Ejercicios propuestos JDeodorant:

1. ¿Qué sucede con el método visualizarEstado presentado en la página 2? ¿Cómo quedaría su refactorización en esta nueva implementación?

Código original

```
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");
        }
}
```

Vamos a aplicar 2 refactorizaciones *Replace Type Code With State/Strategy* y *Replace Conditional with Polimorphism.*

Refactorizacion 1: Replace Type Code With State/Strategy

Paso 1. Cambiar los valores de las condiciones y acciones de los métodos.

```
public String visualizarEstado(){
            if(estado instanceof Parado) {
                  return "Parado";
            } else if(estado instanceof Arrancando) {
                  return "Arrancando";
            } else if(estado instanceof EnMarcha) {
                  return "En marcha";
            } else if(estado instanceof Parando) {
                  return ("Parando");
            } else {
                  throw new RuntimeException("Estado desconocido");
            }
      }
```

Cambiamos las condiciones, para ello en vez de comparar el valor de la variable estado (un int) con cada uno de los estados (otro int) (p.ej. estado == PARADO) lo que hacemos es ver de qué clase es el objeto instanciado (p.ej. `estado instanceof Parado`).

Paso 2. Mover las acciones asociadas a las condiciones a las subclases.

Primero definimos el método en la clase abstracta:

```
public abstract class Estado {
	public abstract int miEstado();
	public abstract void siguienteEstado(Metro metro);
	public abstract String estadoToString();
}
```

Y a continuación lo implementamos el método en las subclases.

```
public class Arrancando extends Estado{
	@Override
	public String estadoToString() {
		return "Arrancando";
	}
}

public class EnMarcha extends Estado{
	@Override
	public String estadoToString() {
		return "En marcha";
	}


public class Parado extends Estado{
	@Override
	public String estadoToString() {
		return "Parado";
	}
}

public class Parando extends Estado{
	@Override
	public String estadoToString() {
		return "Parando";
	}
}
```

Ya hemos terminado la primera refactorización, ahora podemos aplicar la segunda.

Refactorizacion 2: Replace Conditional With Polimorphism

```
public String visualizarEstado(){
		return estado.estadoToString();
	}
```

Utilizando la ligadura dinamica, modificamos el código, para que el resultado del método `visualizarEstado()` dependa del `estado` en el que se encuentre el objeto Metro.

2. Queremos añadir un nuevo estado entre Parando y Parado que sea FinTrayecto. ¿Qué cambios tendrías que realizar para incorporar este nuevo estado? Realiza los cambios tanto en las clases como en el módulo de pruebas.

Paso 1: Añadir una nueva variable a la clase Metro que represente ese nuevo estado.

```
public class Metro {
	private Estado estado = new Parado();
	static final protected int PARADO = 0;
	static final protected int ARRANCANDO = 1;
	static final protected int EN_MARCHA = 2;
	static final protected int PARANDO = 3;
	static final protected int FIN_TRAYECTO = 4;
}
```

Paso 2: Añadir una nueva subclase de estado que se llame FinTrayecto.

```
public class FinTrayecto extends Estado{
	@Override
	public int miEstado() {
		return Metro.FIN_TRAYECTO;
	}
	@Override
	public void siguienteEstado(Metro metro) {
		metro.setEstado(Metro.PARADO);
	}
	@Override
	public String estadoToString() {
		return "Fin Trayecto";
	}

}
```

Paso 3: Modificar el cambio de estado de la clase Parando ya que ahora el siguiente estado será FIN_TRAYECTO y no PARADO

```
public class Parando extends Estado{
	@Override
	public void siguienteEstado(Metro metro) {
		metro.setEstado(Metro.FIN_TRAYECTO);
	}
}
```

Paso 4: Añadir un nuevo caso en el método *setEstado* para el nuevo estado que hemos definido.

```
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;
            case FIN_TRAYECTO:
                  this.estado = new FinTrayecto();
                  break;
            default:
                  this.estado = null;
                  break;
            }
}
```

Gracias a la refactorización que hemos realizado antes, ahora no tenemos que hacer ningún cambio en los métodos de `siguienteEstado()` y de `visualizarEstado()`.

```
public void siguienteEstado() {
            estado.siguienteEstado(this);
}

public String visualizarEstado(){
            return estado.estadoToString();
}
```

Ni en el getter

```
public int getEstado(){
            return estado.miEstado();
}
```

Paso 5: Modificar los casos de prueba.

