Modularidad y orientación a objetos en Java

Albert Rubio Borja Valles

Última versión: febrero de 2005

1 Modularidad y orientación a objetos. Traducción a Java de diseños modulares

En esta asignatura sólo aplicaremos la orientación a objetos a fin de aprovechar mejor los mecanismos que el lenguaje Java ofrece para codificar nuestros diseños modulares. Sin embargo, este estilo de programación conlleva un modo particular de abordar la estructuración de programas. Veamos en primer lugar algunos aspectos conceptuales del mismo para pasar, seguidamente, a aplicarlos en nuestro contexto.

1.1 Clases y objetos

Considerado como unidad de estructuración de programas, el concepto de clase equivale esencialmente al de módulo. Por tanto, sus fases de diseño son las mismas, si bien la forma de especificar, utilizar e implementar difiere ligeramente entre un estilo y el otro. Las clases y los objetos representan una manera alternativa de modelizar los datos (la información) de un programa.

En el estilo modular clásico, un **módulo de datos** define un **tipo** con un dominio de valores y un conjunto de operaciones que trabajan con diversos parámetros, de los que al menos uno es del propio tipo. En programación orientada a objetos, una **clase de datos** define una estructura de atributos y operaciones que se particulariza (también se suele decir que se "instancia") en cada objeto de la misma que se crea. En este aspecto, una clase también hace el papel de "tipo" de dichos objetos.

En el diseño modular clásico, una **variable** posee un valor del tipo al que pertenece y sobre ella se aplican las operaciones de dicho tipo. Podemos decir que una variable *contiene* la información. Sin embargo, cada vez que se crea un **objeto** de una clase de datos, éste se convierte en un representante de la estructura definida en tal clase, de forma que recibe como componentes sus atributos y sus operaciones (también llamadas *métodos*). En este caso el objeto *es* la información.

La mayoría de lenguajes orientados a objetos introducen maneras de trabajar que pueden resultar sorprendentes si previamente se ha estudiado diseño modular, pero que son perfectamente consistentes con la mencionada diferencia conceptual entre ambos estilos.

Por ejemplo, las operaciones de una clase se conciben para ser aplicadas sobre un objeto concreto de la clase, y por eso se parametrizan de una manera especial, de tal manera que el objeto que es creado, modificado o consultado por una operación no aparece explícitamente en la lista de paramétros de la misma. Se mencionará sólo en las llamadas a dicha operación, mientras que en el código de la misma es referido de forma implícita.

Otra práctica habitual es que las clases no se tratan igual que los tipos simples a la hora de parametrizar las operaciones o respecto a la operación de asignación. La primera situación se estudia en detalle en la primera sesión de laboratorio de la asignatura. En cuanto a la segunda, cuando se desea producir dos objetos distintos con la misma información, no se habla de asignar un objeto a otro sino de copiar (o también, clonar) un objeto en otro. De hecho, en la mayoría de lenguajes orientados a objetos se permite realizar asignaciones entre objetos, pero con una

semántica muy distinta a la de la asignación clásica. Al realizarse tal asignación entre objetos, el resultado es que la misma información pasa a tener dos nombres (situación habitualmente denominada “aliasing”), lo que puede dar lugar a efectos laterales difícilmente controlables. Por ello, en nuestro contexto limitaremos estas asignaciones a un reducido número de casos particulares que describiremos en su momento, aunque en otros contextos puede resultar útil explotar esta compartición de información.

En la práctica, esta filosofía de trabajo da lugar a una serie de aspectos metodológicos nuevos que presentamos en los apartados siguientes, aplicados de forma particular al lenguaje Java.

2 Especificación y uso de clases

Normalmente una clase de datos se diseña para la exportación de sus elementos. Por ello en Java ha de declararse pública y ha de albergarse en un fichero del mismo nombre que la clase. Nótese que al hablar de módulos, un módulo y el tipo que define son entidades diferentes, que podrían tener incluso distinto nombre. Esto no ocurre en Java ya que el concepto de clase engloba a ambos.

Por ejemplo, para la clase `Estudiante`, crearíamos un fichero llamado `Estudiante.java` que comenzaría así:

```
public class Estudiante{
```

En cuanto a las operaciones, también llamadas métodos, han de declararse *públicos* para que puedan ser usados por otras clases. Por otra parte, si un método lanza una excepción, ha de informarse de ello al usuario, mediante la palabra `throws`.

