CS 6356 Software Maintenance, Evolution & Re-engineering

Assignment 3

TEAM 02

Team Members:
1. Rohit Kriplani
2. Yash Shyamsunder Pratapwar

The tool used to measure Cohesion and Coupling:

CodeMR (https://www.codemr.co.uk/)

CodeMR is a software quality and static code analysis tool that provides developers and architects with a comprehensive understanding of their codebase. It analyzes the source code of software systems written in various programming languages, such as Java, C#, and C++, to identify and highlight potential issues and defects in the software architecture and code.

CodeMR includes the measurement of coupling and cohesion as part of its architectural software quality and static code analysis capabilities. It can analyze code to identify areas of high coupling and low cohesion, which can indicate potential architectural and design issues that may impact the quality and maintainability of the software. The tool can provide visual representations of these metrics, as well as suggestions for how to improve them.

By providing insight into coupling and cohesion, CodeMR enables developers and architects to make informed decisions about the architecture and design of their software, with the goal of improving its quality, maintainability, and performance.

Metrics selected:

1. Lack of Cohesion in Methods (LCOM)

   It is a software metric that measures the degree of cohesion among the methods of a class. It calculates the number of method pairs that do not share instance variables, divided by the total number of method pairs. The value of LCOM ranges from 0 to 1, with 0 indicating perfect cohesion (i.e., all methods use the same instance variables) and 1 indicating a complete lack of cohesion (i.e., each method uses different instance variables). A higher value of LCOM indicates that the class is less cohesive and may be an indication of potential design issues.

2. Tight Class Cohesion (TCC)

   It is a metric used to measure the cohesion of a software class. TCC calculates the similarity of the types of methods called by a class. If a class calls methods of similar types, then the class is considered to have high cohesion. On the other hand, if a class calls methods of different types, then the class is considered to have low cohesion. TCC calculates this similarity by dividing the number of method pairs that have the same parameter types by the total number of method pairs in the class. A higher TCC value indicates higher cohesion, while a lower TCC value indicates lower cohesion.

3. Coupling Between Objects

   It is a metric that measures the number of classes a particular class is linked to. It considers the classes that use the methods or attributes of the given class, as well as the classes whose attributes or methods are used by the given class. It excludes inheritance relationships. The CBO metric is an indicator of the class's reusability and testability. High coupling signifies that the class is more complex to maintain as changes made in other classes may also require modifications in that class. This results in reduced reusability and increased testing effort.

4. The Response For a Class (RFC)

   It is a metric that counts the number of methods that can be called in response to a public message received by an object of that class. This includes all the methods called in a method's call graph. If the RFC value of a class is high, it implies that the class is more complex and may have higher coupling with other classes. As a result, more effort may be required for testing and maintenance. A high RFC value can lead to code that is more challenging to maintain, which in turn may result in decreased reusability.

Coupling

Coupling between objects [CBO]

- Highest coupling
  - jEdit.java
  - View.java
- Lowest coupling
  - jEditMode.java
  - Marker.java

Response for a class [RFC]

- Highest coupling
  - jEdit.java
  - View.java
- Lowest coupling
  - Marker.java
  - OptionGroup.java

jEdit.java - why is there a high coupling?

High coupling occurs in a software system when there are tight interdependencies between its components. In the case of multiple nested and inherited classes, this often leads to high coupling because changes made to one class can have a ripple effect throughout the system, impacting many other classes that depend on it. This is because nested classes are often used to define more minor, more specialized behaviors within a larger class, while inherited classes build on the functionality of their parent class. As a result, changes made to a nested or inherited class can have wide-ranging effects on the behavior of the larger system, leading to a higher degree of coupling between its components. This can make the system more challenging to maintain, test, and extend over time, as changes to one component may require modifications to many others.

Another type of coupling that is present in this class is control coupling. Control coupling occurs when a method receives a parameter that tells it what action to perform. For example, the method closeBuffer is called with two parameters editPane and buffer. This method has a switch statement that checks the value of the bufferSetManager.getScope(). Depending on the value, it performs different actions, such as removing the buffer from the buffer set manager, closing the buffer, or updating the buffer's markers file. The method _closeBuffer is also called from within the closeBuffer method, which further adds to the control coupling.

View.java - Why there is high coupling

it is difficult to determine the exact type of coupling as it depends on the context and the rest of the codebase. However, some observations can be made:

There are several inner classes present, which may indicate a higher degree of coupling between these classes and the outer class.

Some of the inner classes implement listener interfaces, which suggests that they are designed to interact with other parts of the codebase.

The use of the "EditPane" class suggests that there may be coupling between this code and other parts of the codebase that use the same class.

In conclusion, the type of coupling present in this code cannot be determined definitively without additional context and a deeper analysis of the codebase.

Hence, content coupling is the most prominent in this class.

LOW →

jEditMode.java - why low coupling

JEditMode is designed to have low coupling, which means that it has minimal dependencies on other classes or modules in the system. To achieve this, JEditMode does not rely on any external classes, except for a call to the Log class, which is part of the same package. This ensures that JEditMode does not introduce any coupling with external modules or classes.

