Project Report on

Reverse Engineering on AssertJ

Vidhan Dholakia
Electrical and Computer Engineering
Queens University
Kingston, Canada
18vbd@queensu.ca

Abstract

Software programs are extensive and complex, often containing hundreds of components where source code may or may not be available. To fully understand the timing of the operation of such a system is a daunting task. To this end, a number of technological innovations have been developed that have been passed down over the decades. It includes a comprehensive overview of these methods, including source code analysis tools, tracking, and model detection. The aim of this project is to discover the complex components of the AssertJ Java library and to implement flexible engineering to simplify system complexity.

Keywords— Reverse engineering, Metrics, Static and Dynamic analysis.

I. Introduction

This part represents a broad introduction to the challenge of reverse engineering software behavior. In general terms, the task is to be able to extract valuable knowledge about the runtime dynamics of a system by studying it, either in terms of its source code or by watching it as it executes. The essence of the information that is reverse-engineered can be broad. Examples involve restrictions on the values of variables, the possible sequences in which events occur or statements are executed, or the order in which a GUI provides the input.

The problem of reverse-engineering behavioral models as a research subject is one of the longest-established in Software Engineering. The subject of predicting machine actions is beset by negative outcomes and seemingly fatal limitations. Also identifying the most trivial properties can be generally undecidable if we want to examine the source code. There are an infinite number of inputs if we want to analyze software executions, and we are faced with the circular problem that we need to know about the actions of the program before we can pick an acceptable sample of executions to analyze.

II. Motivation and Related Work

(a) Reverse Engineering Process.

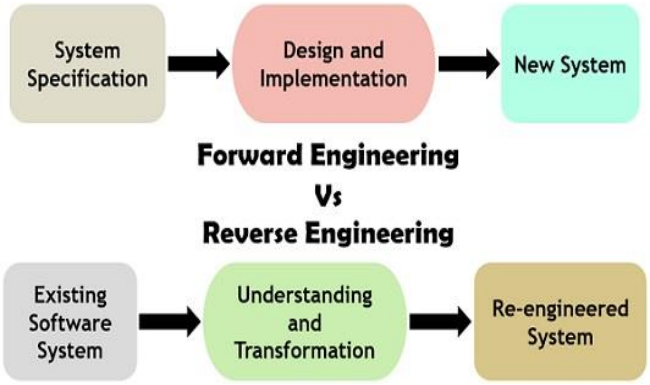


Figure 1. Forward vs Reverse Engineering.

Reverse-engineering begins with an "information collection" process that may include source code review, software execution observation, or any combination of the two. This information often has to be abstracted—it has to be recorded in such a way that it captures the relevant points of interest in the program at a useful level of detail so that the final result is readable and relevant. Lastly, the information gathered has to be used to infer or deduce a behavioral model. This step is mostly automated, but human interaction can be needed.

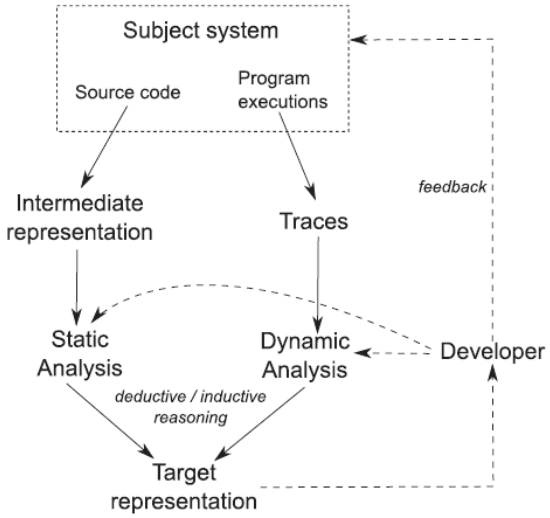


Figure 2. Reverse Engineering Process.

In terms of these steps, all reverse-engineering procedures generating behavioral models can be described. Some depend solely on static research, others on dynamic research, and some combine the two. Some integrate feedback from developers and others are iterative, by requesting additional traces or additional inputs from the developer at each iteration, gradually refining the target representation.

For the rest of this chapter, this schematic view of the reverse-engineering method offers a valuable reference point. It explains where static and dynamic analysis fit in, and how each case can play a role for the developer. This also highlights how processes can be iterative. The final model can be fed back to the reverse engineer who, by considering additional sets of traces or by adding other types of source code analysis, which attempt to refine the result.

(b) Analysis of Behavioural Model

The importance of the behavioral model can be understood by comparing it with the original source code. There are three main reasons for this: Firstly, there is the question of the location of functionality. Depending on the language paradigm, source code relating to a specific facet of functionality can be distributed throughout the system. While only a small fraction of the source code base (e.g., the code in a large drawing package that is responsible for loading a drawing from a file) may be covered by the specific behavior element that we are trying to reverse-engineer, the code in question may not be localized to a particular module or component in the system.

Secondly, there is the related problem of abstraction. Once the appropriate source code has been found, its implementation challenges the understanding of its functional actions. Many variables, such as the choice of programming language, paradigm, architecture, external libraries, etc., depending on the way a specific unit of functionality is encoded in the source code. A software that achieves a specific functionality in one language or paradigm may therefore vary significantly from another.

