Introducción

En este laboratorio se presenta un ejemplo de refactorización del bad smell *switch statements* utilizando las refactorizaciones *Reemplazar código de tipo por Estado/Estrategia* (*Replace type code with State/Strategy*) y *Reemplazar los condicionales con polimorfismo (Replace conditional with polimorphism)*. Finalmente se presentará una herramienta que permite detectar algunos "malos olores" en el código y refactorizarlo de manera automática.

Objetivos

1. Saber detectar el "bad smell" *switch statements.*
2. *Refactorizar el código asociado al bad smell anterior utilizando 2 refactorizaciones.*
3. *Conocer la herramienta de refactorización JDeodorant.*

Ejemplo de aplicación de una Refactorización

Supongamos que tenemos una clase Metro con la siguiente implementación

```
public class Metro {
        private int estado;
        protected static final int PARADO = 0;
        protected static final int ARRANCANDO = 1;
        protected static final int EN_MARCHA = 2;
        protected static final int PARANDO = 3;
        public void siguienteEstado(){
                if(estado==PARADO) {
                        estado = ARRANCANDO;
                } else if(estado==ARRANCANDO) {
                        estado = EN_MARCHA;
                } else if(estado==EN_MARCHA) {
                        estado = PARANDO;
                } else if(estado==PARANDO) {
                        estado = PARADO;
                } else {
                   throw new RuntimeException("Estado desconocido");
                }
        }
        public int getEstado(){
                return estado;
        }

}
```

Esta clase dispone de un método *siguienteEstado()*, que dependiendo del estado actual pasa al siguiente estado.
Si miramos el código a simple vista, el olfato nos está indicando que no desprende un aroma refrescante. En realidad si miramos en la lista de "malos olores" ofrecida por Fowler, podemos ver que este código desprende un olor "*Switch statements*"[4]. Esto nos está indicando un problema en la duplicación de código, donde las condicionales están dispersas entre los métodos de la clase. Imagineros que queremos añadir un nuevo método a la clase que nos devuelva un String con el nombre del estado:

```
public String visualizarEstado(){
        if(estado==PARADO) {
                return "Parado";
        } else if(estado==ARRANCANDO) {
                return "Arrancando";
        } else if(estado==EN_MARCHA) {
                return "En marcha";
        } else if(estado==PARANDO) {
                return ("Parado");
        } else {
                throw new RuntimeException("Estado desconocido");
        }
}
```

Como podéis observar, cada nueva funcionalidad se debe implementar a través de una secuencia de condiciones o "switch" teniendo en cuenta los diferentes valores que pude tomar la variable "estado" (p.ej. PARADO, ARRANCANDO....). Esto hace que el código sea muy difícil de mantener. Además, cualquier cambio en el conjunto de valores que puede tomar la variable estado (p.ej. añadir el estado BAJA), implica modificar todos los métodos de la clase.

Para solucionar este problema vamos a refactorizar el código, de manera que nos convierta la secuencia de if-else en algo extensible y orientado a objetos, más fácil de mantener y de arreglar al adoptar el patrón de diseño State/Strategy. Las refactorizaciones que vamos a utilizar son "Replace Type Code with State/Strategy"[1] y “Replace conditional with polimorphism”[3].

Hay que tener cuenta que antes de realizar cualquier paso, tenemos que tener preparada una batería de pruebas que iremos ejecutaremos para ver que se pasan convenientemente. Es decir, **la signatura y el comportamiento de los métodos de la clase Metro tienen que mantenerse después de realizar la refactorización**. En el anexo de este documento tenéis disponible una batería de pruebas con la que poder empezar.

