Introducción a los conceptos orientados a objetos

Aunque muchos programadores no se dan cuenta, el desarrollo de software orientado a objetos (OO) ha existido desde principios de la década de 1960. No fue hasta mediados y finales de la década de 1990 que el paradigma orientado a objetos comenzó a ganar impulso, a pesar del hecho de que los lenguajes de programación orientados a objetos populares como Smalltalk y C ++ ya se usaban ampliamente.

El auge de las metodologías OO coincide con el surgimiento de Internet como plataforma de negocios y entretenimiento. En resumen, los objetos funcionan bien en una red. Y después de que se hizo evidente que Internet había llegado para quedarse, las tecnologías orientadas a objetos ya estaban bien posicionadas para desarrollar las nuevas tecnologías basadas en la web.

Es importante señalar que el título de este primer capítulo es " Introducción a los conceptos orientados a objetos ". La palabra operativa aquí es "conceptos" y no "tecnologías". Las tecnologías cambian muy rápidamente en la industria del software, mientras que los conceptos evolucionan. Utilizo el término "evolucionar" porque, aunque permanecen relativamente estables, cambian. Y esto es lo realmente genial de centrarse en los conceptos. A pesar de su coherencia, siempre se están reinterpretando y esto permite discusiones muy interesantes.

Esta evolución se puede rastrear fácilmente durante los últimos 25 años a medida que seguimos la progresión de las diversas tecnologías de la industria desde los primeros navegadores primitivos de mediados a finales de la década de 1990 hasta las aplicaciones móviles / telefónicas / web que dominan en la actualidad. Como siempre, los nuevos desarrollos están a la vuelta de la esquina a medida que exploramos aplicaciones híbridas y más. A lo largo de este viaje, los conceptos de OO han estado presentes en cada paso del camino. Por eso los temas de este capítulo son tan importantes. Estos conceptos son tan relevantes hoy como lo eran hace 25 años.

LOS CONCEPTOS FUNDAMENTALES

El objetivo principal de este libro es que piense en cómo se utilizan los conceptos en el diseño de sistemas orientados a objetos. Históricamente, los lenguajes orientados a objetos se definen por lo siguiente: encapsulación, herencia y polimorfismo (lo que yo llamo OO “clásico”). Por lo tanto, si un lenguaje no implementa todos estos, generalmente no se considera completamente orientado a objetos. A lo largo de estos tres términos, siempre incluyo la composición en la mezcla; por lo tanto, mi lista de conceptos orientados a objetos se ve así:

- Encapsulamiento
- Herencia
- Polimorfismo
- Composición

Discutiremos todo esto en detalle a medida que avancemos en el resto del libro.

Uno de los problemas con los que he luchado desde la primera edición de este libro es cómo estos conceptos se relacionan directamente con las prácticas de diseño actuales, que siempre están cambiando. Por ejemplo, siempre ha habido un debate sobre el uso de la herencia en un diseño orientado a objetos. ¿La herencia realmente rompe la encapsulación? (Este tema se tratará en capítulos posteriores). Incluso ahora, muchos desarrolladores intentan evitar la herencia tanto como sea posible. Entonces, esto plantea la pregunta: ¿Debería usarse la herencia en absoluto?

Mi enfoque es, como siempre, ceñirme a los conceptos. Ya sea que use la herencia o no, al menos debe comprender qué es la herencia, lo que le permitirá tomar una decisión de diseño informada. Es importante no olvidar que es casi seguro que la herencia se encuentre en el mantenimiento del código, por lo que debe aprenderla independientemente.

Como se mencionó en la introducción, el público objetivo son aquellos que desean una introducción general a los conceptos fundamentales de OO. Con esta afirmación en mente, en este capítulo presento los conceptos fundamentales orientados a objetos con la esperanza de que luego obtenga una base sólida para tomar decisiones de diseño importantes. Los conceptos cubiertos aquí tocan la mayoría, si no todos, de los temas cubiertos en capítulos posteriores, que exploran estos temas con mucho más detalle.

OBJETOS Y SISTEMAS HEREDADOS

A medida que OO se movió hacia la corriente principal, uno de los problemas que enfrentaron los desarrolladores fue la integración de nuevas tecnologías OO con los sistemas existentes. Se estaban trazando líneas entre la programación orientada a objetos y la programación estructurada (o procedimental), que era el paradigma de desarrollo dominante en ese momento. Siempre me pareció extraño porque, en mi opinión, la programación estructurada y orientada a objetos no compiten entre sí. Son complementarios porque los objetos se integran bien con el código estructurado. Incluso ahora, a menudo escucho esta pregunta: ¿es usted un programador estructurado o un programador orientado a objetos? Sin dudarlo, respondería: ambos.

Del mismo modo, el código orientado a objetos no está destinado a reemplazar el código estructurado. Muchos *sistemas heredados que* no son de OO (es decir, sistemas más antiguos que ya están instalados) están haciendo su trabajo bastante bien, entonces, ¿por qué arriesgarse a un desastre potencial cambiándolos o reemplazándolos? En la mayoría de los casos, no debe cambiarlos, al menos no por el bien del cambio. No hay nada intrínsecamente incorrecto en los sistemas escritos en código que no sea OO. Sin embargo, el desarrollo completamente nuevo definitivamente justifica la consideración de usar tecnologías OO (en algunos casos, no hay más remedio que hacerlo).

Aunque ha habido un crecimiento constante y significativo en el desarrollo de Orientado a Objetos en los últimos 25 años, la dependencia de la comunidad global de redes como Internet e infraestructuras móviles ha ayudado a catapultarlo aún más en la corriente principal. La explosión de transacciones realizadas en navegadores y aplicaciones móviles ha abierto nuevos mercados, donde gran parte del desarrollo de software es nuevo y, en su mayoría, no está afectado por preocupaciones

heredadas. Incluso cuando existen preocupaciones sobre el legado, existe una tendencia a envolver los sistemas legacy en envoltorios de objetos.