Thirdly, a static representation is the source code, while its action is intrinsically dynamic. Therefore, the task of discerning program actions inevitably requires the developer to execute the required source code mentally, to keep track of the values of different relevant variables, and to predict the output. This will incur an intractable mental overhead for any non-trivial source code and non-trivial features.

(c) AssertJ System

- AssertJ is one of the most popular Java projects on Github. It is an enormous program, consisting of tens of thousands of lines of code and over 500 sections.

- AssertJ is a Java Library for writing assertions. Assert core provides assertions for JDK standard types and can be used with JUnit, TestNG and other test frameworks. The repository on GitHub is called assertj-core.

III. Research Questions

RQ 1 Why we require Behavioral Model over Source Code?

A behavioural model describes the interactions between artefacts that result in a specific system behaviour, known as a use-case.

- The reason for this is the source code is relevant but to identify the behavior patterns of the software and possible areas of re-engineering, looking at the source code alone cannot convey such information.

- In addition, the source code of the software often contains an overwhelming amount of irrelevant information that we usually do not care about; as we are only interested in the behavior of the system. In that case, behavioral models such as: Class Diagrams, Metric tables and other forms of visualizations are needed to convey such information. Furthermore, high level interaction of software behavior arises from extensive interaction between packages or modules.

RQ 2 Static or Dynamic Analysis?

- Regarding the metric analysis, the aim is to conduct some static or dynamic analysis with the hopes of identifying many weaknesses in the AssertJ program, which inevitably requires re-engineering.

- We just need to perform static analysis to calculate the LOC (Lines of Code) metric for each class in the assertj-core directory. Static analysis will also be used to calculate the weight of each method and the number of nodes in the CFG, as well as the Cyclomatic Complexity of each method.

RQ 3 Limitations faced while Re-Engineering AssertJ?

- A god class monopolizes power over an application by taking on so many duties.

- It's difficult to evolve the application because almost any transition affects this class and multiple roles.

- The problem with assertJ is that there is another God Class named entity, and any changes made to the application's code structure may have negative consequences for the actions of that class and the method.

IV. Approach

Primary goal is to explore the Java based dataset AssertJ. To gain information about the software's function, key components and architectural features. There are 4 expected approaches and expectations in this project.

1. **Understanding the AssertJ system**

The aim is to gain a broad understanding of the system's structure, key components, and architectural features without the use of any software.

2. **Calculate Metrices from static/dynamic analysis.**

Static or dynamic analysis to measure metrics such as LOC that can provide insight into the system's possible vulnerabilities and strengths.

3. **Data Visual Representation.**

Get the insight and understanding the relations between various metrices which can be further useful to apply the concept of reverse engineering on the system.

4. **Applying Reverse Engineering on the system**

This is the final task in which the reverse engineering is to be applied. Expected outcomes are the test cases to check if the refactoring of the structure did not break anything and be incremental with regular commits.

V. Method

a.) Initial Exploration

The goal of the first phase of this re-engineering project is to get a general understanding of Assert J, so we gather information on the software's function, key components, and architectural characteristics. We can accomplish this by implementing either of the two reengineering methodologies: "Skim Documentation" and "Read Code in an Hour." For this project, we will use the “Skimming Documentation” method.

We discovered AssertJ, a Java library that provides a fluent interface for writing assertions, using this tool. Its aim is to make test code more readable and to make test maintenance easier. AssertJ's goal is to provide a comprehensive and user-friendly set of strongly-typed assertions for unit testing. We can use Map-specific statements, for example, to verify the value of a Map.

AssertJ is made up of six different components, each of which has its own git repository with its source code under the creator's name. Guvana, Joda Time, Neo4J, DB, and Swing module are the five distinct components. In addition, Assertion Generator is a feature that allows developers to generate assertion code based on their own classes. Developers can now write statements that are unique to their domain model vocabulary.

Analysis of directory and Code Structure

The readme files and other forms of documentation (such as CODE OF CONDUCT.md and ISSUE TEMPLATE.md, for example) are kept separate from the source folder, which contains all the logic, in AssertJ's directory structure.

Code Maintainability

The structure and readability of assertj-source core's code are consistent and understandable after reading it. We can notice that the majority of classes have detailed comments about the methods' functionality. However, some classes have outdated comments, which makes it difficult to comprehend their functionality. Similarly, according to OORP (Object Oriented Re-engineering Patterns), this increases the need for human dependency because the system becomes more difficult to maintain as code documentation becomes obsolete.

It also doesn't have a lot of "smelly code" like long methods and data classes, indicating that the software is well-designed. However, it is critical to detect and minimize smelly code.

b.) Static Analysis to Calculate metrices (like LOC).

It aids in the detection of any flaws in the AssertJ program that require re-engineering. The most important step is to generate LOC for each assert-core class. For the code, we will use Shell Scripts, and the output will be written to a CSV file, which can then be saved in the dataFiles Directory.

In the second half of the statistical analysis, we will compute the weighted method count for each method, as well as the number of nodes in the CFG and the Cyclomatic Complexity for each method.

