Nombre: Marlon Eddie Lindao Varas

Trabajo Autónomo – Code Smells y Refactoring

Code Smells

1. Data clumps (Grupo de datos)

Se refiere, cuando ya sea en los atributos de una clase, o en la lista de parámetros de un método, hay datos que se relacionan tanto, que deberían ser pertenecientes a una clase, de tal forma, que ya sea en cualquiera de los dos casos mencionados, ese grupo es reemplazado por un objeto que contenga esos datos. Este mal olor es un problema porque puede hacer una clase muy larga, o hacer que la lista de parámetros del método muy extenso. (“DataClump,” n.d.)

Técnicas de Refactorización para solucionar Data clumps

1. Extraer Clase: Usar esta técnica para agrupar esos datos en una clase nueva, y usarlo como objeto. Una vez realizado esto, buscar donde se encontraban este grupo de datos y reemplazarlos por el objeto de la nueva clase creada. Esto reducirá la lista de parámetro del método o los atributos de la clase donde se encontraban este grupo de datos. (“Chapter 3. Bad Smells in Code - Refactoring: Ruby Edition,” n.d.)

1. Long Class (Clase Larga)

Se da cuando en una clase, por conveniencia, se le agregan varios atributos y varios métodos, de tal forma, que se hace extensa. Esto ocasiona un problema, porque seguramente, no este manejando una sola responsabilidad, incumpliendo con el principio de single responsability, además de que no permite reutilizar código y su mantenimiento se vuelve complejo. (“Code Smells : Bloaters (Long Methods, Long Class) – ByteLanguage.Net,” n.d.)

Técnicas de Refactorización para solucionar Long Class

1. Extraer clase: Reunir atributos que tengan relación y colocarlos en una clase nueva. Solo se lo usa si el conjunto de datos no presenta una relación fuerte con la clase larga. (“Chapter 3. Bad Smells in Code - Refactoring: Ruby Edition,” n.d.)
2. Extraer Subclase: Si hay un grupo de atributos que funcionan como un componente o una variabilidad de la clase, se juntan las variables en una clase hija de la clase principal. (“Chapter 3. Bad Smells in Code - Refactoring: Ruby Edition,” n.d.)
3. Extraer Modulo: Si la clase realiza mas de una responsabilidad, estas responsabilidades deben separarse en clases nuevas para que sean usados como módulos en esta clase. (“Chapter 3. Bad Smells in Code - Refactoring: Ruby Edition,” n.d.)
4. Switch statements (Sentencias switch)

Se da cuando se usa los switch-cases como simulación de polimorfismo, donde el case es por lo general un enum, y los resultados llaman al comportamiento apropiado del tipo. Esto genera que siempre se tenga que modificar el switch cuando hay un tipo nuevo, lo que se vuelve complejo, si se trabajan con muchos tipos, y es peor si esa porción de código se ubica en otras secciones del código. (“Switch Statements are Like Ants - DaedTech,” n.d.)

Técnicas de Refactorización para solucionar Switch statements

1. Usar polimorfismo: Para realizar esto, primero se deberá crear las subclases según los tipos que maneje el switch, y en caso de no contar con una clase general o interfaz que una a las subclases, se deberá crearlas. Después se deber extraer los case statement a realizar por el switch-case con Extraer método, y colocar esa porción de código, en la implementación de esa acción de la subclase respectiva. (“Chapter 3. Bad Smells in Code - Refactoring: Ruby Edition,” n.d.)
2. Refused Bequest (Legado Rechazado)

Se da cuando una o varias subclases no usan todos los métodos de la clase padre, lo que rompe con el principio de sustitución de Liskov. (“Refused Bequest - A Code Smell,” n.d.)

Técnicas de Refactorización para solucionar Refused Bequest

1. Push Down Method: Esta técnica simplemente dice que se cree una clase hermana del padre, y colocar todos aquellos métodos que no son usados por los hijos. (“Chapter 3. Bad Smells in Code - Refactoring: Ruby Edition,” n.d.)
2. Reemplazar Herencia con delegación: Si no se quiere tocar la jerarquía con Push Down Method, se puede reemplazar la herencia simplemente haciendo que la clase hija tenga como atributo a un objeto de la clase padre, y después pone a ese objeto a realizar las tareas que necesite de la clase padre. (“Chapter 3. Bad Smells in Code - Refactoring: Ruby Edition,” n.d.)
3. Divergent Change (Cambio divergente)

Se lo detecta, cuando tenemos una sola clase, que, por muchos cambios distintos en el sistema, tiene que ser modificada según el cambio que se realiza. Esto es un problema, porque muestra que la clase se encuentra relacionada con varias clases, de tal forma, que, si se modifican las otras clases, es muy seguro que terminemos reescribiendo la clase en cuestión. (“Un repaso por los code smells más comunes - intive Argentina Blog,” n.d.)

Técnicas de Refactorización para solucionar Divergent Change

1. Extraer Clase: Para aplicar esta técnica, primero hay que revisar todo lo que cambie por una razón en particular, y extraerlo para juntar ese cambio en una nueva clase. (“Chapter 3. Bad Smells in Code - Refactoring: Ruby Edition,” n.d.)