Name: YU QIU

Username: act19yq

Software Reengineering Report

1. System analysis

Byte Code Engineering Library (BCEL) is a large project that sponsored by Apache Software Foundation. As a part of Jakarta project, BCEL provides a convenience method to deal with java class files in JVM. This project was developed and donated to Apache on 2001 and now it has existed for 20 years, so this project is a very large legacy system.

This section is going to analyze the BCEL project in terms of using the techniques introduced in this module.

Firstly, in order to analyze this system, first step is to determine how the project is organized. The basic file structure is given in the figure 1-1 below. The packages classfile, generic and verifier includes large amount of java files, so those files are considered as the key part of this system and provides critical functions.

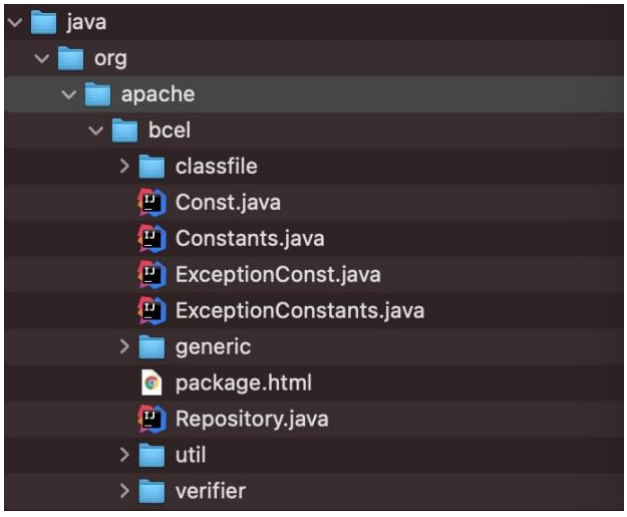

Figure 1-1: system structure

1.1 Bash commands

By using the bash scripts commands, I find that there are 515 java files in this project and the largest file contains 2939 lines of code which is InstConstraintVisitor.java in verifier package. The detailed data of the size of files

distribution is shown in the figure 1-2.

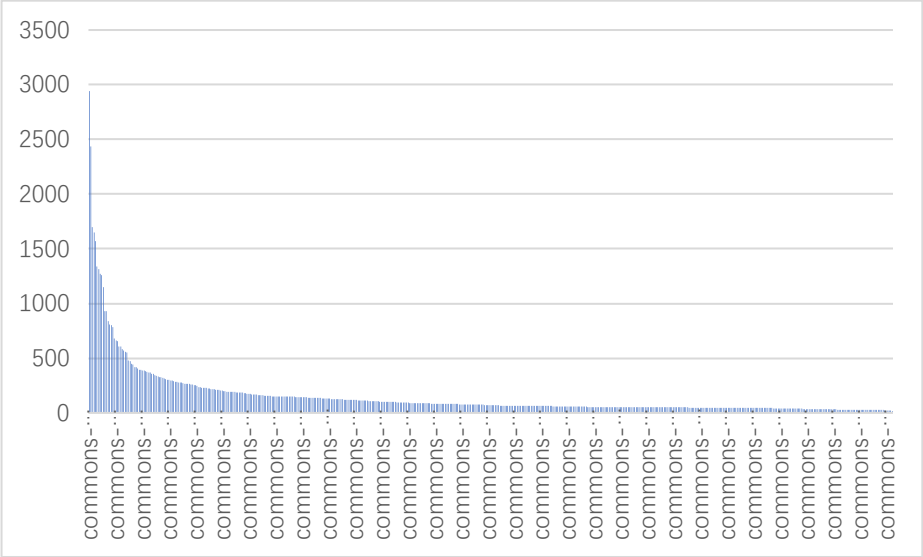

Figure 1-2: file sizes distribution

It is obviously that this system contains some large size classes. The analysis result generated by bash commands shows that there are 10 files have more than thousand lines of code. Generally large size class would make chaos in such a legacy system. It is better to avoid such a god class in re-engineering.

