Codigo Smell
y
Técnicas de refactorización

Indice

- Codigo smell
- Niveles de codigo smell
  - Aplicación
  - Clases
  - Metodos
- Cuando refactorizar
- Como refactorizar
  - Tecnicas
    - Red-Green-Refactor
    - Composing Method
    - Refactoring by Abstraction
- Cuando no refactorizar

Codigo smell

Code smell no son bugs o errores. Más bien, se trata de violaciones absolutas de los fundamentos del desarrollo de software que disminuyen la calidad del código e incrementan la deuda técnica. Ademas no significa ciertamente que el software no funcione.

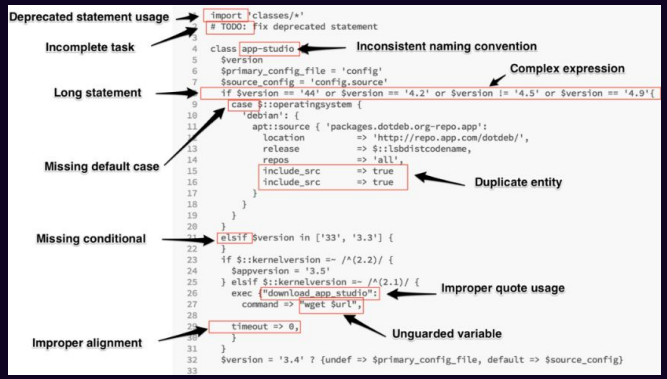


Imagen ilustrativa

Doctrina del desarrollador

Niveles de codigo smell

Nivel Aplicación

- **Duplicación de código (Duplicate Code)**
  Código similar en más de una ubicación

- **Cirugía de escopeta (Shotgun Surgery)**
  Un cambio requiere alterar varias otras clases

- **Complejidad en el diseño (Contrived Complexity)**
  Usar patrones de diseño complejos donde se podría usar un diseño más simple y sin complicaciones

Nivel Clases

- **Clases largas (Large Class)**
  La clase intenta hacer demasiado y tiene demasiadas variables de instancia

- **Código divergente (Divergent Code)**
  Una clase que sufre muchos tipos de cambios para provocar un cambio en un sistema

- **Grupo de datos (Data Clump)**
  Montones de datos que se agrupan en muchos lugares

- **Intimidad inapropiada (Inappropriate Intimacy)**
  Una clase que depende de los detalles de implementación de otra clase

- **Complejidad ciclomática (Cyclomatic Complexity)**
  Clase con demasiadas ramas y bucles

- **Legado rechazado (Refused Bequest)**
  Subclase que no utiliza métodos y datos de superclase

Nivel Metodos

- **Metodos largos (Long Method)**
  Procedimientos largos y difíciles de entender

- **Generalidad especulativa (Speculative Generality)**
  Métodos cuyos únicos usuarios son casos de prueba

- **Cadena de mensajes (Message Chains)**
  Método que llama a un método diferente que llama a un método diferente que llama a un método diferente... y así sucesivamente

- **Demasiados parametros (Too Many Parameters)**
  Una lista muy larga de parámetros

- **Soluciones raras (Oddball Solutions)**
  Cuando se utilizan varios métodos para resolver el mismo problema en un programa, lo que crea inconsistencia

- **Retorno excesivo (Excessive Returner)**
  Un método que devuelve más datos de los que necesita la persona que lo llama.

Cuando refactorizar

Cuando refactorizar

El mejor momento para considerar la refactorización es antes de agregar actualizaciones o nuevas funciones al código existente.

Limpiar el código actual antes de agregar una nueva linea de codigo no solo mejorará la calidad del producto en sí, sino que también facilitará a los futuros desarrolladores (tu) la construcción del mismo.

Como refactorizar

- Tecnicas

Como refactorizar

A la hora de refactorizar tenemos diversas técnicas para hacerlo, pero no debemos olvidar algo muy importante, la refactorización de código no debería cambiar nada sobre el comportamiento del producto.

Tecnicas

- Red-Green-Refactor
- Refactoring by Abstraction
- Composing Method

Red-Green-Refactor

Red-Green-Refactor

¿Que es Red-Green-Refactor?

El desarrollo basado en pruebas (TDD) es un enfoque para el desarrollo de software en el que primero se escriben pruebas y luego se utilizan esas pruebas para impulsar el diseño y el desarrollo de su aplicación de software. En este caso veremos el approach en base a Red-Green-Refactor