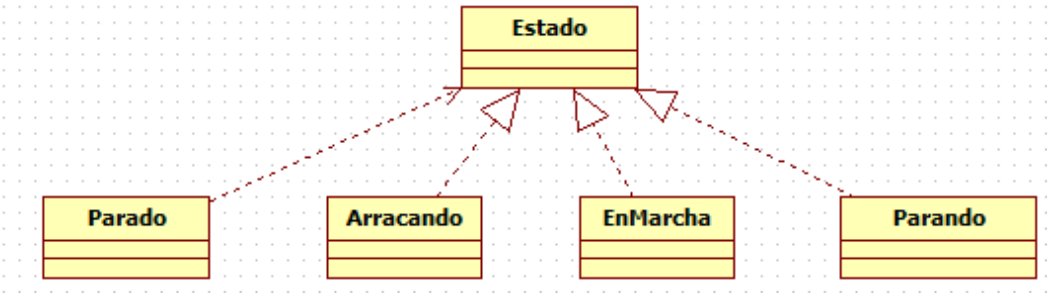
Paso 1: Encapsular la variable a refactorizar.

Tenemos que encapsular la variable que condiciona las diferentes alternativas de ejecución. En nuestro caso si observamos el código inicial, podemos observar que nos interesa la variable estado. Para ello crearemos el método setter para la variable estado, ya que el getter ya estaba disponible.

```
public void setEstado(int estado) {
        this.estado=estado;
}
```

Paso 2: Crear una jerarquía con los diferentes valores que puede tomar la variable

Vamos a crear una jerarquía de clases, definiendo una clase abstracta con el nombre de la variable tomada en la condición (en nuestro caso Estado), y una subclase por cada uno de los valores que puede tomar esta variable tal y como se muestra a continuación:



Todas estas clases extienden de la clase abstracta común Estado.

```
public abstract class Estado {}
public class Parado extends Estado {}
public class Arrancando extends Estado {}
public class EnMarcha extends Estado {}
public class Parando extends Estado {}
```

Paso 3: Crear un método en la nueva jerarquía que devuelva el valor concreto de cada estado.

Para ello creamos un método abstracto en la clase abstracta:

```
public abstract class Estado {
        public abstract int miEstado();

}
```

Y lo implementamos en cada una de las subclases:

```
public class Parado extends Estado {
        public int miEstado() {
                return Metro.PARADO;
        }
}
```

Ten en cuenta que todas las variables de la clase Metro han pasado de ser private a ser protected (las variables son visibles para todas la clases que están en el mismo paquete).

Paso 4: Modificar el tipo de la variable estado.
Vamos a reemplazar la variable estado definida en base a un entero:

```
private int estado=0;
```

Por la misma variable pero que haga referencia al Estado que corresponde a ese entero. Suponemos que estado 0 corresponde a Parado, 1 a Arrancando y así sucesivamente.

```
private Estado estado = new Parado();
```

Esto conlleva que también tengamos que modificar los métodos getter y setter correspondientes a la variable estado (pero ojo, sin cambiar su signatura).

```
public void setEstado(int estado) {
                switch (estado) {
```

```
                case ARRANCANDO:
                        this.estado = new Arrancando();
                        break;
                case PARANDO:
                        this.estado = new Parando();
                        break;
                case EN_MARCHA:
                        this.estado = new EnMarcha();
                        break;
                case PARADO:
                        this.estado = new Parado();
                        break;
                default:
                        this.estado = null;
                        break;
                }
        }
        public int getEstado() {
                return estado.miEstado();
        }
```

Del código superior se pueden recalcar 2 aspectos importantes:

1. Hemos introducido una secuencia de condiciones en el método setEstado, pero la variable estado toma valores de objetos de tipo Estado en vez de números enteros.
2. El valor del método getEstado() se obtiene a través del método miEstado() correspondiente al estado actual.

Paso 5: Cambiar los valores de las condiciones y las acciones de los métodos.
Al cambiar el tipo de la variable estado de tipo **int** a Estado, tendremos que modificar las condiciones, de forma que la condición que antes se hacia sobre un valor entero, ahora se realice sobre el estado que corresponde a ese entero. Como ejemplo si antes la condición se realizaba sobre el valor 2 (es decir, if estado==2), ahora habrá que modificar la condición por "if estado **instanteOf** EnMarcha". El código completo lo tenéis a continuación.

```
public class Metro {
        private Estado estado;
        protected static final int PARADO = 0;
        protected static final int ARRANCANDO = 1;
        protected static final int EN_MARCHA = 2;
        protected static final int PARANDO = 3;

        public void siguienteEstado() {
                if(estado instanceof Parado) {
                        setEstado(ARRANCANDO);
                } else if(estado instanceof Arrancando) {
                        setEstado(EN_MARCHA);
                } else if(estado instanceof EnMarcha) {
                        setEstado(PARANDO);
                } else if(estado instanceof Parando) {
                        setEstado(PARADO);
                } else { throw new RuntimeException("Estado desconocido");
                }
```

