17. Otras relaciones entre objetos

Objetivos:
a) Describir otras relaciones entre clases distintas de la herencia como la asociación, la agregración, la composición o la anidación
b) Interpretar el código fuente de una aplicación Java donde aparecen clases asociadas, agregadas, compuestas y anidadas
c) Construir aplicaciones Java sencillas, convenientemente especificadas, que declaren y utilicen clases relacionadas entre sí.

Además de la relación de herencia las clases empleadas dentro de una aplicación Java, los objetos pueden estar conectados dentro de un programa con otros tipos de relaciones. Estas relaciones puede ser persistentes si establecen si la comunicación entre objetos se registra de algún modo y por tanto puede ser utilizada en cualquier momento. En el caso de relaciones no persistentes entonces el vínculo entre objetos desaparece tras ser empleado. En cualquier caso, en la mayor parte de los casos la resolución de un problema más o menos complejo exige la colaboración entre objetos. Esta colaboración se puede llevar a cabo mediante el establecimiento de relaciones entre clases (relación de herencia o generalización) o entre instancias (relación de asociación y relación todo-parte: agregación y composición). En el capitulo anterior vimos la relación de herencia. En este capítulo se mostrarán otras relaciones entre objetos.

17.1. La asociación

En una asociación, dos instancias A y B relacionadas entre sí existen de forma independiente. No hay una relación *fuerte*. La creación o desaparición de uno de ellos implica unicamente la creación o destrucción de la relación entre ellos y nunca la creación o destrucción del otro. Por ejemplo, un cliente puede tener varios pedidos de compra o ninguno.

La relación de **asociación** expresa una relación (unidireccional o bidireccional) entre las instancias a partir de las clases conectadas. El sentido en que se recorre la asociación se denomina **navegabilidad** de la asociación. Cada extremo de la asociación se caracteriza por el **rol** o papel que juega en dicha relación el objeto situado en cada extremo. La **cardinalidad** o multiplicidad es el número mínimo y máximo de instancias que pueden relacionarse con la otra instancia del extremo opuesto de la relación. Por defecto es 1. El formato en el que se especifica es (mínima..máxima). Por ejemplo:

- 1 Uno y sólo uno (por defecto)
- 0..1 Cero a uno. También (0,1)
- M..N Desde M hasta N (enteros naturales)
- 0..* Cero a muchos
- 1..* Uno a muchos (al menos uno)
- 1,5,9 Uno o cinco o nueve

17.2. La agregación y la composición

En una relación *todo-parte* una instancia forma parte de otra. En la vida real se dice que *A está compuesto de B* o que *A tiene B*. La diferencia entre asociación y relación todo-parte radica en la asimetría presente en toda relación todo-parte. En teoría se distingue entre dos tipos de relación todo-parte:

a) la **agregación** es una asociación binaria que representa una relación todo-parte (*pertenece a tiene un, es parte de*). Por ejemplo, un centro comercial tiene clientes.
b) la **composición** es una agregación *fuerte* en la que una instancia 'parte' está relacionada, como máximo, con una instancia 'todo' en un momento dado, de forma que cuando un objeto 'todo' es eliminado, también son eliminados sus objetos 'parte'. Por ejemplo: un rectángulo tiene cuatro vértices, un centro comercial está organizado mediante un conjunto de secciones de venta…

A nivel práctico se suele llamar *agregación* cuando la relación se plasma mediante referencias (lo que permite que un componente esté referenciado en más de un compuesto). Así, a nivel de implementación una agregación no se diferencia de una asociación binaria. Por ejemplo: un equipo y sus miembros. Por otro lado, se suele llamar *composición* cuando la relación se conforma en una inclusión por valor (lo que implica que un componente está como mucho en un compuesto, pero no impide que haya objetos componentes no relacionados con ningún compuesto). En este caso si se destruye el compuesto se destruyen sus componentes. Por ejemplo: un ser humano y sus miembros. Algunas relaciones pueden ser consideradas agregaciones o composiciones, en función del contexto en que se utilicen.

17.3. La dependencia o relación de uso

Una clase A usa una clase B cuando no contiene atributos de la clase B pero, o bien utiliza alguna instancia de la clase B como parámetro en alguno de sus métodos para realizar una operación, o bien accede a sus atributos (clases con métodos *amigos*).

