**Xavier Ledesma Pons**

**Ejercicio 1**

Un **code smell** es un indicador de que algo en el nuestro código va a producir un problema en el sistema en el futuro o simplemente rompe con los fundamentos del estándar de desarrollo de software y, por tanto, hace decrecer la calidad del código. Un **code smell** no tiene porque ser un bug o un error, simplemente puede ser una mala practica en el desarrollo que hace que no se cumplan ciertos valores de buena programación.

Los **code smell**, se pueden clasificar según las siguientes categorías:

* **Bloaters**: este **code smell** hace referencia a métodos y clases que han crecido substancialmente con el tiempo, se vuelven imposibles de mantener y nadie hace nada para erradicar el problema.
  + **Long method**: métodos que contienen demasiadas líneas de código.
  + **Large class**: clases que contienen demasiados atributos, métodos y líneas de código.
  + **Primitive obsession**: uso de primitivas en vez de pequeños objetos para tareas simples.
  + **Long parameter list**: las funciones reciben más de 3 o 4 parámetros.
  + **Data clumps:** diferentes partes del código contienen grupos de variables idénticos.
* **Object orientantion abusers:** hace referencia a aplicaciones incorrectas de los principios de la programación orientada a objetos.
  + **Switch statements:** switchs complejos o if’s complejos.
  + **Temporary field:** campos que tienen campos temporales y habitualmente están vacíos.
  + **Refused bequest:** las clases que heredan y no usan todos los métodos de la superclase están fuera de lugar y los métodos que no se usan deberían ser redefinidos.
  + **Alternative clases with different interfaces:** dos clases que tienen los mismos métodos, pero estos se llaman diferente.
* **Change preventers:** hace referencia a momentos en los que queremos modificar una parte del código y debemos hacer cambios en otras partes para que este funcione.
  + **Divergent change:** cuando se deben cambiar diversos métodos de una clase para realizar cambios en esa clase.
  + **Shotgun surgery:** cuando al hacer cambios en una clase, se deben modificar pequeñas cosas de diferentes clases.
  + **Parallel inheritance hierarchies:** cuando al crear una subclase de una clase se debe crear otra para otra clase.
* **Dispensables:** hace referencia a código no necesario, el cual su ausencia hace que el código sea mas limpio y fácil de entender.
  + **Comments:** un método se llena de comentarios explicativos.
  + **Duplicate code:** dos fragmentos de código idénticos.
  + **Lazy class:** si una clase no realiza una labor importante se debe eliminar.
  + **Data class:** clases que son contenedores de variables con sus respectivos getters/setters.
  + **Dead code:** variables o parámetros que ya no se usan.
  + **Speculative generality:** existe una clase o método que ya no se usa.
* **Couplers:** hace referencia a un acoplamiento excesivo entre clases
  + **Feature envy:** un método acede a datos de otro objeto más que a sus propios datos.
  + **Inappropiate intimacy:** una clase usa los métodos y parámetros internos de otra clase.
  + **Message chains:** existe una serie de llamadas que se asemejan a()->b()->c()->d().
  + **Middle man:** si una clase solo realiza una acción que es delegada a una segunda clase, se puede eliminar.

**Ficha de 5 code smells**

**Nombre:** Long parameter list

**Problema:** Función que recibe más de 3-4 parámetros.

**Técnica de refactoring:** pasar un objeto en vez de sus propiedades

**Ejemplo:**

**Antes:**

**int** x = point.getX();

**int** y = point.getY();

window.setLocation(x, y);

**Después:**

window.setLocation(point);

**Nombre:** Long method

**Problema:** métodos que contienen demasiadas líneas de código.

**Técnica de refactoring:** extraer parte del método en otra función.

**Ejemplo:**

**Antes:**

**void** **printOwing**() {

printBanner();

*// print details*

System.out.println("name: " + name);

System.out.println("amount: " + getOutstanding());

}

**Después:**

**void** **printOwing**() {

printBanner();

printDetails(getOutstanding());

}

**void** **printDetails**(**double** outstanding) {

System.out.println("name: " + name);

System.out.println("amount: " + outstanding);

}

**Nombre:** Code comments

**Problema:** un metódo se llena de comentarios explicativos.

**Técnica de refactoring:** usar assert en vez de comentarios

**Ejemplo:**

**Antes:**

*// value must not be negative*

**public** **double** **squareRoot**(**double** value) {

*// ...*

}

**Despues:**

**public** **double** **squareRoot**(**double** value) {

Assert.isTrue(value > 0);

