2.1. ¿Qué es un objeto?

Conceptualmente, un *objeto* es una cosa con la que se puede interactuar: se le puede enviar varios *mensajes* y éste reacciona ante ellos. Cómo *se comporte* depende del *estado* interno actual del objeto, el cual puede cambiar, por ejemplo, como parte de la reacción del objeto al recibir un mensaje. Importa con qué objeto se interactúa, y normalmente se dirige al mismo por su nombre; es decir, un objeto tiene una *identidad* que lo distingue del resto de objetos. De esta forma, un objeto es una cosa que tiene comportamiento, estado e identidad: esta caracterización se le debe a Grady Booch [7]. Consideremos estos aspectos con más detalle.

Cosa Es más interesante de lo que puede parecer a primera vista. Un objeto no es sólo una cosa en el sistema, sino que también es una representación de un concepto en el sistema. Por ejemplo, un objeto puede representar una cosa física como es un cliente, un barco, o un reloj. En realidad la primera vez que los objetos aparecieron de forma explícita fue en Simula, donde representaban objetos del mundo real, los objetos simulados. Sin embargo, si el propósito general de los sistemas de software es que se construyan utilizando objetos, un objeto no necesita representar una cosa física. Se volverá de nuevo a la cuestión de cómo identificar objetos en el Capítulo 5.

P: ¿Cuáles de estos elementos podrían ser objetos: martillo, esfera, proceso químico, amor, niebla, río, cólera, gato, tonalidad de gris, gato gris?

Estado El *estado* de un objeto lo constituyen todos los datos que encapsula en un momento determinado. Un objeto normalmente tiene un número de lo que se conoce como *atributos* (o *variables de instancia* o *miembros datos*) cada uno de los cuales tiene un valor. Algunos atributos pueden ser mutables, esto es, que sus valores pueden cambiar. Por ejemplo, un objeto que representa a un cliente podría tener un atributo `dirección` cuyo valor debe cambiar si el cliente se muda de casa. Otros atributos pueden ser inmutables o constantes; por ejemplo, nuestro objeto cliente podría tener un atributo como número único identificativo del cliente, que podría ser el mismo a lo largo de toda la vida del objeto. En la mayoría de los lenguajes orientados a objetos de hoy día el conjunto de atributos de un objeto no puede cambiar durante la vida del objeto, aunque los valores de esos atributos sí pueden hacerlo.

Comportamiento La manera como actúa y reacciona un objeto, en función de sus cambios de estado y el paso de mensajes. Un objeto *comprende* ciertos *mensajes*, lo que quiere decir que puede recibir mensajes y actuar sobre ellos. El conjunto de mensajes que entiende un objeto, al igual que el conjunto de atributos que tiene, normalmente está fijado. La manera en que un objeto reacciona ante un mensaje puede depender de los valores de sus atributos en un momento determinado: de esta manera, incluso cuando (como es normal) no se permite acceder directamente a los atributos, sus valores le pueden estar afectando indirectamente. Este es el sentido en que los valores de los atributos son el *estado* del objeto.

Identidad Es un concepto un poco más escurridizo. La idea es que los objetos no se definen por los valores de sus atributos en un momento determinado. Un objeto tiene una existencia continuada. Por ejemplo los valores de los atributos de este objeto podrían cambiar, quizá como respuesta a un mensaje, pero seguiría siendo el mismo objeto. A un objeto normalmente se le hace referencia por un *nombre* (el valor de una variable en un programa cliente, quizá `unCliente`) pero el nombre del objeto no es lo mismo que el objeto, porque un mismo objeto

puede tener varios nombres diferentes. (Esto sólo sorprende si se es un programador funcional puro; y hay algunos autores distinguidos que defienden que se puede hacer OO de una forma puramente funcional, pero no se tratará esta cuestión en el presente libro.)

2.1.1. Ejemplo

Piense en un objeto al que llamaremos `miReloj`, que comprende los mensajes `informeTiempo`, `iniciarTiempoA(07:43)` e `iniciarTiempoA(12:30)`, que de forma más general sería `iniciarTiempoA(nuevaHora)` para cualquier valor coherente de `nuevaHora`. El objeto hace público esto a través de una *interfaz* que indica que acepta mensajes de la forma `informeTiempo()` e `iniciarTiempoA(nuevaHora: Hora)` donde `Hora` es un tipo[2] cuyos elementos son los valores coherentes de `nuevaHora` antes mencionados.

¿Cómo se implementa esta funcionalidad? El mundo exterior no necesita saber nada (la información debería estar oculta) pero quizá tiene un atributo `tiempo`, cuyo valor lo devuelve el objeto como respuesta al mensaje `informeTiempo`, y que se reinicia a `nuevaHora` cuando el objeto recibe el mensaje `iniciarTiempoA (nuevaHora)`. O quizá pase estos mensajes a cualquier otro objeto que conoce, y ese otro objeto tiene que tratar dichos mensajes.

