1: Introducción a los objetos

La génesis de la revolución de los computadores se encontraba en una máquina, y por ello, la génesis de nuestros lenguajes de programación tiende a parecerse a esa máquina.

Pero los computadores, más que máquinas, pueden considerarse como herramientas que permiten ampliar la mente ("bicicletas para la mente", como se enorgullece de decir Steve Jobs), además de un medio de expresión inherentemente diferente. Como resultado, las herramientas empiezan a parecerse menos a máquinas y más a partes de nuestra mente, al igual que ocurre con otros medios de expresión como la escritura, la pintura, la escultura, la animación o la filmación de películas. La programación orientada a objetos (POO) es una parte de este movimiento dirigido a utilizar los computadores como si de un medio de expresión se tratara.

Este capítulo introducirá al lector en los conceptos básicos de la POO, incluyendo un repaso a los métodos de desarrollo. Este capítulo y todo el libro, toman como premisa que el lector ha tenido experiencia en algún lenguaje de programación procedural (por procedimientos), sea C u otro lenguaje. Si el lector considera que necesita una preparación mayor en programación y/o en la sintaxis de C antes de enfrentarse al presente libro, se recomienda hacer uso del CD ROM formativo *Thinking in C: Foundations for C++ and Java*, que se adjunta con el presente libro, y que puede encontrarse también la URL, *http://www.BruceEckel.com*.

Este capítulo contiene material suplementario, o de trasfondo (*background*). Mucha gente no se siente cómoda cuando se enfrenta a la programación orientada a objetos si no entiende su contexto, a grandes rasgos, previamente. Por ello, se presentan aquí numerosos conceptos con la intención de proporcionar un repaso sólido a la POO. No obstante, también es frecuente encontrar a gente que no acaba de comprender los conceptos hasta que tiene acceso a los mecanismos; estas personas suelen perderse si no se les ofrece algo de código que puedan manipular. Si el lector se siente identificado con este último grupo, estará ansioso por tomar contacto con el lenguaje en sí, por lo que debe sentirse libre de saltarse este capítulo —lo cual no tiene por qué influir en la comprensión que finalmente se adquiera del lenguaje o en la capacidad de escribir programas en él mismo. Sin embargo, tarde o temprano tendrá necesidades ocasionales de volver aquí, para completar sus nociones en aras de lograr una mejor comprensión de la importancia de los objetos y de la necesidad de comprender cómo acometer diseños haciendo uso de ellos.

El progreso de la abstracción

Todos los lenguajes de programación proporcionan abstracciones. Puede incluso afirmarse que la complejidad de los problemas a resolver es directamente proporcional a la clase (tipo) y calidad de las abstracciones a utilizar, entendiendo por tipo "clase", *aquello que se desea abstraer*. El lenguaje

ensamblador es una pequeña abstracción de la máquina subyacente. Muchos de los lenguajes denominados "imperativos" desarrollados a continuación del antes mencionado ensamblador (como Fortran, BASIC y C) eran abstracciones a su vez del lenguaje citado. Estos lenguajes supusieron una gran mejora sobre el lenguaje ensamblador, pero su abstracción principal aún exigía pensar en términos de la estructura del computador más que en la del problema en sí a resolver. El programador que haga uso de estos lenguajes debe establecer una asociación entre el modelo de la máquina (dentro del "espacio de la solución", que es donde se modela el problema, por ejemplo, un computador) y el modelo del problema que de hecho trata de resolver (en el "espacio del problema", que es donde de hecho el problema existe). El esfuerzo necesario para establecer esta correspondencia, y el hecho de que éste no es intrínseco al lenguaje de programación, es causa directa de que sea difícil escribir programas, y de que éstos sean caros de mantener, además de fomentar, como efecto colateral (lateral), toda una la industria de "métodos de programación".

La alternativa al modelado de la máquina es modelar el problema que se trata de resolver. Lenguajes primitivos como LISP o APL eligen vistas parciales o particulares del mundo (considerando respectivamente que los problemas siempre se reducen a "listas" o a "algoritmos"). PROLOG convierte todos los problemas en cadenas de decisiones. Los lenguajes se han creado para programación basada en limitaciones o para programar exclusivamente mediante la manipulación de símbolos gráficos (aunque este último caso resultó ser excesivamente restrictivo). Cada uno de estos enfoques constituye una solución buena para determinadas clases (tipos) de problemas (aquéllos para cuya solución fueron diseñados), pero cuando uno trata de sacarlos de su dominio resultan casi impracticables.

El enfoque orientado a objetos trata de ir más allá, proporcionando herramientas que permitan al programador representar los elementos en el espacio del problema. Esta representación suele ser lo suficientemente general como para evitar al programador limitarse a ningún tipo de problema específico. Nos referiremos a elementos del espacio del problema, denominando "objetos" a sus representaciones dentro del espacio de la solución (por supuesto, también serán necesarios otros objetos que no tienen su análogo dentro del espacio del problema). La idea es que el programa pueda autoadaptarse al lingo del problema simplemente añadiendo nuevos tipos de objetos, de manera que la mera lectura del código que describa la solución constituya la lectura de palabras que expresan el problema. Se trata, en definitiva, de una abstracción del lenguaje mucho más flexible y potente que cualquiera que haya existido previamente. Por consiguiente, la POO permite al lector describir el problema en términos del propio problema, en vez de en términos del sistema en el que se ejecutará el programa final. Sin embargo, sigue existiendo una conexión con el computador, pues cada objeto puede parecer en sí un pequeño computador; tiene un estado, y se le puede pedir que lleve a cabo determinadas operaciones. No obstante, esto no quiere decir que nos encontremos ante una mala analogía del mundo real, al contrario, los objetos del mundo real también tienen características y comportamientos.

Algunos diseñadores de lenguajes han dado por sentado que la programación orientada a objetos, de por sí, no es adecuada para resolver de manera sencilla todos los problemas de programación, y hacen referencia al uso de lenguajes de programación *multiparadigma*[1].

[1] N. del autor: Ver *Multiparadigm Programming in Leda*, por Timothy Budd (Addison-Wesley, 1995).

Alan Kay resumió las cinco características básicas de Smalltalk, el primer lenguaje de programación orientado a objetos que tuvo éxito, además de uno de los lenguajes en los que se basa Java. Estas características constituyen un enfoque puro a la programación orientada a objetos:

1. **Todo es un objeto**. Piense en cualquier objeto como una variable: almacena datos, permite que se le "hagan peticiones", pidiéndole que desempeñe por sí mismo determinadas operaciones, etc. En teoría, puede acogerse cualquier componente conceptual del problema a resolver (bien sean perros, edificios, servicios, etc.) y representarlos como objetos dentro de un programa.

2. **Un programa es un cúmulo de objetos que se dicen entre sí lo que tienen que hacer mediante el envío de mensajes**. Para hacer una petición a un objeto, basta con "enviarle un mensaje". Más concretamente, puede considerarse que un mensaje en sí es una petición para solicitar una llamada a una función que pertenece a un objeto en particular.

3. **Cada objeto tiene su propia memoria, constituida por otros objetos**. Dicho de otra manera, uno crea una nueva clase de objeto construyendo un paquete que contiene objetos ya existentes. Por consiguiente, uno puede incrementar la complejidad de un programa, ocultándola tras la simplicidad de los propios objetos.

4. **Todo objeto es de algún tipo**. Cada objeto es *un elemento de una clase*, entendiendo por "clase" un sinónimo de "tipo". La característica más relevante de una clase la constituyen "el conjunto de mensajes que se le pueden enviar".

5. **Todos los objetos de determinado tipo pueden recibir los mismos mensajes**. Ésta es una afirmación de enorme trascendencia como se verá más tarde. Dado que un objeto de tipo "círculo" es también un objeto de tipo "polígono", se garantiza que todos los objetos "círculo" acepten mensajes propios de "polígono". Esto permite la escritura de código que haga referencia a polígonos, y que de manera automática pueda manejar cualquier elemento que encaje con la descripción de "polígono". Esta capacidad de *suplantación* es uno de los conceptos más potentes de la POO.

Todo objeto tiene una interfaz

Aristóteles fue probablemente el primero en estudiar cuidadosamente el concepto de *tipo*; hablaba de "la clase de los pescados o la clase de los peces". La idea de que todos los objetos, aún siendo únicos, son también parte de una clase de objetos, todos ellos con características y comportamientos en común, ya fue usada en el primer lenguaje orientado a objetos, Simula-67, que ya incluye la palabra clave **clase**, que permite la introducción de un nuevo tipo dentro de un programa.

Simula, como su propio nombre indica, se creó para el desarrollo de simulaciones, como el clásico del cajero de un banco, en el que hay cajero, clientes, cuentas, transacciones y unidades monetarias —un montón de "objetos". Todos los objetos que, con excepción de su estado, son idénticos durante la ejecución de un programa se agrupan en "clases de objetos", que es precisamente de donde proviene la palabra clave **clase**. La creación de tipos abstractos de datos (clases) es un concepto fun-

damental en la programación orientada a objetos. Los tipos abstractos de datos funcionan casi como los tipos de datos propios del lenguaje: es posible la creación de variables de un tipo (que se denominan *objetos* o *instancias* en el dialecto propio de la orientación a objetos) y manipular estas variables (mediante el *envío* o *recepción de mensajes*; se envía un mensaje a un objeto y éste averigua qué debe hacer con él). Los miembros (elementos) de cada clase comparten algunos rasgos comunes: toda cuenta tiene un saldo, todo cajero puede aceptar un ingreso, etc. Al mismo tiempo, cada miembro tiene su propio estado, cada cuenta tiene un saldo distinto, cada cajero tiene un nombre. Por consiguiente, los cajeros, clientes, cuentas, transacciones, etc. también pueden ser representados mediante una entidad única en el programa del computador. Esta entidad es el objeto, y cada objeto pertenece a una clase particular que define sus características y comportamientos.

Por tanto, aunque en la programación orientada a objetos se crean nuevos tipos de datos, virtualmente todos los lenguajes de programación orientada a objetos hacen uso de la palabra clave "clase". Siempre que aparezca la palabra clave "tipo" (*type*) puede sustituirse por "clase" (*class*) y viceversa[2].

Dado que una clase describe a un conjunto de objetos con características (datos) y comportamientos (funcionalidad) idénticos, una clase es realmente un tipo de datos, porque un número en coma flotante, por ejemplo, también tiene un conjunto de características y comportamientos. La diferencia radica en que un programador define una clase para que encaje dentro de un problema en vez de verse forzado a utilizar un tipo de datos existente que fue diseñado para representar una unidad de almacenamiento en una máquina. Es posible extender el lenguaje de programación añadiendo nuevos tipos de datos específicos de las necesidades de cada problema. El sistema de programación acepta las nuevas clases y las cuida, y asigna las comprobaciones que da a los tipos de datos predefinidos.

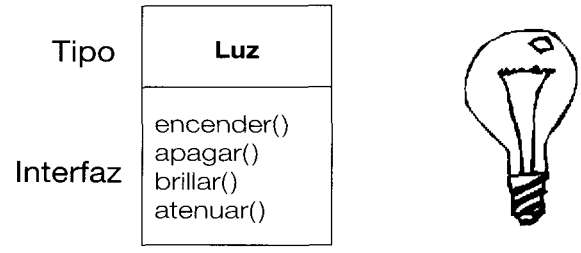
El enfoque orientado a objetos no se limita a la construcción de simulaciones. Uno puede estar de acuerdo o no con la afirmación de que todo programa es una simulación del sistema a diseñar, mientras que las técnicas de POO pueden reducir de manera sencilla un gran conjunto de problemas a una solución simple.

Una vez que se establece una clase, pueden construirse tantos objetos de esa clase como se desee, y manipularlos como si fueran elementos que existen en el problema que se trata de resolver. Sin duda, uno de los retos de la programación orientada a objetos es crear una correspondencia uno a uno entre los elementos del espacio del problema y los objetos en el espacio de la solución.

Pero, ¿cómo se consigue que un objeto haga un trabajo útil para el programador? Debe de haber una forma de hacer peticiones al objeto, de manera que éste desempeñe alguna tarea, como completar una transacción, dibujar algo en la pantalla o encender un interruptor. Además, cada objeto sólo puede satisfacer determinadas peticiones. Las peticiones que se pueden hacer a un objeto se encuentran definidas en su *interfaz*, y es el tipo de objeto el que determina la interfaz. Un ejemplo sencillo sería la representación de una bombilla:

---

[2] Algunas personas establecen una distinción entre ambos, remarcando que un tipo determina la interfaz, mientras que una clase es una implementación particular de una interfaz.




```
Luz lz = new Luz();
lz.encender();
```

La interfaz establece *qué* peticiones pueden hacerse a un objeto particular. Sin embargo, debe hacer código en algún lugar que permita satisfacer esas peticiones. Éste, junto con los datos ocultos, constituye la *implementación*. Desde el punto de vista de un lenguaje de programación procedural, esto no es tan complicado. Un tipo tiene una función asociada a cada posible petición, de manera que cuando se hace una petición particular a un objeto, se invoca a esa función. Este proceso se suele simplificar diciendo que se ha "enviado un mensaje" (hecho una petición) a un objeto, y el objeto averigua qué debe hacer con el mensaje (ejecuta el código).

Aquí, el nombre del tipo o clase es **Luz**, el nombre del objeto **Luz** particular es **lz**, y las peticiones que pueden hacerse a una **Luz** son encender, apagar, brillar o atenuar. Es posible crear una **Luz** definiendo una "referencia" (**lz**) a ese objeto e invocando a **new** para pedir un nuevo objeto de ese tipo. Para enviar un mensaje al objeto, se menta el nombre del objeto y se conecta al mensaje de petición mediante un punto. Desde el punto de vista de un usuario de una clase predefinida, éste es el no va más de la programación con objetos.

El diagrama anteriormente mostrado sigue el formato del Lenguaje de Modelado Unificado o *Unified Modeling Language* (UML). Cada clase se representa mediante una caja, en la que el nombre del tipo se ubica en la parte superior, cualquier dato necesario para describirlo se coloca en la parte central, y las *funciones miembro* (las funciones que pertenecen al objeto) en la parte inferior de la caja. A menudo, solamente se muestran el nombre de la clase y las funciones miembro públicas en los diagramas de diseño UML, ocultando la parte central. Si únicamente interesan los nombres de las clases, tampoco es necesario mostrar la parte inferior de la caja.

La implementación oculta

Suele ser de gran utilidad descomponer el tablero de juego en *creadores de clases* (elementos que crean nuevos tipos de datos) y *programadores clientes*[3] (consumidores de clases que hacen uso de los tipos de datos en sus aplicaciones). La meta del programador cliente es hacer uso de un gran repertorio de clases que le permitan acometer el desarrollo de aplicaciones de manera rápida. La

[3] Nota del autor: Término acuñado por mi amigo Scott Meyers.

meta del creador de clases es construir una clase que únicamente exponga lo que es necesario al programador cliente, manteniendo oculto todo lo demás. ¿Por qué? Porque aquello que esté oculto no puede ser utilizado por el programador cliente, lo que significa que el creador de la clase puede modificar la parte oculta a su voluntad, sin tener que preocuparse de cualquier impacto que esta modificación pueda implicar. La parte oculta suele representar las interioridades de un objeto que podrían ser corrompidas por un programador cliente poco cuidadoso o ignorante, de manera que mientras se mantenga oculta su implementación se reducen los errores en los programas.

En cualquier relación es importante determinar los límites relativos a todos los elementos involucrados. Al crear una biblioteca, se establece una relación con el programador cliente, que es también un programador, además de alguien que está construyendo una aplicación con las piezas que se encuentran en esta biblioteca, quizás con la intención de construir una biblioteca aún mayor.

Si todos los miembros de una clase están disponibles para todo el mundo, el programador cliente puede hacer cualquier cosa con esa clase y no hay forma de imponer reglas. Incluso aunque prefiera que el programador cliente no pueda manipular alguno de los miembros de su clase, sin control de accesos, no hay manera de evitarlo. Todo se presenta desnudo al mundo.

Por ello, la primera razón que justifica el control de accesos es mantener las manos del programador cliente apartadas de las porciones que no deba manipular —partes que son necesarias para las maquinaciones internas de los tipos de datos pero que no forman parte de la interfaz que los usuarios necesitan en aras de resolver sus problemas particulares. De hecho, éste es un servicio a los usuarios que pueden así ver de manera sencilla aquello que es sencillo para ellos, y qué es lo que pueden ignorar.

La segunda razón para un control de accesos es permitir al diseñador de bibliotecas cambiar el funcionamiento interno de la clase sin tener que preocuparse sobre cómo afectará al programador cliente. Por ejemplo, uno puede implementar una clase particular de manera sencilla para simplificar el desarrollo y posteriormente descubrir que tiene la necesidad de reescribirla para que se ejecute más rápidamente. Si tanto la interfaz como la implementación están claramente separadas y protegidas, esto puede ser acometido de manera sencilla.

Java usa tres palabras clave explícitas para establecer los límites en una clase: **public**, **private** y **protected**. Su uso y significado son bastante evidentes. Estos *modificadores de acceso* determinan quién puede usar las definiciones a las que preceden. La palabra **public** significa que las definiciones siguientes están disponibles para todo el mundo. El término **private**, por otro lado, significa que nadie excepto el creador del tipo puede acceder a esas definiciones. Así, **private** es un muro de ladrillos entre el creador y el programador cliente. Si alguien trata de acceder a un miembro **private**, obtendrá un error en tiempo de compilación. La palabra clave **protected** actúa como **private**, con la excepción de que una clase heredada tiene acceso a miembros **protected** pero no a los **private**. La herencia será definida algo más adelante.

Java también tiene un "acceso por defecto", que se utiliza cuando no se especifica ninguna de las palabras clave descritas en el párrafo anterior. Este modo de acceso se suele denominar "amistoso" o *friendly* porque implica que las clases pueden acceder a los miembros amigos de otras clases que

estén en el mismo *package* o paquete, sin embargo, fuera del paquete, estos miembros amigos se convierten en **private**.

Reutilizar la implementación

Una vez que se ha creado y probado una clase, debería (idealmente) representar una unidad útil de código. Resulta que esta reutilización no es siempre tan fácil de lograr como a muchos les gustaría; producir un buen diseño suele exigir experiencia y una visión profunda de la problemática. Pero si se logra un diseño bueno, parece suplicar ser reutilizado. La reutilización de código es una de las mayores ventajas que proporcionan los lenguajes de programación orientados a objetos.

La manera más simple de reutilizar una clase es simplemente usar un objeto de esa clase directamente, pero también es posible ubicar un objeto de esa clase dentro de otra clase. Esto es lo que se denomina la "creación de un objeto miembro ". La nueva clase puede construirse a partir de un número indefinido de otros objetos, de igual o distinto tipo, en cualquier combinación necesaria para lograr la funcionalidad deseada dentro de la nueva clase. Dado que uno está construyendo una nueva clase a partir de otras ya existentes, a este concepto se le denomina *composición* (o, de forma más general, *agregación*). La composición se suele representar mediante una relación "es-parte-de", como en "el motor es una parte de un coche" ("carro" en Latinoamérica).

(Este diagrama UML indica la composición, y el rombo relleno indica que hay un coche. Normalmente, lo representaré simplemente mediante una línea, sin el diamante, para indicar una asociación[4].)

La composición conlleva una gran carga de flexibilidad. Los objetos miembros de la nueva clase suelen ser privados, haciéndolos inaccesibles a los programadores cliente que hagan uso de la clase. Esto permite cambiar los miembros sin que ello afecte al código cliente ya existente. También es posible cambiar los objetos miembros en tiempo de ejecución, para así cambiar de manera dinámica el comportamiento de un programa. La herencia, que se describe a continuación, no tiene esta flexibilidad, pues el compilador debe emplazar restricciones de tiempo de compilación en las clases que se creen mediante la herencia.

Dado que la herencia es tan importante en la programación orientada a objetos, casi siempre se enfatiza mucho su uso, de manera que un programador novato puede llegar a pensar que hay que ha-

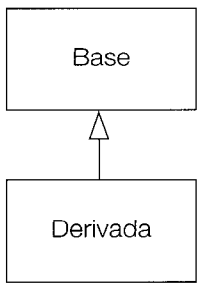
[4] Éste ya suele ser un nivel de detalle suficiente para la gran mayoría de diagramas, de manera que no es necesario indicar de manera explícita si se está utilizando agregación o composición.

cer uso de la misma en todas partes. Este pensamiento puede provocar que se elaboren diseños poco elegantes y desmesuradamente complicados. Por el contrario, primero sería recomendable intentar hacer uso de la composición, mucho más simple y sencilla. Siguiendo esta filosofía se lograrán diseños mucho más limpios. Una vez que se tiene cierto nivel de experiencia, la detección de los casos que precisan de la herencia se convierte en obvia.