Envoltorios de objetos

Los envoltorios de objetos son códigos orientados a objetos que incluyen otro código en su interior. Por ejemplo, puede tomar código estructurado (como bucles y condiciones) y *envolverlo* dentro de un objeto para que parezca un objeto. También puede utilizar envoltorios de objetos para *envolver* funciones como características de seguridad, características de hardware no portátiles, etc. La envoltura del código estructurado se trata en detalle en el Capítulo 6 , " Diseñar con objetos ".

Una de las áreas más interesantes del desarrollo de software es la integración de código heredado con sistemas móviles y basados en web. En muchos casos, una interfaz web móvil finalmente se conecta a los datos que residen en un mainframe. Se necesitan desarrolladores que puedan combinar las habilidades del mainframe y el desarrollo web móvil.

Probablemente experimente objetos en su vida diaria sin siquiera darse cuenta. Estas experiencias pueden tener lugar en su automóvil, cuando está hablando por su teléfono celular, usando su sistema de entretenimiento doméstico, jugando juegos de computadora y muchas otras situaciones. La autopista electrónica se ha convertido, en esencia, en una autopista basada en objetos. A medida que las empresas gravitan hacia la web móvil, gravitan hacia los objetos porque las tecnologías utilizadas para el comercio electrónico son en su mayoría de naturaleza orientada a objetos.

Web móvil

Sin duda, la aparición de Internet proporcionó un gran impulso para el cambio a tecnologías orientadas a objetos. Esto se debe a que los objetos son adecuados para su uso en redes. Aunque Internet estuvo a la vanguardia de este cambio de paradigma, las redes móviles ahora se han sumado a la mezcla de manera importante. En este libro, el término *web móvil* se utilizará en el contexto de conceptos que pertenecen tanto al desarrollo de aplicaciones móviles como al desarrollo web. El término aplicación *híbrida a* veces se usa para referirse a aplicaciones que se procesan en navegadores tanto en la web como en dispositivos móviles.

PROGRAMACIÓN PROCESAL VERSUS PROGRAMACIÓN ORIENTADA A OBJETOS

Antes de profundizar en las ventajas del desarrollo orientado a objetos, consideremos una pregunta más fundamental: ¿Qué es exactamente un objeto? Esta es una pregunta compleja y simple. Es complejo porque aprender cualquier método de desarrollo de software no es trivial. Es simple porque la gente ya piensa en términos de objetos.

TIP

Al ver una conferencia en video de YouTube presentada por el gurú de OO, Robert Martin, su opinión es que la declaración de que "la gente piensa en términos de objetos" fue acuñada por gente de marketing. Solo algo para pensar.

Por ejemplo, cuando miras a una persona, la ves como un objeto. Y un objeto se define por dos componentes: atributos y comportamientos. Una persona tiene atributos, como el color de ojos, la edad, la altura, etc. Una persona también tiene comportamientos, como caminar, hablar, respirar, etc. En su definición básica, un *objeto* es una entidad que contiene *tanto* datos como comportamiento. La palabra *ambos* es la diferencia clave entre la programación OO y otras metodologías de programación. En la programación de procedimientos, por ejemplo, el código se coloca en funciones o procedimientos totalmente distintos. Idealmente, como se muestra en la Figura 1.1, estos procedimientos luego se convierten en "cajas negras", donde entran las entradas y salen las salidas. Los datos se colocan en estructuras independientes y estas funciones o procedimientos los manipulan.

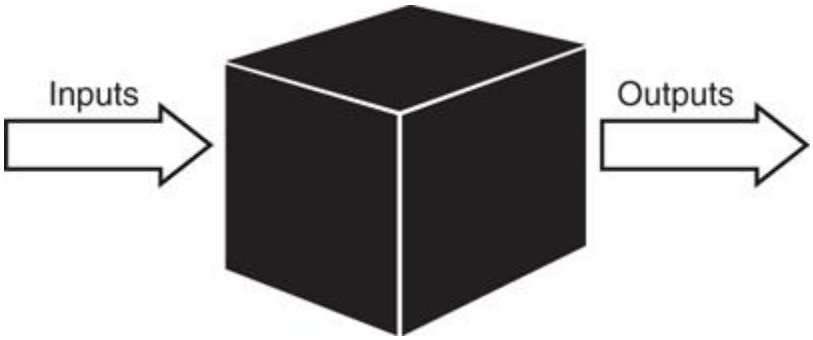

Figura 1.1 Cajas negras.

Diferencia entre OO y procedimental

En el diseño OO, los atributos y comportamientos están contenidos dentro de un solo objeto, mientras que en el diseño procedimental o estructurado, los atributos y comportamientos normalmente están separados.

A medida que el diseño de OO crecía en popularidad, una de las realidades que inicialmente ralentizó su aceptación fue que había muchos sistemas que no eran de OO y que funcionaban perfectamente bien. Por lo tanto, no tenía ningún sentido comercial cambiar los sistemas en aras del cambio. Cualquiera que esté familiarizado con cualquier sistema informático sabe que cualquier cambio puede significar un desastre, incluso si el cambio se percibe como leve.

Esta situación entró en juego con la falta de aceptación de las bases de datos orientadas a objetos. En un momento del surgimiento del desarrollo de OO, parecía algo probable que las bases de datos OO reemplazaran a las bases de datos relacionales. Sin embargo, esto nunca sucedió. Las empresas tienen mucho dinero invertido en bases de datos relacionales y un factor primordial desalienta la conversión: funcionaron. Cuando se hicieron evidentes todos los costos y riesgos de convertir sistemas de bases de datos relacionales a OO, no había ninguna razón de peso para cambiar.

De hecho, las fuerzas empresariales ahora han encontrado un feliz término medio. Muchas de las prácticas de desarrollo de software actuales tienen variantes de varias metodologías de desarrollo, como OO y estructuradas.

