1: Introducción a los Objetos

"Cortamos la naturaleza dividiéndola, la organizamos en conceptos, y le atribuimos significados asimismo, en gran parte porque somos partes de un acuerdo que asentimos a través del discurso de nuestra comunidad y codificamos los patrones de nuestra lengua... no podemos hablar a todos excepto suscribiendo a la organización y clasificación de los datos que acordamos por decreto" Benjamin Lee Whorf (1897-1941)

La génesis de la revolución de la computadora se encuentra en la máquina. La génesis de nuestros lenguajes de programación tiende así a parecerse a esa máquina.

Pero las computadoras no son tanto máquinas como son herramientas que amplifican la mente ("bicicletas para la mente", como Steve Jobs está encariñado en decir) y una clase diferente de medio de expresión. Consecuentemente, las herramientas están comenzando a parecerse menos a las máquinas y más a las partes de nuestras mentes, y también como otras formas de expresión tales como escritura, pintura, escultura, animación, y filmación. La programación orientada al objeto (OOP) es parte de este movimiento para utilizar la computadora como medio de expresión.

Este capítulo lo introducirá a los conceptos básicos de OOP, incluyendo una descripción de los métodos de desarrollo. Este capítulo, y este libro, asumen que usted ha tenido experiencia en un lenguaje de programación procedural, aunque no necesariamente C. Si usted piensa que tiene necesidad de mayor preparación en la programación y la sintaxis de C antes de abordar este libro, debería trabajar a través del CDROM *Foundations for Java* que lo entrena, en el límite posterior de este libro.

Este capítulo es material de fondo y suplementario. Mucha gente no se siente cómoda vadeando la programación orientada al objeto sin entender el cuadro grande primero. Así, hay muchos conceptos que se introducen aquí para darle una descripción sólida de OOP. Sin embargo, la gente puede no comprender los conceptos del cuadro grande hasta el heber visto algo del mecanismo primero; esta gente puede empantanarse y perderse sin un cierto código para conseguir meter sus manos. Si usted forma parte de este último grupo y es impaciente en conseguir lo específico del lenguaje, siéntase libre de saltear más allá este capítulo - saltearlo en este punto no evitará que programe o aprenda la escritura del lenguaje. Sin embargo, usted deseará volver aquí para completar eventualmente su conocimiento así puede completar su conocimiento del porqué los objetos son importantes y cómo diseñar con ellos.

El Progreso de la Abstracción

Todos los lenguajes de programación proporcionan abstracciones. Puede ser discutido que la complejidad de los problemas capaz de solucionar está relacionada directamente con la clase y la calidad de la abstracción. Por la clase quiero decir "¿Qué es lo que está abstrayendo?". El lenguaje ensamblador es una pequeña abstracción de la máquina subyacente. Muchos lenguajes llamados "imperativos" que siguieron (por ejemplo FORTRAN, BASIC, y C) eran abstracciones del lenguaje ensamblador. Estos lenguajes son grandes mejoras del lenguaje ensamblador, pero su abstracción primaria todavía requiere pensar en términos de la estructura de la computadora antes que en la estructura del problema que está intentando resolver. El programador debe establecer la asociación entre el modelo de la máquina (en "el espacio de solución", donde está realizando el modelo de ese problema, tal como una computadora) y el modelo del problema que se está solucionando realmente (en "el espacio del problema", que es el lugar en donde existe el problema). El esfuerzo requerido para realizar este traspaso, y el hecho de que es extrínseco al lenguaje de programación, produce los programas que son difíciles de escribir y costosos de mantener, y como un efecto secundario crean la industria completa de los "métodos programación" .

La alternativa a modelar la máquina es modelar el problema que intenta resolver. Los primeros lenguajes tales como LISP y APL eligieron visiones particulares del mundo ("Los problemas todos son en última instancia listas" o "Los problemas todos son algorítmicos", respectivamente). PROLOG convierte todos los problemas en cadenas de decisiones. Los lenguajes han sido creados para la programación basada en limitaciones y para programar exclusivamente por la manipulación de símbolos gráficos. (El último provisto puede ser demasiado restrictivo). Cada uno de estos acercamientos son una buena solución a la clase particular aquellos problemas para los cuales se diseña la solución, pero cuando usted pasa fuera de este dominio, ellas se hacen torpes.

El acercamiento orientado al objeto va un paso más allá, proporcionando las herramientas para que el programador represente elementos en el espacio del problema. Esta representación es bastante general ya que el programador no está obligado a ningún tipo particular de problema. Nos referimos a los elementos en el espacio del problema y sus representaciones en el espacio de solución como "objetos". (Usted también necesitará otros objetos que no tengan análogos en el espacio del problema). La idea es que el programa esté habilitado para adaptarse a las continuas actualizaciones del problema agregando nuevos tipos de objetos, así cuando lea el código que describe la solución, las palabras de la lectura también expresarán el problema. Ésta es una abstracción más flexible y de mayor alcance del lenguaje las que teníamos antes. **[1]** .Así, la OOP permite que usted describa el problema en los términos del problema, antes que en los términos de la computadora en donde la solución funcionará. Todavía subyace una conexión a la computadora: cada objeto parece un poco a una computadora, que tiene un estado, operaciones que puede pedirle que se realicen. Sin embargo, esto no se parece a una mala analogía de los objetos en el mundo real - donde todo tiene características y comportamientos.

[1] Algunos diseñadores han decidido que la programación orientada al objeto por sí misma no es adecuada para resolver fácilmente todos los problemas de programación, y anuncian la combinación de aproximaciones diferentes dentro de los lenguajes de programación *multiparadigma*. Ver *Multiparadigm Programming in Leda* by Timothy Budd (Addison-Wesley 1995).

Alan Kay resumió cinco características básicas de Smalltalk, el primer lenguaje orientado al objeto certero y una de los lenguajes sobre los cuales se basa Java. Estas características representan un acercamiento puro a la programación orientada al objeto:

1. **Todo es un objeto.** Piense en un objeto como variable de fantasía; almacena datos, pero usted puede hacerle peticiones a ese objeto, pidiéndole realizar operaciones en sí mismo. En teoría, usted puede tomar cualquier componente conceptual en el problema que intenta resolver (los perros, los edificios, los servicios, etc.) y representarlo como objeto en su programa.
2. **Un programa es un manojo de objetos que se dicen entre ellos qué hacer enviándose mensajes.** . Para hacer una petición de un objeto, usted "envía un mensaje" a ese objeto. Más concretamente, usted puede pensar en un mensaje como una petición de llamar a un método que pertenezca a un objeto particular.
3. **Cada objeto tiene su propia memoria compuesta de otros objetos.** . Puesto de otra manera, usted crea una nueva clase de objeto haciendo un paquete que contiene objetos existentes. Así, usted puede construir complejidad en un programa mientras que la oculta detrás de la simplicidad de los objetos.
4. **Cada objeto tiene un tipo.** Utilizando el discurso, cada objeto es una *instancia* de un *class (clase)*, en la cual "clase" es sinónimo de "tipo". La distinción más importante y característica de una clase es "Qué mensajes puede usted enviarle".
5. **Todos los objetos de un tipo particular pueden recibir los mismos mensajes.** Esto es realmente una declaración cargada, como verá más adelante. Porque un objeto del tipo "círculo" es también un objeto de tipo "figura", un círculo está garantizado para aceptar mensajes de tipo figura. Esto significa que usted puede escribir el código que negocia a las figuras y automáticamente manejar cualquier cosa que quepa en la descripción de una figura. Esta sustitución es uno de los conceptos de mayor alcance en OOP.

Booch ofrece una descripción aún más sucinta de un objeto:

Un objeto tiene estado, comportamiento e identidad.

Esto significa que un objeto puede tener datos internos (que le da el estado), métodos (el comportamiento producido), y cada objeto puede ser únicamente distinguido de cualquier otro objeto - para poner esto en un sentido concreto, cada objeto tiene una dirección única en la asignación de la memoria. **[2]**

[2] Este es actualmente un poco restrictivo, ya que los objetos pueden existir concebiblemente en diferentes máquinas y espacios de direcciones, y pueden también ser almacenados en disco. En estos casos, la identidad del objeto debe ser determinado por algún otro que la dirección de memoria.

Un objeto tiene una interfaz

Aristotle es probablemente el primer que comenzó un estudio cuidadoso del concepto *tipo*; él habló de "la clase del los pescados y de la clase de pájaros". La idea que todos los objetos, mientras que sean únicos, son también parte de una clase de los objetos que tienen características y comportamientos en común fue utilizada directamente en el primer lenguaje orientada al objeto, Simula-67, con su palabra clave fundamental **class** que introduce un nuevo tipo en un programa.

Simula, como su nombre lo indica, fue creado para desarrollar las simulaciones tales como el problema clásico de "la caja del banco". En este, usted tiene un manojo de cajeros, clientes, cuentas, transacciones, y unidades monetarias - un conjunto de "objetos". Los objetos que son idénticos a excepción de su estado durante una ejecución de los programas se agrupan juntos en "clases de objetos" y es de donde la palabra clave **class** (clase) viene. Crear los tipos de datos abstractos (clases) es un concepto fundamental en la programación orientada al objeto. Los tipos de datos abstractos trabajan casi exactamente como tipos incorporados: usted puede crear variables de un tipo (llamado los objetos o las instancias en el discurso orientado al objeto) y manipular esas variables (denominado *"envío de mensajes"* o *"peticiones"*; usted envía un mensaje y el objeto calcula qué hacer con él). Los miembros (elementos) de cada parte de la clase tienen cierta concordancia: cada cuenta tiene un saldo, cada caja puede aceptar un depósito, etc. Al mismo tiempo, cada miembro tiene su propio estado: cada cuenta tiene un saldo diferente, cada caja tiene un nombre. Así, las cajas, clientes, cuentas, transacciones, etc., conservan cada uno su representante con una entidad única en el programa de computadora. Esta entidad es el objeto, y cada objeto pertenece a una clase particular que defina sus características y comportamientos.

Así pues, aunque lo que realmente hacemos en la programación orientada al objeto es crear nuevos tipos de datos, virtualmente todos los lenguajes de programación orientados al objeto utilizan la palabra clave "**class**". Cuando usted ve la palabra "tipo" piense en "clase" viceversa. **[3]**

[3] Algunas personas hacen una distinción, aclarando que tipo determina la interfaz mientras clase es una implementación particular de esa interfaz.

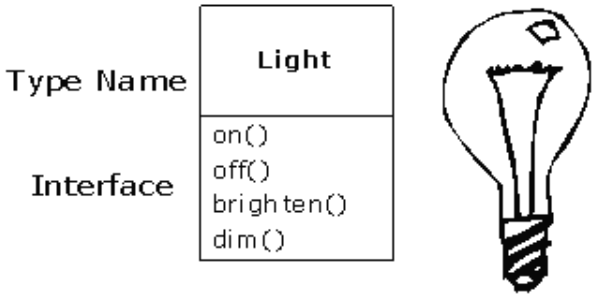
Puesto que una clase describe un sistema de objetos que tienen características idénticas (elementos de datos) y comportamientos (funcionalidad), una clase es realmente un tipo de datos porque un número de coma flotante, por ejemplo, también tiene un sistema de características y de comportamientos. La diferencia es que un programador define una clase para ajustarlo a un problema antes que ser forzado a utilizar un tipo de datos existente que fue

diseñado para representar una unidad del almacenaje en una máquina. Usted amplía el lenguaje de programación agregando nuevos tipos de datos específicos a sus necesidades. El sistema de programación da la bienvenida a las nuevas clases y les brinda todo el cuidado y la tipificación - comprobación de los tipos incorporados.

El acercamiento orientado al objeto no se limita a la construcción de simulaciones. Si o no usted convenga que cualquier programa es una simulación del sistema que diseña, el uso de las técnicas de OOP puede reducir fácilmente un sistema grande de problemas a una solución simple.

Una vez que una clase es establecida, usted poder hacer tantos objetos de esa clase como desee, y luego manipular estos objetos como si esos elemento existieran en el problema que esta intentando resolver. De hecho, uno de los desafíos de la programación orientada al objeto es crear uno a uno la relación entre elemento en el espacio del problema y objeto en el espacio de solución.

¿Pero cómo usted consigue que un objeto realice trabajo útil para usted? Debe haber una manera de hacer una petición al objeto de modo que haga algo, tal como completar una transacción, dibujar algo en la pantalla, o encender un interruptor. Y cada objeto puede satisfacer solamente ciertas peticiones. Las peticiones que usted puede hacer a un objeto son definidas por su *interfaz*, y el tipo es lo que determina la interfaz. Un ejemplo simple puede ser una representación de una bombilla:




```
Luz lt = new Luz();
lt.on();
```

La interfaz establece qué peticiones usted puede hacerle a un objeto particular. Sin embargo, debe haber código en alguna parte para satisfacer esta petición. Esto, junto con los datos ocultados, abarca la *implementación*. Desde un punto de vista de la programación procedural, no es tan complicado. Un tipo tiene un método asociado a cada petición posible, y cuando usted le hace una petición particular a un objeto, ese método es llamado. Este proceso es resumido generalmente diciendo usted está "enviando un mensaje" (haciendo una petición) a un objeto, y el objeto calcula qué hacer con ese mensaje (ejecuta código).

Aquí, el nombre del tipo/clase es **Luz**, el nombre de este objeto particular de **Luz** es **lt**, y las peticiones que usted puede hacer de un objeto **Luz** son encender, apagar, brillar, o desvanecer. Usted crea un objeto **Luz** definiendo una "referencia" (**lt**) para ese objeto y llama a **new** para solicitar un nuevo objeto de ese tipo. Para enviar un mensaje al objeto, usted indica el nombre del objeto y lo conecta con la petición del mensaje con un punto. Desde el punto de vista del usuario de una clase predefinida, esto es mucho que todo lo que está en la programación con los objetos.