2.1.2. Mensajes

En este ejemplo, se ven algunas cosas sobre los mensajes. Un mensaje incluye una palabra clave llamada *selector*; en el ejemplo se han visto los selectores `informeTiempo` e `iniciarTiempoA`. Un mensaje puede, pero no es necesario, incluir uno o más argumentos, cuyos valores se le pasan al objeto tal y como se pasan los valores a una función en una llamada normal. En el ejemplo, 07:43 es un argumento que forma parte del mensaje `iniciarTiempoA(07:43)`. Normalmente, para un selector determinado hay un número «correcto» de argumentos que deben pasarse en un mensaje que empieza por ese selector; no se puede esperar que `miReloj` entienda `iniciarTiempoA` sin ningún argumento, o `iniciarTiempoA(4,12:30)`, etc. Los valores aceptables de los argumentos se establecen[3] en la interfaz del objeto.

P: ¿Cuál es la sintaxis para el envío de mensajes en el lenguaje que está utilizando?

Destacar que los valores (por ejemplo, los valores de atributos de un objeto, o los valores de los argumentos enviados como parte de un mensaje) no tienen que pertenecer a un tipo básico (caracteres, enteros, etc.; cuáles son los tipos básicos, dependen del lenguaje). Pueden incluso ser objetos.

P: ¿Cuáles son los tipos básicos en su lenguaje?

Una de las cosas que un objeto podría hacer como respuesta a un mensaje es enviar un mensaje a otro objeto. Para hacer esto es necesario que tenga alguna manera de saber un nom-

---

[2] O una clase: las clases son tratadas más adelante en el Subapartado 2.1.4.

[3] Indicando la clase (véase 2.1.4) o a veces tan sólo la interfaz, que debería tener el objeto pasado como argumento, o su tipo si perteneciese a un tipo básico como entero o lógico; estos detalles dependen del lenguaje.

bre para el objeto. Quizá, por ejemplo, tiene el objeto como uno de sus atributos, en cuyo caso puede enviar el mensaje utilizando el nombre del atributo.

P: Supóngase que O es un objeto. Aparte de cualquiera de los objetos que O puede tener como valor de sus atributos, ¿a qué objetos podría enviar O mensajes?

Destacar que cuando se envía un mensaje a un objeto, en general no se sabe el código que se va a ejecutar como respuesta, porque dicha información está encapsulada. Esto será importante al final de este capítulo cuando se trate la ligadura dinámica.

2.1.3. Interfaces

La interfaz *pública* de un objeto define qué mensajes se aceptan sin importar de dónde vienen. Normalmente la interfaz almacena los selectores de estos mensajes junto con alguna información sobre qué argumentos se requieren y si se devuelve algo o no. Tal y como se destacó en el Capítulo 1, probablemente se preferiría que la interfaz incluyese algún tipo de especificación sobre las consecuencias de enviar un mensaje a un objeto, pero estas especificaciones normalmente sólo vienen en comentarios y documentación acompañante. En la mayoría de los lenguajes puede haber atributos también en la interfaz pública del objeto; poner un atributo X en la interfaz equivale a declarar que ese objeto tiene un dato X, que el mundo exterior puede inspeccionar y alterar.

Pregunta de discusión 9

Si se utiliza un lenguaje que no admite atributos en la interfaz pública, ¿cómo se puede obtener los mismos efectos utilizando mensajes para acceder a los datos? ¿Hay alguna ventaja al utilizar esta forma incluso en un lenguaje que admite atributos en las interfaces? ¿Y desventajas?

A menudo, no es apropiado permitir que todo el sistema envíe cada uno de los mensajes que entiende un objeto a dicho objeto. Por lo que normalmente un objeto es capaz de entender algunos mensajes que no están en la interfaz pública. Por ejemplo, en el Capítulo 1 se explicó que las estructuras de datos utilizadas por un módulo deberían ser encapsuladas; en este caso se corresponde con la idea de que la interfaz pública de un objeto no debería incluir atributos.

Un objeto siempre puede enviarse *a sí mismo* un mensaje que pueda comprender. Puede parecer extraño o demasiado complicado pensar en un objeto enviándose mensajes a sí mismo: la razón para pensar esto se verá más clara cuando se explique la ligadura dinámica más tarde en el capítulo.