Para los test, primero hemos cambiado el `testEstadoParadoFinal()` introduciéndole una llamada más al método siguiente estado ya que ahora el siguiente a PARANDO es FIN_TRAYECTO y el siguiente a este será PARADO.

```
public void testEstadoParadoFinal() {
            int esperado=0;
            m.siguienteEstado();
            m.siguienteEstado();
            m.siguienteEstado();
            m.siguienteEstado();
            m.siguienteEstado();
            assertTrue(m.getEstado()==esperado);
}
```

También hemos creado un nuevo test para el estado de FinDeTrayecto.

```
public void testEstadoFinDeTrayecto() {
            int esperado=4;
            m.siguienteEstado();
            m.siguienteEstado();
            m.siguienteEstado();
            m.siguienteEstado();
            assertTrue(m.getEstado()==esperado);
}
```

3. Dada la clase Figura del anexo, se pide (a) Crear un componente de pruebas para la clase Figura (únicamente casos válidos) y (b) realizar las refactorizaciones propuestas en este laboratorio de manera manual y utilizando *JDeodorant*. Explicar en un documento cuáles son los cambios que se han realizado de manera manual.

Casos de prueba para la clase Figura

Calcular área

```
@Test
public void testCalculateAreaSquare() {
      double areaEsperada = 4.0;
      double area = 0;
      Figura figura = new Figura();
      figura.setA(2);
      area = figura.calculateArea(Figura.SQUARE);
      assertEquals(areaEsperada, area, 0);
}

@Test
public void testCalculateAreaCircle() {
      double areaEsperada = Math.PI*2*2;
      double area = 0;
      Figura figura = new Figura();
      figura.setR(2);
      area = figura.calculateArea(Figura.CIRCLE);
      assertEquals(areaEsperada, area, 0);
}
```

```
@Test
public void testCalculateAreaRectangle() {
	double areaEsperada = 10.0;
	double area = 0;
	Figura figura = new Figura();
	figura.setA(2);
	figura.setB(5);
	area = figura.calculateArea(Figura.RECTANGLE);
	assertEquals(areaEsperada, area, 0);
}
```

Calcular Perímetro

```
@Test
public void testCalculatePerimetroSquare() {
	double perimetroEsperado = 8.0;
	double perimetro = 0;
	Figura figura = new Figura();
	figura.setA(2);
	perimetro = figura.calculatePerimeter(Figura.SQUARE);
	assertEquals(perimetroEsperado, perimetro, 0);
}
```

```
@Test
public void testCalculatePerimetroCirculo() {
	double perimetroEsperado = Math.PI*2*5;
	double perimetro = 0;
	Figura figura = new Figura();
	figura.setR(5);
	perimetro = figura.calculatePerimeter(Figura.CIRCLE);
	assertEquals(perimetroEsperado, perimetro, 0);
}

@Test
public void testCalculatePerimetroRectangle() {
	double perimetroEsperado = 10.0;
	double perimetro = 0;
	Figura figura = new Figura();
	figura.setA(2);
	figura.setB(3);
	perimetro = figura.calculatePerimeter(Figura.RECTANGLE);
	assertEquals(perimetroEsperado, perimetro, 0);
}
```

Refactorización

Calcular área

Código original

```
public double calculateArea(int shape) {
	double area = 0;
	switch(shape) {
		case SQUARE:
		area = a * a;
		break;
	case RECTANGLE:
		area = a * b;
		break;
	case CIRCLE:
		area = Math.PI * r * r;
		break;
	}
	return area;
}
```

Código refactorizado

```
public double calculateArea(int shape) {
		return getShape(shape).area(this);
	}
```

