamos un ejemplo. Digamos que se están ejecutando varios objetos en un proceso en su computadora personal y necesita comunicarse con otros objetos en ejecución en otro proceso. Tal vez este proceso también esté en su computadora; o tal vez se encuentre en otra parte. Usted no quiere que los objetos de su sistema tengan que preocuparse por encontrar otros objetos en la red ni que tengan que ejecutar llamadas a procedimientos remotos.

Lo que puede hacer es crear un objeto suplente dentro del proceso local para el objeto remoto. El suplente tiene la misma interfaz que el objeto remoto. Los objetos locales le hablan al suplente mediante el envío de mensajes normales del proceso. El suplente es responsable de pasar los mensajes al objeto real, dondequiera que resida.

La Figura 2-2 es un diagrama de clases que ilustra la estructura del patrón Suplente.

Los suplentes son una técnica común que se emplea en redes y otras partes.

Hay una gran experiencia en el empleo de suplentes, en términos del conocimiento de la manera de usados, las ventajas que pueden ofrecer, sus limitaciones y la manera de implementarlas.

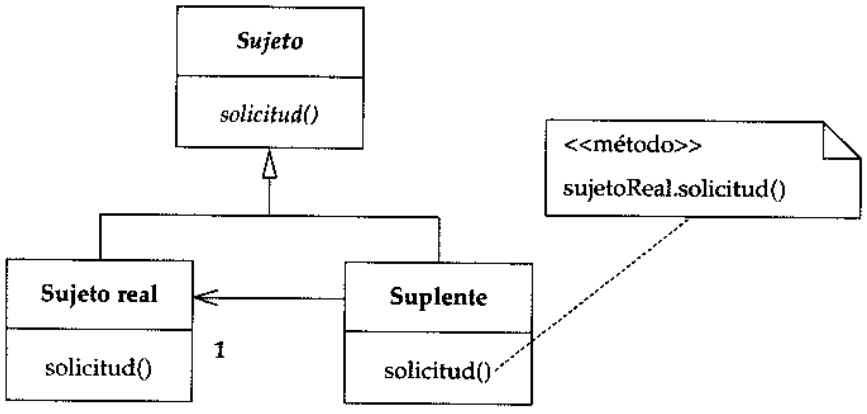

Figura 2-2: *Patrón de diseño de un suplente*

El **suplente** es un patrón de diseño porque describe una técnica de diseño. También pueden existir patrones en otras áreas. Digamos que se diseña un sistema de administración de riesgos en los mercados financieros. Se necesita comprender cómo cambia con el transcurso del tiempo el valor de una cartera de acciones. Se puede hacer lo anterior manteniendo un precio por cada acción y fechando el precio. Sin embargo, también querrá considerar lo que sucederá en casos hipotéticos (por ejemplo, "¿qué pasaría si se desplomara el precio del petróleo?").

Para esto se puede crear un escenario con el conjunto completo de los precios de las acciones. Luego, se puede tener escenarios individuales para los precios de la última semana, la mejor estimación para la siguiente semana, la estimación para la próxima semana, si se desploman los precios del petróleo, y así sucesivamente. Este patrón escenario (véase la Figura 2-3) es un patrón de análisis porque describe una pieza para modelar dominios.

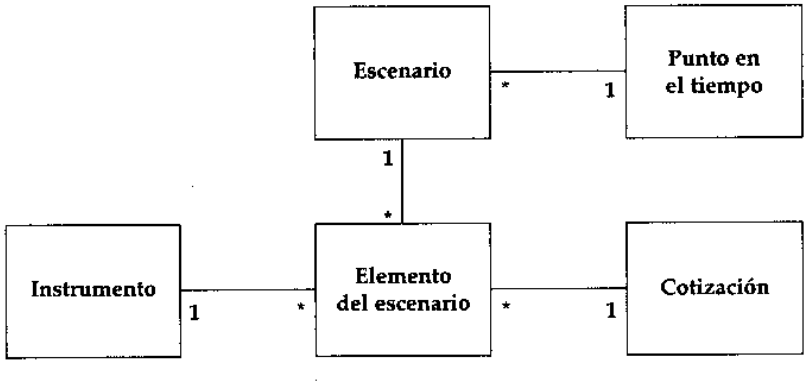

Figura 2-3: *Patrón de análisis de escenarios*

Los patrones de análisis son valiosos porque permiten un mejor comienzo cuando se trabaja con un dominio nuevo. Yo comencé a reunir patrones de análisis porque estaba frustrado por los nuevos dominios. Sabía que no era la primera persona en modelados; sin embargo, cada ocasión tenía que volver a comenzar desde cero.