El diagrama precedente sigue el formato del *Unified Modeling Language* (UML - lenguaje unificado de modelado). Cada clase está representada por una caja, con el nombre del tipo en la parte superior de la caja, de cualquier dato de los miembros usted cuidadosamente describe en la porción central de la caja, y de los *métodos* (las funciones que pertenecen a este objeto, que reciben mensajes cualesquiera que envía a ese objeto) en la parte inferior de la caja. A menudo, solamente el nombre de la clase y los métodos públicos se muestran en diagramas de diseño de UML, así que la porción central no está. Si está interesado solamente en el nombre de la clase, entonces la parte inferior no es necesario sea mostrada.

Un objeto provee servicios

Mientras usted intenta desarrollar o entender un diseño del programa, una de las mejores maneras de pensar sobre los objetos será como "abastecedores de servicios". Su programa mismo proporcionará servicios al usuario, y logrará esto usando los servicios ofrecidos por otros objetos. Su meta es producir (o aún mejor, localizar en bibliotecas existentes de código) un sistema de los objetos que proporcionan servicios ideales para solucionar su problema.

La manera de comenzar a hacer esto es pedir "Si pudiera sacar mágicamente de un sombrero, qué objetos solucionaría mi problema inmediatamente". Por ejemplo, supongamos que está creando un programa de contabilidad. Usted puede ser que imagine algunos objetos que contienen las pantallas de entrada predefinidas de contabilidad, otro sistema de objetos que realizan cálculos de contabilidad, y un objeto que maneje la impresión de cheques y de facturas en todas las diferentes clases de impresoras. ¿Algunos de *estos* objetos existen ya?, ¿y para los que no, cómo debieran ser? ¿Qué servicios proporcionarán *estos* objetos? y ¿qué objetos necesitarán para satisfacer sus obligaciones? Si usted se mantiene haciendo esto, alcanzará eventualmente un punto donde puede decir "que cualquier objeto que parece bastante simple para sentarse y escribir" o "estoy seguro que el objeto ya existe". Esta es una manera razonable de descomponer un problema en un sistema de objetos.

El pensar en un objeto como proveedor de servicio tiene una ventaja adicional: ayuda a mejorar la cohesión del objeto. La alta cohesión es una calidad fundamental del diseño de software: Significa que varios aspectos del componente de software (tal como un objeto, a aunque esto podría también aplicarse a un método o una biblioteca de objetos) está "encajando en conjunto" muy bien. Un problema que la persona tiene cuando diseña objetos

es que está abarrotando demasiada funcionalidad en un objeto. Por ejemplo, sobre su módulo que imprime el cheque, puede decidir la necesidad de un objeto que sepa todo sobre formato e impresión. Descubre probablemente que esto es demasiado para un objeto, y que necesita tres o más objetos. Un objeto puedo ser un catálogo de todas las disposiciones posibles del cheque, al que se puede preguntar sobre la información de cómo imprimir un cheque. +Un objeto o sistema de objetos podría ser una interfaz de impresión genérico que sabe todos sobre diversas clases de impresoras (absolutamente nada sobre contabilidad - este es un candidato a comprar antes que la escritura misma). Y un tercer objeto podría utilizar los servicios de los otros dos para lograr la tarea. Así, cada objeto tiene un sistema de cohesión de servicios que ofrece. En un buen diseño orientado al objeto, cada objeto hace una cosa bien, pero no intente hacer demasiado. Según lo visto aquí, esto permite no solamente el descubrimiento de los objetos que se pudieron comprar (el objeto del interfaz de la impresora), sino también la posibilidad de producir un objeto que se puede reutilizar en otra parte (el catálogo de las disposiciones del cheque).

Tratar los objetos como proveedores de servicio son una gran herramienta de simplificación, y son muy útiles no solamente durante el proceso del diseño, sino también cuando alguien intenta entender su código o reutilizar un objeto - si pueden ver el valor del objeto basado en qué servicio proporciona, ello hace mucho más fácil hacerlo caber en el diseño.

Esconder la implementación

Es útil dividir el campo jugado por *los creadores de clases* (éstos que crean nuevos tipos de datos) y *los programadores de clientes* **[4]** (los consumidores de clase que utilizan los tipos de datos en sus aplicaciones). La meta del programador de cliente es recoger una caja de herramientas completa de clases para utilizar en el desarrollo rápido de aplicaciones. La meta del creador de clase es construir una clase que exponga solamente lo que es necesario al programador de cliente y mantener el resto ocultado. ¿Por qué? Porque si son ocultados, no puede acceder el programador de cliente a él, lo que significa que el creador de la clase puede cambiar la porción ocultada a voluntad sin la preocupación del impacto en otra persona. La porción ocultada representa generalmente los interiores blandos de un objeto que se podrían corromper fácilmente un programador de cliente descuidado o mal informado, así que ocultar la implementación reduce los errores de programación.

[4] estoy agradecido a mi amigo Scott Meyers por este término.

El concepto de ocultar la implementación no puede ser sobre acentuada. En cualquier relación es importante tener límites que son respetados por todos las partes implicadas. Cuando usted crea una biblioteca, establece una relación con el programador del cliente, quien es también programador, aunque uno que está juntando y usando su biblioteca en una aplicación, construyendo posiblemente una biblioteca más grande. Si todos los miembros de una clase están disponibles, entonces el programador de cliente puede hacer cualquier

cosa con esa clase y no hay ninguna manera de hacer cumplir las reglas. Aunque usted puede ser que realmente prefiera que el programador de cliente no manipule directamente algún miembro de su clase, sin control de acceso no hay ninguna manera de prevenirla. Todo estará descubierto al mundo.

La primera razón del control de acceso es guardar de las manos de programadores de cliente las porciones que ellos no debieran tocar - que son necesarios para la operación interna del tipo de datos pero no parte de la interfaz que los usuarios necesitan para solucionar sus problemas particulares. Esto es realmente un servicio a los usuarios porque pueden ver fácilmente qué es importantes para ellos y qué pueden ignorar.

La segunda razón del control de acceso es permitir que el diseñador de la biblioteca cambie los funcionamientos internos de la clase sin la preocupación de cómo afectará al programador de cliente. Por ejemplo, usted puede ser que ponga una clase singular en ejecución en una manera simple para facilitar el desarrollo, y después descubre necesita reescribirlo para hacer que funcione más rápidamente. Si la interfaz y la implementación se separan y se protegen claramente, usted puede lograr esto fácilmente.

Java utiliza tres palabras claves para fijar los límites en una clase: **public**, **private**, y **protected**. Su uso y significado son absolutamente directos. Estos especificadores del acceso determinan quién puede utilizar las definiciones que siguen. **public** quiere significar que el elemento siguiente está disponible para todos. La palabra clave **private**, por otra parte, significa que nadie puede tener acceso a ese elemento excepto usted, el creador del tipo, de los métodos de este tipo. El tipo **private** es una pared del ladrillo entre usted y el programador de cliente. Alguien que intenta tener acceso a un miembro **private** conseguirá un error en tiempo de compilación. La palabra clave **protected** actúa como **private**, con excepción del hecho que una clase heredera tiene el acceso a los miembros **protected**, pero no a miembros **private**. La herencia será introducida pronto.

Java también tiene un acceso por omisión, que viene en juego si usted omite usar los especificadores ya mencionados. Esto generalmente se llama *package access*, acceso de paquete, porque las clases pueden tener acceso a los miembros de otras clases en el mismo paquete, pero fuera del paquete esos mismos miembros aparecen como **private**.

Reutilizar la implementación

Una vez que se haya creado y probado una clase, debiera (idealmente) representar una unidad útil del código. Resulta que esta reutilización no es tan fácil de alcanzar como se esperaría; toma experiencia y perspicacia producir un diseño reutilizable del objeto. Pero una vez que usted tenga tal diseño, puede ser reutilizado. La reutilización de código es una de las ventajas más grandes que los lenguajes de programación orientados al objeto proporcionan.

La manera simple de reutilizar una clase es utilizar un objeto de esta clase directamente, pero también puede ubicar un objeto de esa clase dentro de una clase nueva. Llamamos a esto "creación de un objeto miembro". Su nueva clase puede ser hecha sobre cualquier número y tipo de otros objetos, en cualquier combinación que necesite para agregar la funcionalidad deseada en su nueva clase, este concepto es llamado *composición* (si la composición sucede dinámicamente, es también llamada *agregación*). La composición está siempre referida como una relación "tiene un", como en "un auto tiene un motor".

(Este diagrama de UML indica la composición con el rombo lleno, que indica que hay un coche. Utilizaré típicamente una forma más simple: apenas una línea, sin el rombo, para indicar una asociación. **[5]**)

[5] Esto es usualmente suficiente detalle para la mayoría de los diagramas, y no necesita obtener específico sobre si está usando agregación o composición.

La composición viene con mucha flexibilidad. Los objetos del miembro de su nueva clase son normalmente privados, haciéndolos inaccesibles a los programadores de cliente que están utilizando la clase. Esto permite que usted cambie a esos miembros sin interferir en el código existente del cliente. Usted puede también cambiar los objetos del miembro en el tiempo de ejecución, para cambiar dinámicamente el comportamiento de su programa. La herencia, que se describe después, no tiene esta flexibilidad puesto que el compilador debe poner restricciones en tiempo de compilación en las clases creadas con herencia.

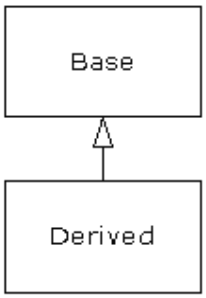
Porque la herencia es tan importante en la programación orientado al objeto a menudo se es altamente acentuada, y el nuevo programador puede conseguir la idea que la herencia se deba utilizar por todas partes. Esto puede dar lugar a diseños torpes y excesivamente complicados. En su lugar, usted debe primero mirar la composición al crear nuevas clases, puesto que es más simple y más flexible. Si usted toma este acercamiento, sus diseños estarán más clarificados. Una vez que el usted tenga cierta experiencia, verá razonablemente obvio cuando necesita la herencia.

Herencia: reutilización de la interfaz

Por sí misma, la idea de un objeto es una herramienta conveniente. Permite que usted empaquete datos y funcionalidad junta por conceptos, así puede representar una idea apropiada del espacio del problema antes que ser forzado a utilizar los idiomas de la máquina subyacente. Estos conceptos son

expresados como unidades fundamentales en el lenguaje de programación usando la palabra clave **class**

se parece una compasión, sin embargo, ir a todo el apuro a crear una clase y después ser forzado a crear una nueva forma que puede tener una funcionalidad similar. Resulta más agradable si podemos tomar la clase existente, la reproducimos, y después hacemos agregados y modificaciones a su copia. Esta es efectivamente lo que usted consigue con la *herencia*, a excepción del hecho que si la clase original (llamada *clase base* o *superclase* o *la clase padre*) se cambia, el "clon" modificado (llamada *clase derivada* o *clase heredada* o *subclase* o *clase hijo*) también refleja esos cambios.



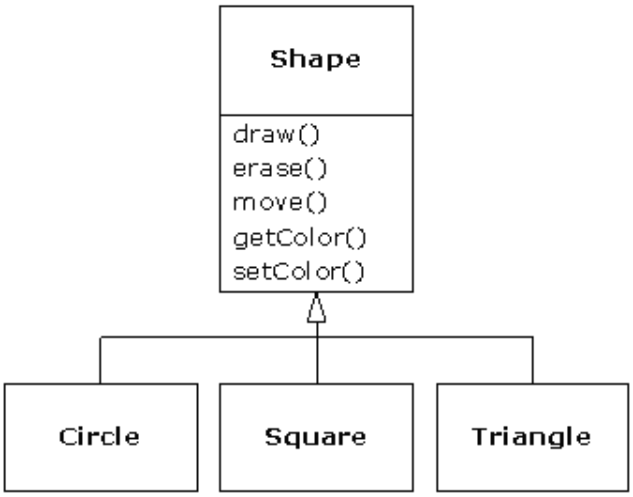
(La flecha de este diagrama UML apunta desde la clase derivada hacia la clase baja. Como usted verá, hay comúnmente más de una clase derivada.)

El tipo hace más que describir las contracciones en un sistema 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 puede también manejar más mensajes (o manejarlos diferentemente). La herencia expresa esta semejanza entre los tipos usando el concepto de tipos bases y de tipos derivados. Un tipo base contiene todas las características y comportamientos que se comparten entre los tipos derivados de él. Usted crea un tipo base para representarlo dentro de sus ideas sobre algunos objetos de su sistema. Del tipo base, usted deriva otros tipos para expresar las distintas maneras en que esta base puede ser observada.

Por ejemplo, una máquina de reciclaje de basura clasifica pedazos de basura. El tipo base es "basura" y que cada porción de basura tiene un peso, un valor, etcétera, y puede ser destruida, derretida, o descompuesta. Después de esto, tipos más específicos de basura son derivados y pueden tener características adicionales (una botella tiene un color) o comportamientos (una lata del aluminio se puede machacar, una lata de acero es magnética). Además, algunos comportamientos pueden ser diferentes (el valor del papel depende de su tipo y condición). Usando herencia, usted puede construir una jerarquía del tipo que exprese EL problema que intenta solucionar en términos de sus tipos.

Un segundo ejemplo es la "figura" clásica, quizás utilizado en un sistema de diseño automatizado o simulación de juego. El tipo base es "figura" y cada una tiene un tamaño, un color, una posición, etcétera. Cada figura se puede dibujar, borrar, mover, colorear, etc. Desde esta, los tipos específicos de

figuras son derivados (heredados) - círculo, cuadrado, triángulo, etcétera - cada uno de las cuales pueden tener características y comportamientos adicionales. Ciertas figuras se pueden estirarse, por ejemplo. Algunos comportamientos pueden ser diferentes, por ejemplo cuando usted desea calcular el área de una figura. La jerarquía del tipo incorpora las semejanzas y las diferencias entre las figuras.