Si observáis el código, podéis ver que la parte acción de las condiciones también se ha

modificado . Ahora el método setEstado() encapsula cómo se realiza el cambio de estado en la clase Metro, y por tanto es transparente a los clientes cuál es su representación interna.

Paso 6: Mover las acciones asociadas a las condiciones a las subclases.
El último paso será proveer de control a las clases nuevas para que sean ellas las que decidan qué acciones ejecutar. Por tanto crearemos un método siguienteEstado en la clase abstracta Estado, y reescribiremos el contenido a dicho método en las subclases de Estado tal y como se muestra a continuación:

```
public abstract class Estado {

	public abstract void siguienteEstado(Metro metro);
}

public class Parado extends Estado {
	public void siguienteEstado(Metro metro) {
		metro.setEstado(Metro.ARRANCANDO);
	}
}

public class Arrancando extends Estado {
	public void siguienteEstado(Metro metro) {
		metro.setEstado(Metro.EN_MARCHA);
	}
}
```

De manera que el código de la clase Metro quede de la siguiente forma:

```
public class Metro {
	private Estado estado;
	static final private int PARADO = 0;
	static final private int ARRANCANDO = 1;
	static final private int EN_MARCHA = 2;
	static final private int PARANDO = 3;

	public void siguienteEstado() {
		if(estado instanceof Parado) {
			estado.siguienteEstado(this);
		} else if(estado instanceof Arrancando) {
			estado.siguienteEstado(this);
		} else if(estado instanceof EnMarcha) {
			estado.siguienteEstado(this);
		} else if(estado instanceof Parando) {
			estado.siguienteEstado(this);
		} else {
			throw new RuntimeException("Estado desconocido");
		}
```

Ya hemos finalizado con el proceso de refactorización. Aunque en esta nueva implementación no hemos eliminado las condicionales(es más, lo hemos aumentado, ya que el método setEstado() también necesita condicionantes), hemos dado un paso importante, ya

que las condiciones en vez de estar basadas en un tipo concreto (es decir, un **int**) ahora se basan en los tipos de las subclases de una jerarquía que hereda de la clase Estado con sus correspondientes métodos. En este punto podemos aplicar la segunda refactorización *Replace conditional with polimorphism*. Únicamente nos falta eliminar los "ifs", de manera que utilizando el mecanismo implícito de la ligadura dinámica, se ejecute el método correspondiente a cada estado sin la necesidad de utilizar "ifs" tal y como se muestra a continuación:

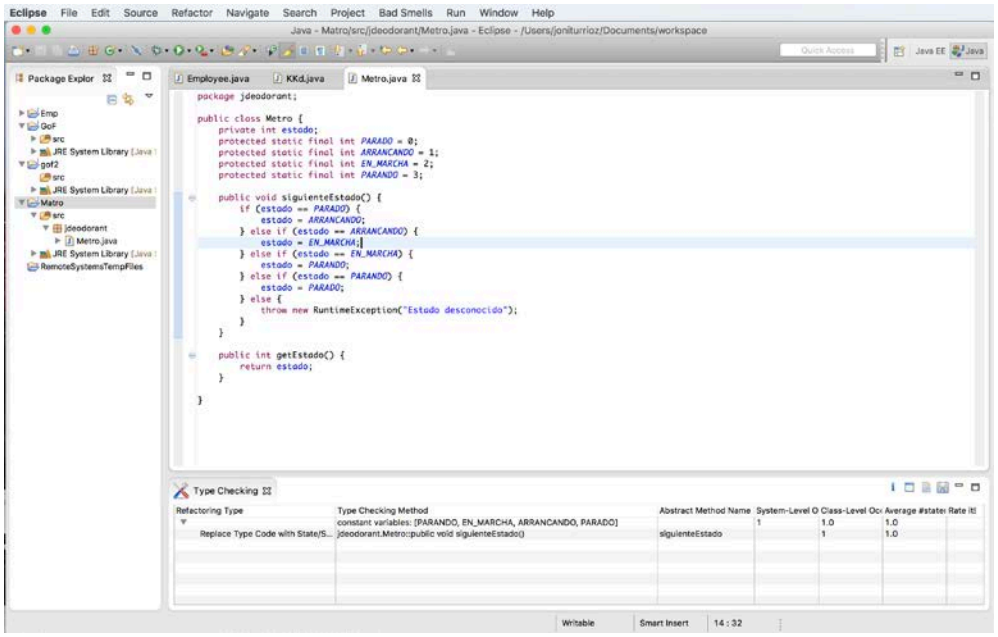
```
public class Metro {
        private Estado estado;
        public void siguienteEstado() {
                estado.siguienteEstado (this);
        }
}
```

