# 3

# Clases y objetos

Tanto el ingeniero como el artista deben estar íntimamente familiarizados con los materiales que manejan. Cuando se usan métodos orientados a objetos para analizar o diseñar un sistema de software complejo, los bloques básicos de construcción son las clases y los objetos. Puesto que hasta ahora se han proporcionado solo definiciones informales de estos dos elementos, este capítulo se vuelca en un estudio detallado de la naturaleza de las clases, los objetos y sus relaciones, y a lo largo del camino se ofrecen varias reglas para construir abstracciones y mecanismos de calidad.

## 3.1 La naturaleza de los objetos

### Qué es y qué no es un objeto

La capacidad de reconocer objetos físicos es una habilidad que los humanos aprenden en edades muy tempranas. Una pelota de colores llamativos atraerá la atención de un niño, pero casi siempre, si se esconde la pelota, el niño no intentará buscarla; cuando el objeto abandona su campo de visión, hasta donde él puede determinar, la pelota ha dejado de existir. Normalmente, hasta la edad de un año un niño no desarrolla lo que se denomina el *concepto de objeto*, una habilidad de importancia crítica para el desarrollo cognitivo futuro. Muéstrese una pelota a un niño de un año y escóndase a continuación, y normalmente la buscará incluso si no está visible. A través del concepto de objeto, un niño llega a darse cuenta de que los objetos tienen una permanencia e identidad además de cualesquiera operaciones sobre ellos [1].

En el capítulo anterior se definió informalmente un objeto como una entidad tangible que exhibe algún comportamiento bien definido. Desde la perspectiva de la cognición humana, un objeto es cualquiera de las siguientes cosas:

- Una cosa tangible y/o visible.
- Algo que puede comprenderse intelectualmente.
- Algo hacia lo que se dirige un pensamiento o acción.

Se añade a la definición informal la idea de que un objeto modela alguna parte de la realidad y es, por tanto, algo que existe en el tiempo y el espacio. En software, el término *objeto* se aplicó formalmente en primer lugar en el lenguaje Simula; los objetos existían en los programas en Simula típicamente para simular algún aspecto de la realidad [2].

Los objetos del mundo real no son el único tipo de objeto de interés en el desarrollo del software. Otros tipos importantes de objetos son invenciones del proceso de diseño cuyas colaboraciones con otros objetos semejantes sirven como mecanismos para desempeñar algún comportamiento de nivel superior [3]. Esto nos lleva a la definición más refinada de Smith y Tockey, que sugieren que "un objeto representa un elemento, unidad o entidad individual e identificable, ya sea real o abstracta, con un papel bien definido en el dominio del problema" [4]. En términos aún más generales, se define un objeto como cualquier cosa que tenga una frontera definida con nitidez [5].

Considérese por un momento una planta de fabricación que procesa materiales compuestos para fabricar elementos tan diversos como cuadros de bicicleta y alas de aeroplano. Las fábricas se dividen con frecuencia en talleres separados: mecánico, químico, eléctrico, etc. Los talleres se dividen además en células, y en cada célula hay alguna colección de máquinas, como troqueladoras, prensas y tornos. A lo largo de una línea de fabricación, se podría encontrar tanques con materias primas, que se utilizan en un proceso químico para producir bloques de materiales compuestos, y a los que a su vez se da forma para producir elementos finales como cuadros para bicicletas y alas de aeroplano. Cada una de las cosas tangibles que se han mencionado hasta aquí es un objeto. Un torno tiene una frontera perfectamente nítida que lo separa del bloque de material compuesto sobre el que opera; un cuadro de bicicleta tiene una frontera perfectamente nítida que lo distingue de la célula o las máquinas que lo producen.

Algunos objetos pueden tener límites conceptuales precisos, pero aun así pueden representar eventos o procesos intangibles. Por ejemplo, un proceso químico en una fábrica puede tratarse como un objeto, porque tiene una frontera conceptual clara, interactúa con otros determinados objetos a lo largo de una colección bien ordenada de operaciones que se despliegan en el tiempo, y exhibe un comportamiento bien definido. Análogamente, considérese un sistema CAD/CAM para modelar sólidos. Donde se intersectan (intersecan) dos sólidos como una esfera y un cubo, pueden formar una línea irregular de intersección. Aunque no existe fuera de la esfera o el cubo, esta línea sigue siendo un objeto con fronteras conceptuales muy precisas.

Algunos objetos pueden ser tangibles y aun así tener fronteras físicas difusas. Objetos como los ríos, la niebla o las multitudes humanas encajan en esta definición.[1] Exactamente igual que la persona que sostiene un martillo tiende a verlo todo a su alrededor como un clavo, el desarrollador con una mentalidad orientada a objetos comienza a pensar que todo lo que hay en el mundo son objetos. Esta perspectiva es un poco ingenua, porque existen algunas cosas que claramente no son objetos. Por ejemplo, atributos como el tiempo, la belleza o el color no son objetos, ni

[1] Esto es cierto solo a un nivel de abstracción lo bastante alto. Para una persona que camina a través de un banco de niebla, es generalmente inútil distinguir "mi niebla" de "tu niebla". Sin embargo, considérese un mapa meteorológico: un banco de niebla sobre San Francisco es un objeto claramente distinto de un banco de niebla sobre Londres.

las emociones como el amor o la ira. Por otro lado, todas estas cosas son potencialmente propiedades de otros objetos. Por ejemplo, se podría decir que un hombre (un objeto) ama a su mujer (otro objeto), o que cierto gato (un objeto más) es gris.

Así, es útil decir que un objeto es algo que tiene fronteras nítidamente definidas, pero esto no es suficiente para servir de guía al distinguir un objeto de otro, ni permite juzgar la calidad de las abstracciones. Por tanto, nuestra experiencia sugiere la siguiente definición:

*Un objeto tiene estado, comportamiento e identidad; la estructura y comportamiento de objetos similares están definidos en su clase común; los términos instancia y objeto son intercambiables.*

## Estado

**Semántica.** Considérese una máquina expendedora de refrescos. El comportamiento usual de esos objetos es tal que cuando se introducen monedas en una ranura y se pulsa un botón para realizar una selección, surge una bebida de la máquina. ¿Qué pasa si un usuario realiza primero la selección y a continuación introduce dinero en la ranura? La mayoría de las máquinas expendedoras se inhiben y no hacen nada, porque el usuario ha violado las suposiciones básicas de su funcionamiento. Dicho de otro modo, la máquina expendedora estaba haciendo un papel (o esperando monedas) que el usuario ignoró (haciendo primero la selección). Análogamente, supóngase que el usuario ignora la luz de advertencia que dice "Solo el dinero exacto" e introduce dinero extra. La mayoría de las máquinas son "hostiles al usuario"; se tragarán felices las monedas sobrantes.

En todas esas circunstancias se ve cómo el comportamiento de un objeto está influenciado por su historia: el orden en el que se opera sobre el objeto es importante. La razón para este comportamiento dependiente del tiempo y los eventos es la existencia de un estado en el interior del objeto. Por ejemplo, un estado esencial asociado con la máquina expendedora es la cantidad de dinero que acaba de introducir un usuario pero que aun no se ha aplicado a una selección. Otras propiedades importantes incluyen la cantidad de cambio disponible y la cantidad de refrescos que tiene.

De este ejemplo se puede extraer la siguiente definición de bajo nivel:

*El estado de un objeto abarca todas las propiedades (normalmente estáticas) del mismo más los valores actuales (normalmente dinámicos) de cada una de esas propiedades.*

Otra propiedad de una máquina expendedora es que puede aceptar monedas. Esta es una propiedad estática (es decir, fija), lo que significa que es una característica esencial de una máquina expendedora. En contraste, la cantidad actual de monedas que ha aceptado en un momento dado representa el valor dinámico de esta propiedad, y se ve afectado por el orden de las operaciones que se efectúan sobre la máquina. Esta cantidad aumenta cuando el usuario introduce monedas, y disminuye cuando un empleado atiende la máquina. Se dice que los valores son "normalmente

dinámicos" porque en algunos casos los valores son estáticos. Por ejemplo, el número de serie de una máquina expendedora es una propiedad estática, y el valor es estático.

Una propiedad es una característica inherente o distintiva, un rasgo o cualidad que contribuye a hacer que un objeto sea ese objeto y no otro. Por ejemplo, una propiedad esencial de un ascensor es que está restringido a moverse arriba y abajo y no horizontalmente. Las propiedades suelen ser estáticas, porque atributos como estos son inmutables y fundamentales para la naturaleza del objeto. Se dice "suelen ser" porque en algunas circunstancias las propiedades de un objeto pueden cambiar. Por ejemplo, considérese un robot autónomo que puede aprender sobre su entorno. Puede reconocer primero un objeto que parece ser una barrera inamovible, para aprender después que este objeto es de hecho una puerta que puede abrirse. En este caso, el objeto creado por el robot a medida que este construye su modelo conceptual del mundo gana nuevas propiedades a medida que adquiere nuevo conocimiento.