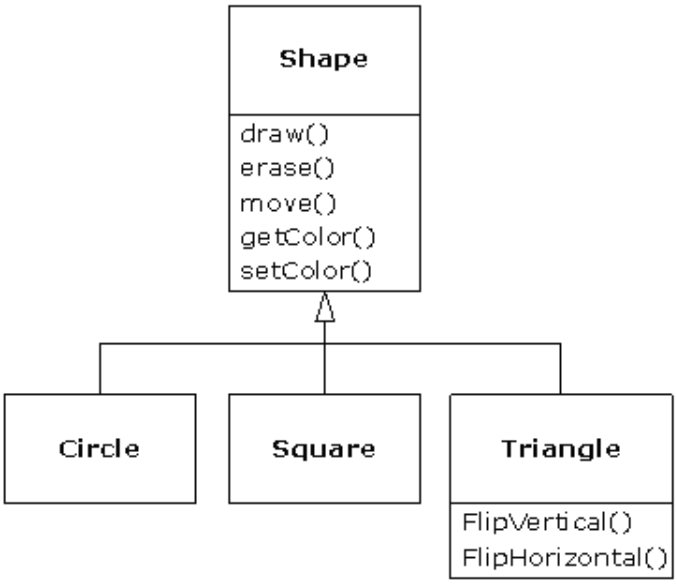
Modelar la solución en los mismos términos que el problema es enormemente beneficioso porque no tiene mucha necesidad de modelos intermedios para conseguir una descripción del problema a una descripción de la solución. Con los objetos, el tipo jerárquico es el primer modelo, así que usted va directamente de la descripción del sistema en el mundo real a la descripción del sistema en código. De hecho, una de las dificultades que la gente tiene con el diseño orientado al objeto es que es demasiado simple conseguir desde el principio al final. Una mente entrenada para buscar soluciones complejas puede inicialmente estar encantada por esta simplicidad.

Cuando hereda de un tipo existente, usted crea un nuevo tipo. Este nuevo tipo contiene no solamente a todos los miembros del tipo existente (aunque los **private** están inaccesibles y ocultados), pero lo más importante duplica la interfaz de la clase base. Es decir, todos los mensajes que usted puede enviar a los objetos de la clase base puede también enviarlos a los objetos de la clase derivada. Puesto que sabemos el tipo de una clase por los mensajes que podemos enviarle, esto significa que la clase derivada *es del mismo tipo que la clase base*. En el ejemplo anterior, "un círculo es una figura". Este tipo de equivalencia vía herencia es una de las entradas fundamentales para entender el significado de la programación orientada al objeto.

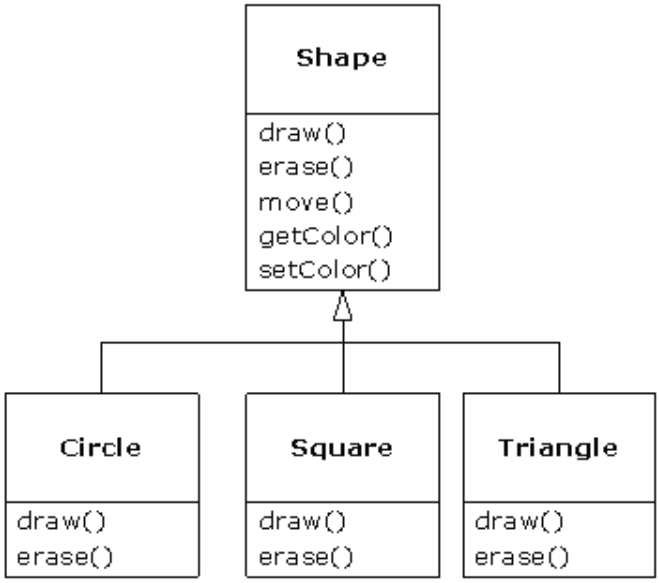
Puesto que la clase base y la clase derivada tienen la misma interfaz fundamental, debe haber una cierta implementación que vaya junto con esa interfaz. Es decir, debe haber cierto código para ejecutarse cuando un objeto recibe un mensaje particular. Si usted hereda simplemente una clase y no le hace algo, los métodos de la interfaz de la clase base vienen directamente a

instalarse en la clase derivada. Esto quiere decir que los objetos de la clase derivada tienen no solamente el mismo tipo, sino también el mismo comportamiento, que no interesa particularmente.

Usted tiene dos maneras de distinguir su nueva clase derivada de la clase base original. El primer es absolutamente directo: agrega simplemente nuevos métodos de marca a la clase derivada. Estos nuevos métodos no son parte de la interfaz de la clase base. Esto significa que la clase base no hace simplemente tanto como lo que desea, así que usted agrega más métodos. Este uso simple y primitivo de la herencia es, ocasionalmente, la solución perfecta a su problema. Sin embargo, usted debe mirar de cerca por la posibilidad de que su clase base puede también necesitar estos métodos adicionales. Este proceso de descubrimiento y de iteración de su diseño sucede regularmente en la programación orientada al objeto.



Aunque la herencia puede implicar a veces (especialmente en Java, donde está la palabra clave **extends** para herencia) que usted va a agregar los nuevos métodos a la interfaz, esto no es necesariamente verdad. La segunda y más importante forma de distinguir su nueva clase es *cambiar* el comportamiento de un método existente de la clase base. Esto es referido *sobrecarga* de aquél método.



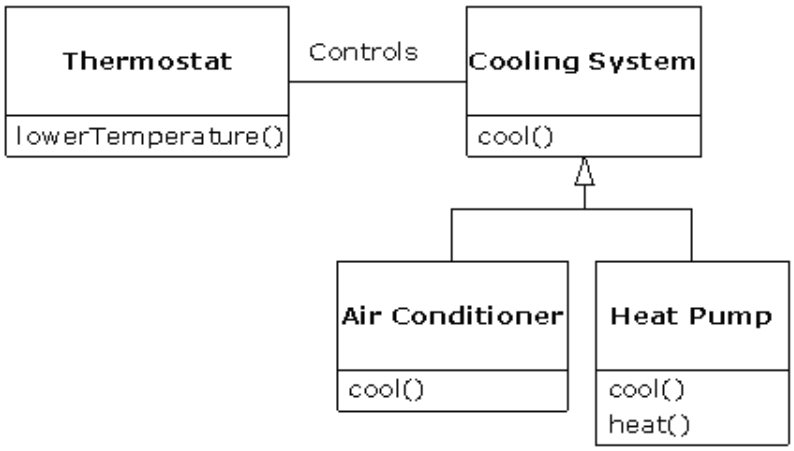
Para sobrecargar un método, usted crea simplemente una nueva definición de ese método en la clase derivada. Usted dice: "estoy usando el mismo método de la interfaz aquí, pero yo quisiera que hiciera algo diferente para mi nuevo tipo".

Relaciones Es un contra ser-como-uno

Hay cierta discusión que puede ocurrir sobre la herencia: ¿"No debiera la herencia *solamente* sobrecargar los métodos de la clase base (y no agregar nuevos métodos que no están en la clase base)"? Esto significaría que el tipo derivado es *exactamente* del mismo tipo que la clase base puesto que tiene exactamente la misma interfaz. Consecuentemente, usted puede sustituir exactamente un objeto de la clase derivada por un objeto de la clase base. Esto se puede pensar como *substitución pura*, y son designados a menudo el *principio de la substitución*. En un sentido, ésta es la manera ideal de tratar la herencia. Nos referimos a menudo a la relación entre la clase base y las clases derivadas como un caso de la relación *"es un"* , porque usted puede decir "el círculo *es una* figura". Una prueba para la herencia es determinar si puede indicar la relación "es un" sobre las clases y hacer que tenga sentido.

Hay momentos en que debe agregar nuevos elementos a la interfaz de un tipo derivado, extendiendo la interfaz y creando un nuevo tipo. El nuevo tipo se puede todavía sustituir por el tipo base, pero la substitución no es perfecta porque sus nuevos métodos no son accesibles desde el tipo base. Esto se puede describir como una relación *ser-como-un*(mi término). El nuevo tipo tiene la interfaz del tipo anterior pero también contiene otros méto dos, así que no puede decir que son realmente iguales con exactitud. Por ejemplo, considere un acondicionador de aire. Supongamos que su casa cableada con todos los controles para refrescarse; es decir, tiene una interfaz que permita que usted controle el frío. Imagínese que el acondicionador de aire se rompe y lo substituye por una bomba de calor, que puede calentar y refrescar. La

bomba de calor *es como un* acondicionador de aire, pero puede hacer mucho más. Porque el sistema de control de su casa está diseñado para controlar solamente el frío, se encuentra restringida a comunicarse con la parte que refresca del nuevo objeto. La interfaz del nuevo objeto ha sido extendida, y el sistema existente no sabe sobre otra cosa que no sea la interfaz original.



Por supuesto, una vez que usted vea este diseño llega a ver claramente que el "sistema de frío" de la clase base no es tan general, y debe ser retitulado como "sistema de control de temperatura" de modo que pueda también incluir calor - en este punto trabajará el principio de la substitución. Sin embargo, este diagrama es un ejemplo de qué puede suceder en el diseño del mundo real.

Cuando usted ve que el principio de la substitución es fácil sentirá como este acercamiento (substitución pura) es la única manera de hacer cosas, y de hecho es agradable si su diseño se resuelve de esa manera. Pero hallará que hay veces en que es igualmente claro que debe agregar nuevos métodos a la interfaz de una clase derivada. Con la inspección ambos casos deben ser razonablemente obvios.

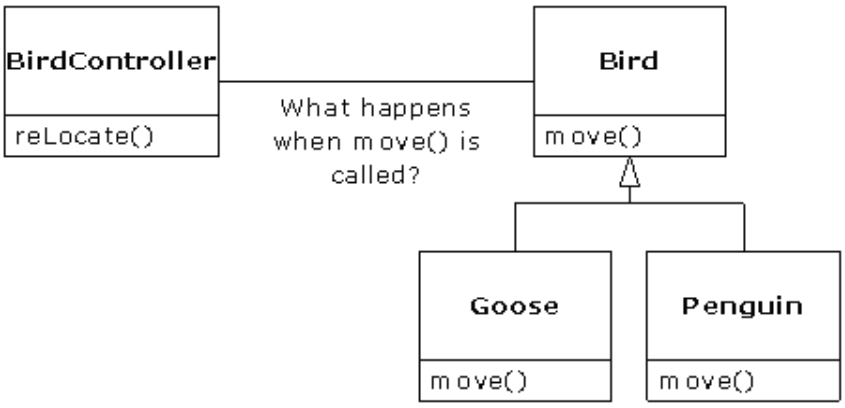
Intercambio de objetos con polimorfismo

Cuando se ocupa de jerarquías de tipo, usted desea a menudo tratar un objeto no como el tipo específico que es, sino que por el contrario como su tipo base. Esto permite que usted escriba código que no depende de un tipo específico. En el ejemplo de la figura, los métodos manipulan formas genéricas sin preocuparse si el ellos son círculos, cuadrados, triángulos, o una figura cualquiera que aún no ha sido definida todavía. Todas las figuras pueden ser dibujadas, borradas y movidas, así que estos métodos simplemente envían un mensaje a un objeto figura; sin preocuparse sobre cómo el objeto hace frente al mensaje.

Tal código no es afectado por la adición de nuevos tipos, y el agregado de nuevos tipos es la forma más común de ampliar un programa orientado al

objeto para manejar nuevas situaciones. Por ejemplo, puede derivar un nuevo subtipo de figura llamado pentágono sin modificar los métodos que se ocupan solamente de figuras genéricas. Esta capacidad de ampliar fácilmente un diseño derivando nuevos subtipos es una de las maneras fundamentales de encapsular el cambio. Esto mejora grandemente los diseños mientras que reduce el costo de mantener el software.

Hay un problema, sin embargo, con intentar tratar objetos del tipo derivados como sus tipos bases genéricos (círculos como formas, bicicletas como vehículos, cormoranes como pájaros, etc.). Si un método le dice a una figura genérica dibujarse, o un vehículo genérico encender, o a un pájaro genérico moverse, el compilador no puede saber en tiempo de compilación qué porción del código será ejecutada exactamente. Este es el punto central - cuando el mensaje es enviado, el programador no *desea* conocer qué porción del código será ejecutado; el método dibujar se puede aplicar igualmente a un círculo, a un cuadrado, o a un triángulo, y el objeto ejecutará el código apropiado dependiendo de su tipo específico. Si usted tiene que saber qué porción del código será ejecutado, entonces cuando agregue un nuevo subtipo, el código que se ejecuta puede ser diferente sin requerir cambios en la llamada al método. Por lo tanto, el compilador no puede saber exactamente qué porción del código se ejecuta, ¿así qué es lo que hace? Por ejemplo, en el diagrama siguiente el objeto de **ControlarAve** trabaja con los objetos genéricos de **Ave** y no sabe de qué tipo exacto son. Esto es conveniente desde la perspectiva de **ControlarAve** porque no tiene que escribir código especial para determinar el tipo exacto de de **Ave** con que trabaja o ese comportamiento de **Ave**. Por tanto ¿cómo sucede esto, cuando se llama a **mover()** mientras que ignora el tipo específico de **Ave**, que el comportamiento correcto ocurra. (un **Ganso** corre, vuela, o nada, y un **pingüino** corre o nada)?