Herencia: reutilizar la interfaz

Por sí misma, la idea de objeto es una herramienta más que buena. Permite empaquetar datos y funcionalidad juntos por *concepto*, de manera que es posible representar cualquier idea del espacio del problema en vez de verse forzado a utilizar idiomas propios de la máquina subyacente. Estos conceptos se expresan como las unidades fundamentales del lenguaje de programación haciendo uso de la palabra clave **class**.

Parece una pena, sin embargo, acometer todo el problema para crear una clase y posteriormente verse forzado a crear una nueva que podría tener una funcionalidad similar. Sería mejor si pudiéramos hacer uso de una clase ya existente, clonarla, y después hacer al "clon" las adiciones y modificaciones que sean necesarias. Efectivamente, esto se logra mediante la *herencia*, con la excepción de que si se cambia la clase original (denominada la *clase base*, *clase super* o *clase padre*), el "clon" modificado (denominado *clase derivada*, *clase heredada*, *subclase* o *clase hijo*) también reflejaría esos cambios.

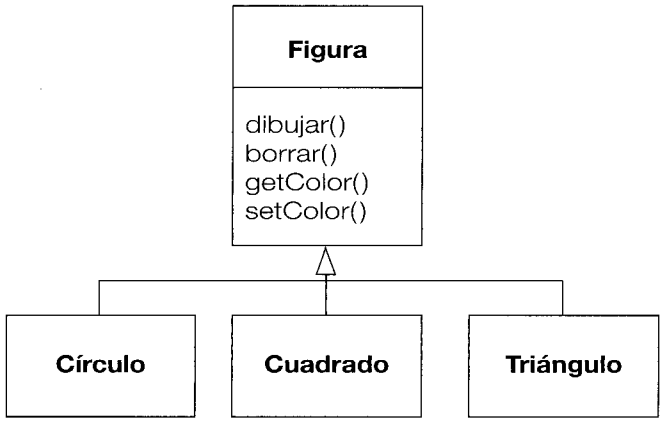


(La flecha de este diagrama UML apunta de la clase derivada a la clase base. Como se verá, puede haber más de una clase derivada.)

Un tipo hace más que definir los límites de un conjunto de objetos; también tiene relaciones con otros tipos. Dos tipos pueden tener características y comportamientos en común, pero un tipo puede contener más características que otro y también puede manipular más mensajes (o gestionarlos de manera distinta). La herencia expresa esta semejanza entre tipos haciendo uso del concepto de tipos base y tipos derivados. Un tipo base contiene todas las características y comportamientos que comparten los tipos que de él se derivan. A partir del tipo base, es posible derivar otros tipos para expresar las distintas maneras de llevar a cabo esta idea.

Por ejemplo, una máquina de reciclaje de basura clasifica los desperdicios. El tipo base es "basura", y cada desperdicio tiene su propio peso, valor, etc. y puede ser fragmentado, derretido o descompuesto. Así, se derivan tipos de basura más específicos que pueden tener características adicionales (una botella tiene un color), o comportamientos (el aluminio se puede modelar, una lata de acero tiene capacidades magnéticas). Además, algunos comportamientos pueden ser distintos (el valor del papel depende de su tipo y condición). El uso de la herencia permite construir una jerarquía de tipos que expresa el problema que se trata de resolver en términos de los propios tipos.

Un segundo ejemplo es el clásico de la "figura geométrica" utilizada generalmente en sistemas de diseño por computador o en simulaciones de juegos. El tipo base es "figura" y cada una de ellas tiene un tamaño, color, posición, etc. Cada figura puede dibujarse, borrarse, moverse, colorearse, etc. A partir de ésta, se pueden derivar (heredar) figuras específicas: círculos, cuadrados, triángulos, etc., pudiendo tener cada uno de los cuales características y comportamientos adicionales. Algunos comportamientos pueden ser distintos, como pudiera ser el cálculo del área de los distintos tipos de figuras. La jerarquía de tipos engloba tanto las similitudes como las diferencias entre las figuras.



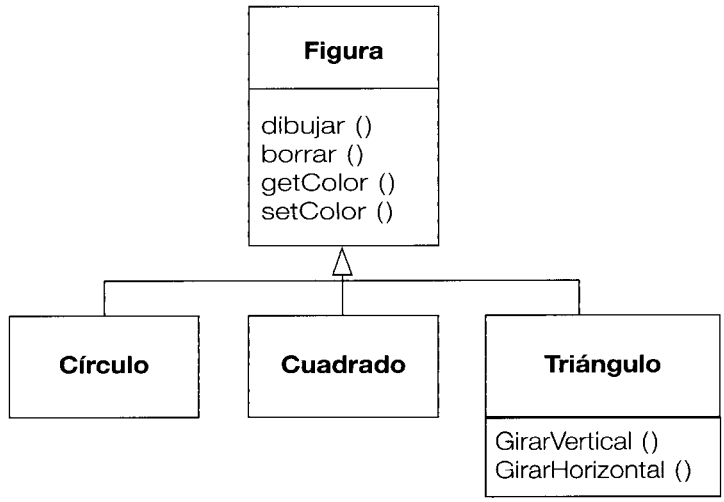
Representar la solución en los mismos términos que el problema es tremendamente beneficioso puesto que no es necesario hacer uso de innumerables modelos intermedios para transformar una descripción del problema en una descripción de la solución. Con los objetos, el modelo principal lo constituye la jerarquía de tipos, de manera que se puede ir directamente de la descripción del sistema en el mundo real a la descripción del sistema en código. Sin duda, una de las dificultades que tiene la gente con el diseño orientado a objetos es la facilidad con que se llega desde el principio al final: las mentes entrenadas para buscar soluciones completas suelen verse aturdidas inicialmente por esta simplicidad.

Al heredar a partir de un tipo existente, se crea un nuevo tipo. Este nuevo tipo contiene no sólo los miembros del tipo existente (aunque los datos privados "**private**" están ocultos e inaccesibles) sino lo que es más importante, duplica la interfaz de la clase base. Es decir, todos los mensajes que pueden ser enviados a objetos de la clase base también pueden enviarse a los objetos de la clase derivada. Dado que sabemos el tipo de una clase en base a los mensajes que se le pueden enviar, la cla-

se derivada *es del mismo tipo que la clase base*. Así, en el ejemplo anterior, "un círculo en una figura". Esta equivalencia de tipos vía herencia es una de las pasarelas fundamentales que conducen al entendimiento del significado de la programación orientada a objetos.

Dado que, tanto la clase base como la derivada tienen la misma interfaz, debe haber alguna implementación que vaya junto con la interfaz. Es decir, debe haber algún código a ejecutar cuando un objeto recibe un mensaje en particular. Si simplemente se hereda la clase sin hacer nada más, los métodos de la interfaz de la clase base se trasladan directamente a la clase derivada. Esto significa que los objetos de la clase derivada no sólo tienen el mismo tipo sino que tienen el mismo comportamiento, aunque este hecho no es particularmente interesante.

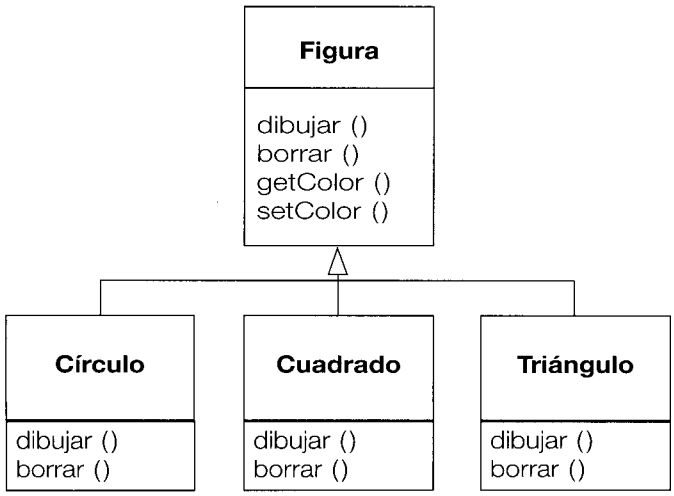
Hay dos formas de diferenciar una clase nueva derivada de la clase base original. El primero es bastante evidente: se añaden nuevas funciones a la clase derivada. Estas funciones nuevas no forman parte de la interfaz de la clase base, lo que significa que la clase base simplemente no hacía todo lo que ahora se deseaba, por lo que fue necesario añadir nuevas funciones. Ese uso simple y primitivo de la herencia es, en ocasiones, la solución perfecta a los problemas. Sin embargo, debería considerarse detenidamente la posibilidad de que la clase base llegue también a necesitar estas funciones adicionales. Este proceso iterativo y de descubrimiento de su diseño es bastante frecuente en la programación orientada a objetos.



Aunque la herencia puede implicar en ocasiones (especialmente en Java, donde la palabra clave que hace referencia a la misma es **extends**) que se van a añadir funcionalidades a una interfaz, esto no tiene por qué ser siempre así. La segunda y probablemente más importante manera de diferenciar una nueva clase es *variar* el comportamiento de una función ya existente en la clase base. A esto se le llama redefinición[5] (anulación o superposición) de la función.

Para redefinir una función simplemente se crea una nueva definición de la función dentro de la clase derivada. De esta manera puede decirse que "se está utilizando la misma función de la interfaz pero se desea que se comporte de manera distinta dentro del nuevo tipo".

[5] En el original *overriding* (N. del T.).

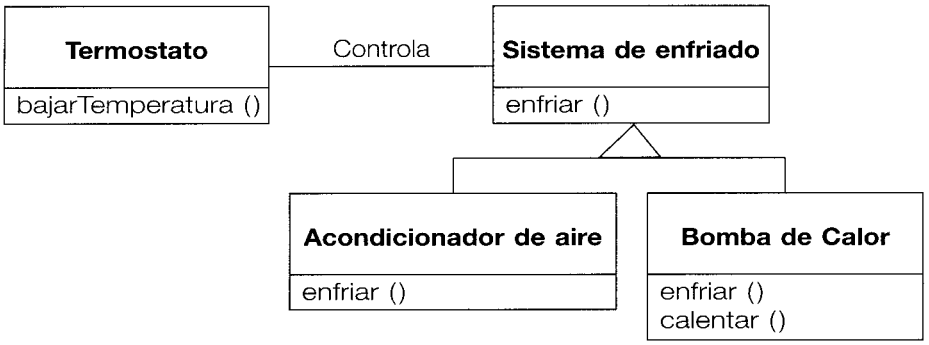


La relación es-un frente a la relación es-como-un

Es habitual que la herencia suscite un pequeño debate: ¿debería la herencia superponer *sólo* las funciones de la clase base (sin añadir nuevas funciones miembro que no se encuentren en ésta)? Esto significaría que el tipo derivado sea *exactamente* el mismo tipo que el de la clase base, puesto que tendría exactamente la misma interfaz. Consecuentemente, es posible sustituir un objeto de la clase derivada por otro de la clase base. A esto se le puede considerar *sustitución pura*, y a menudo se le llama el *principio de sustitución*. De cierta forma, ésta es la manera ideal de tratar la herencia. Habitualmente, a la relación entre la clase base y sus derivadas que sigue esta filosofía se le denomina relación *es-un*, pues es posible decir que "un círculo *es un* polígono". Una manera de probar la herencia es determinar si es posible aplicar la relación *es-un* a las clases en liza, y tiene sentido.

Hay veces en las que es necesario añadir nuevos elementos a la interfaz del tipo derivado, extendiendo así la interfaz y creando un nuevo tipo. Éste puede ser también sustituido por el tipo base, pero la sustitución no es perfecta pues las nuevas funciones no serían accesibles desde el tipo base. Esta relación puede describirse como la relación *es-como-un*[6]; el nuevo tipo tiene la interfaz del viejo pero además contiene otras funciones, así que no se puede decir que sean exactamente iguales. Considérese por ejemplo un acondicionador de aire. Supongamos que una casa está cableada y tiene las botoneras para refrescarla, es decir, tiene una interfaz que permite controlar la temperatura. Imagínese que se estropea el acondicionador de aire y se reemplaza por una bomba de calor que puede tanto enfriar como calentar. La bomba de calor *es-como-un* acondicionador de aire, pero puede hacer más funciones. Dado que el sistema de control de la casa está diseñado exclusivamente para controlar el enfriado, se encuentra restringido a la comunicación con la parte "enfriadora" del nuevo objeto. Es necesario extender la interfaz del nuevo objeto, y el sistema existente únicamente conoce la interfaz original.

[6] Término acuñado por el autor.



Por supuesto, una vez que uno ve este diseño, está claro que la clase base "sistema de enfriado" no es lo suficientemente general, y debería renombrarse a "sistema de control de temperatura" de manera que también pueda incluir calentamiento —punto en el que el principio de sustitución funcionará. Sin embargo, este diagrama es un ejemplo de lo que puede ocurrir en el diseño y en el mundo real.

Cuando se ve el principio de sustitución es fácil sentir que este principio (la sustitución pura) es la única manera de hacer las cosas, y de hecho, *es* bueno para los diseños que funcionen así. Pero hay veces que está claro que hay que añadir nuevas funciones a la interfaz de la clase derivada. Con la experiencia, ambos casos irán pareciendo obvios.

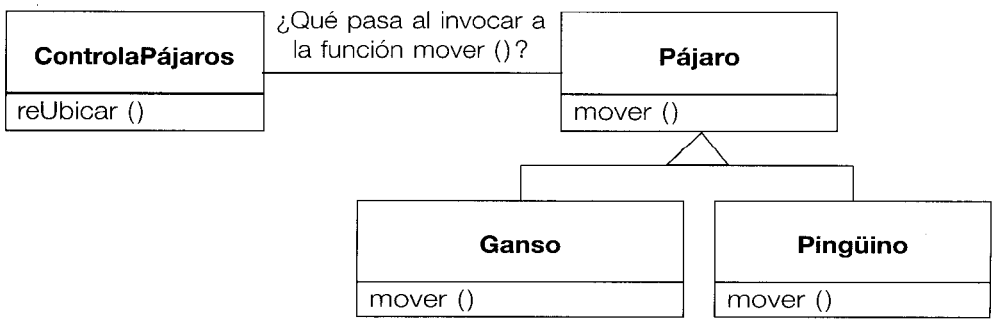
Objetos intercambiables con polimorfismo

Al tratar con las jerarquías de tipos, a menudo se desea tratar un objeto no como el tipo específico del que es, sino como su tipo base. Esto permite escribir código que no depende de tipos específicos. En el ejemplo de los polígonos, las funciones manipulan polígonos genéricos sin que importe si son círculos, cuadrados, triángulos o cualquier otro polígono que no haya sido aún definido. Todos los polígonos pueden dibujarse, borrarse y moverse, por lo que estas funciones simplemente envían un mensaje a un objeto polígono; no se preocupan de qué base hace este objeto con el mensaje.

Este tipo de código no se ve afectado por la adición de nuevos tipos, y esta adición es la manera más común de extender un programa orientado a objetos para que sea capaz de manejar nuevas situaciones. Por ejemplo, es posible derivar un nuevo subtipo de un polígono denominado pentágono sin tener que modificar las funciones que solamente manipulan polígonos genéricos. Esta capacidad para extender un programa de manera sencilla derivando nuevos subtipos es importante, ya que mejora considerablemente los diseños a la vez que reduce el coste del mantenimiento de software.

Sin embargo, hay un problema a la hora de tratar los objetos de tipos derivados como sus tipos base genéricos (los círculos como polígonos, las bicicletas como vehículos, los cormoranes como pájaros, etc.). Si una función va a decir a un polígono genérico que la dibuje, o a un vehículo genérico que se engrane, o a un pájaro genérico que se mueva, el compilador no puede determinar en tiempo de

compilación con exactitud qué fragmento de código se ejecutará. Éste es el punto clave —cuando se envía el mensaje, el programador no *quiere* saber qué fragmento de código se ejecutará; la función dibujar se puede aplicar de manera idéntica a un círculo, un cuadrado o un triángulo, y el objeto ejecutará el código correcto en función de su tipo específico. Si no es necesario saber qué fragmento de código se ejecutará, al añadir un nuevo subtipo, el código que ejecute puede ser diferente sin necesidad de modificar la llamada a la función. Por consiguiente, el compilador no puede saber exactamente qué fragmento de código se está ejecutando, ¿y qué es entonces lo que hace? Por ejemplo, en el diagrama siguiente, el objeto **ControlaPájaros** trabaja con objetos **Pájaro** genéricos, y no sabe exactamente de qué tipo son. Esto es conveniente para la perspectiva de **ControlaPájaros** pues no tiene que escribir código especial para determinar el tipo exacto de **Pájaro** con el que está trabajando, ni el comportamiento de ese **Pájaro**. Entonces, ¿cómo es que cuando se invoca a **mover()** ignorando el tipo específico de **Pájaro** se dará el comportamiento correcto (un **Ganso** corre, vuela o nada, y un **Pingüino** corre o nada)?



La respuesta es una de las principales novedades en la programación orientada a objetos: el compilador no puede hacer una llamada a función en el sentido tradicional. La llamada a función generada por un compilador no-POO hace lo que se denomina una *ligadura temprana*, un término que puede que el lector no haya visto antes porque nunca pensó que se pudiera hacer de otra forma. Esto significa que el compilador genera una llamada a una función con nombre específico, y el montador resuelve esta llamada a la dirección absoluta del código a ejecutar. En POO, el programa no puede determinar la dirección del código hasta tiempo de ejecución, por lo que es necesario otro esquema cuando se envía un mensaje a un objeto genérico.

Para resolver el problema, los lenguajes orientados a objetos usan el concepto de *ligadura tardía*. Al enviar un mensaje a un objeto, no se determina el código invocado hasta tiempo de ejecución. El compilador se asegura de que la función exista y hace la comprobación de tipos de los argumentos y del valor de retorno (un lenguaje en el que esto no se haga así se dice que es *débilmente tipificado*), pero se desconoce el código exacto a ejecutar.

Para llevar a cabo la ligadura tardía, Java utiliza un fragmento de código especial en vez de la llamada absoluta. Este código calcula la dirección del cuerpo de la función utilizando información almacenada en el objeto (este proceso se relata con detalle en el Capítulo 7). Por consiguiente, cada objeto puede comportarse de manera distinta en función de los contenidos de ese fragmento de có-

digo especial. Cuando se envía un mensaje a un objeto, éste, de hecho, averigua qué es lo que debe hacer con ese mensaje.

En algunos lenguajes (C++ en particular) debe establecerse explícitamente que se desea que una función tenga la flexibilidad de las propiedades de la ligadura tardía. En estos lenguajes, por defecto, la correspondencia con las funciones miembro no se establece dinámicamente, por lo que es necesario recordar que hay que añadir ciertas palabras clave extra para lograr el polimorfismo.

Considérese el ejemplo de los polígonos. La familia de clases (todas ellas basadas en la misma interfaz uniforme) ya fue representada anteriormente en este capítulo. Para demostrar el polimorfismo, se escribirá un único fragmento de código que ignora los detalles específicos de tipo y solamente hace referencia a la clase base. El código está *desvinculado* de información específica del tipo, y por consiguiente es más fácil de escribir y entender. Y si se añade un nuevo tipo —por ejemplo un **Hexágono**— mediante herencia, el código que se escriba trabajará tan perfectamente con el nuevo **Polígono** como lo hacía con los tipos ya existentes, y por consiguiente, el programa es *ampliable*. Si se escribe un método en Java (y pronto aprenderá el lector a hacerlo):

```
void hacerAlgo (Poligono p) {
p.borrar();
// ...
p.dibujar();
}
```

Esta función se entiende con cualquier **Polígono**, y por tanto, es independiente del tipo específico de objeto que esté dibujando y borrando. En cualquier otro fragmento del programa se puede usar la función **hacerAlgo( )**:

```
Circulo c = new Circulo();
Triangulo t = new Triangulo ();
Linea l = new Linea();
hacerAlgo(c);
hacerAlgo(t);
hacerAlgo(l);
```

Las llamadas a **hacerAlgo( )** trabajan correctamente, independientemente del tipo de objeto.

De hecho, éste es un truco bastante divertido. Considérese la línea:

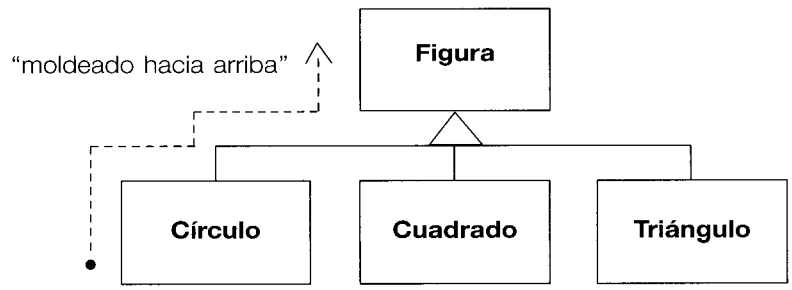
```
hacerAlgo(c);
```

Lo que está ocurriendo es que se está pasando un **Círculo** a una función que espera un **Polígono**. Como un **Círculo** *es un* **Polígono**, puede ser tratado como tal por **hacerAlgo( )**. Es decir, cualquier mensaje que **hacerAlgo( )** pueda enviar a un **Polígono**, también podrá aceptarlo un **Círculo**. Por tanto, obrar así es algo totalmente seguro y lógico.

