Re-engineering AssertJ

Initial Exploration

For the initial part of this re-engineering task, the aim was to gain an overview of AssetJ, so I gathered information about the software's function, it's key components and architectural features. I did this by adopting two of the re-engineering methodologies, which are: "Skim Documentation" and "Read code in an hour".

Skimming Documentation

Using this approach, I discovered that AssertJ is a Java library that provides a fluent interface for writing assertions. Its goal is to improve the readability of test code and make maintenance of tests easier. AssertJ's ambition is to provide rich and intuitive set of strongly-typed assertions for unit-testing. For e.g. if you are checking the value of a Map, you can use Map-specific assertions.

The creator of this Java library is Joel Costigliola. The development team offer open source contribution and provide support to anyone who is willing to contribute.

The creator of the library has made a website for AssertJ using GitHub pages. The website has a nice layout and for every additional information on the library there are links on the readme page, which directs you straight to the website. The website provides information on the different components, a QuickStart tutorial and the latest news on new version releases. At the time of this report the latest AssertJ core version is 3.11.1.

AssertJ consists of six different components, with each of them having their source code kept a different git repository under the creator's name. The five different components are: **Guvana**, **Joda Time**, **Neo4J**, **DB** and **Swing** module.

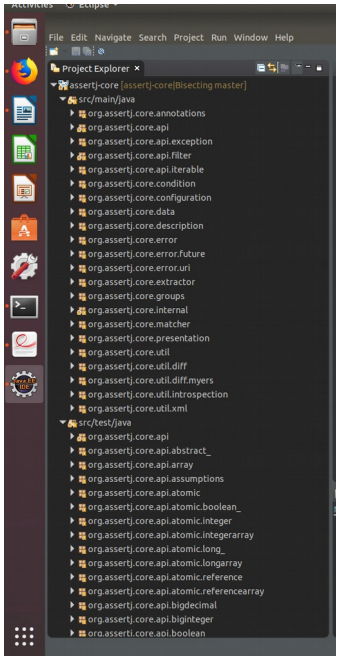
In addition, there is a feature called Assertion Generator, which allows developers to generate assertions code based their own classes. This allows developers to write assertions specific to their domain model vocabulary.

Critic of Documentation

The documentation is overall alright as it provides enough information for one to get an overview of the system. However, if you look at the documentation for the six different components, none of them have adequate documentation. As their main components there should be enough description on the functionality of the components rather than a few sentences. Despite this, each of them does have a link to the website, where you can find all the information about each individual component.

Analysis of Directory and Code Structure

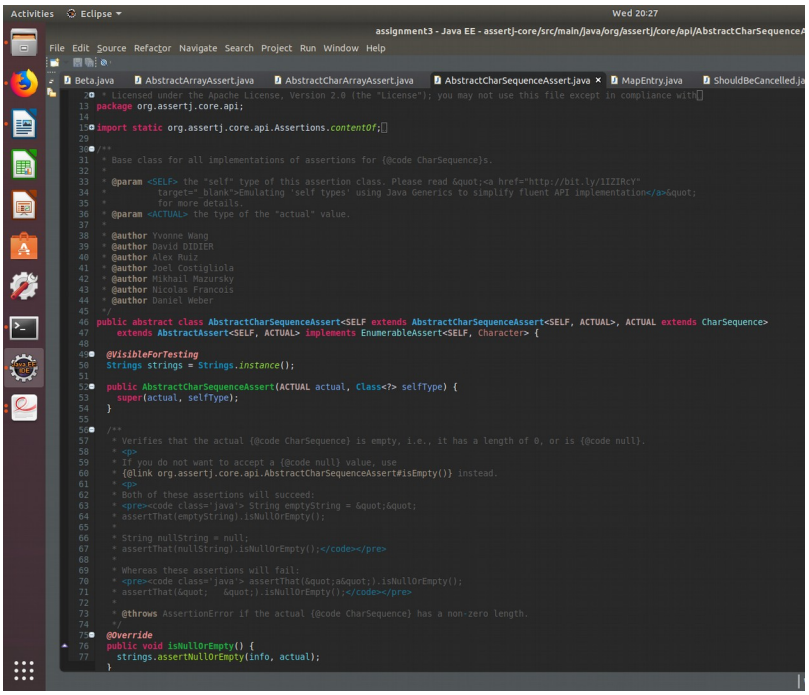
The directory structure of AssertJ is organised in such a way that the readme files and other form of documentation (such as: CODE_OF_CONDUCT.md and ISSUE_TEMPLATE.md etc.) are kept separate from the source folder which contains all the logic. In addition, within the logic the source folder is structured such that the main Java implementation is separated from the test classes in the test folder. However, they the naming conventions and package names are the same. Regarding code readability, it makes it easier to identify the class and its corresponding test class (refer to **figure 1**).