2.1 Parametrización de los métodos

Como hemos mencionado, las operaciones de una clase se aplican sobre un objeto concreto de la misma. En esta línea, Java permite una simplificación sintáctica al declarar las cabeceras de los métodos, consistente en que el objeto sobre el que se aplica un método no aparece como parámetro en su correspondiente cabecera. Se considera que el objeto aparece de forma implícita y como tal lo mencionaremos en la especificación del método. Notad que ello obligará también a algunos cambios en la implementación de las operaciones como veremos en su momento.

Ejemplo: consideremos la función `tiene_nota` del módulo `Estudiante`, cuya cabecera era

```
funcion tiene_nota (e: Estudiante) dev b: booleano
/* cierto */
/* b = e tiene nota */
```

Su traducción a Java será

```
public boolean tiene_nota ()
/* cierto */
/* retorna si el parámetro implícito tiene nota */
```

Al realizar una llamada a un método, el objeto sobre el que se aplica precede al nombre del método y al resto de los parámetros, separado por un punto.

```
<nombre_del_objeto>.<nombre_del_método>(<otros parámetros>)
```

Ejemplo: en la función aterior, el estudiante sobre el cual se realiza la consulta será un parámetro implícito: no aparece en la cabecera y sólo se menciona al usar la operación. Si `est` es un objeto ya creado de la clase `Estudiante` y `b` es una variable booleana, la instrucción

```
b:=tiene_nota(est);
```

se traduce a Java como

```
b=est.tiene_nota();
```

Pero no acaban ahí las diferencias: el objeto sobre el que se aplica al método funciona como un parámetro de entrada/salida ya que su valor inicial puede ser relevante pero si es modificado por el método tal modificación es permanente. Por ejemplo, al implementar en Java la acción `modificar_nota`, podemos aprovechar la modificabilidad del parámetro implícito si la definimos así:

```
public void modificar_nota (double nota) throws Exception
/* nota ha de estar entre 0 y 10; si no, se produce una excepción,
   el parámetro implícito debe tener nota */
/* el parámetro implícito conserva su DNI y su nota pasa a ser "nota" */
```

De este modo, si `est` es un `Estudiante` con nota y `x` es una nota válida, podemos hacer lo siguiente para que la nota de `est` pase a ser `x`.

```
est.modificar_nota(x);
```

Por todo ello, hemos de tener muy claro si deseamos conservar el valor original de un objeto después de una operación de modificación, en cuyo caso habría que obtener previamente una copia del mismo. Como esta situación puede darse con cierta frecuencia, toda clase ha de disponer de una operación de copia (o clonación) de objetos, que puede ser una acción o una función.

En cuanto a los demás parámetros, recordemos que Java considera a los de tipos simples como sólo de entrada, mientras que los objetos son tratados, en lo que se refiere a sus atributos, como de entrada/salida o salida. Por eso es importante mencionar en la especificación de cada operación, una vez traducida a Java, qué parámetros no se van a modificar. De esa forma, si no queremos perder el valor original de un parámetro con riesgo de modificación, habremos de obtener una copia del mismo antes de realizar la llamada.

2.2 Clasificación de las operaciones

Las operaciones de una *clase de datos* deben clasificarse durante su especificación en tres categorías:

- *Creadoras de objetos*. Son funciones que sirven para crear objetos nuevos de la clase, ya sean objetos inicializados con un mínimo de información o el resultado de cálculos más complejos. Las llamadas a estas funciones es uno de los casos particulares donde permitimos la asignación entre objetos.

  Supongamos que hemos especificado la operación de copia de estudiantes como función. Como tal puede traducirse a Java:

  ```
  public Estudiante copiar_estudiante()
  /* cierto */
  /* el resultado es una copia del parámetro implícito */
  ```

  y la podemos invocar así para hacer que `est2` sea una copia de `est1`, que se supone ya creado

  ```
  Estudiante est2=est1.copiar_estudiante();
  ```

  En Java se puede definir un tipo particular de operaciones creadoras de una clase. Se trata de funciones con el mismo nombre que la clase, que devuelven un objeto nuevo de la misma. Notad que como el nombre del método coincide con el tipo del resultado, su cabecera sólo lo menciona una vez. Pueden definirse varias versiones con distintas listas de parámetros. En caso de no definirse ninguna operación de este tipo, Java proporciona una por defecto, sin parámetros, que crea un objeto sin información. Para invocarse, han de estar precedidas por la palabra `new`.