Lo interesante sobre los patrones de análisis es que se aparecen en los lugares más inesperados. Cuando comencé a trabajar en un proyecto de análisis financiero corporativo, fueron de enorme ayuda una serie de patrones que había descubierto en un trabajo previo en la rama de salud.

Un patrón es mucho más que un modelo. El patrón debe incluir la razón por la cual es como es. Se dice con frecuencia que un patrón es una solución a un problema. El patrón debe dejar claro el problema, explicar por qué lo resuelve y además explicar las circunstancias bajo las que funciona y bajo las que no.

Los patrones son importantes porque son la siguiente etapa tras el dominio de los elementos básicos de un lenguaje o técnica para modelar. Los patrones ofrecen una serie de soluciones y enseñan lo que constituye un buen modelo y cómo se construye. Enseñan con el ejemplo.

Cuándo utilizar patrones

Se deben usar patrones todo el tiempo. Siempre que se pretenda desarrollar algo relacionado con análisis, diseño, codificación o administración de proyectos, se deberán buscar patrones que pueden ser de utilidad.

Para mayor información

En la actualidad, el campo de los patrones aún es muy joven, por lo cual no hay mucho material (lo cual es una ventaja en cierto modo, ya que la comunidad de los patrones no ha determinado aún la manera de indexar el material).

La fuente central de información sobre patrones es la Patterns Home Page (Página base de patrones) en la red: <http://st-www.cs.uiuc.edu/research-gp.html>. Aquí encontrará información clave sobre libros, conferencias y demás. Hay una serie de patrones e información acerca de éstos en la página Portland Patterns Repository (Depósito de patrones de Portland) de Ward Cunningham: <http://c2.com/ppr/index.html>.

TARJETAS CRC

Resumen
Índice de Contenidos

Tarjetas de CRC

A fines de la década de 1980, uno de los centros más grandes de tecnología de objetos era el laboratorio de investigación de Tektronix, en Portland, Oregon, Estados Unidos. Este laboratorio tenía algunos de los principales usuarios de Smalltalk y muchas de las ideas clave de la tecnología de objetos se desarrollaron allí. Dos de sus programadores renombrados de Smalltalk eran Ward Cunningham y Kent Beck.

Tanto Cunningham como Beck estaban y siguen preocupados por cómo enseñar los profundos conocimientos de Smalltalk que habían logrado. De esta pregunta sobre cómo enseñar objetos surgió la sencilla técnica de las tarjetas de Clase-Responsabilidad-Colaboración (CRC).

En lugar de utilizar diagramas para desarrollar modelos, como lo hacían la mayoría de los metodólogos, Cunningham y Beck representaron las clases en tarjetas 4 x 6 [pulgadas]. Y en lugar de indicar atributos y métodos en las tarjetas, escribieron responsabilidades.

Ahora bien, ¿qué es una responsabilidad? En realidad es una descripción de alto nivel del propósito de una clase. La idea es tratar de eliminar la descripción de pedazos de datos y procesos y, en cambio, captar el propósito de la clase en unas cuantas frases. El que se haya seleccionado una tarjeta es deliberado. No se permite escribir más de lo que cabe en una tarjeta (véase la figura 4-4).

Nombre de la Clase : Pedido	
Responsabilidad	Colaboración
Revisa si hay elementos en existencia	Línea de pedido
Determina precio	Línea de pedido
Revisa si el pago es válido	Cliente
Despacha a la dirección de entrega	

Figura 4-4 Tarjeta de Clase-Responsabilidad-Colaboración (CRC)

La segunda C se refiere a los colaboradores. Con cada responsabilidad se indica cuáles son las otras clases con las que se tiene que trabajar para cumplida. Esto da cierta idea sobre los vínculos entre las clases, siempre a alto nivel.