Como se ilustra en la Figura 1.2 , en la programación estructurada, los datos a menudo se separan de los procedimientos y, a menudo, los datos son globales, por lo que es fácil modificar los datos que están fuera del alcance de su código. Esto significa que el acceso a los datos es incontrolado e impredecible (es decir, varias

funciones pueden tener acceso a los datos globales). En segundo lugar, debido a que no tiene control sobre quién tiene acceso a los datos, las pruebas y la depuración son mucho más difíciles. Los objetos abordan estos problemas combinando datos y comportamiento en un paquete completo y agradable.

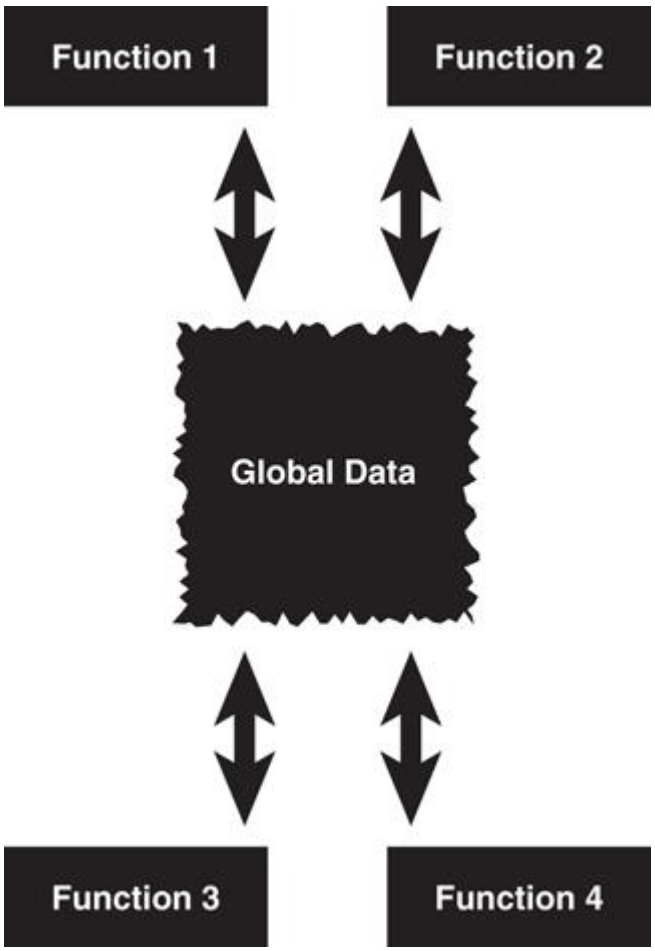

Figura 1.2 Usando datos globales.

Diseño adecuado

Podemos afirmar que cuando se diseña correctamente, no existen datos globales en un modelo orientado a objetos. Este hecho proporciona una gran cantidad de integridad de datos en los sistemas OO.

En lugar de reemplazar otros paradigmas de desarrollo de software, los objetos son una respuesta evolutiva. Los programas estructurados tienen estructuras de datos complejas, como matrices, etc. C ++ tiene estructuras que tienen muchas de las características de los objetos (clases).

Sin embargo, los objetos son mucho más que estructuras de datos y tipos de datos primitivos, como números enteros y cadenas. Aunque los objetos contienen entidades como números enteros y cadenas, que se utilizan para representar atributos, también contienen métodos, que representan comportamientos. En un objeto, los métodos se utilizan para realizar operaciones en los datos, así como otras acciones. Quizás lo más importante es que puede controlar el acceso a los miembros de un objeto (tanto atributos como métodos). Esto significa que algunos miembros, tanto atributos como métodos, pueden ocultarse de otros objetos. Por ejemplo, un objeto llamado `Math`podría contener dos números enteros,

llamados `myInt1` y `myInt2`. Lo más probable es que el `Math` objeto también contenga los métodos necesarios para establecer y recuperar los valores de `myInt1` y `myInt2`. También puede contener un método llamado `sum()` para sumar los dos enteros.

Ocultación de datos

En la terminología OO, los datos se denominan *atributos* y los comportamientos se denominan métodos. Restringir el acceso a ciertos atributos y / o métodos se denomina *ocultación de* datos.

Al combinar los atributos y métodos en la misma entidad, en el lenguaje OO se llama *encapsulación*, podemos controlar el acceso a los datos en el objeto Math. Al definir estos números enteros como fuera de los límites, otra función lógicamente desconectada no puede manipular los números enteros **`myInt1`** y **`myInt2`** sólo el objeto Math puede hacer eso.

Pautas de diseño de clases de sonido

Tenga en cuenta que es posible crear clases OO mal diseñadas que no restrinjan el acceso a los atributos de la clase. La conclusión es que puede diseñar código incorrecto con la misma eficacia con el diseño OO que con cualquier otra metodología de programación. Simplemente tenga cuidado de adherirse a las pautas de diseño de clases sólidas.

¿Qué sucede cuando otro objeto, por ejemplo, **`myObject`** desea acceder a la suma de **`myInt1`** y **`myInt2`**? Pregunta al `Math` objeto: `myObject` envía un mensaje al `Math` objeto. La figura 1.3 muestra cómo los dos objetos se comunican entre sí a través de sus métodos. El mensaje es realmente una llamada al método `sum` del objeto Math. El método `sum` a continuación, devuelve el valor a **`myObject`**. La belleza de esto es que **`myObject`** no es necesario saber cómo se calcula la suma (aunque estoy seguro de que se puede adivinar). Con esta metodología de diseño implementada, puede cambiar la forma en que el objeto Math calcula la suma sin realizar un cambio a **`myObject`** (siempre que los medios para recuperar la suma no cambien). Todo lo que quieres es la suma, *no importa* cómo se calcula.