The aim is also to try and calculate the metrices using a well-known Java based tool “PMD”. PMD is a free and open-source static source code analyzer that detects and reports on problems in application code. Built-in rule sets are included in PMD, as well as the ability to write custom rules. Because PMD can only process well-formed source files, it does not report compilation errors.

```
echo "Class Name,Lines of Code">"../../../../../../../..dataFiles/LinesofCode.csv"
for file in 'find .-name'*.java'
do
    line=$(wc -l < $file)
   comment1=$(grep")\/\"$file |wc -l)
   comment2=$(grep")\/*"$file |wc -l)
   comment=$(($comment1 + $comment2)
   total=$(($line-$comment))
   echo "$file,$total"

   echo "$file,$total" >> "../../../../../../../..dataFiles/LinesofCode.csv"
   done
```

Figure 3. Shell Script for LOC

Class Name	Lines Of Code
./condition/AnyOf.java	31
./condition/DoesNotHave.java	17
./condition/Not.java	17
./condition/Join.java	38
./condition/AllOf.java	31
./condition/Negative.java	18
./internal/AbstractComparisonStrategy.java	69
./internal/RealNumbers.java	29
./internal/IterableDiff.java	51
./internal/BinaryDiff.java	50
./internal/IntArrays.java	108
./internal/OnFieldsComparator.java	60
./internal/IterableElementComparisonStrategy.java	52
./internal/Dates.java	349
./internal/FloatArrays.java	108
./internal/Characters.java	34
./internal/InputStreamsException.java	9
./internal/Throwables.java	139
./internal/Uris.java	170
./internal/Lists.java	155
./internal/BigDecimals.java	44
./internal/Failures.java	113

Figure 4. LOC Metric

Class Name	Weighted Method Count
org/assertj/core/api/AbstractIterableAssert	250
org/assertj/core/internal/Arrays	190
org/assertj/core/internal/Iterables	179
org/assertj/core/api/AbstractObjectArrayAssert	171
org/assertj/core/api/AtomicReferenceArrayAssert	166
org/assertj/core/api/Assertions	162
org/assertj/core/api/WithAssertions	161
org/assertj/core/internal/Strings	148
org/assertj/core/presentation/StandardRepresentation	131
org/assertj/core/api/Java6Assertions	127

Figure 5. Weighted Method Count Metric

Methods	Number of Nodes	Cyclomatic Complexity
org/assertj/core/internal/DeepDifference.determineDifferences(Ljava/lang/Object;Ljava/lang/Object;Ljava/util/List;Ljava/util/Map;Lorg/assertj/core/internal/TypeComparators;)Ljava/util/List;	621	36
org/assertj/core/util/diff/DiffUtils.parseUnifiedDiff(Ljava/util/List;)Lorg/assertj/core/util/diff/Patch;	420	25
org/assertj/core/util/diff/DiffUtils.processDeltas(Ljava/util/List;Ljava/util/List;I)Ljava/util/List;	350	9
org/assertj/core/presentation/StandardRepresentation.toStringOf(Ljava/lang/Object;)Ljava/lang/String;	345	33
org/assertj/core/util/DateUtil.formatTimeDifference(Ljava/util/Date;Ljava/util/Date;)Ljava/lang/String;	336	24
org/assertj/core/util/diff/myers/MyersDiff.buildPath(Ljava/util/List;Ljava/util/List;)Lorg/assertj/core/util/diff/myers/PathNode;	283	14
org/assertj/core/internal/DeepDifference.deepHashCode(Ljava/lang/Object;)I	214	12
org/assertj/core/util/diff/myers/MyersDiff.buildRevision(Lorg/assertj/core/util/diff/myers/PathNode;Ljava/util/List;Ljava/util/List;)Lorg/assertj/core/util/diff/Patch;	210	14
org/assertj/core/util/diff/DiffUtils.generateUnifiedDiff(Ljava/lang/String;Ljava/lang/String;Ljava/util/List;Lorg/assertj/core/util/diff/Patch;I)Ljava/util/List;	194	5

Figure 6. Cyclomatic Complexity Metric

Method Name	Method Tightness
org/assertj/core/presentation/StandardRepresentation.toStringOf	0.006535948
org/assertj/core/presentation/StandardRepresentation.format	0.005813953
org/assertj/core/internal/Objects.assertIsEqualToIgnoringNullFields	0.005780347
org/assertj/core/internal/Strings.assertContainsSequence	0.005524862
org/assertj/core/internal/DeepDifference.determineDifferences	0.005235602
org/assertj/core/internal/DeepDifference.initStack	0.005208333
org/assertj/core/internal/DeepDifference.deepHashCode	0.005154639
org/assertj/core/internal/Comparables.assertIsBetween	0.004761905
org/assertj/core/util/diff/myers/MyersDiff.buildPath	0.004672897
org/assertj/core/util/diff/myers/MyersDiff.buildRevision	0.003533569

Figure 7. Method Tightness Metric

c.) Dynamic Analysis

Our main concern for the dynamic analysis stage was the analysis of the software as it ran. Based on the results of the metrics we obtained from performing the static analysis, we determined that the top 10 classes with the highest wmc should be further investigated. we narrowed my choices of classes to the top 5 based on the weighted system count. The reason we did this was to narrow the range of my attention to specific classes with a high wmc. The top 10 classes with the highest weighted method count (wmc) in ascending order.