La respuesta es el primer giro en la programación orientada al objeto: el compilador no puede hacer una llamada de la función en el sentido tradicional. La llamada de la función generada por un compilador de no-OOP causa lo que se llama *amarre temprano*, un término que usted pudo no haber oído antes porque nunca pensó en él de otra manera. Significa que el compilador genera una llamada a un nombre específico de la función y el linqueador resuelve esta llamada a la dirección absoluta del código que se ejecutará. En OOP, el programa no puede determinar la dirección del código hasta tiempo de

ejecución, así que cierto esquema diferente es necesario cuando un mensaje se envía a un objeto genérico.

Para solucionar el problema, los lenguajes orientados al objeto utilizan el concepto del *amarre final*. Cuando envía un mensaje a un objeto, el código que está siendo llamado no se determina hasta tiempo de ejecución. El compilador se asegura de que el método exista y realiza la comprobación del tipo sobre los argumentos y el valor de retorno (un lenguaje en el cual esto no es verdad se llama de *tipificación débil*), pero no conoce el código exacto para ejecutarse.

Para realizar el amarre final, Java utiliza un bit especial del código en lugar de la llamada absoluta. Este código calcula la dirección del cuerpo del método, usando información almacenada en el objeto (este proceso se cubre en gran detalle en el capítulo 7). Así, cada objeto puede comportarse diferentemente según el contenido de ese bit especial del código. Cuando envía un mensaje a un objeto, el objeto calcula qué hacer con ese mensaje.

En algunos lenguajes usted debe indicar explícitamente que quisiera que un método tuviera para obtener la flexibilidad de las características amarre final (en las aplicaciones de C++ la palabra clave **virtual** hace esto). En estos lenguajes, por defecto, los métodos *no* son limitados dinámicamente. En Java, el amarre dinámico es el comportamiento por defecto y usted no tiene necesidad de recordar agregar otra palabra clave adicional para conseguir polimorfismo.

Considera el ejemplo de la figura. La familia de clases (basadas todas en la misma interfaz uniforme) está diagramada al principio de este capítulo. Para demostrar polimorfismo, deseamos escribir una sola porción de código que no hace caso de los detalles específicos del tipo y se encarga solamente con la clase base. Que el código esté *des-unido* de la información específica de tipo, así es más simple de escribir y más fácil de entender. Y, si un nuevo tipo - un **Hexágono**, por ejemplo - es agregado por herencia, el código que usted escribe trabajará tan bien para el nuevo tipo de **Figura** como en los tipos existentes. Así, el programa es *extensible*.

Si usted escribe un método en Java (pronto aprenderá cómo hacerlo):

```
void
 hacerAlgo(Figura f) {
  f.borrar();
  // ...
  f.dibujar();
}
```

Este método habla a cualquier **Figura**, así que es independiente del tipo específico de objeto que dibuja y borra. Si alguna otra parte del programa utiliza el método del **hacerAlgo()**:

```
Círculo c = new Círculo();
Triángulo t = new Triángulo();
```

```
Línea l = new Línea();
hacerAlgo(c);
hacerAlgo(t);
hacerAlgo(l);
```

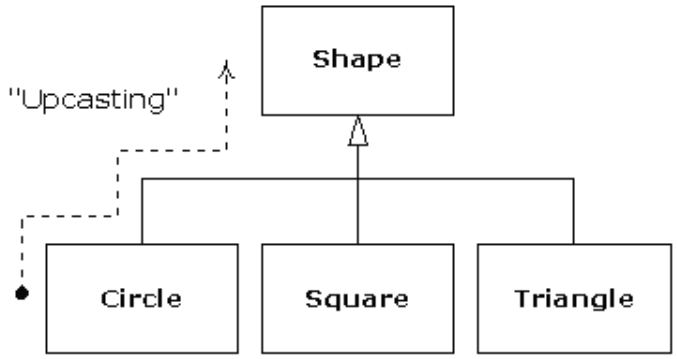
Las llamadas para **hacerAlgo()** trabajan automáticamente de manera correcta, sin importar el tipo exacto del objeto.

Este es un truco asombroso. Considere la línea:

```
hacerAlgo(c);
```

Lo que sucede aquí es que un **Círculo** se está pasando a un método que espera una **Figura**. Puesto que un **Círculo** *es* una **Figura** puede ser tratada como si fuera una por **hacerAlgo()**. Es decir, cualquier mensaje que **hacerAlgo()** pueda enviar a una **Figura**, un **Círculo** puede aceptarla. Es una cosa totalmente segura y lógica de hacer.

Llamamos este proceso de tratar un tipo derivado como si fuera su tipo base *elevación de molde*. El nombre de *molde* se utiliza en el sentido de amoldar en un molde y *elevación* viene de la manera en que el diagrama de la herencia está realizado típicamente, con el tipo base en la parte superior y las clases derivadas abajo desplegadas. Así, modelar un tipo base es elevar en el diagrama de herencia: "elevación de molde".



Un programa orientado al objeto contiene alguna elevación de moldes en alguna parte, porque esto es cómo usted se des-une de conocer sobre el tipo exacto con el que trabaja. Mire el código en **hacerAlgo()** :

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

Note que no dice "Si es un **Círculo**, haga esto, si es un **Cuadrado**, esto otro, etc." Si usted escribe aquél de manera que el código compruebe por todos los tipos posibles de **Figura** que puede actualmente ser, será sucio y necesitará cambiar cada vez que agrega una nueva clase de **Figura**. Aquí, usted dice: "Si es una Figura, sé que puede **borrar()** y **dibujar()** a sí misma, hacer esto, y tomar con cuidado los detalles correctamente".

Lo que es impresionante del código en **hacerAlgo()** es que, de alguna manera, suceden las cosas rectamente. Llamar a **dibujar()** para el **Círculo** hace código diferente para ser ejecutada que al llamar el **dibujar()** para un **Cuadrado** o una **Línea**, pero cuando el mensaje de **dibujar()** es enviado a una **Figura** anónima, el comportamiento correcto ocurre basado en el tipo real de la **Figura**. Esto es asombroso porque, según lo mencionado anteriormente, cuando el compilador de Java está compilando el código para **hacerAlgo()**, no puede saber exactamente de qué tipos se está ocupando. De ordinario, usted espera que termine llamando la versión de **borrar()** y **dibujar()** para la clase base de **Figura**, y no específica para **Círculo**, **Cuadrado**, o **Línea**. Pero la cosa sucede bien debido al polimorfismo. El compilador y el sistema de tiempo de ejecución manejan los detalles; todo lo que necesita saber ahora es que sucede, y más importantemente, cómo diseñar con él. Cuando usted envía un mensaje a un objeto, el objeto hará las cosas correctamente, incluso cuando la elevación de molde esté implicada.

Las clases bases abstractas y las interfaces

A menudo en un diseño, usted quisiera que la clase base presentara *solamente* una interfaz para sus clases derivadas. Es decir, no quiere que cualquier persona cree realmente un objeto de la clase base, sólo para elevar el molde para poder utilizar su interfaz. Esta es logrado haciendo esa clase *abstracta* usando la palabra clave **abstract**. Si cualquier persona intenta hacer un objeto de una clase **abstract**, el compilador la prevendrá. Esto es una herramienta para hacer cumplir un diseño particular.

Usted puede también utilizar la palabra clave **abstract** para describir un método que el aún no ha sido implementado - como una porción que indica "aquí está un método interfaz para todos los tipos heredados de esta clase, pero en este punto no se ha implementado nada con él". Un método **abstract** se puede crear solamente dentro de una clase **abstract**. Cuando se hereda la clase, ese método debe ser implementado, o la clase heredada viene a ser **abstract** también. Crear un método **abstract** permite que usted ponga un método en una interfaz sin estar forzado a proporcionar un posible cuerpo de código sin sentido para ese método.

La palabra clave **interface** toma el concepto de una clase **abstract** de la un poco más lejos previniendo cualquier definición de método alguno. El **interface** es una herramienta muy práctica y comúnmente usada, pues proporciona la separación perfecta de la interfaz y de la implementación. Además, usted puede combinar muchas interfaces juntas, si usted desea, mientras que la herencia de múltiples clases regulares o de clases abstractas no es posible.

Creación de objetos, utilización y tiempos de vida

Técnicamente, la OOP está justo sobre la tipificación de datos abstractos, herencia, y polimorfismo, pero otras visiones pueden ser al menos importantes. Esta sección cubrirá estas visiones.

Uno de los factores más importantes de los objetos es la manera en que se crean y se destruyen. ¿Dónde están los datos para un objeto y cuál es el tiempo de vida del objeto se controla? Hay diferentes filosofías de trabajo aquí. C++ toma el acercamiento de que el control de la eficacia es la visión más importante, así que le da al programador una opción. Para la máxima velocidad en tiempo de ejecución, el almacenamiento y el tiempo de vida se pueden determinar mientras que se está escribiendo el programa, poniendo los objetos en la pila (éstas se llaman a veces variables *automáticos* o *scoped* variables) o en el área de almacenamiento estático. Esto pone una prioridad en la velocidad de la asignación y liberación de almacenamiento, y el control de éstos puede tener mucho valor en algunas situaciones. Sin embargo, usted sacrifica flexibilidad porque debe saber la cantidad, el tiempo de vida, y el tipo exacto de objeto mientras que está escribiendo el programa. Si usted está intentando solucionar un problema más general tal como diseño automatizado, administración de depósito, o control del tráfico aéreo, éste es demasiado restrictivo.

La segundo aproximación es crear los objetos dinámicamente en un conjunto de memoria llamado el montón. En esta aproximación, usted no sabe hasta tiempo de ejecución cuántos objetos necesitará, cuál es su tiempo de vida, o cuál es su tipo exacto. Éstos son determinados en el momento que son incentivados mientras el programa está corriendo. Si necesita un nuevo objeto, simplemente lo crea en el montón, en el punto que lo necesite. Porque el almacenamiento se administra dinámicamente, en tiempo de ejecución, la cantidad de tiempo requerida para asignar el almacenamiento en el montón puede ser perceptiblemente más extenso que cuando crea el almacenamiento en pila. (Crear el almacenamiento en la pila es a menudo una sola instrucción de ensamblador para mover el puntero de pila hacia abajo y otro para moverlo de regreso. El momento para crear el almacenamiento de montón depende del diseño del mecanismo del almacenamiento). La aproximación dinámica hace la presunción generalmente lógica que los objetos tienden a ser complicados, así que los gastos indirectos adicionales de encontrar el almacenamiento y de liberar aquél no tendrá un impacto importante en la creación de un objeto. Además, la mayor flexibilidad es esencial de solucionar para el problema general de programación.

Java utiliza la segunda aproximación, exclusivamente. **[6]** Cada vez que usted desea crear un objeto, utiliza la palabra clave **new** para construir un instancia dinámica de ese objeto.

[6] Los tipos primitivos, sobre los cuales aprenderá más tarde, son un caso especial.

Hay otra cuestión, sin embargo, y ése es el tiempo de vida de un objeto. Con los lenguajes que permiten que los objetos sean creados en la pila, el

compilador determina cuánto tiempo dura el objeto y puede destruirlo automáticamente. Sin embargo, si usted lo crea en el montón el compilador no tiene conocimiento de su tiempo de vida. En una lenguaje como C++, usted debe determinar en tiempo de programación cuando destruir el objeto, lo cual puede inducir a pérdidas de memoria si no lo hace correctamente (y éste es un problema común en programas de C++). Java proporciona una característica llamada *recolector de basura (garbage collector)* que descubre automáticamente cuando un objeto no es más usado por largo tiempo y lo destruye. Un recolector de basura es mucho más conveniente porque reduce el número de los caminos que debe seguir y del código que debe escribir. Más importante, el recolector de basura proporciona un nivel mucho más alto de seguridad contra el problema insidioso de las fugas de memoria (el cual ha traído a muchos un proyecto de C++ a sus rodillas).

Colecciones e iteradores

Si usted no conoce cuántos objetos va a necesitar para solucionar un problema particular, o cuánto tiempo durarán, tampoco conoce cómo almacenar esos objetos. ¿Cómo puede usted saber cuánto espacio necesita para crear esos objetos? Usted no puede, entonces esa información no es conocida hasta tiempo de ejecución.

La solución a la mayoría de los problemas en diseño orientado al objeto parece ligero: Usted crea otro tipo de objeto. El nuevo tipo de objeto que soluciona este problema particular mantiene referencias a otros objetos. De hecho, usted puede hacer la misma cosa con un arreglo (array), que está disponible en la mayoría de los lenguajes. Pero este nuevo objeto, generalmente llamado un *contenedor (container)* (también llamado una *colección (collection)*, pero las aplicaciones de la biblioteca de Java las llaman con diferentes sentidos así que este libro utilizará “contenedor (container)”), se ampliará siempre que sea necesario para acomodar todo lo que coloque dentro de él. Así no necesita conocer cuántos objetos va a mantener un contenedor. Apenas crea un objeto contenedor y deja que el tome cuidado de los detalles.

Afortunadamente, un buen lenguaje de POO viene con un sistema de contenedores como parte del paquete. En C++, esta es parte de la biblioteca estándar de C++ y a veces se llaman Biblioteca Estándar de Plantillas (Standard Template Library-STL). Object PASCAL tiene contenedores en su Biblioteca de Componentes Visuales (Standard Template Library-VCL). Smalltalk tiene un sistema muy completo de contenedores. Java también tiene contenedores en su biblioteca estándar. En algunas bibliotecas, un contenedor genérico se considera suficientemente bueno para todas las necesidades, y en otros (Java, por ejemplo) la biblioteca tiene diversos tipos de contenedores para distintas necesidades: diferentes formas de clases **List** (llevan a cabo secuencias), clases **Map** (también conocido como *arreglos de asociación*, para asociar objetos a otros objetos), y el clases **Set** (para mantener uno de cada tipo de objeto). Las bibliotecas de contenedores pueden también incluir colas (qeues), árboles (trees), pilas (stacks), etc.