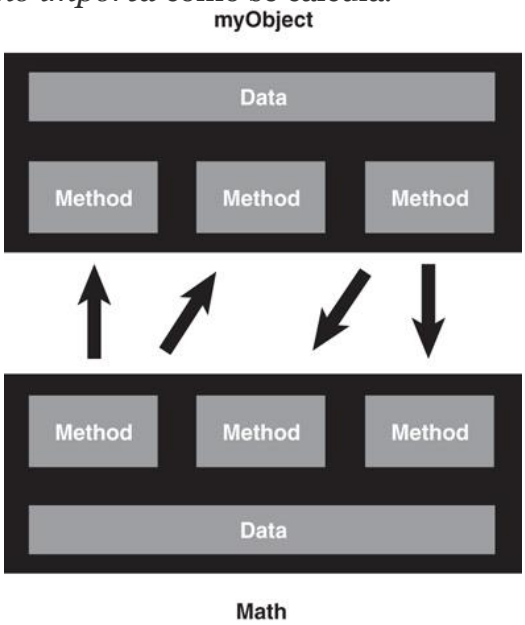

Figura 1.3 Comunicación de objeto a objeto.

El uso de un ejemplo de calculadora simple ilustra este concepto. Al determinar una suma con una calculadora, todo lo que usa es la interfaz de la calculadora: el teclado y la pantalla LED. La calculadora tiene un método suma que se invoca cuando presiona la secuencia de teclas correcta. Puede que le devuelvan la respuesta correcta; sin embargo, no tiene idea de cómo se obtuvo el resultado, ya sea de forma electrónica o algorítmica.

Calcular la suma no es responsabilidad de `myObject`, es responsabilidad del objeto Math. Siempre que `myObject`tenga acceso al objeto Math, puede enviar los mensajes correspondientes y obtener el resultado solicitado. En general, los objetos no deben manipular los datos internos de otros objetos (es decir, `myObject` no deben cambiar directamente el valor de `myInt1`y `myInt2`). Y, por razones que exploraremos más adelante, normalmente es mejor construir objetos pequeños con tareas específicas en lugar de construir objetos grandes que realizan muchas.

PASAR DEL DESARROLLO PROCEDIMENTAL AL DESARROLLO ORIENTADO A OBJETOS

Ahora que tenemos una comprensión general sobre algunas de las diferencias entre las tecnologías de procedimiento y las orientadas a objetos, profundicemos un poco más en ambas.

Programación procedimental

La programación por procedimientos normalmente separa los datos de un sistema de las operaciones que manipulan los datos. Por ejemplo, si desea enviar información a través de una red, solo se envían los datos relevantes (consulte la Figura 1.4), con la expectativa de que el programa en el otro extremo de la tubería de la red sepa qué hacer con ellos. En otras palabras, debe existir algún tipo de acuerdo de intercambio entre el cliente y el servidor para transmitir los datos. En este modelo, es muy posible que no se envíe ningún código por cable.

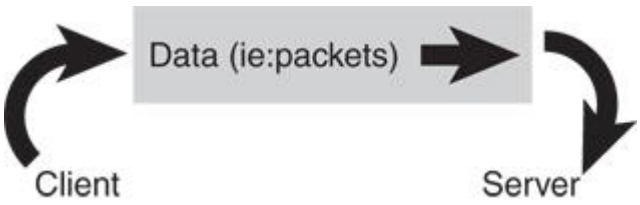

Figura 1.4 Datos transmitidos a través de un cable.

Programación OO

La ventaja fundamental de la programación OO es que los datos y las operaciones que manipulan los datos (el código) están encapsulados en el objeto. Por ejemplo, cuando un objeto se transporta a través de una red, todo el objeto, incluidos los datos y el comportamiento, lo acompaña.

Una sola entidad

Aunque pensar en términos de una sola entidad es genial en teoría, en muchos casos, los comportamientos en sí pueden no enviarse porque ambos lados tienen copias del código. Sin embargo, es importante pensar en términos de que todo el objeto se envía a través de la red como una sola entidad.

En la Figura 1.5 , el `Employee` objeto se envía a través de la red.

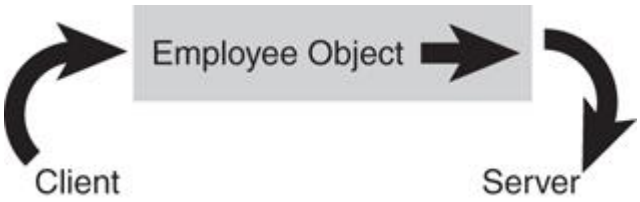

Figura 1.5 Objetos transmitidos a través de un cable.

Diseño adecuado

Un buen ejemplo de este concepto es un objeto que se carga mediante un navegador. A menudo, el navegador no tiene idea de lo que hará el objeto de antemano porque el código no está allí previamente. Cuando se carga el objeto, el navegador ejecuta el código dentro del objeto y usa los datos contenidos dentro del objeto.

¿QUÉ ES EXACTAMENTE UN OBJETO?

Los objetos son los componentes básicos de un programa orientado a objetos. Un programa que utiliza tecnología OO es básicamente una colección de objetos. Para ilustrar, consideremos que un sistema corporativo contiene objetos que representan a los empleados de esa empresa. Cada uno de estos objetos se compone de los datos y el comportamiento que se describen en las siguientes secciones.

Datos del objeto

Los datos almacenados dentro de un objeto representan el estado del objeto. En la terminología de programación OO, estos datos se denominan *atributos* . En nuestro ejemplo, como se muestra en la Figura 1.6 , los atributos de los empleados podrían ser números de Seguro Social, fecha de nacimiento, sexo, número de teléfono, etc. Los atributos contienen la información que diferencia entre los distintos objetos, en este caso los empleados. Los atributos se tratan con más detalle más adelante en este capítulo en la discusión sobre clases.

Figura 1.6 Atributos de los empleados.

