APÉNDICE 1

INTRODUCCIÓN A UML[1]

CONCEPTOS CLAVE

El *Lenguaje de Modelado Unificado* (UML) es "un lenguaje estándar para escribir diseños de software. El UML puede usarse para visualizar, especificar, construir y documentar los artefactos de un sistema de software intensivo" [Boo05]. En otras palabras, tal como los arquitectos de edificios crean planos para que los use una compañía constructora, los arquitectos de software crean diagramas de UML para ayudar a los desarrolladores de software a construir el software. Si usted entiende el vocabulario del UML (los elementos pictóricos de los diagramas y su significado) puede comprender y especificar con mucha más facilidad un sistema, y explicar su diseño a otros.

Grady Booch, Jim Rumbaugh e Ivar Jacobson desarrollaron el UML a mediados de los años noventa del siglo pasado con mucha realimentación de la comunidad de desarrollo de software. El UML fusionó algunas notaciones de modelado que competían entre sí y que se usaban en la industria del software en la época. En 1997, UML 1.0 se envió al Object Management Group, un consorcio sin fines de lucro involucrado en especificaciones de mantenimiento para su empleo en la industria de la computación. El UML 1.0 se revisó y dio como resultado la adopción del UML 1.1 ese mismo año. El estándar actual es UML 2.0 y ahora es un estándar ISO. Puesto que este estándar es tan nuevo, muchas antiguas referencias, como [Gam95], no usan notación de UML.

UML 2.0 proporciona 13 diferentes diagramas para su uso en modelado de software. En este apéndice se analizarán solamente diagramas de *clase, implementación, caso de uso, secuencia, comunicación, actividad* y *estado*, diagramas que se usan en esta edición de *Ingeniería del software. Un enfoque práctico*.

Debe observar que existen muchas características opcionales en diagramas de UML. El UML ofrece dichas opciones (en ocasiones complejas) de modo que pueda expresar todos los aspectos importantes de un sistema. Al mismo tiempo, tiene la flexibilidad para suprimir aquellas partes del diagrama que no son relevantes para el aspecto que se va a modelar, con la finalidad de evitar confundir el diagrama con detalles irrelevantes. Por tanto, la omisión de una característica particular no significa que ésta se encuentre ausente; puede significar que la característica se suprimió. En este apéndice, *no* se presenta la cobertura exhaustiva de todas las características de los diagramas de UML. El apéndice se enfocará en las opciones estándar, en especial en aquellas que se usaron en este libro.

DIAGRAMAS DE CLASE

Para modelar clases, incluidos sus atributos, operaciones, relaciones y asociaciones con otras clases,[2] el UML proporciona un *diagrama de clase*, que aporta una visión estática o de estructura de un sistema, sin mostrar la naturaleza dinámica de las comunicaciones entre los objetos de las clases.

1 Este apéndice fue una aportación de Dale Skrien y se adaptó de su libro *An Introduction to Object-Oriented Design and Design Patterns in Java* (McGraw-Hill, 2008). Todo el contenido se usa con permiso.

2 Si el lector no está familiarizado con los conceptos orientados a objeto, en el apéndice 2 se presenta una breve introducción.

Los elementos principales de un diagrama de clase son cajas, que son los íconos utilizados para representar clases e interfaces. Cada caja se divide en partes horizontales. La parte superior contiene el nombre de la clase. La sección media menciona sus atributos. Un *atributo* es algo que un objeto de dicha clase conoce o puede proporcionar todo el tiempo. Por lo general, los atributos se implementan como campos de la clase, pero no necesitan serlo. Podrían ser valores que la clase puede calcular a partir de sus variables o valores instancia y que puede obtener de otros objetos de los cuales está compuesto. Por ejemplo, un objeto puede conocer siempre la hora actual y regresarla siempre que se le solicite. Por tanto, sería adecuado mencionar la hora actual como un atributo de dicha clase de objetos. Sin embargo, el objeto muy probablemente no tendría dicha hora almacenada en una de sus variables instancia, porque necesitaría actualizar de manera continua ese campo. En vez de ello, el objeto probablemente calcularía la hora actual (por ejemplo, a través de consulta con objetos de otras clases) en el momento en el que se le solicite la hora. La tercera sección del diagrama de clase contiene las operaciones o comportamientos de la clase. Una *operación* es lo que pueden hacer los objetos de la clase. Por lo general, se implementa como un método de la clase.

La figura A1.1 presenta un ejemplo simple de una clase **Thoroughbred** (pura sangre) que modela caballos de pura sangre. Muestra tres atributos: **mother** (madre), **father** (padre) y **birthyear** (año de nacimiento). El diagrama también muestra tres operaciones: *getCurrentAge()* (obtener edad actual), *getFather()* (obtener padre) y *getMother()* (obtener madre); puede haber otros atributos y operaciones suprimidos que no se muestren en el diagrama.

Cada atributo puede tener un nombre, un tipo y un nivel de visibilidad. El tipo y la visibilidad son opcionales. El tipo sigue al nombre y se separa de él mediante dos puntos. La visibilidad se indica mediante un –, #, ~ o + precedente, que indica, respectivamente, visibilidad *privada*, *protegida*, *paquete* o *pública*. En la figura A1.1, todos los atributos tienen visibilidad privada, como se indica mediante el signo menos que los antecede (–). También es posible especificar que un atributo es estático o de clase, subrayándolo. Cada operación puede desplegarse con un nivel de visibilidad, parámetros con nombres y tipos, y un tipo de retorno.

Una clase abstracta o un método abstracto se indica con el uso de cursivas en el nombre del diagrama de clase. Vea, por ejemplo, la clase **Horse** (caballo) en la figura A1.2. Una interfaz se indica con la frase "<<interface>>" (llamada *estereotipo*) arriba del nombre. Vea la interfaz **OwnedObject** (objeto posesión) en la figura A1.2. Una interfaz también puede representarse gráficamente mediante un círculo hueco.

Vale la pena mencionar que el ícono que representa una clase puede tener otras partes opcionales. Por ejemplo, puede usarse una cuarta sección en el fondo de la caja de clase para mencionar las responsabilidades de la clase. Esta sección es particularmente útil cuando se realiza la transición de tarjetas CRC (capítulo 6) a diagramas de clase, donde las responsabilidades mencionadas en las tarjetas CRC pueden agregarse a esta cuarta sección en la caja de clase en el diagrama UML antes de crear los atributos y operaciones que llevan a cabo dichas responsabilidades. Esta cuarta sección no se muestra en ninguna de las figuras de este apéndice.

FIGURA A1.1
Diagrama de clase para una clase Thoroughbred

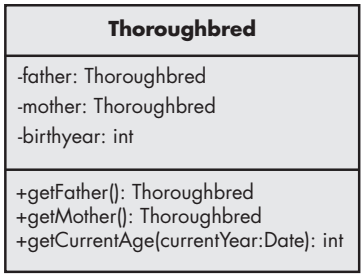
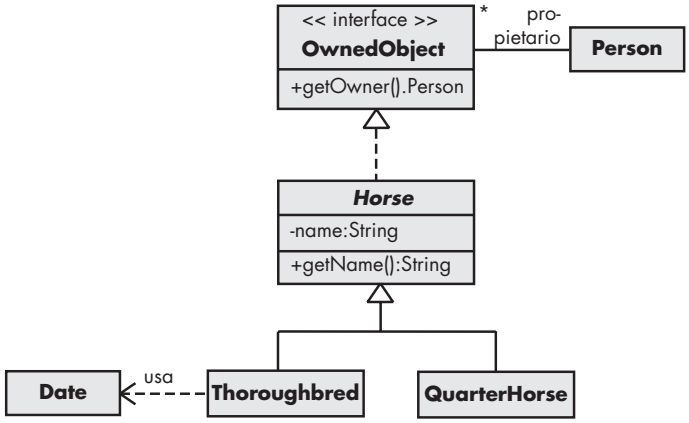


FIGURA A1.2
Diagrama de clase concerniente a caballos



Los diagramas de clase también pueden mostrar relaciones entre clases. Una clase que sea una subclase de otra clase se conecta con ella mediante una flecha con una línea sólida y con una punta triangular hueca. La flecha apunta de la subclase a la superclase. En UML, tal relación se llama *generalización*. Por ejemplo, en la figura A1.2, las clases **Thoroughbred** y **QuarterHorse** (caballo cuarto de milla) se muestran como subclases de la clase abstracta **Horse**. Una flecha con una línea punteada indica implementación de una interfaz. En UML, tal relación se llama *realización*. Por ejemplo, en la figura A1.2, la clase **Horse** implementa o realiza la interfaz **OwnedObject**.