Uno de los principales beneficios de las tarjetas de CRC es que alientan la disertación animada entre los desarrolladores. Son especialmente eficaces cuando se está en medio de un caso de uso para ver cómo lo van a implementar las clases. Los desarrolladores escogen tarjetas a medida que cada clase colabora en el caso de uso. Conforme se van formando ideas sobre las responsabilidades, se pueden escribir en las tarjetas. Es importante pensar en las responsabilidades, ya que evita pensar en las clases como simples depositarias de datos, y ayuda a que el equipo se centre en comprender el comportamiento de alto nivel de cada clase.

Un error común que veo que comete la gente es generar largas listas de responsabilidades de bajo nivel. Este procedimiento es completamente fallido. Las responsabilidades deben caber sin dificultad en una tarjeta. Yo cuestionaría cualquier tarjeta que tenga más de tres responsabilidades. Plantéese la pregunta de si se deberá dividir la clase y si las responsabilidades se podrían indicar mejor integrándolas en enunciados de un mayor nivel.

Cuándo usar las tarjetas de CRC

Algunos consideran maravillosas las tarjetas de CRC; en cambio, a otros, esta técnica los deja indiferentes.

Yo considero definitivamente que se deberían probar, a fin de saber si al equipo de trabajo le gusta trabajar con ellas. Se deben usar, en particular, si el equipo se ha empantanado en demasiados detalles o si parecen identificar clases apelmazadas y carentes de definiciones claras.

Se pueden emplear diagramas de clase y diagramas de interacciones y para captar y formalizar los resultados del modelado CRC en un diseño con notación de UML. Asegúrese de que cada clase en su diagrama de clase tiene un enunciado de sus responsabilidades.

Para mayor información

Desafortunadamente, Cunningham y Beck no han escrito un libro sobre las CRC, pero usted puede encontrar su artículo original (Beck y Cunningham 1989) en la Webhttp://c2.com/doc/oopsla89/paper.html. En general, el libro que mejor describe esta técnica y, de hecho, todo el concepto del uso de responsabilidades es el de Rebeca Wirfs-Brock (1990). Es un libro relativamente viejo según las normas de la OO, pero se ha añejado bien.

Diseño por contrato

El diseño por contrato es una técnica diseñada por Bertrand Meyer. Esta técnica es una característica central del lenguaje Eiffel, que él desarrolló. Sin embargo, el Diseño por contrato no es específico de Eiffel; es una técnica valiosa que se puede usar con cualquier lenguaje de programación.

En el corazón del Diseño por contrato se encuentra la afirmación. Una **afirmación** es un enunciado Booleano que nunca debe ser falso y, por tanto, sólo lo será debido a una falla. Por lo común, las afirmaciones se comprueban sólo durante la depuración y no durante la ejecución en producción. De hecho, un programa nunca debe suponer que se están comprobando las afirmaciones.

El Diseño por contrato se vale de tres tipos de afirmaciones: poscondiciones, precondiciones e invariantes.

Las precondiciones y las poscondiciones se aplican a las operaciones. Una**poscondición** es un enunciado sobre cómo debería verse el mundo después de la ejecución de una operación. Por ejemplo, si definimos la operación "cuadrado" sobre un número, la poscondición adoptaría la forma de *resultado = éste * éste*, donde *resultado* es el producto y *éste* es el objeto sobre el cual se invocó la operación. La poscondición es una forma útil de decir

qué es lo que hacemos, sin decir cómo lo hacemos; en otras palabras, de separar la interfaz de la implementación.

La **precondición** es un enunciado de cómo esperamos encontrar el mundo, antes de ejecutar una operación. Podemos definir una precondición para la operación *"cuadrado" de éste > = 0*. Tal precondición, dice que es un error invocar a "cuadrado" sobre un número negativo, y que las consecuencias de hacerlo son indefinidas.

A primera vista, ésta no parece ser una buena idea, pues deberíamos poner una comprobación en alguna parte para garantizar que se invoque correctamente a "cuadrado". La cuestión importante es quien es responsable de hacerlo.

