Paradigma
Orientado a Objetos

Módulo 06: Objetos anónimos.
Repaso polimorfismo.

por Fernando Dodino
Versión 1.0
Enero 2018

Indice

1 Ejercicio: viajes en taxi

Para una aplicación de viajes en taxi queremos modelar varios choferes, pero en particular nos interesa Daniel, quien lleva a todos los pasajeros que no sean jóvenes (cada pasajero se define como joven o viejo según sus características)

2 Repaso de objetos

¿Cómo representaremos a Daniel que lleva a los pasajeros que no sean jóvenes? Mediante

- un objeto conocido Daniel
- que tiene como comportamiento un método puedeLlevarA(). ¿qué necesita ese método? ¿puede ser la edad? No, porque eso no determina si es viejo o no. Esa no es responsabilidad de Daniel, debe delegarlo en cada pasajero.

```
object daniel {
     method llevaA(pasajero) = !pasajero.esJoven()
}
```

3 Pruebas

Para verificar que la definición de Daniel es correcta, armamos los casos de prueba:

Caso de prueba	Resultado esperado
Preguntamos si Daniel lleva a un pasajero joven	No lo lleva
Preguntamos si Daniel lleva a un pasajero que no es joven	Sí, lo lleva

Bastante simple, solo que para poder implementar los tests necesito tener

- a Daniel, que es un objeto bien definido (*wko*)
- un pasajero joven
- un pasajero que no sea joven

Pero todavía no tengo aquí demasiadas definiciones sobre los pasajeros. Aquí tenemos dos opciones:

- hacer un relevamiento sobre el comportamiento esperado de los pasajeros
- o bien no atacar este problema pero sí cerrar una prueba unitaria sobre Daniel, que es lo que vamos a hacer a continuación.

4 Implementando pasajeros de prueba con objetos anónimos

Wollok permite crear objetos anónimos, que se asignan a una referencia y se pueden utilizar en ese contexto. Vemos la implementación de ambos tests:

```
import choferes.*

describe "Tests de Daniel" {

	test "Daniel no lleva a un pasajero joven" {
		const pasajeroJoven = object {
			method esJoven() = true
		}
		assert.notThat(daniel.llevaA(pasajeroJoven))
	}

	test "Daniel lleva a un pasajero que no es joven" {
		const pasajeroViejo = object {
			method esJoven() = false
		}
		assert.that(daniel.llevaA(pasajeroViejo))
	}

}
```

En cada test definimos un objeto que no tiene nombre (a diferencia de los objetos autodefinidos o *well-known objects*), asignándolo a las referencias locales pasajeroJoven y pasajeroViejo, respectivamente. Esa referencia está disponible durante la ejecución de cada uno de los tests.

¿Qué interfaz necesitamos tener para el pasajero? Solo es preciso que implemente el método esJoven, sin parámetros y que devuelva un valor booleano.

5 Evitando repetir ideas: method en un describe

El lector habrá notado que en el ejemplo anterior estamos repitiendo la misma idea al crear el objeto anónimo

```
test "Daniel no lleva a un pasajero joven" {
	const pasajeroJoven = object {
		method esJoven() = true
	}
```

```
	assert.notThat(daniel.llevaA(pasajeroJoven))
}

test "Daniel lleva a un pasajero que no es joven" {
	const pasajeroViejo = object {
		method esJoven() = false
	}
	assert.that(daniel.llevaA(pasajeroViejo))
}
```

Cuando los pasajeros tienen más rasgos diferenciales, como la edad, la altura, el peso, etc. o bien cuando aparecen nuevos choferes que también precisan tener un pasajero, crear los objetos anónimos es una tarea que se vuelve más burocrática y tediosa. Para ello vamos a abstraer la creación del pasajero en un **método** dentro del **describe**, donde vamos a permitir pasarle como parámetro si es o no joven:

```
import choferes.*

describe "Tests de Daniel" {

	method crearPasajero(joven) = object {
		method esJoven() = joven
	}

	test "Daniel no lleva a un pasajero joven" {
		const pasajeroJoven = self.crearPasajero(true)
		assert.notThat(daniel.llevaA(pasajeroJoven))
	}

	test "Daniel lleva a un pasajero que no es joven" {
		const pasajeroViejo = self.crearPasajero(false)
		assert.that(daniel.llevaA(pasajeroViejo))
	}

}
```

Aquí vemos que crearPasajero es un método que devuelve un objeto anónimo, algo que también podemos escribir así:

```
	method crearPasajero(joven) {
		return object {
			method esJoven() = joven
```

```
        }
    }
```

El método recibe como parámetro si es joven, y lo utiliza en la definición del método esJoven.

6 Convivencia de objetos anónimos y wko

¿Es necesario hacer esto en todos los casos? No, por supuesto que eso depende de las definiciones de nuestro negocio. Por ejemplo, tenemos a la pasajera Magalí que tiene 19 años, y cree que una persona es joven hasta los 30.

```
object magali {

    var edad = 19

    method cumplirAnios() {
        edad = edad + 1
    }

    method esJoven() = edad < 30
}
```

Daniel entonces no llevará a Magalí, como podemos comprobarlo en este test[1]:

```
    test "Daniel no lleva a alguien que piensa que es joven" {
        assert.notThat(daniel.llevaA(magali))
    }
```

Lo interesante es que Magalí y nuestros objetos anónimos son polimórficos en el contexto de uso de Daniel, ya que todos implementan el método esJoven().

7 Resumen

En este capítulo hemos conocido los objetos anónimos, que sirven para generar abstracciones sencillas. Esto resulta útil cuando no nos interesa reutilizarlos en otro contexto, como en el testeo unitario. Los objetos anónimos tienen atributos y comportamiento como los *wko*, y pueden ser utilizados en forma polimórfica entre sí.

[1] Fíjense que en este caso el test no dice “Daniel no lleva a Magalí”, sino qué es lo que está probando, en una forma más general y menos sujeta a cambios. Si las reglas de negocio cambian, solo debemos modificar el assert y no la descripción del test.