Moreover, JEditMode has no public methods and only overrides methods from its parent class, Mode. This makes it easier to maintain, test, and modify the code as changes in one module or class are less likely to affect others. Additionally, JEditMode uses only basic data types or classes from the java.lang package, which is part of the core Java API and does not introduce any coupling with external libraries or modules.

In summary, JEditMode is designed to have low coupling by minimizing its dependencies on other classes or modules in the system, using local dependencies, and using only basic data types or classes from the core Java API.

Marker.java - why low coupling

The Marker class has low coupling because it has a very focused purpose, which is to represent a bookmark in a Buffer. The class has

a limited set of attributes and methods that are all related to this purpose. For instance, it has a constructor to initialize the object with a Buffer, shortcut character, and offset. It also has methods to get and set the shortcut character, get and set the position of the marker, and create and remove a Position object associated with the marker. All these methods have a direct relationship with the primary goal of the Marker class and do not have any extraneous side effects or dependencies on other parts of the system.

This design approach results in an easily understandable and maintainable Marker class, which can be modified without affecting other parts of the system. As a result, the class can be considered as an example of low coupling.

OptionGroup.java - why low coupling

In the given class OptionGroup, there is low coupling because it has limited dependencies on other classes or modules. Coupling refers to the level of interdependence between different components of a software system. Low coupling indicates that a module has minimal dependencies on other modules and can function independently.

In the case of OptionGroup, it is a simple class that represents a set of options displayed in one branch of a dialog box. It has only a few dependencies, including a Vector data structure to store its members and a StringTokenizer class to tokenize a string of option names. Other than that, it only depends on jEdit - a text editor that provides properties for each option pane's label.

The class's limited dependencies make it highly modular and easier to maintain, test, and modify. It is also more flexible, as changes to other modules are less likely to affect OptionGroup, and changes to OptionGroup are less likely to affect other modules. This allows for better code reuse and more efficient development.

Overall, the low coupling is a desirable characteristic in software design as it promotes modularity, flexibility, and scalability. OptionGroup is an excellent example of a well-designed, loosely coupled class.

—----------

Difference between classes with highest coupling and lowest coupling

Classes with control coupling have a higher degree of interdependence than classes with low coupling. This means that changes to one class are likely to require changes to other classes as well. On the other hand, classes with content coupling have a moderate degree of interdependence, with changes to one class potentially affecting other classes that share similar data or methods.

The differences between these classes can lead to different levels of maintainability, scalability, and flexibility in the software system. Classes with low coupling are generally more modular and easier to maintain, as changes to one class do not affect the behavior of other classes. However, they may also require more code duplication or a higher level of abstraction to achieve the desired functionality.

Classes with control coupling may be more challenging to maintain and scale, as changes to one class may require changes to other classes in the system. However, they can also provide more flexibility in the system's behavior, as changes to one class can have ripple effects throughout the system. Classes with content coupling strike a balance between these two extremes, offering some flexibility while still maintaining a level of modularity and ease of maintenance.

type of coupling you observe, and contrast why it is not another type

I observed content coupling and control coupling because of one class has direct access to the internal data of another class, leading to tight dependencies between the two classes. Also, one class controls the behavior of another class by passing instructions or signals to it, leading to a high degree of interdependence between the two classes.

Why it is not global coupling - because there are no functions that are using global data.

Why it is not stamp coupling - because there are no functions that share a composite data structure and use only a part of it.

Why it is not Data coupling - because the manner or degree by which one class influences the execution of another class is low.

COHESION

- We used the following two metrics provided by the CodeMR to measure the degree of cohesion in the jEdit system
    - Lack of Cohesion Among Methods
    - Lack of Tight Class Cohesion

- Classes with a high value of LCAM indicate low cohesion and they are as given below:
    - View.java

      The View class had a low level of cohesion. The class contains methods that are related to managing and interacting with the user interface of the jEdit text editor. These methods include getting and setting the current buffer, managing the edit panes, getting the text area, and managing the view configuration.

      While these methods are related to the same general task of managing the user interface, they could potentially be grouped into more specific, cohesive classes. For example, the methods related to managing the edit panes could be grouped into an EditPaneManager class, while the methods related to managing the view configuration could be grouped into a ViewConfigManager class.

      Overall, the cohesion of the View class is low, and could potentially benefit from some refactoring to improve its organization and modularity.

      The class View exhibits functional cohesion. Functional cohesion refers to the situation where the methods within a module (in this case, the View.java file) are grouped together to perform a single, well-defined task or responsibility, which is to provide a view for the user to interact with the document. This is consistent when investigated the source code of the View.java file.

    - TextArea.java

      The TextArea.java file in jEdit exhibits moderately low functional cohesion because all the methods and variables within the class are not directly related to the functionality of a text area. Not all the methods within the class are responsible for managing the text area's content, selection, caret position, and style.

      The variables within the class are related to the text area's state and behavior. For example, there are variables that store the text area's content, selection, and style information. There are also methods that modify these variables to update the text area's appearance and behavior.