Piensa en lo que quieres desarrollar

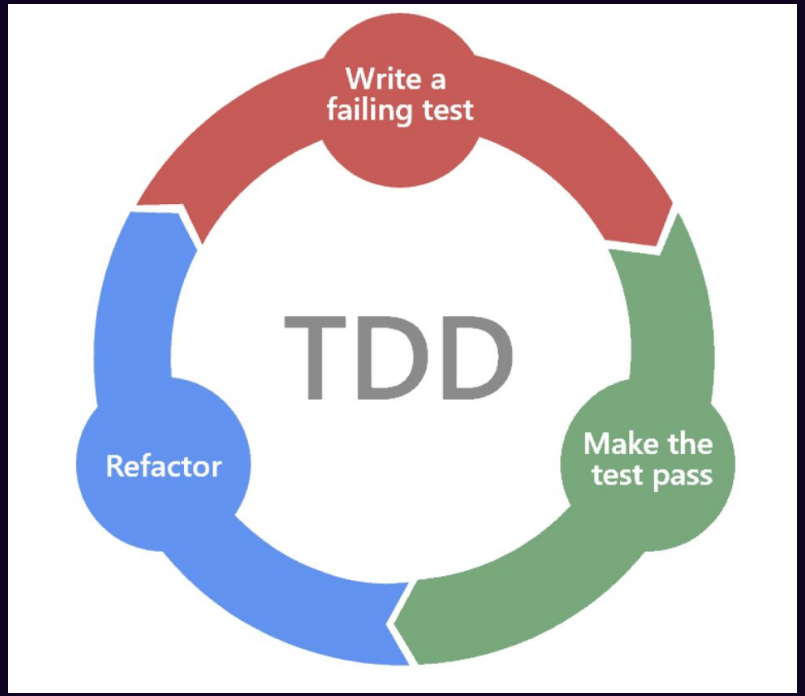
Piense en cómo aprobar sus pruebas

Piense en cómo mejorar su implementación existente

Leyes de TDD

- No escribirás código de producción sin antes escribir un test que falle.
- No escribirás más de un test unitario suficiente para fallar (y no compilar es fallar).
- No escribirás más código del necesario para hacer pasar el test.

Red-Green-Refactor



¿Como implementar Red-Green-Refactor?

- Red: Primero escribir un test que falle, es decir, tenemos que realizar el test antes de escribir la implementación. Normalmente se suelen utilizar test unitarios, aunque en algunos contextos puede tener sentido hacer TDD con test de integración

- Green: Una vez creado el test que falla, implementaremos el mínimo código necesario para que el test pase

- Refactor: Por último, tras conseguir que nuestro código pase el test, debemos examinarlo para ver si hay alguna mejora que podamos realizar

Composing Method

Composing Method

¿Que es Composing Method?

En la mayoría de los casos, los métodos excesivamente largos son la raíz de todos los males. Las implementaciones dentro de estos métodos ocultan la lógica de ejecución y hacen que el método sea extremadamente difícil de entender, y aún más difícil de cambiar.

Categorias

- Extract Method
- Inline Method
- Extract Variable
- Split Temporary Variable
- Remove Assignments to Parameters

Composing Method - Extract Method

¿En que consiste Extract Method?

La extracción implica dividir el código en trozos más pequeños para encontrar y "extraer" la fragmentación. Luego, el código fragmentado se mueve a un método separado y se reemplaza con una llamada a este nuevo método

Problema

Tenemos código duplicado en diversos métodos que se puede agrupar en un solo método

Solucion

Mover este código a un nuevo método independiente y reemplazar el código anterior con una llamada al nuevo método

```
class Bad {
    // Bad implementation
    printOwing() {
        printBanner();

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


class Good {
    // Good implementation
    printOwing() {
        printBanner();
        printDetails(getOutstanding()); // Call new method
    }

    printDetails(outstanding) { // Separate implementation in a new method
        console.log("lastName: " + lastName);
        console.log("amount: " + outstanding);
    }
}
```

Composing Method - Inline Method

¿En que consiste Inline Method?

La refactorización en línea es una forma de reducir la cantidad de métodos innecesarios mientras se simplifica el código. Al encontrar todas las llamadas al método y reemplazarlas con el contenido del método, el método se puede eliminar

Problema

Cuando el cuerpo de un método es más obvio que el método en sí, utilice esta técnica

Solucion

Reemplace las llamadas al método con el contenido del método y elimine el método en sí

```
class Bad {
    // Bad implementation
    isOfLegalAge() {
        return calculateAge();
    }

    calculateAge() {
        return this.user.getAge() >= 18;
    }
}


class Good {
    // Good implementation
    isOfLegalAge() {
        return this.user.getAge() >= 18;
    }
}
```

Composing Method - Extract Variable

¿En que consiste Extract Variable?

La refactorización de variables es una forma de reducir la complejidad cognitiva de un método, lo cual resulta difícil de entender

Problema

Tienes una expresión que es difícil de entender

Solucion

Coloque el resultado de la expresión o sus partes en variables separadas que se explican por sí mismas

```
class Bad {
    // Bad implementation
    isOfLegalAge() {
        if ((user.age >= 18) &&
            (user.name == "Pablo") &&
            (user.country == "Argentina")
        ) {
            // Do something
        }
    }
}


class Good {

    // Create const variables
    NAME_PABLO_VALIDATION = "Pablo";
    COUNTR_ARGENTINA_VALIDATION = "Argentina";

    // Good implementation
    isOfLegalAge() {
        const isOfLegalAge = user.getAge() >= 18;
        const name = user.getName();
        const country = user.getCountry();

        // Simplify conditional
        if (isOfLegalAge &&
            (name == NAME_PABLO_VALIDATION) &&
            (country == COUNTR_ARGENTINA_VALIDATION)
        ) {
            // Do something
        }
    }
}
```

Composing Method - Split Temporary Variable

¿En que consiste Split Temporary Variable?

