**CS 6356: Software Maintenance, Evolution & Re-engineering**

**Assignment 3: Coupling and Cohesion**

**TEAM 17**

## Team Members:

1. Shivani Talatam
2. Zamaan Bawa
3. Pritul Dave

**GENERAL DEFINITIONS:**

**Coupling:**

A measure of how closely connected two routines or modules are.

The strength of the relationships between models:

High coupling:

Modules should ideally be independent of other modules. In high coupling, the modules would know far too much about the other modules.This can reduce the strength of the individual modules.

Low coupling:

This is the desired approach for coupling as each module can focus on their own individual function. Plus it is easier to work with and modify.

Types of coupling from bad to good:

* Content: Directly refers to another module
* Common: Uses global elements
* Control: One module passes an element of control to another module
* Stamp: One module passes more data than needed to another module
* Data: Only the required data is passed from one module to another

**Cohesion:**

Degree to which elements of a module belong together.

High cohesion:

This is the more desirable approach since it is related to many desirable traits of good software: *Robustness*, *Reliability*, *Reusability* and *Understandability*.

Low cohesion:

This is the opposite: Difficult to maintain, test, reuse and understand.

Types of cohesion from bad to good:

* Coincidental: Elements perform multiple, unrelated actions
* Logical: A module that performs multiple actions, one of which is calling itself. Parts of it are related in a logical and not functional way.
* Temporal: Modules that perform a series of actions that are related by time.
* Procedural: Elements of a component are only related to ensure a particular order of execution.
* Communicational: Action based on ordering steps that are all on the same data.
* Informational: Performs a number of actions. Each one is independent and has its own entry point. Actions are performed on a shared data structure.
* Functional: Module that performs a single action or achieves a single goal.

**TOOL USED :**

Metrics Reloaded for IntelliJ

Inspect Code - Intellij

**3 CLASSES FOR COHESION:**

For cohesion, the first metric we will measure is **communicational**. A good example of this would be the TextArea.java class located in org/gjt/sp/jedit/textarea. We used this class in the second homework assignment and had a fair bit of time to examine this lengthy class, but when using Metrics Reloaded, we found this had the second highest weighted metric complexity (WMC), but was one of the lower options for average operation complexity (OCavg). We ran this test by using the complexity metrics option in the menu. The exact values were 799 for the WMC and 2.93 for the OCavg. The class uses good cohesion since the methods and other content in this class are all related to the main function to create a text area for the user to access JEdit.The reason why we thought this class was a good example of communicational cohesion was because the methods all worked together to form the text area. Looking at the code, the initial TextArea constructor is calling multiple methods that are being declared in the same class. TextAreaPainter() and many of the scrolling features are created in this class. Besides this, many other methods utilize different methods that are also declared here which is a form of communication inside the class. All the methods have a purpose to create the text area and can do so by calling each other.

Another set of classes that had really good cohesion were the various widget factory classes. There were other trivial classes, but we are not going to look into those as per the requirements of the assignment. The widget factory classes had a OCavg of 1.00 and a WMC of 1. These were the lowest possible amounts a class could get using this tool. The specific one we will be examining will be the BufferSetWidgetFactory. This is located in the org/gjt/sp/jedit/statusbar directory. This class also uses communicational cohesion since the initial function getWidget creates an object bufferSetWidget which calls the static class BufferSetWidget which has all the different methods needed to create the buffer set widget. In the constructor class, we have add and remove notify methods to add to the bus.There are also mouse listeners and methods to change/update properties.Besides it being communicational, it can also be considered functional since the entire goal is to set up the buffer set widget which is the basis of functional cohesion as that revolves around a program that focuses on a single action or goal which this does fulfill.

The last class we will look at for high cohesion will be the AddAbbrevDialog.java class which is located in org/gjt/sp/jedit/gui. In our metrics test, this also had fairly low results. The OCavg was 2 and the WMC was also 2. Looking at this class, we can see how its methods are all related to the goal of adding an abbreviation to the dialog. In the class, there are private variables and other classes with methods that are declared to add to the end goal. This is also an example of communicational cohesion.The methods declared in the Action and Key Handler classes are the actionPerformed and keyPressed ones. These are relatively simple methods that are called in the constructor of the AddAbbrevDialog. Looking at the other forms of cohesion, this could possibly be considered to be functional. The additional classes felt informational, but the primary AddAbbrevDialog is dependent on both of those sub-classes. The ultimate goal here is to add abbreviated dialog which all these methods and classes work to fulfill.

Now that we have looked at the examples of high cohesion, we will be looking at low cohesion examples. These will be the classes that feature high values for complex operations. The primary metric we will be trying to use here is logical because we predict many of the classes will be trying to accomplish one goal, but will have many actions that can add bloat to the class and reduce the functionality.

