**Diseño de Software**

**Trabajo Autónomo: Refactoring**

**Feature Envy (Envidia de Características)**

Sucede cuando un método accede a los datos de otros objetos mas que a sus propios datos, lo cual es un problema porque va en contra del primer principio SOLID, Single Responsability Principle. (Martín, 2018)

**Técnicas de Refactorización:**

* Si el método puede ser movido a otra clase, entonces hay que moverlo.
* Si solo una parte del método accede a datos de otra clase, entonces se puede crear y usar un método externo implementándolo en la otra clase.
* Si el método accede a varias otras clases, primero hay que determinar cual es la clase a la que mas se accede y mover el método hacia ella. También se puede usar métodos externos e ir posicionándolos en las clases que sean necesarias.

(Refactoring Guru, s.f.)

**Ejemplo:**

<https://github.com/Josalcab/TareaRefactoring/tree/master/Refactoring/src/feature_envy> (Kerievsky)

**Primitive Obsession (Obsesión Primitiva)**

Se da cuando se usan muchos datos primitivos para tareas sencillas en lugar de utilizar objetos pequeños que puedan englobar esos datos. También se puede dar cuando se codifica información utilizando constantes.

Es un problema porque al usar demasiados datos pequeños se reduce la legibilidad y comprensión del código, además que los datos primitivos no brindan la flexibilidad que ofrecen los objetos. (Refactoring Guru, s.f.)

**Técnicas de Refactorización:**

* Si hay muchos datos de tipo primitivo, de ser posible hay que agruparlos en clases representativas y además mover a estas clases todos los comportamientos relacionados a sus datos.
* Si estos primitivos son usados como parámetros de métodos se pueden usar en su lugar los tipos de objetos construidos a partir de esos datos.
* Si hay una serie de datos tipo String en un array, se puede convertir ese array en un objeto.

**Ejemplo:**

<https://github.com/Josalcab/TareaRefactoring/tree/master/Refactoring/src/primitive_obsession>

**Refused Bequest (Legado Rechazado)**

Si una subclase usa solo algunos de los métodos y propiedades heredados de sus padres, la jerarquía se verá comprometida. Los métodos innecesarios pueden simplemente no usarse o redefinirse y emitir excepciones.

Se da usualmente cuando el programador quiere reutilizar código de una superclase, pero el código de la superclase y el de la subclase no guardan suficiente relación. (Refactoring Guru, s.f.)

**Técnicas de Refactorización:**

* Si entre la subclase y la superclase no hay una relación realmente clara, lo mejor es eliminar la relación de herencia y hacer que la subclase se convierta en una clase independiente.
* Si la herencia no es el problema, solo hace falta eliminar de la subclase los atributos y métodos innecesarios. También se tendría que extraer e implementar los atributos y métodos necesarios para la subclase desde la clase principal, entonces se colocarían en una nueva subclase y se dejarían listas para heredar desde ahí.

**Ejemplo:**

<https://github.com/Josalcab/TareaRefactoring/tree/master/Refactoring/src/refused_bequest> (Ershad, 2017)

**Switch Statements (Sentencias Switch)**

Se da cuando hay implementaciones complejas del operador “switch” o secuencias de sentencias “if”.

El uso de sentencias switch es bastante raro, por lo que a menudo se implementan de manera ineficiente, muchas veces el código para una sola sentencia switch puede estar dispersado y para agregar una condición se deben encontrar y modificar varias partes del código. (Refactoring Guru, s.f.)

**Técnicas de Refactorización:**

* Es posible que sea necesario aislar la sentencia Switch y extraer el método que lo contenga y moverlo a la clase correcta.
* Se puede usar polimorfismo para arreglar este mal olor, por lo que habría que crear clases, superclases o subclases según sea necesario para establecer relaciones de herencia y hacer que un mismo método funcione de manera adecuada para cada una de las clases hijas.
* En caso de que en el Switch todas las condiciones llamen al mismo método, pero con diferentes parámetros o que el código sea parecido en todos, el polimorfismo es la manera más adecuada de reparar el código.
* Si una de las condiciones es nula se puede incluir un Null Object.

**Ejemplo:**

<https://github.com/Josalcab/TareaRefactoring/tree/master/Refactoring/src/switch_statements> (Nogal, 2016)

**Shotgun Surgery (Cirugía de Escopeta)**

Se da cuando se requieren hacer muchos cambios pequeños en el código para poder modificar una sola funcionalidad

Es un problema porque va directamente en contra del primer principio SOLID, Single Responsability Principle, ya que una sola responsabilidad esta repartida en diferentes clases. (Refactoring Guru, s.f.)

**Técnicas de Refactorización:**

* Se pueden mover métodos y atributos de varias clases a una clase única que pueda representar bien todos esos comportamientos y si no existe la clase apropiada para ellos hay que crear una nueva.
* Si mover el código a una sola clase hace que una o varias de las clases originales quede casi vacía, entonces hay que deshacernos de estas clases y mover sus comportamientos a otra clase que esté relacionada.

**Ejemplo:**

<https://github.com/Josalcab/TareaRefactoring/tree/master/Refactoring/src/shotgun_surgery> (Mitra, 2016)

**Repositorio:**

<https://github.com/Josalcab/TareaRefactoring>

**Referencias**

Ershad, G. M. (5 de Enero de 2017). Refused Bequest - A Code Smell. Obtenido de C# Corner: https://www.c-sharpcorner.com/article/refused-bequest-a-code-smell/

Kerievsky, J. (s.f.). Code Smells: A Catalog of Wisdom about Hazards in Code — Java Edition. Recuperado el 10 de Enero de 2020, de https://elearning.industriallogic.com/gh/submit?Action=PageAction&album=recognizingSmells&path=recognizingSmells/featureEnvy/featureEnvyExample&devLanguage=Java

Martín, M. J. (22 de Noviembre de 2018). SOLID: los 5 principios que te ayudarán a desarrollar software de calidad. Obtenido de Profile: https://profile.es/blog/principios-solid-desarrollo-software-calidad/

Mitra, S. (16 de Septiembre de 2016). The Shotgun Surgery Code Smell. Obtenido de DZone: https://dzone.com/articles/code-smell-shot-surgery

Nogal, A. (31 de Octubre de 2016). Code Smells (Part I). Obtenido de DZone: https://dzone.com/articles/code-smellspart-i

Refactoring Guru. (s.f.). Feature Envy. Obtenido de Refactoring Guru: https://refactoring.guru/smells/feature-envy

Refactoring Guru. (s.f.). Primitive Obsession. Obtenido de Refactoring Guru: https://refactoring.guru/smells/primitive-obsession

Refactoring Guru. (s.f.). Refused Bequest. Obtenido de Refactoring Guru: https://refactoring.guru/smells/refused-bequest

Refactoring Guru. (s.f.). Shotgun Surgery. Obtenido de Refactoring Guru: https://refactoring.guru/smells/shotgun-surgery

Refactoring Guru. (s.f.). Switch Statements. Obtenido de Refactoring Guru: https://refactoring.guru/smells/switch-statements