Furthermore, git repository also gives important information like date, contributor, commit message and the versions. By checking the code submit record from git repository, I find that the most frequently changed files. Those files that have been frequently modified may include the fundamental functions of this system. What is more, those files also are highly possible to have potential weakness on design that need to be re-engineered. It is important to focus on those most frequnent changed files. The repository results are attached in file Repository_result.csv. The most frequent modified file while ignoring the pom.xml of maven is Utility.java.

1.2 Class diagram

The class diagram shows the connections between java classes like their inheritance, dependency and other association relationships. Analyzing through class diagrams help us understand the structure of the system or certain parts of the system and find out the super classes and its subclasses. The class diagram could be generated by using the re-engineer toolkit to analyze the fat jar packed by

maven. As this is a large project and the global association between classes is complex, it is hard to involve the graph in this report. The whole call graph is attached in files bcel.dot and Class diagram.png. The image below presents the class diagram inside of the package verifier.

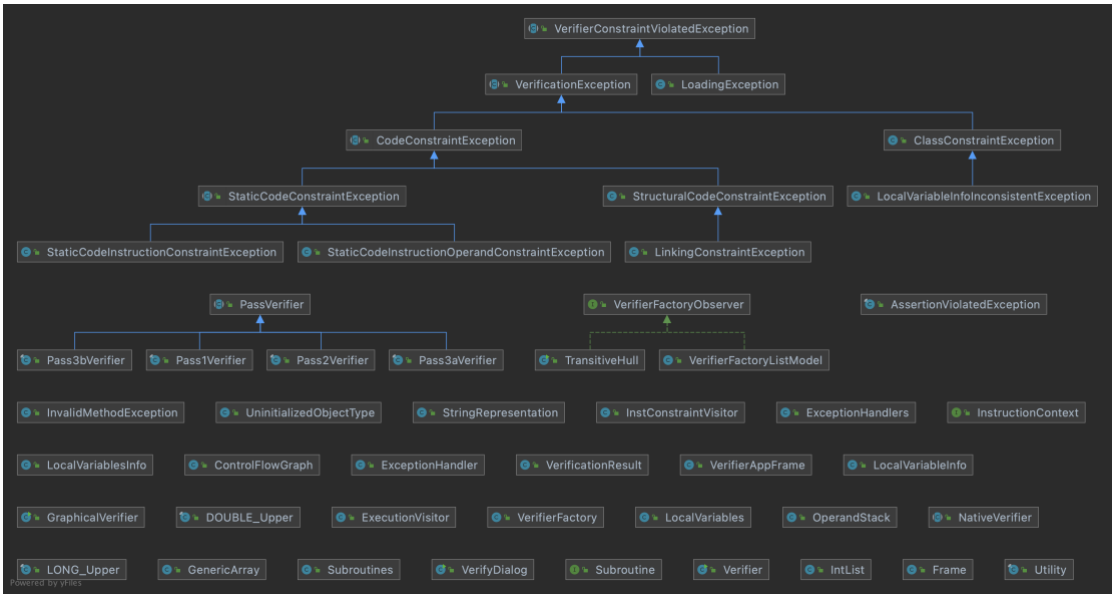

Figure 1-3: class diagram in package verifier

1.3 Call graph

The call graph represents the method calling relationship between each java class in this program. It tracks on each class and shows how it affects on other classes. It helps us understand the process in this program and also find the classes that have never been called.

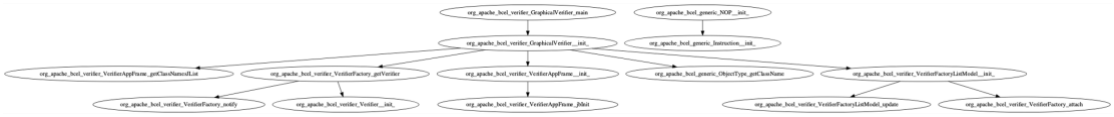

Figure 1-4: partial functions call graph

The most called method is:

org/apache/bcel/verifier/structurals/ExecutionVisitor.stack

1.4 Code clone

The duplicate code means that the different classes or functions that contain same code expression. Too much duplicate code would increase the complexity of the system and waste spaces to store same computations. Generally, an effective method to avoid this kind of situation is to reuse the code such as extract the