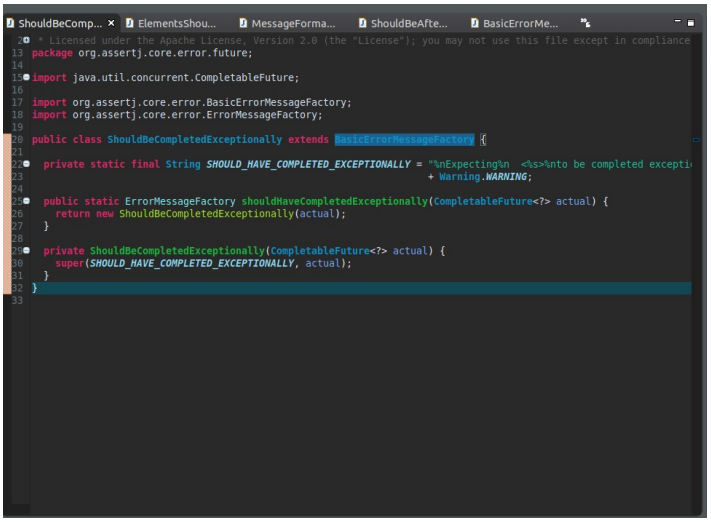
[figure 1, code structrue – project consists of 12 packages with each having a corresponding test package]

Code Maintainability

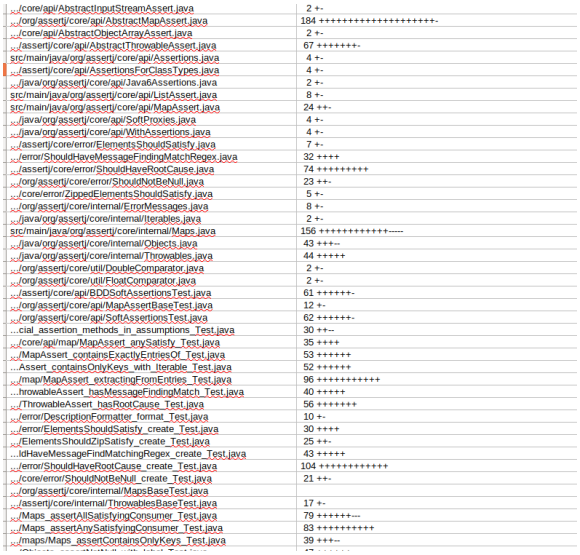
After reviewing the source code of assertj-core, the structure and the readability of the code is clear and understandable. I observed that majority of classes contain detailed comments about the functionality of the methods (for an example refer to **figure 2**). However, some classes contain obsolete comments and it makes it difficult to understand the functionality of those classes (refer to **figure 3**). Likewise, according to OORP (Object Oriented Re-engineering Patterns), this increases the need of human dependency because obsolete to less documentation of code makes the system more difficult to maintain. In addition, it does not contain a lot of “smelly code” (OORP, p.6) such as long methods and data classes which suggests that the software is well designed. However, it is important to detect smelly code, and reduce it, that is a technical task that I will address later on when I conduct my static and dynamic analysis.



[figure 2 the class AbastractCarArrayAssert.java has been sufficiently documented]



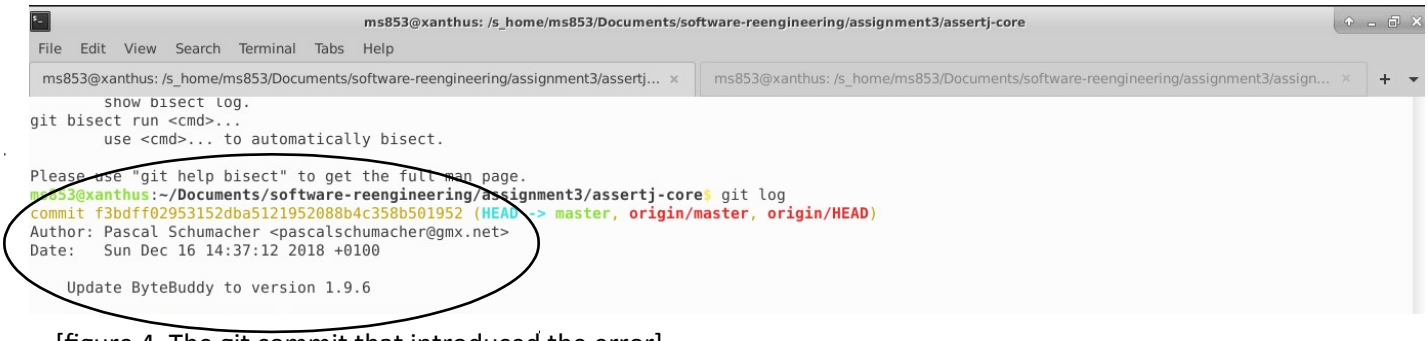
[figure 3 – an example of a class that does not have any comments]



The image above show all of the recent commit history and all the classes that have been changed.

