Refactoring

- Refactoring es una transformación que preserva el comportamiento, pero mejora el diseño
- Es el proceso a través del cual se cambia un sistema de software
  - Para mejorar la organización, legibilidad, adaptabilidad y mantenibilidad del código luego que ha sido escrito
  - Que NO altera el comportamiento externo del sistema

Manera de aplicar refactoring

1. Indico mal olor o malos olores
2. Indico refactorings a aplicar
3. Escribo codigo refactorizado

Códigos de ejemplo

Ejercicios de Práctica

Code smells y refactorings a aplicar

- **Codigo duplicado**
  - Extract method
  - Pull up method
  - Form template method
  - Extract classs
- **Nombres poco descriptivos**
  - Renombrar (Rename method, Rename class, Rename variable)
- **Reinventar la rueda**
  - Replace loop with pipeline
  - Replace temp with query
- **Temporary field**
  - Replace Temp with Query
- **Envidia de atributo (Feature Envy)**
  - Move method
  - Este mal olor se identifica cuando un método accede mucho a datos de otro objeto (muchos getters). ¿La solución? El método claramente necesita estar en otro lado
  - Muchas veces parte del método sufre envidia y no todo el método. En tal caso, usar extract method y luego move method
- **Metodos largos**
  - Extract method
  - Decompose conditional
  - Replace temp with query

- **Clases grandes**
  - Extract class
  - Extract subclass
- **Muchos parametros**
  - Replace parameter with Method
  - Preserve whole object
  - Introduce Parameter Object
- **Data Class**
  - Move method
- **Sentencia Switch**
  - **Replace Conditional with Polymorphism**
    - Se usa este refactoring cuando se tiene un condicional que elije el comportamiento dependiendo del tipo de un objeto
  - Va de la mano tambien con Replace Type Code with Subclasses y con Replace Type Code with State/Strategy
- **Obsesion por tipos primitivos (Primitive obsession)**
  - **Replace Data Value with Object**
  - **Replace Type Code with Subclasses**
    - Pasas de tener Empleado con una v.i “tipo” a tener una clase Empleado de la que heredan los tipos de Empleado
    - En este caso, el tipo de un objeto no cambia, es más estatico
  - **Replace Type Code with State/Strategy**
    - Similar al caso anterior, tenes una clase con una v.i “tipo” pero no te alcanza solo con crear subclases
    - Se usa este refactoring cuando el “tipo” cambia durante el ciclo de vida del objeto
- **Cadena de mensajes**

    - Hide delegate
    - Extract method and Move method
- **Middle man**
    - Remove middle man
- **Cambios divergentes**
    - Ocurre cuando un solo módulo o clase necesita ser modificado por múltiples razones distintas en diferentes momentos. Esto suele ser una señal de que la clase o módulo tiene demasiadas responsabilidades, violando el Principio de Responsabilidad Única
    - Extract class
- **Comentarios**
    - Extract method
    - Rename method

Code smells en Detalle

- **Lazy Class**
    - El code smell **Lazy Class** ocurre cuando una clase no justifica su existencia porque hace muy poco o nada significativo. Puede haber sido creada con la intención de organizar mejor el código, pero termina sin suficiente responsabilidad para justificar su presencia, lo que introduce una sobrecarga innecesaria en la estructura del proyecto
    - ¿Cómo lo identifico?
        - Clase con muy pocos métodos o atributos
        - Clase que simplemente delega su trabajo a otra

        - Clase que solo actúa como un contenedor de datos sin lógica (Podría ser Data Class tambien)

- **Middle Man**
    - El code smell **Middle Man** ocurre cuando una clase no agrega valor real y solo actúa como un intermediario entre otras clases, delegando casi todas sus responsabilidades sin hacer ningún trabajo significativo
    - ¿Cómo lo identifico?
        - You look at a class's interface and find half the methods are delegating to this other class
        - La clase solo delega llamadas a otra clase sin agregar lógica propia
        - El 80% o más de los métodos son delegaciones
        - El código se vuelve más difícil de seguir

- **Shotgun surgery**
    - El **Shotgun Surgery** ocurre cuando un pequeño cambio en una clase requiere modificar muchas otras clases o archivos al mismo tiempo.