Una *asociación* entre dos clases significa que existe una relación estructural entre ellas. Las asociaciones se representan mediante líneas sólidas. Una asociación tiene muchas partes opcionales. Puede etiquetarse, así como cada una de sus terminaciones, para indicar el papel de cada clase en la asociación. Por ejemplo, en la figura A1.2, existe una asociación entre **OwnedObject** y **Person**, en la que **Person** juega el papel de owner (propietario). Las flechas en cualquiera o en ambos lados de una línea de asociación indican navegabilidad. Además, cada extremo de la línea de asociación puede tener un valor de multiplicidad desplegado. Navegabilidad y multiplicidad se explican con más detalle más adelante, en esta sección. Una asociación también puede conectar una clase consigo misma, mediante un bucle. Tal asociación indica la conexión de un objeto de la clase con otros objetos de la misma clase.

Una asociación con una flecha en un extremo indica navegabilidad en un sentido. La flecha significa que, desde una clase, es posible acceder con facilidad a la segunda clase asociada hacia la que apunta la asociación; sin embargo, desde la segunda clase, no necesariamente puede accederse con facilidad a la primera clase. Otra forma de pensar en esto es que la primera clase está al tanto de la segunda, pero el segundo objeto de clase no necesariamente está directamente al tanto de la primera clase. Una asociación sin flechas por lo general indica una asociación de dos vías, que es lo que se pretende en la figura A1.2; pero también simplemente podría significar que la navegabilidad no es importante y, por tanto, que queda fuera.

Debe observarse que un atributo de una clase es muy parecido a una asociación de la clase con el tipo de clase del atributo. Es decir, para indicar que una clase tiene una propiedad llamada "name" (nombre) de tipo String, podría desplegarse dicha propiedad como atributo, como en la clase **Horse** en la figura A1.2. De manera alternativa, podría crearse una asociación de una vía desde la clase **Horse** hasta la clase **String** donde el papel de la clase String es "name". El enfoque de atributo es mejor para tipos de datos primitivos, mientras que el enfoque de asociación con frecuencia es mejor si la clase propietaria juega un papel principal en el diseño, en cuyo caso es valioso tener una caja de clase para dicho tipo.

Una relación de *dependencia* representa otra conexión entre clases y se indica mediante una línea punteada (con flechas opcionales en los extremos y con etiquetas opcionales). Una clase depende de otra si los cambios en la segunda clase pueden requerir cambios en la primera. Una asociación de una clase con otra automáticamente indica una dependencia. No se necesitan líneas punteadas entre clases si ya existe una asociación entre ellas. Sin embargo, para una relación transitoria (es decir, una clase que no mantiene alguna conexión de largo plazo con otra, sino que usa dicha clase de manera ocasional), debe dibujarse una línea punteada desde la primera clase hasta la segunda. Por ejemplo, en la figura A1.2, la clase **Thoroughbred** usa la clase **Date** (fecha) siempre que se invoca su método *getCurrentAge()*; por eso la dependencia se etiqueta "usa".

La *multiplicidad* de un extremo de una asociación significa el número de objetos de dicha clase que se asocia con la otra clase. Una multiplicidad se especifica mediante un entero no negativo o mediante un rango de enteros. Una multiplicidad especificada por "0..1" significa que existen 0 o 1 objetos en dicho extremo de la asociación. Por ejemplo, cada persona en el mundo tiene un número de seguridad social o no lo tiene (especialmente si no son ciudadanos estadounidenses); por tanto, una multiplicidad de 0..1 podría usarse en una asociación entre una clase **Person** y una clase **SocialSecurityNumber** (número de seguridad social) en un diagrama de clase. Una multiplicidad que se especifica como "1...*" significa uno o más, y una multiplicidad especificada como "0...*" o sólo "*" significa cero o más. Un * se usa como la multiplicidad en el extremo **OwnedObject** de la asociación con la clase **Person** en la figura A1.2, porque una **Person** puede poseer cero o más objetos.

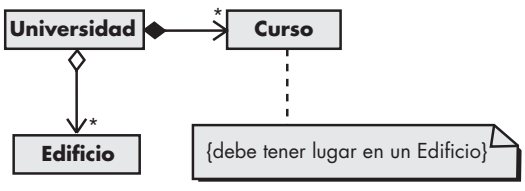
Si un extremo de una asociación tiene multiplicidad mayor que 1, entonces los objetos de la clase a la que se refiere en dicho extremo de la asociación probablemente se almacenan en una colección, como un conjunto o lista ordenada. También podría incluirse dicha clase colección en sí misma en el diagrama UML, pero tal clase por lo general queda fuera y se supone, de manera implícita, que está ahí debido a la multiplicidad de la asociación.

Una *agregación* es un tipo especial de asociación que se indica mediante un diamante hueco en un extremo del ícono. Ello indica una relación "entero/parte", en la que la clase a la que apunta la flecha se considera como una "parte" de la clase en el extremo diamante de la asociación. Una *composición* es una agregación que indica fuerte propiedad de las partes. En una composición, las partes viven y mueren con el propietario porque no tienen papel en el sistema de software independiente del propietario. Vea la figura A1.3 para ejemplos de agregación y composición.

Una **Universidad** tiene una agregación de objetos **Edificio**, que representan los edificios que constituyen el campus. La universidad también tiene una colección de cursos. Si la universidad quebrara, los edificios todavía existirían (si se supone que la universidad no se destruye físicamente) y podría usar otras cosas, pero un objeto **Curso** no tiene uso fuera de la universidad en la cual se ofrece. Si la universidad deja de existir como una entidad empresarial, el objeto **Curso** ya no sería útil y, por tanto, también dejaría de existir.

Otro elemento común de un diagrama de clase es una *nota*, que se representa mediante una caja con una esquina doblada y se conecta a otros íconos mediante una línea punteada. Puede

FIGURA A1.3
Relación entre universidades, cursos y edificios



tener contenido arbitrario (texto y gráficos) y es similar a comentarios en lenguajes de programación. Puede contener comentarios acerca del papel de una clase o restricciones que todos los objetos de dicha clase deban satisfacer. Si los contenidos son una restricción, se encierran entre llaves. Observe la restricción unida a la clase **Curso** en la figura A1.3.

