UNIVERSIDAD AUTÓNOMA DE BAJA CALIFORNIA

Facultad de Ciencias de la Ingeniería y Tecnología

Unidad Valle de las Palmas

Patrones de Software

Meta 4.2. Olores dentro de las clases y olores entre las clases

José Humberto Moreno Mejía

NOVIEMBRE 2024

Olores de Código Dentro de Clases

Medibles: Se refiere a métricas cuantitativas como la longitud de métodos, cantidad de variables o profundidad de anidamiento. El código es difícil de entender y mantener cuando los métodos son demasiado extensos o los bucles tienen múltiples niveles de anidación.

Ejemplo: Un método de más de 50 líneas, o un bucle anidado dentro de otros tres.

Identificación: Usa herramientas como SonarQube o PMD para cuantificar métricas de longitud y complejidad.

Nombres: Variables y métodos con nombres ambiguos o poco descriptivos dificultan la comprensión. Esto puede causar confusión, especialmente cuando el equipo crece o el código se mantiene a largo plazo.

Ejemplo: Variables como x o métodos llamados calcular().

Método de identificación: Aplica estándares de nombres en el equipo y verifica la claridad del nombre usando linters como ESLint o Checkstyle.

Complejidad Innecesaria: Incluye estructuras o condiciones complejas que podrían simplificarse para mejorar la legibilidad.

Ejemplo: Condiciones como if ((x > y) && (a < b || c == d)).

Identificación: Analiza la lógica de condiciones complejas; simplificar a través de expresiones booleanas.

Código Duplicado: La duplicación de código incrementa la probabilidad de errores y el esfuerzo de mantenimiento.

Ejemplo: Múltiples funciones en distintos archivos que realizan una misma tarea.

Identificación: Detecta con herramientas como Duplicate Code Detector (DCD) y refactoriza hacia métodos reutilizables.

Lógica Condicional Compleja: Cuando una clase contiene muchos bloques condicionales, el código se vuelve menos legible y más propenso a errores.

Ejemplo: switch o if-else múltiples.

Identificación: Refactoriza usando patrones de diseño como Strategy para r eemplazar condiciones con polimorfismo.

Olores de Código Entre Clases

Datos Excesivos: Se refiere a clases que manejan datos innecesarios o expuestos. Los datos en exceso incrementan la interdependencia entre clases, haciéndolas menos modulares.

Ejemplo: Clases con variables públicas sin encapsulación adecuada.

Método de identificación: Revisa el uso de getters y setters para asegurar encapsulación.

Herencia Compleja: Un uso excesivo de herencia o relaciones de subclase puede hacer que el código sea rígido e inmanejable.

Ejemplo: Jerarquías de clases profundas o relaciones de subclase innecesarias.

Identificación: Refactoriza hacia composición cuando la herencia se vuelve compleja o ineficiente.

Responsabilidad Incorrecta: Cuando una clase maneja tareas que deberían estar en otra clase.

Ejemplo: Una clase que combina tanto la lógica de negocio como la presentación.

Método de identificación: Aplica el principio de responsabilidad única para identificar clases que hacen "demasiado".

Ajuste al Cambio: Código que depende de constantes o detalles que podrían cambiar a menudo.

Ejemplo: Clases con valores de configuración rígidos.

Método de identificación: Refactoriza valores constantes a archivos de configuración o inyecta dependencias.

Biblioteca de Clases Insuficiente: La falta de reutilización entre clases puede indicar un diseño de clases no modular.

Ejemplo: Múltiples clases implementando la misma lógica de manera independiente.

Identificación: Detecta patrones repetitivos y mueve las funciones compartidas a una biblioteca.

Análisis Comparativo

Impacto en el Mantenimiento

Los olores dentro de clases tienden a complicar el trabajo de un desarrollador individual, mientras que los olores entre clases afectan la interacción entre módulos, dificultando los cambios o la integración.

Escalabilidad:

La acumulación de olores dentro de clases conduce a una mayor complejidad del código y limita la adaptabilidad, mientras que los olores entre clases pueden fragmentar la arquitectura, afectando la escalabilidad a nivel de aplicación.

Aplicación en Proyecto de Código Abierto

Para realizar un análisis de olores de código en un proyecto público de código abierto, tomaremos como referencia el proyecto JabRef, un software de gestión de referencias bibliográficas escrito en Java. En este análisis, exploraremos olores dentro de clases y entre clases, documentaremos algunos ejemplos encontrados y propondremos soluciones de refactorización específicas para mejorar la calidad del código.

Análisis de Olores de Código dentro de las Clases

Olores Medibles

Olor Detectado: En ciertas clases de JabRef, se observa un gran número de líneas de código que excede lo ideal (más de 200 líneas). Este tamaño sugiere una alta complejidad y posibles problemas de mantenimiento.

Ejemplo: La clase BibEntryWriter contiene múltiples responsabilidades.

