**CS 6356.001: Software Maintenance, Evolution & Re-Engineering**

**Assignment 4: Code Smells & Refactoring**

**Team 17**

**Team Members:**

* Shivani Talatam
* Zamaan Bawa
* Pritul Dave

**MangoDB Code Smells:**

**Class:** DataSourceDao.java

**Location:** src/com/serotonin/mango/db/dao

**Code Smell 1: Feature Envy**

Feature Envy is a code smell in the Couplers category where a method is more interested in the state of another object than its own.This will lead to low cohesion and poor code design. DataSourceDao.java uses a lot of the fields and variables seen in the DataSourceVO.java class. There are objects being passed around from DataSourceVO and DataSourceDao. This can specifically be seen in methods such as “saveDataSource”, “insertDataSource” and “updateDataSource”. In these methods, we are extracting information from DataSourceVO and manipulating it outside of the class. This is definitely a code smell and should be resolved.

In order to resolve this, we can move these methods into the DataSourceVO class and will allow the class to control its own state. By moving the methods into that class, DataSourceDao will now delegate to the DataSourceVO class. This will resolve the feature envy seen in the class and will improve maintainability and understandability. This was done by taking the three methods in DataSourceDao and moving them to DataSourceVO. We slightly modified the code to work with the file and then were easily able to call them by using the object “vo”.

**Code Smell 2: Duplicated Code**

Duplicated Code is a code smell in the Dispensables category where we have code segments that do the exact same or similar things. This is something that can happen when multiple developers are working on the same program at the same time. The problem here is that if one of the clones is changed, then the other has to change. In our class DataSourceDao.java, we see duplicated code very early on with the two “getDataSource” methods on line 64 and 69. The only difference is that one is for an int and the other is for a String.

We can resolve this by extracting the method into one private method and having the two methods call from that. This will allow the logic to be allocated in one method. This improves maintainability and readability. Whenever a change needs to be made, the developer just has to make the change in one method rather than doing it in multiple places. This was resolved by creating a private method where we stored the logic that was being shared into one method and then calling the new private method in each of the prior methods and using the int or string as a parameter.

**Class:** StatisticsChartRenderer.java

**Location:** src/com/serotonin/mango/view/chart

**Code Smell 3: Long Method**

Long Method is a code smell in the Bloaters category where we have a method with many lines of code. This happens because it can be more difficult to create a method instead of just adding to the current one. This will cause issues when it comes to readability and understandability since long methods are generally tough to read through and understand the logic. In our class, we have a method “addDataToModel” which is a very long method that does a lot of things. We can reduce the size of this method by extracting each portion into their own relevant method. This will clean the code up immensely and improve readability. This is a code smell since it violates the Single Responsibility Principle.

We can resolve this by extracting the method into smaller methods. We modified “addDataToModel” and streamlined it to mainly set new variables and then call the new methods based on the conditionals set prior. Each of the new methods, “addRuntimeStatsToModel”, “addAnalogStatsToModel”, “addValueChangedCountToModel” were extractions of the previous method for “addDataToModel”. Now each of the methods featured a single responsibility. We also made another method “getStartValue” which handled the conditional in the beginning of the method. This method is called in our new methods as well. This improves the readability and overall understandability of the code.

**JEdit Code Smells:**

**Class:** ActionBar.java

**Location:** org/gjt/sp/jedit/gui/ActionBar.java

**Code Smell 1: Long Method**

One code smell in the given code is the Long Method smell in the invoke() method. The method is quite long and performs several tasks such as parsing the user input, handling property changes, recording macros, evaluating BeanShell snippets, and more. This makes the method difficult to understand and maintain. It should be refactored to smaller methods that perform individual tasks.

The code smell in the invoke() method is the violation of the Single Responsibility Principle (SRP). The method is responsible for two separate tasks:

1. Iterating over the collection of objects.
2. Invoking the execute() method on each object.

By combining these two tasks into a single method, the invoke() method is violating the SRP. As a result, it makes the code less maintainable, less testable, and harder to understand.

If we want to change the way objects are iterated or the way the execute() method is invoked, we have to modify the invoke() method, which affects the entire functionality of the method. If we separate these two tasks into separate methods, we can change them independently, making the code easier to maintain and test.

Therefore, it is considered an actual code smell, and it is recommended to refactor the code to separate the responsibilities of the invoke() method.

We can resolve this by extracting the method into smaller methods.

Extracting methods: The original code had a long and complex method invoke(). The refactored code extracted several smaller methods from it, such as getCommand(), getActionCommand(), hasProperty(), and constructPropertySnippet(). This makes the code easier to read, understand, and maintain.

Separation of concerns: The refactored code separates different concerns into different methods. For example, getActionCommand() and hasProperty() are responsible for parsing and validating the input string, while constructPropertySnippet() is responsible for constructing a code snippet that sets a property. This separation of concerns makes the code easier to understand and modify.

Improving method names: The refactored code uses more descriptive method names that convey the purpose of each method. For example, getCommand() is a more meaningful name than getCmd(), and constructPropertySnippet() is more descriptive than getProp().