Diagramas de implementación

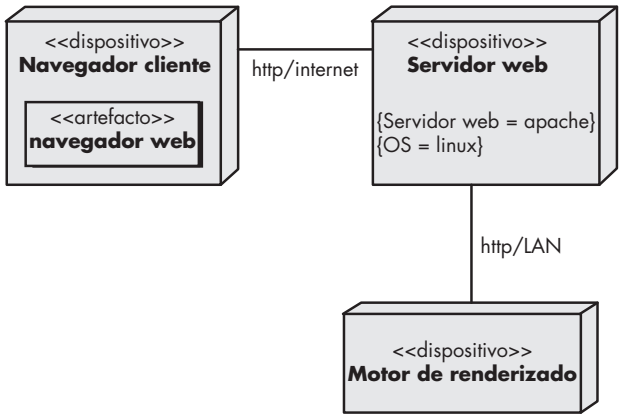
Un *diagrama de implementación* UML se enfoca en la estructura de un sistema de software y es útil para mostrar la distribución física de un sistema de software entre plataformas de hardware y entornos de ejecución. Suponga, por ejemplo, que desarrolla un paquete de renderizado de gráficos basado en web. Los usuarios de su paquete usarán su navegador web para ir a su sitio web e ingresar la información que se va a renderizar. Su sitio web renderizaría una imagen gráfica de acuerdo con la especificación del usuario y la enviaría de vuelta al usuario. Puesto que las gráficas renderizadas pueden ser computacionalmente costosas, usted decide mover el renderizado afuera del servidor web y hacia una plataforma separada. Por tanto, habrá tres dispositivos de hardware involucrados en su sistema: el cliente web (la computadora que corre un navegador del usuario), la computadora que alberga el servidor web y la computadora que alberga el motor de renderizado.

La figura A1.4 muestra el diagrama de implementación para tal paquete. En ese diagrama, los componentes de hardware se dibujan en cajas marcadas con "<<dispositivo>>". Las rutas de comunicación entre componentes de hardware se dibujan con líneas con etiquetas opcionales. En la figura A1.4, las rutas se etiquetan con el protocolo de comunicación y con el tipo de red utilizado para conectar los dispositivos.

Cada nodo que hay en un diagrama de implementación también puede anotarse con detalles del dispositivo. Por ejemplo, en la figura A1.4 se ilustra el navegador cliente para mostrar que contiene un artefacto que consiste en el software del navegador web. Un artefacto por lo general es un archivo que contiene software que corre en un dispositivo. También puede especificar valores etiquetados, como se muestra en la figura A1.4 en el nodo del servidor web. Dichos valores definen al proveedor del servidor web y al sistema operativo que usa el servidor.

Los diagramas de implementación también pueden mostrar nodos de entorno de ejecución, que se dibujan como cajas que contienen la etiqueta "<<entorno de ejecución>>". Dichos nodos representan sistemas, como sistemas operativos, que pueden albergar otro software.

Figura A1.4
Diagrama de implementación



Diagramas de uso de caso

Los casos de uso (capítulos 5 y 6) y el *diagrama de uso de caso* UML ayudan a determinar la funcionalidad y características del software desde la perspectiva del usuario. Para proporcionarle una aproximación a la manera en la que funcionan los casos de uso y los diagramas de uso de caso, se crearán algunos para una aplicación de software que gestiona archivos de música digital, similar al software iTunes de Apple. Algunas de las cosas que puede incluir el software son funciones para:

- Descargar un archivo de música MP3 y almacenarlo en la biblioteca de la aplicación.
- Capturar música de streaming (transmisión continua) y almacenarla en la biblioteca de la aplicación.
- Gestionar la biblioteca de la aplicación (por ejemplo, borrar canciones u organizarlas en listas de reproducción).
- Quemar en CD una lista de las canciones de la biblioteca.
- Cargar una lista de las canciones de la biblioteca en un iPod o reproductor MP3.
- Convertir una canción de formato MP3 a formato AAC y viceversa.

Ésta no es una lista exhaustiva, pero es suficiente para entender el papel de los casos de uso y los diagramas de uso de caso.

Un *caso de uso* describe la manera en la que un usuario interactúa con el sistema, definiendo los pasos requeridos para lograr una meta específica (por ejemplo, quemar una lista de canciones en un CD). Las variaciones en la secuencia de pasos describen varios escenarios (por ejemplo, ¿y si todas las canciones de la lista no caben en un CD?).

Un diagrama UML de uso de caso es un panorama de todos los casos de uso y sus relaciones. El mismo proporciona un gran cuadro de la funcionalidad del sistema. En la figura A1.5 se muestra un diagrama de uso de caso para la aplicación de música digital.

En este diagrama, la figura de palitos representa a un *actor* (capítulo 5) que se asocia con una categoría de usuario (u otro elemento de interacción). Por lo general, los sistemas complejos tienen más de un actor. Por ejemplo, una aplicación de máquina expendedora puede tener tres actores que representan clientes, personal de reparación y proveedores que rellenan la máquina.

En el diagrama de uso de caso, los casos de uso se muestran como óvalos. Los actores se conectan mediante líneas a los casos de uso que realizan. Observe que ninguno de los detalles de los casos de uso se incluye en el diagrama y, en vez de ello, necesita almacenarse por separado. Observe también que los casos de uso se colocan en un rectángulo, pero los actores no. Este rectángulo es un recordatorio visual de las fronteras del sistema y de que los actores están afuera del sistema.

Algunos casos de uso en un sistema pueden relacionarse mutuamente. Por ejemplo, existen pasos similares al de quemar una lista de canciones en un CD y cargar una lista de canciones en un iPod. En ambos casos, el usuario crea primero una lista vacía y luego agrega las canciones de la biblioteca a la lista. Para evitar duplicación en casos de uso, por lo general es mejor crear un nuevo caso de uso que represente la actividad duplicada y luego dejar que los otros casos de uso incluyan este nuevo caso de uso como uno de sus pasos. Tal inclusión se indica en los diagramas de uso de caso, como en la figura A1.6, mediante una flecha punteada etiquetada como “incluye”, que conecta un caso de uso con un caso de uso incluido.

Dado que despliega todos los casos de uso, un diagrama de uso de caso es un auxiliar útil para asegurar que cubrió toda la funcionalidad del sistema. En el organizador de música digital, seguramente querría más casos de uso, tal como uno para reproducir una canción de la biblioteca. Pero tenga en mente que la contribución más valiosa de casos de uso al proceso de desa-

FIGURA A1.5

Diagrama de caso de uso para el sistema de música

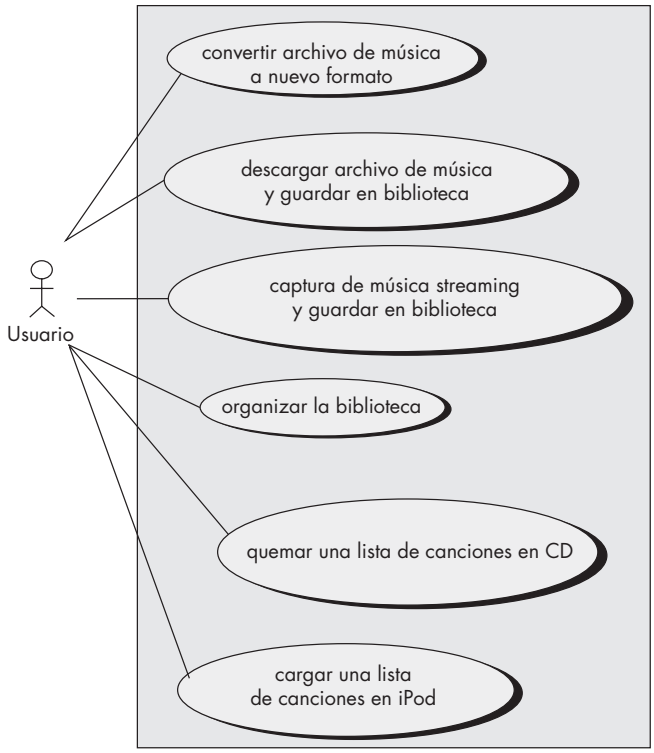
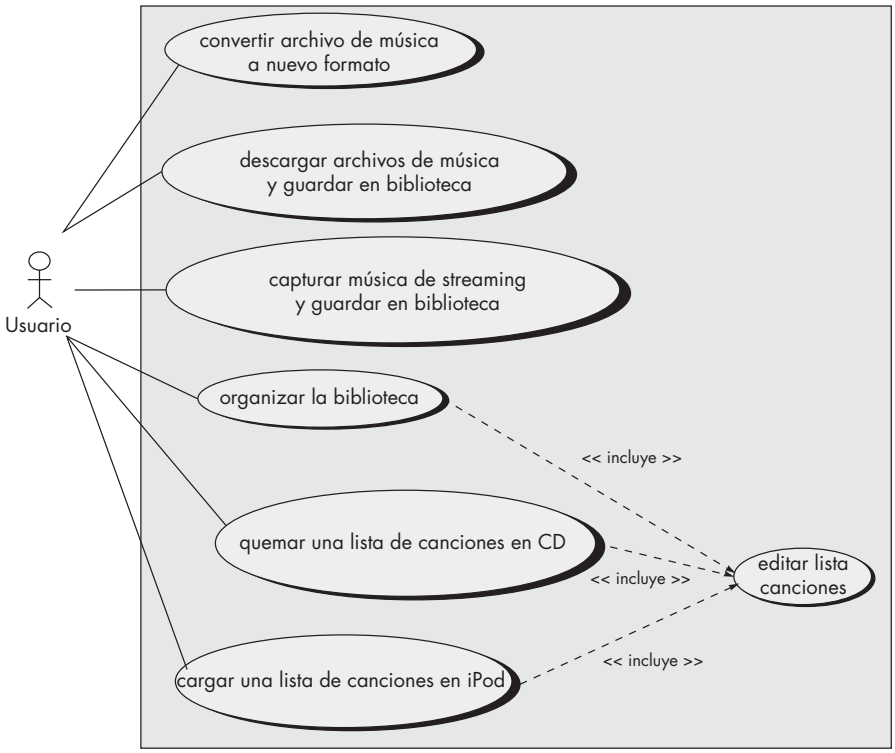