Analysis of Commits and Issues

For this part of the exploration, to examine the codebase of the AssertJ project, I cloned the main git repository of assert-core. I ran the **git bisect** command on the master repository, which performs a binary search on the repository and identifies the commit that introduced a bug. After running the command, I ran the git log command in order to check the author and the date of that commit that caused the bug. It appears that the bug was a result of a commit made to the master repository by Pascal as shown in **figure 4**.



[figure 4, The git commit that introduced the error]

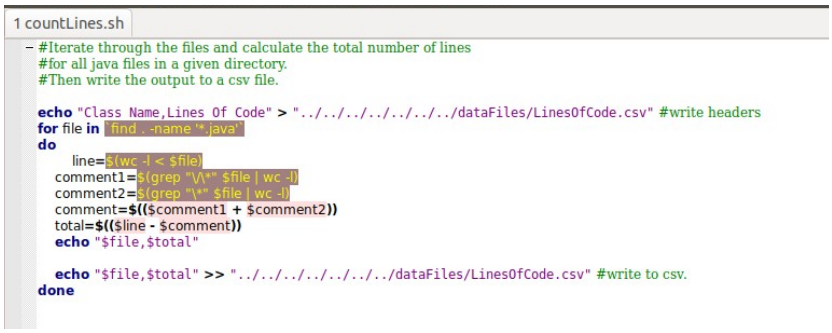
Furthermore, after reviewing the issues for assert-core on GitHub, one key observation I made was on the number of closed issues. It appears that there are more closed issues than open issues. The total number of open issues are 61 pales in comparison to the 626 closed issues.

Metrics and Analysis (Static Analysis)

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

Static Analysis was used to conduct.

Firstly, I conducted some static analysis whereby I calculated the LOC (Lines of Code) for each class in the assertj-core directory. I achieved this by writing a shell script that counts the number of lines for all java files in a given directory (refer to **figure 5**). Then once the calculation is complete, the result is written to a csv file . I then stored the csv file in the dataFiles directory.



[figure 5 – here is picture of the shell script that counts the number of lines of code for a every java class].

Regarding the second half of the statistic-analysis I calculated the weighted method count for each method along with the number of nodes in the CFG, and the Cyclomatic Complexity for each method.

Firstly, I started by modifying the ClassMetrics.java file, such that it processed the direcory that contains all class files relevant to the AssertJ's functionality. Furthermore, I then proceeded to refactor the class paths in such a way that the input stream instance can process the class path as a valid file path. In order for the classes to be processed, I generated, I called the **processDirectory()** method from the ClassDiagramSolution class. Then I stored the contents of that list into a list of type class. Then I wrote a for-loop that iterates over the list of classes. So for every class in the list, I calculate the **Cyclomatic Complexity (C.C.)**, the number of nodes and weighted method count (wmc) of that particular class.

The first metrics I did was class metrics where for each class the C.C. and number of nodes were calculated and I did this for every single method in a given class. I generated this metric in hopes that it will provide me with a general overview of the AssertJ software and which methods are complex according to the C.C value. Further, this data was writen to a csv file, I then selected the top 10 based on the number of nodes (as shown in **figure 6 which is the classMetrics.csv file**). Based on this metric I conclude that AssertJ consists of **6444** methods.

After some research, I decided that the weighted metric that I will use to calculate the **wmc (weighted method count)** is the cyclomatic complexity. The reason for this is because the cyclomatic complexity gives an insight into the control structure and complexity of the software. Moreover, if the C.C. value is high, then the class is classed to be very complex and as a systems engineer, I can suggest that it requires indepth testing. So I calculated the wmc on the basis that it will be the total sum of the C.C. value, for every method, per class. Then I stored the aggregated values of the wmc to its corresponding class in a hashmap. The reason why I used this data-structure is because hashmaps preserve uniqueness by storing the values in key-value pairs, so I saw this as an efficient way of storing that information. Afterwards, I printed the values into a csv file. In **figure 7**, is a picture of the csv file that displays the class name and the total weighted method count. Based on this metric I can conclude that AssertJ consists of **596 classes**.

Lastly, the last metrics I produced was that of the method tightness. I did this by creating an object of the program dependecy graph class. Then based on the threshold whether the number of nodes (statements in a program), exceeds 150. The reason for this threshold is because I wanted to narrow down to classes that had a lot of statements, which will give me a better outlook on the system. Then I called the method **computeTightness()**, to calculate the thightness for each method node. The compute tightness method is a slice-based metric, that calculates the proportion of nodes in a control flow graph, that occur in every possible slice of the control flow graph. As a result, all of the class methods, have a tightness which is less than **0.01 with the highest tightness value being 0.0065**. This suggests that the methods in this system have a high cohesion. Cohesion measures to what extend which data and functions within a class/module are "related" (Rojas and Walkinshaw, 2018). This shows that the the methods, that have tightness result of **0.005** or greater, cannot be tampered with, because if major changes are introduced to those method, it can change the behaviour and functionaity of the software. The output can be shown in **figure 8** with the name of the csv file being methodCohesiveness.csv.