Todas las propiedades tienen algún valor. Este valor puede ser una mera cantidad, o puede denotar a otro objeto. Por ejemplo, parte del estado de un ascensor puede tener el valor 3, que denota el piso actual en el que está. En el caso de la máquina expendedora, el estado de la misma abarca muchos otros objetos, tales como una colección de refrescos. Los refrescos individuales son de hecho objetos distintos; sus propiedades son diferentes de las de la máquina (pueden consumirse, mientras que una expendedora no puede), y puede operarse sobre ellos de formas claramente diferentes. Así, se distingue entre objetos y valores simples: las cantidades simples como el número 3 son "intemporales, inmutables y no instanciadas", mientras que los objetos "existen en el tiempo, son modificables, tienen estado, son instanciados y pueden crearse, destruirse y compartirse" [6].

El hecho de que todo objeto tenga un estado implica que todo objeto toma cierta cantidad de espacio, ya sea en el mundo físico o en la memoria del computador.

**Ejemplo:** considérese la estructura de un registro de personal. En C++ podría escribirse:

```
struct RegistroPersonal
{
    char nombre[100];
    int numeroSeguridadSocial;
    char departamento[10];
    float salario;
};
```

Cada parte de esta estructura denota una propiedad particular de la abstracción de un registro de personal. Esta declaración denota una clase, no un objeto, porque no representa una instancia específica.[2] Para declarar objetos de esta clase, se escribe:

[2] Para ser precisos, esta declaración denota una estructura, una construcción de registro de C++ de nivel inferior cuya semántica es la misma que la de una clase con todos sus miembros public. Las estructuras denotan así una abstracción no encapsulada.

```
RegistroPersonal deb, dave, karen, jim, tom, denise, kaitlyn, krista, elyse;
```

Aquí hay nueve objetos distintos, cada uno de los cuales ocupa una cierta cantidad de espacio en memoria. Ninguno de estos objetos comparte su espacio con ningún otro objeto, aunque todos ellos tienen las mismas propiedades; por tanto, sus estados tienen una representación común.

Es una buena práctica de ingeniería el encapsular el estado de un objeto en vez de exponerlo como en la declaración precedente. Por ejemplo, se podría reescribir la declaración de clase como sigue:

```
class RegistroPersonal {
public:

    char *nombreEmpleado() const;
    int numeroSeguridadSocialEmpleado() const;
    char *departamentoEmpleado( ) const;

protected:

    char nombre[100];
    int numeroSeguridadSocial;
    char departamento[10];
    float salario;
};
```

Esta declaración es ligeramente más complicada que la anterior, pero es mucho mejor por varias razones.[3] Específicamente, se ha escrito esta clase de forma que su representación está oculta para todos los clientes externos. Si se cambia su representación, habrá que recompilar algún código, pero semánticamente ningún cliente externo resultará afectado por este cambio (en otras palabras, el código ya existente no se estropeará). Además, se han capturado algunas decisiones acerca del espacio del problema, estableciendo explícitamente algunas de las operaciones que los clientes pueden realizar sobre objetos de esta clase. En particular, se garantiza a todos los clientes el derecho a recuperar el nombre, número de la seguridad social y departamento de un empleado. Solo clientes especiales (es decir, subclases de esta clase) tienen permiso para modificar los valores de estas propiedades. Más aun, solo estos clientes especiales pueden modificar o recuperar el salario de un empleado, mientras que los clientes externos no pueden.

[3] Un problema de estilo: la clase `RegistroPersonal` tal como se ha declarado aquí no es una clase de altísima calidad, de acuerdo con las métricas que se describen más adelante en este capítulo (este ejemplo solo sirve para ilustrar la semántica del estado de una clase). Tener una función miembro que devuelve un valor de tipo `char*` es a menudo peligroso, porque esto viola uno de los principios de seguridad de la memoria: si el método crea espacio de almacenamiento del que el cliente no se hace responsable, la recolección de memoria (basura) implicará dejar referencias (punteros) colgadas. En sistemas de producción, es preferible usar una clase parametrizada de cadenas de longitud variable. Además, las clases son más simples que la `struct` de C envueltas en sintaxis de C++; como se explica en el capítulo 4, la clasificación requiere una atención deliberada hacia estructura y comportamiento comunes.

Otra razón por la que esta declaración es mejor tiene que ver con la reutilización. Como se verá en una sección posterior, la herencia hace posible la reutilización de esta abstracción, y refinarla o especializarla de diversas formas.

Puede decirse que todos los objetos de un sistema encapsulan algún estado, y que todo el estado de un sistema está encapsulado en objetos. Sin embargo, encapsular el estado de un objeto es un punto de partida, pero no es suficiente para permitir que se capturen todos los designios de las abstracciones que se descubren e inventan durante el desarrollo. Por esta razón, hay que considerar también cómo se comportan los objetos.

## Comportamiento

**El significado del comportamiento.** Ningún objeto existe de forma aislada. En vez de eso, los objetos reciben acciones, y ellos mismos actúan sobre otros objetos. Así, puede decirse que

> *El comportamiento es cómo actúa y reacciona un objeto, en términos de sus cambios de estado y paso de mensajes.*

En otras palabras, el comportamiento de un objeto representa su actividad visible y comprobable exteriormente.

Una operación es una acción que un objeto efectúa sobre otro con el fin de provocar una reacción. Por ejemplo, un cliente podría invocar las operaciones `anadir` y `extraer` para aumentar y disminuir un objeto `cola`, respectivamente. Un cliente podría también invocar la operación `longitud`, que devuelve un valor denotando el tamaño del objeto `cola` pero no altera el estado de la misma. En lenguajes orientados a objetos puros como Smalltalk, se habla de que un objeto pasa un mensaje a otro. En lenguajes como C++, que deriva de antecesores más procedimentales, se habla de que un objeto invoca una función miembro de otro. Generalmente, un mensaje es simplemente una operación que un objeto realiza sobre otro, aunque los mecanismos subyacentes para atenderla son distintos. Para nuestros propósitos, los términos *operación* y *mensaje* son intercambiables.

En la mayoría de los lenguajes de programación orientados a objetos, las operaciones que los clientes pueden realizar sobre un objeto suelen declararse como *métodos*, que forman parte de la declaración de la clase. C++ usa el término *función miembro* para denotar el mismo concepto; se utilizarán ambos vocablos de forma equivalente.

El paso de mensajes es una de las partes de la ecuación que define el comportamiento de un objeto; la definición de "comportamiento" también recoge que el estado de un objeto afecta asimismo a su comportamiento. Considérese de nuevo el ejemplo de la máquina expendedora. Se puede invocar alguna operación para hacer una selección, pero la expendedora se comportará de modo diferente según su estado. Si no se deposita suficiente dinero para nuestra selección, posiblemente la máquina no hará nada. Si se introduce suficiente dinero, la máquina tomará el dinero y suministrará lo que se ha seleccionado (alterando con ello su estado). Así, se puede decir

que el comportamiento de un objeto es función de su estado así como de la operación que se realiza sobre él, teniendo algunas operaciones el efecto lateral de modificar el estado del objeto. Este concepto de efecto lateral conduce así a refinar la definición de estado:

*El estado de un objeto representa los resultados acumulados de su comportamiento.*

Los objetos más interesantes no tienen un estado estático; antes bien, su estado tiene propiedades cuyos valores se modifican y consultan según se actúa sobre el objeto.

**Ejemplo:** considérese la siguiente declaración de una clase `cola` en C++:

```
class Cola {
public:

    Cola();
    Cola(const Cola&);
    virtual ~Cola();

    virtual Cola& operator=(const Cola&);
    virtual int operator=(const Cola&) const;
    int operator!=(const Cola&) const;

    virtual void borrar();
    virtual void anadir(const void*);
    virtual void extraer();
    virtual void eliminar(int donde);

    virtual int longitud() const;
    virtual int estaVacia() const;
    virtual const void *cabecera() const;
    virtual int posicion(const void *);

protected:
    ...
};
```

Esta clase utiliza el modismo habitual de C por el que se fijan y recuperan valores mediante `void*`, que proporciona la abstracción de una cola heterogénea, lo que significa que los clientes pueden añadir objetos de cualquier clase a un objeto `cola`. Este enfoque no es demasiado seguro respecto a los tipos, porque el cliente debe recordar la clase de los objetos que hay en la cola. Además, el uso de `void*` evita que el objeto `Cola` "posea" sus elementos, lo que quiere decir que no se puede confiar en la actuación del destructor de la cola (`~Cola()`) para destruir los elementos de la misma. En una próxima sección se estudiarán los tipos parametrizados, que mitigan estos problemas.

Ya que la declaración `Cola` representa una clase, no un objeto, hay que declarar instancias que los clientes puedan manipular:

```
Cola a, b, c, d;
```

A continuación, se puede operar sobre estos objetos como en el código que sigue:

```
a.anadir(&deb);
a.anadir(&karen);
a.anadir(&denise);
b = a;
a.extraer();
```

Después de ejecutar estas sentencias, la cola denotada por `a` contiene dos elementos (con un puntero al registro `karen` en su cabecera), y la cola denotada por `b` contiene tres elementos (con el registro `deb` en su cabecera). De este modo, cada uno de estos objetos de cola contiene algún estado distinto, y este estado afecta al comportamiento futuro de cada objeto. Por ejemplo, se puede extraer elementos de `b` de forma segura tres veces más, pero sobre `a` solo puede efectuarse esta operación de forma segura otras dos veces.