FIGURA A1.6

Diagrama de caso de uso con casos de uso incluidos



rrollo de software es la descripción textual de cada caso de uso, no el diagrama de uso de caso global [Fow04b]. Es a través de las descripciones que usted puede tener una comprensión clara de las metas del sistema que desarrolla.

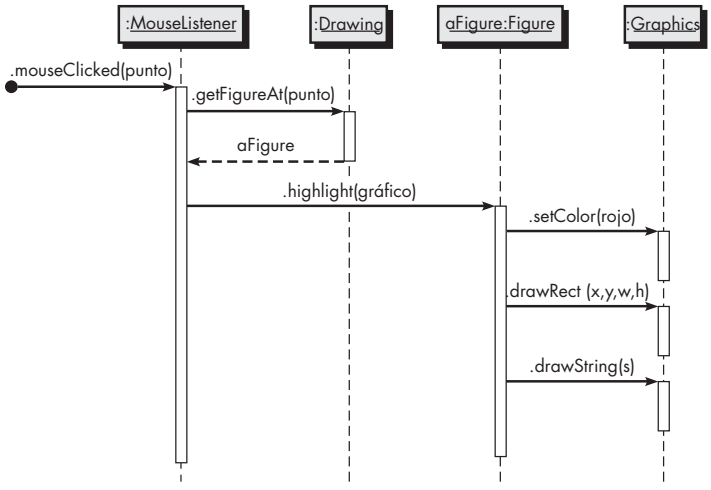
Diagramas de secuencia

En contraste con los diagramas de clase y con los diagramas de implementación, que muestran la estructura estática de un componente de software, un *diagrama de secuencia* se usa para mostrar las comunicaciones dinámicas entre objetos durante la ejecución de una tarea. Este tipo de diagrama muestra el orden temporal en el que los mensajes se envían entre los objetos para lograr dicha tarea. Puede usarse un diagrama de secuencia para mostrar las interacciones en un caso de uso o en un escenario de un sistema de software.

En la figura A1.7 se ve un diagrama de secuencia para un programa de dibujo. El diagrama muestra los pasos involucrados, resaltando una figura en un dibujo cuando se le da clic. Por lo general, cada caja de la fila que hay en la parte superior del diagrama corresponde a un objeto, aunque es posible hacer que las cajas modelen otras cosas, como clases. Si la caja representa un objeto (como es el caso en todos los ejemplos), entonces dentro de la caja puede establecerse de manera opcional el tipo del objeto, precedido por dos puntos. También se puede escribir un nombre del objeto antes de los dos puntos, como se muestra en la tercera caja de la figura A1.7. Abajo de cada caja hay una línea punteada llamada *línea de vida* del objeto. El eje vertical que hay en el diagrama de secuencia corresponde al tiempo, donde el tiempo aumenta conforme se avanza hacia abajo.

Un diagrama de secuencia muestra llamadas de método usando flechas horizontales desde el *llamador* hasta el *llamado*, etiquetado con el nombre del método y que opcionalmente incluye sus parámetros, sus tipos y el tipo de retorno. Por ejemplo, en la figura A1.7, **MouseListener** (escucha de ratón) llama al método *getFigureAt()* (obtener figura en) de **Drawing** (dibujo). Cuando un objeto ejecuta un método (es decir, cuando tiene un marco de activación en la pila), opcionalmente puede mostrar una barra blanca, llamada *barra de activación*, abajo de la línea de vida del objeto. En la figura A1.7, las barras de activación se dibujan para todas las llamadas de método. El diagrama también puede mostrar opcionalmente el retorno de una llamada de método con una flecha punteada y una etiqueta opcional. En la figura A1.7, el retorno de la llamada de método *getFigureAt()* se muestra con una etiqueta del nombre del objeto que regresa.

Figura A1.7
Ejemplo de diagrama de secuencia



Una práctica común, como se hizo en la figura A1.7, es dejar fuera la flecha de retorno cuando se llama un método nulo, pues desordena el diagrama a la vez que proporciona poca información de importancia. Un círculo negro con una flecha que sale de él indica un *mensaje encontrado* cuya fuente se desconoce o es irrelevante.

Ahora debe poder entenderse la tarea que implementa la figura A1.7. Una fuente desconocida llama al método *mouseClicked()* (clic del ratón) de un **MouseListener**, que pasa como argumento el punto donde ocurrió el clic. A su vez, el **MouseListener** llama al método *getFigureAt()* de un **Drawing**, que regresa una **Figure**. Luego el **MouseListener** llama el método resaltado de **Figure** y lo pasa como argumento al objeto **Graphics**. En respuesta, **Figure** llama tres métodos del objeto **Graphics** para dibujar la figura en rojo.

El diagrama en la figura A1.7 es muy directo y no contiene condicionales o bucles. Si se requieren estructuras de control lógico, probablemente sea mejor dibujar un diagrama de secuencia separado para cada caso, es decir, si el flujo del mensaje puede tomar dos rutas diferentes dependiendo de una condición, entonces dibuje dos diagramas de secuencia separados, uno para cada posibilidad.

Si se insiste en incluir bucles, condicionales y otras estructuras de control en un diagrama de secuencia, se pueden usar *marcos de interacción*, que son rectángulos que rodean partes del diagrama y que se etiquetan con el tipo de estructuras de control que representan. La figura A1.8 ilustra lo anterior, y muestra el proceso involucrado, resaltando todas las figuras dentro de un rectángulo determinado. El **MouseListener** envía el mensaje **rectDragged** (arrastre de recta). Entonces el **MouseListener** dice al dibujo que resalte todas las figuras en el rectángulo, llamando al método *highlightFigures()* (resaltar figuras) y pasando al rectángulo como argumento. El método hace bucles a través de todos los objetos **Figure** en el objeto **Drawing** y, si **Figure** intersecta el rectángulo, se pide a **Figure** que se resalte a sí mismo. Las frases entre corchetes se llaman *guardias*, que son condiciones booleanas que deben ser verdaderas si debe continuar la acción dentro del marco de interacción.