- org/assertj/core/api/AbstractIterableAssert
- org/assertj/core/internal/Arrays
- org/assertj/core/internal/Iterables
- org/assertj/core/api/AbstractObjectArrayAssert
- org/assertj/core/api/AtomicReferenceArrayAssert
- org/assertj/core/api/Assertions
- org/assertj/core/api/WithAssertions
- org/assertj/core/internal/Strings
- org/assertj/core/presentation/StandardRepresentation
- org/assertj/core/api/Java6Assertions

Figure 8. Top 10 classes with highest WMC

The five class names I've highlighted are the ones for which I've generated traces.logs for their respective test-class equivalents. The following test-classes we chose for the tracing are:

org.assertj.core.api.WithAssertions_delegation_Test

org.assertj.core.api.iterable.IterableAssert_extracting_Test

org.assertj.core.api.BDDAssertions_then_Test

org.assertj.core.presentation.StandardRepresentation_toStringOf_Test

org.assertj.core.api.objectarray.ObjectArrayAssert_extracting_Test

Figure 9. Chosen Classes

In this step of the study, the technological challenge was to evaluate the behavior of the test-classes and the software system as a whole. By creating the Trace.java file, we used aspect-oriented programming techniques to log every class method call invoked by the 5 test-classes we chose.

After that, we created the DynamicAnalysis.java class, which reads the contents of all the trace files and stores them in a HashMap. Since it retains individuality, which is suitable for the mission, we chose to use this data structure. The task's aim was to count up all of the classes' occurrences through the five trace files. The knowledge we gleaned from that section was then used to dynamically scale my Class Diagram. The Class Diagram we created was color coded, and it was scaled larger than the other classes. We did this to indicate that certain groups would have a higher call frequency than the others. The LOC (Lines of Code) metric was often used to evaluate which classes have a large number of lines of code. In terms of the metrics used, we believe that the number of lines of code was a useful metric to use because it gave us a clearer idea of which candidates needed to be re-engineered. Because by defining classes with large lines of code, we can use that knowledge to re-engineer the code and make it simpler, more functional, and maintainable, both of which are important criteria for legacy applications like AssertJ. We wrote the ClassDiagramTest.java class to process the target classes and generate the Class Diagram, as we stated in the previous section of the article. Unlike my previous class diagram, this one includes complex analysis and is scaled using the following metrics: total number of occurrences, LOC, and, most importantly, weighted process count.

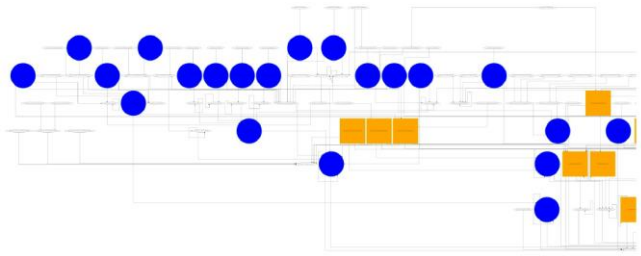


Figure 10. Class Diagram

The graph above depicts a dense network of interactions between various groups. The most important thing to note is that I've scaled the class diagram to identify classes with LOC values greater than or equal to 100.

We did this to see if there was a connection between the number of lines of code and the complexity of a class. According to this report, one notable class appeared in the wmc metric for the test -groups, and that class is org.assertj.core.IterableAssert.

Here are several groups that have more than 500 lines of code. It's worth noting that the four groups highlighted in yellow have a high wmc value and a LOC value of just over half a thousand.

- org.assertj.core.api.AtomicReferenceArrayAssert
- org.assertj.core.internal.Iterables
- org.assertj.core.api.AbstractObjectArrayAssert
- org.assertj.core.internal.DeepDifference (A parent class that I will possibly look into to reduce the lines of code).
- org.assertj.core.internal.DeepDifference$DualKey
- org.assertj.core.internal.DeepDifference$Difference
- org.assertj.core.internal.Arrays
- org.assertj.core.api.AssertionsForInterfaceTypes
- org.assertj.core.api.AssertionsForClassTypes
- org.assertj.core.api.Assertions
- org.assertj.core.api.AbstractIterableAssert
- org.assertj.core.api.Java6Assertions

Figure 11. Groups with more than 500 LOC

This suggests that these classes are not only complex in nature but their complexity is related to the length of code and statements in systems. Notable classes from wmc matrix are the same classes that appeared to have a large LOC number. This would mean that repetitive engineering work, reducing the code line in certain ways for those sections, would certainly reduce the complexity of those sections. A lot Importantly, this will also reduce the amount of unwanted code in the software system.