**Operaciones.** Una operación denota un servicio que una clase ofrece a sus clientes. En la práctica, se ha visto que un cliente realiza típicamente cinco tipos de operaciones sobre un objeto.[4] Los tres tipos más comunes de operaciones son los siguientes:

- Modificador — Una operación que altera el estado de un objeto.
- Selector — Una operación que accede al estado de un objeto, pero no altera ese estado.
- Iterador — Una operación que permite acceder a todas las partes de un objeto en algún orden perfectamente establecido.

Puesto que estas operaciones son tan distintas lógicamente, se ha hallado útil aplicar un estilo de codificación que subraya sus diferencias. Por ejemplo, en la declaración de la clase `cola`, se declaran primero todos los modificadores como funciones miembro no–Const (las operaciones `borrar`, `anadir`, `extraer` y `eliminar`), seguidas por todos los selectores como funciones const (las operaciones `longitud`, `estaVacia`, `cabecera` y `posicion`). El estilo que se propone es definir una clase separada que actúa como el agente responsable de iterar a lo largo de las colas.

[4] Lippman sugiere una categorización ligeramente diferente: funciones de manejo, funciones de implantación, funciones de asistencia (todo tipo de modificadores), y funciones de acceso (equivalentes a selectores) [7].

Hay otros dos tipos de operaciones habituales; representan la infraestructura necesaria para crear y destruir instancias de una clase:

- Constructor — Una operación que crea un objeto y/o inicializa su estado.
- Destructor — Una operación que libera el estado de un objeto y/o destruye el propio objeto.

En C++, los constructores y destructores se declaran como parte de las definiciones de una clase (los miembros `Cola` y `~Cola`), mientras que en Smalltalk y CLOS tales operaciones suelen ser parte del protocolo de una metaclase (es decir, la clase de una clase).

En lenguajes orientados a objetos puros como Smalltalk, solo pueden declararse las operaciones como métodos, ya que el lenguaje no permite declarar procedimientos o funciones separados de ninguna clase. Por contra, los lenguajes como Object Pascal, C++, CLOS y Ada permiten al desarrollador escribir operaciones como subprogramas libres; en C++, se les llama funciones no miembro. Los *subprogramas libres* son procedimientos o funciones que sirven como operaciones no primitivas sobre un objeto u objetos de la misma o de distintas clases. Los subprogramas libres están típicamente agrupados según las clases sobre las que se los ha construido; por tanto, se llama a tales colecciones de subprogramas libres *utilidades de clase*. Por ejemplo, dada la declaración precedente del paquete `cola`, se podría escribir la siguiente función no miembro:

```
void copiarHastaEncontrado(Cola& fuente, Cola& destino, void* elemento)
{
    while ((!fuente.estaVacia()) &&
           (fuente.cabecera() != elemento)) {
        destino.anadir(fuente.cabecera());
        fuente.extraer();
    }
}
```

El propósito de esta operación es copiar repetidamente y entonces extraer el contenido de una cola hasta que se encuentra el elemento dado en la cabecera de la misma. Esta operación no es primitiva; puede construirse a partir de operaciones de nivel inferior que ya son parte de la clase `Cola`.

Forma parte del estilo habitual de C++ (y de Smalltalk) recoger todos los subprogramas libres relacionados lógicamente y declararlos como parte de una clase que no tiene estado. En particular, en C++ esto se hace static.

Así, puede decirse que todos los métodos son operaciones, pero no todas las operaciones son métodos: algunas operaciones pueden expresarse como subprogramas libres. En la práctica, es preferible declarar la mayoría de las operaciones como métodos, si bien, como se observa en una sección posterior, hay a veces razones de peso para obrar de otro modo, como cuando una operación concreta afecta a dos o más objetos de clases diferentes, y no se deriva ningún beneficio especial al declarar esa operación en una clase y no en la otra.

**Papeles (roles) y responsabilidades.** Colectivamente, todos los métodos y subprogramas libres asociados con un objeto concreto forman su *protocolo*. El protocolo define así la envoltura del comportamiento admisible en un objeto, y por tanto engloba la visión estática y dinámica completa del mismo. Para la mayoría de las abstracciones no triviales, es útil dividir este protocolo mayor en grupos lógicos de comportamiento. Estas colecciones, que constituyen una partición del espacio de comportamiento de un objeto, denotan los *papeles* que un objeto puede desempeñar. Como sugiere Adams, un papel es una máscara que se pone un objeto [8] y por tanto define un contrato entre una abstracción y sus clientes.

Unificando nuestras definiciones de estado y comportamiento, Wirfs-Brock define las *responsabilidades* de un objeto de forma que "incluyen dos elementos clave: el conocimiento que un objeto mantiene y las acciones que puede llevar a cabo. Las responsabilidades están encaminadas a transmitir un sentido del propósito de un objeto y de su lugar en el sistema. Las responsabilidades de un objeto son todos los servicios que proporciona para todos los contratos que soporta" [9]. En otras palabras, se puede decir que el estado y comportamiento de un objeto definen en conjunto los papeles que puede representar un objeto en el mundo, los cuales a su vez cumplen las responsabilidades de la abstracción.

Realmente, los objetos más interesantes pueden representar muchos papeles diferentes durante su tiempo de vida; por ejemplo [10]:

- Una cuenta bancaria puede estar en buena o mala situación, y el papel en el que esté afecta a la semántica de un reintegro.
- Para un comerciante, una acción de bolsa representa una entidad con cierto valor que puede comprarse o venderse; para un abogado, la misma acción denota un instrumento legal que abarca ciertos derechos.
- En el curso de un día, la misma persona puede representar el papel de madre, doctor, jardinero y crítico de cine.

En el caso de la cuenta bancaria, los papeles que este objeto puede desempeñar son dinámicos, pero exclusivos mutuamente: puede estar saneada o en números rojos, pero no ambas cosas a la vez. En el caso de las acciones, sus papeles se superponen ligeramente, pero cada papel es estático en relación con el cliente que interacciona con ellas. En el caso de la persona, sus papeles son bastante dinámicos, y pueden cambiar de un momento a otro.

Con frecuencia, se inicia el análisis de un problema examinando los distintos papeles que puede desempeñar un objeto. Durante el diseño, se refinan estos papeles descubriendo las operaciones particulares que llevan a cabo las responsabilidades de cada papel.

**Los objetos como máquinas.** La existencia de un estado en el seno de un objeto significa que el orden en el que se invocan las operaciones es importante. Esto da lugar a la idea de que cada objeto es como una pequeña máquina independiente [11]. Realmente, para algunos objetos, esta ordenación de las operaciones respecto a los eventos y al tiempo es tan penetrante que se puede

caracterizar mejor formalmente el comportamiento de tales objetos en términos de una máquina de estados finitos equivalente.

Siguiendo con la metáfora de las máquinas, se pueden clasificar los objetos como activos o pasivos. Un *objeto activo* es aquel que comprende su propio hilo de control, mientras que un *objeto pasivo* no. Los objetos activos suelen ser autónomos, lo que quiere decir que pueden exhibir algún comportamiento sin que ningún otro objeto opere sobre ellos. Los objetos pasivos, por otra parte, solo pueden padecer un cambio de estado cuando se actúa explícitamente sobre ellos. De este modo, los objetos activos del sistema sirven como las raíces del control. Si el sistema comprende múltiples hilos de control, habrá generalmente múltiples objetos activos. Los sistemas secuenciales, por contra, suelen tener exactamente un objeto activo, tal como un objeto de tipo ventana principal responsable de manejar un bucle de eventos que despacha mensajes. En tales arquitecturas, todos los demás objetos son pasivos, y su comportamiento en última instancia es disparado por mensajes del único objeto activo. En otros tipos de arquitecturas de sistemas secuenciales (como los sistemas de procesamiento de transacciones), no existe ningún objeto activo central evidente y, por tanto, el control tiende a estar distribuido entre los objetos pasivos del sistema.

## Identidad

**Semántica.** Khoshafian y Copeland ofrecen la siguiente definición:

> *"La identidad es aquella propiedad de un objeto que lo distingue de todos los demás objetos"* [12].

A continuación hacen notar que "la mayoría de los lenguajes de programación y de bases de datos utilizan nombres de variable para distinguir objetos temporales, mezclando la posibilidad de acceder a ellos con su identidad. La mayoría de los sistemas de bases de datos utilizan claves de identificación para distinguir objetos persistentes, mezclando el valor de un dato con la identidad.". El fracaso en reconocer la diferencia entre el nombre de un objeto y el objeto en sí mismo es fuente de muchos tipos de errores en la programación orientada a objetos.

**Ejemplo:** considérense las siguientes declaraciones en C++. Primero, se va a comenzar con una estructura simple que denota un punto en el espacio:

```
struct Punto {
    int x;
    int y;
    Punto() : x(0), y(0) {}
    Punto(int valorX, int valorY) : x(valorX), y(valorY) {}
};
```

Aquí se ha elegido declarar `Punto` como una estructura, no como una clase en toda su dimensión. La regla que se aplica para hacer esta distinción es simple. Si la abstracción representa un simple registro de otros objetos y no tiene un comportamiento verdaderamente interesante que se aplique al objeto en su conjunto, hágase una estructura. Sin embargo, si la abstracción exige un comportamiento más intenso que la simple introducción y recuperación de elementos altamente independientes del registro, hágase una clase. En el caso de la abstracción `Punto`, se define un punto como la representación de unas coordenadas (`x`, `y`) en el espacio. Por conveniencia, se suministra un constructor que proporciona un valor (0,0) por defecto, y otro constructor que inicializa un punto con un valor explícito (`x`, `y`).