La refactorización de variables temporales es una forma de asignar correctamente variables, cada una de ellas debe tener una única responsabilidad

Problema

Tiene una variable local que se usa para almacenar varios valores intermedios dentro de un método (excepto las variables de ciclo como un for)

Solucion

Utilice diferentes variables para diferentes valores. Cada variable debe ser responsable de una sola cosa en particular

```
class Bad {
    // Bad implementation
    isOfLegalAge() {
        let age = user.age;
        console.log(age);
        age = user.age + 1;
        console.log(age);
    }
}


class Good {
    // Good implementation
    isOfLegalAge() {
        const age = user.getAge();
        console.log(age);
        const newAge = user.getAge(); + 1;
        console.log(newAge);
    }
}
```

Composing Method - Remove Assignments to Parameters

¿En que consiste Remove Assignments to Parameters?

La refactorización asignación de variables consta de no modificar una variable que nos llegó por parámetro. Si vamos a manipularla debemos asignarla a una nueva variable

Problema

Se asigna algún valor a un parámetro dentro del cuerpo del método y este parámetro si se modificó podemos generar bugs por que su valor ya no es el inicial

Solucion

Utilice una variable local en lugar de un parámetro

```
class Bad {
    // Bad implementation
    isOfLegalAge(comparativeAge) {
        if (user.age == comparativeAge) {
            comparativeAge++;
        }
        // Do something
    }
}


class Good {
    // Good implementation
    isOfLegalAge(comparativeAgeParam) {
        const comparativeAge = comparativeAgeParam;

        if (user.age == comparativeAgeParam) {
            comparativeAge++;
        }
        // Do something
    }
}
```

Refactoring by Abstraction

Refactoring by Abstraction

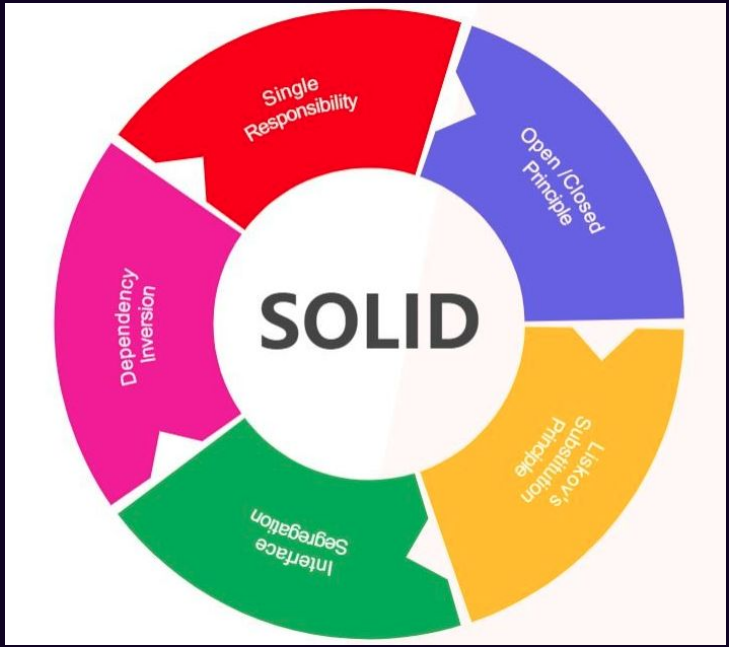
¿Que es Refactoring by Abstraction?

La refactorización por abstracción es una técnica en la que se identifica una funcionalidad común compartida por múltiples clases o métodos y se extrae en una clase o interfaz abstracta separada. Este proceso ayuda a reducir la duplicación de código, promueve la reutilización, facilita la gestión y el mantenimiento de la funcionalidad compartida

Ejemplo

Considere dos clases con métodos similares que realizan los mismos cálculos. Al extraer la lógica compartida en una clase abstracta o una interfaz, puede crear una implementación única que ambas clases pueden heredar o implementar. Esta abstracción reduce la cantidad de código duplicado y facilita la actualización o modificación de la funcionalidad compartida en el futuro.

Refactoring by Abstraction



Dentro de esta técnica podemos hacer una referencia a los principios SOLID, mas precisamente a Interface Segregation Principle y Single Responsibility principle

*En otro documento se explica con detalle cada principio

Referencias

- **Softwarecrafters:**
  https://softwarecrafters.io/javascript/tdd-test-driven-development
- **Codecademy:**
  https://www.codecademy.com/article/tdd-red-green-refactor
- **Medium:**
  https://medium.com/@tunkhine126/red-green-refactor-42b5b643b506
- **Codesee:**
  https://www.codesee.io/learning-center/code-refactoring#:~:text=Refactoring%20by%20abstraction%20is%20a,and%20maintain%20the%20shared%20functionality
- **Refactoring.guru:**
  https://refactoring.guru/refactoring/techniques/composing-methods