Groups with a high incidence, on the other hand, were changed to appear in purple boxes, as shown in the class diagram in figure below

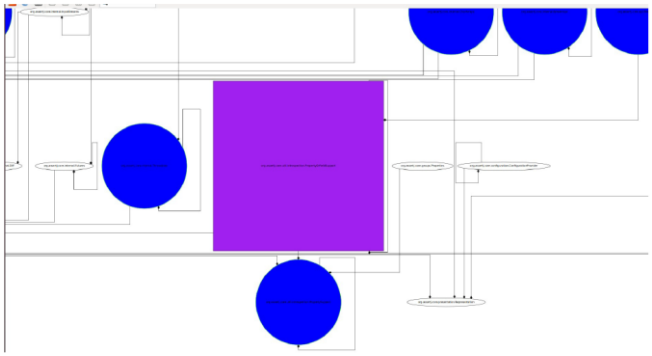


Figure 12. Class Diagram

Boxes in purple indicate classes with high occurrences. In particular, there is the PropertyOrFieldSupport class, which has a large number of associations in particular to the PropertySupport class.

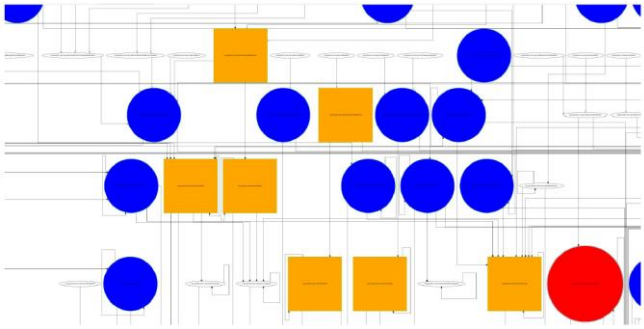


Figure 13. Class Diagram

Many of the complex groups are color-coded in orange in this section of the class diagram. The weighted method count threshold is set for groups with values greater than 100. Iterables, Arrays, and Strings are the classes that occur here where there is a high proportion of inheritance and connections between various classes. Similarly, these classes are among the top ten classes in the AssertJ project with the highest weighted method count values. However, since they have a lot of connections and inheritances, we need to be careful if we try to re-engineer any of those classes, because AssertJ's behavior can change drastically.

In contrast, other categories are similar - Assertions,StandardRepresentations,Java6Assertions, and DeepDifference, which have very high complexity values found in areas where there is a low asset. This means correction will not have a significant impact on the behavior of other classes. However, reducing the complexity of these classes will increase software retention.

d.) Visualization

To generate visualizations, we started by reading the data from the 5 trace-files which we used as part of the dynamic analysis. The 5 trace files, which are files listed up which generated based on the test classes, which are counterparts of the re-engineering we wrote a new class called ParseTraceToCSV.java**,** which parse the contents of the trace file to csv. For each individual trace file, we read the class name and the number of times it's called (occurrence). Afterwards, the total is written to a csv file corresponding to each trace-file.

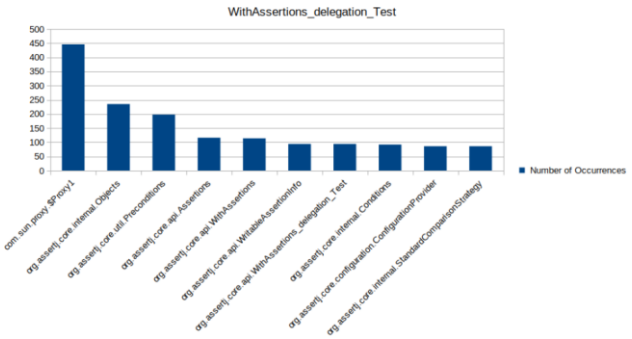


Figure 14. WithAssertions class

This bar chart depicts the ten classes that this test file has commonly executed. Here are the most frequently used classes: Objects, Preconditions, Assertions, and WithAssertions are all types of assertions. Assertions are the second most executed class in this situation, but I know it has a high wmc because of my previous analysis.

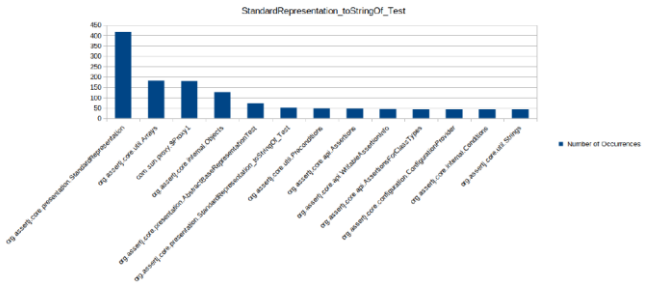


Figure 15. StandardRepresentation class

StandardRepresentation is the highest class that this test-class executes (over 400 times). Similarly, the Arrays class, which is the second highest class executed, has a wmc value of 100 or greater.

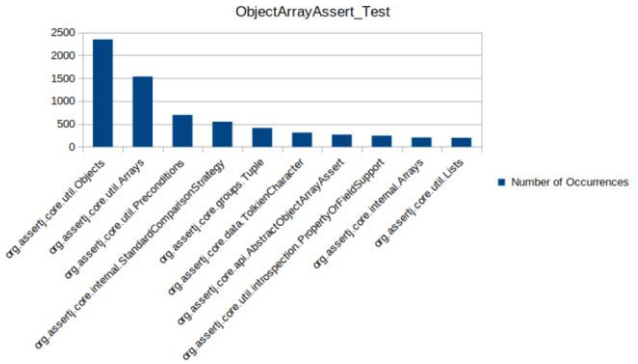


Figure 16. ObjectArrayAssert class

This ObjectArrayAssert Test class, like the other test files, frequently runs ObjectArrayAssert Test.