A continuación se proporciona una clase que denota un elemento de pantalla. Un elemento de pantalla es una abstracción habitual en todos los sistemas basados en IGU: representa la clase base de todos los objetos que tienen una representación visual en alguna ventana, y así refleja la estructura y comportamiento comunes a todos esos objetos. He aquí una abstracción que es más que un simple registro de datos. Los clientes esperan ser capaces de dibujar, seleccionar y mover elementos de pantalla, así como interrogar sobre su estado de selección y su posición. Se puede plasmar la abstracción en la siguiente declaración de C++:

```
class ElementoPantalla {
public:

    ElementoPantalla();
    ElementoPantalla(const Punto& posicion);
    virtual ~ElementoPantalla();

    virtual void dibujar();
    virtual void borrar();
    virtual void seleccionar();
    virtual void quitarSeleccion();
    virtual void mover(cons Punto& posicion);

    int estaSeleccionado() const;
    Punto posicion() const;
    int estaBajo(const Punto& posicion) const;

protected:
    ...
};
```

Esta declaración está incompleta: se han omitido intencionadamente todos los constructores y operadores necesarios para manejar la copia, asignación y pruebas de igualdad. Se considerarán estos aspectos de la abstracción en la siguiente sección.

Puesto que se espera que los clientes declaren subclases de esta clase, se ha declarado el destructor y todos sus modificadores como virtual. En particular, se espera que las subclases concretas

redefinan `dibujar` para reflejar el comportamiento de dibujar en una ventana elementos específicos de cada dominio. No se ha declarado ninguno de los selectores como virtual, porque no se espera que las subclases redefinan este comportamiento. Nótese también que el selector `estaBajo` implica algo más que la recuperación de un simple valor del estado. Aquí, la semántica de esta operación requiere que el objeto calcule si el punto dado está en cualquier lugar del interior de la trama del elemento de pantalla.

Para declarar instancias de esta clase, se podría escribir lo siguiente:

```
ElementoPantallaelemento1;
ElementoPantalla*elemento2=newElementoPantalla(Punto(75,75));
ElementoPantalla*elemento3=newElementoPantalla(Punto(100,100));
ElementoPantalla*elemento4=0;
```

Como muestra la figura 3.1a, la elaboración de estas declaraciones crea cuatro nombres y tres objetos distintos. Específicamente, aparecen cuatro asignaciones distintas de memoria cuyos nombres son `elemento1`, `elemento2`, `elemento3` y `elemento4`, respectivamente. Además, `elemento1` es el nombre de un objeto `ElementoPantalla` concreto, pero los otros tres nombres denotan cada uno un *puntero* a un objeto `ElementoPantalla`. Solo `elemento2` y `elemento3` apuntan realmente a objetos `ElementoPantalla` precisos (porque solo sus declaraciones asignan espacio a un nuevo objeto `ElementoPantalla`); `elemento4` no designa tal objeto. Es más, los nombres de los objetos apuntados por `elemento2` y `elemento3` son anónimos; solo puede hacerse referencia a esos objetos concretos indirectamente, *desreferenciando* el valor de su puntero. Así, perfectamente se puede decir que `elemento2` apunta a un objeto `ElementoPantalla` concreto, cuyo nombre se puede expresar indirectamente como `*elemento2`. La identidad única (pero no necesariamente el nombre) de cada objeto se preserva durante el tiempo de vida del mismo, incluso cuando su estado cambia. Es como la cuestión Zen sobre los ríos: ¿es un río el mismo río de un día a otro, incluso aunque nunca fluye la misma agua a través de él? Por ejemplo, considérense los resultados de ejecutar las siguientes sentencias:

```
elemento1.mover(elemento3->posicion());
elemento4 = elemento3;
elemento4->mover(Punto(38, 100 ));
```

La figura 3.1b ilustra estos resultados. Aquí se aprecia que `elemento1` y el objeto designado por `elemento2` tienen el mismo estado en su posición, y que `elemento4` ahora designa también el mismo objeto que `elemento3`. Nótese que se usa la frase "el objeto designado por `elemento2`" en vez de decir "el objeto `elemento2`". La primera expresión es más precisa, aunque a veces se utilizarán esas frases de forma equivalente.

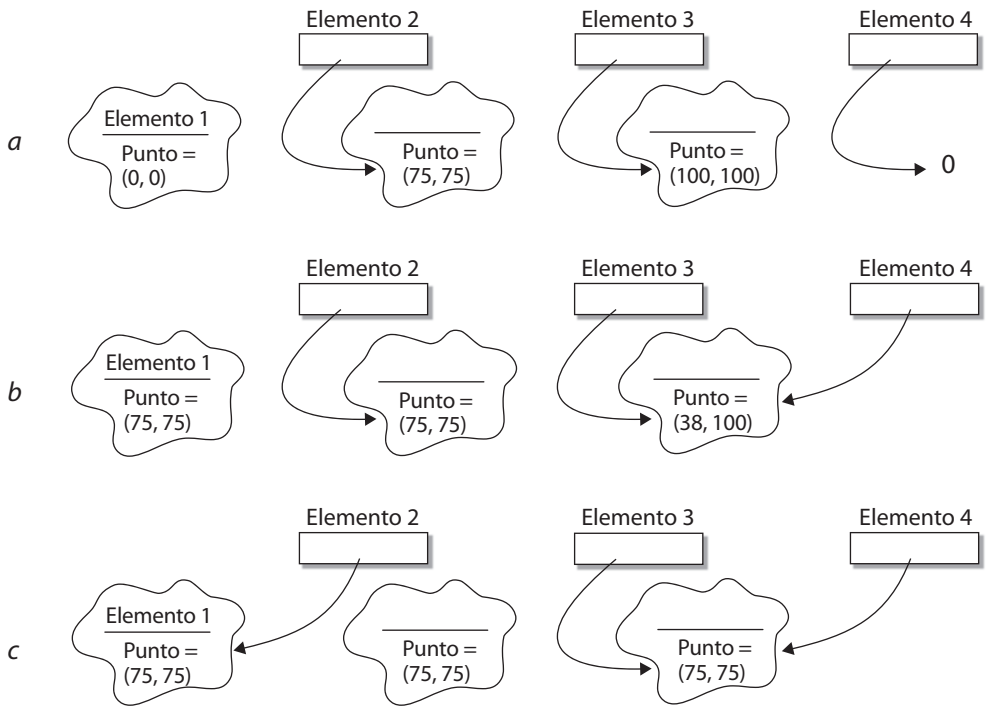


**Figura 3.1.** *Identidad de los objetos.*

Aunque `elemento1` y el objeto designado por `elemento2` tienen el mismo estado, representan objetos distintos. Además, nótese que se ha cambiado el estado del objeto designado por `elemento3` operando sobre él mediante su nuevo nombre indirecto, `elemento4`. Esta es una situación que se denomina *compartición estructural*, lo que quiere decir que un objeto dado puede nombrarse de más de una manera; en otras palabras, existen alias para el objeto. La compartición estructural es fuente de muchos problemas en programación orientada a objetos. El fracaso al reconocer los efectos laterales de operar sobre un objeto a través de alias lleva muchas veces a pérdidas de memoria, violaciones de acceso a memoria y, peor aún, cambios de estado inesperados. Por ejemplo, si se destruyese el objeto designado por `elemento3` utilizando la expresión `delete elemento3`, entonces el valor del puntero `elemento4` podría no tener significado; esta situación se denomina *referencia colgada* (*dangling reference*).

Considérese también la figura 3.1c, que ilustra los resultados de ejecutar las siguientes sentencias:

```
elemento2 = &elemento1;
elemento4->mover(elemento2->posicion());
```

La primera sentencia introduce un alias, porque ahora `elemento2` designa el mismo objeto que `elemento1`; la segunda sentencia accede al estado de `elemento1` a través del nuevo alias. Desgraciadamente, se ha introducido una pérdida de memoria: el objeto originalmente designado por `elemento2` ya no puede ser llamado, ya sea directa o indirectamente, y así su identidad se ha perdido. En lenguajes como Smalltalk y CLOS, tales objetos serán localizados por el recolector de basura y su espacio de memoria recobrado automáticamente, pero en lenguajes como C++, su espacio no se recobrará hasta que el programa que los creó finalice. Especialmente para programas de ejecución larga, las pérdidas de memoria de este tipo son fastidiosas o desastrosas.[5]

**Copia, asignación e igualdad.** La compartición estructural se da cuando la identidad de un objeto recibe un alias a través de un segundo nombre. En la mayoría de las aplicaciones interesantes orientadas a objetos, el uso de alias simplemente no puede evitarse. Por ejemplo, considérense las siguientes dos declaraciones de funciones en C++:

```
void remarcar(ElementoPantalla& i);
void arrastrar(ElementoPantalla i); // Peligroso
```

Invocar la primera función con el argumento `elemento1` crea un alias: el parámetro formal `i` denota una referencia al objeto designado por el parámetro actual, y desde ahora `elemento1` e `i` nombrarán al mismo objeto en tiempo de ejecución. Por contra, la invocación de la segunda función con el argumento `elemento1` produce una copia del parámetro actual, y por tanto no existe alias: `i` denota un objeto completamente diferente (pero con el mismo estado) que `elemento1`. En lenguajes como C++ donde existe una distinción entre pasar argumentos por referencia o por valor, hay que tener el cuidado de evitar operar sobre una copia de un objeto, cuando lo que se pretendía era operar sobre el objeto original.[6] Realmente, como se verá en una próxima sección, el pasar objetos por referencia en C++ es esencial para promover un comportamiento polimórfico. En general, el paso de objetos por referencia es la práctica más deseable para objetos no primitivos, porque su semántica solo involucra el copiar referencias, no estados; y de aquí que sea mucho más eficiente para pasar cualquier cosa que sea más grande que valores elementales.