A este proceso de tratar un tipo derivado como si fuera el tipo base se le llama conversión de tipos (*moldeado*) *hacia arriba*[7]. El nombre moldear (cast) se utiliza en el sentido de moldear (convertir) un

[7] En el original inglés, *casting*.

molde, y es *hacia arriba* siguiendo la manera en que se representa en los diagramas la herencia, con el tipo base en la parte superior y las derivadas colgando hacia abajo. Por consiguiente, hacer un moldeado (*casting*) a la clase base es moverse hacia arriba por el diagrama de herencias: *moldeado hacia arriba*.



Un programa orientado a objetos siempre tiene algún moldeado hacia arriba pues ésta es la manera de desvincularse de tener que conocer el tipo exacto con que se trabaja en cada instante. Si se echa un vistazo al código de **hacerAlgo( )**:

```
p.borrar();
// ...
p.dibujar();
```

Obsérvese que no se dice "caso de ser un **Círculo**, hacer esto; caso de ser un **Cuadrado**, hacer esto otro, etc.". Si se escribe código que compruebe todos los tipos posibles que puede ser un **Polígono**, el tipo de código se complica, además de hacerse necesario modificarlo cada vez que se añade un nuevo tipo de **Polígono**. Aquí, simplemente se dice que "en el caso de los polígonos, se sabe que es posible aplicarles las operaciones de **borrar( )** y **dibujar( )**, eso sí, teniendo en cuenta todos los detalles de manera correcta".

Lo que más llama la atención del código de **hacerAlgo( )** es que, de alguna manera, se hace lo correcto. Invocar a **dibujar( )** para **Círculo** hace algo distinto que invocar a **dibujar( )** para **Cuadrado** o **Línea**, pero cuando se envía el mensaje **dibujar( )** a un **Polígono** anónimo, se da el comportamiento correcto basándose en el tipo actual de **Polígono**. Esto es sorprendente porque, como se mencionó anteriormente, cuando el compilador de Java está compilando el código de **hacerAlgo( )**, no puede saber exactamente qué tipos está manipulando. Por ello, generalmente, se espera que acabe invocando a la versión de **borrar( )** y **dibujar( )** de la clase base **Polígono** y no a las específicas de **Círculo**, **Cuadrado** y **Línea**. Y sigue ocurriendo lo correcto por el polimorfismo. El compilador y el sistema de tiempo real se hacen cargo de los detalles; todo lo que hace falta saber es qué ocurre, y lo que es más importante, cómo diseñar haciendo uso de ello. Al enviar un mensaje a un objeto, el objeto hará lo correcto, incluso cuando se vea involucrado el moldeado hacia arriba.

Clases base abstractas e interfaces

A menudo es deseable que la clase base *únicamente* presente una interfaz para sus clases derivadas. Es decir, no se desea que nadie cree objetos de la clase base, sino que sólo se hagan moldeados ha-

cia arriba de la misma de manera que se pueda usar su interfaz. Esto se logra convirtiendo esa clase en *abstracta* usando la palabra clave **abstract**. Si alguien trata de construir un objeto de una clase **abstracta** el compilador lo evita. Esto es una herramienta para fortalecer determinados diseños.

También es posible utilizar la palabra clave **abstract** para describir un método que no ha sido aún implementado —indicando "he aquí una función interfaz para todos los tipos que se hereden de esta clase, pero hasta la fecha no existe una implementación de la misma". Se puede crear un método **abstracto** sólo dentro de una clase abstracta. Cuando se hereda la clase, debe implementarse el método o de lo contrario también la clase heredada se convierte en **abstracta**. La creación de métodos **abstractos** permite poner un método en una interfaz sin verse forzado a proporcionar un fragmento de código, posiblemente sin significado, para ese método.

La palabra clave **interface** toma el concepto de clase **abstracta** un paso más allá, evitando totalmente las definiciones de funciones. La **interfaz** es una herramienta muy útil y utilizada, ya que proporcionar la separación perfecta entre interfaz e implementación. Además, si se desea, es posible combinar muchos elementos juntos mientras que no es posible heredar de múltiples clases normales o abstractas.

Localización de objetos y longevidad

Técnicamente, la POO consiste simplemente en tipos de datos abstractos, herencia y polimorfismo, aunque también hay otros aspectos no menos importantes. El resto de esta sección trata de analizar esos aspectos.

Uno de los factores más importantes es la manera de crear y destruir objetos. ¿Dónde están los datos de un objeto y cómo se controla su longevidad (tiempo de vida)? En este punto hay varias filosofías de trabajo. En C++ el enfoque principal es el control de la eficiencia, proporcionando una alternativa al programador. Para lograr un tiempo de ejecución óptimo, es posible determinar el espacio de almacenamiento y la longevidad en tiempo de programación, ubicando los objetos en la pila (creando las variables *scoped* o *automatic*) o en el área de almacenamiento estático. De esta manera se prioriza la velocidad de la asignación y liberación de espacio de almacenamiento, cuyo control puede ser de gran valor en determinadas situaciones. Sin embargo, se sacrifica en flexibilidad puesto que es necesario conocer la cantidad exacta de objetos, además de su longevidad y su tipo, mientras se escribe el programa. Si se está tratando de resolver un problema más general como un diseño asistido por computador, la gestión de un almacén o el control de tráfico aéreo, este enfoque resulta demasiado restrictivo.

El segundo enfoque es crear objetos dinámicamente en un espacio de memoria denominado el montículo o montón (*heap*). En este enfoque, no es necesario conocer hasta tiempo de ejecución el número de objetos necesario, cuál es su longevidad o a qué tipo exacto pertenecen. Estos aspectos se determinarán justo en el preciso momento en que se ejecute el programa. Si se necesita un nuevo objeto, simplemente se construye en el *montículo* en el instante en que sea necesario. Dado que el almacenamiento se gestiona dinámicamente, en tiempo de ejecución, la cantidad de tiempo necesaria

para asignar espacio de almacenamiento en el montículo es bastante mayor que el tiempo necesario para asignar espacio a la pila. (La creación de espacio en la pila suele consistir simplemente en una instrucción al ensamblador que mueve hacia abajo el puntero de pila, y otra para moverlo de nuevo hacia arriba.) El enfoque dinámico provoca generalmente el pensamiento lógico de que los objetos tienden a ser complicados, por lo que la sobrecarga debida a la localización de espacio de almacenamiento y su liberación no deberían tener un impacto significativo en la creación del objeto. Es más, esta mayor flexibilidad es esencial para resolver en general el problema de programación.

Java utiliza exclusivamente el segundo enfoque[8]. Cada vez que se desea crear un objeto se usa la palabra clave **new** para construir una instancia dinámica de ese objeto.

Hay otro aspecto, sin embargo, a considerar: la longevidad de un objeto. Con los lenguajes que permiten la creación de objetos en la pila, el compilador determina cuánto dura cada objeto y puede destruirlo cuando no es necesario. Sin embargo, si se crea en el montículo, el compilador no tiene conocimiento alguno sobre su longevidad. En un lenguaje como C++ hay que determinar en tiempo de programación cuándo destruir el objeto, lo cual puede conducir a fallos de memoria si no se hace de manera correcta (y este problema es bastante común en los programas en C++). Java proporciona un recolector de basura que descubre automáticamente cuándo se ha dejado de utilizar un objeto, que puede, por consiguiente, ser destruido. Un recolector de basura es muy conveniente, al reducir el número de aspectos a tener en cuenta, así como la cantidad de código a escribir. Y lo que es más importante, el recolector de basura proporciona un nivel de seguridad mucho mayor contra el problema de los fallos de memoria (que ha hecho abandonar más de un proyecto en C++).

El resto de esta sección se centra en factores adicionales relativos a la longevidad de los objetos y su localización.

Colecciones e iteradores

Si se desconoce el número de objetos necesarios para resolver un problema en concreto o cuánto deben durar, también se desconocerá cómo almacenar esos objetos. ¿Cómo se puede saber el espacio a reservar para los mismos? De hecho, no se puede, pues esa información se desconocerá hasta tiempo de ejecución.

La solución a la mayoría de problemas de diseño en la orientación a objetos parece sorprendente: se crea otro tipo de objeto. El nuevo tipo de objeto que resuelve este problema particular tiene referencias a otros objetos. Por supuesto, es posible hacer lo mismo con un array, disponible en la mayoría de lenguajes. Pero hay más. Este nuevo objeto, generalmente llamado *contenedor* (llamado también *colección*, pero la biblioteca de Java usa este término con un sentido distinto así que este libro empleará la palabra "contenedor"), se expandirá a sí mismo cuando sea necesario para albergar cuanto se coloque dentro del contenedor. Simplemente se crea el objeto contenedor, y él se encarga de los detalles.

Afortunadamente, un buen lenguaje POO viene con un conjunto de contenedores como parte del propio lenguaje. En C++, es parte de la Biblioteca Estándar C++ (Standard C++ Library), que en oca-

[8] Los tipos primitivos, de los que se hablará más adelante, son un caso especial.

siones se denomina la *Standard Template Library, Biblioteca de plantillas estándar,* (STL). Object Pascal tiene contenedores en su *Visual Component Library* (VCL). Smalltalk tiene un conjunto de contenedores muy completo, y Java también tiene contenedores en su biblioteca estándar. En algunas bibliotecas, se considera que un contenedor genérico es lo suficientemente bueno para todas las necesidades, mientras que en otras (como en Java, por ejemplo) la biblioteca tiene distintos tipos de contenedores para distintas necesidades: un vector (denominado en Java **ArrayList**) para acceso consistente a todos los elementos, y una lista enlazada para inserciones consistentes en todos los elementos, por ejemplo, con lo que es posible elegir el tipo particular que satisface las necesidades de cada momento. Las bibliotecas de contenedores también suelen incluir conjuntos, colas, tablas de *hasing*, árboles, pilas, etc.

Todos los contenedores tienen alguna manera de introducir y extraer cosas; suele haber funciones para añadir elementos a un contenedor, y otras para extraer de nuevo esos elementos. Pero sacar los elementos puede ser más problemático porque una función de selección única suele ser restrictiva. ¿Qué ocurre si se desea manipular o comparar un conjunto de elementos del contenedor y no uno sólo?

La solución es un iterador, que es un objeto cuyo trabajo es seleccionar los elementos de dentro de un contenedor y presentárselos al usuario del iterador. Como clase, también proporciona cierto nivel de abstracción. Esta abstracción puede ser usada para separar los detalles del contenedor del código al que éste está accediendo. El contenedor, a través del iterador, se llega a abstraer hasta convertirse en una simple secuencia, que puede ser recorrida gracias al iterador sin tener que preocuparse de la estructura subyacente —es decir, sin preocuparse de si es un **ArrayList** (lista de arrays), un **LinkedList** (lista enlazado), un **Stack**, (pila) u otra cosa. Esto proporciona la flexibilidad de cambiar fácilmente la estructura de datos subyacente sin que el código de un programa se vea afectado. Java comenzó (en las versiones 1.0 y 1.1) con un iterador estándar denominado **Enumeration**, para todas sus clases contenedor. Java 2 ha añadido una biblioteca mucho más completa de contenedores que contiene un iterador denominado **Iterator** mucho más potente que el antiguo **Enumeration**.

Desde el punto de vista del diseño, todo lo realmente necesario es una secuencia que puede ser manipulada en aras de resolver un problema. Si una secuencia de un sólo tipo satisface todas las necesidades de un problema, entonces no es necesario hacer uso de distintos tipos. Hay dos razones por las que es necesaria una selección de contenedores. En primer lugar, los contenedores proporcionan distintos tipos de interfaces y comportamientos externos. Una pila tiene una interfaz y un comportamiento distintos del de las colas, que son a su vez distintas de los conjuntos y las listas. Cualquiera de éstos podría proporcionar una solución mucho más flexible a un problema. En segundo lugar, distintos contenedores tienen distintas eficiencias en función de las operaciones. El mejor ejemplo está en **ArrayList** y **LinkedList**. Ambos son secuencias sencillas que pueden tener interfaces y comportamientos externos idénticos. Pero determinadas operaciones pueden tener costes radicalmente distintos. Los accesos aleatorios a **ArrayList** tienen tiempos de acceso constante; se invierte el mismo tiempo independientemente del elemento seleccionado. Sin embargo, en una **LinkedList** moverse de elemento en elemento a lo largo de la lista para seleccionar uno al azar es altamente costoso, y es necesario muchísimo más tiempo para localizar un elemento cuanto más adelante se encuentre. Por otro lado, si se desea insertar un elemento en el medio de una secuencia, es mucho menos costoso hacerlo en un **LinkedList** que en un **ArrayList**. Ésta y otras operaciones tienen eficiencias distintas en función de la estructura de la secuencia subyacente. En la fase de diseño, podría comenzarse con una **LinkedList** y, al primar el rendimiento, cambiar a un

ArrayList. Dado que la abstracción se lleva a cabo a través de iteradores, es posible cambiar de uno a otro con un impacto mínimo en el código.

Finalmente, debe recordarse que un contenedor es sólo un espacio de almacenamiento en el que colocar objetos. Si este espacio resuelve todas las necesidades, no importa realmente cómo está implementado (concepto compartido con la mayoría de tipos de objetos). Si se está trabajando en un entorno de programación que tiene una sobrecarga inherente debido a otros factores, la diferencia de costes entre **ArrayList** y **LinkedList** podría no importar. Con un único tipo de secuencia podría valer. Incluso es posible imaginar la abstracción contenedora "perfecta", que pueda cambiar su implementación subyacente automáticamente en función de su uso.

La jerarquía de raíz única

Uno de los aspectos de la POO que se ha convertido especialmente prominente desde la irrupción de C++ es si todas las clases en última instancia deberían ser heredadas de una única clase base. En Java (como virtualmente en todos los lenguajes POO) la respuesta es "sí" y el nombre de esta última clase base es simplemente **Object**. Resulta que los beneficios de una jerarquía de raíz única son enormes.

Todos los objetos en una jerarquía de raíz única tienen una interfaz en común, por lo que en última instancia son del mismo tipo. La alternativa (proporcionada por C++) es el desconocimiento de que todo pertenece al mismo tipo fundamental. Desde el punto de vista de la retrocompatibilidad, esto encaja en el modelo de C mejor, y puede pensarse que es menos restrictivo, pero cuando se desea hacer programación orientada a objetos pura, es necesario proporcionar una jerarquía completa para lograr el mismo nivel de conveniencia intrínseco a otros lenguajes POO. Y en cualquier nueva biblioteca de clases que se adquiera, se utilizará alguna interfaz incompatible. Hacer funcionar esta nueva interfaz en un diseño lleva un gran esfuerzo (y posiblemente herencia múltiple). Así que ¿merece la pena la "flexibilidad" extra de C++? Si se necesita —si se dispone de una gran cantidad de código en C— es más que valiosa. Si se empieza de cero, otras alternativas, como Java, resultarán mucho más productivas.

Puede garantizarse que todos los objetos de una jerarquía de raíz única (como la proporcionada por Java) tienen cierta funcionalidad. Se sabe que es posible llevar a cabo ciertas operaciones básicas con todos los objetos del sistema. Una jerarquía de raíz única, junto con la creación de todos los objetos en el montículo, simplifica enormemente el paso de argumentos (uno de los temas más complicados de C++).

Una jerarquía de raíz única simplifica muchísimo la implementación de un recolector de basura (incluido en Java). El soporte necesario para el mismo puede instalarse en la clase base, y el recolector de basura podrá así enviar los mensajes apropiados a todos los objetos del sistema. Si no existiera este tipo de jerarquía ni la posibilidad de manipular un objeto a través de referencias, sería muy difícil implementar un recolector de basura.

Dado que está garantizado que en tiempo de ejecución la información de tipos está en todos los objetos, jamás será posible encontrar un objeto cuyo tipo no pueda ser determinado. Esto es especial-

mente importante con operaciones a nivel de sistema, tales como el manejo de excepciones, además de proporcionar una gran flexibilidad a la hora de programar.

Bibliotecas de colecciones y soporte al fácil manejo de colecciones

Dado que un contenedor es una herramienta de uso frecuente, tiene sentido tener una biblioteca de contenedores construidos para ser reutilizados, de manera que se puede elegir uno de la estantería y enchufarlo en un programa determinado. Java proporciona una biblioteca de este tipo, que satisface la gran mayoría de necesidades.

Moldeado hacia abajo frente a plantillas/genéricos

Para lograr que estos contenedores sean reutilizables, guardan un tipo universal en Java ya mencionado anteriormente: **Object** (*Objeto*). La jerarquía de raíz única implica que todo sea un **Object**, de forma que un contenedor que tiene objetos de tipo **Object** puede contener de todo, logrando así que los contenedores sean fácil de reutilizar.

Para utilizar uno de estos contenedores, basta con añadirle referencias a objetos y finalmente preguntar por ellas. Pero dado que el contenedor sólo guarda objetos de tipo **Object**, al añadir una referencia al contenedor, se hace un moldeado hacia arriba a **Object**, perdiendo por consiguiente su identidad. Al recuperarlo, se obtiene una referencia a **Object** y no una referencia al tipo que se había introducido. ¿Y cómo se convierte de nuevo en algo útil con la interfaz del objeto que se introdujo en el contenedor?

En este caso, también se hace uso del moldeado, pero en esta ocasión en vez de hacerlo hacia arriba siguiendo la jerarquía de las herencias hacia un tipo más general, se hace hacia abajo, hacia un tipo más específico. Esta forma de moldeado se denomina moldeado hacia abajo. Con el moldeado hacia arriba, como se sabe, un **Círculo**, por ejemplo, es un tipo de **Polígono**, con lo que este tipo de moldeado es seguro, pero lo que no se sabe es si un **Object** es un **Círculo** o un **Polígono**, por lo que no es muy seguro hacer moldeado hacia abajo a menos que se sepa exactamente qué es lo que se está manipulando.

Esto no es completamente peligroso, sin embargo, dado que si se hace un moldeado hacia abajo, a un tipo erróneo, se mostrará un error de tiempo de ejecución denominado *excepción*, que se describirá en breve. Sin embargo, al recuperar referencias a objetos de un contenedor, es necesario tener alguna manera de recordar exactamente lo que son para poder llevar a cabo correctamente un moldeado hacia abajo.

El moldeado hacia abajo y las comprobaciones en tiempo de ejecución requieren un tiempo extra durante la ejecución del programa, además de un esfuerzo extra por parte del programador. ¿No tendría sentido crear, de alguna manera, el contenedor de manera que conozca los tipos que guarda, eliminando la necesidad de hacer moldeado hacia abajo y por tanto, de que aparezca algún error? La solución la constituyen los tipos parametrizados, que son clases que el compilador puede adaptar automáticamente para que trabajen con tipos determinados. Por ejemplo, con un contenedor parametrizado, el compilador podría adaptar el propio contenedor para que solamente aceptara y per-

mitiera la recuperación de Polígonos.

Los tipos parametrizados son un elemento importante en C++, en parte porque este lenguaje no tiene una jerarquía de raíz única. En C++, la palabra clave que implementa los tipos parametrizados es "template". Java actualmente no tiene tipos parametrizados pues se puede lograr lo mismo —aunque de manera complicada— explotando la unicidad de raíz de su jerarquía. Sin embargo, una propuesta actualmente en curso para implementar tipos parametrizados utiliza una sintaxis muy semejante a las plantillas (*templates*) de C++.

El dilema de las labores del hogar: ¿quién limpia la casa?

Cada objeto requiere recursos simplemente para poder existir, fundamentalmente memoria. Cuando un objeto deja de ser necesario debe ser eliminado de manera que estos recursos queden disponibles para poder reutilizarse. En situaciones de programación sencillas la cuestión de cuándo eliminar un objeto no se antoja complicada: se crea el objeto, se utiliza mientras es necesario y posteriormente debe ser destruido. Sin embargo, no es difícil encontrar situaciones en las que esto se complica.

Supóngase, por ejemplo, que se está diseñando un sistema para gestionar el tráfico aéreo de un aeropuerto. (El mismo modelo podría funcionar también para gestionar paquetes en un almacén, o un sistema de alquiler de vídeos, o una residencia canina.) A primera vista, parece simple: construir un contenedor para albergar aviones, crear a continuación un nuevo avión y ubicarlo en el contenedor (para cada avión que aparezca en la zona a controlar). En el momento de eliminación, se borra (suprime) simplemente el objeto aeroplano correcto cuando un avión sale de la zona barrida.