De manera que un objeto normalmente tiene al menos dos interfaces: la interfaz pública, que cualquier parte del sistema puede utilizar, y la interfaz privada que la pueden utilizar el propio objeto y algunas partes privilegiadas del sistema. A veces es útil otra interfaz intermedia que proporcione más prestaciones que la pública, pero menos que la privada. Hay veces que en un contexto determinado sólo se necesita una parte de la interfaz pública de un objeto; puede ser útil almacenar en una interfaz de menor tamaño que la pública del objeto,

las características que son realmente necesarias. Debido a esto, un objeto puede tener (o *realizar)* más de una interfaz. A la inversa, varios objetos diferentes pueden implementar la misma interfaz. Si no se especifica lo contrario, «interfaz» será interfaz pública.

2.1.4. Clases

Hasta aquí se ha hablado de objetos como si cada uno estuviese definido por separado, con su propia interfaz, su propia manera de controlar qué otros objetos podrían enviarle y qué mensajes. Por supuesto, ésta no es la manera más coherente de construir un sistema típico, porque los objetos tienen mucho en común unos con otros. Si una empresa tiene 10 000 clientes y se quiere construir un sistema con un objeto que represente a cada una de estas personas, ¡los objetos que representan a los clientes tienen mucho en común! Se querrá que se comporten de manera consistente para que los desarrolladores y los encargados de mantenimiento del sistema puedan entenderlo. Se tendrá una *clase* de objetos que representen a los clientes. Una clase describe un conjunto de objetos que tienen un rol o roles equivalentes en el sistema.

En los lenguajes orientados a objetos basados en clases, cada objeto pertenece a una clase, y la clase de un objeto es quien determina su interfaz[4]. Por ejemplo, a lo mejor `miReloj` pertenece a una clase `Reloj` cuya interfaz pública especifica que cada objeto de la clase `Reloj` proporcionará la operación `iniciarTiempoA(nuevaHora: Hora)`; esto es, entenderá mensajes que tengan el selector `iniciarTiempoA` y un único argumento que es un objeto de la clase (o tipo) `Hora`.

En realidad, la clase del objeto, junto con los valores de los atributos del objeto, es quien determina incluso la manera en que reacciona un objeto. Un *método* es una parte de código específica que implementa la operación. Sólo es visible en la interfaz del objeto el hecho de que dicho objeto proporciona la operación; el método que implementa la prestación se encuentra oculto.

De forma similar el conjunto de atributos que tiene un objeto viene determinado por su clase, aunque, por supuesto, los valores que toman dichos atributos no están determinados por la clase, pudiendo variar. Por ejemplo, a lo mejor los objetos que pertenecen a la clase `Reloj` tienen un atributo privado llamado `hora`. Si se envía el mismo mensaje a dos objetos que pertenecen a una misma clase, se ejecuta el mismo método en ambos casos, aunque el efecto de la ejecución del método puede ser muy diferente si los objetos tienen distintos valores en sus atributos.

En resumen, los objetos de una misma clase tienen las mismas interfaces. Se podría describir las interfaces pública y privada para la clase `Reloj` de la siguiente manera:

```
- hora : Hora.
+ informeTiempo() : Hora.
+ iniciarTiempoA(nuevaHora: Hora).
```

[4] Realmente un objeto puede pertenecer a más de una clase, y la clase *más específica* a la que pertenece es la que determina su interfaz.

La interfaz pública está formada por las líneas marcadas con +; la interfaz privada incluye las líneas marcadas con –.

P: En su lenguaje, ¿cómo define el programador una clase como `Reloj`? ¿Se definen por separado la interfaz y la implementación? ¿Cómo se llaman las interfaces pública y privada? (Probablemente `public` y `private`) ¿Hay otras posibilidades? ¿Qué significan dichas posibilidades?

Al proceso de creación de un nuevo objeto que pertenece a una clase `Reloj` se le llama *instanciación* de la clase `Reloj`, y al objeto resultante se le denomina *instancia* de la clase `Reloj`; por esto, por supuesto, es por lo que las variables cuyos valores pertenecen al objeto y pueden variar a lo largo de su tiempo de vida se denominan variables instancia. La creación de un objeto consiste en la creación de un nuevo elemento con su propio estado y su propia identidad, que se comportará de forma consistente con todos los demás objetos de la clase. El proceso de creación normalmente incluye el establecimiento de los valores de los atributos; los valores se pueden especificar en la petición de creación, o los puede especificar la clase del objeto como valores iniciales por defecto. En lenguajes tales como C++ y Java, la clase tiene un rol directo en la creación de nuevos objetos, realmente hace todo el trabajo, en vez de actuar como una plantilla. Esto es, una clase puede verse con una *factoría de objetos*.

P: En su lenguaje, ¿cómo se crea un nuevo objeto de una clase determinada?