Furthermore, I performed the same static analysis on the test-classes that corresponded to the classes in the assertj-core project. I did this to see if there will be any similarities between the outcome of the metrics, of the classes and their test-classes. I named the files weightedMethodCount2.csv and classMetrics2.csv respectively.

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

org/assertj/core/internal/Strings.assertContainsSequence(Lorg/assertj/core/api/AssertionInfo;Ljava/lang/CharSequence;[Ljava/lang/CharSequence;)V	192	9

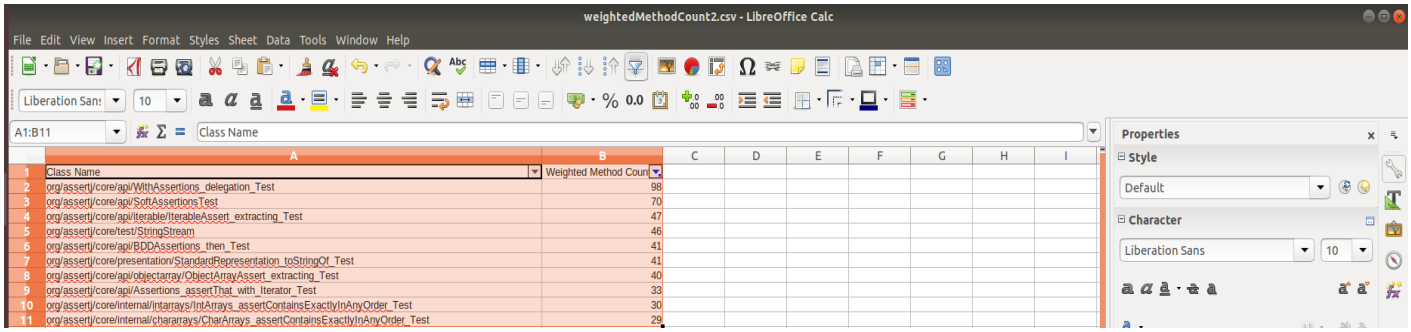
[figure 6 – Top 10 results from the classMetrics.csv, sorted based on the number of nodes.]

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 7 – Table showing the top 10 classes with the highest weighted method cout]

	A	B
1	Method Name	Method Tightness
2	org/assertj/core/util/introspection/FieldUtils.getField	0.006535947712418
3	org/assertj/core/util/diff/DiffUtils.parseUnifiedDiff	0.005813953488372
4	org/assertj/core/util/diff/DiffUtils.generateUnifiedDiff	0.005780346820809
5	org/assertj/core/util/diff/DiffUtils.processDeltas	0.005524861878453
6	org/assertj/core/util/diff/myers/MyersDiff.buildPath	0.005235602094241
7	org/assertj/core/util/diff/myers/MyersDiff.buildRevision	0.005208333333333
8	org/assertj/core/util/DateUtil.formatTimeDifference	0.005154639175258
9	org/assertj/core/presentation/StandardRepresentation.toStringOf	0.004761904761905
10	org/assertj/core/presentation/StandardRepresentation.format	0.004672897196262
11	org/assertj/core/internal/DeepDifference.determineDifferences	0.003533568904594
17		
18		

[Figure 8 – Top 10 table for class methods with a really high cohesion]



[Figure 9 – Top 10 results of the Weighted Method Count for the test-classes]

Lastly, for the final part of the static analysis, I generated a class diagram with no scaling. I did this so that I can get an overview of the different inheritance and associations and relationships between the classes in the AssertJ software system. To generate the Class Diagram, I made a class called **ClassDiagramTest.java**, where I point to the directory target/classes directory to load and write the dot file for the class diagram.

***Please refer to the ClassDiagramStatic.pdf to view the class diagram. (Note that I have deleted the old Class Diagram dot file because I no longer have any use for it because I have classed it as a redundant file).**

Why did I use behavioural models instead of there is source code?

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 I do not care about; as I am only interested in the behavior of the system. In that case, behavioural models such as: Class Diagrams, Metric tables and other forms of visualizations are needed to convey such infromation. Furthermore, high level interaction of software behavior arises from extensive interaction between packages or modules (Walkinshaw, 2013 p.14).

Metrics and Analysis (Dynamic Analysis)

For the dynamic analysis stage, my main concern was the analyses of the program while it executes.

Based on the results of the metrics I obtained from conducting the static analysis, I decided that based on the **top 10 classes** with high wmc, I selected those classes to be analyzed further.

Based on the weighted method count I narrowed my choice of classes to the **top 5**. The reason why I did this is to reduce the scope of focus to particular classes that a really high wmc.