      After understanding the source code, it can be concluded that the TextArea exhibits sequential cohesion.

- Classes with a low value of LCAM indicate high cohesion and they are as given below:
    - ViewOptionPane.java

      The ViewOptionPane class extends the AbstractOptionPane class and contains a constructor and an _init() method that initializes the view options panel with various components, including checkboxes, buttons, and a text field.

The purpose of this class is to create a panel that contains various options related to the view of the application, such as dock layout, floatable toolbars, abbreviate pathnames, and more.

As for the type of cohesion, this code exhibits functional cohesion, where all the components are related to the same functionality or purpose, which is the view options of the application.

Looking at the above points, it can be said that the ViewOptionPane class exhibits high cohesion, as its methods and attributes seem to be closely related to this common purpose.

- EditingOptionPane.java

  This class has a clear responsibility of managing and displaying options related to editing settings in a text editor. All the methods and instance variables are related to this responsibility, such as ModeSettingsPane, EditModesPane, and UndoPane which are all involved in providing different settings related to text editing.

  Furthermore, the class is not doing any unrelated functionality and doesn't have any extraneous instance variables or methods that do not contribute to its primary responsibility.

  Thus, the EditingOptionPane class appears to have high cohesion.

  Based on the code you provided, it appears that the EditingOptionPane class exhibits functional cohesion, which is a type of high cohesion.

  Functional cohesion is present when the methods within a class are related by performing a similar function or working towards a common purpose. In this case, the class has three private instance variables editModes, modeSettings, and undoSettings, and all of the methods in the class interact with these variables to create a user interface for editing settings.

  Overall, the methods in the class are focused on providing functionality for the editing user interface, and the instance variables are used consistently throughout the class. This demonstrates functional cohesion, which is generally considered a desirable characteristic of well-designed code.

- Classes with a high value of LoTCC indicate low cohesion and they are as given below:
  - Types.java

    The Types class appears to exhibit low cohesion. Cohesion refers to the degree to which the elements of a module (e.g., class, function) are related to each other. High cohesion indicates that the elements are strongly related and work together towards a single, well-defined purpose, whereas low cohesion indicates that the elements are weakly related and serve multiple, potentially unrelated purposes.

    In the case of the Types class, there are several methods and constants that appear to have different purposes and do not necessarily work together towards a single, well-defined goal. For example, the class includes methods for checking whether one type is assignable to another (isJavaAssignable, isJavaBoxTypesAssignable, isBshAssignable, etc.), a method for obtaining the types of a given set of objects (getTypes), and several

constants that appear to be related to the above methods (JAVA_BASE_ASSIGNABLE, JAVA_BOX_TYPES_ASSIGNABLE, etc.).

Overall, the class appears to have several disparate methods and constants that serve different purposes and may not be strongly related to each other. This suggests that the Types class exhibits low cohesion.

- CollectionManager.java

  The cohesion of the CollectionManager class appears to be relatively low because it contains multiple methods with distinct responsibilities, such as isBshIterable(), getBshIterator(), isMap(), getFromMap(), and putInMap(). Although these methods are related to collections and iteration, they don't necessarily share a common theme or purpose. Additionally, the class includes nested classes and some static methods that further increase its complexity and reduce its cohesion.

  It's important to note that cohesion is a matter of degree, and it's possible to argue that the CollectionManager class exhibits some level of cohesion because all of its methods relate to collections and iteration in some way. However, in general, a high-cohesion class is one that contains methods with closely related responsibilities, while a low-cohesion class is one that contains methods with unrelated or loosely related responsibilities.

- Classes with a low value of LoTCC indicate high cohesion and they are as given below:
  - EditAction.java

    This class appears to exhibit functional cohesion, as all the methods and fields are related to defining and executing an edit action. Specifically, the class contains methods for retrieving the label, mouse-over text, and tooltip for the action, as well as determining whether the action is a toggle, whether it should be repeated, recorded, or remembered as the most recently invoked action, and obtaining the BeanShell code that will replay the action. The invoke() method is also present, which is the main method for executing the action. All of these methods and fields relate to the same task of defining and executing an edit action, indicating functional cohesion.

    It exhibits high cohesion with the kind being functional cohesion.

  - TextAreaDialog.java

    TextAreaDialog class is used to create a dialog box with a text area. Based on a brief analysis of the code, it seems that the class exhibits high cohesion.

    The methods in the TextAreaDialog class are focused on a single responsibility, which is to create and manage a dialog box with a text area. All of the methods are related to this task, such as the constructors that initialize the dialog box with appropriate components, the init() method that sets up the dialog box, and the ok() and cancel() methods that handle user input.

    Therefore, the kind of cohesion that is exhibited in the TextAreaDialog class is functional cohesion, which means that the methods are organized based on the functionality they provide and work together to achieve a single task or responsibility.

Work:

Rohit worked on the measuring the degree of cohesion while Yash worked on measuring the degree of coupling.

Contribution Information:

Rohit Kriplani - 50%

Yash Shyamsunder Pratapwar - 50%

Total – 100%