Pero quizás, se tiene otro sistema para guardar los datos de los aviones, datos que no requieren atención inmediata, como la función de control principal. Quizás, se trata de un registro del plan de viaje de todos los pequeños aviones que abandonan el aeropuerto. Es decir, se dispone de un segundo contenedor de aviones pequeños, y siempre que se crea un objeto avión también se introduce en este segundo contenedor si se trata de un avión pequeño. Posteriormente, algún proceso en segundo plano lleva a cabo operaciones con los objetos de este segundo contenedor cada vez que el sistema está ocioso.

Ahora el problema se complica: ¿cómo se sabe cuándo destruir los objetos? Cuando el programa principal (el controlador) ha acabado de usar el objeto, puede que otra parte del sistema lo esté usando (o lo vaya a usar en un futuro). Este problema surge en numerosísimas ocasiones, y los sistemas de programación (como C++) en los que los objetos deben de borrarse explícitamente cuando acaba de usarlos, pueden volverse bastante complejos.

Con Java, el problema de vigilar que se libere la memoria se ha implementado en el recolector de basura (aunque no incluye otros aspectos de la supresión de objetos). El recolector "sabe" cuándo se ha dejado de utilizar un objeto y libera la memoria que ocupaba automáticamente. Esto (combinado con el hecho de que todos los objetos son heredados de la clase raíz única **Object** y con la existencia de una única forma de crear objetos, en el montículo) hace que el proceso de programar en Java sea mucho más sencillo que el hacerlo en C++. Hay muchas menos decisiones que tomar y menos obstáculos que sortear.

Los recolectores de basura frente a la eficiencia y flexibilidad

Si todo esto es tan buena idea, ¿por qué no se hizo lo mismo en C++? Bien, por supuesto, hay un precio que pagar por todas estas comodidades de programación, y este precio consiste en sobrecarga en tiempo de ejecución. Como se mencionó anteriormente, en C++ es posible crear objetos en la pila, y en este caso, éstos se eliminan automáticamente (pero no se dispone de la flexibilidad de crear tantos como se desee en tiempo de ejecución). La creación de objetos en la pila es la manera más eficiente de asignar espacio a los objetos y de liberarlo. La creación de objetos en el montículo puede ser mucho más costosa. Heredar siempre de una clase base y hacer que todas las llamadas a función sean polimórficas también conlleva un pequeño peaje. Pero el recolector de basura es un problema concreto pues nunca se sabe cuándo se va a poner en marcha y cuánto tiempo conllevará su ejecución. Esto significa que hay inconsistencias en los *ratios* (velocidad) de ejecución de los programas escritos en Java, por lo que éstos no pueden ser utilizados en determinadas situaciones, como por ejemplo, cuando el tiempo de ejecución de un programa es uniformemente crítico. (Se trata de los programas denominados generalmente de tiempo real, aunque no todos los problemas de programación en tiempo real son tan rígidos.)

Los diseñadores del lenguaje C++, trataron de ganarse a los programadores de C (en lo cual tuvieron bastante éxito), no quisieron añadir nuevas características al lenguaje que pudiesen influir en la velocidad o el uso de C++ en cualquier situación en la que los programadores pudiesen decantarse por C. Se logró la meta, pero a cambio de una mayor complejidad cuando se programa en C++. Java es más simple que C++, pero a cambio es menos eficiente y en ocasiones ni siquiera aplicable. Sin embargo, para un porcentaje elevado de problemas de programación, Java es la mejor elección.

Manejo de excepciones: tratar con errores

El manejo de errores ha sido, desde el principio de la existencia de los lenguajes de programación, uno de los aspectos más difíciles de abordar. Dado que es muy complicado diseñar un buen esquema de manejo de errores, muchos lenguajes simplemente ignoran este aspecto, pasando el problema a los diseñadores de bibliotecas que suelen contestar con soluciones a medias que funcionan en la mayoría de situaciones, pero que pueden ser burladas de manera sencilla; es decir, simplemente ignorándolas. Un problema importante con la mayoría de los esquemas de tratamiento de errores es que dependen de la vigilancia del programador de cara al seguimiento de una convención preestablecida no especialmente promovida por el propio lenguaje. Si el programador no está atento —cosa que ocurre muchas veces, especialmente si se tiene prisa— es posible olvidar estos esquemas con relativa facilidad.

El manejo de excepciones está íntimamente relacionado con el lenguaje de programación y a veces incluso con el sistema operativo. Una excepción es un objeto que es "lanzado", "arrojado"[9] desde el lugar en que se produce el error, y que puede ser "capturado" por el gestor de excepción apropiado

[9] N. del traductor: en inglés se emplea el verbo *throw*.

diseñado para manejar ese tipo de error en concreto. Es como si la gestión de excepciones constituyera un cauce de ejecución diferente, paralelo, que puede tomarse cuando algo va mal. Y dado que usa un cauce de ejecución distinto, no tiene por qué interferir con el código de ejecución normal. De esta manera el código es más simple de escribir puesto que no hay que estar comprobando los errores continuamente. Además, una excepción lanzada no es como un valor de error devuelto por una función, o un indicador (bandera) que una función pone a uno para indicar que se ha dado cierta condición de error (éstos podrían ser ignorados). Una excepción no se puede ignorar, por lo que se garantiza que será tratada antes o después. Finalmente, las excepciones proporcionan una manera de recuperarse de manera segura de una situación anormal. En vez de simplemente salir, muchas veces es posible volver a poner las cosas en su sitio y reestablecer la ejecución del programa, logrando así que éstos sean mucho más robustos.

El manejo de excepciones de Java destaca entre los lenguajes de programación pues en Java, éste se encuentra imbuido desde el principio, y es obligatorio utilizarlo. Si no se escribe un código de manera que maneje excepciones correctamente, se obtendrá un mensaje de error en tiempo de compilación. Esta garantía de consistencia hace que la gestión de errores sea mucho más sencilla.

Es importante destacar el hecho de que el manejo de excepciones no es una característica orientada a objetos, aunque en los lenguajes orientados a objetos las excepciones se suelan representar mediante un objeto. El manejo de excepciones existe desde antes de los lenguajes orientados a objetos.

Multihilo

Un concepto fundamental en la programación de computadores es la idea de manejar más de una tarea en cada instante. Muchos problemas de programación requieren que el programa sea capaz de detener lo que esté haciendo, llevar a cabo algún otro problema, y volver a continuación al proceso principal. Se han buscando múltiples soluciones a este problema. Inicialmente, los programadores que tenían un conocimiento de bajo nivel de la máquina, escribían rutinas de servicio de interrupciones, logrando la suspensión del proceso principal mediante interrupciones hardware. Aunque este enfoque funcionaba bastante bien, era dificultoso y no portable, por lo que causaba que transportar un programa a una plataforma distinta de la original fuera lento y caro.

A veces, es necesario hacer uso de las interrupciones para el manejo de tareas críticas en el tiempo, pero hay una gran cantidad de problemas en los que simplemente se intenta dividir un problema en fragmentos de código que pueden ser ejecutados por separado, de manera que se logra un menor tiempo de respuesta para todo el programa en general. Dentro de un programa, estos fragmentos de código que pueden ser ejecutados por separado, se denominan hilos, y el concepto general se denomina *multihilos*. Un ejemplo común de aplicación multihilo es la interfaz de usuario. Gracias al uso de hilos, un usuario puede presionar un botón y lograr una respuesta rápida en vez de verse forzado a esperar a que el programa acabe su tarea actual.

Normalmente, los hilos no son más que una herramienta para facilitar la planificación en un monoprocesador. Pero si el sistema operativo soporta múltiples procesadores, es posible asignar cada hilo a un procesador distinto de manera que los hilos se ejecuten verdaderamente en paralelo. Uno de los aspectos más destacables de la programación multihilo es que el programador no tiene que pre-

ocuparse de si hay uno o varios procesadores. El programa se divide de forma lógica en hilos y si hay más de un procesador, se ejecuta más rápidamente, sin que sea necesario llevar a cabo ningún ajuste adicional sobre el código.

Todo esto hace que el manejo de hilos parezca muy sencillo. Hay un inconveniente: los recursos compartidos. Si se tiene más de un hilo en ejecución tratando de acceder al mismo recurso, se plantea un problema. Por ejemplo, dos procesos no pueden enviar simultáneamente información a una impresora. Para resolver el problema, los recursos que pueden ser compartidos como la impresora, deben bloquearse mientras se están usando. Por tanto, un hilo bloquea un recurso, completa su tarea y después libera el bloqueo de manera que alguien más pueda usar ese recurso.

El hilo de Java está incluido en el propio lenguaje, lo que hace que un tema de por sí complicado se presente de forma muy sencilla. El manejo de hilos se soporta a nivel de objeto, de manera que un hilo de ejecución se representa por un objeto. Java también proporciona bloqueo de recursos limitados; puede bloquear la memoria de cualquier objeto (que en el fondo, no deja de ser un tipo de recurso compartido) de manera que sólo un objeto pueda usarlo en un instante dado. Esto se logra mediante la palabra clave **synchronized**. Otros tipos de recursos deben ser explícitamente bloqueados por el programador, generalmente, creando un objeto que represente el bloqueo que todos los hilos deben comprobar antes de acceder al recurso.

Persistencia

Al crear un objeto, existe tanto tiempo como sea necesario, pero bajo ninguna circunstancia sigue existiendo una vez que el programa acaba. Si bien esta circunstancia parece tener sentido a primera vista, hay situaciones en las que sería increíblemente útil el que un objeto pudiera existir y mantener su información incluso cuando el programa ya no esté en ejecución. De esta forma, la siguiente vez que se lance el programa, el objeto estará ahí y seguirá teniendo la misma información que tenía la última vez que se ejecutó el programa. Por supuesto, es posible lograr un efecto similar escribiendo la información en un archivo o en una base de datos, pero con la idea de hacer que todo sea un objeto, sería deseable poder declarar un objeto como persistente y hacer que alguien o algo se encargue de todos los detalles, sin tener que hacerlo uno mismo.

Java proporciona soporte para "persistencia ligera", lo que significa que es posible almacenar objetos de manera sencilla en un disco para más tarde recuperarlos. La razón de que sea "ligera" es que es necesario hacer llamadas explícitas a este almacenamiento y recuperación. Además, los JavaSpaces (descritos en el Capítulo 15) proporcionan cierto tipo de almacenamiento persistente de los objetos. En alguna versión futura, podría aparecer un soporte completo para la persistencia.

Java e Internet

Si Java es, de hecho, otro lenguaje de programación de computadores entonces uno podría preguntarse por qué es tan importante y por qué debería promocionarse como un paso revolucionario en la programación de computadores. La respuesta no es obvia ni inmediata si se procede de la perspectiva de programación tradicional. Aunque Java es muy útil de cara a la solución de problemas de

programación tradicionales autónomos, también es importante por resolver problemas de programación en la World Wide Web.

¿Qué es la Web?

La Web puede parecer un gran misterio a primera vista, especialmente cuando se oye hablar de "navegar", "presencia" y "páginas iniciales" (*home pages*). Ha habido incluso una reacción creciente contra la "Internet-manía", cuestionando el valor económico y el beneficio de un movimiento tan radical. Es útil dar un paso atrás y ver lo que es realmente, pero para hacer esto es necesario entender los sistemas cliente/servidor, otro elemento de la computación lleno de aspectos que causan también confusión.

Computación cliente/servidor

La idea principal de un sistema cliente/servidor es que se dispone de un depósito (*repositorio*) central de información —cierto tipo de datos, generalmente en una base de datos— que se desea distribuir bajo demanda a cierto conjunto de máquinas o personas. Una clave para comprender el concepto de cliente/servidor es que el depósito de información está ubicado centralmente, de manera que puede ser modificado y de forma que los cambios se propaguen a los consumidores de la información. A la(s) máquina(s) en las que se ubican conjuntamente el depósito de información y el software que la distribuye se la denomina el servidor. El software que reside en la máquina remota se comunica con el servidor, toma la información, la procesa y después la muestra en la máquina remota, denominada el *cliente*.

El concepto básico de la computación cliente/servidor, por tanto, no es tan complicado. Aparecen problemas porque se tiene un único servidor que trata de dar servicio a múltiples clientes simultáneamente. Generalmente, está involucrado algún sistema gestor de base de datos de manera que el diseñador "reparte" la capa de datos entre distintas tablas para lograr un uso óptimo de los mismos. Además, estos sistemas suelen admitir que un cliente inserte nueva información en el servidor. Esto significa que es necesario asegurarse de que el nuevo dato de un cliente no machaque los nuevos datos de otro cliente, o que no se pierda este dato en el proceso de su adición a la base de datos (a esto se le denomina procesamiento de la transacción). Al cambiar el software cliente, debe ser construido, depurado e instalado en las máquinas cliente, lo cual se vuelve bastante más complicado y caro de lo que pudiera parecer. Es especialmente problemático dar soporte a varios tipos de computadores y sistemas operativos. Finalmente, hay un aspecto de rendimiento muy importante: es posible tener cientos de clientes haciendo peticiones simultáneas a un mismo servidor, de forma que un mínimo retraso sea crucial. Para minimizar la latencia, los programadores deben empeñarse a fondo para disminuir las cargas de las tareas en proceso, generalmente repartiéndolas con las máquinas cliente, pero en ocasiones, se dirige la carga hacia otras máquinas ubicadas junto con el servidor, denominadas intermediarios "*middleware*" (que también se utiliza para mejorar la *mantenibilidad* del sistema global).

La simple idea de distribuir la información a la gente, tiene muchas capas de complejidad en la fase de implementación, y el problema como un todo puede parecer desesperanzador. E incluso puede ser crucial: la computación cliente/servidor se lleva prácticamente la mitad de todas las actividades de programación. Es responsable de todo, desde recibir las órdenes y transacciones de tarjetas de

crédito hasta la distribución de cualquier tipo de datos —mercado de valores, datos científicos, del gobierno,... Hasta la fecha, en el pasado, se han intentado y desarrollado soluciones individuales para problemas específicos, inventando una solución nueva cada vez. Estas soluciones eran difíciles de crear y utilizar, y el usuario debía aprenderse nuevas interfaces para cada una de ellas. El problema cliente/servidor completo debe resolverse con un enfoque global.

La Web como un servidor gigante

La Web es, de hecho, un sistema cliente/servidor gigante. Es un poco peor que eso, puesto que todos los servidores y clientes coexisten en una única red a la vez. No es necesario, sin embargo, ser conscientes de este hecho, puesto que simplemente es necesario preocuparse de saber cómo conectarse y cómo interactuar con un servidor en un momento dado (incluso aunque sea necesario merodear por todo el mundo para encontrar el servidor correcto).

Inicialmente, este proceso era unidireccional. Se hacía una petición de un servidor y éste te proporcionaba un archivo que el software navegador (por ejemplo, el cliente) de tu máquina podía interpretar dándole el formato adecuado en la máquina local. Pero en poco tiempo, la gente empezó a demandar más servicios que simplemente recibir páginas de un servidor. Se pedían capacidades cliente/servidor completas, de manera que el cliente pudiera retroalimentar de información al servidor; por ejemplo, para hacer búsquedas en base de datos en el servidor, añadir nueva información al mismo, o para ubicar una orden (lo que requería un nivel de seguridad mucho mayor que el que ofrecían los sistemas originales). Éstos son los cambios de los que hemos sido testigos a lo largo del desarrollo de la Web.

El navegador de la Web fue un gran paso hacia delante: el concepto de que un fragmento de información pudiera ser mostrado en cualquier tipo de computador sin necesidad de modificarlo. Sin embargo, los navegadores seguían siendo bastante primitivos y pronto se pasaron de moda debido a las demandas que se les hacían. No eran especialmente interactivos, y tendían a saturar tanto el servidor como Internet puesto que cada vez que requería hacer algo que exigiera programación había que enviar información de vuelta al servidor para que fuera procesada. Encontrar algo que por ejemplo, se había tecleado incorrectamente en una solicitud, podía llevar muchos minutos o segundos. Dado que el navegador era únicamente un visor no podía desempeñar ni siquiera las tareas de computación más simples. (Por otro lado, era seguro, puesto que no podía ejecutar programas en la máquina local que pudiera contener errores (*bugs*) o virus.)

Para resolver el problema, se han intentado distintos enfoques. El primero de ellos consistió en mejorar los estándares gráficos para permitir mejores animaciones y vídeos dentro de los navegadores. El resto del problema se puede resolver incorporando simplemente la capacidad de ejecutar programas en el cliente final, bajo el navegador. Esto se denomina programación en la parte cliente.

Programación en el lado del cliente

El diseño original servidor-navegador de la Web proporcionaba contenidos interactivos, pero la capacidad de interacción la proporcionaba completamente el servidor. Éste producía páginas estáticas para el navegador del cliente, que simplemente las interpretaba y visualizaba. El HTML básico

contiene mecanismos simples para la recopilación de datos: cajas de entrada de textos, cajas de prueba, cajas de radio, listas y listas desplegables, además de un botón que sólo podía programarse para borrar los datos del formulario o "enviar" los datos del formulario de vuelta al servidor. Este envío de datos se lleva a cabo a través del *Common Gateway Interface* (CGI), proporcionado por todos los servidores web. El texto del envío transmite a CGI qué debe hacer con él. La acción más común es ejecutar un programa localizado en el servidor en un directorio denominado generalmente "cgi-bin". (Si se echa un vistazo a la ventana de direcciones de la parte superior del navegador al presionar un botón de una página Web, es posible ver en ocasiones la cadena "cgi-bin" entre otras cosas.) Estos programas pueden escribirse en la mayoría de los lenguajes. Perl es una elección bastante frecuente pues fue diseñado para la manipulación de textos, y es interpretado, lo que permite que pueda ser instalado en cualquier servidor sin que importe el procesador o sistema operativo instalado.

Muchos de los sitios web importantes de hoy en día se siguen construyendo estrictamente con CGI, y es posible, de hecho, hacer casi cualquier cosa con él. Sin embargo, los sitios web cuyo funcionamiento se basa en programas CGI se suelen volver difíciles de mantener, y presentan además problemas de tiempo de respuesta. (Además, poner en marcha programas CGI suele ser bastante lento.) Los diseñadores iniciales de la Web no previeron la rapidez con que se agotaría el ancho de banda para los tipos de aplicaciones que se desarrollaron. Por ejemplo, es imposible llevar a cabo cualquier tipo de generación dinámica de gráficos con consistencia, pues es necesario crear un archivo GIF que pueda ser después trasladado del servidor al cliente para cada versión del gráfico. Y seguro que todo el mundo ha tenido alguna experiencia con algo tan simple como validar datos en un formulario de entrada. Se presiona el botón de enviar de una página; los datos se envían de vuelta al servidor; el servidor pone en marcha un programa CGI y descubre un error, da formato a una página HTML informando del error y después vuelve a mandar la página de vuelta; entonces es necesario recuperar el formulario y volver a empezar. Esto no es solamente lento, sino que es además poco elegante.

La solución es la programación en el lado del cliente. La mayoria de las máquinas que ejecutan navegadores Web son motores potentes capaces de llevar a cabo grandes cantidades de trabajo, y con el enfoque HTML estático original, simplemente estaban allí "sentadas", esperando ociosas a que el servidor se encargara de la página siguiente. La programación en el lado del cliente quiere decir que el servidor web tiene permiso para hacer cualquier trabajo del que sea capaz, y el resultado para el usuario es una experiencia mucho más rápida e interactiva en el sitio web.

El problema con las discusiones sobre la programación en el lado cliente es que no son muy distintas de las discusiones de programación en general. Los parámetros son casi los mismos, pero la plataforma es distinta: un navegador web es como un sistema operativo limitado. Al final, uno debe seguir programando, y esto hace que siga existiendo el clásico conjunto de problemas y soluciones, producidos en este caso por la programación en el lado del cliente. El resto de esta sección proporciona un repaso de los aspectos y enfoques en la programación en el lado del cliente.

Conectables *(plug-ins)*

Uno de los mayores avances en la programación en la parte cliente es el desarrollo de los conectables *(plug-ins)*. Éstos son modos en los que un programador puede añadir nueva funcionalidad al

navegador descargando fragmentos de código que se conecta en el punto adecuado del navegador. Le dice al navegador "de ahora en adelante eres capaz de llevar a cabo esta nueva actividad". (Es necesario descargar cada conectable únicamente una vez.) A través de los conectables, se añade comportamiento rápido y potente al navegador, pero la escritura de un conectable no es trivial y desde luego no es una parte deseable para hacer como parte de un proceso de construcción de un sitio web. El valor del conectable para la programación en el lado cliente es tal que permite a un programador experto desarrollar un nuevo lenguaje y añadirlo a un navegador sin permiso de la parte que desarrolló el propio navegador. Por consiguiente, los navegadores proporcionan una "puerta trasera que permite la creación de nuevos lenguajes de programación en el lado cliente (aunque no todos los lenguajes se encuentren implementados como conectables).