En algunas circunstancias, sin embargo, la copia es la semántica que se pretendía utilizar, y en lenguajes como C++ es posible controlar la semántica de la copia. En particular, se puede

[5] Considérense los efectos de la pérdida de memoria en el software que controla un satélite o un marcapasos. Reiniciar el computador en un satélite que está a varios millones de kilómetros de la Tierra es bastante poco conveniente. Análogamente, la impredecible entrada en funcionamiento de una recolección de basura automáticamente en el software de un marcapasos será probablemente fatal. Por estas razones, los desarrolladores de sistemas en tiempo real evitan a menudo la asignación indiscriminada de espacio para los objetos en la memoria dinámica.

[6] En Smalltalk, la semántica de pasar objetos como argumentos para métodos es el equivalente del paso de argumentos por referencia en C++.

introducir un constructor de copia en la declaración de una clase, como en el siguiente fragmento de código, que se declararía como parte de la declaración para `ElementoPantalla`:

```
ElementoPantalla(const ElementoPantalla&);
```

En C++, puede invocarse un constructor de copia explícitamente (como parte de la declaración de un objeto) o implícitamente (como cuando se pasa un objeto por valor). La omisión de este constructor especial invoca el constructor de copia por defecto, cuya semántica está definida como una copia de miembros. Sin embargo, para objetos cuyo estado propio involucra a punteros o referencias a otros objetos, la copia por defecto de miembros suele ser peligrosa, porque la copia introduce entonces de forma implícita alias de nivel inferior. La regla que se aplica, por lo tanto, es que se omite un constructor de copia explícito solo para aquellas abstracciones cuyo estado se compone de valores simples, primitivos; en todos los demás casos, normalmente se proporciona un constructor de copia explícito.

Esta práctica distingue lo que algunos lenguajes llaman copia *superficial* versus *profunda*. Smalltalk, por ejemplo, proporciona los métodos `shallowCopy` (o copia superficial, que copia el objeto pero comparte su estado) y `deepCopy` (o copia en profundidad, que copia el objeto así como su estado, y así recursivamente). La redefinición de estas operaciones para clases agregadas permite una mezcla de semánticas: la copia de un objeto de nivel más alto podría copiar la mayor parte de su estado, pero podría introducir alias para ciertos elementos de nivel más bajo.

La asignación es del mismo modo en general una operación de copia, y en lenguajes como C++, su semántica puede controlarse también. Por ejemplo, se podría añadir la siguiente declaración a la declaración realizada para `ElementoPantalla`:

```
virtual ElementoPantalla& operator=(const ElementoPantalla&);
```

Se declara este operador como virtual, porque se espera que una subclase redefina su comportamiento. Como con el constructor de copia, se puede implantar esta operación para proporcionar una semántica de copia superficial o en profundidad. La omisión de esta declaración explícita invoca el operador de asignación por defecto, cuya semántica se define como una copia de miembros.

El problema de la igualdad está en relación muy estrecha con el de la asignación. Aunque se presenta como un concepto simple, la igualdad puede significar una de dos cosas.

Primero, la igualdad puede significar que dos nombres designan el mismo objeto. Segundo, la igualdad puede significar que dos nombres designan objetos distintos cuyos estados son iguales. Por ejemplo, en la figura 3.1c, ambos tipos de igualdad se evalúan como ciertos entre

`elemento1` y `elemento2`. Sin embargo, solo el segundo tipo de igualdad se evalúa como cierto entre `elemento1` y `elemento3`.

En C++ no hay un operador de igualdad por defecto, así que hay que establecer la semántica que se desee introduciendo los operadores explícitos para igualdad y desigualdad como parte de la declaración de `ElementoPantalla`.

```
virtual int operator==(const ElementoPantalla&) const;
int operator!=(const ElementoPantallas&) const;
```

El estilo adoptado es declarar el operador de igualdad como virtual (porque se espera que las subclases redefinan su comportamiento) y declarar el operador de desigualdad como no virtual (se desea siempre que el operador de desigualdad signifique la negación lógica de la igualdad; las subclases no deberían sobrescribir este comportamiento).

De manera similar, se puede definir explícitamente el significado de los operadores de ordenación, como las pruebas para menor-que o mayor-que entre dos objetos.

**Espacio de vida de un objeto.** El tiempo de vida de un objeto se extiende desde el momento en que se crea por primera vez (y consume así espacio por primera vez) hasta que ese espacio se recupera. Para crear explícitamente un objeto, hay que declararlo o bien asignarle memoria dinámicamente.

Declarar un objeto (como `elemento1` en el ejemplo anterior) crea una nueva instancia en la pila. Reservar espacio para un objeto (como `elemento3`) crea una nueva instancia en el montículo (*heap*). En C++, en ambos casos, siempre que se crea un objeto, se invoca automáticamente a su constructor, cuyo propósito es asignar espacio para el objeto y establecer un estado inicial estable. En lenguajes como Smalltalk, tales operaciones de constructor son realmente parte de la metaclase del objeto, no de la clase (se examinará la semántica de las metaclases más adelante en este capítulo).

Frecuentemente, los objetos se crean de forma implícita. Por ejemplo, en C++ el paso de un objeto por valor crea en la pila un nuevo objeto que es una copia del parámetro actual. Es más, la creación de objetos es transitiva: crear un objeto agregado también crea cualquier objeto que sea físicamente parte del conjunto. La redefinición de la semántica del constructor de copia y del operador de asignación en C++ permite un control explícito sobre el momento en que tales partes se crean y destruyen. Además, en C++ es posible redefinir la semántica del operador `new` (que asigna espacio para instancias en el heap), de forma que cada clase puede proporcionar su propia política de manejo de memoria.

En lenguajes como Smalltalk, un objeto se destruye automáticamente como parte de la recolección de basura cuando todas las referencias a él se han perdido. En lenguajes sin recolección de basura, como C++, un objeto sigue existiendo y consume espacio incluso si todas las referencias a él han desaparecido. Los objetos creados en la pila son destruidos de manera implícita siempre que el control sale del bloque en el que se declaró el objeto. Los objetos creados en el heap con

el operador `new` deben ser destruidos explícitamente con el operador `delete`. Si esto no se hace bien se producirán pérdidas de memoria, como se vio anteriormente. Liberar dos veces el espacio de memoria de un objeto (habitualmente a causa de un alias) es igualmente indeseable, y puede manifestarse en corrupciones de la memoria o en una caída completa del sistema.

En C++, siempre que se destruye un objeto ya sea implícita o explícitamente, se invoca automáticamente a su destructor, cuyo propósito es devolver el espacio asignado al objeto y sus partes, y llevar a cabo cualquier otra limpieza posterior a la existencia del objeto (como cerrar archivos o liberar recursos).[7]

Los objetos persistentes tienen una semántica ligeramente diferente respecto a la destrucción. Como se discutió en el capítulo anterior, ciertos objetos pueden ser persistentes, lo que quiere decir que su tiempo de vida trasciende al tiempo de vida del programa que los creó. Los objetos persistentes suelen ser elementos de algún marco de referencia mayor de una base de datos orientada a objetos, y así la semántica de la destrucción (y la creación) son en gran medida función de la política de la base de datos concreta. En tales sistemas, el enfoque más habitual de la persistencia es el uso de una clase aditiva persistente. Todos los objetos para los que se desea persistencia tienen así a esta clase aditiva como superclase en algún punto de su trama de herencias de clases.

## 3.2. Relaciones entre objetos

### Tipos de relaciones

Un objeto por sí mismo es bastante poco interesante. Los objetos contribuyen al comportamiento de un sistema colaborando con otros. Como sugiere Ingalls, "en lugar de un procesador triturador de bits que golpea y saquea estructuras de datos, tenemos un universo de objetos bien educados que cortésmente solicitan a los demás que lleven a cabo sus diversos deseos" [13]. Por ejemplo, considérese la estructura de objetos de un aeroplano, que se ha definido como "una colección de partes con una tendencia innata a caer a tierra, y que requiere esfuerzos y supervisión constantes para atajar ese suceso" [14]. Solo los esfuerzos en colaboración de todos los objetos componentes de un aeroplano lo hacen capaz de volar.