La mayoría de los lenguajes permiten que las clases se comporten como si fuesen objetos por derecho, conteniendo atributos y entendiendo mensajes por sí mismos. Por ejemplo, la *clase* `Cliente` podría tener un atributo `númeroDeInstancias` que podría ser un entero que se incrementa cada vez que se crea un nuevo *objeto* `Cliente`. Smalltalk tiene esta visión particularmente consistente, pero C++ y Java también tienen indicios de ella. En Smalltalk, por ejemplo, se crearía un nuevo objeto `Cliente` mediante el envío de un mensaje (por ejemplo `new`) a la *clase* `Cliente`. La reacción de una clase sería crear un nuevo objeto y entregárselo al llamador.

Pregunta de discusión 10

Considere qué pasaría si se trata de aplicar esta idea de forma consistente. Se ha dicho que cada objeto tiene una clase; así que cuando se considera una clase `C` como un objeto, ¿cuál es la clase de `C`? ¿Y la clase de la clase de `C`?

P: ¿Tiene su lenguaje características como éstas? ¿Exactamente cuáles?

Digresión: ¿por qué tener clases?

¿Por qué no tener sólo objetos, que tienen estado, comportamiento e identidad que es lo que se pide?

Las clases en los lenguajes orientados a objetos sirven para dos propósitos. Primero, son una manera cómoda de describir una colección (una clase) de objetos que tienen las mismas propiedades. Los lenguajes de programación orientados a objetos pueden utilizar las clases

esencialmente como una ventaja. Por ejemplo, no es necesario almacenar una copia del código que representa el comportamiento de un objeto en cada objeto, incluso aunque se piense que cada objeto encapsule su propio código. En vez de esto, los desarrolladores escriben la definición de una clase una sola vez, y los compiladores crean una única representación de una clase, y permite a los objetos de esa clase acceder a la representación de clase para obtener el código que necesitan.

Recuérdese que la comodidad no es una trivialidad. Aquí, el principal objetivo es crear sistemas más fáciles de comprender, de manera que el desarrollo y el mantenimiento sea lo más sencillo, barato y fiable posible. Hacer que los objetos parecidos compartan el mismo diseño e implementación es un paso muy importante para conseguir esto.

Este es un caso particular de un principio[5] que es tan importante en la ingeniería del software, como en el código, y es:

> ¡ESCRIBA UNA SOLA VEZ!

Quiere decir que si dos copias de un mismo artefacto de ingeniería del software (parte de cualquier tipo de código, parte de un diagrama, parte de texto de un documento) tienen que ser siempre consistentes, entonces no debería *haber* dos copias. Cada vez que se almacena la misma información en más de un lugar, no sólo se gasta esfuerzo en la repetición, que podría utilizarse mejor en hacer algo nuevo, sino que además se crean problemas de mantenimiento, ya que las dos versiones no se mantendrán sincronizadas sin esfuerzo. La capacidad de aplicar el principio está en comprobar si se tiene o no dos copias de la misma información, o dos partes de información potencialmente diferentes que resultan, de momento, ser la misma.

En el último caso, lo apropiado es copiar y pegar. En el primer caso, la actitud de resentimiento por duplicar la información en más de un lugar es en realidad una ventaja: sólo se debe aceptar hacer esto por una buena razón.

Pregunta de discusión 11

¿Este principio quiere decir que es una mala idea pensar en la duplicación de datos en una base de datos distribuida? ¿Por qué?

Se puede ver que hay otras maneras de crear arbitrariamente muchos objetos similares mientras se escribe su código una sola vez; por ejemplo, se podría definir un objeto prototipo, y después definir otros objetos que son «como ese otro, pero con estas diferencias». Otra vez, se necesitaría mantener una sola copia de código. Esto es aproximadamente lo que hace el lenguaje basado en prototipos Self. Sin embargo, los lenguajes basados en clases dominan hoy por hoy la orientación a objetos, y probablemente, continuarán haciéndolo.

[5] Enfatizado, en particular, por Kent Beck.

En segundo lugar, en los lenguajes OO más modernos, se utilizan las clases de la misma manera en que se utilizan los tipos en otros muchos lenguajes: para especificar qué valores se aceptan en un contexto determinado, y así permitir al compilador, y, con la misma importancia, al lector, comprender la intención del programador y comprobar que no se den en el programa ciertos tipos de errores.

A menudo la gente cree que las clases y los tipos son lo mismo (es cómodo de veras, y a menudo no es erróneo, hacerlo). Sin embargo, está mal. Recuérdese que una clase no sólo define qué mensajes entiende un objeto (que realmente es todo lo que necesitarías saber para comprobar si es aceptable en algún contexto), sino que también define lo que hace el objeto como respuesta a un mensaje.