Lenguajes de guiones

Los conectables condujeron a la explosión de los lenguajes de guiones *(scripting)*. Con uno de estos lenguajes se integra el código fuente del programa de la parte cliente directamente en la página HTML, y el conectable que interpreta ese lenguaje se activa automáticamente a medida que se muestra la página HTML. Estos lenguajes tienden a ser bastante sencillos de entender y, dado que son simplemente texto que forma parte de una página HTML, se cargan muy rápidamente como parte del único acceso al servidor mediante el que se accede a esa página. El sacrificio es que todo el mundo puede ver (y robar) el código así transportado. Sin embargo, generalmente, si no se pretende hacer cosas excesivamente complicadas, los lenguajes de guiones constituyen una buena herramienta, al no ser complicados.

Esto muestra que los lenguajes de guiones utilizados dentro de los navegadores web se desarrollaron verdaderamente para resolver tipos de problemas específicos, en particular la creación de interfaces gráficos de usuario (IGUs) más interactivos y ricos. Sin embargo, uno de estos lenguajes puede resolver el 80% de los problemas que se presentan en la programación en el lado cliente. Este 80% podría además abarcar todos los problemas de muchos programadores, y dado que los lenguajes de programación permiten desarrollos mucho más sencillos y rápidos, es útil pensar en utilizar uno de estos lenguajes antes de buscar soluciones más complicadas, como la programación en Java o ActiveX.

Los lenguajes de guiones de navegador más comunes son JavaScript (que no tiene nada que ver con Java; se denominó así simplemente para aprovechar el buen momento de marketing de Java), VBScript (que se parece bastante a Visual Basic), y Tcl/Tk, que proviene del popular lenguaje de construcción de IGU (Interfaz Gráfica de Usuario) de plataformas cruzadas. Hay otros más, y seguro que se desarrollarán muchos más.

JavaScript es probablemente el que recibe más apoyo. Viene incorporado tanto en el navegador Netscape Navigator como en el Microsoft Internet Explorer (IE). Además, hay probablemente más libros de JavaScritpt que de otros lenguajes de navegador, y algunas herramientas crean páginas automáticamente haciendo uso de JavaScript. Sin embargo, si se tiene habilidad en el manejo de Visual Basic o Tcl/Tk, será más productivo hacer uso de esos lenguajes de guiones en vez de aprender uno nuevo. (De hecho, ya se habrá hecho uso de aspectos web para estas alturas.)

Java

Si un lenguaje de programación puede resolver el 80 por ciento de los problemas de programación en el lado cliente, ¿qué ocurre con el 20 por ciento restante —que constituyen de hecho "la

parte seria del problema"? La solución más popular hoy en día es Java. No sólo se trata de un lenguaje de programación potente, construido para ser seguro, de plataforma cruzada (multiplataforma) e internacional, sino que se está extendiendo continuamente para proporcionar aspectos de lenguaje y bibliotecas que manejan de manera elegante problemas que son complicados para los lenguajes de programación tradicionales, como la ejecución multihilo, el acceso a base de datos, la programación en red, y la computación distribuida. Java permite programación en el lado cliente a través del *applet*.

Un *applet* es un miniprograma que se ejecutará únicamente bajo un navegador web. El *applet* se descarga automáticamente como parte de una página web (igual que, por ejemplo, se descarga un gráfico, de manera automática). Cuando se activa un *applet*, ejecuta un programa. Ésta es parte de su belleza —proporciona una manera de distribuir automáticamente software cliente desde el servidor justo cuando el usuario necesita software cliente, y no antes. El usuario se hace con la última versión del software cliente, sin posibilidad de fallo, y sin tener que llevar a cabo reinstalaciones complicadas. Gracias a cómo se ha diseñado Java, el programado simplemente tiene que crear un único programa, y este programa trabaja automáticamente en todos los computadores que tengan navegadores que incluyan intérpretes de Java. (Esto incluye seguramente a la gran mayoría de plataformas.) Dado que Java es un lenguaje de programación para novatos, es posible hacer tanto trabajo como sea posible en el cliente, antes y después de hacer peticiones al servidor. Por ejemplo, no se deseará enviar un formulario de petición a través de Internet para descubrir que se tiene una fecha o algún otro parámetro erróneo, y el computador cliente puede llevar a cabo rápidamente la labor de registrar información en vez de tener que esperar a que lo haga el servidor para enviar después una imagen gráfica de vuelta. No sólo se consigue un incremento de velocidad y capacidad de respuesta inmediatas, sino que el tráfico en general de la red y la carga en los servidores se reduce considerablemente, evitando que toda Internet se vaya ralentizando.

Una ventaja que tienen los *applets* de Java sobre los lenguajes de guiones es que se encuentra en formato compilado, de forma que el código fuente no está disponible para el cliente. Por otro lado, es posible descompilar un *applet* Java sin excesivo trabajo, pero esconder un código no es un problema generalmente importante. Hay otros dos factores que pueden ser importantes. Como se verá más tarde en este libro, un *applet* Java compilado puede incluir varios módulos e implicar varios accesos al servidor para su descarga (en Java 1.1 y superiores, esto se minimiza mediante archivos Java, denominados archivos JAR, que permiten que los módulos se empaqueten todos juntos y se compriman después para que baste con una única descarga). Un programa de guiones se integrará simplemente en una página web como parte de su texto (y generalmente será más pequeño reduciendo los accesos al servidor). Esto podría ser importante de cara al tiempo de respuesta del sitio web. Otro factor importante es la curva de aprendizaje. A pesar de lo que haya podido oírse, Java no es un lenguaje cuyo aprendizaje resulte trivial. Para los programadores en Visual Basic, moverse a VBScript siempre será la solución más rápida, y dado que probablemente este lenguaje resolverá los problemas cliente/servidor más típicos, puede resultar bastante complicado justificar la necesidad de aprender Java. Si uno ya tiene experiencia con un lenguaje de guiones, seguro que se obtendrán beneficios simplemente haciendo uso de JavaScript o VBScript antes de lanzarse a utilizar Java, ya que estos lenguajes pueden resolver todos los problemas de manera sencilla, y se logrará un nivel de productividad elevado en un tiempo menor.

ActiveX

Hasta cierto grado, el competidor principal de Java es el ActiveX de Microsoft, aunque se base en un enfoque totalmente diferente. ActiveX era originalmente una solución válida exclusivamente para Windows, aunque ahora se está desarrollando mediante un consorcio independiente de manera que acabará siendo multiplataforma (plataforma cruzada). Efectivamente, ActiveX se basa en que "si un programa se conecta a su entorno de manera que puede ser depositado en una página web y ejecutado en un navegador, entonces soporta ActiveX". (IE soporta ActiveX directamente, y Netscape también, haciendo uso de un conectable.) Por consiguiente, ActiveX no se limita a un lenguaje particular. Si, por ejemplo, uno es un programador Windows experimentado, haciendo uso de un lenguaje como C++, Visual Basic o en Delphi de Borland, es posible crear componentes ActiveX sin casi tener que hacer ningún cambio a los conocimientos de programación que ya se tengan. Además, ActiveX proporciona un modo de usar código antiguo (base dado) en páginas web.

Seguridad

La capacidad para descargar y ejecutar programas a través de Internet puede parecer el sueño de un constructor de virus. ActiveX atrae especialmente el espinoso tema de la seguridad en la programación en la parte cliente. Si se hace *clic* en el sitio web, es posible descargar automáticamente cualquier número de cosas junto con la página HTML: archivos GIF, código de guiones, código Java compilado, y componentes ActiveX. Algunos de estos elementos son benignos: los archivos GIF no pueden hacer ningún daño, y los lenguajes de guiones se encuentran generalmente limitados en lo que pueden hacer. Java también fue diseñado para ejecutar sus *applets* dentro de un "envoltorio" de seguridad, lo que evita que escriba en el disco o acceda a la memoria externa a ese envoltorio.

ActiveX está en el rango opuesto del espectro. Programar con ActiveX es como programar Windows —es posible hacer cualquier cosa. De esta manera, si se hace *clic* en una página web que descarga un componente ActiveX, ese componente podría llegar a dañar los archivos de un disco. Por supuesto, los programas que uno carga en un computador, y que no están restringidos a ejecutarse dentro del navegador web podrían hacer lo mismo. Los virus que se descargaban desde una *BBS (Bulletin-Board Systems)* hace ya tiempo que son un problema, pero la velocidad de Internet amplifica su gravedad.

La solución parecen aportarla las "firmas digitales", que permiten la verificación de la autoría del código. Este sistema se basa en la idea de que un virus funciona porque su creador puede ser anónimo, de manera que si se evita la ejecución de programas anónimos, se obligará a cada persona a ser responsable de sus actos. Esto parece una buena idea, pues permite a los programas ser mucho más funcionales, y sospecho que eliminará las diabluras maliciosas. Si, sin embargo, un programa tiene un error (*bug*) inintencionadamente destructivo, seguirá causando problemas.

El enfoque de Java es prevenir que estos problemas ocurran, a través del envoltorio. El intérprete de Java que reside en el navegador web local examina el *applet* buscando instrucciones adversas a medida que se carga el *applet*. Más en concreto, el *applet* no puede escribir ficheros en el disco o borrar ficheros (una de las principales vías de ataque de los virus). Los *applets* se consideran generalmente seguros, y dado que esto es esencial para lograr sistemas cliente/servidor de confianza, cualquier error (*bug*) que produzca virus en lenguaje Java será rápidamente reparado. (Merece la

pena destacar que el software navegador, de hecho, refuerza estas restricciones de seguridad, y algunos navegadores permiten seleccionar distintos niveles de seguridad para proporcionar distintos grados de acceso a un sistema.)

También podría uno ser escéptico sobre esta restricción tan draconiana en contra de la escritura de ficheros en un disco local. Por ejemplo, uno puede desear construir una base de datos o almacenar datos para utilizarlos posteriormente, finalizada la conexión. La visión inicial parecía ser tal que eventualmente todo el mundo podría conseguir hacer cualquier cosa importante estando conectado, pero pronto se vio que esta visión no era práctica (aunque los "elementos Internet" de bajo coste puedan satisfacer algún día las necesidades de un segmento de usuarios significativo). La solución es el "*applet* firmado" que utiliza cifrado de clave pública para verificar que un *applet* viene efectivamente de donde dice venir. Un *applet* firmado puede seguir destrozando un disco local, pero la teoría es que dado que ahora es posible localizar al creador del *applet*, éstos no actuarán de manera perniciosa. Java proporciona un marco de trabajo para las firmas digitales, de forma que será posible permitir que un *applet* llegue a salir fuera del envoltorio si es necesario.

Las firmas digitales han olvidado un aspecto importante, que es la velocidad con la que la gente se mueve por Internet. Si se descarga un programa con errores (*bugs*) que hace algo dañino, ¿cuánto tiempo se tardará en descubrir el daño? Podrían pasar días o incluso semanas. Para entonces, ¿cómo localizar el programa que lo ha causado? ¿Y será todo el mundo capaz de hacerlo?

Internet frente a Intranet

La Web es la solución más general al problema cliente/servidor, de forma que tiene sentido que se pueda utilizar la misma tecnología para resolver un subconjunto del problema, en particular, el clásico problema cliente/servidor *dentro* de una compañía. Con los enfoques cliente/servidor tradicionales, se tiene el problema de la multiplicidad de tipos de máquinas cliente, además de la dificultad de instalar un nuevo software cliente, si bien ambos problemas pueden resolverse sencillamente con navegadores web y programación en el lado cliente. Cuando se utiliza tecnología web para una red de información restringida a una compañía en particular, se la denomina una Intranet. Las Intranets proporcionan un nivel de seguridad mucho mayor que el de Internet, puesto que se puede controlar físicamente el acceso a los servidores dentro de la propia compañía. En términos de formación, parece que una vez que la gente entiende el concepto general de navegador es mucho más sencillo que se enfrenten a distintas páginas y *applets*, de manera que la curva de aprendizaje para nuevos tipos de sistemas parece reducirse.

El problema de la seguridad nos conduce ante una de las divisiones que parece estar formándose automáticamente en el mundo de la programación en el lado del cliente. Si un programa se ejecuta en Internet, no se sabe bajo qué plataforma estará funcionando, y si se desea ser extremadamente cauto, no se diseminará código con error. Es necesario algo multiplataforma y seguro, como un lenguaje de guiones o Java.

Si se está ejecutando código en una Intranet, es posible tener un conjunto de limitaciones distinto. No es extraño que las máquinas de una red puedan ser todas plataformas Intel/Windows. En una Intranet, uno es responsable de la calidad de su propio código y puede reparar errores en el momento en que se descubren. Además, se podría tener cierta cantidad de código antiguo (heredado, *legacy*) que se ha

estado utilizando en un enfoque cliente/servidor más tradicional, en cuyo caso sería necesario instalar físicamente programas cliente cada vez que se construya una versión más moderna. El tiempo malgastado en instalar actualizaciones (*upgrades*) es la razón más apabullante para comenzar a usar navegadores, en los que estas actualizaciones son invisibles y automáticas. Para aquéllos que tengan intranets, el enfoque más sensato es tomar el camino más corto que permita usar el código base existente, en vez de volver a codificar todos los programas en un nuevo lenguaje.

Se ha hecho frente a este problema presentando un conjunto desconcertante de soluciones al problema de programación en el lado cliente, y la mejor determinación para cada caso es la que determine un análisis coste-beneficio. Deben considerarse las restricciones de cada problema y cuál sería el camino más corto para encontrar la solución en cada caso. Dado que la programación en la parte cliente sigue siendo programación, suele ser buena idea tomar el enfoque de desarrollo más rápido para cada situación. Ésta es una postura agresiva para prepararse de cara a inevitables enfrentamientos con los problemas del desarrollo de programas.

Programación en el lado del servidor

Hasta la fecha toda discusión ha ignorado el problema de la programación en el lado del servidor. ¿Qué ocurre cuando se hace una petición a un servidor? La mayoría de las veces la petición es simplemente "envíame este archivo". A continuación, el navegador interpreta el archivo de la manera adecuada: como una página HTML, como una imagen gráfica, un *applet* Java, un programa de guiones, etc. Una petición más complicada hecha a un servidor puede involucrar una transacción de base de datos. Un escenario común involucra una petición para una búsqueda compleja en una base de datos, que el servidor formatea en una página HTML para enviarla a modo de resultado (por supuesto, si el cliente tiene una inteligencia mayor vía Java o un lenguaje de guiones, pueden enviarse los datos simplemente, sin formato, y será el extremo cliente el que les dé el formato adecuado, lo cual es más rápido, además de implicar una carga menor para el servidor). Un usuario también podría querer registrar su nombre en una base de datos al incorporarse a un grupo o presentar una orden, lo cual implica cambios a esa base de datos. Estas peticiones deben procesarse vía algún código en el lado servidor, que se denomina generalmente programación en el lado servidor. Tradicionalmente, esta programación se ha desempeñado mediante Perl y guiones CGI, pero han ido apareciendo sistemas más sofisticados. Entre éstos se encuentran los servidores web basados en Java que permiten llevar a cabo toda la programación del lado servidor en Java escribiendo lo que se denominan *servlets*. Éstos y sus descendientes, los JSP, son las dos razones principales por las que las compañías que desarrollan sitios web se están pasando a Java, especialmente porque eliminan los problemas de tener que tratar con navegadores de distintas características.

Un ruedo separado: las aplicaciones

Muchos de los comentarios en torno a Java se referían a los *applets*. Java es actualmente un lenguaje de programación de propósito general que puede resolver cualquier tipo de problema —al menos en teoría. Y como se ha señalado anteriormente, cuando uno se sale del ruedo de los *applets* (y simultáneamente se salta las restricciones, como la contraria a la escritura en el disco) se entra en el mundo de las aplicaciones de propósito general que se ejecutan independientemente, sin un nave-

gador web, al igual que hace cualquier programa ordinario. Aquí, la fuerza de Java no es sólo su portabilidad, sino también su programabilidad (facilidad de programación). Como se verá a lo largo del presente libro, Java tiene muchos aspectos que permiten la creación de programas robustos en un período de tiempo menor al que requerían los lenguajes de programación anteriores.

Uno debe ser consciente de que esta bendición no lo es del todo. El precio a pagar por todas estas mejoras es una velocidad de ejecución menor (aunque se está haciendo bastante trabajo en este área —JDK 1.3, en particular, presenta las mejoras de rendimiento denominadas "*hotspot*"). Como cualquier lenguaje, Java tiene limitaciones intrínsecas que podrían hacerlo inadecuado para resolver cierto tipo de problemas de programación. Java es un lenguaje que evoluciona rápidamente, no obstante, y cada vez que aparece una nueva versión, se presenta más y más atractivo de cara a la solución de conjuntos mayores de problemas.

Análisis y diseño

El paradigma de la orientación a objetos es una nueva manera de enfocar la programación. Son muchos los que tienen problemas a primera vista para enfrentarse a un proyecto de POO. Dado que se supone que todo es un objeto, y a medida que se aprende a pensar de forma orientada a objetos, es posible empezar a crear "buenos" diseños y sacar ventaja de todos los beneficios que la POO puede ofrecer.

Una *metodología* es un conjunto de procesos y heurísticas utilizadas para descomponer la complejidad de un problema de programación. Se han formulado muchos métodos de POO desde que enunció la programación orientada a objetos. Esta sección presenta una idea de lo que se trata de lograr al utilizar un método.

Especialmente en la POO, la metodología es un área de intensa experimentación, por lo que es importante entender qué problema está intentando resolver el método antes de considerar la adopción de uno de ellos. Esto es particularmente cierto con Java, donde el lenguaje de programación se ha desarrollado para reducir la complejidad (en comparación con C) necesaria para expresar un programa. Esto puede, de hecho, aliviar la necesidad de metodologías cada vez más complejas. En vez de esto, puede que las metodologías simples sean suficientes en Java para conjuntos de problemas mucho mayores que los que se podrían manipular utilizando metodologías simples con lenguajes procedimentales.

También es importante darse cuenta de que el término "metodología" es a menudo muy general y promete demasiado. Se haga lo que se haga al diseñar y escribir un programa, se sigue un método. Puede que sea el método propio de uno, e incluso puede que uno no sea consciente de utilizarlo, pero es un proceso que se sigue al crear un programa. Si el proceso es efectivo, puede que simplemente sea necesario afinarlo ligeramente para poder trabajar con Java. Si no se está satisfecho con el nivel de productividad y la manera en que se comportan los programas, puede ser buena idea considerar la adopción de un método formal, o la selección de fragmentos de entre los muchos métodos formales existentes.

Mientras se está en el propio proceso de desarrollo, el aspecto más importante es no perderse, aunque puede resultar fácil. Muchos de los métodos de análisis y desarrollo fueron concebidos para re-

solver los problemas más grandes. Hay que recordar que la mayoría de proyectos no encajan en esta categoría, siendo posible muchas veces lograr un análisis y un diseño con éxito con sólo un pequeño subconjunto de los que el método recomienda[10]. Pero algunos tipos de procesos, sin importar lo limitados que puedan ser, le permitirán encontrar el camino de manera más sencilla que si simplemente se empieza a codificar.

También es fácil desesperarse, caer en "parálisis de análisis", cuando se siente que no se puede avanzar porque no se han cubierto todos los detalles en la etapa actual. Debe recordarse que, independientemente de cuánto análisis lleve a cabo, hay cosas de un sistema que no aparecerán hasta la fase de diseño, y otras no aflorarán incluso hasta la fase de codificación o en un extremo, hasta que el programa esté acabado y en ejecución. Debido a esto, es crucial moverse lo suficientemente rápido a través de las etapas de análisis y diseño e implementar un prototipo del sistema propuesto.

Debe prestarse un especial énfasis a este punto. Dado que ya se conoce la misma historia con los lenguajes *procedimentales*, es recomendable que el equipo proceda de manera cuidadosa y comprenda cada detalle antes de pasar del diseño a la implementación. Ciertamente, al crear un SGBD, esto pasa por comprender completamente la necesidad del cliente. Pero un SGBD es la clase de problema bien formulada y bien entendida; en muchos programas, *es* la estructura de la base de datos la que debe ser desmenuzada. La clase de problema de programación examinada en el presente capítulo es un "juego de azar"[11], en el que la solución no es simplemente la formulación de una solución bien conocida, sino que involucra además a uno o más "factores de azar" —elementos para los que no existe una solución previa bien entendida, y para los cuales es necesario algún tipo de proceso de investigación[12]. Intentar analizar completamente un problema al azar antes de pasar al diseño e implementación conduce a una parálisis en el análisis, al no tener suficiente información para resolver este tipo de problemas durante la fase de análisis. Resolver un problema así, requiere iterar todo el ciclo, y precisa de un comportamiento que asuma riesgos (lo cual tiene sentido, pues está intentando hacer algo nuevo y las recompensas potenciales crecen). Puede parecer que el riesgo se agrava al precipitarse hacia una implementación preliminar, pero ésta puede reducir el riesgo en los problemas al azar porque se está averiguando muy pronto si un enfoque particular al problema es o no viable. El desarrollo de productos conlleva una gestión del riesgo.