Existen muchas otras características especiales que pueden incluirse en un diagrama de secuencia. Por ejemplo:

1. Se puede distinguir entre mensajes sincrónicos y asíncronos. Los primeros se muestran con puntas de flecha sólidas mientras que los asíncronos lo hacen con puntas de flecha huecas.
2. Se puede mostrar un objeto que envía él mismo un mensaje con una flecha que parte del objeto, gira hacia abajo y luego apunta de vuelta hacia el mismo objeto.

FIGURA A1.8
Diagrama de secuencia con dos marcos de interacción

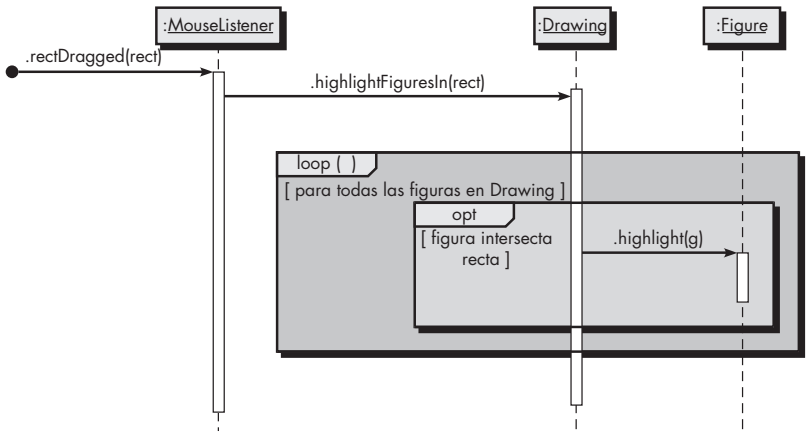
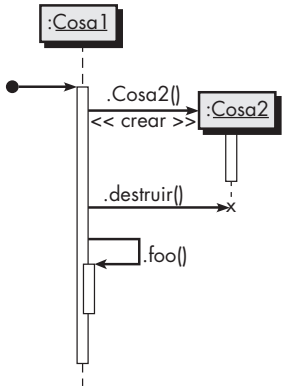


FIGURA A1.9
Creación, destrucción y bucles en diagramas de secuencia



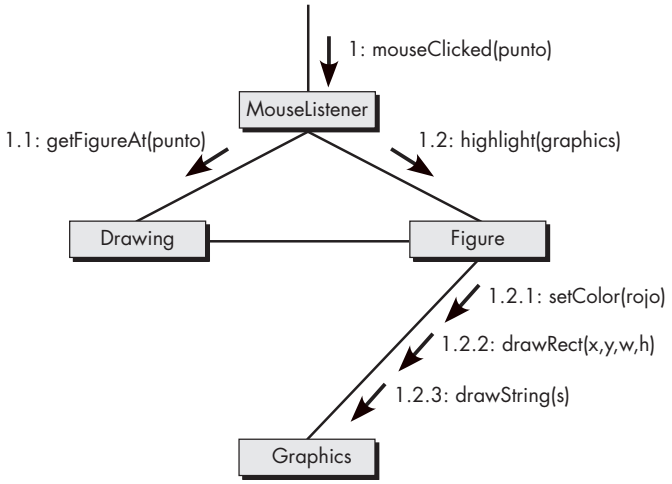
3. Se puede mostrar la creación de objeto dibujando una flecha etiquetada de manera adecuada (por ejemplo, con una etiqueta <<crear>>) hacia una caja de objeto. En este caso, la caja aparecerá en el diagrama más abajo que las cajas correspondientes a objetos que ya existen cuando comienza la acción.
4. Se puede mostrar destrucción de objeto mediante una gran X al final de la línea de vida del objeto. Otros objetos pueden destruir un objeto, en cuyo caso una flecha apunta desde el otro objeto hacia la X. Una X también es útil para indicar que un objeto ya no se usa y que, por tanto, está listo para la colección de basura.

Las últimas tres características se muestran en el diagrama de secuencia de la figura A1.9.

DIAGRAMAS DE COMUNICACIÓN

El *diagrama de comunicación* UML (llamado "diagrama de colaboración" en UML 1.X) proporciona otro indicio del orden temporal de las comunicaciones, pero enfatiza las relaciones entre los objetos y clases en lugar del orden temporal. El diagrama de comunicación que se ilustra en la figura A1.10 despliega las mismas acciones que se muestran en el diagrama de secuencia de la figura A1.7.

FIGURA A1.10
Diagrama de comunicación UML



En un diagrama de comunicación, los objetos interactuantes se representan mediante rectángulos. Las asociaciones entre objetos lo hacen mediante líneas que conectan los rectángulos. Por lo general, en el diagrama existe una flecha entrante hacia un objeto que comienza la secuencia de pase de mensaje. Esa flecha se etiqueta con un número y un nombre de mensaje. Si el mensaje entrante se etiqueta con el número 1 y si hace que el objeto receptor invoque otros mensajes en otros objetos, entonces los mencionados mensajes se representan mediante flechas desde el emisor hacia el receptor a lo largo de una línea de asociación y reciben números 1.1, 1.2, etc., en el orden en el que se llaman. Si tales mensajes a su vez invocan otros mensajes, se agrega otro punto decimal y otro número al número que etiqueta dichos mensajes para indicar un anidado posterior del pase de mensaje.

En la figura A1.10 se ve que el mensaje **mouseClicked** invoca los métodos *getFigureAt()* y luego *highlight()*. El mensaje *highligh t()* invoca otros tres mensajes: *setColor()* (establecer color), *drawRect()* (dibujar recta) y *drawstring()* (dibujar cadena). La numeración en cada etiqueta muestra el anidado y la naturaleza secuencial de cada mensaje.

Existen muchas características opcionales que pueden agregarse a las etiquetas de flecha. Por ejemplo, puede preceder el número con una letra. Una flecha entrante podría etiquetarse **A1: mouseClicked(point)**, lo que indica una hebra de ejecución, A. Si otros mensajes se ejecutan en otras hebras, su etiqueta estaría precedida por una letra diferente. Por ejemplo, si el método *mouseClicked()* se ejecuta en la hebra A, pero crea una nueva hebra B e invoca *highlight()* en dicha hebra, entonces la flecha desde **MouseListener** hacia **Figure** se etiquetaría **1.B2: highlight(graphics)**.

Si el lector está interesado en mostrar las relaciones entre los objetos, además de los mensajes que se envíen entre ellos, probablemente el diagrama de comunicación es una mejor opción que el diagrama de secuencia. Si está más interesado en el orden temporal del paso de mensajes, entonces un diagrama de secuencia probablemente es mejor.

Diagramas de actividad

Un *diagrama de actividad* UML muestra el comportamiento dinámico de un sistema o de parte de un sistema a través del flujo de control entre acciones que realiza el sistema. Es similar a un diagrama de flujo, excepto porque un diagrama de actividad puede mostrar flujos concurrentes.

El componente principal de un diagrama de actividad es un nodo *acción*, representado mediante un rectángulo redondeado, que corresponde a una tarea realizada por el sistema de software. Las flechas desde un nodo acción hasta otro indican el flujo de control; es decir, una flecha entre dos nodos acción significa que, después de completar la primera acción, comienza la segunda acción. Un punto negro sólido forma el *nodo inicial* que indica el punto de inicio de la actividad. Un punto negro rodeado por un círculo negro es el *nodo final* que indica el fin de la actividad.

Un *tenedor* (*fork*) representa la separación de actividades en dos o más actividades concurrentes. Se dibuja como una barra negra horizontal con una flecha apuntando hacia ella y dos o más flechas apuntando en sentido opuesto. Cada flecha continua representa un flujo de control que puede ejecutarse de manera concurrente con los flujos correspondientes a las otras flechas continuas. Dichas actividades concurrentes pueden realizarse en una computadora, usando diferentes hebras o incluso diferentes computadoras.