Comportamientos de los objetos

El *comportamiento* de un objeto representa lo que el objeto puede hacer. En los lenguajes de procedimiento, el comportamiento se define mediante procedimientos, funciones y subrutinas. En la terminología de programación OO, estos comportamientos están contenidos en *métodos*, y se invoca un método enviándole un mensaje. En nuestro ejemplo de empleado, considere que uno de los comportamientos requeridos de un objeto de empleado es establecer y devolver los valores de los diversos atributos. Por lo tanto, cada atributo tendría métodos correspondientes, como `setGender()` y `getGender()`. En este caso, cuando otro objeto necesita esta información, puede enviar un mensaje a un objeto empleado y preguntarle cuál es su género.

No es sorprendente que la aplicación de captadores y definidores, como ocurre con gran parte de la tecnología orientada a objetos, haya evolucionado desde que se publicó la primera edición de este libro. Esto es especialmente cierto cuando se trata de datos. Recuerde que una de las ventajas más interesantes, por no mencionar poderosas, de usar objetos es que los datos son parte del paquete, no están separados del código.

La aparición de XML no solo ha centrado la atención en la presentación de datos de forma portátil; también ha facilitado formas alternativas para que el código acceda a los datos. En las técnicas .NET, los captadores y definidores se consideran propiedades de los datos en sí.

Por ejemplo, considere un atributo llamado **`Name`**, usando Java, que se parece a lo siguiente:

```
Nombre de cadena pública;
```

El getter y setter correspondientes se verían así:

```
public void setName (String n) {name = n;};
public String getName () {return name;};
```

Ahora, al crear un atributo XML llamado **`Name`**, la definición en C # .NET puede verse así, aunque ciertamente puede usar el mismo enfoque que el ejemplo de Java:

```
 private string strName;

 public String Name
 {
     get { return this.strName; }
     set {
         if (value == null) return;
         this.strName = value;
     }
 }
```

En esta técnica, los captadores y definidores son en realidad *propiedades* de los atributos; en este caso **`Name`**.

Independientemente del enfoque, el propósito es el mismo: acceso controlado al atributo. Para este capítulo, primero quiero concentrarme en la naturaleza

conceptual de los métodos de acceso; veremos más sobre las propiedades en capítulos posteriores.

Getters y Setters

El concepto de captadores y definidores respalda el concepto de ocultación de datos. Debido a que otros objetos no deben manipular datos directamente dentro de otro objeto, los captadores y definidores brindan acceso controlado a los datos de un objeto. Los getters y setters a veces se denominan métodos de acceso y métodos de mutación, respectivamente.

Tenga en cuenta que solo mostramos la interfaz de los métodos y no la implementación. La siguiente información es todo lo que el usuario necesita saber para utilizar eficazmente los métodos:

- el nombre del método
- Los parámetros pasados al método
- El tipo de retorno del método.

Para ilustrar los comportamientos, considere la Figura 1.7 .

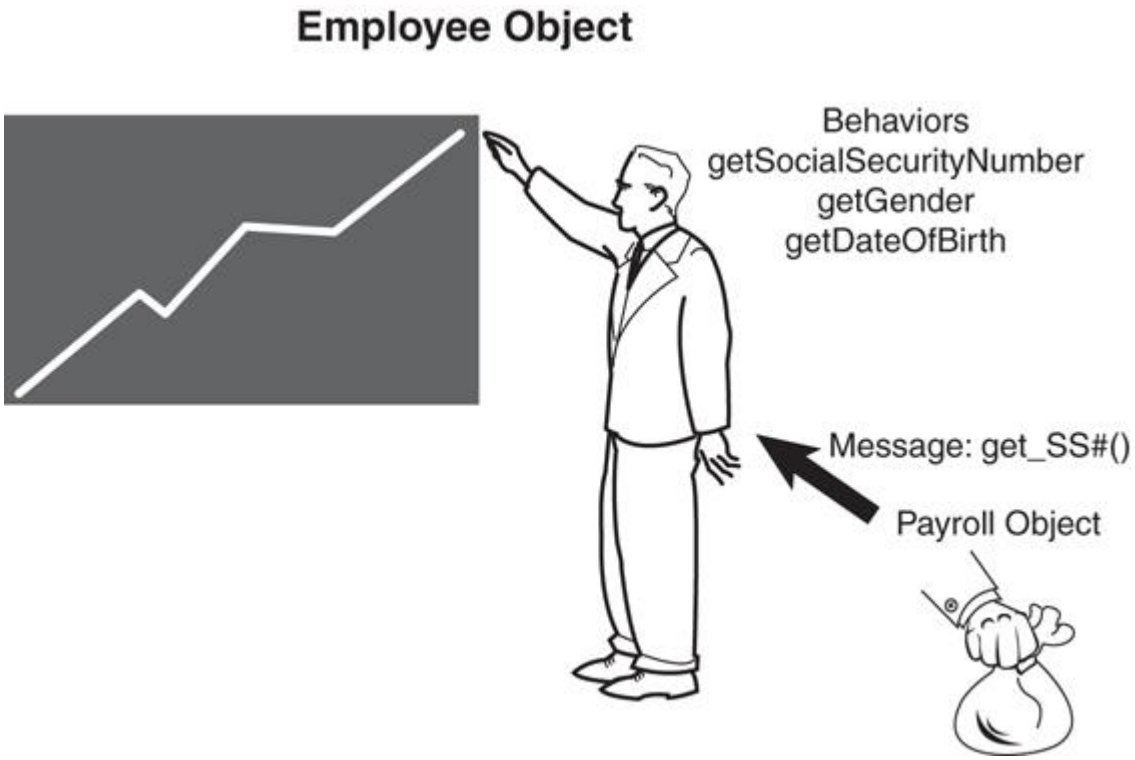

Figura 1.7 Comportamientos de los empleados.