La precondición hace explícito que quien invoca es el responsable de la comprobación. Sin este enunciado explícito de responsabilidades, podemos tener muy poca comprobación (pues ambas partes suponen que la otra es la responsable) o tenerla en demasía (cuando ambas partes lo hacen). Demasiada comprobación es mala, ya que provoca que se duplique muchas veces el código de comprobación, lo cual puede complicar de manera importante un programa. El ser explícito sobre quién es el responsable contribuye a reducir esta complejidad. Se reduce el peligro de que el invocador olvide comprobar por el hecho de que, por lo general, las afirmaciones se comprueban durante las pruebas y la depuración.

A partir de estas definiciones de precondición y poscondición, podemos ver una definición sólida del término **excepción,** que ocurre cuando se invoca una operación estando satisfecha su precondición y, sin embargo, no puede regresar con su poscondición satisfecha.

Una **invariante** es una afirmación acerca de una clase. Por ejemplo, la clase Cuenta puede tener una invariante que diga que *balance = sum(entradas.cantidad())* La invariante "siempre" es verdadera para todas las instancias de la clase. En este contexto,"*siempre* " significa *"siempre que el objeto esté disponible para que se invoque una operación sobre ella"*.

En esencia, lo anterior significa que la invariante se agrega a las precondiciones y poscondiciones asociadas a todas las operaciones públicas de la clase dada. La invariante puede volverse falsa durante la ejecución de un método pero debe haberse restablecido a verdadera para el momento en que cualquier otro objeto pueda hacerle algo al receptor.

Las afirmaciones pueden desempeñar un papel único en la subclasificación.

Uno de los **peligros del polimorfismo** es que se podrían redefinir las operaciones de una subclase, de tal modo que fueran inconsistentes con las operaciones de la superclase. Las afirmaciones impiden hacer esto. Las invariantes y poscondiciones de una clase deben aplicarse a todas las subclases. Las subclases pueden elegir el refuerzo de estas afirmaciones, pero no pueden debilitadas. La precondición, por otra parte, no puede reforzarse, aunque sí debilitarse.

Esto parecerá extraño a primera vista, pero es importante para permitir vínculos dinámicos. Siempre debería ser posible tratar un objeto de subclase como si fuera una instancia de su superclase (de acuerdo con el principio de la sustituibilidad). Si una subclase retuerza su precondición, entonces podría fallar una operación de superclase, cuando se aplicara a la subclase.

En esencia, las afirmaciones sólo pueden aumentar las responsabilidades de la subclase. Las precondiciones son un enunciado para trasladar una responsabilidad al invocador; se incrementan las responsabilidades de una clase cuando se debilita una precondición. En la práctica, todo esto permite un mucho mejor control de la subclasificación y ayuda a asegurar que las subclases se comporten de manera adecuada.

Idealmente, las afirmaciones se deben incluir en el código, como parte de la definición de la interfaz. Los compiladores deben poder encender la comprobación por afirmaciones durante las depuraciones y apagada durante la operación en producción. Se pueden manejar varias etapas de comprobación de afirmaciones. Las precondiciones a menudo ofrecen las mejores posibilidades de atrapar errores con la menor cantidad de sobrecarga de proceso.

Cuándo utilizar el Diseño por contrato

El Diseño por contrato es una técnica valiosa que siempre se debe emplear cuando se programa. Es particularmente útil para la construcción de interfaces claras.

Sólo Eiffel maneja afirmaciones como parte de su lenguaje, pero desafortunadamente Eiffel no es un lenguaje de amplía difusión. La adición de mecanismos a C ++ y Smalltalk que manejen algunas afirmaciones es directa. Es mucho más engorroso hacer algo similar en Java, pero es posible.

El UML no habla mucho acerca de las afirmaciones, pero se pueden usar sin problemas. Las invariantes son equivalentes a las reglas de condición en los diagramas de clase y se deberán usar tanto como sea Posible. Las precondiciones y poscondiciones de operación se deben documentar con las definiciones de las operaciones.

Para mayor información

El libro de Meyer (1997) es una obra clásica (aunque, a estas alturas, muy denso) sobre el diseño OO, donde se habla mucho de las afirmaciones. Kim Walden y Jean-Marc Nerson (1995) y Steve Cook y John Daniels (1994) utilizan ampliamente el Diseño por contrato en sus obras.

También se puede obtener mayor información de ISE (la compañía de Bertrand Meyer).