The first class we will look at is the ParseTokenManager.java class. This is located in the org/gjt/sp/jedit/bsh directory. The OCavg came out to be the highest at 14.3 and its WMC is 718.The maximum operational complexity was also the highest at 241.This class looks to have a very low cohesion since it seems to be a dump for various methods to parse. This is a good example of logical cohesion since the methods all seem to work together logically, but they all serve different purposes for parsing. Unlike the classes above, each method works to solve something slightly different.The readability of this code is also reduced greatly due to the bulk code dump. Many methods also are bloated and should be reduced.

The next class we will look at is the ExtendedGridLayout.java class. This is located in org/gjt/sp/jedit/gui. IT’s OCavg came out to 8.00 and WMC was 136. Its maximum operational complexity was 49. Unlike the ParseTokenManager class, the complexity was not nearly as high, but this class also features logical cohesion. The methods in this class primarily revolve around general layout modifications. The coding practice to just bunch all the layout methods into one large class is not good. Many methods are empty and some are very bloated. An example of an empty method is removeLayoutComponent and a bloated method is getSize which has over 300 lines of code. Generally, the logic is there to keep these methods together, but it is not functional cohesion since there is not one action or goal that all these methods are working towards. Besides showing signs of logical cohesion, it also shows coincidental cohesion since there are methods that just are not related. Some examples are the getSize or buildGrid methods in comparison to the layout methods. It makes sense that the layout methods are needed to work with the getSize or buildGrid methods, but this class could easily be split up into 3 or even 4 classes that would make the content much more relevant.

The last class we will look at for low cohesion is CodeWriter.java. The results for this class were 7.35 for the OCavg and 228 for the WMC. The maximum operational complexity was 59. This class is located in org/gjt/sp/jedit/bsh/org/objectweb/asm. This is another large class with nearly 2000 lines of code. Many methods work logically, but functionally, this class is bad. This is another good example of logical cohesion since the methods and variables created in this class are all made to create the code writer, but similar to the above class, the methods could be divided into smaller and more readable classes.The first nearly 300 lines of code are just variable declarations with intense commenting. The constructor itself isn’t even declared until line 490. It would be ideal to declare the variables in another class and then call them when needed. The ideal situation would be to have functional cohesion. This means the CodeWriter class would need the essential methods to fulfill the task. Looking further into the code, we have methods to implement the interface which makes sense. The problem is that after these methods are created, we have multiple “utility methods” being declared which adds to the bloat. Logically, it can make sense to incorporate all these together, but it just isn’t functional.

Now that we have examined three classes that feature high cohesion and three classes that feature low cohesion, it is safe to say the high cohesion classes are objectively easier to understand and work with. Comparing BufferSetWidgetFactory (BSWF) with ParseTokenManager (PTM) is pretty interesting since we have the lowest versus the highest in terms of the complexity metrics measured using the Metrics Reloaded tool.The BSWF class is readable and features actions/methods that are all related. They all work together to fulfill the goal. We determined this was communicational cohesion, but it showed signs of functional cohesion since there was a single goal to create the buffer set widget. The PTM, on the other hand, had many methods that had similar names and functions, but the goal for each of them was different. A majority of the methods all revolved around parsing tokens, but it was not wise to keep all of them in one class. It is the equivalent of making a Course class and then having methods for each and every course in the institution in this one class. It would be much wiser to split the class up and divide it into subclasses that were pertinent to a certain subject or even a topic in the subject, The same could be applied here. We saw this exact implementation with the BSWF. As mentioned above, there were many classes for the different widgets. Instead of having each widget implemented in one single class, it divided each widget into their own class. This was much better coding practice, hence why the cohesion was much higher with those.

**3 CLASSES FOR COUPLING:**

Coupling between objects metric:

In software engineering, coupling refers to the level of interdependence between different modules or components in a software system. Object-oriented programming languages represent the software system as a collection of objects that interact with each other to achieve a certain goal. The degree of coupling between objects in the system can have a significant impact on the maintainability, reliability, and scalability of the software.

To measure the level of coupling between objects, various metrics are used. One such metric is the Coupling Between Objects (CBO), which counts the number of classes to which a class is coupled. A high CBO value indicates that a class is heavily dependent on other classes in the system, whereas a low CBO value indicates that a class is relatively independent.

In jEdit, the class jEdit.java has the highest CBO value of 387, indicating that it is heavily dependent on other classes in the system. This means that changes to jEdit.java may have a significant impact on the behavior of other classes. Similarly, the classes jeditBuffer.java and TextArea.java have CBO values of 115 and 121, respectively, indicating that they are also highly coupled to other classes.