La figura A1.11 muestra un ejemplo de diagrama de actividad que involucra hornear un pastel. El primer paso es encontrar la receta. Una vez encontrada pueden medirse los ingredientes secos y líquidos, mezclarse y precalentar el horno. La mezcla de los ingredientes secos puede hacerse en paralelo con la mezcla de los ingredientes líquidos y el precalentado del horno.

Una *unión* (*join*) es una forma de sincronizar flujos de control concurrentes. Se representa mediante una barra negra horizontal con dos o más flechas entrantes y una flecha saliente. El

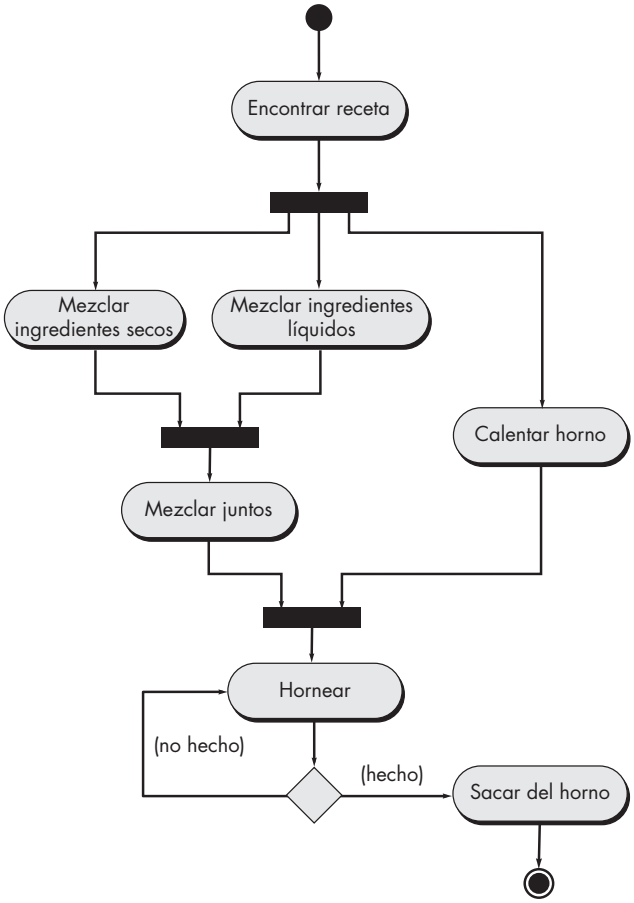


FIGURA A1.11
Diagrama de actividad UML que muestra cómo hornear un pastel

flujo de control representado por la flecha saliente no puede comenzar la ejecución hasta que todos los flujos representados por las flechas entrantes se hayan completado. En la figura A1.11 se tiene una unión antes de la acción de mezclar en conjunto los ingredientes líquidos y secos. Esta unión indica que todos los ingredientes secos deben mezclarse y que debe hacerse lo mismo con todos los ingredientes líquidos antes de poder combinar las dos mezclas. La segunda unión en la figura indica que, antes de comenzar a hornear el pastel, todos los ingredientes deben mezclarse juntos y el horno debe estar a la temperatura correcta.

Un nodo de *decisión* corresponde a una rama en el flujo de control con base en una condición. Tal nodo se despliega como un triángulo blanco con una flecha entrante y dos o más flechas salientes. Cada flecha saliente se etiqueta con una guardia (una condición dentro de corchetes). El flujo de control sigue la flecha saliente cuya guardia es verdadera. Es recomendable asegurarse de que las condiciones cubran todas las posibilidades, de modo que exactamente una de ellas sea verdadera cada vez que se llegue a un nodo de decisión. La figura A1.11 muestra un nodo de decisión que sigue al horneado del pastel. Si el pastel está hecho, entonces se saca del horno. De otro modo, se hornea un poco más.

Una de las cosas que no dice el diagrama de actividad de la figura A1.11 es quién o qué hace cada una de las acciones. Con frecuencia, no importa la división exacta de la mano de obra. Pero si quiere indicar cómo se dividen las acciones entre los participantes, puede decorar el diagrama de actividad con "canales", como se muestra en la figura A1.12. Los *canales*, como el nombre

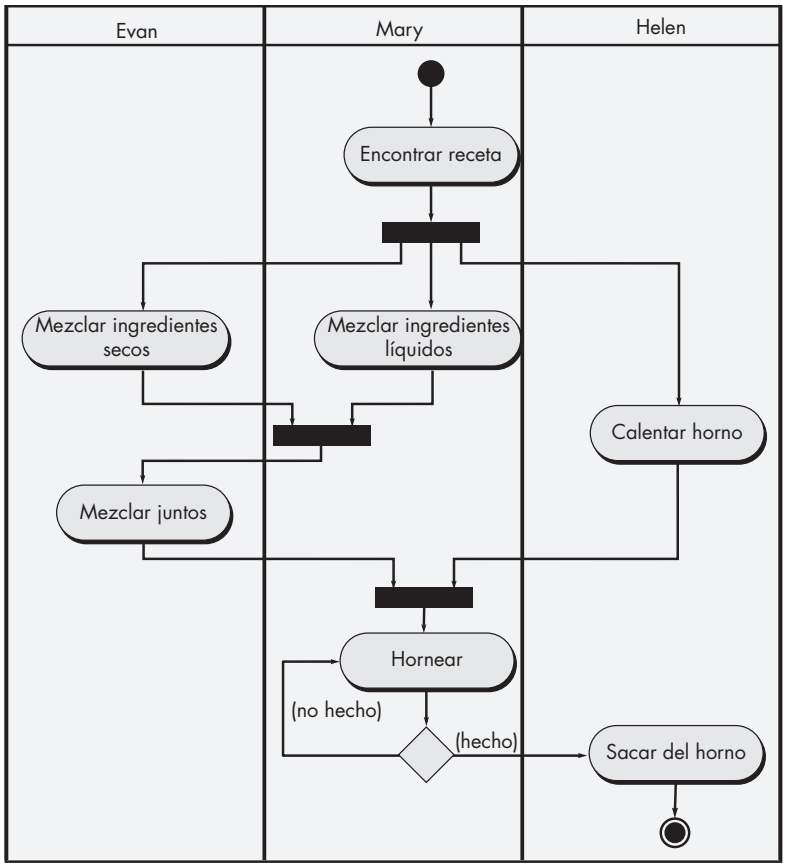


FIGURA A1.12
Diagrama de actividad de horneado de pastel con "carriles de natación" agregados

implica, se forman dividiendo el diagrama en tiras o "carriles" (como si fuera una alberca con carriles de natación), cada uno de los cuales corresponde a uno de los participantes. Todas las acciones en un carril las realiza el participante correspondiente. En la figura A1.12, Evan es responsable de la mezcla de los ingredientes secos y, luego, de mezclar juntos los ingredientes secos y los líquidos; Helen es responsable de calentar el horno y sacar el pastel; y Mary es responsable de todo lo demás.

DIAGRAMAS DE ESTADO

El comportamiento de un objeto en un punto particular en el tiempo con frecuencia depende del estado del objeto; es decir, de los valores de sus variables en dicho momento. Como ejemplo trivial, considere un objeto con una variable de instancia booleana. Cuando se pide realizar una operación, el objeto puede hacer una cosa si dicha variable es *verdadera* y hacer algo más si es *falsa*.

Un *diagrama de estado* UML modela los estados de un objeto, las acciones que se realizan dependiendo de dichos estados y las transiciones entre los estados del objeto.

Como ejemplo, considere el diagrama de estado para una parte de un compilador Java. La entrada al compilador es un archivo de texto, que puede considerarse como una larga cadena de caracteres. El compilador lee caracteres uno a uno y a partir de ellos determina la estructura del programa. Una pequeña parte de este proceso de lectura de caracteres involucra ignorar

caracteres de "espacio blanco" (por ejemplo, los caracteres *espacio*, *tabulador*, *línea nueva* y *retorno*) y caracteres dentro de un comentario.