common method as a separate class and allows other classes to call it. In order to find if this system contains code duplication, I use the GlobalFileComparison in reengineer-toolkit to produce the similarity of each pair of files in this system. The result is attached in globalFilesComparison.csv. The result shows that many files have a high similarity with others, especially classes under same package. This means that those classes contain codes with same expression but exist in two different places. For example, the INVOKESPECIAL.java and INVOKEVIRTUAL.java under generic package have the highest similarity which is 72%. This means that there must be serious code duplicate problem between those two classes. Moreover, the classes shown below with allied name under generic package all get a same high score on similarity. Those classes should be used to deal with different objects but most of their methods are same expressed. It is better to extract out those method to make the system cleaner.

/Users/rebeccalyu/reengineer/commons-bcel/src/main/java/org/apache/bcel/generic/LDIV.java	/Users/rebeccalyu/reengineer/commons-bcel/src/main/java/org/apache/bcel/generic/LREM.java	0.65217391
/Users/rebeccalyu/reengineer/commons-bcel/src/main/java/org/apache/bcel/generic/LDIV.java	/Users/rebeccalyu/reengineer/commons-bcel/src/main/java/org/apache/bcel/generic/IREM.java	0.65217391
/Users/rebeccalyu/reengineer/commons-bcel/src/main/java/org/apache/bcel/generic/LDIV.java	/Users/rebeccalyu/reengineer/commons-bcel/src/main/java/org/apache/bcel/generic/IDIV.java	0.65217391
/Users/rebeccalyu/reengineer/commons-bcel/src/main/java/org/apache/bcel/generic/LREM.java	/Users/rebeccalyu/reengineer/commons-bcel/src/main/java/org/apache/bcel/generic/IREM.java	0.65217391
/Users/rebeccalyu/reengineer/commons-bcel/src/main/java/org/apache/bcel/generic/LREM.java	/Users/rebeccalyu/reengineer/commons-bcel/src/main/java/org/apache/bcel/generic/IDIV.java	0.65217391
/Users/rebeccalyu/reengineer/commons-bcel/src/main/java/org/apache/bcel/generic/IREM.java	/Users/rebeccalyu/reengineer/commons-bcel/src/main/java/org/apache/bcel/generic/IDIV.java	0.65217391

Figure 1-5: similar classes

In addition, if we just focus on find the position of duplicate code inside two classes, the general method is using dot plot. The x-axis and y-axis on dot plot graph represent two different class respectively. This graph produces a visual able window to highlight the area existing same code expression.

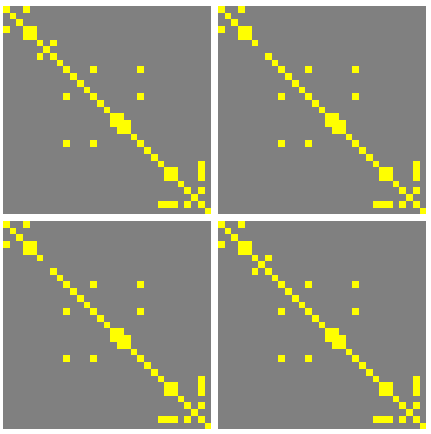

Figure 1-6: dot plot on similar classes

1.5 The computation of appropriate metrics

A good design of system should follow good metrics. The code metrics has measured three indexes in the reengineer toolkit, which are CBO (Coupling Between Objects), DIT (Depth of Inheritance Tree) and WMC (Weight Method Count). The result is stored in classMetrics.csv.

Coupling Between Objects index measures exact data transmission in classes, however, the reengineer toolkit is only based on the call graph. According to the result, some classes like instructionConstants and instructionConst have a very strong connection with other module in the system which the measurement is higher than 100. Coupling represent the dependencies in classes such as control, call and data transfer. High level of coupling may raise the problem that over-coupling caused complex relationship in objects. The coupling of system directly influences the maintenance and upgrade of the program. In order to reduce the coupling level, it requires to find if the method is directly called from original class or method is transferred from class to class. In that case, we have to simplify the transmission process and it also helps make a better relationship on call graph.