The top 10 classes with the highest weighted method count (wmc) in ascending order :

***Note that the five class names I have highlighted here are the classes which I have produced traces.logs for their each of their test-class counterparts.**

- **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**

The following test-classes I 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

Aspect Oriented Programming (AspectJ)

AOP (Aspect Oriented Programming) is used to complement Object Oriented Programming it allows adding executable blocks around source code, without having to modify it (Yegor, 2014). The technical challenge which I had to address in this phase of the analysis, was to asses the behaviour of the test-classes (which I have idenified as potential candidates for re-engineering) and the software system as a whole. I applied aspect oriented programming techniques by creating the **Trace.java** file which I used to log every class method call invoked by the 5 test-classes I selected.

```
46      pointcut traceMethods() : (execution(* *(..))&& !cflow(within(Trace)) && !within(org.junit..*));
47
48
49
50      before(): traceMethods(){
51              Signature sig = thisJoinPointStaticPart.getSignature();
52              String sourceName = thisJoinPointStaticPart.getSourceLocation().getWithinType().getCanonicalName();
53
54              //Get the parameters of the methods.
55              String[] parameters = ((CodeSignature) thisJoinPointStaticPart.getSignature()).getParameterNames();
56              StackTraceElement[] stackDepth = Thread.currentThread().getStackTrace(); //array for getting the stack dep
57              int realStackDepth = stackDepth.length -1;
58
59              StringBuilder paramSb = new StringBuilder();
60              String stack_depth = Integer.toString(stackDepth.length);
61
62              for(int i = 0; i < parameters.length; i++) {
63                  paramSb.append(parameters[i] + " ");
64
65              }
66
67
68
69              Logger.getLogger("Tracing").log(
70                      Level.INFO,
71                      //sig.getDeclaringType().getSimpleName().toString() + " " +
72                      sourceName + " " + sig.getName()
73                      + " " + paramSb.toString().replaceAll(","," ") //replace all commas
74                      + " " + realStackDepth
75              );
76      }
77  }
```

Here you can the code statement called the pointcut which I used to intercept every execution of the class and log it into the trace file.

Here I record the method stack-depth, along with the method parameters and the class name.

Afterwards, I wrote a class called **DynamicAnalysis.java,** where I read the contents of all the trace files and I store those contents inside a hashmap. The reason why I decided to use this datastructure is because it preserves uniqueness which is appropriate to the task. The aim of the task was to aggregate the total occurrence of all the classes across the 5 trace files.

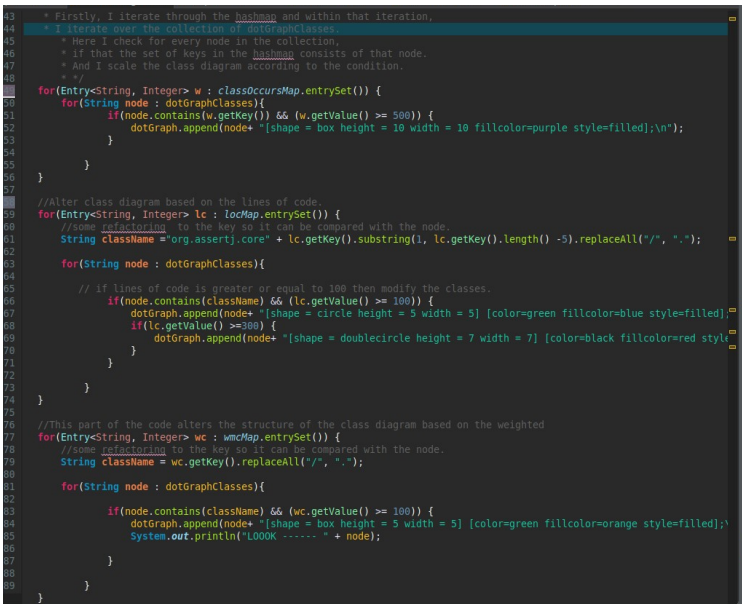
Then I used the information I obtained from that part to dynamically scale of my Class Diagram. The Class Diagram I produced was colour coded and the size was scaled relatively larger to the other classes. I did this to inidicate that the those classes may have a relatively high frequency of calls in comparison to the other classes. The other metric I used was the LOC (Lines of Code), to determine the particular what classes have a large lines of code. In relation to the metrics that I have chosen, I hereby claim that the number of lines of code was a useful metric to use, as it provided a better insight to what candidates that need to be re-engineered. Because by identifying classes with huge lines of code, I can use that information to reduce the lines of code as re-engineering task and therefore make the code more cleaner, efficient and maintainable, which are crucial requirements a legacy software like AssertJ.

As I have mentioned in the previous part of the report, I wrote the **ClassDiagramTest.java** class to process the target classes and generate the Class Diagram. In contrast to my old class diagram, this one encompases the dynamic analysis and it is scaled based on the following metrices: total number of occurrence, LOC and most importantly weighted method count.