Suponga que el compilador delega a un **EliminadorEspacioBlancoyComentario** la labor de avanzar sobre los caracteres de espacio blanco y sobre los caracteres dentro de un comentario, es decir, la labor de dicho objeto es leer los caracteres de entrada hasta que todos los caracteres de espacio blanco y comentario se leyeron, punto donde regresa el control al compilador para leer y procesar caracteres no en blanco y no de comentario. Piense en la manera en la que el objeto **EliminadorEspacioBlancoyComentario** lee los caracteres y determina si el siguiente carácter es de espacio blanco o parte de un comentario. El objeto puede verificar los espacios blancos, probando los siguientes caracteres contra " ", "\t", "\n" y "\r". ¿Pero cómo determina si el siguiente carácter es parte de un comentario? Por ejemplo, cuando ve "/" por primera vez, todavía no sabe si dicho carácter representa un operador división, parte del operador /= o el comienzo de una línea o bloque de comentario. Para hacer esta determinación, **EliminadorEspacioBlancoyComentario** necesita anotar el hecho de que vio una "/" y luego moverse hacia el siguiente carácter. Si el carácter siguiente a "/" es otra "/" o un "*", entonces **EliminadorEspacioBlancoyComentario** sabe que ahora lee un comentario y puede avanzar al final del comentario sin procesar o guardar algún carácter. Si el carácter siguiente al primer "/" es distinto a "/" o a "*", entonces **EliminadorEspacioBlancoyComentario** sabe que "/" representa el operador división o parte del operador /= y, por tanto, avanza a través de los caracteres.

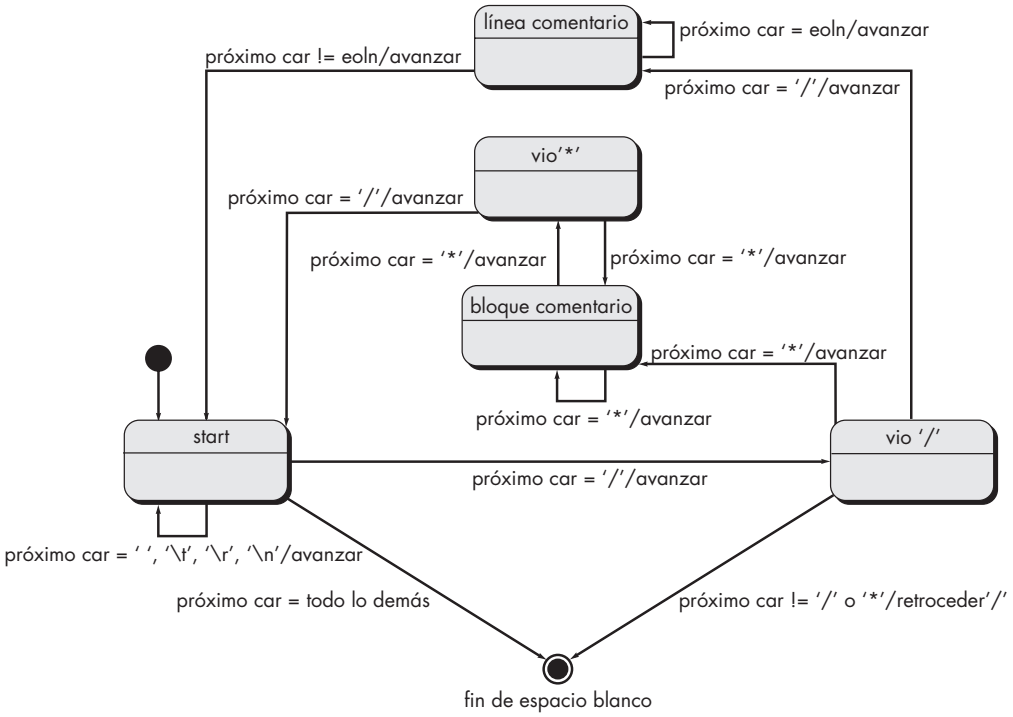
En resumen, conforme **EliminadorEspacioBlancoyComentario** lee los caracteres, necesita seguir la pista de muchas cosas, incluido si el carácter actual es espacio blanco, si el carácter previo que lee fue "/", si actualmente lee caracteres en un comentario, si llegó al final del comentario, etc. Todos éstos corresponden a diferentes estados del objeto **EliminadorEspacioBlancoyComentario**. En cada uno de estos estados, **EliminadorEspacioBlancoyComentario** se comporta de manera diferente con respecto al siguiente carácter que lee.

Para ayudarlo a visualizar todos los estados de este objeto y la manera en la que cambia de estado, puede usar un diagrama de estado UML, como se muestra en la figura A1.13. Un diagrama de estado muestra los estados mediante rectángulos redondeados, cada uno de los cuales tiene un nombre en su mitad superior. También existe un círculo llamado "pseudoestado inicial", que en realidad no es un estado y en vez de ello sólo apunta al estado inicial. En la figura A1.13, el estado **start** es el estado inicial. Las flechas de un estado a otro estado indican transiciones o cambios en el estado del objeto. Cada transición se etiqueta con un evento disparador, una diagonal (/) y una actividad. Todas las partes de las etiquetas de transición son opcionales en los diagramas de estado. Si el objeto está en un estado y el evento disparador ocurre para una de sus transiciones, entonces se realiza dicha actividad de transición y el objeto toma un nuevo estado, indicado por la transición. Por ejemplo, en la figura A1.13, si el objeto **EliminadorEspacioBlancoyComentario** está en el estado **start** y el siguiente carácter es "/", entonces **EliminadorEspacioBlancoyComentario** avanza desde dicho carácter y cambia al estado **vio '/'**. Si el carácter después de "/" es otra "/", entonces el objeto avanza al estado **línea comentario** y permanece ahí hasta que lee un carácter de fin de línea. Si en vez de ello el siguiente carácter después de "/" es "*", entonces el objeto avanza al estado **bloque comentario** y permanece ahí hasta que ve otro "*" seguido por un "/", que indica el final del bloque comentario. Estudie el diagrama para asegurarse de que lo entiende. Observe que, después de avanzar por el espacio en blanco o por un comentario, **EliminadorEspacioBlancoyComentario** regresa al estado **start** y comienza de nuevo. Dicho comportamiento es necesario, pues puede haber varios comentarios sucesivos o caracteres de espacio en blanco antes de cualquier otro carácter en el código fuente Java.

Un objeto puede transitar a un estado final, lo que se indica mediante un círculo negro con un círculo blanco alrededor de él, lo que indica que ya no hay más transiciones. En la figura

FIGURA A1.13

Diagrama de estado para avanzar sobre espacio blanco y comentarios en Java



A1.13, el objeto **EliminadorEspacioBlancoyComentario** termina cuando el siguiente carácter no es espacio en blanco o parte de un comentario. Observe que todas las transiciones, excepto las dos que conducen al estado final, tienen actividades que consisten en avanzar al siguiente carácter. Las dos transiciones hacia el estado final no avanzan sobre el siguiente carácter porque el siguiente carácter es parte de una palabra o símbolo de interés para el compilador. Observe que, si el objeto está en el estado **vio '/'**, pero el siguiente carácter no es "/" o "*", entonces "/" es un operador división o parte del operador /= y, por tanto, no se quiere avanzar. De hecho, se quiere regresar un carácter para hacer el "/" en el siguiente carácter, de modo que "/" pueda usarse por parte del compilador. En la figura A1.13, esta actividad se etiqueta como retroceder '/'.

Un diagrama de estado le ayudará a descubrir situaciones perdidas o inesperadas, es decir, con un diagrama de estado, es relativamente sencillo garantizar que todos los posibles eventos disparadores para todos los estados posibles se representaron. Por ejemplo, en la figura A1.13, puede verificar fácilmente que cada estado incluyó transiciones para todos los posibles caracteres.