  Por ejemplo, podemos crear un estudiante sin datos con la instrucción

  ```
  Estudiante est = new Estudiante();
  ```

  Si queremos crear un estudiante inicializado con un DNI, declararemos una operación con esta cabecera

  ```
  public Estudiante (int dni) throws Exception
  /* dni>=0; si no, se produce una excepción */
  /* devuelve un estudiante con DNI=dni y sin nota */
  ```

  La usamos para obtener un estudiante con DNI igual a `n` y sin nota mediante la llamada

```
Estudiante est = new Estudiante(n);
```

Notad que esta función es una posible traducción a Java de `crear_estudiante`.

- *Modificadoras*. Transforman el objeto que las invoca, si es necesario con información aportada por otros parámetros. Conceptualmente no deberían poder modificar otros objetos aunque, por supuesto, Java permite hacerlo mediante efectos laterales. Normalmente serán acciones.

  Ejemplo: la operación `añadir_nota` se traduciría

```
public void anadir_nota (double nota) throws Exception
/* nota ha de estar entre 0 y 10; si no, se produce una excepción,
   el parámetro implícito no debe tener nota */
/* el parámetro implícito conserva su DNI y su nota es "nota"  */
```

  En ocasiones, una operación creadora se puede definir de manera equivalente como modificadora, siempre que antes de invocar ésta se obtenga un objeto nuevo mediante la creadora por defecto. Por ejemplo, si la operación `crear_estudiante`, se especificase como una acción podría traducirse como vemos a continuación, haciendo innecesaria la segunda versión de la creadora anterior.

```
public void crear_estudiante(int dni) throws Exception
/* dni>=0; si no, se produce una excepción */
/* el parámetro implícito es un estudiante con DNI=dni y sin nota */
```

  Con ella se puede crear un estudiante con un DNI igual a `n` y sin nota así

```
Estudiante est = new Estudiante();
est.crear_estudiante(n)
```

  Del mismo modo, si la operación de copia se especificase como acción, al traducirla a Java resultaría esencialmente una modificadora:

```
public void copiar_estudiante(Estudiante est)
/* cierto */
/* el parámetro implícito pasa a ser una copia de est */
```

  y la podemos invocar para crear un estudiante `est2` que sea una copia de `est1`

```
Estudiante est2 = new Estudiante();
est2.copiar_estudiante(est1)
```

Como elemento metodológico, intentaremos siempre que sea posible realizar estas dos instrucciones de manera consecutiva.

- *Consultoras.* Proporcionan información sobre el objeto que las invoca, quizá con ayuda de otros parámetros. Generalmente son funciones, salvo si devuelven varios resultados, en cuyo caso podrían convertirse en acciones con varios parámetros de salida. Ejemplo: la operación `consultar_nota`

  ```
  public double consultar_nota() throws Exception
  /* Provoca una excepción si el estudiante no tiene nota */
  /* Devuelve la nota del parámetro implícito */
  ```

  Para obtener la nota de un estudiante `est` y guardarla en una variable x escribimos

  ```
  double x = est.consultar_nota();
  ```

  Por último, recordemos que si una consultora devuelve objetos de otras clases, éstos han de ser nuevos al igual que en el caso de las creadoras.

Para terminar, consideremos las operaciones de *lectura* y *escritura*. En todos los casos pedimos que se incluya como parámetro un objeto de la clase `InOut`, sobre el cual se realizarán los `read` y los `write`. Por ello, se han de transmitir las excepciones de estas operaciones.

La operación de lectura puede ser creadora o modificadora. En el primer caso produce un `Estudiante` nuevo, mientras que en el segundo modifica los componentes de uno ya existente.

La creadora se especifica así

```
public Estudiante(InOut canal) throws Exception
   /* Se leen pares <int dni, double nota>. Si se lee una nota no
      valida (fuera del intervalo [0..10]), el estudiante resultante
      queda sin nota. */
```

y la modificadora así

```
public void leer_estudiante(InOut canal) throws Exception
  /* Se leen pares <int dni, double nota>. Si se lee una nota no
     valida (fuera del intervalo [0..10]), el estudiante resultante
     queda sin nota. */
```

La de escritura se define de manera similar

```
public void escribir_estudiante(InOut canal) throws Exception
  /* Si el estudiante no tiene nota escribe NP. */
```

Ejemplos de llamadas:

```
e = new Estudiante(io);
e.leer_estudiante(io);
e.escribir_estudiante(io);
```

2.3 Métodos estáticos

Java también permite definir métodos que no sean propiedad de ningún objeto sino de la clase entera. Por tanto no han de ser invocados sobre un parámetro implícito, sino de una manera idéntica a las acciones y funciones del lenguaje algorítmico, con las particularidades ya comentadas de la parametrización en Java: los tipos simples son siempre de entrada, mientras que los objetos son tratados como de entrada/salida o salida. Para conseguir que un método sea estático, éste se declara precedido del calificador `static`.