La relación entre dos objetos cualesquiera abarca las suposiciones que cada uno realiza acerca del otro, incluyendo qué operaciones pueden realizarse y qué comportamiento se obtiene. Se ha encontrado que hay dos tipos de jerarquías de objetos de interés especial en el análisis y diseño orientados a objetos, a saber:

- Enlaces.
- Agregación.

Seidewitz y Stark las llaman relaciones de *antigüedad* y de *padre/hijo*, respectivamente [15].

[7] Los destructores no recuperan automáticamente el espacio asignado por el operador `new`; los programadores deben recobrar este espacio explícitamente como parte de la destrucción.

## Enlaces

**Semántica.** El término *enlace* deriva de Rumbaugh, que lo define como "una conexión física o conceptual entre objetos" [16]. Un objeto colabora con otros objetos a través de sus enlaces con estos. Dicho de otro modo, un enlace denota la asociación específica por la cual un objeto (el cliente) utiliza los servicios de otro objeto (el suministrador o servidor), o a través de la cual un objeto puede comunicarse con otro.

La figura 3.2 ilustra varios enlaces diferentes. En esta figura, una línea entre dos iconos de objeto representa la existencia de un enlace entre ambos y significa que pueden pasar mensajes a través de esta vía. Los mensajes se muestran como líneas dirigidas que representan su dirección, con una etiqueta que nombra al propio mensaje. Por ejemplo, se ve que el objeto `unControlador` tiene enlaces hacia dos instancias de `ElementoPantalla` (los objetos `a` y `b`). Aunque tanto `a` como `b` tienen probablemente enlaces hacia la vista en la que aparecen, se ha elegido remarcar una sola vez tal enlace, desde `a` hasta `unaVista`. Solo a través de estos enlaces puede un objeto enviar mensajes a otro.

El paso de mensajes entre dos objetos es típicamente unidireccional, aunque ocasionalmente puede ser bidireccional. En el ejemplo, el objeto `unControlador` solo invoca operaciones sobre los dos objetos de pantalla (para moverlos y averiguar su posición), pero los objetos de pantalla no operan sobre el objeto controlador. Esta separación de intereses es bastante común en sistemas orientados a objetos bien estructurados.[8] Nótese también que, aunque el paso de mensajes es iniciado por el cliente (como `unControlador`) y dirigido hacia el servidor (como el objeto `a`), los datos pueden fluir en ambas direcciones a través de un enlace. Por ejemplo, cuando `unControlador` invoca la operación `mover` sobre `a`, los datos fluyen desde el cliente hacia el servidor. Sin embargo, cuando `unControlador` invoca la operación `estaBajo` sobre el objeto `b`, el resultado pasa del servidor al cliente.

Como participante de un enlace, un objeto puede desempeñar uno de tres papeles:

- Actor — Un objeto que puede operar sobre otros objetos pero nunca se opera sobre él por parte de otros objetos; en algunos contextos, los términos *objeto activo* y *actor* son equivalentes.
- Servidor — Un objeto que nunca opera sobre otros objetos; solo otros objetos operan sobre él.
- Agente — Un objeto que puede operar sobre otros objetos y además otros objetos pueden operar sobre él; un agente se crea normalmente para realizar algún trabajo en nombre de un actor u otro agente.

Ciñéndose al contexto de la figura 3.2, `unControlador` representa un objeto actor, `unaVista` representa un objeto servidor, y `a` representa un agente que lleva a cabo la petición del controlador para dibujar el elemento en la vista.

[8] De hecho, esta organización de controlador, vista y elemento de pantalla es tan común que podemos identificarla como un patrón de diseño, que puede por tanto reutilizarse. En Smalltalk, esto se llama un mecanismo MVC, de modelo/vista/controlador.

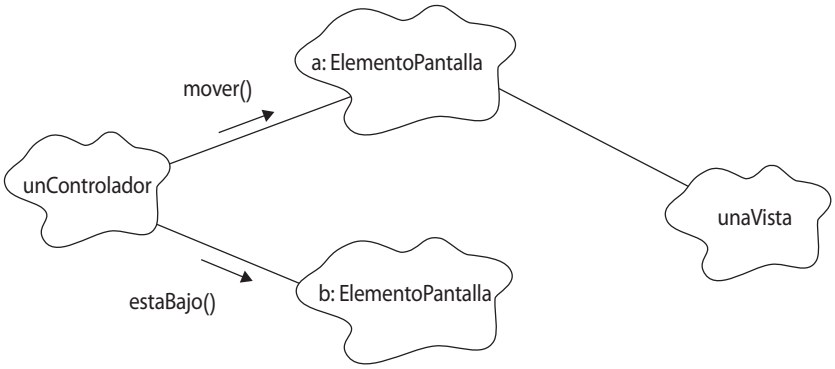


**Figura 3.2.** *Enlaces.*

**Ejemplo:** en muchos tipos diferentes de procesos industriales, algunas reacciones requieren un gradiente de temperatura, en la que se eleva la temperatura de alguna sustancia, se mantiene a tal temperatura durante un tiempo determinado, y entonces se deja enfriar hasta la temperatura ambiente. Procesos diferentes requieren perfiles diferentes: algunos objetos (como espejos de telescopios) deben enfriarse lentamente, mientras que otros materiales (como el acero) deben ser enfriados con rapidez. Esta abstracción de un gradiente de temperatura tiene un comportamiento lo suficientemente bien definido para justificar la creación de una clase, como la que sigue. Primero, se introduce una definición de tipos cuyos valores representan el tiempo transcurrido en minutos:

```
// Número que denota minutos transcurridos
typedef unsigned int Minuto;
```

Este typedef es similar a los de `Dia` y `Hora` que se introdujeron en el capítulo 2. A continuación, se proporciona la clase `GradienteTemperatura`, que es conceptualmente una correspondencia tiempo/temperatura:

```
class GradienteTemperatura {
public:

    GradienteTemperatura();
    virtual ~GradienteTemperatura();

    virtual void borrar();

    virtual void ligar(Temperatura, Minuto);

    Temperatura temperaturaEn(Minuto);
```

```
protected:
    ...
};
```

En consonancia con el estilo adoptado, se han declarado varias operaciones como virtual, porque se espera que existan subclases de esta clase.

En realidad, el comportamiento de esta abstracción es algo más que una mera correspondencia tiempo/temperatura. Por ejemplo, se podría fijar un gradiente de temperatura que requiere que la temperatura sea de 120° C en el instante 60 (una hora transcurrida en la rampa de temperatura) y de 65° C en el instante 180 (tres horas del proceso), pero entonces se desearía saber cuál debería ser la temperatura en el instante 120. Esto requiere interpolación lineal, que es otra parte del comportamiento que se espera de esta abstracción.

Un comportamiento que no se requiere explícitamente de esta abstracción es el control de un calentador que lleve a cabo un gradiente de temperatura concreta. En vez de eso, se prefiere una mayor separación de intereses, en la que este comportamiento se consigue mediante la colaboración de tres objetos: una instancia de gradiente de temperatura, un calentador y un controlador de temperatura. Por ejemplo, se podría introducir la siguiente clase:

```
class ControladorTemperatura {
public:

    ControladorTemperatura(Posicion);
    ~ControladorTemperatura();

    void procesar(const GradienteTemperatura&);

    Minuto planificar(const GradienteTemperatura&) const;

private:
    ...
};
```

Esta clase usa el tipo definido `Posicion` introducido en el capítulo 2. Nótese que no se espera que exista ninguna subclase de esta clase, y por eso no se ha hecho ninguna operación virtual.

La operación `procesar` suministra el comportamiento central de esta abstracción; su propósito es ejecutar el gradiente de temperatura dada en el calentador de la posición indicada. Por ejemplo, dadas las declaraciones siguientes:

```
GradienteTemperatura gradienteCreciente;
ControladorTemperatura controladorGradiente(7);
```

podría establecerse entonces un gradiente de temperatura particular, y decir al controlador que efectuase este perfil:

```
gradienteTemperatura.ligar(120, 60);
gradienteTemperatura.ligar(65, 180);

controladorGradiente.procesar(gradienteCreciente);
```

Considérese la relación entre los objetos `gradienteCreciente` y `controladorGradiente`: el objeto `controladorGradiente` es un agente responsable de efectuar un `gradiente` de temperatura, y así utiliza el objeto `gradienteCreciente` como servidor. Este enlace se manifiesta en el hecho de que el objeto `controladorGradiente` Utiliza el objeto `gradienteCreciente` como argumento para una de sus operaciones.

Un comentario al respecto del estilo: a primera vista, puede parecer que se ha ideado una abstracción cuyo único propósito es envolver una descomposición funcional en el seno de una clase para que parezca noble y orientada a objetos. La operación `planificar` sugiere que no es este el caso. Los objetos de la clase `ControladorTemperatura` tienen conocimiento suficiente para determinar cuándo un perfil determinado debería ser planificado, y así se expone esta operación como un comportamiento adicional de la abstracción. En algunos procesos industriales de alta energía (como la fabricación de acero), calentar una sustancia es un evento costoso, y es importante tener en cuenta cualquier calor que subsista de un proceso previo, así como el enfriamiento normal de cualquier calentador desatendido. La operación `planificar` existe para que los clientes puedan solicitar a un objeto `ControladorTemperatura` que determine el siguiente momento óptimo para procesar un gradiente de temperatura concreta.