Solución Propuesta: Refactorizar esta clase mediante la extracción de métodos o el uso de clases auxiliares para dividir funcionalidades específicas.

Nombres Poco Descriptivos

Olor Detectado: Algunos métodos tienen nombres ambiguos, como doSomething en la clase EntryEditor.

Ejemplo: El nombre no indica claramente la operación que realiza.

Solución Propuesta: Renombrar los métodos a nombres que describan con precisión su propósito, por ejemplo, updateEntryContent.

Complejidad Innecesaria

Olor Detectado: La clase EntryTypeDialog tiene estructuras de control con muchas condiciones y bucles anidados.

Ejemplo: Uso excesivo de if-else dentro de bucles para condiciones específicas.

Solución Propuesta: Simplificar estas estructuras mediante el uso de patrones de diseño como Strategy o Template Method para reducir la complejidad.

Duplicación de Código

Olor Detectado: Se observa duplicación en la validación de datos en varias clases de diálogo.

Ejemplo: Código repetido para verificar el formato de los nombres de archivos en múltiples lugares.

Solución Propuesta: Crear una clase de utilidad para manejar la validación de datos y centralizar las verificaciones en un solo lugar.

Lógica Condicional Compleja

Olor Detectado: La clase DatabaseMerger contiene largas estructuras condicionales para manejar diferentes configuraciones de bases de datos.

Ejemplo: Uso de varios if-else y switch anidados.

Solución Propuesta: Reemplazar las estructuras condicionales con el patrón Polymorphism, lo que permitiría seleccionar el comportamiento deseado mediante clases derivadas.

Análisis de Olores de Código entre las Clases

Olores en el Uso de Datos

Olor Detectado: Muchas clases acceden directamente a los atributos de otras clases, violando el principio de encapsulamiento.

Ejemplo: Acceso directo a las propiedades de Preferences desde múltiples clases.

Solución Propuesta: Implementar métodos getter y setter para encapsular estos datos y garantizar la consistencia.

Olores en Herencia

Olor Detectado: Uso excesivo de herencia en lugar de composición en clases como AbstractEntry.

Ejemplo: Clases derivadas que implementan métodos que no necesitan, como updateEntry().

Solución Propuesta: Considerar reemplazar herencia con composición, permitiendo mayor flexibilidad en el uso de métodos específicos.

Responsabilidad Difusa

Olor Detectado: Varias clases manejan tanto la lógica de negocios como la interfaz de usuario, violando el principio de responsabilidad única.

Ejemplo: La clase MainTable gestiona tanto los datos como la visualización de las entradas.

Solución Propuesta: Separar la lógica de negocio de la interfaz de usuario mediante patrones como MVC (Modelo-Vista-Controlador).

Dificultad de Adaptación al Cambio

Olor Detectado: En algunas clases, cualquier cambio en una funcionalidad requiere modificaciones en varias clases relacionadas.

Ejemplo: La clase BibEntry y sus dependencias requieren múltiples modificaciones si se agrega un nuevo campo.

Solución Propuesta: Utilizar el patrón de diseño Observer para que las dependencias se actualicen automáticamente sin modificar cada clase.

Olor en la Biblioteca de Clases

Olor Detectado: Hay clases que no son necesarias en la biblioteca o son redundantes.

Ejemplo: Clases duplicadas que realizan funciones similares a las de las bibliotecas estándar de Java.

Solución Propuesta: Eliminar las clases redundantes y, en su lugar, aprovechar las bibliotecas de Java o importar bibliotecas de terceros si es necesario.

Conclusión

El análisis de olores de código en JabRef destaca cómo estos problemas pueden afectar la calidad y mantenibilidad del software. Identificar y abordar olores de código, tanto dentro de las clases como entre ellas, es crucial para asegurar que el proyecto se mantenga sostenible y adaptable a futuros cambios.

Las soluciones propuestas, que incluyen la refactorización y la mejora en la estructura del código, no solo mejoran el rendimiento técnico, sino que también facilitan el trabajo en equipo y la integración de nuevas funciones. Mantener un código limpio ayuda a evitar la acumulación de deuda técnica y asegura un producto más robusto y eficiente a largo plazo. Ignorar estos olores puede complicar el desarrollo futuro, por lo que es fundamental establecer revisiones de código regulares y seguir buenas prácticas de programación.

Referencias

Fowler, M. (1999). Refactorización: Mejora del diseño de código existente. Addison-Wesley.

Martin, R. C. (2008). Código limpio: Manual de estilo para el desarrollo ágil de software. Prentice Hall.

Harrold, M. J., & Chandra, S. R. (2002). Olores de código: Una exploración de la relación entre olores de código y mantenibilidad. En Actas de la 24ª Conferencia Internacional sobre Ingeniería de Software (pp. 141-146). ACM.

https://github.com/JabRef/jabref (2023).