Todos los contenedores tienen alguna manera de poner y sacar cosas; hay métodos usualmente para agregar elementos a un contenedor, y otros para extraer estos elementos de vuelta. Aunque traer elementos puede ser más problemático, porque un método de selección simple es restrictivo. ¿Pero qué si desea comparar o manipular un grupo de elementos del contenedor en vez de únicamente uno?

La solución es un *iterador (iterator)*, el cual es un objeto cuyo trabajo es seleccionar los elementos dentro de un contenedor y presentarlos al usuario del iterador. Como una clase, también provee un nivel de abstracción. Esta abstracción puede ser utilizada para separar los detalles del contenedor del código al que está accediendo ese contenedor. El contenedor, vía el iterador, es abstraído a una secuencia simple. El iterador le permite atravesar esta secuencia sin preocuparlo de la estructura implícita - esto es, ya sea una **lista de arreglos (ArrayList)**, una **lista enlazada (LinkedList)**, una **Pila (Stack)**, o algo similar. Esto le permite flexibilidad para cambiar fácilmente la estructura de datos esencial sin distorsionar el código de su programa. Java comenzó (en la versión 1.0 y 1.1) con un iterador estándar, llamado **Enumeration**, para todas sus clases contenedor. Java 2 agrega una librería contenedor mucho más completa que contiene un iterador llamado **Iterator** que hace mucho más que el viejo **Enumeration**.

Desde el punto de vista del diseño, todo lo que realmente desea un una secuencia que puede ser manipulada para resolver su problema. Si un tipo único de secuencia satisface todas sus necesidades, no hay razón para tener diferentes maneras. Hay dos razones que necesitan una elección de contenedores. Primero, los contenedores proveen diferentes tipos de interfaces y comportamientos externos. Una pila tiene una interfaz y comportamientos diferentes de una cola, la cual es diferente de un grupo o una lista. Una de estas debe proveer mayor flexibilidad a la solución de su problema que algún otro. Segundo, diferentes contenedores tienen diferentes rendimientos para ciertas operaciones. El mejor ejemplo es comparar dos tipos de **List**: un **ArrayList** y un **LinkedList**. Ambos son simples secuencias que pueden tener idénticas interfaces y comportamientos. Pero ciertas operaciones pueden tener costos radicalmente diferentes. El acceso azaroso (randomly) de elementos en un **ArrayList** es una operación de tiempo constante; toma la misma cantidad de tiempo a pesar del elemento seleccionado. Sin embargo, en un **ArrayList** es caro moverse a través de la lista para seleccionar un elemento, y toma muchísimo tiempo encontrar un elemento que está en el último lugar de la lista. De otra manera, si desea insertar un elemento en el medio de una secuencia, es más barato en una **LinkedList** que en un **ArrayList**. Estas y otras operaciones tienen diferentes rendimientos dependiendo de la estructura esencial de la secuencia. En la fase de diseño, debe empezar con un **LinkedList** y, cuando afine el funcionamiento, cambiar a un **ArrayList**. Porque vía la abstracción de la clase base **List** y los iteradores, puede cambiar de uno hacia otro con mínimo impacto en su código.

La jerarquía de raíz única

Una de las razones en que la POO viene especialmente sobresaliendo desde la introducción del C++ es donde todas las clases deben finalmente ser herederas de una única clase base. En Java (como virtualmente todos los otros lenguajes POO, a excepción de C++) la respuesta es sí, y el nombre de esta clase base primaria es simplemente **Object**. De ello resulta que los beneficios de la jerarquía de raíz única sean muchos.

Todos los objetos en una jerarquía de raíz única tienen una interfaz en común, así que ellas son finalmente del mismo tipo fundamental. La alternativa (provista por C++) es que no conozca que todo sea del mismo tipo básico. Desde un punto de vista de compatibilidad hacia atrás esto llena el modelo de C mejor y puede ser pensado de una manera menos restrictiva, pero cuando desea hacer algo en programación orientada completamente al objeto debe entonces construir su propia jerarquía para proveer la misma conveniencia que construir dentro de otros lenguajes POO. Y en cualquier nueva librería de clases que adquiera, alguna otra interfaz incompatible será usada. Ello requiere esfuerzo (y posiblemente herencia múltiple) para trabajar la interfaz nueva dentro de su diseño. ¿Es la "flexibilidad" de C++ lo buscado? Lo es si lo necesita - si tiene una gran inversión en C - es suficientemente valorable. Si está empezando desde cero, otras alternativas tales como Java pueden ofrecerle mayor productividad.

Todos los objetos en una jerarquía de raíz única (tal como Java provee) pueden garantizarle tener cierta funcionalidad. Conocerá que puede llevar a cabo operaciones básicas seguras sobre todos los objetos de su sistema. Una jerarquía de raíz única, a lo largo del tiempo mientras crea todos los objetos en el montón, simplifica mayormente el pasaje de argumentos (uno de los tópicos más complejos en C++).

Una jerarquía de raíz única hace mucho más fácil implementar un recolector de basura (el cual está apropiadamente construido dentro de Java). El soporte necesario puede ser instalado en la clase base, y el recolector de basura puede enviarles los mensajes apropiados a cada objeto en el sistema. Sin una jerarquía de raíz única y un sistema para manipular un objeto mediante referencia, es difícil implementar un recolector de basura.

Desde tiempo de ejecución el tipo de información está garantizado será en todos los casos objetos, nunca acabará en un objeto cuyo tipo no pueda determinar. Esto es especialmente importante con operaciones a nivel de sistema, tales como manejo de excepciones, y permitir gran flexibilidad en la programación.

"Modelado hacia abajo" (Downcasting) contra plantillas/genéricas

Para hacer estos contenedores reutilizables, ellos mantienen un tipo universal en Java:**Object**. La jerarquía de raíz única significa que todo es un **Objeto (Object)**, así un contenedor que se sostiene en **Object**s puede mantener todo **[7]** Esto hace a los contenedores fáciles de reutilizar.

[7] Excepto, desafortunadamente, para las primitivas. Esto es discutido en detalle más adelante en este libro.

Para usar un contenedor, simplemente agréguele referencias a objetos y a lo último pregunte por ellos de nuevo. Pero, ya que el contenedor conserva solamente **Object**s, cuando agrega su referencia a objeto en el contenedor esta "modelando hacia arriba" hacia **Object**, esto es pierde su identidad. Cuando lo trae de vuelta, obtiene una referencia a **Object**, y no una referencia al tipo que puso en él. ¿Así cuando regresa dentro de aquella interfaz utilizada para el objeto que puso en el contenedor?

Aquí el modelado es usado nuevamente, pero esta vez no es hacia arriba la heredad de la jerarquía hacia un tipo más general. En contrario, usted modela hacia abajo de la jerarquía para un tipo más específico. Esta interpretación del modelado es llamada *modelado hacia abajo (downcasting)*. Con el modelado hacia arriba conoce, por ejemplo, que un **Círculo** es un tipo de **Figura**, por lo cual está seguro del modelado hacia arriba, pero no conoce necesariamente que un **Object** es un **Círculo** o una **Figura** por tanto no está tan seguro del modelado hacia abajo a menos que conozca exactamente sobre el tratamiento que hace.

No es, sin embargo, completamente peligroso, porque si modela hacia abajo y ocurre un error obtendrá un error en tiempo de ejecución llamado *excepción (Exception)*, el cual será descrito en breve. Cuando extrae las referencias a objeto desde el contenedor, asimismo, debe tener alguna manera de recordar exactamente qué son ellos así puede realizar un modelado hacia abajo apropiadamente.

El modelado hacia abajo y el chequeo en tiempo de ejecución requieren tiempo extra para correr el programa y sobreesfuerzo para el programador. ¿No tendría sentido crear el contenedor así conoce los tipos que mantiene, eliminando la necesidad para el modelado hacia abajo y un posible error? La solución es llamada mecanismo de *tipo parametrizado (parameterized type)*. Un tipo parametrizado es una clase que el compilador puede personalizar automáticamente para trabajar con tipos particulares. Por ejemplo, con un contenedor parametrizado, el compilador debiera modelar que el contenedor deba aceptar únicamente **Figura** y extraer solamente **Figura**s.

Los tipos parametrizados son una parte importante de C++, particularmente porque C++ 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 ya que es posible para ello obtener por - no obstante la incomodidad - medio de la jerarquía de raíz única. Sin embargo, una propuesta reciente para los tipos parametrizados utiliza una sintaxis que está llamativamente parecida a los templates de C++, y podemos esperar ver tipos parametrizados (que serán llamados *generics* en la siguiente versión de Java).

Asegurar la limpieza justa

Cada objeto requiere recursos, memoria mayormente, en cuanto a su existencia. Cuando un objeto no es utilizado por un largo período debe ser limpiado así esos recursos están disponibles para su rehúso. En situaciones de programaciones sencillas la cuestión de cuándo un objeto está para limpiarse no se ve tan difícil: crea un objeto, lo usa mientras sea necesario, y luego lo destruye. Sin embargo, no es difícil encontrar situaciones más complejas.

Suponga, por ejemplo, que está diseñando un sistema para administrar el tráfico aéreo de un aeropuerto. (El mismo modelo debe también trabajar para manejar cajas en un depósito, o un sistema de videos rentados, o un canil en una guardería de mascotas). Al principio se verá simple: hace un contenedor para sostener los aviones, entonces crea un nuevo avión y lo ubica en el contenedor para cada avión que ingresa a la zona de control de tráfico aéreo. Para limpiar, simplemente borra el objeto del avión apropiado cuando uno deja la zona.

Pero puede que tenga otro sistema para registrar datos sobre los aviones; datos que quizás no necesiten atención inmediata como la función de control principal. Podría ser un registro de un plan de vuelo de un avión pequeño, y si crea un objeto plan también lo pone en este contenedor segundo si es un plan pequeño. Entonces algunos procesos de segundo plano realizan operaciones sobre estos objetos en este contenedor durante momentos inactivos.

Ahora el problema es más difícil: ¿Cómo puede posibilitar conocer cuando destruir los objetos? Cuando está con el objeto, pueden otras partes del sistema no estarlo. Este mismo problema puede aparecer en un número de otras situaciones, y en sistemas de programación (tales como en C++) en los cuales debe explicitar el borrado de un objeto cuando esté completado el uso de él lo cual es un tanto complejo.

Con Java, el recolector de basura está diseñado para tener cuidado del problema de liberación de memoria (aunque esto no incluye otros aspectos de limpieza de un objeto). El recolector de basura conoce cuando un objeto no está mucho tiempo en uso, y entonces libera automáticamente la memoria de aquél objeto. Esto (combinado con el hecho de que todos los objetos son heredados de la clase de raíz única **Object** y que puede crear objetos solo de una manera - en el montón) hace que el proceso de programación en Java mucho más sencillo que la programación en C++. Usted tiene pocas decisiones para hacer y saltear los obstáculos.

Recolector de basura contra eficiencia y flexibilidad

Si todo esto es una buena idea, ¿porque no hacen la misma cosa en C++? Bien, porque de hecho hay un precio a pagar por toda la conveniente programación, y ese es la sobrecarga en tiempo de ejecución. Como mencionamos antes, en C++ puede crear objetos en la pila, y en ese caso serán automáticamente limpiados (pero no tendrá la flexibilidad de crear cuantos desee en tiempo de ejecución). Crear objetos en la pila es la manera más eficiente de localizar el almacenamiento de los objetos y luego liberarlo. Crear objetos en el montón puede ser más caro en verdad. Siempre heredar

de una clase base y hacer todas las llamadas a métodos polimórficos conllevan una pequeña tasa. Pero el recolector de basura es un tema singular pro que nunca conocerá cuando está iniciando o cuánto tiempo tomará. Esto significa que hay una inconsistencia en el transcurso de ejecución de un programa Java, de tal manera que no puede usarlo en determinadas situaciones, tales como cuando la velocidad de ejecución de un programa es uniformemente crítica. (Estos son llamados generalmente programas de tiempo real, aunque no todos los problemas de programación en tiempo real son estrictos).

Los diseñadores del lenguaje C++, intentaron agradar a los programadores de C (y en muchos casos lo lograron), no deseando agregar características al lenguaje que fueran a impactar la velocidad o el uso de C++ en cualquier situación donde los programadores puedan elegir C alternativamente. Esta meta fue conseguida, pero al precio de una alta complejidad cuando se programa en C++. Java es mucho más simple que C++, pero la depreciación está en la eficiencia y cierta aplicabilidad. Para una porción significativa de problemas de programación, sin embargo, Java es una elección superior.

Manejo de excepciones: tratando con errores

Ya desde los primeros lenguajes de programación, el manejo de errores ha sido una de las cuestiones más difíciles. Porque es sumamente difícil diseñar un buen esquema de manejo de errores, algunos lenguajes simplemente ignoran la cuestión, dejando el problema a los diseñadores de la biblioteca quienes las toman a mitad de camino por lo cual las medidas trabajaran en algunas situaciones y en otras será fácilmente evitadas, generalmente porque las ignoran. Un problema mayor con la mayoría de los esquemas de manejo de error es que ellos confían en la vigilancia del programador en seguir una convención sobre acordada que no está forzada por el lenguaje. Si el programador no desea controlar - como es el caso si ellos están apresurados - estos esquemas pueden ser fácilmente olvidados.