**Visibilidad.** Considérense dos objetos, `A` y `B`, con un enlace entre ambos. Con el fin de que `A` envíe un mensaje a `B`, `B` debe ser visible para `A` de algún modo. Durante el análisis de un problema, se pueden ignorar perfectamente los problemas de visibilidad, pero una vez que se comienza a idear implantaciones concretas, hay que considerar la visibilidad a través de los enlaces, porque las decisiones en este punto dictan el ámbito y acceso de los objetos a cada lado del enlace.

En el ejemplo anterior, el objeto `controladorGradiente` tiene visibilidad hacia el objeto `gradienteCreciente`, porque ambos están declarados dentro del mismo ámbito, y `gradienteCreciente` se presenta como un argumento de una operación sobre el objeto `controladorGradiente`. Realmente, esta es solo una de las cuatro formas diferentes en que un objeto puede tener visibilidad para otro:

- El objeto servidor es global para el cliente.
- El objeto servidor es un parámetro de alguna operación del cliente.
- El objeto servidor es parte del objeto cliente.

- El objeto servidor es un objeto declarado localmente en alguna operación del cliente.

Cómo un objeto se hace visible a otro es un problema de diseño táctico.

**Sincronización.** Siempre que un objeto pasa un mensaje a otro a través de un enlace, se dice que los dos objetos están *sincronizados*. Para objetos de una aplicación completamente secuencial, esta sincronización suele realizarse mediante una simple invocación de métodos. Sin embargo, en presencia de múltiples hilos de control, los objetos requieren un paso de mensajes más sofisticado con el fin de tratar los problemas de exclusión mutua que pueden ocurrir en sistemas concurrentes. Como se describió anteriormente, los objetos activos contienen su propio hilo de control, y así se espera que su semántica esté garantizada en presencia de otros objetos activos. No obstante, cuando un objeto activo tiene un enlace con uno pasivo, hay que elegir uno de tres enfoques para la sincronización:

- Secuencial — La semántica del objeto pasivo está garantizada solo en presencia de un único objeto activo simultáneamente.
- Vigilado — La semántica del objeto pasivo está garantizada en presencia de múltiples hilos de control, pero los clientes activos deben colaborar para lograr la exclusión mutua.
- Síncrono — La semántica del objeto pasivo está garantizada en presencia de múltiples hilos de control, y el servidor garantiza la exclusión mutua.

## Agregación

**Semántica.** Mientras que los enlaces denotan relaciones igual-a-igual o cliente/servidor, la agregación denota una jerarquía todo/parte, con la capacidad de ir desde el todo (también llamado el *agregado*) hasta sus partes (conocidas también como *atributos*). En este sentido, la agregación es un tipo especializado de asociación. Por ejemplo, como se muestra en la figura 3.3, el objeto `controladorGradiente` tiene un enlace al objeto `gradienteCreciente` así como un atributo `h` cuya clase es `Calentador`. El objeto `controladorGradiente` es así el todo, y `h` es una de sus partes. En otras palabras, `h` es una parte del estado del objeto `controladorGradiente`. Dado el objeto `controladorGradiente`, es posible encontrar su calentador correspondiente `h`. Dado un objeto como `h`, es posible llegar al objeto que lo encierra (también llamado su *contenedor*) si y solo si este conocimiento es parte del estado de `h`.

La agregación puede o no denotar contención física. Por ejemplo, un aeroplano se compone de alas, motores, tren de aterrizaje, etc.: es un caso de contención física. Por contra, la relación entre un accionista y sus acciones es una relación de agregación que no requiere contención física. El accionista únicamente posee acciones, pero las acciones no son de ninguna manera parte física del accionista. Antes bien, esta relación todo/parte es más conceptual y por tanto menos directa que la agregación física de las partes que forman un aeroplano.

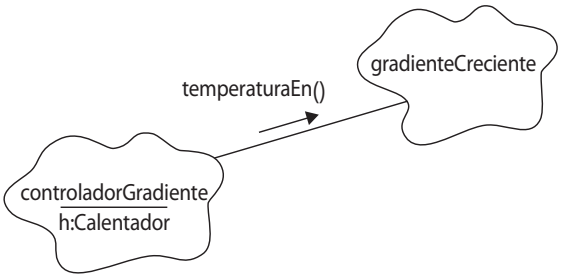


**Figura 3.3**. *Agregación.*

Existen claros pros y contras entre los enlaces y la agregación. La agregación es a veces mejor porque encapsula partes y secretos del todo. A veces son mejores los enlaces porque permiten acoplamientos más débiles entre los objetos. Las decisiones de ingeniería inteligentes requieren sopesar cuidadosamente ambos factores.

Por implicación, un objeto que es atributo de otro tiene un enlace a su agregado. A través de este enlace, el agregado puede enviar mensajes a sus partes.

**Ejemplo:** para continuar con la declaración de la clase `ControladorTemperatura`, podría completarse su parte private como sigue:

```
Calentador h;
```

Esto declara a `h` como una parte de cada instancia de `ControladorTemperatura`. De acuerdo con la declaración de la clase `Calentador` realizada en el capítulo anterior, hay que crear correctamente este atributo, porque su clase no suministra un constructor por defecto. Así, se podría escribir el constructor para `ControladorTemperatura` como sigue:

```
ControladorTemperatura::ControladorTemperatura(Posicion p)
    : h(p) {}
```

## 3.3. La naturaleza de una clase

### Qué es y qué no es una clase

Los conceptos de clase y objeto están estrechamente entretejidos, porque no puede hablarse de un objeto sin atención a su clase. Sin embargo, existen diferencias importantes entre ambos términos. Mientras que un objeto es una entidad concreta que existe en el tiempo y el espacio, una clase

representa solo una abstracción, la "esencia" de un objeto. Así, se puede hablar de la clase `Mamifero`, que representa las características comunes a todos los mamíferos. Para identificar a un mamífero particular en esta clase, hay que hablar de "este mamífero" o "aquel mamífero".

En términos corrientes, se puede definir una clase como "un grupo, conjunto o tipo marcado por atributos comunes o un atributo común; una división, distinción o clasificación de grupos basada en la calidad, grado de competencia o condición" [17].[9] En el contexto del análisis y diseño orientados a objetos, se define una clase como sigue:

> *Una clase es un conjunto de objetos que comparten una estructura común y un comportamiento común.*

Un solo objeto no es más que una instancia de una clase.

¿Qué no es una clase? Un objeto no es una clase, aunque, curiosamente, como se describirá después, una clase puede ser un objeto. Los objetos que no comparten estructura y comportamiento similares no pueden agruparse en una clase porque, por definición, no están relacionados entre sí a no ser por su naturaleza general como objetos.

Es importante notar que la clase –tal como la define la mayoría de los lenguajes de programación– es un vehículo necesario pero no suficiente para la descomposición. A veces las abstracciones son tan complejas que no pueden expresarse convenientemente en términos de una sola declaración de clase. Por ejemplo, a un nivel de abstracción lo bastante alto, un marco de referencia de un IGU, una base de datos, o un sistema completo de inventario son conceptualmente objetos individuales, ninguno de los cuales puede expresarse como una sola clase.[10] En lugar de eso, es mucho mejor capturar esas abstracciones como una agrupación de clases cuyas instancias colaboran para proporcionar el comportamiento y estructura deseados. Stroustrup llama a tal agrupamiento un *componente* [18]. Aquí, en cambio, se llama a tales grupos una *categoría de clases*.

## Interfaz e implementación

Meyer [19] y Snyder [20] han sugerido que la programación es en gran medida un asunto de "contratos": las diversas funciones de un problema mayor se descomponen en problemas más pequeños mediante subcontratas a diferentes elementos del diseño. En ningún sitio es más evidente esta idea que en el diseño de clases.

Mientras que un objeto individual es una entidad concreta que desempeña algún papel en el sistema global, la clase captura la estructura y comportamiento comunes a todos los objetos

---

[9] Con permiso de *Webster 's Third New International Dictionary* (C) 1986 por Merriam-Webster Inc., editor de los diccionarios Merriam-Webster (R).

[10] Uno puede verse tentado a expresar tales abstracciones en una sola clase, pero la granularidad de reutilización y cambio sería nefasta. Tener una interfaz grande es mala práctica, porque la mayoría de los clientes querrán referenciar solo un pequeño subconjunto de los servicios proporcionados. Es más, el cambio de una parte de una interfaz enorme deja obsoletos a todos los clientes, incluso a los que no tienen que ver con las partes que cambiaron. La anidación de clases no elimina estos problemas; solo los aplaza.

relacionados. Así, una clase sirve como una especie de contrato que vincula a una abstracción y todos sus clientes. Capturando estas decisiones en la interfaz de una clase, un lenguaje con comprobación estricta de tipos puede detectar violaciones de este contrato durante la compilación.