Un caso habitual de métodos estáticos son los pertenecientes a los programas principales, dado que éstos no definen una clase de datos sobre los que aplicar tales métodos. De ahí que tanto su método `main` como los demás que puedan necesitarse se declaran estáticos.

Ejemplo: el redondeo de la nota de un estudiante constituye un ejemplo de uso de la clase `Estudiante`. Puede realizarse en el programa principal `red1.java` con una acción estática

```
public static void redondear (Estudiante est) throws Exception
/* est es un estudiante con nota; si no, se lanza una excepción */
   {
     est.modificar_nota(((int)(10*(est.consultar_nota()+.05))) /10.0);
   }
/* est conserva su DNI y su nota nueva es el resultado de redondear
   la original. */
```

Para redondear la nota del estudiante `e`, suponiendo que tuviera nota, haríamos

```
redondear(e);
```

2.4 Asignación entre objetos

Como ya hemos mencionado, la asignación entre objetos es una operación peligrosa por los efectos laterales que puede producir. Por ejemplo, consideremos dos estudiantes `est1` y `est2`. Si aplicamos la asignación

```
est2=est1;
```

lo que ocurre es que `est1` y `est2` pasan a ser dos nombres distintos ("alias") del mismo objeto, de forma que si modificamos uno también cambia el otro.

Por esa razón, sólo permitiremos la asignación en el caso de las llamadas a funciones que devuelvan objetos nuevos. Para asegurarnos de que esta restricción se cumple, obligaremos a que nuestras funciones sólo puedan devolver objetos nuevos. Aplicaremos tal restricción tanto a las funciones creadoras (devuelven objetos de la clase a la que pertenecen) como a las consultoras (devuelven objetos de otras clases) y a las funciones estáticas.

De este modo, el significado real de una asignación entre objetos se reduce a una de las dos siguientes posibilidades:

- si se trata de la primera vez que se realiza la asignación sobre un cierto objeto, tendremos la *creación* del mismo.
- si ya se han realizado asignaciones con anterioridad sobre dicho objeto, el resultado es la *destrucción* del mismo y la *creación* de otro nuevo con el mismo nombre.

Por ejemplo, si deseamos redondear la nota de los estudiantes de una secuencia, podemos emplear el mismo objeto de clase `Estudiante` para leer y tratar a todos los estudiante de la secuencia. La primera vez que se le da un valor al objeto estamos creándolo, las restantes veces se destruye el objeto antiguo y se crea uno nuevo.

Por otra parte, emplearemos la política de que si un objeto se crea como resultado de una función, hay que declararlo en ese momento, como en los ejemplos anteriores.

3 Implementación de clases

La traducción a Java de una clase sigue los mismos pasos que la implementación de un módulo de datos, es decir, hay que obtener una representación de la misma a partir de las clases ya existentes y, a continuación, codificar sus operaciones en Java.

Recordemos que para garantizar que la clase sea independiente de su representación, el acceso a los atributos de sus objetos ha de estar prohibido, salvo si se realiza mediante las operaciones de modificación y consulta (la mayoría de lenguajes proporciona mecanismos para eludir esta prohibición; en otros contextos su uso puede ser ventajoso pero aquí no lo permitiremos).

En el ejemplo de la clase `Estudiante`, comenzamos por sus campos. Implementemos la versión que consta de un campo entero para el DNI, otro real para la nota y otro booleano para saber si la nota está definida. Todo ellos son declarados `private` para obligar a que el acceso a los mismos desde fuera de la clase se realice mediante las operaciones correspondientes. Para limitar el conjunto de notas válidas, introducimos una constante (`final`) `MAX_NOTA`, que además será declarada `static` (ya que si no, cada objeto tendría su propia constante como atributo) y `private` (sólo se podrá consultar si se define la operación correspondiente que, por cierto, deberá ser un método estático).

```
public class Estudiante{

 private static final double MAX_NOTA =10;
 private int DNI;
 private double nota;
 private boolean tiene_nota;
 ....
```

Notad que Java no dispone de un constructor de tipos equivalente a la **tupla**. Simplemente, dentro de una clase se declaran los campos. Si necesitamos tuplas auxiliares (es decir, una tupla que forma parte de otra tupla), emplearemos clases auxiliares.