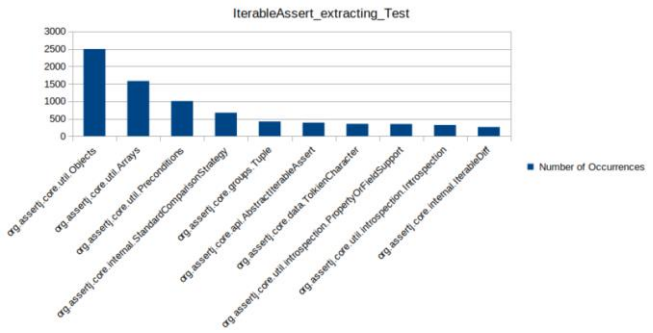


Figure 17. IterableAssert class

As we can observe here, this test class uses AbstractIterableAssert, which, according to my static analysis metrics, has a wmc of 250, which is the largest.

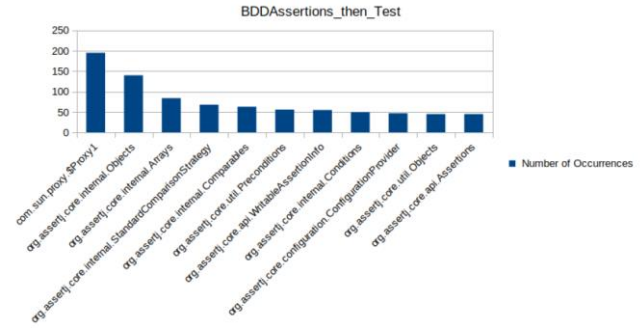


Figure 18. BDDAssertions class

This test class, like the others, calls the Object class regularly (145 times), which is significantly less than the other test classes, which call the Object class more frequently. It also uses the Arrays class from the iterables kit. Based on my static analysis, the class is considered to be very complex and is a potential candidate for code refactoring.

e.) Re-engineering Assert J.

Static and dynamic analysis were the two types of analysis we used in this re-engineering project. Without running the software, we performed static analysis to assess the structure of the source code syntax. The static analysis metrics gave me insight into the classes are complex (based on the weighted method count) and how coherent the class methods are based on the methods with a large number of statements that exceeded or equaled 100. This was accomplished by using the computeTightness method on each method in the AssertJ project. The method was invoked on the software dependency graph, which is made up of a mix of control dependency and dataflow between statements. As a result, we gained insight into the program's dataflow and, more specifically, which classes are considered complex.

After that, we used dynamic analysis to invoke the 5 test-classes by using aspectJ tools to log class executions and method calls. We were able to determine if there was any connection between the metrics we derived from the static analysis and the results of the tracing. In addition, when it came to the visualization of the metrics, we used graph visualizations and the scaling of the class diagram to explain the behavior between the classes and the complex classes.

As a result, I've chosen the following candidates to be re-engineered:

org.assertj.core.internal.DeepDifference

We will refactor the code into two classes, resulting in fewer lines of code and a separation of the class's responsibilities.

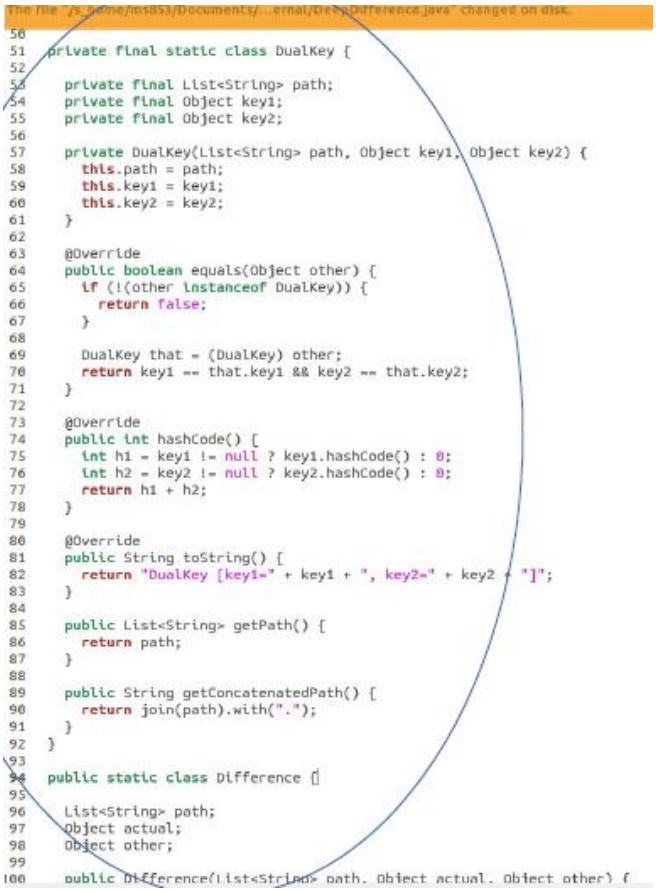
org/assertj/core/presentation/StandardRepresentation

As a result, we refactor the toStringOf method to make it more effective.