17.4. Ejemplos de agregación o composición

Como se ha comentado anteriormente, la *agregación* o *composición* son mecanismos diferentes de la herencia que consiste en que uno o más atributos de una clase pertenecen a una o más clases previamente declaradas. Es decir, un objeto puede componerse de otros pertenecientes a otras clases.

Por ejemplo, la clase `Persona` se compone de dos variables de instancia, una de la clase `String` y otra de la clase `Fecha`:

```
/**
 *  Declaracion de la clase Persona
 *  @author A. Garcia-Beltran
 *  Ultima revision: noviembre, 2005
 */
public class Persona {
    String nombre;
    Fecha fechaNacimiento;
    public void asignaDatos(String nombre, Fecha f) {
        this.nombre = nombre;
        fechaNacimiento = f;
    }
```

```
    public String toString() {
        return nombre + " nacido el dia " + fechaNacimiento.toString();
    }
}
```

Se dice que la clase `Persona` es una agregación de las clases `String` y `Fecha`. La clase `PruebaPersona` muestra un ejemplo de uso de la clase `Persona`:

```
/**
 *  Ejemplo de uso de la clase Persona
 *  @author A. Garcia-Beltran
 *  Ultima revision: noviembre, 2005
 */
public class PruebaPersona {
    public static void main (String [] args ) {
        Persona p = new Persona();
        Fecha n = new Fecha(11,2,2002);
        p.asignaDatos("Miguel Angel Garcia", n);
        System.out.println(p.toString());
    }
}
```

La salida por pantalla al ejecutar el código anterior es:

```
$>java PruebaPersona↵

Miguel Angel Garcia nacido el dia 11/2/2002
```

17.5. Anidación o descomposición de código

La asociación de clases puede emplearse como técnica que implementa la denominada anidación o descomposición de código. En el siguiente ejemplo, la clase `Fecha` se compone de otras tres clases: `Dia`, `Mes` y `Anho`. Estas tres clases se encapsulan como clases *internas* dentro de la clase `Fecha`.

```
/**
 *  Declaracion de la clase Fecha
 *  @author A. Garcia-Beltran
 *  Ultima revision: septiembre, 2007
 */
public class Fecha {

  private Dia dd;
  private Mes mm;
  private Anho aa;
  public Fecha(int d, int m, int a) {
    this.dd = new Dia(d);
    this.mm = new Mes(m);
    this.aa = new Anho(a);
  }
  public void siguienteDia() {
    dd.siguiente();
  }

  private class Dia {
    private int d;   //   1 <= d <= mes.dias()

    public Dia(int d){
      this.d = d;
```

```
    }
    public void siguiente() {
      d = d + 1;
      verificar();
    }
    private void verificar() {
      if (d > mm.dias()) {
         d = 1;
         mm.siguiente();
      }
    }
  }    // Clase Dia

  private class Mes {
    private int m;
    private final int[] diasDelMes =
      {0,31,28,31,30,31,30,31,31,30,31,30,31};
      /*  1  2  3  4  5  6  7  8  9 10 11 12) */
    public Mes (int m) {
      this.m = m;
    }
    public int dias() {
      int result = diasDelMes[m];
      if (m==2 && aa.esBisiesto()) {
         result = result+1;
      }
      return result;
    }
    public void siguiente() {
      m = m + 1;
      verificar();
    }
    private void verificar() {
      if (m>12) {
         m=1;
         aa.siguiente();
      }
    }
  }    // Clase Mes

  private class Anho {
    private int a;
    public Anho(int a) {
      this.a = a;
    }
    public void siguiente() {
      a=a+1;
    }
    public boolean esBisiesto() {
      return ((a % 4 == 0) && (a % 100 !=0) || (a % 400 == 0));
    }
  }    // Clase Anho
}
```

Las clases `Dia`, `Mes` y `Anho` son muy simples y se distribuyen las funcionalidades de una forma fácilmente reconocible. Los identificadores facilitan la legibilidad del código lo que ayuda a que la descomposición *extrema* sea la base de una comprehensión, desarrollo y verificación independiente e incremental de cada una de las cuatro clases, así como la de sus métodos.