El manejo de excepción ata el manejo de error directamente dentro del lenguaje de programación y algunas veces aún al sistema operativo. Una excepción es un objeto que es "lanzado" desde el sitio del error y puede ser "tomado" por un apropiado administrador de excepciones diseñado para manejar ese tipo singular de error. Es como si el manejo de excepción es diferente, camino paralelo de ejecución que puede ser tomado cuando las cosas van mal. Y como usa un camino de ejecución separado, no necesita interferir con la ejecución normal de código. Esto logra que el código se simplifique al escribir porque no está forzado constantemente a chequear por errores. Agregado a ello, una excepción lanzada no se parece a un valor de error que es devuelto por un método o una bandera que es fijada por un método en orden a indicar una condición de error - estos pueden ser ignorados. Una excepción no puede ser ignorada, así que está garantizada a ser tratada en algún punto. Finalmente, las excepciones proveen una manera confiable de recuperarse de una mala situación. En vez de solo salir del

programa, está habilitado a fijar las cosas correctamente y restaurar la ejecución, lo cual produce programas mucho más robustos.

El manejo de excepción de Java sobresale a lo largo de los lenguajes de programación, porque en Java, el manejo de excepción está atado desde el principio y usted se ve forzado a usarlo. Si no escribe el código apropiadamente para manejar las excepciones, obtendrá un mensaje de error en tiempo de compilación. Esto consistencia garantizada puede hacer el manejo de errores mucho más simple algunas veces.

Es de valor notar que el manejo de excepción no es una característica orientada a objeto, sino que en los lenguajes orientados a objetos la excepción es normalmente representada por un objeto. El manejo de excepción existe desde antes de los lenguajes orientados a objetos.

Concurrencia

Un concepto fundamental en la programación de computadora es la idea de manejar más de una tarea al mismo tiempo. Algunos problemas de programación requieren que el programa esté habilitado a parar cuando lo hacen, tratarlos con algún otro problema, y regresar al proceso principal. La solución ha sido aproximada de muchas maneras. Inicialmente, los programadores con conocimiento de bajo nivel de la máquina escribieron rutinas de interrupción de servicio, y la suspensión del proceso principal fue iniciada a través de una interrupción hardware. Aunque esto trabaja muy bien, es dificultoso y no portable, ya que hacer un programa para mover a un nuevo tipo de máquina es lento y caro.

Algunas veces, las interrupciones son necesarias para el manejo de tareas en tiempo crítico, pero hay grandes clases de problemas en los cuales simplemente intenta partir el problema en piezas de ejecución separada así el corazón del programa puede ser mucho más reactivos. Dentro de un programa, estas piezas de ejecución separada son llamadas hilos, y el concepto general es denominado *concurrencia (concurrency)* o *multihilo (multithreading)*. Un ejemplo común de multihilo es la interfaz de usuario. Utilizando hilos, un usuario puede presionar un botón y obtener una respuesta rápida en vez que sea forzado a esperar hasta que el programa finalice su tarea actual.

Comúnmente, los hilos son una manera de localizar el tiempo de un procesador único. Pero si el sistema operativo soporta muchos procesadores, cada hilo puede ser asignado a un procesador diferente, y pueden verdaderamente correr en paralelo. Una de las características convenientes del multihilo en el nivel de lenguaje es que el programador no necesita preocuparse si hay muchos procesadores o solo uno. El programa está dividido lógicamente dentro de hilos y si la máquina tiene más de un procesador, entonces el programa correrá rápido sin ajustes especiales.

Todo esto hace un hermoso sonido hilado simple. Hay una trampa: los recursos compartidos. Si tiene más de un hilo corriendo que está esperando para acceder al mismo recurso, tiene un problema. Por ejemplo, dos procesadores no pueden enviar simultáneamente información a una impresora. Para resolver el problema, los recursos que pueden ser compartidos, tales como una impresora, deben ser bloqueados mientras están siendo usados. De tal manera los hilos bloquean un recurso, completan su tarea, y luego liberan el bloqueo ya que alguien más puede usar el recurso.

El hilado de Java está construido dentro del lenguaje, lo cual hace un supuesto complicado mucho más simple. El hilado está soportado a nivel de objeto, así que un hilo de ejecución está representado por un objeto. Java provee también de bloqueo de recursos limitados. Puede bloquear la memoria de algún objeto (lo cual es, después de todo, una manera de compartir recursos) para que solamente un hilo pueda usarlo a la vez. Esto es conseguido con la palabra clave **synchronized**. Otros tipos de recursos deben ser bloqueados explícitamente por el programador, típicamente creando un objeto para representar el bloqueo que todos lo hilos deben chequear antes de acceder a ese recurso.

Persistencia

Cuando crea un objeto, existe tanto tiempo como sea necesario, pero bajo ninguna circunstancia existe cuando el programa finaliza. Mientras esto tiene sentido al principio, hay situaciones en las cuales sería increíblemente útil si un objeto pueda existir y mantener su información aún mientras el programa no esté corriendo. Entonces la próxima vez que inicia el programa, el objeto estará ahí y tendrá la misma información que tuvo en el momento anterior cuando el programa hubo corrido. De hecho, puede obtener un efecto similar escribiendo la información a un archivo o una base de datos, pero en el espíritu de hacer todo un objeto, sería un tanto conveniente estar habilitado para declarar un objeto persistente y tener todos los detalles que toman cuidado para usted.

Java provee soporte para "persistencia liviana", lo cual significa que puede fácilmente almacenar objetos en disco y más tarde recuperarlos. La razón de "liviana" es que está forzado a hacer llamadas explícitas para almacenar y recuperar. La persistencia liviana puede ser implementada ya sea a través de la *serialización de objeto (object serialization)* (tratada en el Capítulo 12) y en *Java Data Objects* (JDO, tratada en *Thinking in Enterprise Java*

Java y la Internet

Si Java es, de hecho, aún otro lenguaje de programación de computadora, se preguntará por qué es tan importante y está siendo promocionado como un paso revolucionario en la programación de computadora. La respuesta no es inmediatamente obvia si proviene de la perspectiva de programación tradicional. Aunque Java es muy útil para resolver problemas de programación

solitarios tradicionales, es también importante porque resolverán problemas de programación en la Amplia Telaraña Mundial.

¿Qué es la Web?

La Web puede verse con un poco de misterio al principio, con todo lo que se habla de "surfear", "presencia" y "páginas de hogar". Es útil parar, volverse y ver qué es realmente esto, pero para hacerlo debe entender los sistemas cliente/servidor, otro aspecto de la computación que está llena de cuestiones confusas.

Computación Cliente/Servidor

La idea primera de un sistema cliente/servidor es que tiene un almacén central de información - algún tipo de dato, ofrecido en una base de datos - que desea distribuir a demanda para otro grupo de personas o máquinas. Una clave para el concepto cliente/servidor es que el almacenamiento de reposición está localizado centralmente así que puede ser cambiado y que estos cambios se propaguen hacia los consumidores de información. Tomado todo junto, el almacén de información, el software que distribuye la información, y la máquina donde la información y el software residen se denomina el *servidor*. El software que reside en la máquina remota, se comunica con el servidor, pide la información, la procesa, y la muestra en la máquina remota es llamada el *cliente*.

El concepto básico de computación cliente/servidor, entonces, no es tan complicado. Los problemas surgen porque usted tiene un único servidor intentando servir a muchos clientes a la vez. Generalmente, un sistema administrador de base de datos está involucrado, así el diseñador "balancea" la capa de datos dentro de tablas para uso óptimo. Además, los sistemas pueden permitir a un cliente insertar información nueva dentro del servidor. Esto significa que debe asegurar que los datos nuevos de un cliente no pase por encima de los datos nuevos de otro cliente, o que los datos no se pierdan en el proceso de agregarlo a la base de datos (esto se denomina proceso de transacción). Como el software cliente cambia, debe ser construido, depurado e instalado en las máquinas clientes, lo cual resultará ser más complicado y caro de lo que pueda pensarse. Esto es especialmente problemático para soportar múltiples tipos de computadoras y sistemas operativos. Finalmente, hay cuestiones importantísimas de rendimiento: debe tener cientos de clientes haciendo peticiones a su servidor al mismo tiempo, así que un pequeño tiempo de espera es crucial. Para minimizar la latencia, los programadores trabajan duro para descargar las tareas de procesamiento, a veces a la máquina del cliente, pero otras a otras máquinas en el sitio del servidor, utilizando lo que se llama *middleware* (El Middleware también se usa para incrementar el mantenimiento).

La idea simple de distribuir información tiene muchas capas de complejidad que el problema central puede verse desesperadamente enigmático. Y aún es crucial: la computación cliente/servidor cuenta para endurecer la mitad de todas las actividades de programación. El es responsable para todo desde

tomar órdenes y transacciones con tarjetas de crédito hasta la distribución de cualquier tipo de datos - stock de mercado, científicos, gubernamentales, nómbrelo usted. Lo que venimos teniendo desde el pasado son soluciones individuales para problemas individuales, inventar una nueva solución cada vez. Estos son difíciles de crear y de usar, y el usuario tiene que aprender una nueva interfaz cada vez. El problema cliente/servidor necesita resolverse en una gran manera.

La Web es un servidor gigante

La Web es actualmente un sistema cliente/servidor gigante. Es un tanto desagradable entonces que, ya que tiene todos los servidores y clientes coexistiendo en una única red a la vez. Usted no necesita conocer esto, porque todo su cuidado es sobre la conexión y la interacción con un servidor a la vez (aún a pesar que puede estar saltando alrededor del mundo en la búsqueda por el servidor correcto).

Inicialmente fue un proceso simple de una sola vía. Usted hace una petición a un servidor y este le entrega un archivo, el cual su software navegador de su máquina (el cliente) debe interpretar para formatear dentro de su máquina local. Pero en corto tiempo la gente empezó a desear hacer más que solicitar páginas de un servidor. Ellos quieren capacidades completas de cliente/servidor para que el cliente pueda cargar información al servidor, por ejemplo, para hacer búsquedas en la base de datos del servidor, para agregar nueva información al servidor, o para realizar una orden (lo cual requiere más seguridad que la ofrecida en los sistemas originales). Estos son cambios que hemos estado viendo en el desarrollo de la web.

El navegador Web fue un gran paso al frente: el concepto que una pieza de información puede ser exhibida en cualquier tipo de computadora sin cambio. Sin embargo, los navegadores son algo bastante primitivos y rápidamente se empantanaron por las demandas ubicadas en ellos. Ellos no son interactivos justamente, y tienden a colgarse ambos, servidor e Internet, porque cualquier momento necesita para hacer algo que requiere programación tiene que enviar información nuevamente al servidor para ser procesados. Puede tomar algunos segundos o minutos resolver si tiene algo errado en su petición. Ya que el navegador era simplemente un visualizador no podía realizar aún simples tareas de computación. (De otra manera, es seguro, porque no puede ejecutar programas en su máquina local que pueda contener errores o virus).

Para resolver este problema, diferentes aproximaciones han sido tomadas. Para empezar, los gráficos estándar han sido mejorados para permitir mejores animaciones y videos dentro de los navegadores. Los restantes problemas pueden ser solucionados solamente incorporando la habilidad de correr programas en el cliente final, bajo el navegador. Esto es la llamada programación del lado cliente.

Programación del lado cliente

En la Web inicial de diseño servidor-navegador proveía de contenido interactivo, pero la interactividad era completamente provista por el servidor. El servidor producía páginas estáticas para el navegador cliente, quien simplemente interpretaría y las mostraría. El *Lenguaje de Marcados para Hipertextos (HyperText Markup Language* - HTML) contiene mecanismos simples para reunir los datos: cajas para ingresar textos, cajas de chequeo, cajas de radio, listas y listas desplegables, así como un botón que solamente puede ser programado para resetear los datos en el formulario o "enviar" esos datos en el formulario hacia el servidor. Esta presentación pasa a través de *Interfaz de Puente Común (Common Gateway Interface* - CGI) provista en todos los servidores Web. El texto dentro del envío indica a CGI que hacer con ellos. La acción más común es correr un programa localizado en el servidor en un directorio típicamente denominado "cgi-bin". (Si observa la barra de dirección en la parte superior de su navegador cuando pulsa un botón de una página Web, puede ver a veces el "cgi-bin" dentro de toda esa jeringaza). Estos programas pueden ser escritos en muchos lenguajes. Perl ha sido una elección común porque está diseñado para manipulación de texto e interpretación, así puede ser instalado en cualquier servidor a pesar del procesador o sistema operativo. Sin embargo, Python (mi favorito - ver www.Python.org) ha estado haciendo avances porque es más poderoso y simple.

Muchos sitios Web poderosos de hoy están construidos estrictamente en CGI, y pueden de hecho hacer casi cualquier cosa con CGI. Sin embargo, los sitios Web construidos sobre programas CGI pueden rápidamente venir a convertirse en complicados para mantener, y hay aún un problema de tiempo de respuesta. La respuesta de un programa CGI depende sobre cuántos datos deben ser enviados, así como también la carga en ambos, servidor e Internet. (Por encima de todo esto, iniciar un programa CGI tiende a ser lento). Los primeros diseñadores de la Web no predijeron cuán rápidamente este ancho de banda sería saturado para las clases de aplicaciones que la gente desarrolló. Por ejemplo, cualquier ordenamiento de gráficos dinámicos es casi imposible de realizar con consistencia porque un archivo *Formato para Intercambiar Gráficos (Graphics Interchange Format* - GIF) debe ser creado y movido desde el servidor hacia el cliente por cada versión del gráfico. Y no tenga dudas que ha dirigido la experiencia con algo como una simple validación de datos en el formulario de entrada. Presiona el botón de envío en una página; los datos son embarcados al servidor; el servidor inicia un programa CGI el cual descubre un error, formatea una página HTML informando del error, y envía la página de regreso a usted; quien debe entonces regresar a la página e intentarlo de nuevo. No solamente es lento, sino falto de elegancia.