En la Figura 1.7 , el objeto `Payroll` contiene un método llamado `CalculatePay()` que calcula el pago de un empleado específico. Entre otra información, el objeto `Payroll` debe obtener el número de Seguro Social de este empleado. Para obtener esta información, el objeto de nómina debe enviar un mensaje al objeto `Employee` (en este caso, el método `getSocialSecurityNumber()`). Básicamente, esto significa que el objeto `Payroll` llama al `getSocialSecurityNumber()` método del objeto `Employee`. El objeto empleado reconoce el mensaje y devuelve la información solicitada.

Para ilustrar más, la Figura 1.8 es un diagrama de clases que representa el sistema `Employee/` del `Payroll` que hemos estado hablando.

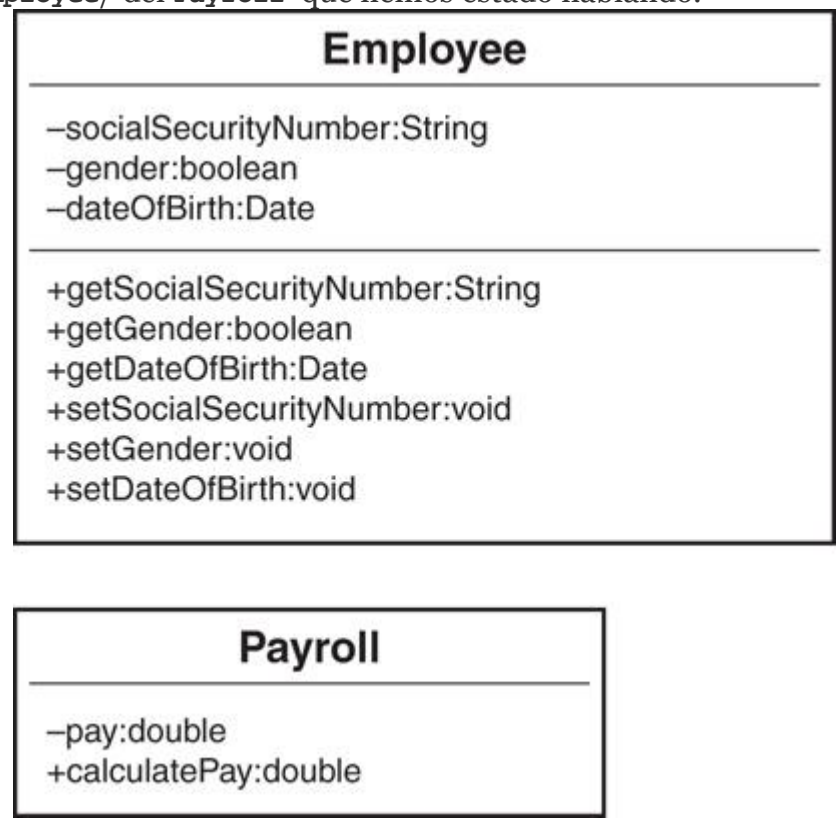

Figura 1.8 Diagramas de clases de empleados y nóminas.

Diagramas de clases UML

Debido a que este es el primer diagrama de clases que hemos visto, es muy básico y carece de algunas de las construcciones (como los constructores) que debería contener una clase adecuada. No temas: analizaremos los diagramas de clases y los constructores con más detalle.

Cada diagrama de clases está definido por tres secciones separadas: el nombre en sí, los datos (atributos) y los comportamientos (métodos). En la Figura 1.8 , la `Employee`sección de atributos del diagrama de clases contiene `SocialSecurityNumber`, `Gender`y `DateofBirth`, mientras que la sección de métodos contiene los métodos que operan sobre estos atributos. Puede utilizar herramientas de modelado UML para crear y mantener diagramas de clases que correspondan al código real.

Herramientas de modelado

Las herramientas de modelado visual proporcionan un mecanismo para crear y manipular diagramas de clases utilizando el Lenguaje de modelado unificado (UML). Los diagramas de clases se utilizan y analizan a lo largo de este libro. Se utilizan como una herramienta para ayudar a visualizar clases y sus relaciones con otras clases. El uso de UML en este libro se limita a los diagramas de clases.

Entraremos en las relaciones entre clases y objetos más adelante en este capítulo, pero por ahora puede pensar en una clase como una plantilla a partir de la cual se hacen los objetos. Cuando se crea un objeto, decimos que se instancian los objetos. Por lo tanto, si creamos tres empleados, en realidad estamos creando tres

instancias totalmente distintas de una clase `Employee`. Cada objeto contiene su propia copia de los atributos y métodos. Por ejemplo, considere la Figura 1.9 . Un objeto empleado llamado `John`(John es su identidad) tiene su propia copia de todos los atributos y métodos definidos en la clase `Employee`.Un objeto empleado llamado `Mary`tiene su propia copia de atributos y métodos. Ambos tienen una copia separada del atributo `DateOfBirth` y el método `getDateOfBirth`.