On the other hand, the class Abbrevs.java has a low CBO value of 15, and the class ActionSet.java has a CBO value of 21, indicating that they are less dependent on other classes in the system. Furthermore, you mentioned that the class KeymapMigration.java has the lowest CBO value of 4, indicating that it has the least number of dependencies on other classes in the system.

Classes with high CBO (jEdit.java and jedit.Buffer.java) have many other classes depending on them, while classes with low CBO (KeymapMigration.java and Abbrevs.java) have fewer dependencies on other classes.

Response for a class metric:

The methods of a class represent the operations that can be performed on objects of that class. The Response for a Class (RFC) metric measures the number of methods within a class that can be executed in response to an external request or event.

A high RFC value for a class can indicate that it has too many responsibilities and is taking on too much functionality, which can make it more difficult to manage and maintain over time. When a class has a high RFC value, it can be challenging to understand its behavior, debug issues, and make changes without unintended consequences.

In contrast, a low RFC value can indicate that a class has a well-defined responsibility and is easier to manage and maintain. When a class has a low RFC value, it is often easier to understand and test its behavior, and changes to the class are less likely to have unintended consequences. Therefore, low RFC values are often seen as a sign of good software design, and classes with low RFC values are generally considered to be more modular and maintainable.

The class jEdit.java has the highest RFC value of 568, indicating that it has a large number of methods that can be executed in response to external requests or events. Similarly, the classes TextArea.java and jeditBuffer.java have RFC values of 530 and 246, respectively, indicating that they are also highly responsive to external requests. These high RFC values can be a sign of high coupling, indicating that these classes may be taking on too many responsibilities and may benefit from being refactored into smaller, more focused classes.

Conversely, the class KeymapMigration.java has the lowest RFC value of 19, and the classes Abbrevs.java and ActionSet.java have RFC values of 86 and 46, respectively, indicating that they have fewer methods that can be executed in response to external requests. These low RFC values can be a sign of low coupling, indicating that these classes have a narrower, more well-defined responsibility, and are easier to manage and maintain.

Classes with high RFC (jEdit.java and TextArea.java) have many responsibilities, while classes with low RFC (KeymapMigration.java and Abbrevs.java) have fewer responsibilities.

Summary for coupling metrics:

|  |  |  |  |
| --- | --- | --- | --- |
| **Class** | **CBO** | **RFC** | **Level of Coupling** |
| jEdit.java | 387 | 586 | Highly Coupling |
| TextArea.java | 121 | 530 |
| JeditBuffer.java | 115 | 246 |
| Abbrevs.java | 15 | 86 | Low Coupling |
| ActionSet.java | 21 | 46 |
| KeymapMigration.java | 4 | 19 |

Class paths:

|  |  |
| --- | --- |
| jEdit.java | org/gjt/sp/jedit/jEdit.java |
| TextArea.java | org/gjt/sp/jedit/textarea/TextArea.java |
| JeditBuffer.java | org/gjt/sp/jedit/buffer/JEditBuffer.java |
| Abbrevs.java | org/gjt/sp/jedit/Abbrevs.java |
| ActionSet.java | org/gjt/sp/jedit/ActionSet.java |
| KeymapMigration.java | org/jedit/migration/KeymapMigration.java |

The type of coupling observed by us is Data Coupling.

* Content coupling is not observed in these classes as it measures the direct referencing of one class's code from another class, while the coupling in these classes is measured by dependencies between classes and objects.
* Common coupling is also not observed as it measures the sharing of global data, which is not present in these classes.
* Control coupling is not explicitly observed, but it could be present in the classes with high RFC, as it measures dependencies based on control flow statements

.

* Stamp coupling is not observed as it measures dependencies based on the physical layout of a system, which is not relevant to the coupling metrics used in this analysis.
* Data coupling is observed in these classes, as it measures dependencies based on the passing of data between classes. For example, the number of parameters passed between classes can indicate the level of data coupling between them.

**PATH OF THE METRICS IN THE REPOSITORY:**

<https://github.com/pritul2/jEditTeam17/tree/main/Coupling_Cohesion_Metrics>

**TEAM 17 CONTRIBUTIONS:**

Pritul Dave pmd2200000

* Obtaing the coupling metrics from Intelij Inspect Code and Metrics Reload

• Effort: 30%

Shivani Talatam sxt220047

* Analyzing the High Coupling and Low Coupling from obtained metrics

• Effort: 30%

Zamaan Bawa zsb160030

* Obtaing the cohesion metrics from Metrics Reload
* Analyzing the High Cohesion and Low Cohesion from obtained metrics

• Effort: 40%

Total Effort:

• 100%