Below is the code snippet I used to generate the class diagram:

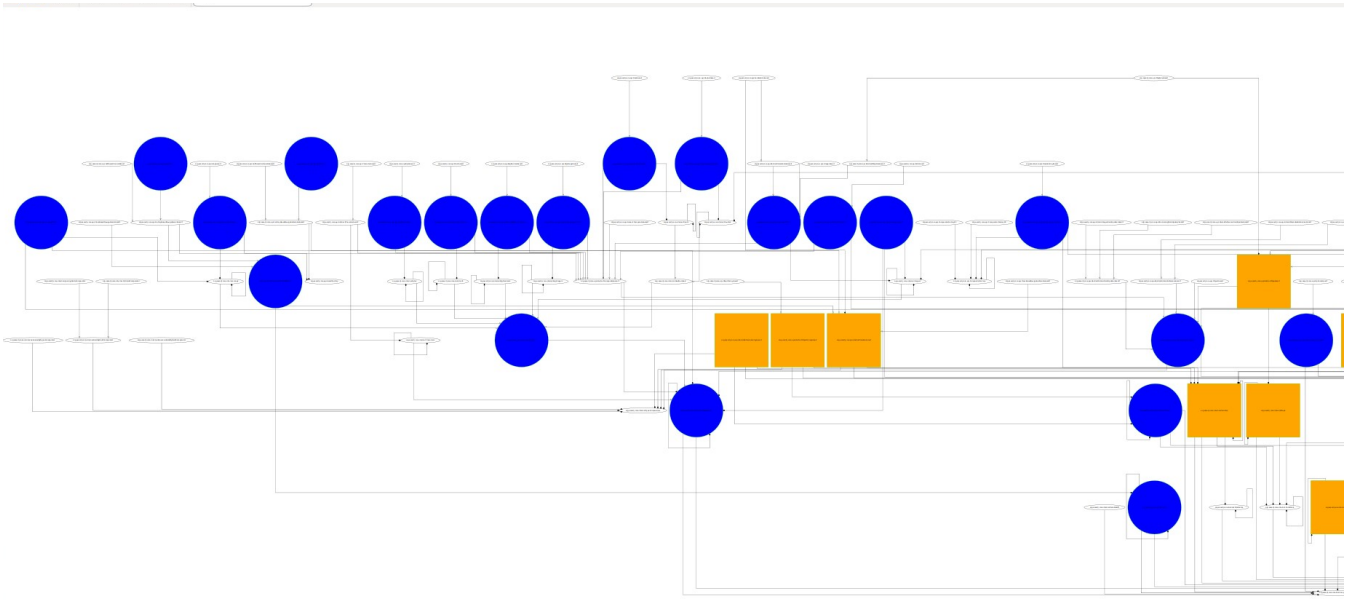
```
54  public static void main(String[] a) throws IOException {
55
56      String fileName1, fileName2, fileName3;
57      fileName1 = "/s home/ms853/Documents/software-reengineering/assignment3/assignment3-ms853/dataFiles/LinesOfCode.cs
58      fileName2 = "/s home/ms853/Documents/software-reengineering/assignment3/assignment3-ms853/dataFiles/classOccurren
59      fileName3 = "/s home/ms853/Documents/software-reengineering/assignment3/assignment3-ms853/dataFiles/weightedMetho
60      BufferedReader reader;
61      BufferedReader reader2 = null;
62      BufferedReader reader3 = null;
63      //BufferedReader reader3 = null;
64      reader = new BufferedReader(new FileReader(fileName1));
65      String[] output;
66      String line = "";
67      String headerLine = "";
68      headerLine = reader.readLine(); //to skip the first header csv file.
69      System.out.println(headerLine);
70      while ((line = reader.readLine()) != null) {
71          output = line.split(",");
72          locMap.put(output[0], Integer.parseInt(output[1]));
73
74      }
75      reader.close();
76
77      reader2 = new BufferedReader(new FileReader(fileName2));
78      String[] output1;
79      String line1 = "";
80      String headerLine1 = "";
81      headerLine1 = reader2.readLine(); //to skip the first header csv file.
82      System.out.println(headerLine1);
83      while ((line1 = reader2.readLine()) != null) {
84          output1 = line1.split(",");
85          classOccursMap.put(output1[0], Integer.parseInt(output1[1].substring(1)));
86
87      }
88      reader2.close();
89
90      reader3 = new BufferedReader(new FileReader(fileName3));
91      String[] output2;
92      String line2 = "";
93      String headerLine2 = "";
94      headerLine2 = reader3.readLine(); //to skip the first header csv file.
95      System.out.println(headerLine2);
96      while ((line2 = reader3.readLine()) != null) {
97          output2 = line2.split(",");
98          wmcMap.put(output2[0], Integer.parseInt(output2[1]));
99
100     }
```