Another index is the Depth of Inheritance Tree. It shows how many generations of inheritance relationships exists in each class. Based on analyzing data, the longest inheritance tree is 5. Generally, a long inheritance tree is not suggested as it may include unnecessary middle level that increase the coupling.

1.6 Conclusion

To conclude, BCEL system do exist some bad taste on design. Those designs might lead to the system become bloated and complex consequently the system is becoming difficult to maintain and expend new functions. A good design of system must follow certain principles. For instance, solid Object-Oriented Design Principles firstly ask the system to reduce the code duplicate. Second, classes and methods should have single responsibility. Then, system objects have to be extendable rather than modification. Furthermore, objects could be replaced by their sub classes without affecting system function. Afterwards, interface is better to be used by multiple clients. Lastly, objects should depend upon abstractions, not

concretions.

In order to re-engineer the BCEL system, we have to focus on the certain design weakness of this system.

The first weakness is that this system contains a lot of code duplicates. Based on the data from global file comparison, many classes have a high repetition rate, especially the examples shown in figure 1-5, those classes should be sibling sub classes, but they all have a common method that makes same similarity. In this case, we have to extract the common part in those sibling sub classes and push to their super classes. According to Object-Oriented Design Principles, it is necessary to reuse the code wherever possible to optimize system structure.

Secondly, some oversized class or method is inappropriate in good system design. Those large classes are usually caused by unreasonable abstract, so it reduces the chance for code reusing. For instance, the largest class (**InstConstraintVisitor.java)** in BCEL contains lots of overwritten methods which give similar functions. This weakness is also referred to a bad inheritance relationship. In addition, oversized classes and methods generally contain complex logic inside, the chance of making bugs will sharply increase and also becoming hard to maintain. Besides, a long class usually plays different role and take too much responsibilities. As a result, long class is not in line with the good system design principle that one class only carry single responsibility. Similarly, another serious problem in the large class of BCEL is that author leaves too much unused method under current version. However, the author cannot predict if those method will be used in future, but it indeed increases the redundancy of system.

The last weakness I find in BCEL is that some classes are over coupling. A good design on a large project requires that low coupling and high cohesion.

2. System Reengineering

System reengineer is to adjust the internal structure of the system. The purpose of system reengineer is to improve the comprehensibility of code and reduce the cost of modification without changing the system function. The reengineer processes

aim to rearrange the finished code followed right principle so as to reduce system errors.

BCEL is a large legacy system. The weaknesses of BCEL have been discussed in previous section. In this section, I am going to focus on specific problem and apply a suitable re-engineer strategy to improve the maintainability, readability and testability. The weakness I would like to reengineer is the code clone. As the data from previous section shows that BCEL system contains many large sized classes. One reason that caused those oversized classes is there exist too much duplicate code.

In order to reduce the repeat code, the main idea is to extract the shared methods. However, there are different situation for duplicate codes. For instance, the duplicate code exists in single class or in different classes. Furthermore, the situation for different class also divided into classes are sibling classes that share same parent and classes are irrelevant, but they have same method. Thus, the reengineering work have to combine with the actual situation.

1. While the duplicate code in a same method, this is the most common situation in a large system. The preventive solution to this is that get this method out and makes a new method that allow to call it from origin method
2. If the duplicate code in sibling classes, it means this must be their common features and belongs to their parents. The best solution is to extract the method and put the method into their parent class. Then the sibling sub classes could inherit from parent. This could obviously reduce the code duplicate especially there are many sibling subclasses.
3. When the irrelevant classes have same method. This is really a rare situation, but the solution is same to situation 1. We just need to separate the repeat part to a new class.

The actual practice on re-engineer duplicate code will be committed with this report. However, this is a really huge system so my re-engineer strategy could not cover every corner. The code committed to GitHub just provide ideas on practice. Overall, no matter how we re-engineer with the large class or duplicate code, the

main idea is same, which is divide large objects to small, take single responsibility and clear cohesion between objects. In the same way, in order to guarantee the maintenance of system, programmer should also pay attention to the reengineer principle when developing new functions. It is easy to have a maintainable and expandable program when the program is created.