El ejemplo presentado en este ejercicio está recogido en [2], y aunque la refactorización que propone no es exactamente la misma, sigue los mismos principios.

Uso de la herramienta JDeodorant.

JDeodorant[5] es una herramienta que detecta algunos malos olores en el código y realiza la refactorización necesaria para solucionarlo.

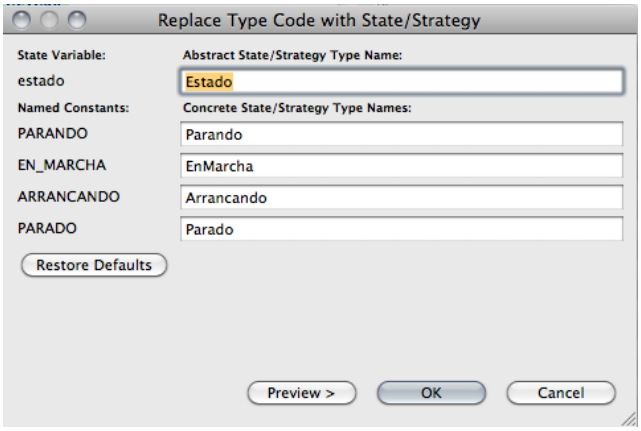
Una vez instalado reiniciamos eclipse y podemos observar que el menú superior tenemos disponible una nueva pestaña "Bad Smell". La herramienta es muy fácil de utilizar. Antes de empezar a utilizar la herramienta crear un nuevo paquete llamado jdeodorant y añadir la clase que aparece en la página 1. A continuación seleccionar el bad smell que queréis detectar, en nuestro caso siguiendo con el ejemplo de este laboratorio, seleccionaremos "Type Checking". Como consecuencia nos aparecerá una ventana asociada a ese bad smell tal y como podéis ver en la siguiente figura:



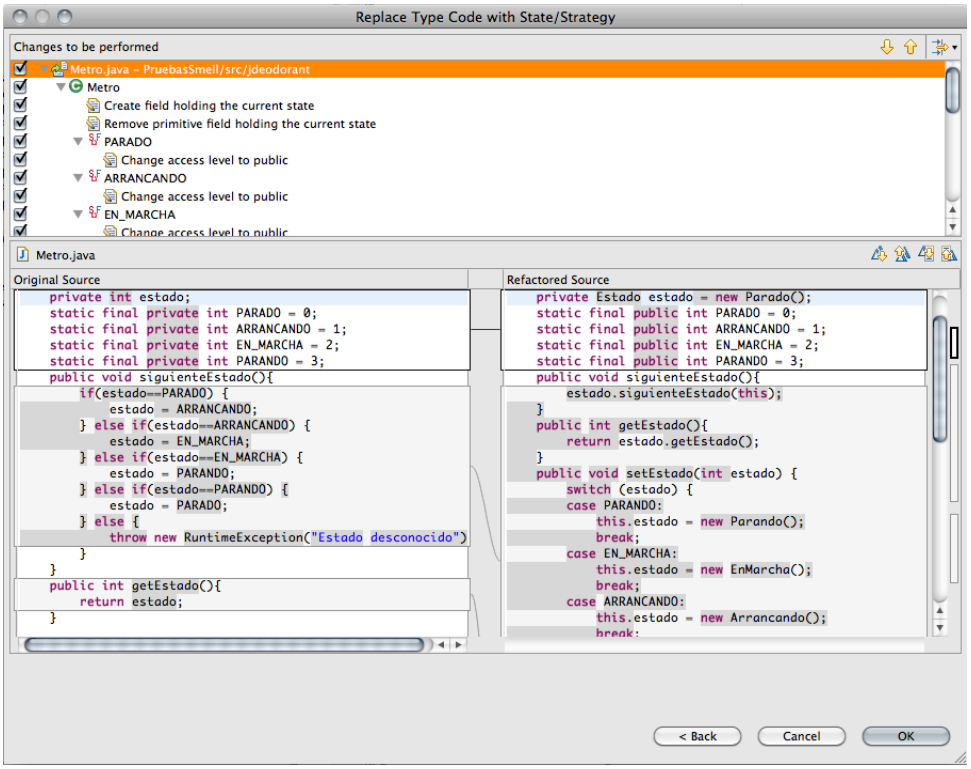
Para detectar el mal olor seleccionado, escogéis una clase o paquete de la parte izquierda, y pinchando en el botón i en la ventana inferior del Type checking os aparece el método

siguienteEstado como fragmento de código que desprenden ese olor. Si seleccionáis el método la herramienta os indicará en verde el fragmento de código donde ha detectado el mal olor.

Si queremos refactorizar el código, pinchando en el botón nos aparecerá la ventana que se muestra a continuación, donde habrá que indicar el nombre de la clase abstracta, además del nombre de todas las subclases.



Si pulsamos en el botón Preview, podremos ver en la parte superior la secuencia de pasos que se han dando en esta refactorización. Si deseamos que alguno de los pasos no se realice podemos quitar la pestaña de selección correspondiente. En la parte inferior, en la parte izquierda aparece el código inicial, y en la parte derecha aparece el código refactorizado.



Si estamos de acuerdo con la refactorización pulsamos en el bótón OK, y *voilà*, ya tenemos todas las clases que componen la refactorización. Estudiar la refactorización realizada y comprobar que habéis entendido perfectamente todos los pasos y compararla con la refactorización manual que habéis realizado.

Referencias:

[1] http://sourcemaking.com/refactoring/replace-type-code-with-state-strategy.
[2] Refactorización en la práctica. Peligros y soluciones.
http://mestreacasa.gva.es/c/document_library/get_file?fname=WillyDev_Refactorizacion%5B1%5D.pdf&uuid=e8f9dc54-fe95-463e-a3b1-bc543d3fc9f9&groupId=4700530668
[3] http://sourcemaking.com/refactoring/replace-conditional-with-polymorphism
[4] http://sourcemaking.com/refactoring/switch-statements
[5] Herramienta JDeodorant.
http://users.encs.concordia.ca/~nikolaos/jdeodorant/

Anexo: Pruebas JUnit para la clase Metro.

```
import junit.framework.TestCase;
public class MetroTest extends TestCase {
	Metro m;
	public void setUp(){
		m=new Metro();
	}
	public void testEstadoParado() {
		int esperado=0;
		assertTrue(m.getEstado()==esperado);
	}
	public void testEstadoArrancando() {
		int esperado=1;
		m.siguienteEstado();
		assertTrue(m.getEstado()==esperado);
	}
	public void testEstadoEnMarcha() {
		int esperado=2;
		m.siguienteEstado();
		m.siguienteEstado();
		assertTrue(m.getEstado()==esperado);
	}
	public void testEstadoParando() {
		int esperado=3;
		m.siguienteEstado();
		m.siguienteEstado();
		m.siguienteEstado();
		assertTrue(m.getEstado()==esperado);
	}
	public void testEstadoParadoFinal() {
		int esperado=0;
		m.siguienteEstado();
		m.siguienteEstado();
		m.siguienteEstado();
		m.siguienteEstado();
		assertTrue(m.getEstado()==esperado);
	}
}
```