This code snippet is in the main method which where I read three different CSV files for my matrices, which populate my hashmap which I later on used to scale the diagram. Also it's in the same main method where I invoke the **writeDot()** method, which writes out the class diagram as a **dot file**.



```
    * Firstly, I iterate through the hashmap and within that iteration,
    * I iterate over the collection of dotGraphClasses.
        * Here I check for every node in the collection,
        * if that the set of keys in the hashmap consists of that node.
        * And I scale the class diagram according to the condition.
        * */
    for(Entry<String, Integer> w : classOccursMap.entrySet()) {
        for(String node : dotGraphClasses){
                if(node.contains(w.getKey()) && (w.getValue() >= 500)) {
                    dotGraph.append(node+ "[shape = box height = 10 width = 10 fillcolor=purple style=filled];\n");
                }

            }
    }

    //Alter class diagram based on the lines of code.
    for(Entry<String, Integer> lc : locMap.entrySet()) {
        //some refactoring  to the key so it can be compared with the node.
        String className ="org.assertj.core" + lc.getKey().substring(1, lc.getKey().length() -5).replaceAll("/", ".");

        for(String node : dotGraphClasses){

            // if lines of code is greater or equal to 100 then modify the classes.
                if(node.contains(className) && (lc.getValue() >= 100)) {
                    dotGraph.append(node+ "[shape = circle height = 5 width = 5] [color=green fillcolor=blue style=filled];
                    if(lc.getValue() >=300) {
                        dotGraph.append(node+ "[shape = doublecircle height = 7 width = 7] [color=black fillcolor=red style
                    }
                }

            }
    }

    //This part of the code alters the structure of the class diagram based on the weighted
    for(Entry<String, Integer> wc : wmcMap.entrySet()) {
        //some refactoring to the key so it can be compared with the node.
        String className = wc.getKey().replaceAll("/", ".");

        for(String node : dotGraphClasses){

                if(node.contains(className) && (wc.getValue() >= 100)) {
                    dotGraph.append(node+ "[shape = box height = 5 width = 5] [color=green fillcolor=orange style=filled];
                    System.out.println("LOOOK ------ " + node);

                }

            }
    }
```

[Figure 10 – code implementation for the scaling of the class diagram based on the metrics]



[Figure 11 – Shows a high density of interactions between different classes. The key observation here is that I have scaled the class diagram such that it identifies classes LOC values greater than or equal to 100.]

I did this to draw some correlation between the lines of code and the complexity of a given class. Based on this analysis, there was one noticable classe that appeared in the wmc metric for the test -classes, that class is:

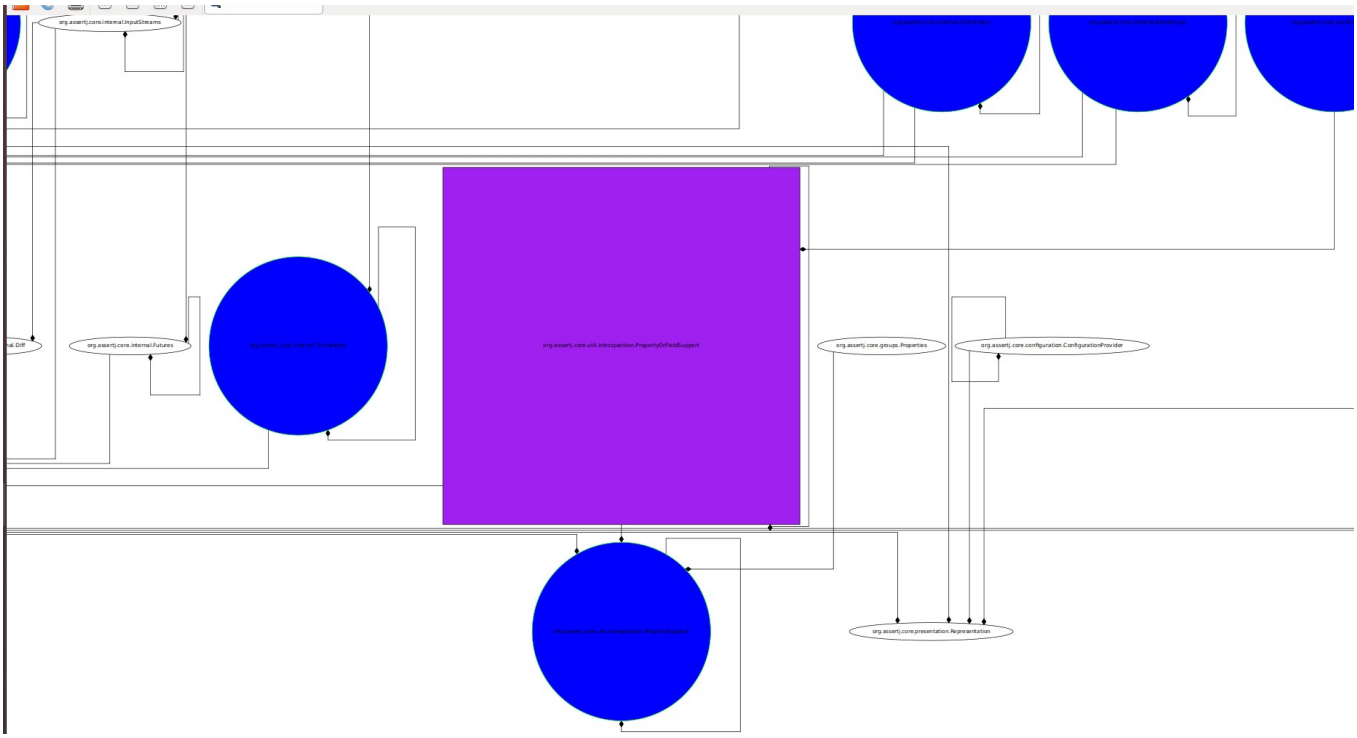
- org.assertj.core.IterableAssert (which is the class counterpart to the IterableAssert_extracting_Test class).