For the first re-engineering attempt, we wanted to reduce the number of lines of code as well as the complexity of the DeepDifference class, which is located in the Assertj project's internal package.
we started by analysing the class and discovered that it had two internal classes: ‘DualKey' and ‘Difference.'

Both classes were declared with the following modifiers: **‘public final static class’**

It was necessary to add some abstraction to the DeepDifference class so that it doesn't have to deal with the internal classes and instead uses them as external classes.

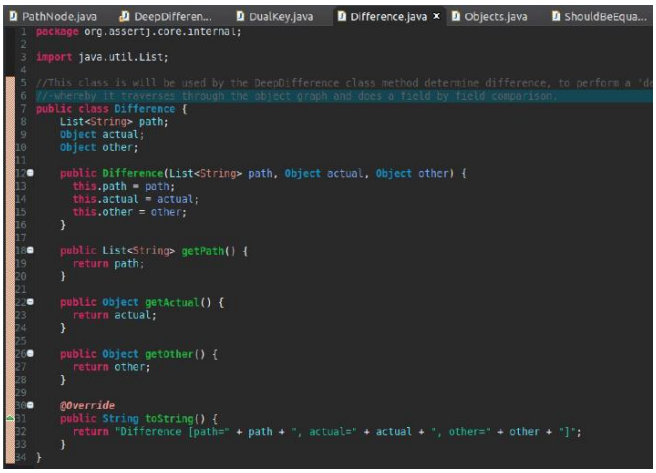


```
50
51  private final static class DualKey {
52
53      private final List<String> path;
54      private final Object key1;
55      private final Object key2;
56
57      private DualKey(List<String> path, Object key1, Object key2) {
58        this.path = path;
59        this.key1 = key1;
60        this.key2 = key2;
61      }
62
63      @Override
64      public boolean equals(Object other) {
65        if (!(other instanceof DualKey)) {
66          return false;
67        }
68
69        DualKey that = (DualKey) other;
70        return key1 == that.key1 && key2 == that.key2;
71      }
72
73      @Override
74      public int hashCode() {
75        int h1 = key1 != null ? key1.hashCode() : 0;
76        int h2 = key2 != null ? key2.hashCode() : 0;
77        return h1 + h2;
78      }
79
80      @Override
81      public String toString() {
82        return "DualKey [key1=" + key1 + ", key2=" + key2 + "]";
83      }
84
85      public List<String> getPath() {
86        return path;
87      }
88
89      public String getConcatenatedPath() {
90        return join(path).with(".");
91      }
92  }
93
94  public static class Difference {
95
96      List<String> path;
97      Object actual;
98      Object other;
99
100     public Difference(List<String> path, Object actual, Object other) {
```

Figure 19. Re-Engineering Attempt

In the picture above, the DeepDifference Class can be seen to have two external calls.

Then we created separate class files for those classes and modified the modifiers so that other AssertJ classes, especially the Objects class, could access the class's data members and call the necessary method to determine the differences. After that, we fixed any compile errors that arose as a result of the re-engineering project.



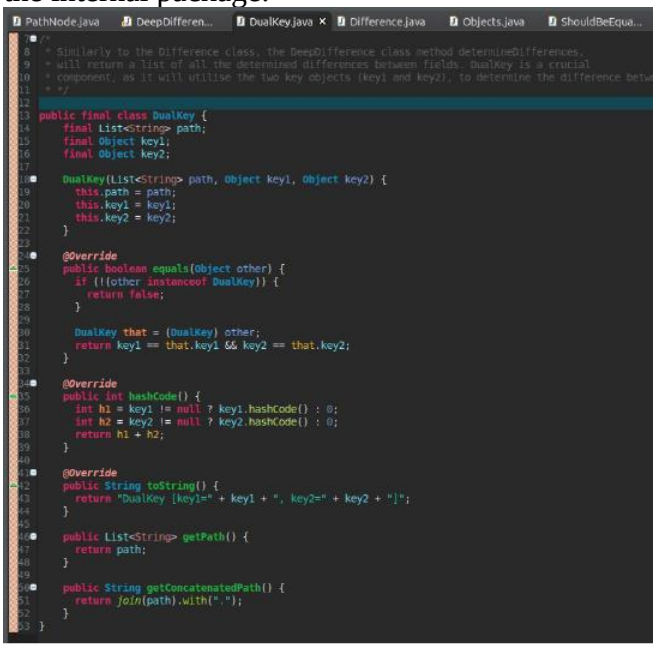
```
PathNode.java  DeepDifferen...  DualKey.java  Difference.java ×  Objects.java  ShouldBeEqua...
1  package org.assertj.core.internal;
2
3  import java.util.List;
4
5  //This class is will be used by the DeepDifference class method determine difference, to perform a 'de
6  //-whereby it traverses through the object graph and does a field by field comparison.
7  public class Difference {
8      List<String> path;
9      Object actual;
10     Object other;
11
12     public Difference(List<String> path, Object actual, Object other) {
13       this.path = path;
14       this.actual = actual;
15       this.other = other;
16     }
17
18     public List<String> getPath() {
19       return path;
20     }
21
22     public Object getActual() {
23       return actual;
24     }
25
26     public Object getOther() {
27       return other;
28     }
29
30     @Override
31     public String toString() {
32       return "Difference [path=" + path + ", actual=" + actual + ", other=" + other + "]";
33     }
34 }
```

Figure 20. Difference Class

Here is the Difference Class, with comments added to clarify the class's function and how it will be used.