*// ...*

}

**Nombre:** Dead code

**Problema:** variables o parámetros que ya no se usan.

**Técnica de refactoring:** eliminar el código que ya no se usa

**Ejemplo:**

**Antes:**

**public** **double** **squareRoot**(**double** value) {

*// ...*

}

**public** **double** **add**(**double** value1, doublé value2) {

*// ...*

}

squareRoot(5);

**Despues:**

**public** **double** **squareRoot**(**double** value) {

*// ...*

}

squareRoot(5);

**Nombre:** Primitive obsession

**Problema:** uso de primitivas en vez de pequeños objetos para tareas simples.

**Técnica de refactoring:** reemplazar tipo de datos por objetos

**Ejemplo:**

**Antes:**

**public** **class** **Person** {

**private** String firstName;

**private** String lastName;

**private** **boolean** isFemale;

}

**Despues:**

**public** **class** **Person** {

**private** Name name;

**private** Gender gender;

}

**public** **class** **Name** {

**private** String firstName;

**private** String lastName;

}

**public** **enum** **Gender** {

FEMALE, MALE;

}

**Ejercicio 2**

**Nombre:** Extract method

**Cuando usarla:** cuando un método tiene muchas líneas y es complejo de mantenerlo y entender que hace.

**Ejemplo:**

**Antes:**

printOwing(): **void** {

printBanner();

// Print details.

console.log("name: " + name);

console.log("amount: " + getOutstanding());

}

**Después:**

printOwing(): **void** {

printBanner();

printDetails(getOutstanding());

}

printDetails(outstanding: **number**): **void** {

console.log("name: " + name);

console.log("amount: " + outstanding);

}

**Nombre:** Extract variable

**Cuando usarla:** cuando tenemos una expresión compleja y queremos que sea entendible.

**Ejemplo:**

**Antes:**

renderBanner(): **void** {

if ((platform.toUpperCase().indexOf("MAC") > -1) &&

(browser.toUpperCase().indexOf("IE") > -1) &&

wasInitialized() && resize > 0 )

{

// do something

}

}

**Después:**

renderBanner(): **void** {

const isMacOs = platform.toUpperCase().indexOf("MAC") > -1;

**const** isIE = browser.toUpperCase().indexOf("IE") > -1;

**const** wasResized = resize > 0;

**if** (isMacOs && isIE && wasInitialized() && wasResized) {

// do something

}

}

**Nombre:** Split temporary variable

**Cuando usarla:** existe una variable local que se usa para guardar diversos valores intermedios

**Ejemplo:**

**Antes:**

**let** **temp** = 2 \* (height + width);

console.log(temp);

temp = height \* width;

console.log(temp);

**Después:**

**const** **perimeter** = 2 \* (height + width);

console.log(perimeter);

**const** **area** = height \* width;

console.log(area);

**Nombre:** Replace method with method object

**Cuando usarla:** existe un largo método en el cual las variables locales están entrelazadas y no se puede aplicar el método de extracción

**Ejemplo:**

**Antes:**

**class** **Order** {

// ...

price(): **number** {

**let** **primaryBasePrice**;

**let** **secondaryBasePrice**;

**let** **tertiaryBasePrice**;

// Perform long computation.

}

}

**Después:**

**class** **Order** {

// ...

price(): **number** {

**return** **new** PriceCalculator(**this**).compute();

}

}

**class** **PriceCalculator** {

**private** \_primaryBasePrice: **number**;

**private** \_secondaryBasePrice: **number**;

**private** \_tertiaryBasePrice: **number**;

constructor(**order**: **Order**) {

// Copy relevant information from the

// order object.

}

compute(): **number** {

// Perform long computation.

}

}

**Nombre:** Inline method

**Cuando usarla:** cuando el cuerpo de la función es más obvio que el propio método

**Ejemplo:**

**Antes:**

**class** **PizzaDelivery** {

// ...

getRating(): **number** {

**return** moreThanFiveLateDeliveries() ? 2 : 1;

}

moreThanFiveLateDeliveries(): **boolean** {

**return** numberOfLateDeliveries > 5;

}

}

**Después:**

**class** **PizzaDelivery** {

// ...

getRating(): **number** {

**return** numberOfLateDeliveries > 5 ? 2 : 1;

}

}