A menudo, se propone "construir uno para desecharlo". Con POO es posible tirar *parte*, pero dado que el código está encapsulado en clases, durante la primera pasada siempre se producirá algún diseño de clases útil y se desarrollarán ideas que merezcan la pena para el diseño del sistema de las que no habrá que deshacerse. Por tanto, la primera pasada rápida por un problema no sólo suministra información crítica para las ulteriores pasadas por análisis, diseño e implementación, sino que también crea la base del código.

---

[10] Un ejemplo excelente de esto es *UML Distilled*, 2.ª edición, de Martin Fowler (Addison-Wesley 2000), que reduce el proceso, en ocasiones aplastante, a un subconjunto manejable (existe versión española con el título *UML gota a gota*).

[11] N. del traductor: Término *wild-card*, acuñado por el autor original.

[12] Regla del pulgar —acuñada por el autor— para estimar este tipo de proyectos: si hay más de un factor al azar, ni siquiera debe intentarse planificar la duración o el coste del proyecto hasta que no se ha creado un prototipo que funcione. Existen demasiados grados de libertad.

Dicho esto, si se está buscando una metodología que contenga un nivel de detalle tremendo, y sugiera muchos pasos y documentos, puede seguir siendo difícil saber cuándo parar. Debe recomendarse lo que se está intentando descubrir.

1. ¿Cuáles son los objetos? (¿Cómo se descompone su proyecto en sus componentes?)

2. ¿Cuáles son las interfaces? (¿Qué mensajes es necesario enviar a cada objeto?)

Si se delimitan los objetos y sus interfaces, ya es posible escribir un programa. Por diversas razones, puede que sean necesarias más descripciones y documentos que éste, pero no es posible avanzar con menos.

El proceso puede descomponerse en cinco fases, y la Fase 0 no es más que la adopción de un compromiso para utilizar algún tipo de estructura.

Fase 0: Elaborar un plan

En primer lugar, debe decidirse qué pasos debe haber en un determinado proceso. Suena bastante simple (de hecho, *todo* esto suena simple) y la gente no obstante, suele seguir sin tomar esta decisión antes de empezar a codificar. Si el plan consiste en "empecemos codificando", entonces, perfecto (en ocasiones, esto es apropiado, si uno se está enfrentando a un problema que conoce perfectamente). Al menos, hay que estar de acuerdo en que eso también es tener un plan.

También podría decidirse en esta fase que es necesaria alguna estructura adicional de proceso, pero no toda una metodología completa. Para que nos entendamos, a algunos programadores les gusta trabajar en "modo vacación", en el que no se imponga ninguna estructura en el proceso de desarrollar de su trabajo; "se hará cuando se haga". Esto puede resultar atractivo a primera vista, pero a medida que se tiene algo de experiencia uno se da cuenta de que es mejor ordenar y distribuir el esfuerzo en distintas etapas en vez de lanzarse directamente a "finalizar el proyecto". Además, de esta manera se divide el proyecto en fragmentos más asequibles, y se resta miedo a la tarea de enfrentarse al mismo (además, las distintas fases o hitos proporcionan más motivos de celebración).

Cuando empecé a estudiar la estructura de la historia (con el propósito de acabar escribiendo algún día una novela), inicialmente, la idea que más me disgustaba era la de la estructura, pues parecía que uno escribe mejor si simplemente se dedica a rellenar páginas. Pero más tarde descubrí que al escribir sobre computadores, tenía la estructura tan clara que no había que pensar demasiado en ella. Pero aún así, el trabajo se estructuraba, aunque sólo fuera semiconscientemente en mi cabeza. Incluso cuando uno piensa que el plan consiste simplemente en empezar a codificar, todavía se atraviesan algunas fases al plantear y contestar ciertas preguntas.

El enunciado de la misión

Cualquier sistema que uno construya, independientemente de lo complicado que sea, tiene un propósito fundamental: el negocio intrínseco en el mismo, la necesidad básica que cumple. Si uno puede mirar a través de la interfaz de usuario, a los detalles específicos del hardware o del sistema, los algoritmos de codificación y los problemas de eficiencia, entonces se encuentra el centro de su existencia —simple y directo. Como el denominado alto concepto (*high concept*) en las películas de

Hollywood, uno puede describir el propósito de un programa en dos o tres fases. Esta descripción, pura, es el punto de partida.

El alto concepto es bastante importante porque establece el tono del proyecto; es el enunciado de su misión. Uno no tiene por qué acertar necesariamente a la primera (puede ser que uno esté en una fase posterior del problema cuando se le ocurra el enunciado completamente correcto) pero hay que seguir intentándolo hasta tener la certeza de que está bien. Por ejemplo, en un sistema de control de tráfico aéreo, uno puede comenzar con un alto concepto centrado en el sistema que se está construyendo: "El programa de la torre hace un seguimiento del avión". Pero considérese qué ocurre cuando se introduce el sistema en un pequeño aeródromo; quizás sólo hay un controlador humano, o incluso ninguno. Un modelo más usual no abordará la solución que se está creando como describe el problema: "Los aviones llegan, descargan, son mantenidos y recargan, a continuación, salen".

Fase 1: ¿Qué estamos construyendo?

En la generación previa del diseño del programa (denominada *diseño procedural*) a esta fase se le denominaba "creación del *análisis de requisitos* y *especificación del sistema*". Éstas, por supuesto, eran fases en las que uno se perdía; documentos con nombres intimidadores que podían de por sí convertirse en grandes proyectos. Sin embargo, su intención era buena. El análisis de requisitos dice: "Construya una lista de directrices que se utilizarán para saber cuándo se ha acabado el trabajo y cuándo el cliente está satisfecho". La especificación del sistema dice: "He aquí una descripción de lo *que* el programa hará (pero no *cómo*) para satisfacer los requisitos hallados". El análisis de requisitos es verdaderamente un contrato entre usted y el cliente (incluso si el cliente trabaja en la misma compañía o es cualquier otro objeto o sistema). La especificación del sistema es una exploración de alto nivel en el problema, y en cierta medida, un descubrimiento de si puede hacerse y cuánto tiempo llevará. Dado que ambos requieren de consenso entre la gente (y dado que generalmente variarán a lo largo del tiempo) lo mejor es mantenerlos lo más desnudos posible —idealmente, tratará de listas y diagramas básicos para ahorrar tiempo. Se podría tener otras limitaciones que exijan expandirlos en documentos de mayor tamaño, pero si se mantiene que el documento inicial sea pequeño y conciso, es posible crearlo en unas pocas sesiones de tormenta de ideas (*brainstorming*) en grupo, con un líder que dinámicamente va creando la descripción. Este proceso no sólo exige que todos aporten sus ideas sino que fomenta el que todos los miembros del equipo lleguen a un acuerdo inicial. Quizás lo más importante es que puede incluso ayudar a que se acometa el proyecto con una gran dosis de entusiasmo.

Es necesario mantenerse centrado en el corazón de lo que se está intentando acometer en esta fase: determinar qué es lo que se supone que debe hacer el sistema. La herramienta más valiosa para esto es una colección de lo que se denomina "casos de uso". Los casos de uso identifican los aspectos claves del sistema, que acabarán por revelar las clases fundamentales que se usarán en éste. De hecho, los casos de uso son esencialmente soluciones descriptivas a preguntas como[13]:

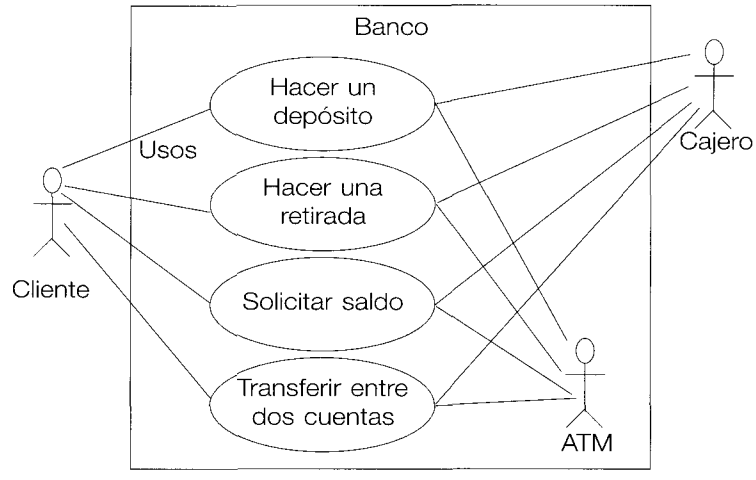
- "¿Quién usará el sistema?"
- "¿Qué pueden hacer esos actores con el sistema?"

[13] Agradecemos la ayuda de James H. Jarrett.

- ¿Cómo se las ingenia *cada actor* para *hacer* eso con este sistema?"
- "¿De qué otra forma podría funcionar esto si alguien más lo estuviera haciendo, o si el mismo actor tuviera un objetivo distinto?" (Para encontrar posibles variaciones.)
- "¿Qué problemas podrían surgir mientras se hace esto con el sistema?" (Para localizar posibles excepciones.)

Si se está diseñando, por ejemplo, un cajero automático, el caso de uso para un aspecto particular de la funcionalidad del sistema debe ser capaz de describir qué hace el cajero en cada situación posible. Cada una de estas "situaciones" se denomina un *escenario*, y un caso de uso puede considerarse como una colección de escenarios. Uno puede pensar que un escenario es como una pregunta que empieza por: "¿Qué hace el sistema si...?". Por ejemplo: "¿Qué hace el cajero si un cliente acaba de depositar durante las últimas 24 horas un cheque y no hay dinero suficiente en la cuenta, sin haber procesado el cheque, para proporcionarle la retirada el efectivo que ha solicitado?"

Deben utilizarse diagramas de caso de uso intencionadamente simples para evitar ahogarse prematuramente en detalles de implementación del sistema :

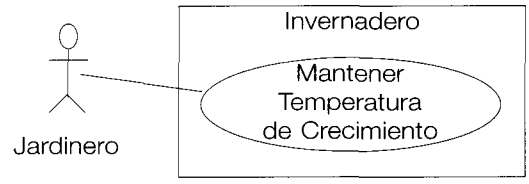


Cada uno de los monigotes representa a un "actor", que suele ser generalmente un humano o cualquier otro tipo de agente (por ejemplo, otro sistema de computación, como "ATM")[14]. La caja representa los límites de nuestro sistema. Las elipses representan los casos de uso, que son descripciones del trabajo útil que puede hacerse dentro del sistema. Las líneas entre los actores y los casos de uso representan las interacciones.

De hecho no importa cómo esté el sistema implementado, siempre y cuando tenga una apariencia como ésta para el usuario.

[14] ATM, siglas en inglés de cajero automático. (N. del T.)

Un caso de uso no tiene por qué ser terriblemente complejo, aunque el sistema subyacente sea complejo. Solamente se pretende que muestre el sistema tal y como éste se muestra al usuario. Por ejemplo:



Los casos de uso proporcionan las especificaciones de requisitos determinando todas las interacciones que el usuario podría tener con el sistema. Se trata de descubrir un conjunto completo de casos de uso para su sistema, y una vez hecho esto, se tiene el núcleo de lo que el sistema se supone que hará. Lo mejor de centrarse en los casos de uso es que siempre permiten volver a la esencia manteniéndose alejado de aspectos que no son críticos para lograr culminar el trabajo. Es decir, si se tiene un conjunto completo de casos de uso, es posible describir el sistema y pasar a la siguiente fase. Posiblemente no se podrá configurar todo a la primera, pero no pasa nada. Todo irá surgiendo a su tiempo, y si se demanda una especificación perfecta del sistema en este punto, uno se quedará parado.

Cuando uno se quede bloqueado, es posible comenzar esta fase utilizando una extensa herramienta de aproximación: describir el sistema en unos pocos párrafos y después localizar los sustantivos y los verbos. Los sustantivos pueden sugerir quiénes son los actores, el contexto del caso de uso (por ejemplo, "corredor"), o artefactos manipulados en el caso de uso. Los verbos pueden sugerir interacciones entre los actores y los casos de uso, y especificar los pasos dentro del caso de uso. También será posible descubrir que los sustantivos y los verbos producen objetos y mensajes durante la fase de diseño (y debe tenerse en cuenta que los casos de uso describen interacciones entre subsistemas, de forma que la técnica de "el sustantivo y el verbo" puede usarse sólo como una herramienta de tormenta de ideas, pues no genera casos de uso)[15].

La frontera entre un caso de uso y un actor puede señalar la existencia de una interfaz de usuario, pero no lo define. Para ver el proceso de cómo definir y crear interfaces de usuario, véase *Software for Use* de Larry Constantine y Lucy Lockwood, (Addison-Wesley Longman, 1999) o ir a *http://www.ForUse.com.*

Aunque parezca magia negra, en este punto es necesario algún tipo de planificación. Ahora se tiene una visión de lo que se está construyendo, por lo que probablemente se pueda tener una idea de cuánto tiempo le llevará. En este momento intervienen muchos factores. Si se estima una planificación larga, la compañía puede decidir no construirlo (y por consiguiente usar sus recursos en algo más razonable —esto es *bueno*). Pero un director podría tener decidido de antemano cuánto tiempo debería llevar el proyecto y podría tratar de influir en la estimación. Pero lo mejor es tener una estimación honesta desde el principio y tratar las decisiones duras al principio. Ha habido muchos in-

[15] Puede encontrarse más información sobre casos de uso en *Applying Use Cases*, de Schneider & Winters (Addison-Weley 1998) y *Use Case Driven Object modeling with UML* de Rosenberg (Addison-Welsey 1999).

tentos de desarrollar técnicas de planificación exactas (muy parecidas a las técnicas de predicción del mercado de valores), pero probablemente el mejor enfoque es confiar en la experiencia e intuición. Debería empezarse por una primera estimación del tiempo que llevaría, para posteriormente multiplicarla por dos y añadirle el 10 por ciento. La estimación inicial puede que sea correcta; a lo mejor *se puede* hacer que algo funcione en ese tiempo. Al "doblarlo" resultará que se consigue algo decente, y en el 10 por ciento añadido se puede acabar de pulir y tratar los detalles finales[16]. Sin embargo, es necesario explicarlo, y dejando de lado las manipulaciones y quejas que surgen al presentar una planificación de este tipo, normalmente funcionará.

Fase 2: ¿Cómo construirlo?

En esta fase debe lograrse un diseño que describe cómo son las clases y cómo interactuarán. Una técnica excelente para determinar las clases e interacciones es la tarjeta *Clase-Responsabilidad-Colaboración* (CRC)[17]. Parte del valor de esta herramienta se basa en que es de muy baja tecnología: se comienza con un conjunto de tarjetas de 3 x 5, y se escribe en ellas. Cada tarjeta representa una única clase, y en ella se escribe:

1. El nombre de la clase. Es importante que este nombre capture la esencia de lo que hace la clase, de manera que tenga sentido a primera vista.
2. Las "responsabilidades" de la clase: qué debería hacer. Esto puede resumirse típicamente escribiendo simplemente los nombres de las funciones miembros (dado que esas funciones deberían ser descriptivas en un buen diseño), pero no excluye otras anotaciones. Si se necesita ayuda, basta con mirar el problema desde el punto de vista de un programador holgazán: ¿qué objetos te gustaría que apareciesen por arte de magia para resolver el problema?
3. Las "colaboraciones" de la clase: ¿con qué otras clases interactúa? "Interactuar" es un término amplio intencionadamente; vendría a significar agregación, o simplemente que cualquier otro objeto existente ejecutara servicios para un objeto de la clase. Las colaboraciones deberían considerar también la audiencia de esa clase. Por ejemplo, si se crea una clase **Petardo**, ¿quién la va a observar, un **Químico** o un **Observador**? En el primer caso estamos hablando de punto de vista del químico que va a construirlo, mientras que en el segundo se hace referencia a los colores y las formas que libere al explotar.

Uno puede pensar que las tarjetas deberían ser más grandes para que cupiera en ellas toda la información que se deseara escribir, pero son pequeñas a propósito, no sólo para mantener pequeño el tamaño de las clases, sino también para evitar que se caiga en demasiado nivel de detalle muy pronto. Si uno no puede encajar todo lo que necesita saber de una clase en una pequeña tarjeta, la clase es demasiado compleja (o se está entrando en demasiado nivel de detalle, o se debería crear más de una clase). La clase ideal debería ser comprensible a primera vista. La idea de las tarjetas CRC es ayudar a obtener un primer diseño de manera que se tenga un dibujo a grandes rasgos que pueda ser después refinado.

[16] Mi opinión en este sentido ha cambiado últimamente. Al doblar y añadir el 10 por ciento se obtiene una estimación razonablemente exacta (asumiendo que no hay demasiados factores al azar) pero todavía hay que trabajar con bastante diligencia para finalizar en ese tiempo. Si se desea tener tiempo suficiente para lograr un producto verdaderamente elegante y disfrutar durante el proceso, el multiplicador correcto, en mi opinión, puede ser por tres o por cuatro.

[17] En inglés, *Class-Responsibility-Collaboration*. (N. del R.T.)

Una de las mayores ventajas de las tarjetas CRC se logra en la comunicación. Cuando mejor se hace es en tiempo real, en grupo y sin computadores. Cada persona se considera responsable de varias clases (que al principio no tienen ni nombres ni otras informaciones). Se ejecuta una simulación en directo resolviendo cada vez un escenario, decidiendo qué mensajes se mandan a los distintos objetos para satisfacer cada escenario. A medida que se averiguan las responsabilidades y colaboraciones de cada una, se van rellenando las tarjetas correspondientes. Cuando se han recorrido todos los casos de uso, se debería tener un diseño bastante completo.

Antes de empezar a usar tarjetas CRC, tuve una experiencia de consultoría de gran éxito, que me permitió presentar un diseño inicial a todo el equipo, que jamás había participado en un proyecto de POO, y que consistió en ir dibujando objetos en una pizarra, después de hablar sobre cómo se deberían comunicar los objetos entre sí, y tras borrar algunos y reemplazar otros. Efectivamente, estaban haciendo uso de "tarjetas CRC" en la propia pizarra. El equipo (que sabía que el proyecto se iba a hacer) creó, de hecho, el diseño; ellos eran los "propietarios" del diseño, más que recibirlo hecho directamente. Todo lo que yo hacía era guiar el proceso haciendo en cada momento las preguntas adecuadas, poniendo a prueba los distintos supuestos, y tomando la realimentación del equipo para ir modificando los supuestos. La verdadera belleza del proyecto es que el equipo aprendió cómo hacer diseño orientado a objetos no repasando ejemplos o resúmenes de ejemplos, sino trabajando en el diseño que les pareció más interesante en ese momento: el de ellos mismos.

Una vez que se tiene un conjunto de tarjetas CRC se desea crear una descripción más formal del diseño haciendo uso de UML[18]. No es necesario utilizar UML, pero puede ser de gran ayuda, especialmente si se desea poner un diagrama en la pared para que todo el mundo pueda ponderarlo, lo cual es una gran idea. Una alternativa a UML es una descripción textual de los objetos y sus interfaces, o, dependiendo del lenguaje de programación, el propio código[19].

UML también proporciona una notación para diagramas que permiten describir el modelo dinámico del sistema. Esto es útil en situaciones en las que las transiciones de estado de un sistema o subsistema son lo suficientemente dominantes como para necesitar sus propios diagramas (como ocurre en un sistema de control). También puede ser necesario describir las estructuras de datos, en sistemas o subsistemas en los que los datos sean un factor dominante (como una base de datos).

Sabemos que la Fase 2 ha acabado cuando se han descrito los objetos y sus interfaces. Bueno, la mayoría —hay generalmente unos pocos que quedan ocultos y que no se dan a conocer hasta la Fase 3. Pero esto es correcto. En lo que a uno respecta, esto es todo lo que se ha podido descubrir de los objetos a manipular. Es bonito descubrirlos en las primeras etapas del proceso pero la POO proporciona una estructura tal, que no presenta problema si se descubren más tarde. De hecho, el diseño de un objeto tiende a darse en cinco etapas, a través del proceso completo de desarrollo de un programa.

---

[18] Para los principiantes, recomiendo *UML Distilled*, 2.ª edición.

[19] Python *(http://www.Python.org)* suele utilizarse como "pseudocódigo ejecutable".