Paso 1. Crear una jerarquía con los diferentes valores que pueda tomar la variable.

```
public abstract class Shape {}
public class Square extends Shape{}
public class Rectangle extends Shape{}
public class Circle extends Shape{}
```

Paso 2. Crear los getters y setters para cada uno de los atributos de la clase Figura.

```
public double getA(){
	return a;
}

public double getB(){
	return b;
}

public double getR(){
	return r;
}
public void setA(double a){
	this.a = a;
}

public void setB(double b){
	this.b = b;
}
```

```
public void setR(double r){
	this.r = r;
}
```

Y también un getter que nos devuelva el tipo de figura que es

```
public Shape getShape(int shape){
		switch(shape){
		case SQUARE:
			return new Square();
		case RECTANGLE:
			return new Rectangle();
		case CIRCLE:
			return new Circle();
		default:
			return null;
		}
}
```

Paso 3. Crear un método en la jerarquía que devuelva el área de las diferentes figuras.

Primero creamos el método en la clase abstracta

```
public abstract class Shape {
	public abstract double area(Figura figura);
}
```

Y luego lo implementamos en cada una de las subclases

```
public class Square extends Shape{
	@Override
	public double area(Figura figura) {
		return figura.getA()*figura.getA();
	}
}

public class Rectangle extends Shape{
	@Override
	public double area(Figura figura) {
		return figura.getA()*figura.getB();
	}
}

public class Circle extends Shape{
	@Override
	public double area(Figura figura) {
		return Math.PI*Math.pow(figura.getR(), 2);
	}
}
```

Paso 4. Aplicar la refactorización replace conditional with polimorphism.

De esta manera el código del método calcular área queda así

```
public double calculateArea(int shape) {
		return getShape(shape).area(this);
}
```

Calcular perímetro

Código original

```
public double calculatePerimeter(int shape) {
	double perimeter = 0;
	switch(shape) {
		case SQUARE:
			perimeter = 4 * a;
			break;
		case RECTANGLE:
			perimeter = 2 * (a + b);
			break;
		case CIRCLE:
			perimeter = 2 * Math.PI * r;
			break;
		}
	return perimeter;
}
```

Código refactorizado

```
public double calculatePerimeter(int shape) {
	return getShape(shape).perimetro(this);
}
```

Para la refactorización de este método algunos de los pasos ya están hechos con el método anterior.

Paso 1. Crear un método en la jerarquía que devuelva el perímetro de las diferentes figuras.

Añadimos un método en la clase abstracta para calcular el perímetro

```
public abstract class Shape {
	public abstract double area(Figura figura);
	public abstract double perimetro(Figura figura);
}
```

Y luego implementamos ese método en las subclases

```
public class Circle extends Shape{
	@Override
	public double perimetro(Figura figura) {
		return 2*Math.PI*figura.getR();
	}
}

public class Square extends Shape{
	@Override
	public double perimetro(Figura figura) {
	return 4*figura.getA();
	}
}
```

```
public class Rectangle extends Shape{
	@Override
	public double perimetro(Figura figura) {
		return 2* (figura.getA()+figura.getB());
	}
}
```

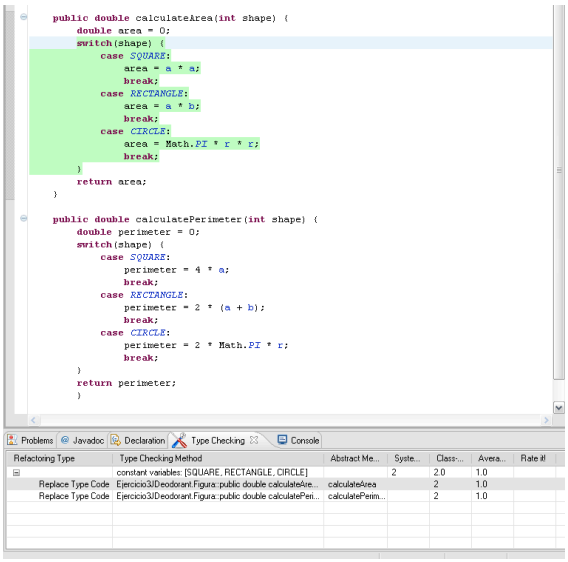
Paso 2. Aplicar la refactorización replace conditional with polimorphism.

```
public double calculatePerimeter(int shape) {
		return getShape(shape).perimetro(this);
}
```