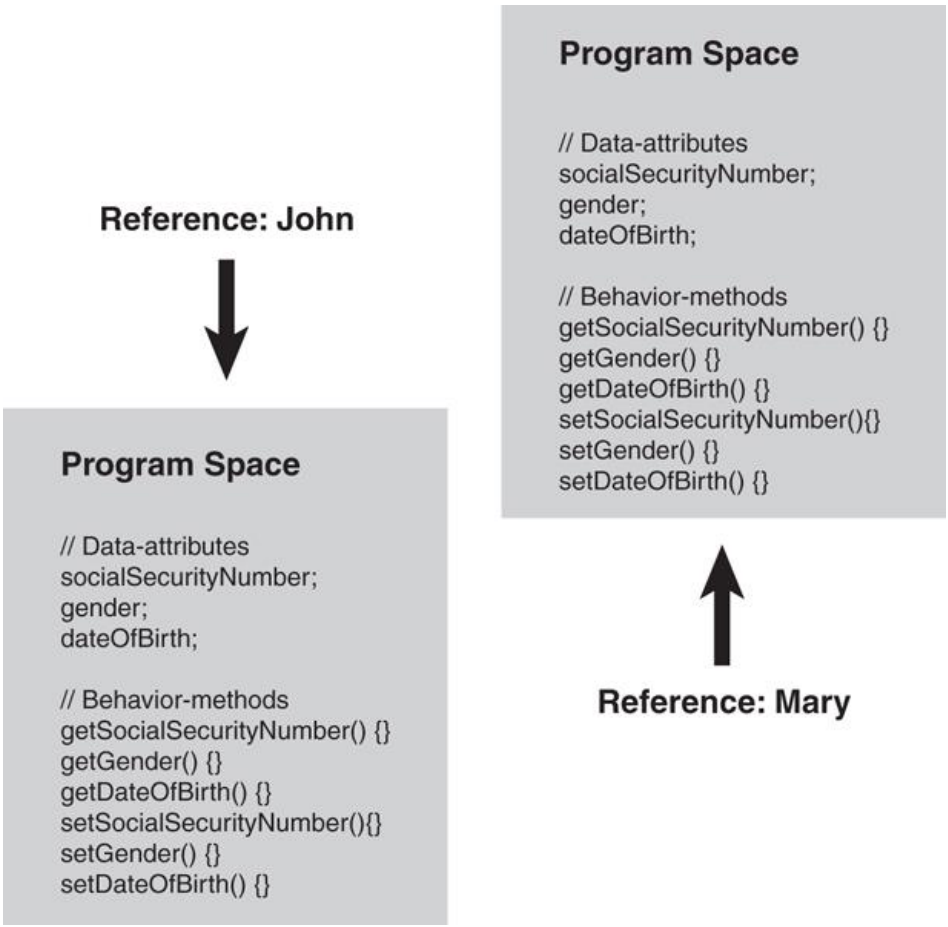

Figura 1.9 Espacios del programa.

Un problema de implementación

Tenga en cuenta que no hay necesariamente una copia física de cada método para cada objeto. Más bien, cada objeto apunta a la misma implementación. Sin embargo, este es un problema que queda en manos del compilador / plataforma operativa. Desde un nivel conceptual, puede pensar en los objetos como totalmente independientes y con sus propios atributos y métodos.

¿QUÉ ES EXACTAMENTE UNA CLASE?

En resumen, una clase es un modelo para un objeto. Cuando crea una instancia de un objeto, usa una clase como base de cómo se construye el objeto. De hecho, tratar de explicar clases y objetos es realmente un dilema del huevo y la gallina. Es difícil describir una clase sin usar el término *objeto* y viceversa. Por ejemplo, una bicicleta individual específica es un objeto. Sin embargo, alguien tuvo que crear o diseñar los planos (es decir, la clase) para construir la bicicleta. En el software de OO, a

diferencia del dilema del huevo y la gallina, sabemos qué es lo primero: la clase. No se puede crear una instancia de un objeto sin una clase. Por tanto, muchos de los conceptos de esta sección son similares a los presentados anteriormente en el capítulo, especialmente cuando hablamos de atributos y métodos.

Aunque este libro se centra en los conceptos del software OO y no en una implementación específica, a menudo es útil usar ejemplos de código para explicar algunos conceptos, por lo que los fragmentos de código Java se utilizan a lo largo del libro para ayudar a explicar algunos conceptos cuando sea apropiado. Sin embargo, para ciertos ejemplos clave, el código se proporciona en varios idiomas como descargas.

Las siguientes secciones describen algunos de los conceptos fundamentales de las clases y cómo interactúan.

Creando Objetos

Las clases se pueden considerar como plantillas, o cortadores de galletas, para objetos como se ve en la Figura 1.10 . Una clase se usa para crear un objeto.

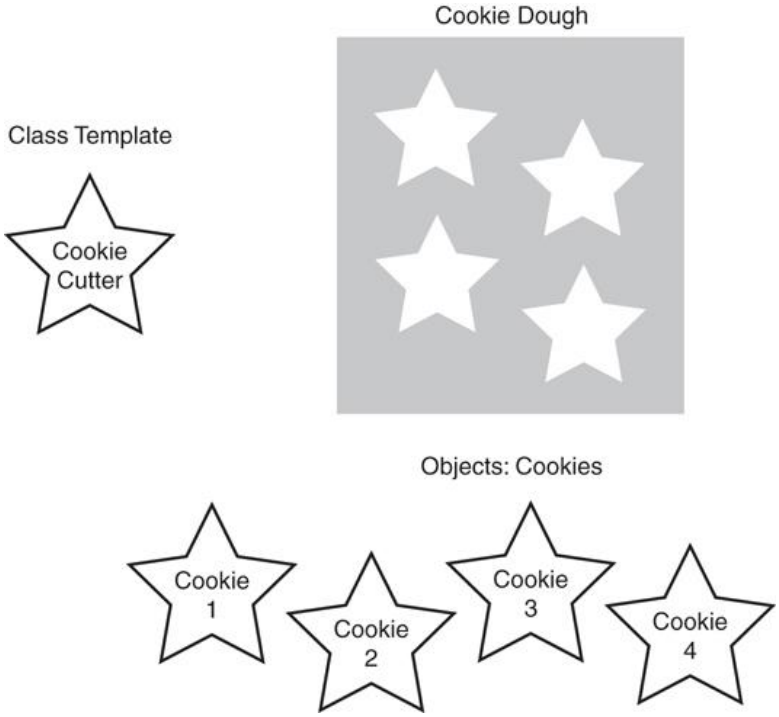

Figura 1.10 Plantilla de clase.

Se puede pensar en una clase como una especie de tipo de datos de nivel superior. Por ejemplo, al igual que crea un número entero o un flotante:

```
int x;
flotar y;
```

también puede crear un objeto usando una clase predefinida:

```
myClassmyObject;
```

En este ejemplo, los nombres mismos hacen obvio que `myClass` es la clase y `myObject` es el objeto.