- **Temporary field**
    - Un caso comun de “atributos temporales” ocurre cuando un algoritmo complicado necesita muchas variables
    - Se soluciona usando Extract Class para darle un lugar a dichas variables
    - Oftentimes, temporary fields are created for use in an algorithm that requires a large amount of inputs. So instead of creating a large number of parameters in the method, the programmer decides to create fields for this data in the class

- **Duplicated code**

- El caso más simple es cuando tenes la misma expresión en dos métodos de la misma clase. Esto se arregla con un Extract method

```
// Caso 1: Misma expresión en dos métodos de la misma clase
// Code Smell: Código duplicado en los métodos calcularTotalConDescuento
class Pedido {
    private double precio;
    private int cantidad;

    public double calcularTotalConDescuento() {
        double total = precio * cantidad;
        total = total * 0.9; // 10% de descuento
        return total;
    }

    public double calcularTotalConImpuesto() {
        double total = precio * cantidad;
        total = total * 1.21; // 21% de IVA
        return total;
    }
}
```

- Otra forma comun de duplicacion es cuando tenes la misma expresión en dos subclases hermanas. Para solucionarlo, es posible usar Extract method y despues Pull Up method

```
// Caso 2: Misma expresión en dos subclases hermanas
// Code Smell: Código duplicado en calcularBonificacion de ambas subclases
abstract class Empleado {
    protected double salarioBase;
}

class EmpleadoTiempoCompleto extends Empleado {
    public double calcularSalarioFinal() {
        return salarioBase + (salarioBase * 0.1); // 10% de bonificación
```

```
    }
}

class EmpleadoMedioTiempo extends Empleado {
    public double calcularSalarioFinal() {
        return salarioBase + (salarioBase * 0.05); // 5% de bonificación
    }
}
```

- Similar al caso anterior, si el codigo es similar pero no el mismo, hay que usar Extract method para separar las partes similares de las diferentes, y luego será posible usar Form template method

```
// Caso 3: Código similar pero no idéntico en subclases hermanas
// Code Smell: Código estructuralmente similar pero con diferencias menores
class ReporteVentas {
    public void generarReporte() {
        System.out.println("Obteniendo datos de ventas");
        System.out.println("Procesando datos de ventas");
        System.out.println("Imprimiendo reporte de ventas");
    }
}

class ReporteInventario {
    public void generarReporte() {
        System.out.println("Obteniendo datos de inventario");
        System.out.println("Procesando datos de inventario");
        System.out.println("Imprimiendo reporte de inventario");
    }
}
```

- Si hay codigo duplicado en 2 clases no relacionadas, es posible usar Extract class en una de las clases y luego usar el nuevo componente en la otra

```
// Caso 4: Código duplicado en clases no relacionadas
// Code Smell: Lógica de logging repetida en diferentes clases no relacionadas
class ServicioPago {
    public void procesarPago() {
        System.out.println("LOG: Procesando pago...");
        // Lógica de pago
    }
}

class ServicioEnvio {
    public void procesarEnvio() {
        System.out.println("LOG: Procesando envío...");
        // Lógica de envío
    }
}
```

- **Feature envy**
    - Ocurre cuando un método parece estar más interesado en otra clase que a la clase a la que pertenece
    - El enfoque más claro de la envidia es en los datos. Ocurre cuando un método invoca miles de getters de otro objeto para calcular un valor
    - La facil para solucionarlo es usar Move method
    - Si solo una parte del método sufre Feature envy entonces conviene Extract method de la parte “celosa” y usar Move method

- **Long method**
    - La mayoria del tiempo, lo unico que hay que hacer para acortar un método es usar Extract method. Encontrar las partes que parecen ir bien juntas y hacer un nuevo método

- Si se tiene un método con muchos parametros y variables temporales, estos elementos se meten en el camino del Extract method. En este caso, conviene usar Replace temp with query para eliminar los temps. La lista larga de parmetros puede achicarse con Introduce Parameter Object y Preserve Whole Object
- ¿Cómo identificar los grupos de código a extraer? Comentarios, condicionales y loops dan señales de extracción
  - Para lidiar con condicionales → Decompose conditional
  - Para lidiar con loops → Extraer el loop y el codigo dentro en su propio método

- **Primitive obsession**
  - It is possible to use Replace Data Value with Object on individual data values
  - If the data value is a type code, use Replace Type Code with Class if the value does not affect behavior
  - If you have conditionals that depend on the type code, use Replace Type Code with Subclasses or Replace Type Code with State/Strategy

- **Switch statement**
  - La mayoria del tiempo que se ve un switch statemente, hay que considerar el polimorfismo
  - Generalmente, estas sentencias cambian segun un Type code. Para solucionarlo :
    - Primero hay que reemplazar el type code con subclases o state/strategy
    - Una vez configurado la estructura de herencia, es posible aplicar Replace Conditional with Polymorphism

Refactorings en Detalle

- **Extract method** :
  - Mecánica :
    - Crear un nuevo metodo cuyo nombre explique su proposito
    - Copiar el codigo a extraer al nuevo metodo
    - Revisar las variables locales del original