Jdeodorant

Calcular área

Si utilizamos la herramienta JDeodorant sobre este método, podemos ver que nos dice que desprende algunos malos olores, y nos los marca en verde



Cuando indicamos a la herramienta JDeodorant que aplique la refactorización, utiliza la refactorización *Repalce type code with State/Strategy* y podemos ver como ha hecho practicamente lo mismo que hemos hecho antes manualmente.

El método calcular área ha quedado así:

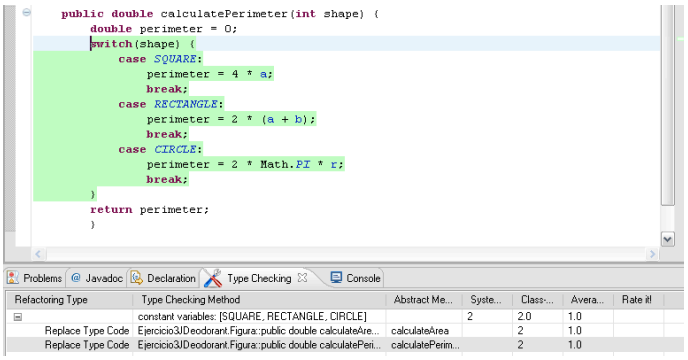
```
public double calculateArea(int shape) {
		double area = 0;
		area = getShapeObject(shape).calculateArea(area, this);
		return area;
}
```

Y ha creado una jerarquía de objetos con una clase abstracta Shape y sus subclases que serán las que implementen el método abstracto de la clase Shape calcular área

```
public abstract class Shape {
     public abstract double calculateArea(double area, Figura figura);
}
```

```
public class Circle extends Shape {
	public double calculateArea(double area, Figura figura) {
		area = Math.PI * figura.getR() * figura.getR();
		return area;
	}
}

public class Rectangle extends Shape {
	public double calculateArea(double area, Figura figura) {
		area = figura.getA() * figura.getB();
		return area;
	}
}

public class Square extends Shape {
	public double calculateArea(double area, Figura figura) {
		area = figura.getA() * figura.getA();
		return area;
	}
}
```

Calcular perímetro



Para la refactorización del método calcular perímetro aplica la misma técnica que en el anterior, pero esta vez ya están la clase abstract y sus subclases creadas, por lo que no deja crearlas con el mismo nombre otra vez.
Si aplicamos la técnica sobre este método el resultado es el siguiente:

```
public double calculatePerimeter(int shape) {
	double perimeter = 0;
	perimeter = getShapeObject(shape).calculatePerimeter(perimeter,
this);
	return perimeter;
}
```

Y la jerarquía de clases que crea quedaría así:

```
public abstract class Shape {
	public abstract double calculatePerimeter(double perimeter,
Figura figura);
}
```

```
public class Circle extends Shape {
	public double calculatePerimeter(double perimeter, Figura
figura) {
		perimeter = 2 * Math.PI * figura.getR();
		return perimeter;
	}
}
public class Rectangle extends Shape {
	public double calculatePerimeter(double perimeter, Figura
figura) {
		perimeter = 2 * (figura.getA() + figura.getB());
		return perimeter;
	}
}

public class Square extends Shape {
	public double calculatePerimeter(double perimeter, Figura
figura) {
		perimeter = 4 * figura.getA();
		return perimeter;
	}
}
```