Improving variable names: The refactored code uses more descriptive variable names that make it easier to understand the code. For example, action is renamed to inputField to clarify its purpose, and cmd is renamed to command to improve its readability.

Removing duplicate code: The refactored code removes duplicate code by extracting common functionality into methods. For example, the code to set a property on a buffer or jEdit is consolidated into a single method constructPropertySnippet(), which is called from multiple places in the code.

**Class:** Buffer.java

**Location:** org/gjt/sp/jedit/Buffer.java

**Code Smell 2: Feature Envy**

The feature envy present in the code is the multiple references to the markers object. The markers object is accessed multiple times throughout the code, including getting its size, getting elements from it using get(), removing elements from it using removeElementAt(), and inserting elements into it using insertElementAt(). This indicates that the logic related to managing markers is spread across multiple places in the code, rather than being encapsulated within the Marker class itself.

This violates the principle of encapsulation, as the code that manipulates the markers object should ideally be encapsulated within the Marker class itself, rather than being scattered throughout the calling code. This can lead to maintenance issues and make the code harder to understand and modify in the future. A potential solution to this feature envy issue could be to move the logic related to managing markers into the Marker class, so that the calling code only needs to interact with the Marker object and not directly with the markers object.

One way to refactor the code to address the feature envy issue is by extracting the logic for removing markers with a specific shortcut and position into separate methods (removeMarkersWithShortcut() and removeMarkersWithPosition()). This helps to reduce duplication of code and makes the code more modular and easier to understand. The main addMarker() method now focuses on adding the marker at the correct position in the markers list.

We extracted logic for removing markers with a specific shortcut: In the original code, there was a loop that iterated through the markers list and checked if each marker had a shortcut that matched the input shortcut parameter. This logic was extracted into a separate private method called removeMarkersWithShortcut(char shortcut). This method takes the shortcut parameter as input and iterates through the markers list to remove markers with a matching shortcut. This helps to encapsulate the logic for removing markers with a specific shortcut in a separate method, making the code more modular and easier to understand.

Thereafter, we extracted logic for removing markers with a specific position: Similar to removing markers with a specific shortcut, the logic for removing markers with a specific position was also extracted into a separate private method called removeMarkersWithPosition(int pos). This method takes the pos parameter as input and iterates through the markers list to remove markers with a matching position. This helps to encapsulate the logic for removing markers with a specific position in a separate method, improving code readability and maintainability.

After that, we simplified the addMarker() method: After extracting the logic for removing markers with a specific shortcut and position into separate methods, the main addMarker() method is now simplified and focuses only on adding the marker at the correct position in the markers list. This makes the code more concise and easier to understand.

Lastly, improved code organization: By extracting the feature envy logic into separate methods, the code is now better organized and follows the Single Responsibility Principle (SRP) of SOLID design principles. Each method has a clear responsibility and encapsulates a specific piece of functionality, which makes the code more modular and maintainable.

Overall, the refactored code eliminates feature envy by encapsulating the logic for removing markers with a specific shortcut and position into separate methods, simplifying the main addMarker() method, and improving code organization. This results in cleaner, more maintainable, and easier-to-understand code.

**Class:** RecentFilesProvider.java

**Location:** org/gjt/sp/jedit/menu/RecentFilesProvider.java

**Code Smell 3: Duplicated Code**

Duplicate code smell is a term used in software development to describe code that has been repeated, sometimes almost identically, in two or more places within a codebase. Duplicate code can be a sign of poor design, because it violates the DRY (Don't Repeat Yourself) principle, which advocates for code reusability, maintainability, and efficiency. Duplicate code can cause several problems, including increasing the chances of introducing bugs and making it harder to maintain the codebase. Duplicate code should be refactored into reusable functions or classes, to improve the code's maintainability and reduce the risk of bugs.

In our case there is code duplication in the update() method of the RecentFilesProvider class. This code creates a new sub-menu called "More" and adds it to the main menu if the maximum number of items in the menu has been reached. This code block appears twice in the method, with only a minor difference in the argument passed to the JMenu constructor. This is a clear indication of code duplication, which can be a code smell that makes the code harder to maintain and increases the risk of introducing bugs.

To refactor the duplicate code smell in the RecentFilesProvider class, we can extract the common functionality of adding menu items to a separate method and then call that method twice in the update method.

In the refactored code, we extract the code for creating the menu items into a separate method createMenuItems. This method takes a list of menu items, the recent file history, and the ActionListener and ChangeListener as arguments. It then loops through the recent file history, creates a JMenuItem for each entry, and adds it to the list of menu items. We also extract the code for adding the menu items to a separate method addMenuItems.

**Team Contributions:**

* Shivani Talatam
  + Analyzed jEdit and found the code smells
  + Refactored the code smells for jEdit
  + Effort: 30%
* Zamaan Bawa
  + Analyzed MangoDB repository and found code smells
  + Refactored code smells for MangoDB
  + Effort: 40%
* Pritul Dave
  + Analyzed jEdit and found the code smells
  + Refactored the code smells for jEdit
  + Effort: 30%