Paradigma
Orientado a Objetos

Módulo 15:
Elementos de Diseño.

por Fernando Dodino
Versión 2.1
Octubre 2017

Indice

1 Elementos de Diseño
- 1.1 Componentes
- 1.2 Cohesión
- 1.3 Acoplamiento

2 Resumen

1 Elementos de Diseño

A lo largo de estos capítulos hemos incorporado algunas técnicas para el diseño en el paradigma de objetos. Repasaremos a continuación algunos de esos conceptos aprendidos. Primero que todo, señalemos que diseñar es

- encontrar componentes
- sus responsabilidades
- y la relación que cada componente tiene con los demás componentes para formar un sistema

1.1 Componentes

¿De qué hablamos cuando hablamos de componente?
Existen varios puntos de vista:

- un componente puede ser un objeto, o una clase
- o también puede ser un conjunto de objetos/clases que cumplen un objetivo. Ej: un alumno con sus notas modela la carrera académica de un alumno.
- a más alto nivel, un componente puede ser un módulo o un sistema que se comunica con otro módulo o sistema. Ej: el módulo de seguimiento de carrera académica y el módulo de inscripciones a los cursos.

Cuando decimos que un componente tiene responsabilidades, ¿cómo se marcan esas responsabilidades? A partir de los métodos que implementa.

1.2 Cohesión

- Una clase es cohesiva si podemos definirle un objetivo claro y puntual.
- Un método es cohesivo si tiene un único objetivo.

Emitir una factura y calcular el total de facturación está bueno que estén en diferentes métodos. En general, **separar los métodos con efecto colateral** (emitir factura, realizar un descuento, firmar una libreta de un alumno, cambiar el sueldo básico a un empleado) **y los métodos que no tengan efecto colateral** (conocer el sueldo de un empleado, saber el promedio de notas de un alumno en finales, conocer el total de facturación de un mes para un cliente, etc.) **es una buena práctica**. También es bueno abstraer ideas que se repiten en la misma clase dándole un nombre y dejándolo en un método aparte. Así por un lado evitamos duplicar código y por otro aumenta la cohesión de un método: se concentra en hacer sólo una cosa por vez.

Veamos este ejemplo que resuelve cuándo un alumno estudia para un parcial:

```
class Alumno {
	const notas = []
	var tipoAlumno = "E"
	/*
```

```
	 * E = Estudioso, estudia siempre
	 * V = Vago, estudia solo cuando le fue mal en el último
examen,
	 * H = Hijo del rigor, estudia si el examen va a ser difícil
	 * (o sea, si el parcial tiene más de 5 preguntas)
	 */

	method rendir(_nota) {
		notas.add(_nota)
	}

	method estudiaPara(_parcial) {
		if (tipoAlumno == "E") {
			// Estudioso
			return true
		}
		if (tipoAlumno == "V") {
			// Vago
			return notas.last() < 6
		}
		if (tipoAlumno == "H") {
			// Hijo del rigor
			return parcial.cantidadPreguntas() > 5
		}
		return false
	}
}

class Parcial {
	const property cantidadPreguntas

	constructor(_cantidadPreguntas) {
		cantidadPreguntas = _cantidadPreguntas
	}

}
```

Consideremos en particular este método:

```
	method estudiaPara(_parcial) {
		if (tipoAlumno.equals("E")) {
			// Estudioso
			return true
		}
		if (tipoAlumno.equals("V")) {
			// Vago
			return notas.last() < 6
		}
```

```
            // es Hijo del rigor
            return _parcial.cantidadPreguntas() > 5
      }
```

Cada uno de los colores representa una responsabilidad que está resolviendo el alumno:

- sabe cuándo un alumno estudioso estudia
- sabe cuándo un alumno vago estudia
    - y sabe que el alumno tiene una lista de notas a la cual le puede pedir el último elemento
- sabe cuándo un alumno hijo del rigor estudia
    - y además sabe cuándo un parcial es difícil

Son en total 5 responsabilidades... ¿bastante, no? ¿Qué pasaría si pudiéramos trabajar con objetos polimórficos, que modelen el criterio de cuándo estudia un alumno? Pero además, si pensamos en mantener los objetos altamente cohesivos...