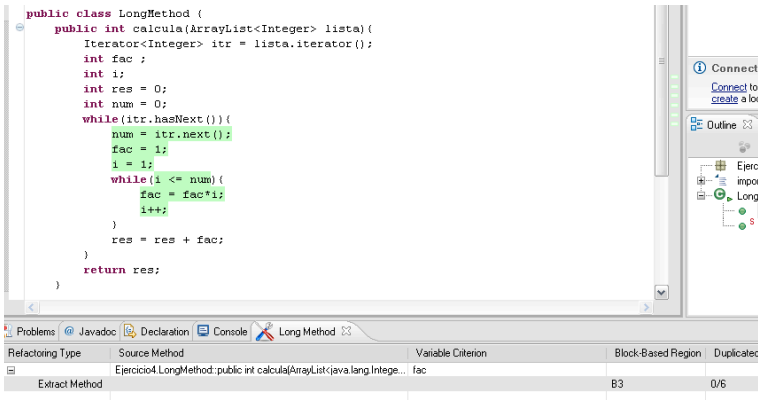
4. Utilizando la herramienta *JDeodorant* buscar un ejemplo de bad smell *God class*, *Long method* y *Feature envy*, realizar la refactorización con *JDeodorant*, y presentar en un documento el código inicial, el código refactorizado y una explicación indicando cuáles son los cambios que se han realizado.

Refactorización con JDeodorant

Long Method

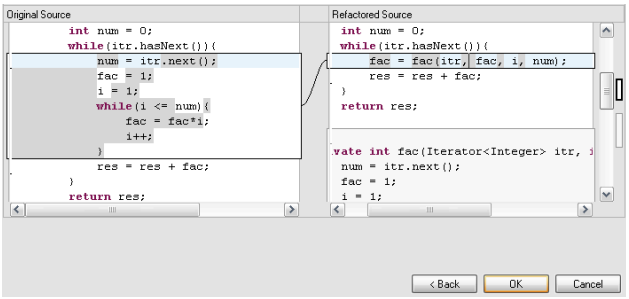
Este BadSmell se produce cuando un método tiene demasiadas líneas de código, dificultando así su manejo y su entendimiento.
El siguiente programa calcula el factorial de cada uno de los números de la lista que nos pasan como parámetro y devuelve la suma de todos.

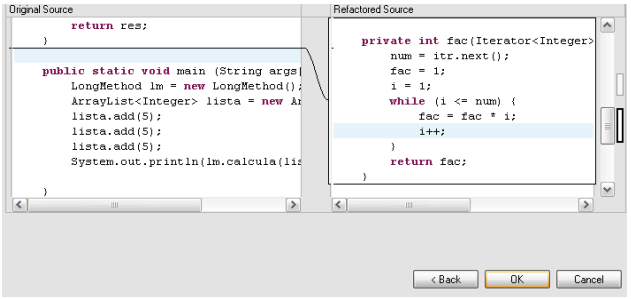
```
public class LongMethod {
public int calcula(ArrayList<Integer> lista){
	Iterator<Integer> itr = lista.iterator();
	int fac ;
	int i;
	int res = 0;
	int num = 0;
	while(itr.hasNext()){
		num = itr.next();
		fac = 1;
		i = 1;
		while(i <= num){
			fac = fac*i;
			i++;
		}
		res = res + fac;
	}
	return res;
}
```



Sí buscamos en la pestaña de BadSmells, el BadSmell Long Method, podemos observar que nos ha marcado en verde donde ha identificado este mal olor.

Si le decimos que aplique la refactorización, podemos ver qué cambios hará pulsando a *Preview*.






```
public int calcula(ArrayList<Integer> lista){
	Iterator<Integer> itr = lista.iterator();
	int fac = 0;
	int i = 0;
	int res = 0;
	int num = 0;
	while(itr.hasNext()){
		fac = fac(itr, fac, i, num);
		res = res + fac;
	}
	return res;
}
```