Pasemos a codificar los métodos. En las instrucciones de un método será habitual referirse al parámetro implícito: para ello Java reserva la palabra `this`. Hay que notar que ésta no es siempre obligatoria: si mencionamos el nombre de un campo de la clase, Java interpreta que nos referimos al correspondiente campo del parámetro implícito. Sin embargo, en casos como los de las operaciones `añadir_nota` y `modificar_nota` en los que existe la posibilidad de confusión entre el campo `nota` y el parámetro del mismo nombre, el uso del `this` es imprescindible. También lo es cuando se hace referencia al parámetro implícito en su conjunto, por ejemplo, si se necesita pasarlo como parámetro no implícito de alguna operación.

Por último, veréis que algunos métodos definen excepciones para controlar situaciones no previstas en la precondición. En general, no pediremos que lo hagáis así en vuestros programas, sino que bastará con comprobar que una precondición se cumple antes de usar la correspondiente operación.

Comenzamos con la creadora que genera un estudiante vacío. Las demás posibilidades quedan como ejercicio.

```
public Estudiante(){}
```

Entre las modificadoras incluimos las operaciones `crear_estudiante` y `copiar_estudiante`. Se verán otras opciones en los ejercicios.

```
public void crear_estudiante(int dni) throws Exception
/* dni>=0; si no, se produce una excepcion */
/* el parametro implicito es un estudiante con DNI=dni y sin nota */
 {
  if (dni<0) throw new Exception("DNI negativo");
  DNI = dni;
  tiene_nota=false;
 }

public void anadir_nota (double nota) throws Exception
/* nota ha de estar entre 0 y MAX_NOTA; si no, se produce una excepción,
   el parámetro implícito no debe tener nota */
/* el parámetro implícito conserva su DNI y su nota es "nota"  */
 {
  if (tiene_nota) throw new Exception("Nota ya inicializada");
  if (nota<0 || nota>MAX_NOTA) throw new Exception("Valor de nota no válido");
  this.nota=nota;
  this.tiene_nota=true;
 }

public void modificar_nota (double nota) throws Exception
/* nota ha de estar entre 0 y MAX_NOTA; si no, se produce una excepción,
   el parámetro implícito debe tener nota */
/* el parámetro implícito conserva su DNI y su nota pasa a ser nota */
 {
  if (!tiene_nota) throw new Exception("Nota no inicializada");
  if (nota<0 || nota>MAX_NOTA) throw new Exception("Valor de nota no válido");
  this.nota=nota;
  }

public void copiar_estudiante(Estudiante est)
/* cierto */
/* El parametro implicito pasa a ser una copia de est */
 {
  DNI=est.DNI;
  nota=est.nota;
  tiene_nota=est.tiene_nota;
 }
```

Pasemos a las operaciones consultoras. En este ejemplo, sólo tenemos las consultas a los valores de los campos, pero en otras situaciones podríamos necesitar cálculos más complicados.

```
/* Consultoras */

 public int consultar_DNI(){
 /* cierto */
 /* Retorna el DNI del parametro implicito */
   return DNI;
 }

 public double consultar_nota() throws Exception
 /* El parametro implicito tiene nota */
 /* Retorna la nota del parametro implicito */
 {
   if (!this.tiene_nota) throw new Exception("Nota no inicializada");
   return nota;
 }

 public boolean tiene_nota (){
 /* cierto */
 /* Retorna el parametro implicito tiene nota */
   return tiene_nota;
 }
```

Por último, las operaciones de lectura y escritura:

```
  public Estudiante(InOut canal) throws Exception
    /* Se leen pares <int dni, double nota>. Si se lee una nota no
       valida (fuera del intervalo [0..MAX_NOTA]), el estudiante resultante
       queda sin nota. */

    {
        this.crear_estudiante(canal.readint());
        double nota = canal.readdouble();
        if (nota>=0 && nota<=MAX_NOTA) this.anadir_nota(nota);
    }

  public void leer_estudiante(InOut canal) throws Exception
    /* Se leen pares <int dni, double nota>. Si se lee una nota no
       valida (fuera del intervalo [0..MAX_NOTA]), el estudiante resultante
       queda sin nota. */
```

```
    {
        this.crear_estudiante(canal.readint());
        double nota = canal.readdouble();
        if (nota>=0 && nota<=MAX_NOTA) this.anadir_nota(nota);
    }

    public void escribir_estudiante(InOut canal) throws Exception
    /* Si el estudiante no tiene nota se escribe NP */

    {
        if (this.tiene_nota())
            canal.writeln(this.consultar_DNI() + " " + this.consultar_nota());
        else canal.writeln(this.consultar_DNI() + " " + "NP");
    }
}
```