- **Move method**
  - Motivación
    - Un metodo esta usando o usara muchos servicios que estan definidos en una clase diferente a la suya
    - Hay una mala asignacion de responsabilidades
  - Mecánica :
    - Revisar la variables de instancia usadas por el metodo a mover
    - Revisar super y subclases por otras declaracion del metodo. Si hay otras tal vez no se pueda mover
    - Crear un nuevo metodo en la clase target cuyo nombre explique su proposito
    - Copiar el codigo a mover al nuevo metodo. Ajustar lo que haga falta

- **Replace Temp with Query** :
  - Este refactoring suele ser importante antes de aplicar Extract method. Las variables locales suelen ser dificiles de extraer, asi que conviene reemplazarlas con queries de ser posible

- **Replace Conditional with Polymorphism**
  - Descripción : tenes un condicional que elije diferentes comportamientos dependiendo de el tipo de un objeto
  - Mecánica :
    - **Antes de aplicar este refactoring**, es necesario tener una estructura de herencia
    - Si el condicional es parte de un metodo largo, sacar la sentencia condicional usando Extract method
    - Usando Move Method, mover el método con el switch a la parte más alta de la jerarquia
    - Elegir una de las subclases y crear un metodo allí que sobreescriba el metodo antes mencionado, usando el cuerpo de una de las condiciones
    - Luego, quitar esa condicion y cuerpo del metodo con el switch y continuar haciendo esto hasta que no queden condiciones
    - Finalmente, volver el metodo con el switch abstracto

- **Extract Class**
  - Descripción : Tenes una clase haciendo lo que deberian hacer dos
  - Explicación simple : Crear una nueva clase y mover los atributos y metodos relevantes de la clase vieja a la nueva

- **Extract Subclass**
  - Descripción : una clase tiene atributos (o métodos) que sólo son usados en algunas instancias
  - Explicación simple : crear una subclase para ese subconjunto de atributos (o métodos)

    - Motivación :
        - El mayor disparador para usar este refactoring es darte cuenta que una clase tiene comportamiento usado por algunas instancias y no por otras. Generalmente esto es señalado por un type code, en cuyo caso se puede usar Replace type code with subclasses o state/strategy
        - La mayor alternativa a Extract subclass es Extract class. Esta es una elección entre delegación (dejo que otro haga el laburo) y herencia

- **Extract Superclass**
    - Descripción : tenes dos clases con características (features) similares
    - Explicación simple : Crear una superclase y mover los atributos comunes allí

- **Pull Up method**
    - Motivación
        - Cuando varias subclases tienen métodos idénticos o muy similares
        - La duplicación de código en múltiples subclases puede generar problemas de mantenimiento, ya que cualquier cambio en la lógica debe replicarse en todas las
          versiones del método

- **Replace Type Code with Class**
- **Replace Type Code with Subclasses**
- **Replace Type Code with State/Strategy**
    - Descripción : Tenes un type code que afecta el comportamiento de una clase, pero no podes crear subclases
    - Es similar a Replace Type Code with Subclasses pero puede ser usado si :
        - El type code cambia durante la vida del objeto o

        - Si otra razon previene la creación de subclases (por ej tener una clase usuario con tipo de suscripción)
    - Pasos para implementar :
        - Crear una nueva clase con el proposito del Type code (state object)
        - Agregarle las subclases correspondientes
        - Agregar un abstract query en el state object para retornar el type code. Sobreescribir este método en las subclases
        - Crear un atributo en la vieja clase para el nuevo state object
        - ...

- **Convert Procedural Design to Objects**
    - Descripción : convertir la información en objetos, separar el comportamiento y moverlo a los objetos

Refactorings relacionados con parametros

- **Replace parameter with method**
    - Se debe usar cuando se puede obtener la información **en un solo parametro** haciendo un request a un objeto. Este objeto puede ser un atributo u otro parametro
- **Preserve whole object**
    - Se debe usar cuando una función o método recibe múltiples valores extraídos de un objeto en lugar de recibir directamente el objeto completo
- **Introduce Parameter Object**
    - Se debe usar cuando una función o método recibe múltiples parámetros relacionados que pueden agruparse en un solo objeto. Si un método recibe varios parámetros que están conceptualmente relacionados, es mejor encapsularlos en un solo objeto

- **Decompose Conditional**
  - Se tiene un condicional complicado con varios if-then-ele
  - Extraer la condicion en su propio método (crear método que evalue la condicion)
  - Extraer la parte del **then y la parte del else** en sus propios métodos