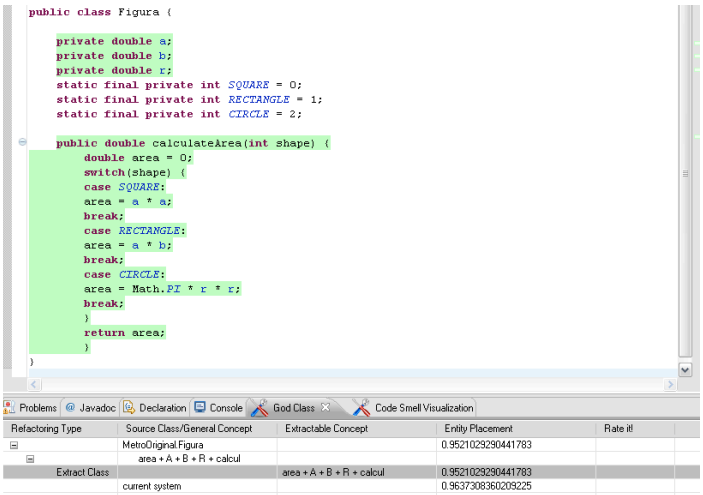
Podemos ver cómo en el método original nos ha extraído toda la parte que se correspondía al cálculo del factorial y nos ha creado un nuevo método con éllo (ha aplicado la refactorización *Extract Method*). Este es el nuevo método que ha creado:

```
private int fac(Iterator<Integer> itr, int fac, int i, int num) {
	num = itr.next();
	fac = 1;
	i = 1;
	while (i <= num) {
		fac = fac * i;
		i++;
	}
	return fac;
}
```

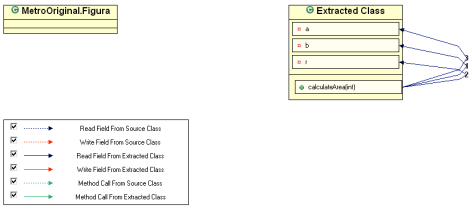
God Class

Para este ejemplo usaremos el código original de la clase Figura y su método `calculateArea(int shape)`.

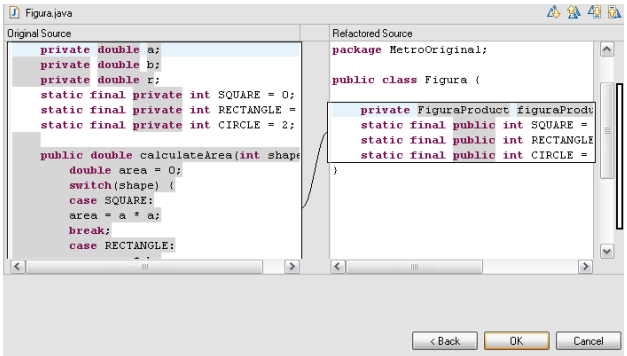
```
public class Figura {
private double a;
private double b;
private double r;
static final private int SQUARE = 0;
static final private int RECTANGLE = 1;
static final private int CIRCLE = 2;

public double calculateArea(int shape) {
      double area = 0;
      switch(shape) {
      case SQUARE:
      area = a * a;
      break;
      case RECTANGLE:
      area = a * b;
      break;
      case CIRCLE:
      area = Math.PI * r * r;
      break;
      }
      return area;
      }
}
```



La herramienta de JDeodorant nos muestra también mediante un diagrama de clases las refactorizaciones que hará. En este caso para refactorizar este código, lo que hará será aplicar la refactorización Extract class.



Preview:



Al aplicar *Extract Class* lo que ha hecho ha sido crear una nueva clase con algunos de los atributos, y ha puesto en esa clase los métodos que correspondían a esos atributos.

La clase original ahora queda así:

```
public class Figura {

	private FiguraProduct figuraProduct = new FiguraProduct();
	static final public int SQUARE = 0;
	static final public int RECTANGLE = 1;
	static final public int CIRCLE = 2;
}
```

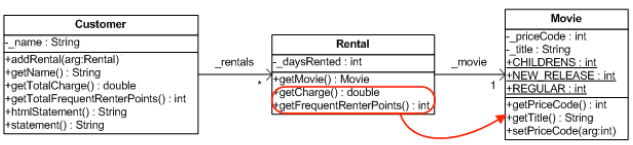
Y la nueva clase que ha creado es esta:

```
public class FiguraProduct {
	private double a;
	private double b;
	private double r;

	public double getA() {
	public double calculateArea(int shape) {
		double area = 0;
		switch (shape) {
		case Figura.SQUARE:
			area = a * a;
			break;
		case Figura.RECTANGLE:
			area = a * b;
			break;
		case Figura.CIRCLE:
			area = Math.PI * r * r;
			break;
		}
		return area;
	}
}
```

Podemos ver cómo en la nueva clase ha introducido los atributos a b y r y el método de `calculateArea`. Para poder extraer a esa clase el método `calculateArea` se han tenido que modificar la visibilidad de los atributos SQUARE, CIRCLE y RECTANGLE, ya que sus valores se utilizaban en el método. También se ha creado en la nueva clase los getters y setters correspondientes a sus atributos a b y r.