Los diagramas de estado UML pueden contener muchas otras características no incluidas en la figura A1.13. Por ejemplo, cuando un objeto está en un estado, por lo general no hace más que sentarse y esperar que ocurra un evento disparador. Sin embargo, existe un tipo especial de estado, llamado *estado de actividad*, donde el objeto realiza alguna actividad, llamada *hacer actividad*, mientras está en dicho estado. Para indicar que un estado es un estado de actividad en el diagrama de estado, se incluye, en la mitad inferior del rectángulo redondeado del estado, la frase "do/" seguida por la actividad que debe realizar mientras está en dicho estado. El "hacer actividad" puede terminar antes de que ocurra cualquier transición de estado, después de lo cual el estado de actividad se comporta como un estado normal de espera. Si una transición del estado de actividad ocurre antes de terminar el "hacer actividad", entonces se interrumpe el "hacer actividad".

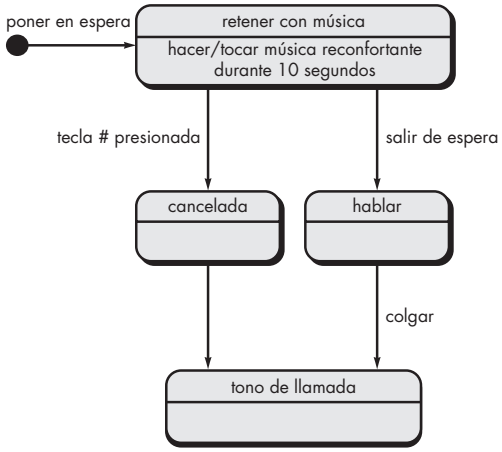


FIGURA A1.14
Diagrama de estado con un estado de actividad y una transición sin disparador

Puesto que un evento disparador es opcional cuando ocurre una transición, es posible que ningún evento disparador pueda mencionarse como parte de una etiqueta de transición. En tales casos, para estados de espera normales, el objeto inmediatamente transitará de dicho estado al nuevo estado. Para estados de actividad, tal transición se realiza tan pronto como termina el "hacer actividad".

La figura A1.14 ilustra esta situación, usando los estados para la operación de un teléfono. Cuando un llamador se coloca en espera, la llamada pasa al estado **retener con música** (la música reconfortante suena durante 10 segundos). Después de 10 segundos, el "hacer actividad" del estado se completa y el estado se comporta como un estado normal de no actividad. Si el llamador presiona la tecla # cuando la llamada está en el estado **retener con música**, la llamada transita hacia el estado **cancelado** e inmediatamente al estado **tono de llamada**. Si la tecla # se presiona antes de completar los 10 segundos de música reconfortante, el "hacer actividad" se interrumpe y la música cesa inmediatamente.

LENGUAJE DE RESTRICCIÓN DE OBJETO (PANORAMA)

La gran variedad de diagramas disponibles como parte de UML le brindan un rico conjunto de formas representativas para el modelo de diseño. Sin embargo, con frecuencia son suficientes las representaciones gráficas. Acaso necesite un mecanismo para representar información de manera explícita y formal que restrinja algún elemento del modelo de diseño. Desde luego, es posible describir las restricciones en lenguaje natural como el inglés, pero este enfoque invariablemente conduce a inconsistencia y ambigüedad. Por esta razón, parece apropiado un lenguaje más formal, extraído de la teoría de conjuntos y de los lenguajes de especificación formales (vea el capítulo 21), pero que tenga la sintaxis un tanto menos matemática de un lenguaje de programación.

El *lenguaje de restricción de objeto* (LRO) complementa al UML, permitiéndole usar una gramática y sintaxis formales para construir enunciados sin ambigüedades acerca de varios elementos del modelo de diseño (por ejemplo, clases y objetos, eventos, mensajes, interfaces). Los enunciados LRO más simples se construyen en cuatro partes: 1) un *contexto* que define la situación limitada en la que es válido el enunciado, 2) una *propiedad* que representa algunas características del contexto (por ejemplo, una propiedad puede ser un atributo si el contexto es una clase), 3) una *operación* (por ejemplo, aritmética, orientada a conjunto) que manipule o califique

una propiedad y 4) palabras clave (por ejemplo, **if, then, else, and, or, not, implies**) que se usan para especificar expresiones condicionales.

Como ejemplo simple de una expresión LRO, considere el sistema de impresión que se estudió en el capítulo 10. La condición guardia se coloca en el evento **jobCostAccepted** que causa una transición entre los estados *computingJobCost* y *formingJob* dentro del diagrama de estado para el objeto **PrintJob** (figura 10.9). En el diagrama (figura 10.9), la condición guardia se expresa en lenguaje natural e implica que la autorización sólo puede ocurrir si el cliente está autorizado para aprobar el costo del trabajo. En LRO, la expresión puede tomar la forma:

```
customer
   self.authorizationAuthority = 'yes'
```

donde un atributo booleano, **authorizationAuthority**, de la clase (en realidad una instancia específica de la clase) llamada **Customer** debe establecerse en "sí" para que se satisfaga la condición guardia.

Conforme se crea el modelo de diseño, con frecuencia existen instancias en las que deben satisfacerse pre o poscondiciones antes de completar alguna acción especificada por el diseño. LRO proporciona una poderosa herramienta para especificar pre y poscondiciones de manera formal. Como ejemplo, considere una extensión al sistema local de impresión (que se estudió como ejemplo en el capítulo 10) donde el cliente proporcione un límite de costo superior (upper cost bound) para el trabajo de impresión y una fecha de entrega "fatal", al mismo tiempo que se especifican otras características del trabajo de impresión. Si las estimaciones de costo y entrega superan dichos límites, el trabajo no se presenta y debe notificársele al cliente. En LRO, un conjunto de pre y poscondiciones puede especificarse de la forma siguiente:

```
context PrintJob::validate(upperCostBound : Integer, custDeliveryReq :
   Integer)
 pre: upperCostBound > 0
   and custDeliveryReq > 0
   and self.jobAuthorization = 'no'
 post: if self.totalJobCost <= upperCostBound
       and self.deliveryDate <= custDeliveryReq
     then
      self.jobAuthorization = 'yes'
     endif
```

Este enunciado LRO define una invariante (**inv**): condiciones que deben existir antes (pre) y después (pos) de algún comportamiento. Inicialmente, una precondición establece que el límite de costo y la fecha de entrega deben especificarse por parte del cliente, y la autorización debe establecerse en "no". Después de determinar costos y entrega, se aplica la poscondición especificada. También debe observarse que la expresión:

```
self.jobAuthorization = 'yes'
```

no asigna el valor "sí", sino que declara que **jobAuthorization** debe establecerse en "sí" para cuando la operación termine. Una descripción completa del LRO está más allá del ámbito de este apéndice. La especificación LRO completa puede obtenerse en **www.omg.org/technology/documents/formal/ocl.htm**

Lecturas y fuentes de información adicionales

Decenas de libros estudian UML. Los que abordan la versión más reciente incluyen: Miles y Hamilton (*Learning UML 2.0,* O'Reilly Media, Inc., 2006); Booch, Rumbaugh, y Jacobson (*Unified Modeling Language User*

Guide, 2a. ed., Addison-Wesley, 2005), Ambler (*The Elements of UML 2.0 Style,* Cambridge University Press, 2005), y Pilone y Pitman (*UML 2.0 in a Nutshell,* O'Reilly Media, Inc., 2005).

En internet está disponible una gran variedad de fuentes de información acerca del UML en el modelado de ingeniería del software. Una lista actualizada de referencias en la World Wide Web puede encontrarse bajo "análisis" y "diseño" en el sitio del libro: **www.mhhe.com/engcs/compsci/pressman/professional/olc/ser.htm**