Esta visión de la programación como un contrato lleva a distinguir entre la visión externa y la visión interna de una clase. La *interfaz* de una clase proporciona su visión externa y por tanto enfatiza la abstracción a la vez que oculta su estructura y los secretos de su comportamiento. Esta interfaz se compone principalmente de las declaraciones de todas las operaciones aplicables a instancias de esta clase, pero también puede incluir la declaración de otras clases, constantes, variables y excepciones, según se necesiten para completar la abstracción. Por contraste, la *implementación* de una clase es su visión interna, que engloba los secretos de su comportamiento. La implementación de una clase se compone principalmente de la implementación de todas las operaciones definidas en la interfaz de la misma.

Se puede dividir además la interfaz de una clase en tres partes:

- Public (*pública*) Una declaración accesible a todos los clientes.
- Protected (*protegida*) Una declaración accesible solo a la propia clase, sus subclases, y sus clases amigas (friends).
- Private (*privada*) Una declaración accesible solo a la propia clase y sus clases amigas.

Los distintos lenguajes de programación ofrecen diversas mezclas de partes public, protected y private, entre las que los desarrolladores pueden elegir para establecer derechos específicos de acceso para cada parte de la interfaz de una clase y de este modo ejercer un control sobre qué pueden ver los clientes y qué no pueden ver.

En particular, C++ permite a los desarrolladores hacer distinciones explícitas entre estas tres partes distintas.[11] El mecanismo de "amistad" de C++ permite a una clase distinguir ciertas clases privilegiadas a las que se otorga el derecho de ver las partes protected y private de otra. La amistad rompe el encapsulamiento de una clase, y así, al igual que en la vida real, hay que elegirla cuidadosamente. En contraste, Ada permite declaraciones public o private, pero no protected. En Smalltalk, todas las variables de instancia son private, y todos los métodos son public. En Object Pascal, tanto los campos como las operaciones son public y, por tanto, no están encapsulados. En CLOS, las funciones genéricas son public y las ranuras *(slots)* deben hacerse private, aunque su acceso puede romperse mediante la función `slot-value`.

El estado de un objeto debe tener alguna representación en su clase correspondiente, y por eso se expresa típicamente como declaraciones de constantes y variables situadas en la parte private o protected de la interfaz de una clase. De este modo, se encapsula la representación común a todas las instancias de una clase, y los cambios en esta representación no afectan funcionalmente a ningún cliente externo.

El lector cuidadoso puede preguntarse por qué la representación de un objeto es parte de la interfaz de una clase (aunque sea una parte no pública) y no de su implementación. Por razones

[11] La `struct` de C++ es un caso especial, en el sentido de que una `struct` es un tipo de clase con todos sus elementos `public`.

prácticas, hacer lo contrario requeriría o bien hardware orientado a objetos o bien una tecnología de compiladores muy sofisticada. Específicamente, cuando un compilador procesa una declaración de un objeto como la siguiente en C++:

```
ElementoPantalla elemento1;
```

debe saber cuánta memoria hay que asignar al objeto `elemento1`. Si se hubiera definido la representación del objeto en la implementación de la clase, habría que completar la implementación de la clase antes de poder usar ningún cliente, frustrando así el verdadero propósito de la separación entre las visiones externa e interna de la clase.

Las constantes y variables que forman la representación de una clase se conocen bajo varias denominaciones, dependiendo del lenguaje concreto que se utilice. Por ejemplo, Smalltalk usa el término *variable de instancia*, Object Pascal usa el término *campo* (field), C++ usa el término *objeto miembro* y CLOS usa el término *ranura* (slot). Se utilizarán estos términos indistintamente para denotar las partes de una clase que sirven como representación del estado de su instancia.

### Ciclo de vida de las clases

Se puede llegar a la comprensión del comportamiento de una clase simple con solo comprender la semántica de sus distintas operaciones públicas de forma aislada. Sin embargo, el comportamiento de clases más interesantes (como el movimiento de una instancia de la clase `ElementoPantalla`, o la planificación de una instancia de la clase `ControladorTemperatura`) implica la interacción de sus diversas operaciones a lo largo del tiempo de vida de cada una de sus instancias. Como se describió antes en este capítulo, las instancias de tales clases actúan como pequeñas máquinas y, ya que todas esas instancias incorporan el mismo comportamiento, se puede utilizar la clase para capturar esta semántica común de orden respecto al tiempo y los eventos. Puede describirse tal comportamiento dinámico para ciertas clases interesantes mediante el uso de máquinas de estados finitos.

## 3.4 Relaciones entre clases

### Tipos de relaciones

Considérense por un momento las analogías y diferencias entre las siguientes clases de objetos: flores, margaritas, rosas rojas, rosas amarillas, pétalos y mariquitas. Pueden hacerse las observaciones siguientes:

- Una margarita es un tipo de flor.
- Una rosa es un tipo (distinto) de flor.

- Las rosas rojas y las rosas amarillas son tipos de rosas.
- Un pétalo es una parte de ambos tipos de flores.
- Las mariquitas se comen a ciertas plagas como los pulgones, que pueden infectar ciertos tipos de flores.

Partiendo de este simple ejemplo se concluye que las clases, al igual que los objetos, no existen aisladamente. Antes bien, para un dominio de problema específico, las abstracciones clave suelen estar relacionadas por vías muy diversas e interesantes, formando la estructura de clases del diseño [21].

Se establecen relaciones entre dos clases por una de dos razones. Primero, una relación entre clases podría indicar algún tipo de compartición. Por ejemplo, las margaritas y las rosas son tipos de flores, lo que quiere decir que ambas tienen pétalos con colores llamativos, ambas emiten una fragancia, etc. Segundo, una relación entre clases podría indicar algún tipo de conexión semántica. Así, se dice que las rosas rojas y las rosas amarillas se parecen más que las margaritas y las rosas, y las margaritas y las rosas se relacionan más estrechamente que los pétalos y las flores. Análogamente, existe una conexión simbiótica entre las mariquitas y las flores: las mariquitas protegen a las flores de ciertas plagas, que a su vez sirven de fuente de alimento para la mariquita.

En total, existen tres tipos básicos de relaciones entre clases [22]. La primera es la generalización/especialización, que denota una relación "**es-un**" (*is a*). Por ejemplo, una rosa es un tipo de flor, lo que quiere decir que una rosa es una subclase especializada de una clase más general, la de las flores. La segunda es la relación **todo/parte** (*whole/part*), que denota una relación "**parte-de**" (*part of* ). Así, un pétalo no es un tipo de flor; es una parte de una flor. La tercera es la asociación, que denota alguna dependencia semántica entre clases de otro modo independientes, como entre las mariquitas y las flores. Un ejemplo más: las rosas y las velas son clases claramente independientes, pero ambas representan cosas que podrían utilizarse para decorar la mesa de una cena.

En los lenguajes de programación han evolucionado varios enfoques comunes para plasmar relaciones de generalización/especialización, todo/parte y asociación. Específicamente, la mayoría de los lenguajes orientados a objetos ofrecen soporte directo para alguna combinación de las siguientes relaciones:

- Asociación.
- Herencia.
- Agregación.
- Uso.
- **Instanciación** (creación de instancias o ejemplares).
- Metaclase.

Un enfoque alternativo para la herencia involucra un mecanismo lingüístico llamado *delegación*, en el que los objetos se consideran prototipos (también llamados *ejemplares*) que delegan su comportamiento en objetos relacionados, eliminando así la necesidad de clases [23].

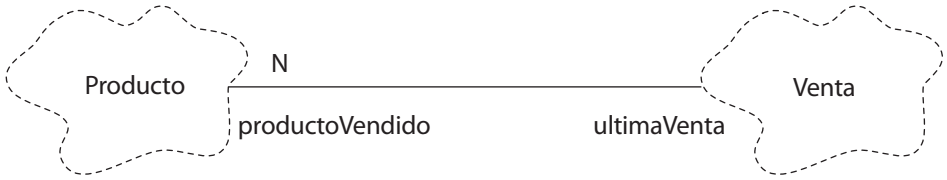


**Figura 3.4.** *Asociación.*

De estos seis tipos diferentes de relaciones entre clases, las asociaciones son el más general, pero también el de mayor debilidad semántica. La identificación de asociaciones entre clases es frecuentemente una actividad de análisis y de diseño inicial, momento en el cual se comienza a descubrir las dependencias generales entre las abstracciones. A medida que se continúa el diseño y la implementación, se refinarán a menudo estas asociaciones débiles orientándolas hacia una de las otras relaciones de clase más concretas.

La herencia es quizás la más interesante, semánticamente hablando, de estas relaciones concretas, y existe para expresar relaciones de generalización/especialización. Según nuestra experiencia, sin embargo, la herencia es un medio insuficiente para expresar todas las ricas relaciones que pueden darse entre las abstracciones clave en un dominio de problema dado. Se necesitan también relaciones de agregación, que suministran las relaciones todo/parte que se manifiestan en las instancias de las clases. Además, son necesarias las relaciones de uso, que establecen los enlaces entre las instancias de las clases. Para lenguajes como Ada, C++ y Eiffel, se necesitan también las relaciones de instanciación que, al igual que la herencia, soportan un tipo de generalización, aunque de forma completamente diferente. Las relaciones de metaclase son bastante distintas y solo las soportan explícitamente lenguajes como Smalltalk y CLOS. Básicamente, una metaclase es la clase de una clase, un concepto que permite tratar a las clases como objetos.