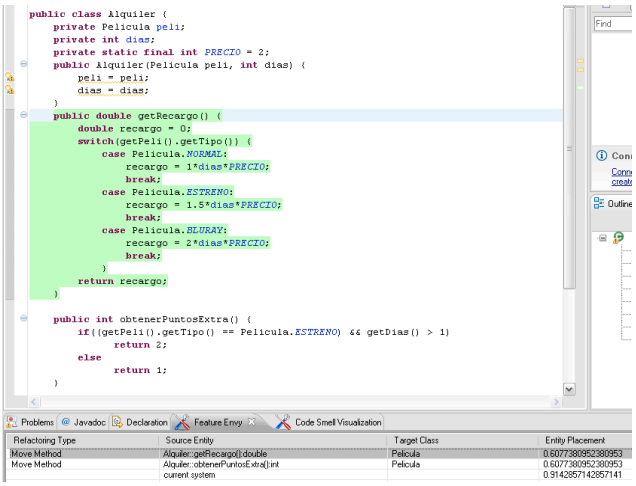
Feature Envy

Este BadSmell se produce cuando un método utiliza más métodos de otra clase que la clase en la que está definido, (está más interesado en los métodos de la otra clase que en los de su propia clase). Para refactorizar este BadSmell, se suele aplicar la refactorización Move Method.
Tenemos las clases *Alquiler* y *Película*.
En la clase alquiler tenemos los métodos *getRecargo()* y *obtenerPuntosExtra()*.
Estos dos métodos realizan muchas llamadas a métodos de la clase Película por lo que convendría que estos métodos estuviesen ahí.



Código Original

```
public class Alquiler {
private Pelicula peli;
private int dias;
private static final int PRECIO = 2;
public Alquiler(Pelicula peli, int dias)
{
      peli = peli;
      dias = dias;
}
public double getRecargo() {
   double recargo = 0;
   switch(getPeli().getTipo()) {
            case Pelicula.NORMAL:
                 recargo = 1*dias*PRECIO;
                break;
            case Pelicula.ESTRENO:
                  recargo = 1.5*dias*PRECIO;
                break;
            case Pelicula.BLURAY:
                  recargo = 2*dias*PRECIO;
                break;
            }
        return recargo;
    }
```



Código despues de aplicar la refactorización

```
public class Alquiler {
      private Pelicula peli;
      private int dias;
      public static final int PRECIO = 2;
      public Alquiler(Pelicula peli, int dias) {
            peli = peli;
            dias = dias;
      }
```

Podemos ver que ha movido el método de la clase *Alquiler* y lo ha puesto en la clase *Película.*

```
public class Pelicula {
      public static final int ESTRENO = 2;
      public static final int NORMAL = 1;
      public static final int BLURAY = 3;
      private String titulo;
      private int tipo;
      public double getRecargo(int dias) {
            double recargo = 0;
            switch (getTipo()) {
            case Pelicula.NORMAL:
                  recargo = 1 * dias * Alquiler.PRECIO;
                  break;
            case Pelicula.ESTRENO:
                  recargo = 1.5 * dias * Alquiler.PRECIO;
                  break;
            case Pelicula.BLURAY:
                  recargo = 2 * dias * Alquiler.PRECIO;
                  break;
            }
            return recargo;
      }
```

En el preview que nos ofrece eclipse podemos ver los cambios que va a realiza, primero va a cambiar la visibilidad del atributo precio a *public*, para que así sea accesible desde la clase *Película*, y luego va a mover el método *getRecargo()* a la clase Película.