Recuerde que cada objeto tiene sus propios atributos (datos) y comportamientos (funciones o rutinas). Una clase define los atributos y comportamientos que poseerán todos los objetos creados con esta clase. Las clases son fragmentos de código. Los objetos instanciados de clases se pueden distribuir individualmente o como parte de una biblioteca. Debido a que los objetos se crean a partir de clases, se deduce que las clases deben definir los bloques de construcción básicos de los objetos (atributos, comportamiento y mensajes). En resumen, debe diseñar una clase antes de poder crear un objeto.

Por ejemplo, aquí hay una definición de una clase `Person`:

```
public class Person{

    //Atributos
    private String name;
    private String address;

    //Métodos
    public String getName(){
        return name;
    }
    public void setName(String n){
        name = n;
    }

    public String getAddress(){
        return address;
    }
    public void setAddress(String adr){
        address = adr;
    }

}
```

Atributos

Como ya vio, los datos de una clase están representados por atributos. Cada clase debe definir los atributos que almacenarán el estado de cada objeto instanciado de esa clase. En el ejemplo `Person` de clase de la sección anterior, la clase `Person` define atributos `name` y `address`.

Designaciones de acceso

Cuando un tipo de datos o método se define como público, otros objetos pueden acceder directamente a él. Cuando un tipo de datos o método se define como privado, solo ese objeto específico puede acceder a él. Otro modificador de acceso, protegido, permite el acceso de objetos relacionados, sobre los que aprenderá en el Capítulo 3 .

Métodos

Como aprendió anteriormente en este capítulo, los métodos implementan el comportamiento requerido de una clase. Cada objeto instanciado de esta clase incluye métodos definidos por la clase. Los métodos pueden implementar comportamientos que se llaman desde otros objetos (mensajes) o proporcionar el comportamiento interno fundamental de la clase. Los comportamientos internos son

métodos privados a los que no pueden acceder otros objetos. En la `Person`clase, los comportamientos son `getName()`, `setName()`, `getAddress()`, y `setAddress()`. Estos métodos permiten que otros objetos inspeccionen y cambien los valores de los atributos del objeto. Esta es una técnica común en los sistemas OO. En todos los casos, el acceso a los atributos dentro de un objeto debe ser controlado por el propio objeto; ningún otro objeto debe cambiar directamente un atributo de otro.

Mensajes

Los mensajes son el mecanismo de comunicación entre objetos. Por ejemplo, cuando el Objeto A invoca un método del Objeto B, el Objeto A envía un mensaje al Objeto B. La respuesta del Objeto B se define por su valor de retorno. Solo los métodos públicos, no los métodos privados, de un objeto pueden ser invocados por otro objeto. El siguiente código ilustra este concepto:

```
public class Payroll{

    String name;
    Person p = new Person();
    p.setName("Joe");

        ... code

    name = p.getName();
}
```

En este ejemplo (asumiendo que `Payroll`se crea una instancia de un objeto), el objeto **`Payroll`** está enviando un mensaje a un objeto **`Person`**, con el propósito de recuperar el nombre a través del método **`getName()`**. Nuevamente, no se preocupe demasiado por el código real, porque estamos realmente interesados en los conceptos. Abordamos el código en detalle a medida que avanzamos en el libro.

USAR DIAGRAMAS DE CLASES COMO HERRAMIENTA VISUAL

A lo largo de los años, se han desarrollado muchas herramientas y metodologías de modelado para ayudar en el diseño de sistemas de software. Desde el principio, he usado diagramas de clases UML para ayudar en el proceso educativo. Aunque está más allá del alcance de este libro describir UML en detalle, usaremos diagramas de clases UML para ilustrar las clases que construimos. De hecho, ya hemos utilizado diagramas de clases en este capítulo. La figura 1.11 muestra el `Person`diagrama de clases que discutimos anteriormente en este capítulo.

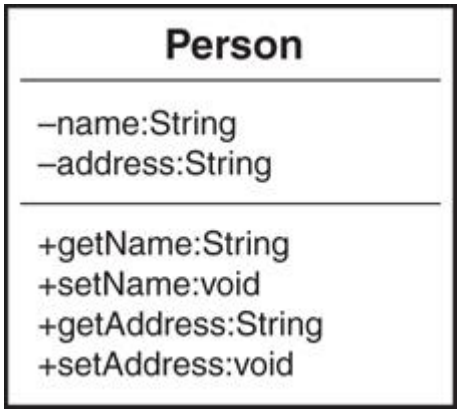

Figura 1.11 El diagrama de clases `de Person`.

Como vimos anteriormente, observe que los atributos y los métodos están separados (los atributos en la parte superior y los métodos en la parte inferior). A medida que profundicemos en el diseño de OO, estos diagramas de clases se volverán mucho más sofisticados y transmitirán mucha más información sobre cómo las diferentes clases interactúan entre sí.

ENCAPSULACIÓN Y OCULTACIÓN DE DATOS

Una de las principales ventajas de usar objetos es que el objeto no necesita revelar todos sus atributos y comportamientos. En un buen diseño de OO (al menos lo que generalmente se acepta como bueno), un objeto debe revelar solo las interfaces que otros objetos deben tener para interactuar con él. Los detalles que no sean pertinentes para el uso del objeto deben ocultarse de todos los demás objetos, básicamente una base de "necesidad de conocer".

La encapsulación se define por el hecho de que los objetos contienen tanto los atributos como los comportamientos. La ocultación de datos es una parte importante de la encapsulación.

Por ejemplo, un objeto que calcula el cuadrado de un número debe proporcionar una interfaz para obtener el resultado. Sin embargo, los atributos y algoritmos internos utilizados para calcular el cuadrado no necesitan estar disponibles para el objeto solicitante. Las clases robustas están diseñadas teniendo en cuenta la encapsulación. En las siguientes secciones, cubrimos los conceptos de interfaz e implementación, que son la base del encapsulado.

Fuente: Matt Weisfeld, 2019, The Object-Oriented Thought Process, Ed. Addison-Wesley Professional