La solución es la programación del lado cliente. La mayoría de las máquinas que corren navegadores Web son poderosas máquinas capaces de hacer amplios trabajos, y con la aproximación original de HTML estático ellas están sentadas ahí, desocupadas esperando por el servidor entregue la siguiente página. La programación del lado cliente significa que el navegador Web está aprovechando para hacer todo el trabajo que puede, y el resultado es una experiencia mucho más rápida e interactiva en su sitio Web.

El problema con las discusiones de la programación del lado cliente es que no son muy diferentes de las discusiones de programación en general. Los parámetros son mayormente los mismos, pero la plataforma es diferente; un navegador Web es como un sistema operativo limitado. En el final, debe aún programar, y esto cuenta para un arsenal embotador de problemas y soluciones producidas por la programación del lado cliente. El resto de esta sección provee una visión de situaciones y aproximación en la programación del lado cliente.

Plug-ins

Uno de los pasos más significativos en el avance de la programación de lado cliente es el desarrollo del plug-in. Esta es una manera en que un programador agrega nueva funcionalidad al navegador para bajar una porción de código que engarza dentro del lugar apropiado en el navegador. El le dirá al navegador "desde ahora puede hacer esta nueva actividad". (Necesita bajar el plug-in una sola vez). Algunos comportamientos rápidos y poderosos son agregados a los navegadores vía los plug-ins, pero escribir uno no es una tarea trivial, no es algo que desee para hacer como parte del proceso de construir un sitio particular. El valor del plug-in para la programación del lado cliente es que permita a un programador experto desarrollar un nuevo lenguaje y agregarlo al navegador sin los permisos del fabricante del navegador. Esto es, los plug-ins proveen una "puerta trasera" que permite la creación de nuevos lenguajes de programación del lado cliente (aunque no todos los lenguajes son implementados como plug-ins).

Lenguajes Scripting

Los plug-ins resultaron en una explosión de lenguajes scripting. Con un lenguaje scripting, usted embebe el código fuente para el programa del lado cliente directamente dentro de su página Web, y el plug-in interpreta el lenguaje que es automáticamente activado mientras una página HTML está siendo mostrada. Los lenguajes scripting tienden a ser razonablemente fáciles para entender y, porque son textos simplemente que están en una página HTML, se cargan rápidamente como parte del único pedido al servidor requerido para procurar la página. La ganancia conseguida es que su código está expuesto para que la vean todos (y la hurten). Generalmente, por otro lado, no hará cosas sofisticadas sorprendentes con los lenguajes scripting, por lo cual no una pérdida tan grande.

Esto apunta a que los lenguajes scripting sean usados dentro de los navegadores Web para intentar resolver tipos específicos de problemas, primariamente la creación de interfaces gráficas de usuario (GUIs) más interactivas y ricas. Sin embargo, un lenguaje scripting debe resolver el 80 por ciento de los problemas encontrados en la programación del lado cliente. Su problema puede muy bien estar completamente dentro de ese 80 por ciento, y ya que los lenguajes de scripting pueden desarrollar rápido y fácilmente, debiera probablemente considerar un lenguaje scripting antes de buscar una solución más avanzada tal como programación Java o ActiveX.

La discusión más común de los lenguajes scripting para navegadores son JavaScript (el cual no tiene nada con Java; es llamado con ese nombre para aprovechar el momento comercial de Java), VBScript (el cual se parece a Visual BASIC) y Tcl/Tk, que proviene del popular lenguaje de construcción de GUI para plataformas cruzadas. Hay otros aparte de estos, y no dude que haya más en desarrollo.

JavaScript es probablemente el más soportado comúnmente. Fue construido para sendos Netscape Navigator y Microsoft Internet Explorer (IE). Desafortunadamente, el condimentado de JavaScript en los dos navegadores puede variar ampliamente (el navegador Mozilla, liberado para bajar desde www.Mozilla.org, soporta el estándar ECMAScript, que puede un día llegar a ser soportado universalmente). Además, hay probablemente más libros de JavaScript disponibles que los que para otros lenguajes de navegadores, y algunas herramientas automáticamente crean páginas usando JavaScript. Sin embargo, si se siente cómodo hoy en Visual BASIC o Tcl/Tk, será más productivo utilizando estos lenguajes scripting en vez de aprender uno nuevo. (Tendrá sus manos llenas para tratar con los usos de la Web de hoy).

Java

Si un lenguaje scripting puede resolver el 80 por ciento de los problemas del lado cliente, ¿Qué sucede con el otro 20 por ciento - "La sección realmente difícil"? Java es la solución popular para esto. No es solamente un lenguaje de programación poderoso construido para ser seguro, plataforma cruzadas, e internacionales, sino que Java está siendo continuamente extendido para proveer características de lenguaje y librerías que elegantemente manejan problemas que son difíciles en los lenguajes de programación tradicional, tales como multihilos, acceso a base de datos, programación de red, y computación distribuida. Java permite la programación del lado cliente vía el *applet* y con *Java Web Start*.

Un applet es un programa mínimo que solamente correrá bajo un navegador Web. El applet es bajado automáticamente como parte de una página Web (como, por ejemplo, un gráfico es bajado). Cuando el applet es activado, ejecuta un programa. Esta es la parte de su maravilla - provee una manera automática de distribuir el software cliente desde el servidor en el momento que el usuario lo necesita al software cliente, y no antes. El usuario obtiene la última versión del software cliente sin fallas ni reinstalaciones complicadas. Por la manera en que Java está diseñado, el programador necesita crear solamente un programa, y este trabaja automáticamente con todas las computadoras que tienen navegadores con intérpretes Java. (Esto seguramente incluye a una amplia mayoría de máquinas). Ya que Java es un lenguaje de programación de vuelo completo, puede hacer tanto trabajo como sea posible en el cliente antes y después de hacer peticiones del servidor. Por ejemplo, no necesita enviar un formulario de petición a través de la Internet para descubrir que tiene un dato o algún parámetro erróneo, y su computadora cliente puede sencillamente hacer el trabajo de armado de los datos en vez de esperar por que el servidor haga un entramado e ingrese la imagen grafica pre-hecha. No solamente obtiene una ganancia de velocidad y

respuesta, sino que el tráfico de la red en general y la carga de los servidores puede ser reducida, previniendo a la Internet entera de un bajón de velocidad.

Una de las ventajas de un applet Java por sobre un programa script es que está en forma compilada, lo cual es el código fuente no está disponible para el cliente. De otra manera, un applet Java puede ser decompilado sin muchos problemas, pero esconder su código no es una cuestión importante. Otros dos factores pueden ser importantes. Como verá más tarde en este libro, un applet Java compilado puede requerir tiempo extra para bajarse, si es muy grande. Un programa script estará integrado dentro de la página Web como parte de su texto (y generalmente es más pequeño y reduce los llamados al servidor). Esto puede ser importante para mejorar la respuesta de su sitio Web. Otro factor es la alta importancia de la curva de aprendizaje. Sin tener encuentra de que usted sea un cabeza dura, Java no es un lenguaje trivial para aprender. Si es usted un programador Visual BASIC, pasarse a VBScript será una solución más rápida (asumiendo que puede restringir a sus consumidores a plataformas Windows), y ya que probablemente resolverá los problemas cliente/servidor más típicos, puede ser duramente presionado a justificar el aprendizaje de Java. Si experimentó con lenguaje scripting ciertamente se beneficiará de observar JavaScript y VBScript antes que meterse con Java, porque ellos pueden completar sus necesidades a mano y ser más productivo en corto tiempo.

.NET and C#

Por un tiempo, el principal competidor de los applet Java fue ActiveX de Microsoft, a pesar de requerir que el cliente esté corriendo Windows. Desde entonces, Microsoft produjo un competidor completo para Java en la forma de la plataforma **.NET** y el lenguaje de programación **C#** . La plataforma **.NET** es aproximadamente la misma que la máquina virtual Java y las librerías Java, y **C#** lleva manifiestamente similaridades con Java. Este es seguramente el mejor trabajo que Microsoft ha hecho en la arena de los lenguajes de programación y los ambientes de programación. De hecho, tienen la considerable ventaja de estar avisados qué trabaja bien y qué no tanto en Java, y construir sobre ello, pero lo construido ellos lo tienen. Esta es la primera vez que desde la aparición de Java tienen una competencia real, y si todo va bien, el resultado será que los diseñadores Java de Sun tomaran una cuidadosa vista de C# y por qué los programadores pueden desear moverse a C#, y responder haciendo mejoras fundamentales en Java.

Actualmente, la principal vulnerabilidad y cuestión importante concerniente a **.NET** es si Microsoft permitirá sea portado *completamente* hacia otras plataformas. Ellos indican que no hay problemas en hacer esto, y el proyecto Mono (www.go-mono.com) tiene una implementación parcial de **.NET** trabajando sobre Linux, pero hasta la implementación esté completa y Microsoft no decida aplastar cualquier parte de él, **.NET** tiene una solución de plataforma cruzada que aún es una apuesta arriesgada.

Para aprender más sobre **.NET** y **C#**, vea *Thinking in C#* de Larry O'Brien y Bruce Eckel, Prentice Hall 2003.

Seguridad

Bajar automáticamente programas y ejecutarlos a través de la Internet puede sonar como el sueño de los constructores de virus. Si cliquea en un sitio Web, puede automáticamente bajar cualquier número de cosas a lo largo de la página HTML: archivos GIF, código script, código Java compilado, y componentes ActiveX. Algunos de estos son benignos: los archivos GIF no pueden dañar nada, y los lenguajes scripting están limitados generalmente en lo que pueden hacer. Java también está diseñado para ejecutar sus applets dentro de un "sandbox" de seguridad, el cual previene de escribir al disco o acceder a memoria fuera del sandbox.

ActiveX de Microsoft es el oponente final en el espectro. Programar con ActiveX es como programar Windows - puede hacer lo que desee. Así si cliquea una página que baja un componente ActiveX, ese componente puede causar daños a los archivos de su disco. De hecho, los programas que carga dentro de su computadora que no están restringidos para ejecutarse dentro de un navegador Web puede hacer la misma cosa. Los virus bajados desde los Bulletin-Board Systems (BBSs) han sido largamente un problema, pero la velocidad de la Internet amplifica la dificultad.

La solución parece ser las "firmas digitales" donde el código está verificado para mostrar el autor de este. Esto está basdo en la idea de que un virus trabaja porque su creador puede ser anónimo, así si saca la anonimidad, los individuos serán forzados a ser responsables de sus acciones. Esto se ve como un buen plan porque permite a los programas ser mucho más funcionales, y sospecho que eliminarán las destrucciones maliciosas. Si, sin embargo, un programa tiene un error de destrucción sin intenciones, aún causará problemas.

La aproximación de Java es prevenir estos problemas de que ocurran, mediante el sandbox. El intérprete de Java que está en su navegador Web local examinada el applet por cualquier instrucción inesperada cuando el applet está siendo cargado. En particular, el applet no puede escribir archivos a disco ni borrarlos (uno de los pilares de los virus). Los applets están considerados generalmente para ser seguros, y ya que esto es esencial para la confianza de los sistemas cliente-servidor, cualquier error en el lenguaje Java que permita a los virus sean rápidamente reparados. (Esto es no tiene sentido que el navegador actualmente mejore estas restricciones de seguridad, y algunos navegadores permiten seleccionar diferentes niveles de seguridad para proveer varios grados de acceso a su sistema).

Puede que sea escéptico de esta restricción algo draconiana en contra de escribir archivos a su disco local. Por ejemplo, puede que desee construir una base de datos local o guardar datos para después usarlos sin conexión. La visión inicial parece ser que todo eventualmente se obtendrá en línea para hacer cualquier cosa importante, pero esto pronto se verá impráctico (aunque el bajo costo de las "aplicaciones Internet" pueden algún día satisfacer las necesidades de un segmento significativo de usuarios). La solución es la "señalización de applets" que usa encriptación de llave pública para verificar

que un applet no haga solamente aquello que indica de donde procede. Un applet señalizado puede aún destruir su disco, pero la teoría es que ya que puede ahora mantener la cantidad de creadores de applet, ellos no hagan cosas maliciosas. Java provee un marco para la firma digital así que eventualmente estará habilitado para permitir a un applet pararse fuera del sandbox si es necesario. El capítulo 14 contiene un ejemplo de cómo señalizar un applet.

Además, Java Web Start es una forma relativamente nueva para distribuir fácilmente programas únicos que no necesiten un navegador web en los cuales correr. Esta tecnología tiene el potencial para resolver muchos problemas del lado cliente asociados con programas que corren dentro de un navegador. Los programas Web Start pueden ya sea estar señalizados, o pueden preguntar al cliente para que permita cada vez que ellos estén haciendo algo potencialmente peligroso en el sistema local. El capítulo 14 tiene un ejemplo simple y explayado sobre Java Web Start.

La firma digital ha errado en una cuestión importante, la cual es la velocidad que la gente se mueve a través de la Internet. Si baja un programa con errores y hace algo inesperado, ¿cuánto tiempo estará antes de descubrir el daño?. Pueden ser días o aún semanas. Para entonces, ¿cómo rastreará hacia atrás lo que el programa ha hecho? ¿Y qué bien hará en este punto?

Internet contra intranet

La web es la solución más general para el problema cliente/servidor, así tiene sentido para utilizar la misma tecnología para resolver un subgrupo de problemas, en particular el clásico problema cliente/servidor *dentro* de una empresa. Con la aproximación tradicional cliente/servidor tiene el problema de diferentes tipos de computadoras cliente, tanto como la dificultad de instalar un nuevo software cliente, ambos son difíciles de solucionar con navegadores Web y programación del lado cliente. Cuando la tecnología Web está usada para una red de información que está restringida a una empresa particular, es referida como una intranet. Las intranets proveen mucha mayor seguridad que la Internet, ya que usted puede controlar físicamente el acceso a los servidores dentro de su compañía. En términos de capacitación, se ve que una persona entiende el concepto general de navegador es mucho más fácil para ella tratar con las diferencias en la forma de páginas y applets, así la curva de aprendizaje de una nueva forma de sistemas se ve reducida.