Las cinco etapas del diseño de un objeto

La duración del diseño de un objeto no se limita al tiempo empleado en la escritura del programa, sino que el diseño de un objeto conlleva una serie de etapas. Es útil tener esta perspectiva porque se deja de esperar la perfección; por el contrario, uno comprende lo que hace un objeto y el nombre que debería tener surge con el tiempo. Esta visión también se aplica al diseño de varios tipos de programas; el patrón para un tipo de programa particular emerge al enfrentarse una y otra vez con el problema (esto se encuentra descrito en el libro *Thinking in Patterns with Java*, descargable de *http://www.BruceEckel.com*). Los objetos, también tienen su patrón, que emerge a través de su entendimiento, uso y reutilización.

1. **Descubrimiento de los objetos**. Esta etapa ocurre durante el análisis inicial del programa. Se descubren los objetos al buscar factores externos y limitaciones, elementos duplicados en el sistema, y las unidades conceptuales más pequeñas. Algunos objetos son obvios si ya se tiene un conjunto de bibliotecas de clases. La comunidad entre clases que sugieren clases bases y herencia, puede aparecer también en este momento, o más tarde dentro del proceso de diseño.

2. **Ensamblaje de objetos**. Al construir un objeto se descubre la necesidad de nuevos miembros que no aparecieron durante el descubrimiento. Las necesidades internas del objeto pueden requerir de otras clases que lo soporten.

3. **Construcción del sistema**. De nuevo, pueden aparecer en esta etapa más tardía nuevos requisitos para el objeto. Así se aprende que los objetos van evolucionando. La necesidad de un objeto de comunicarse e interconectarse con otros del sistema puede hacer que las necesidades de las clases existentes cambien, e incluso hacer necesarias nuevas clases. Por ejemplo, se puede descubrir la necesidad de clases que faciliten o ayuden, como una lista enlazada, que contiene poca o ninguna información de estado y simplemente ayuda a la función de otras clases.

4. **Aplicación del sistema**. A medida que se añaden nuevos aspectos al sistema, puede que se descubra que el diseño previo no soporta una ampliación sencilla del sistema. Con esta nueva información, puede ser necesario reestructurar partes del sistema, generalmente añadiendo nuevas clases o nuevas jerarquías de clases.

5. **Reutilización de objetos**. Ésta es la verdadera prueba de diseño para una clase. Si alguien trata de reutilizarla en una situación completamente nueva, puede que descubra pequeños inconvenientes. Al cambiar una clase para adaptarla a más programas nuevos, los principios generales de la clase se mostrarán más claros, hasta tener un tipo verdaderamente reutilizable. Sin embargo, no debe esperarse que la mayoría de objetos en un sistema se diseñen para ser reutilizados —es perfectamente aceptable que un porcentaje alto de los objetos sean específicos del sistema para el que fueron diseñados. Los tipos reutilizables tienden a ser menos comunes, y deben resolver problemas más generales para ser reutilizables.

Guías para el desarrollo de objetos

Estas etapas sugieren algunas indicaciones que ayudarán a la hora de pensar en el desarrollo de clases:

1. Debe permitirse que un problema específico genere una clase, y después dejar que la clase crezca y madure durante la solución de otros problemas.

2. Debe recordarse que descubrir las clases (y sus interfaces) que uno necesita es la tarea principal del diseño del sistema. Si ya se disponía de esas clases, el proyecto será fácil.

3. No hay que forzar a nadie a saber todo desde el principio; se aprende sobre la marcha.Y esto ocurrirá poco a poco.

4. Hay que empezar programando; es bueno lograr algo que funcione de manera que se pueda probar la validez o no de un diseño. No hay que tener miedo a acabar con un código de estilo procedimental malo —las clases segmentan el problema y ayudan a controlar la anarquía y la entropía. Las clases malas no estropean las clases buenas.

5. Hay que mantener todo lo más simple posible. Los objetos pequeños y limpios con utilidad obvia son mucho mejores que interfaces grandes y complicadas. Cuando aparecen puntos de diseño puede seguirse el enfoque de una afeitadora de Occam: se consideran las alternativas y se selecciona la más simple, porque las clases simples casi siempre resultan mejor. Hay que empezar con algo pequeño y sencillo, siendo posible ampliar la interfaz de la clase al entenderla mejor. A medida que avance el tiempo será difícil eliminar elementos de una clase.

Fase 3: Construir el núcleo

Ésta es la conversión inicial de diseño pobre en un código compilable y ejecutable que pueda ser probado, y especialmente, que pueda probar la validez o no de la arquitectura diseñada. Este proceso no se puede hacer de una pasada, sino que consistirá más bien en una serie de pasos que permitirán construir el sistema de manera iterativa, como se verá en la Fase 4.

Su objetivo es encontrar el núcleo de la arquitectura del sistema que necesita implementar para generar un sistema ejecutable, sin que importe lo incompleto que pueda estar este sistema en esta fase inicial. Está creando un armazón sobre el que construir en posteriores iteraciones. También se está llevando a cabo la primera de las muchas integraciones y pruebas del sistema, a la vez que proporcionando a los usuarios una realimentación sobre la apariencia que tendrá su sistema, y cómo va progresando. Idealmente, se están además asumiendo algunos riesgos críticos. De hecho, se descubrirán posibles cambios y mejoras que se pueden hacer sobre el diseño original —cosas que no se hubieran descubierto de no haber implementado el sistema.

Una parte de la construcción del sistema es comprobar que realmente se cumple el análisis de requisitos y la especificación del sistema que realmente cumple el analisis de requisitos y la especificación del sistema (independientemente de la forma en que estén planteados). Debe asegurarse que las pruebas verifican los requerimientos y los casos de uso. Cuando el corazón del sistema sea estable, será posible pasar a la siguiente fase y añadir nuevas funcionalidades.

Fase 4: Iterar los casos de uso

Una vez que el núcleo del sistema está en ejecución, cada característica que se añada es en sí misma un pequeño proyecto. Durante cada *iteración*, entendida como un periodo de desarrollo razonablemente pequeño, se añade un conjunto de características.

¿Cúal debe ser la duración de una iteración? Idealmente, cada iteración dura de una a tres semanas (la duración puede variar en función del lenguaje de implementación). Al final de ese periodo, se tiene un sistema integrado y probado con una funcionalidad mayor a la que tenía previamente. Pero lo particularmente interesante es la base de la iteración: un único caso de uso. Cada caso de uso es un paquete de funcionalidad relacionada que se construye en el sistema de un golpe, durante una iteración. Esto no sólo proporciona una mejor idea de lo que debería ser el ámbito de un caso de uso, sino que además proporciona una validación mayor de la idea del mismo, dado que el concepto no queda descartado hasta después del análisis y del diseño, pues éste es una unidad de desarrollo fundamental a lo largo de todo el proceso de construcción de software.

Se deja de iterar al lograr la funcionalidad objetivo, o si llega un plazo y el cliente se encuentra satisfecho con la versión actual (debe recordarse que el software es un negocio de suscripción). Dado que el proceso es iterativo, uno puede tener muchas oportunidades de lanzar un producto, más que tener un único punto final; los proyectos abiertos trabajan exclusivamente en entornos iterativos de gran nivel de realimentación, que es precisamente lo que les permite acabar con éxito.

Un proceso de desarrollo iterativo tiene gran valor por muchas razones. Si uno puede averiguar y resolver pronto los riesgos críticos, los clientes pueden tener muchas oportunidades de cambiar de idea, la satisfacción del programador es mayor, y el proyecto puede guiarse con mayor precisión. Pero otro beneficio adicional importante es la realimentación a los usuarios, que pueden ver a través del estado actual del producto cómo va todo. Así es posible reducir o eliminar la necesidad de reuniones de estado "entumece-mentes" e incrementar la confianza y el soporte de los usuarios.

Fase 5: Evolución

Éste es el punto del ciclo de desarrollo que se ha denominado tradicionalmente "mantenimiento", un término global que quiere decir cualquier cosa, desde "hacer que funcione de la manera que se suponía que lo haría en primer lugar", hasta "añadir aspectos varios que el cliente olvidó mencionar", pasando por el tradicional "arreglar los errores que puedan aparecer" o "la adición de nuevas características a medida que aparecen nuevas necesidades". Por ello, al término "mantenimiento" se le han aplicado numerosos conceptos erróneos, lo que ha ocasionado un descenso progresivo de su calidad, en parte porque sugiere que se construyó una primera versión del programa en la cual hay que ir cambiando partes, además de engrasarlo para evitar que se oxide. Quizás haya un término mejor para describir lo que está pasando.

Prefiero el término *evolución*[20]. De esta forma, "uno no acierta a la primera, por lo que debe concederse la libertad de aprender y volver a hacer nuevos cambios". Podríamos necesitar muchos cambios a medida que vamos aprendiendo y comprendiendo con más detenimiento el problema. A corto y largo plazo, será el propio programa el que se verá beneficiado de este proceso continuo de evolución. De hecho, ésta permitirá que el programa pase de bueno a genial, haciendo que se aclaren aquellos aspectos que no fueron verdaderamente entendidos en la primera pasada. También es

[20] El libro de Martin Fowler *Refactoring: improving the design of existing code* (Addison-Wesley, 1999) cubre al menos un aspecto de la evolución, utilizando exclusivamente ejemplos en Java.

en este proceso en el que las clases se convierten en recursos reutilizables, en vez de clases diseñadas para su uso en un solo proyecto.

"Hacer el proyecto bien" no sólo implica que el programa funcione de acuerdo con los requisitos y casos de uso. También quiere decir que la estructura interna del código tenga sentido, y que parezca que encaja bien, sin aparentar tener una sintaxis extraña, objetos de tamaño excesivo o con fragmentos inútiles de código. Además, uno debe tener la sensación de que la estructura del programa sobrevivirá a los cambios que inevitablemente irá sufriendo a lo largo de su vida, y de que esos cambios se podrán hacer de forma sencilla y limpia. Esto no es trivial. Uno no sólo debe entender qué es lo que está construyendo, sino también cómo evolucionará el programa (lo que yo denomino el *vector del cambio*). Afortunadamente, los lenguajes de programación orientada a objetos son especialmente propicios para soportar este tipo de modificación continua —los límites creados por los objetos son los que tienden a lograr una estructura sólida. También permiten hacer cambios —que en un programa procedural parecerían drásticos— sin causar terremotos a lo largo del código. De hecho, el soporte a la evolución podría ser el beneficio más importante de la POO.

Con la evolución, se crea algo que al menos se aproxima a lo que se piensa que se está construyendo, se compara con los requisitos, y se ve dónde se ha quedado corto. Después, se puede volver y ajustarlo diseñando y volviendo a implementar las porciones del programa que no funcionaron correctamente[21]. De hecho, es posible que se necesite resolver un problema, o determinado aspecto de un problema, varias veces antes de dar con la solución correcta (suele ser bastante útil estudiar en este momento *el Diseño de Patrones)*. También es posible encontrar información en *Thinking in Patterns with Java*, descargable de *http://www.BruceEckel.com*).

La evolución también se da al construir un sistema, ver que éste se corresponda con los requisitos, y descubrir después que no era, de hecho, lo que se pretendía. Al ver un sistema en funcionamiento, se puede descubrir que verdaderamente se pretendía que solucionase otro problema. Si uno espera que se dé este tipo de evolución, entonces se debe construir la primera versión lo más rápidamente posible con el propósito de averiguar sin lugar a dudas qué es exactamente lo que se desea.

Quizás lo más importante que se ha de recordar es que por defecto, si se modifica una clase, sus súper y subclases seguirán funcionando. Uno no debe tener miedo a la modificación (especialmente si se dispone de un conjunto de pruebas, o alguna prueba individual que permita verificar la corrección de las modificaciones). Los cambios no tienen por qué estropear el programa, sino que cualquiera de las consecuencias de un cambio se limitarán a las subclases y/o colaboradores específicos de la clase que se modifica.

Los planes merecen la pena

Por supuesto, uno jamás construiría una casa sin unos planos cuidadosamente elaborados. Si construyéramos un hangar o la casa de un perro, los planes no tendrían tanto nivel de detalle, pero pro-

[21] Esto es semejante a la elaboración de "prototipos rápidos", donde se supone que uno construye una versión "rápida y sucia" que permite comprender mejor el sistema, pero que es después desechada para construirlo correctamente. El problema con el prototipado rápido es que los equipos de desarrollo no suelen desechar completamente el prototipo, sino que lo utilizan como base sobre la que construir. Si se combina, en la programación procedural, con la falta de estructura, se generan sistemas totalmente complicados, y difíciles de mantener.

bablemente comenzaríamos con una serie de esbozos que nos permitiesen guiar el proceso. El desarrollo de software ha llegado a extremos. Durante mucho tiempo, la gente llevaba a cabo desarrollos sin mucha estructura, pero después, comenzaron a fallar los grandes procesos. Como reacción, todos acabamos con metodologías que conllevan una cantidad considerable de estructura y detalle, eso sí, diseñadas, en principio, para estos grandes proyectos. Estas metodologías eran demasiado tediosas de usar —parecía que uno invertiría todo su tiempo en escribir documentos, y que no le quedaría tiempo para programar (y esto ocurría a menudo). Espero haber mostrado aquí una serie de sugerencias intermedias. Independientemente de lo pequeño que sea, es necesario *algún* tipo de plan, que redundará en una gran mejora en el proyecto, especialmente respecto del que se obtendría si no se hiciera ningún plan de ningún tipo. Es necesario recordar que en muchas estimaciones, falla más del 50 por ciento del proyecto (¡incluso en ocasiones se llega al 70 por ciento!).

Siguiendo un plan —preferentemente uno simple y breve— y siguiendo una estructura de diseño antes de la codificación, se descubre que los elementos encajan mejor, de modo más sencillo que si uno se zambulle y empieza a escribir código sin ton ni son. También se alcanzará un nivel de satisfacción elevado. La experiencia dice que al lograr una solución elegante uno acaba completamente satisfecho, a un nivel totalmente diferente; uno se siente más cercano al arte que a la tecnología. Y la elegancia siempre merece la pena; no se trata de una persecución frívola. De hecho, no solamente proporciona un programa más fácil de construir y depurar, sino que éste es mucho más fácil de entender y mantener, que es precisamente donde reside su valor financiero.

Programación extrema

Una vez estudiadas las técnicas de análisis y diseño, por activa y por pasiva durante mucho tiempo, quizás el concepto de programación extrema (*Extreme Programming*, XP) sea el más radical y sorprendente que he visto. Es posible encontrar información sobre él mismo en *Extreme Programming Explained*, de Kent Beck (Addison-Wesley 2000), que puede encontrarse también en la Web en *http://www.xprogramming.com*.

XP es tanto una filosofía del trabajo de programación como un conjunto de guías para acometer esta tarea. Algunas de estas guías se reflejan en otras metodologías recientes, pero las dos contribuciones más distintivas e importantes en mi opinión son "escribir las pruebas en primer lugar" y "la programación a pares". Aunque Beck discute bastante todo el proceso en sí, señala que si se adoptan únicamente estas dos prácticas, uno mejorará enormemente su productividad y nivel de confianza.

Escritura de las pruebas en primer lugar

El proceso de prueba casi siempre ha quedado relegado al final de un proyecto, una vez que "se tiene todo trabajando, pero hay que asegurarlo". Implícitamente, tenía una prioridad bastante baja, y la gente que se especializa en las pruebas nunca ha gozado de un gran estatus, e incluso suele estar ubicada en el sótano, lejos de los "programadores de verdad". Los equipos de pruebas se han amoldado tanto a esta consideración que incluso han llegado a vestir de negro, y han chismorreado alegremente cada vez que lograban encontrar algún fallo (para ser honestos, ésta es la misma sensación que yo tenía cada vez que lograba encontrar algún fallo en un compilador).

XP revoluciona completamente el concepto de prueba dándole una prioridad igual (o incluso mayor) que a la codificación. De hecho, se escriben los tests *antes* de escribir el código a probar, y los códigos se mantienen para siempre junto con su código destino. Es necesario ejecutar con éxito los tests cada vez que se lleva a cabo un proceso de integración del proyecto (lo cual ocurre a menudo, en ocasiones más de una vez al día).

Al principio la escritura de las pruebas tiene dos efectos extremadamente importantes.

El primero es que fuerza una definición clara de la interfaz de cada clase. Yo, en numerosas ocasiones he sugerido que la gente "imagine la clase perfecta para resolver un problema particular" como una herramienta a utilizar a la hora de intentar diseñar el sistema. La estrategia de pruebas XP va más allá —especifica exactamente qué apariencia debe tener la clase para el consumidor de la clase, y cómo ésta debe comportarse exactamente. No puede haber nada sin concretar. Es posible escribir toda la prosa o crear todos los diagramas que se desee, describiendo cómo debería comportarse una clase, pero nada es igual que un conjunto de pruebas. Lo primero es una lista de deseos, pero las pruebas son un contrato reforzado por el compilador y el programa en ejecución. Cuesta imaginar una descripción más exacta de una clase que la de los tests.

Al crear los tests, uno se ve forzado a pensar completamente en la clase, y a menudo, descubre la funcionalidad deseada que podría haber quedado en el tintero durante las experiencias de pensamiento de los diagramas XML, las tarjetas CRC, los casos de uso, etc.

El segundo efecto importante de escribir las pruebas en primer lugar, proviene de la ejecución de las pruebas cada vez que se construye un producto software. Esta actividad proporciona la otra mitad de las pruebas que lleva a cabo el compilador. Si se observa la evolución de los lenguajes de programación desde esta perspectiva, se llegará a la conclusión de que las verdaderas mejoras en lo que a tecnología se refiere han tenido que ver con las pruebas. El lenguaje ensamblador solamente comprobaba la sintaxis, pero C imponía algunas restricciones semánticas, que han evitado que se produzca cierto tipo de errores. Los lenguajes POO imponen incluso más restricciones semánticas, que miradas así no son, de hecho, sino métodos de prueba. "¿Se está utilizando correctamente este tipo de datos?", y "¿se está invocando correctamente a esta función?" son algunos de los tipos de preguntas que hace un compilador o un sistema en tiempo de ejecución. Se han visto los resultados de tener estas pruebas ya incluidas en el lenguaje: la gente parece ser capaz de escribir sistemas más completos y hacer que funcionen, con menos cantidad de tiempo y esfuerzo. He intentado siempre averiguar la razón, pero ahora lo tengo claro, son las pruebas: cada vez que se hace algo mal, la red de pruebas de seguridad integradas dice que hay un problema y determina dónde.

Pero las pruebas integradas permitidas por el diseño del lenguaje no pueden ir mucho más allá. En cierto punto, *cada uno* debe continuar y añadir el resto de pruebas que producen una batería de pruebas completa (en cooperación con el compilador y el sistema en tiempo de ejecución) que verifique todo el programa. Y, exactamente igual que si se dispusiera de un compilador observando por encima del hombro, ¿no desearía uno que estas pruebas le ayudasen a hacer todo bien desde el principio? Por eso es necesario escribir las pruebas en primer lugar y ejecutarlas cada vez que se reconstruya el sistema. Las pruebas se convierten en una extensión de la red de seguridad proporcionada por el lenguaje.

Una de las cosas que he descubierto respecto del uso de lenguajes de programación cada vez más y más potentes es que conducen a la realización de experimentos cada vez más duros, pues se sabe a priori que el propio lenguaje evitará pérdidas innecesarias de tiempo en la localización de errores. El esquema de pruebas XP hace lo mismo para todo el proyecto. Dado que se sabe que las pruebas localizarán cualquier problema que pueda aparecer en la vida del proyecto (y cada vez que se nos ocurra alguno), simplemente se introducen nuevas pruebas), es posible hacer cambios, incluso grandes, cuando sea necesario sin preocuparse de que éstos puedan cargarse todo el proyecto. Esto es increíblemente potente.

Programación a pares

La programación a pares (por parejas) va más allá del férreo individualismo al que hemos sido adoctrinados desde el principio, a través de las escuelas (donde es uno mismo el que fracasa o tiene éxito), de los medios de comunicación, especialmente las películas de Hollywood, en las que el héroe siempre lucha contra la conformidad sin sentido[22]. Los programadores, también, suelen considerarse abanderados de la individualidad —"los vaqueros codificadores" como suele llamarlos Larry Constantine. Y por el contrario, XP, que trata, de por sí, de luchar contra el pensamiento convencional, enuncia lo contrario, afirmando que el código debería siempre escribirse entre dos personas por cada estación de trabajo. Y esto debería hacerse en áreas en las que haya grupos de estaciones de trabajo, sin las barreras de las que la gente de facilidades de diseño suelen estar tan orgullosos. De hecho, Beck dice que la primera tarea para convertirse a XP es aparecer con destornilladores y llaves Allen y desmontar todo aquello que parezca imponer barreras o separaciones[23] (esto exige contar con un director capaz de hacer frente a todas las quejas del departamento de infraestructuras).