- quién sabe decirnos si un parcial es difícil: el parcial
- quién sabe decirnos la nota del último examen: un alumno
- quién sabe decirnos si un alumno estudiará para un parcial: el alumno... pero delegando al criterio que tiene ese alumno para estudiar

Construimos entonces la solución alternativa, en la que cada objeto tiene una responsabilidad clara y definida[1]:

```
class Estudioso {
      method estudiaPara(parcial, alumno) = true
}

class Vago {
      method estudiaPara(parcial, alumno) =
            alumno.notaUltimoExamen() < 6
}

class HijoDelRigor {
      method estudiaPara(parcial, alumno) = parcial.esDificil()
}

class Alumno {
      const notas = []
      var property criterioParaEstudiar = new Estudioso()

      method rendir(nota) {
            notas.add(nota)
```

[1] El ejemplo completo puede descargarse en https://github.com/wollok/cohesion-alumnos

```
	}

	method estudiaPara(parcial) =
		criterioParaEstudiar.estudiaPara(parcial, self)

	method notaUltimoExamen() = notas.last()
}

class Parcial {
	const property cantidadPreguntas

	method esDificil() = cantidadPreguntas > 5
}
```

El parcial se construye como un *value object* inmutable.

1.3 Acoplamiento

Es el grado en que los componentes de un sistema se conocen. En el ejemplo anterior, el criterio de estudio de un alumno vago podría haberse escrito de la siguiente manera[2]:

```
class Vago {
	method estudiaPara(parcial, alumno) =
		alumno.notas().last() < 6
}
```

Pero esto implica que el criterio Vago sabe cosas de más del alumno: sabe que guarda una colección ordenada de notas. Si el Alumno cambia el tipo de colección de notas de List a Set la clase Vago se ve afectada porque **deja de funcionar** (Set no entiende el mensaje last() porque sus elementos no tienen orden).

Por eso, delegando, bajamos el acoplamiento indeseado entre Alumno y Vago. Queremos que se conozcan, no buscamos acoplamiento cero, porque es necesario que ambas clases trabajen juntas para resolver el objetivo. Pero también queremos que ese acoplamiento sea bajo, mantenido a partir del envío del mensaje notaUltimoExamen(), que es responsabilidad del alumno:

```
class Vago {
	method estudiaPara(parcial, alumno) =
		alumno.notaUltimoExamen() < 6
}
```

[2] Asumiendo la existencia del getter de notas.

De la misma manera, el alumno que estudia como hijo del rigor no necesita saber que el parcial tiene una cierta cantidad de preguntas que determina si es difícil o no, simplemente debe delegar la pregunta al parcial:

```
class HijoDelRigor {
	method estudiaPara(parcial, alumno) = parcial.esDificil()
}
```

Entonces

- si la lógica que determina cuándo un parcial es difícil cambia, eso implica modificar la definición **en un solo lugar**.
- la dificultad de un parcial queda **reificado en un método,** eso facilita encontrarlo cuando lo necesite usar más adelante. Si hubiéramos optado por escribir el método de esta manera:

```
class HijoDelRigor {
	method estudiaPara(parcial, alumno) =
		parcial.cantidadPreguntas() > 5
}
```

¿De qué manera podemos asociar la pregunta del hijo del rigor al concepto “el parcial es difícil? No queda claro en ningún momento y es muy difícil que podamos reutilizar esa idea cuando lo necesitemos.

2 Resumen

En este capítulo utilizamos el ejemplo para mostrar cómo el polimorfismo y la delegación nos ayudan a mantener alta la cohesión de los objetos y con un nivel de acoplamiento lo suficientemente bajo para ayudar a que nuestro sistema sea mantenible a largo plazo.