Similarly, we explained how the data members of DualKey would communicate with the DeepDifference class in the same way.

This is a screenshot of the DualKey.java class from the internal package:



```
PathNode.java  DeepDifferen...  DualKey.java ×  Difference.java  Objects.java  ShouldBeEqua...
1  /*
5   * Similarly to the Difference class, the DeepDifference class method determineDifferences,
9   * will return a list of all the determined differences between fields. DualKey is a crucial
10  * component, as it will utilise the two key objects (key1 and key2), to determine the difference betw
11  * */
12
13 public final class DualKey {
14     final List<String> path;
15     final Object key1;
16     final Object key2;
17
18     DualKey(List<String> path, Object key1, Object key2) {
19       this.path = path;
20       this.key1 = key1;
21       this.key2 = key2;
22     }
23
24     @Override
25     public boolean equals(Object other) {
26       if (!(other instanceof DualKey)) {
27         return false;
28       }
29
30       DualKey that = (DualKey) other;
31       return key1 == that.key1 && key2 == that.key2;
32     }
33
34     @Override
35     public int hashCode() {
36       int h1 = key1 != null ? key1.hashCode() : 0;
37       int h2 = key2 != null ? key2.hashCode() : 0;
38       return h1 + h2;
39     }
40
41     @Override
42     public String toString() {
43       return "DualKey [key1=" + key1 + ", key2=" + key2 + "]";
44     }
45
46     public List<String> getPath() {
47       return path;
48     }
49
50     public String getConcatenatedPath() {
51       return join(path).with(".");
52     }
53 }
```

Figure 21. DualKey Class

We are able to break up the responsibilities of a God Class, DeepDifference.java, in this re-engineering mission. This was accomplished by removing internal classes and dividing them into smaller classes. A god class monopolizes power over an application by taking on so many duties. The application's evolution is complicated because virtually every transition has an effect on this class and multiple responsibilities. This is the problem with assertJ since there is another God Class named entity, and any changes made to the application's code structure can have negative consequences for the behavior of that class and the method.
In addition, the number of lines of code for DeepDifference was reduced from 547 to 474. We are able to effectively modify the lines of code as a result of the code refactoring. As a result, the re-engineering task's key goal has been met.

VI. Conclusion

The software can be complex and cumbersome with lots of components for which the source code may not be always available. It is often worth considering the behavioral model i.e., getting an overview using diagrams, etc., instead of working on the complicated source code.

The project has gained some productivity as a result of the re-engineering task that we completed. For example, by breaking up the DeepDifference class, we were able to adjust the project's behaviour and reduce the overall number of lines of code for DeepDifference. Not only did we gain a better understanding of the program and its behavior, but we were also able to spot possible system flaws that needed to be fixed and refactored.

VII. Future Work

In this project, we have used manual techniques to generate various metrics like LOC, weighted method count, etc. We have also used the manual approach for the visualization part to get a better understanding of the complexity level of the entire system. The expectation in the future is to achieve similar results using some type of automated metric calculation tool like CLOC. Also, some data visualization tools like PMD or Java Visualizer.

VIII. References

1. https://assertj.github.io/doc/
2. https://github.com/joel-costigliola/assertj-core/issues
3. https://reverseengineering.stackexchange.com/questions/3473/what-is-the-difference-between-static-disassembly-and-dynamic-disassembly
4. https://r4ds.had.co.nz/data-visualisation.html
5. https://link.springer.com/article/10.1007/s11219-020-09507-0
6. Cyclomatic Complexity. Available at: http://www.arisa.se/compendium/node96.html (Accessed: 20th December2018)
7. Rojas, J. and Walkinshaw, N. (2018). *Measuring Software Quality*. (Accessed: 15th January 2019).
8. Yegor (2014), Java Method Logging with AspectJ.https://www.yegor256.com/2014/06/01/aop-aspectj-java-method-logging.html (Accessed: 8 th December 2018).
9. Yegor (2014), Java Method Logging with AspectJ. Available at: https://www.yegor256.com/2014/06/01/aop-aspectj-java-method-logging.html (Accessed: 8 th December 2018).
10. Walkinshaw, N. (2013). *Reverse-Engineering Software Behavior*. Leicester: Elsevier IncU, pp.14, 23, 31, 34, 44, 46, 51. (Accessed: 19th January 2019).
11. D. Marks (2008), Lines of Code and Unintended Consequences. Available at: https://www.javaworld.com/article/2072312/lines-of-code-and-unintended-consequences.html (Accessed: 20th January 2019).
12. https://pmd.github.io/