El problema de seguridad que nos trae para una de las divisiones que veremos ser automáticamente formada en al mundo de la programación del lado cliente. Si su programa está ejecutándose en la Internet, no conoce qué plataforma estará corriendo debajo, y querrá ser extremadamente cuidadoso que no disemine código con errores. Necesita algo de plataforma cruzada y seguridad, como un lenguaje de scripting o Java.

Si está corriendo en una intranet, debe tener un grupo de consideraciones diferentes. No es poco común que sus máquinas puedan ser todas plataformas Intel/Windows. En una intranet, es responsable por la calidad de su propio

código y poder reparar errores cuando sean descubiertos. Además, puede que tenga hoy un cuerpo de código legal que ha sido usado en la aproximación cliente/servidor más tradicional, donde debe instalar físicamente los programas cliente cada vez que hace una actualización. El tiempo gastado en instalar actualizaciones es la razón más importante a moverse a los navegadores, por que las actualizaciones son invisibles y automáticas (Java Web Start es también una solución a este problema). Si está involucrado dentro de una intranet, la aproximación más sensible para tomar el camino más corto que permita usar su código base existente, en vez de intentar recodificar sus programas en un nuevo lenguaje.

Cuando encara con este desconcertante arreglo de soluciones para el problema de programación del lado cliente, el mejor plan de ataque el un análisis de costo-beneficio. Considere las restricciones de su problema y cuál sería el camino más corto para su solución. Ya que la programación del lado cliente es aún programación, es todavía una buena idea tomar la aproximación de desarrollo más rápida para su situación particular. Esta es una postura agresiva para preparar encuentros inevitables con los problemas de desarrollo de programas.

Programación del lado servidor

Esta discusión primordial ha ignorado el caso de la programación del lado servidor. ¿Qué sucede cuando hace una petición al servidor? La mayor de las veces es una petición del estilo "envíame este archivo". Su navegador interpreta entonces el archivo de manera apropiada: como una página HTML, una imagen gráfica, un Applet Java, un programa script, etc. Una petición más compleja generalmente involucra una transacción con base de datos. Un escenario común envuelve una petición para una búsqueda compleja en una base de datos, la cual el servidor da forma luego dentro de una página HTML y le envía como resultado. (De hecho, si el cliente tiene más inteligencia vía lenguaje Java o scripting, el dato en crudo puede ser enviado y formados en el cliente final, lo cual será más rápido y menor carga para el servidor). O puede desear registrar su nombre en una base de datos cuando se une a un grupo o ubicar un orden, lo cual incluye cambios en aquella base de datos. Estas peticiones deben ser procesadas mediante algún código en el lado servidor, el cual es referido normalmente como programación del lado servidor. Tradicionalmente, la programación del lado servidor ha sido realizada utilizando Perl, Python, C++, o algún otro lenguaje, parra crear programas CGI, pero sistemas más sofisticados han ido apareciendo. Estos incluyen servidores Web basados en Java que permiten realizar programación del lado servidor en Java para escribir lo que se denomina servlets. Estos y sus consecutivos, JSPs, son dos de las más importantes razones por las cuales las compañías que desarrollan sitios Web están cambiando a Java, especialmente porque eliminan el problema de tratar con diferentes navegadores (estos tópicos están tratados en *Thinking in Enterprise Java*).

Aplicaciones

Mucho de lo que se habla sobre Java ha sido sobre los applets. Java es actualmente un lenguaje de programación de propósitos generales - al menos en teoría. Y como se apuntó previamente, esta debe ser la manera más efectiva de resolver la mayoría de los problemas cliente/servidor. Cuando se mueve por fuera del terreno del applet (y simultáneamente libera las restricciones, tales como el peliagudo escritura de disco) ingresa el mundo de las aplicaciones de propósito general que corren en solitario, sin un navegador Web, como cualquier programa ordinario hace. Aquí, Java es fuerte no solamente en portabilidad, sino también en programabilidad. Como verá a través de este libro, Java tiene muchas características que permiten crear programas robustos en períodos más cortos que con los lenguajes de programación anteriores.

Estamos conscientes que esto es una bendición mezclada. Paga por la mejoras a través de velocidad de ejecución más lenta (aunque hay un trabajo significativo avanzando en esta área - en particular, la mejora de rendimiento llamada "hotspot" en versiones recientes de Java). Como algunos lenguajes, Java tiene limitaciones de construcción que pueden hacerlo inapropiado para solucionar ciertos tipos de problemas de programación. Java es un lenguaje de rápida evolución, sin embargo, y con cada nuevo lanzamiento viene con más y más atractivos para resolver grandes grupos de problemas.

¿Por qué Java satisface?

La razón que Java haya sido tan satisfactorio es que la meta fue solucionar muchos problemas que los desarrolladores encaran actualmente. Una meta fundamental de Java es incrementar la productividad. Esta productividad viene de muchas maneras, pero el lenguaje está diseñado para ser significativamente mejor que sus antecesores, y proveer importantes beneficios para el programador.

Los sistemas son fáciles de expresar y entender

Las clases diseñadas para completar el problema tienden a expresarlo mejor. Esto significa que cuando escribe código, está describiendo su solución en términos del espacio del problema ("Ponga el grommet en el bin") en vez de los términos de la computadora, el cual es el espacio de solución ("Fije el bit en el chip que significa que el relevado se cerrará"). Trata con conceptos de alto nivel y puede hacer mucho más con una simple línea de código.

El otro beneficio de esta facilidad de expresión es la mantención, el cual (si los reportes pueden ser creídos) es una porción importante del sobre costo en el tiempo de vida del programa. Si los programas son fáciles de entender, son fáciles de mantener. Esto puede también reducir el costo de crear y mantener la documentación.

Influencia máxima con las librerías

La forma más rápida para crear un programa es usar el código que ya está escrito: una librería. Una meta superlativa en Java es hacer una librería de fácil uso. Esto es cumplimentado mediante el modelado de librerías dentro de nuevos tipos de datos (clases), así que el acrecentamiento en una librería significa agregar nuevos tipos al lenguaje. Porque el compilador Java toma cuidadosamente como es utilizada la librería - garantizando la inicialización apropiada y limpieza, y asegurar que los métodos sean llamados apropiadamente - puede focalizarse en qué desea que haga la librería, y no cómo tiene que hacerlo.

Manejo de error

El manejo de errores en C es un problema notorio, y uno que es siempre ignorado - el cruce de dedos está involucrado usualmente. Si está construyendo un grande, complejo problema, no hay nada tan preocupante como tener un error que no se tiene pistas de donde viene. El *manejo de excepciones* es una forma de garantizar que un error sea informado, y que algo suceda como respuesta.

Programando en grande

Muchos programas tradicionales tienen limitaciones en la construcción para tamaños de programa y complejidad. BASIC, por ejemplo, puede ser grandioso par conseguir soluciones rápidas en ciertas clases de problemas, pero si el programa obtiene más que unas pocas páginas de largo, es como intentar nadar en un fluido por demás viscoso. No hay línea clara que indique cuando su programa está fallándole, y si lo hay, usted lo ignora. Usted no dice "Mi programa BASIC está siendo demasiado grande, tendré que rescribirlo en C". En vez de ello, intenta calzar unas pocas líneas más para agregar una nueva característica. Así los costos extras vienen a acrecentarlo.

Java está diseñado para ayudarlo a *programar en grande* - esto es, borrar estos límites de complejidad creciente entre un programa pequeño y uno grande. Seguramente no necesita usar POO cuando escriba un programa del estilo "Hola, mundo", pero las características están allí cuando las necesite. Y el compilador es agresivo para descubrir los errores producidos en programas tanto pequeños como grandes.

¿Java contra C++?

Java se parece muchísimo a C++, así que naturalmente debiera parecer que C++ será remplazada por Java. Pero estoy empezando a discernir de esta lógica. Por un lado, C++ aún tiene características que Java no tiene, y aunque en este punto han sido un montón de promesas sobre que Java algún día será tan rápido o más rápido que C++, estamos viendo algunas mejoras pero sin progresos alarmantes. También, ahí parece estar continuamente interesados en C++, así que no pienso que el lenguaje está yendo al olvido en corto tiempo. Los lenguajes parecen estar siempre ahí.

Estoy empezando a pensar que la fortaleza de Java miente en un terreno ligeramente diferente que C++, el cual es un lenguaje que no intenta llenar un molde. Ciertamente ha sido adaptado en un número de formas para resolver los problemas particulares. Algunas herramientas de C++ combinan librerías, modelos componentes y herramientas de generación de código para solucionar el problema de desarrollar aplicaciones de ventana para usuarios de aplicaciones (para Microsoft Windows). ¿Y aún así, cuál usan la amplia mayoría de los desarrolladores Windows? Microsoft Visual BASIC (VB). Esto desconcierta por el hecho que VB produce el tipo de código que se hace inmanejable cuando el programa es solamente unas pocas páginas de largo (y la sintaxis que puede ser positivamente desconcertante). Tan exitoso y popular como VB es, no hay un buen ejemplo de lenguaje de diseño. Sería hermoso tener la facilidad y poder de VB sin el código resultante inmanejable. Y hay donde pienso que Java deslumbrará: como el "próximo VB" **[8]**. Usted puede o no estremecerse para oír esto, pero piense sobre ello: como mucho de Java es orientado para hacer fácil a los programadores resolver problemas a nivel de aplicación como redes y UI para plataforma cruzada, y aún tiene un lenguaje de diseño que permita la creación de cuerpos muy grandes y flexibles de código. Agregue a esto el hecho de que el chequeo de tipo y manejo de error de Java es una impresionante mejora sobre la mayoría de los lenguajes y tiene la creación de un salto significativo hacia adelante en la productividad de programación.

[8] Microsoft está efectivamente diciendo "no es tan rápido" como C# y .NET. Numerosas personas han planteado la cuestión de si los programadores de VB desean cambiar *cualquier* opción, ya sea Java, C# o aún VB.NET.

Si está desarrollando todo su código desde el principio, entonces la simplicidad de Java sobre C++ significará acortar su tiempo de desarrollo - la evidencia anecdótica (cuentan los grupos de C++ con que yo he hablado quiénes han cambiados a Java) sugiere un doble de velocidad de desarrollo sobre C++. Si el rendimiento de Java no es problema o puede de alguna manera compensarlo, la cuestión de tiempo de mercado hace difícil elegir C++ por sobre Java.

La cuestión más grande es el rendimiento. La interpretación de Java ha sido lenta, cerca de 20 a 50 veces más lento que C en los intérpretes de Java original. Esto ha mejorado grandemente el sobre tiempo (especialmente con las más recientes versiones de Java), pero aún el sobrante es un número importante. Las computadoras están sobre la velocidad; si no es significativamente rápida para hacer algo entonces lo hago con mis manos. (Siempre oí sugerir que inicie con Java, engañe con el tiempo corto de desarrollo, entonces usa una herramienta y librerías de soporte para traducir su código a C++, si necesita velocidad de ejecución rápida).

La clave para hacer Java viable para muchos proyectos de desarrollo es la apariencia de mejoras de velocidad como los compiladores llamados "just-in-time (JIT)", la tecnología propietaria de Sun "hotspot" y aún compiladores de código nativo. De hecho, los compiladores de código nativo eliminarán la

ejecución de plataforma cruzada revendida de los programas compilados, pero también acercarán la velocidad de los ejecutables muy cerca de C y de C++. (En teoría, es recompilar, pero esta promesa ha sido hecha antes para otros lenguajes).

Resumen

Este capítulo intenta darle un sentido a los usos de la programación orientada al objeto y Java, incluyendo por qué POO es diferente, y porque Java en particular es diferente.

La POO y Java puede no ser para todo. es importante evaluar sus propias necesidades y decidir si Java óptimamente satisfará estas necesidades, o si puede ser mejor pasar a otro sistema de programación (incluyendo el que actualmente está usando). Si conoce que sus necesidades estarán muy especializadas para el para el futuro previsible y si tiene restricciones específicas que no pueden ser satisfechas con Java, entonces su deber para sí mismo es investigar alternativas (en particular, recomiendo mirar Python; vea www.Python.org). Incluso si eventualmente elige Java como su lenguaje, al menos entenderá que las opciones están y tendrá una visión clara de porque tomar esa dirección.

Usted conoce a qué se parece un programa procedural: definiciones de datos y llamadas a funciones. Para encontrar significado a tales programas, tiene que trabajar un poco, observar a través de las llamadas a funciones y conceptos de bajo nivel para crear un modelo en su mente. Esta es la razón porque necesitamos representaciones intermedias cuando diseñamos programas procedurales - por sí mismos, estos programas tienden a ser confusos porque los términos de expresión están orientados más hacia la computadora que al problema que está resolviendo.

Porque Java agrega algunos conceptos nuevos arriba de los que encuentra en un lenguaje procedural, su interpretación natural puede ser que el **main( )** en un programa Java será un poco más complicado que para un programa C equivalente. Aquí, estará placenteramente sorprendido: un programa bien escrito en Java es generalmente un tanto más simple y mucho más fácil de entender que el programa equivalente en C. Lo que verá son las definiciones de objetos que representan conceptos en el espacio del problema (en vez de las representaciones de las cuestiones de la computadora) y enviará mensajes a estos objetos para representar las actividades en ese espacio. Uno de los puntales de la programación orientada al objeto es que, con un programa bien diseñado, es fácil entender el código par leerlo. Usualmente, hay un poco menos de código asimismo, porque muchos de sus problemas serán resueltos reutilizando la librería de código existente.