El valor de la programación en pareja es que una persona puede estar, de hecho, codificando mientras la otra piensa en lo que se está haciendo. El pensador es el que tiene en la cabeza todo el esbozo —y no sólo una imagen del problema que se está tratando en ese momento, sino todas las guías del XP. Si son dos las personas que están trabajando, es menos probable que uno de ellos huya diciendo "No quiero escribir las pruebas lo primero", por ejemplo. Y si el codificador se queda clavado, pueden cambiar de sitio. Si los dos se quedan parados, puede que alguien más del área de trabajo pueda contribuir al oír sus meditaciones. Trabajar a pares hace que todo fluya mejor y a tiempo. Y lo que probablemente es más importante: convierte la programación en una tarea mucho más divertida y social.

He comenzado a hacer uso de la programación en pareja durante los periodos de ejercitación en algunos de mis seminarios, llegando a la conclusión de que mejora significativamente la experiencia de todos.

---

[22] Aunque probablemente ésta sea más una perspectiva americana, las historias de Hollywood llegan a todas partes.

[23] Incluido (especialmente) el sistema PA. Trabajé una vez en una compañía que insistía en difundir a todo el mundo cualquier llamada entrante que recibieran los ejecutivos, lo cual interrumpía continuamente la productividad del equipo (pero los directores no podían empezar siquiera a pensar en prescindir de un servicio tan importante como el PA). Al final, y cuando nadie me veía, me encargué de cortar los cables de los altavoces.

Por qué Java tiene éxito

La razón por la que Java ha tenido tanto éxito es que su propósito era resolver muchos de los problemas a los que los desarrolladores se enfrentan hoy en día. El objetivo de Java es mejorar la productividad. Esta productividad se traduce en varios aspectos, pero el lenguaje fue diseñado para ayudar lo máximo posible, dejando en manos de cada uno la mínima cantidad posible, tanto de reglas arbitrarias, como de requisitos a usar en determinados conjuntos de aspectos. Java fue diseñado para ser práctico; las decisiones de diseño del lenguaje Java se basaban en proporcionar al programador la mayor cantidad de beneficios posibles.

Los sistemas son más fáciles de expresar y entender

Las clases diseñadas para encajar en el problema tienden a expresarlo mejor. Esto significa que al escribir el código uno está describiendo su solución en términos del espacio del problema, en vez de en términos del computador, que es el espacio de la solución ("Pon el bit en el chip que indica que el relé se va cerrar"). Uno maneja conceptos de alto nivel y puede hacer mucho más con una única línea de código.

El otro beneficio del uso de esta expresión es la mantenibilidad que (si pueden creerse los informes) se lleva una porción enorme del coste de un programa durante toda su vida. Si un programa es fácil de entender, entonces es fácil de mantener. Esto también puede reducir el coste de crear y mantener la documentación.

Ventajas máximas con las bibliotecas

La manera más rápida de crear un programa es utilizar código que ya esté escrito: una biblioteca. Uno de los principales objetivos de Java es facilitar el uso de bibliotecas. Esta meta se logra convirtiendo las bibliotecas en nuevos tipos de datos (clases), de forma que la incorporación de una biblioteca equivale a la inserción de nuevos tipos al lenguaje. Dado que el compilador de Java se encarga del buen uso de las bibliotecas —garantizando una inicialización y eliminación completas, y asegurando que se invoca correctamente a las funciones— uno puede centrarse en lo que desea que haga la biblioteca en vez de cómo tiene que hacerlo.

Manejo de errores

El manejo de errores en C es un importante problema, que suele ser frecuentemente ignorado o que se trata de evitar cruzando los dedos. Si se está construyendo un programa grande y complejo, no hay nada peor que tener un error enterrado en algún sitio sin tener ni siquiera una pista de dónde puede estar. El *manejo de excepciones* de Java es una forma de garantizar que se notifiquen los errores, y que todo ocurre como consecuencia de algo.

Programación a lo grande

Muchos lenguajes de programación "tradicionales" tenían limitaciones intrínsecas en lo que al tamaño y complejidad del programa se refiere. BASIC, por ejemplo, puede ser muy bueno para poner juntas soluciones rápidas para cierto tipo de problemas, pero si el programa se hace mayor de varias páginas, o se sale del dominio normal del problema, es como intentar nadar en un fluido cada vez más viscoso. No hay una línea clara que permita separar cuándo está fallando el lenguaje, y si la hubiera, la ignoraríamos. Uno no dice "Mi programa en BASIC simplemente creció demasiado; tendré que volver a escribirlo en C". Más bien se intenta meter con calzador unas pocas líneas para añadir alguna nueva característica. Por tanto, el coste extra viene dependiendo de uno mismo.

Java está diseñado para ayudar a *programar a lo grande* —es decir, para borrar esos límites de complejidad entre un programa pequeño y uno grande. Uno no tiene por qué usar POO al escribir un programa de utilidad del estilo de "¡Hola, mundo!", pero estas características siempre están ahí cuando son necesarias. Y el compilador se muestra agresivo a la hora de descubrir las causas generadoras de errores, tanto en el caso de programas grandes, como pequeños.

Estrategias para la transición

Si uno se introduce en la POO, la siguiente pregunta será probablemente "¿Cómo puedo hacer que mi director, mis colegas, mi departamento,... empiecen a usar objetos?". Uno debe pensar en cómo él mismo —un programador independiente— se sentiría a la hora de aprender un nuevo lenguaje y un nuevo paradigma de programación. A fin de cuentas, ya lo ha hecho antes. Lo primero es la educación y el uso de ejemplos, después viene un proyecto de prueba que proporcione una idea clara de los fundamentos sin hacer algo demasiado confuso. Después viene un proyecto "del mundo real" que, de hecho, haga algo útil. A lo largo de los primeros proyectos, uno sigue su educación leyendo y preguntando a los expertos, a la vez que solucionando pequeños inconvenientes con los colegas. Éste es el enfoque que muchos programadores experimentados sugieren de cara a migrar a Java. Cambiar una compañía entera, por supuesto implicaría la introducción de alguna dinámica de grupo, pero ayudará a recordar en cada paso cómo debería desenvolverse cada uno.

Guías

He aquí algunas ideas o guías a tener en cuenta cuando se haga la transición a POO y Java:

1. Formación

El primer paso es algún tipo de educación. Hay que recordar la inversión en código de la compañía, e intentar no tirar todo a la basura durante los seis a nueve meses que lleve a todo el mundo enterarse de cómo funcionan las interfaces. Es mejor seleccionar un pequeño grupo para adoctrinarles, compuesto preferentemente por personas curiosas, y que trabajen bien en grupo, que pueda luego funcionar como una red de soporte propia mientras se esté aprendiendo Java.

Un enfoque alternativo recomendado en ocasiones, es formar a todos los niveles de la compañía a la vez, incluidos cursos muy por encima para los directores de estrategia, además de cursos de diseño y programación para los constructores de proyectos. Esto es especialmente bueno para las pequeñas compañías que cambian continuamente la manera de hacer las cosas, o a nivel de divisiones en aquellas compañías de gran tamaño. Dado que el coste es elevado, sin embargo, hay que elegir empezar de alguna manera con la formación a nivel de proyecto, llevar a cabo un proyecto piloto (posiblemente con un formador externo) y dejar que el equipo de proyecto se convierta en el grupo de profesores del resto de la compañía.

2. Proyecto de bajo riesgo

Es necesario empezar con un proyecto de bajo riesgo y permitir los errores. Una vez que se ha adquirido cierta experiencia, uno puede alimentarse bien de proyectos de miembros del mismo equipo, o bien utilizar a los miembros del equipo como personal de soporte técnico para POO. Puede que el primer proyecto no funcione correctamente a la primera, por lo que no debería ser crítico con la misión de la compañía. Debería ser simple, independiente, e instructivo; esto significa que debería conllevar la creación de clases con significado para cuando les llegue el turno de aprender Java al resto de empleados de la compañía.

3. Modelo que ya ha tenido éxito

Es necesario buscar ejemplos con un buen diseño orientado a objetos en vez de empezar de la nada. Hay muchas posibilidades de que exista alguien que ya haya solucionado el problema en cuestión, o que si no lo ha solucionado del todo pueda aplicar lo ya aprendido sobre la abstracción para modificar un diseño ya existente en aras de que se ajuste a tus necesidades. Éste es el concepto general de los *patrones de diseño*, cubiertos en *Thinking in Patterns with Java*, descargable de *http://www.BruceEckel.com.*

4. Utilizar bibliotecas de clases existentes

La motivación económica principal para cambiar a POO es la facilidad de usar código ya existente en forma de bibliotecas de clases (en particular las bibliotecas estándares de Java, cubiertas completamente a lo largo de este libro). Se obtendrá el ciclo de desarrollo más pequeño posible cuando se puedan crear y utilizar objetos de bibliotecas preconfeccionadas. Sin embargo, algunos programadores novatos no entienden este concepto, y son inconscientes de la existencia de bibliotecas de clases. El éxito con la POO y Java será óptimo si se hace un esfuerzo para buscar y reutilizar el código ya desarrollado cuanto antes en el proceso de transición.

5. No reescribir en Java código ya existente

No suele ser la mejor de las ideas tomar código ya existente y que funcione y reescribirlo en Java (si se convierten en objetos, es posible interactuar con código ya escrito en C o C++ haciendo uso de la Interfaz Nativa Java (*Java Native Interface*) descrito en el Apéndice B). Hay beneficios incrementales, especialmente si el código va a ser reutilizado. Pero todas las opciones pasan porque no se darán los incrementos drásticos de productividad que uno pudiera esperar para su primer pro-

yecto a no ser que se acometa uno totalmente nuevo. Java y la POO brillan mucho más cuando se pasa de un proyecto conceptual al real correspondiente.

Obstáculos de gestión

Para aquél que sea director, su trabajo consiste en adquirir los recursos para el equipo, superar las barreras que puedan dificultar el éxito del equipo, y en general, intentar proporcionar el entorno más productivo que permita al equipo disfrutar y conseguir así, llevar a cabo esos milagros que siempre se exigen. Pasarse a Java implica estas tres características, y sería maravilloso que el coste fuera, además nulo. Aunque pasarse a Java puede ser más barato —en función de las limitaciones de cada uno— que las alternativas de la POO para un equipo de programadores en C (y probablemente para los programadores de otros lenguajes procedurales) no es gratuito, y hay obstáculos de los que uno debería ser consciente a la hora de vender el pasarse a Java dentro de una compañía y quedar totalmente embarrancado.

Costes iniciales

El coste de pasarse a Java es mayor que el de adquirir compiladores de Java (el compilador de Java de Sun es gratuito, así que éste difícilmente podría constituir un obstáculo). Los costes a medio y largo plazo se minimizan si se invierte en formación (y posiblemente si se utiliza un formador durante el primer proyecto) y también si se identifica y adquiere una biblioteca de clases que solucione el problema en vez de intentar construir esas bibliotecas uno mismo. Éstos son costes muy elevados que deben ser cuantificados en una propuesta realista. Además, están los costes ocultos de la pérdida de productividad implícita en el aprendizaje de un nuevo lenguaje, y probablemente un nuevo entorno de programación. La formación y la búsqueda de un formador pueden, a ciencia cierta, minimizar estos costes, pero los miembros del equipo deberán sobreponerse a sus propios problemas para comprender la nueva tecnología. Durante este proceso, ellos cometerán más fallos (éste es un aspecto importante, pues los errores reconocidos son la forma más rápida de aprender) y serán menos productivos. Incluso entonces, con algunos tipos de problemas de programación, las clases correctas y el entorno de desarrollo correcto, es posible ser más productivo mientras se está aprendiendo Java (incluso considerando que se están cometiendo más fallos y escribiendo menos líneas de código cada día) que si se continuara con C.

Aspectos de rendimiento

Una pregunta frecuente es "¿La POO hace que los programas se conviertan en más grandes y lentos automáticamente?". La respuesta es: "Depende". Los aspectos extra de seguridad de Java tradicionalmente han conllevado una penalización en el rendimiento, frente a lenguajes como C++. Las tecnologías como "hotspot" y las tecnologías de compilación han mejorado significativamente la velocidad en la mayoría de los casos, y se continúan haciendo esfuerzos para lograr un rendimiento aún mayor.

Cuando uno se centra en un prototipado rápido, es posible desechar conjuntamente componentes lo más rápido posible, a la vez que se ignoran ciertos aspectos de eficiencia. Si se utilizan bibliotecas de un tercero, éstas suelen estar optimizadas por el propio fabricante; en cualquier caso, esto no es un problema cuando uno está en modo de desarrollo rápido. Cuando se tiene el sistema deseado,

que sea lo suficientemente pequeño y rápido, entonces, ya está. Si no, hay que empezar a reescribir pequeñas porciones de código. Si a pesar de esto no se mejora, hay que pensar cómo hacer modificaciones en la implementacion subyacente, de forma que no haya ningún código que use una clase concreta que vaya a ser modificada. Sólo si no se encuentra ninguna otra solución al problema se acometerán posibles cambios en el diseño. El hecho de que el rendimiento sea crítico en esa porción del diseño es un indicador que debe formar parte del criterio de diseño principal. La utilización del desarrollo rápido ofrece la ventaja de poder averiguar esto muy pronto.

Si se encuentra una función que constituya un cuello de botella, es posible reescribirla en C/C++ haciendo uso de los *métodos nativos* de Java, sobre los que versa el Apéndice B.

Errores de diseño comunes

Cuando un equipo empieza a trabajar en POO y Java, los programadores cometerán una serie de errores de diseño comunes. Esto ocurre a menudo debido a que hay una realimentación insuficiente por parte de los expertos durante el diseño e implementación de los primeros proyectos, puesto que no han aparecido expertos dentro de la compañía y porque puede que haya cierta resistencia en la empresa para retener a los consultores. Es fácil que si alguien cree entender la POO desde las primeras etapas del ciclo trate de atajar a través de una tangente errónea. Algo que es obvio a los ojos de una persona experta en el lenguaje, puede llegar a constituir un gran problema o debate interno para un novato. Podría evitarse un porcentaje elevado de este trauma si se utilizara un experto externo experimentado como formador y consejero.

¿Java frente a C++?

Java se parece bastante a C++, y naturalmente podría parecer que C++ está siendo reemplazado por Java. Pero me empiezo a cuestionar esta lógica. Para algunas cosas, C++ sigue teniendo una serie de características que Java no tiene, y aunque ha habido muchas promesas de que algún día Java llegará a ser tan o más rápido que C++, hasta la fecha solamente hemos sido testigos de ligeras mejoras, sin innovaciones drásticas. También parece que sigue habiendo un interés continuo en C++, por lo que es improbable que este lenguaje desaparezca con el tiempo. (Los lenguajes siempre merodean por ahí. En uno de los "Seminarios de Java Intermedio/Avanzado" del autor Allen Holub afirmó que los lenguajes más comúnmente utilizados son Rexx y COBOL, en ese orden.)

Comienzo a pensar que la fuerza de Java reside en un ruedo ligeramente diferente al de C++. Éste es un lenguaje que no trata de encajar en un molde. Verdaderamente, se ha adaptado de distintas maneras para resolver problemas particulares. Algunas herramientas de C++ combinan bibliotecas, modelos de componente, y herramientas de generación de código para resolver el problema de desarrollar aplicaciones de ventanas para usuarios finales (para Microsoft Windows). Y sin embargo, ¿qué es lo que utilizan la gran mayoría de desarrolladores en Windows? Visual Basic (VB) de Microsoft. Y esto a pesar del hecho de que VB produce el tipo de código que se convierte en inmanejable en cuanto el programa tiene una ampliación de unas pocas páginas (además de proporcionar una sintaxis que puede ser incluso mística). VB es tan mal ejemplo de lenguaje de diseño como exitoso y popular. Por ello sería bueno disponer de la facilidad y potencia de VB sin que el resultado fuera código imposible de gestionar. Y es aquí donde Java debería destacar: como el "próxi-

mo VB". Uno puede estremecerse al leer esto, o no, pero al menos debería pensar en ello: es tan grande la porción de Java diseñada para facilitar la tarea del programador a la hora de enfrenarse a problemas de nivel de aplicación como las redes o las interfaces de usuario multiplataforma, y además tiene un diseño de lenguaje que hace posible la creación de bloques de código flexibles y de gran tamaño. Si se añade a esto el hecho de que Java es el lenguaje con los sistemas de comprobación de tipos y manejo de errores más robustos jamás vistas en un lenguaje, se tienen las bases para dar un gran paso adelante en lo que se refiere a productividad de la programación.

¿Debería utilizarse Java en vez de C++ para un proyecto determinado? En vez de *applets* de web, deben considerarse dos aspectos. El primero es que si se desea utilizar muchas de las bibliotecas de C++ ya existentes (logrando una considerable ganancia en productividad) o si se dispone de un código base ya existente en C o C++, Java podría ralentizar el desarrollo en vez de acelerarlo.

Si se está desarrollando el código por primera vez desde la nada, la simplicidad de Java frente a C++ acortará significativamente el tiempo de desarrollo —la evidencia anecdótica (historias de equipos que desarrollan en C++ y que siempre cuento a aquéllos que se pasan a Java) sugiere que se doble la velocidad de desarrollo frente a C++. Si el rendimiento de Java no importa o puede compensarse, los aspectos puramente de tiempo de lanzamiento hacen difícil justificar la elección de C++ frente a Java.

El aspecto más importante es el rendimiento. El código Java interpretado siempre ha sido lento, incluso entre 20 y 50 veces más lento que C en el caso de los primeros intérpretes de Java. Este aspecto, no obstante, ha mejorado considerablemente a lo largo del tiempo, aunque sigue siendo del orden de varias veces superior. Los computadores se fundamentan en la velocidad; si hacer algo en un computador no es considerablemente más rápido, lo hacemos a mano. (Incluso se sugiere que se empiece con Java, para reducir el tiempo de desarrollo, para posteriormente utilizar una herramienta y bibliotecas de soporte que permitan traducir el código a C++, cuando se necesite una velocidad de ejecución más rápida.)

La clave para hacer Java adecuado para la mayoría de proyectos de desarrollo es la aparición de mejoras en cuanto a velocidad, como los denominados compiladores *just-in-time* (JIT), la tecnología "hotspot" de Sun, e incluso compiladores de código nativo. Por supuesto, estos últimos eliminan la ejecución multiplataforma de los programas compilados, pero también proporcionan una mejora de velocidad al ejecutable, que se acerca a la que se lograría con C y C++. Y compilar un programa multiplataforma en Java sería bastante más sencillo que hacerlo en C o C++. (En teoría, simplemente es necesario recompilar, pero esto ya se ha prometido también antes en otros lenguajes de programación).

Es posible encontrar comparaciones entre Java y C++, y observaciones sobre las realidades de Java en los apéndices de la primera edición de este libro (disponible en el CD ROM que acompaña al presente texto, además de en *http://www.BruceEckel.com)*.

Resumen

Este capítulo trata de dar un repaso a los aspectos más importantes de la programación orientada a objetos y Java, incluyendo el porqué la POO es diferente, y por qué Java en particular es diferente,

conceptos de metodologías de POO, y finalmente las situaciones que se dan al hacer que una compañía pase a POO y Java.

La POO y Java pueden no ser para todo el mundo. Es importante evaluar las propias necesidades y decidir si Java podría satisfacer completamente esas necesidades, o si no sería mejor hacer uso de otro sistema de programación (incluyendo el que se esté utilizando actualmente). Si se sabe que las necesidades serán muy especializadas en un futuro próximo y que se tienen limitaciones específicas, puede que Java no sea la solución más satisfactoria, por lo que uno debe investigar las posibles alternativas[24]. Incluso si eventualmente se elige Java como lenguaje, uno debe al menos entender cuáles eran las opciones y tener una visión clara de por qué eligió dirigirse en esa dirección.

La apariencia de un lenguaje de programación procedural es conocida: definiciones de datos y llamadas a funciones. Para averiguar el significado de estos programas hay que invertir cierto tiempo, echando un vistazo a las llamadas a función y a conceptos de bajo nivel para crearse un modelo en la mente. Ésta es la razón por la que son necesarias representaciones intermedias al diseñar programas procedurales —por sí mismos, estos programas tienden a ser confusos porque los términos de expresión suelen estar más orientados hacia el computador que hacia el problema que se trata de resolver.

Dado que Java añade muchos conceptos nuevos sobre lo que tienen los lenguajes procedurales, es algo natural pensar que el método **main()** de un programa en Java será bastante más complicado que su equivalente en un programa en C. Se verá que las definiciones de los objetos que representan conceptos en el espacio del problema (en vez de hacer uso de aspectos de representación del computador) además de los mensajes que se envían a los mismos, representan las actividades en ese mismo espacio. Una de las maravillas de la programación orientada a objetos es ésa: con un programa bien diseñado, es fácil entender el código simplemente leyéndolo. Generalmente hay también menos código, porque muchos de los problemas se resolverán reutilizando código de las bibliotecas ya existentes.

[24] Recomiendo, en particular, echar un vistazo a Python (http://www.Python.org).