Here are classes with over 500 lines of code **(Note that the 4 classes which I have highlighted in yellow are classes that have a high wmc value and has a high LOC value of just over half 1000)**:

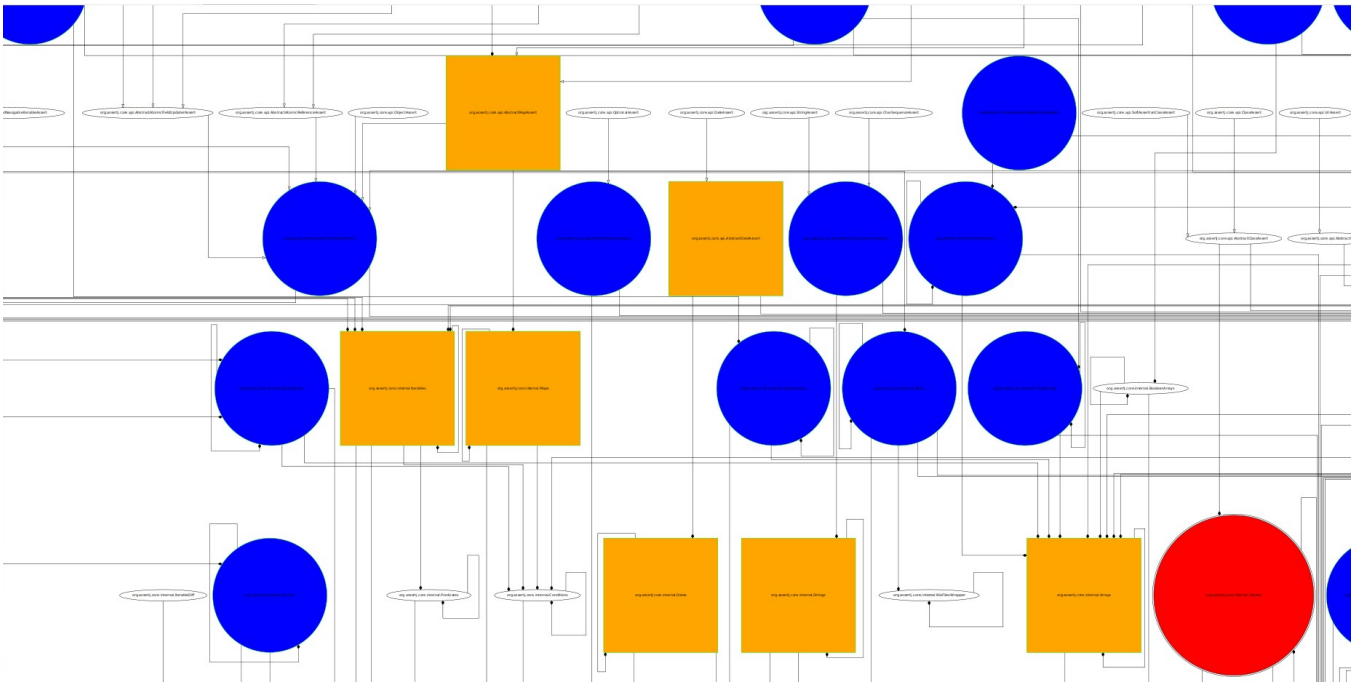
- org.assertj.core.api.AtomicReferenceArrayAssert
- org.assertj.core.internal.Iterables
- org.assertj.core.api.AbstractObjectArrayAssert
- org.assertj.core.internal.DeepDifference (A parent class that I will possibly look into to reduce the lines of code).
- org.assertj.core.internal.DeepDifference$DualKey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

This suggests that these classes are not only complex in nature but their complexity has some relation to the length of code and statements in the programs. The noticable classes that appeared in the wmc metrics are the same classes that have appeared to have a large number of LOC. This can possible mean as a re-engineering task, reducing the line of code in particular methods of those classes, can certainly decrease the complexity of those classes. More importantly this will also reduce the amount of redundant code in the software system.

On the other hand, classes that had a high occurrence (in other words, high number of calls by the test classes used to trace them), were modified such that the they appear in purple boxes as seen in the class diagram in figure 11.



[figure 11 – Boxes in purple indicate classes with high occurrences. In particular, there is the **PropertyOrFieldSupport** class, which has a large number of association in particular to the **PropertySupport** class]



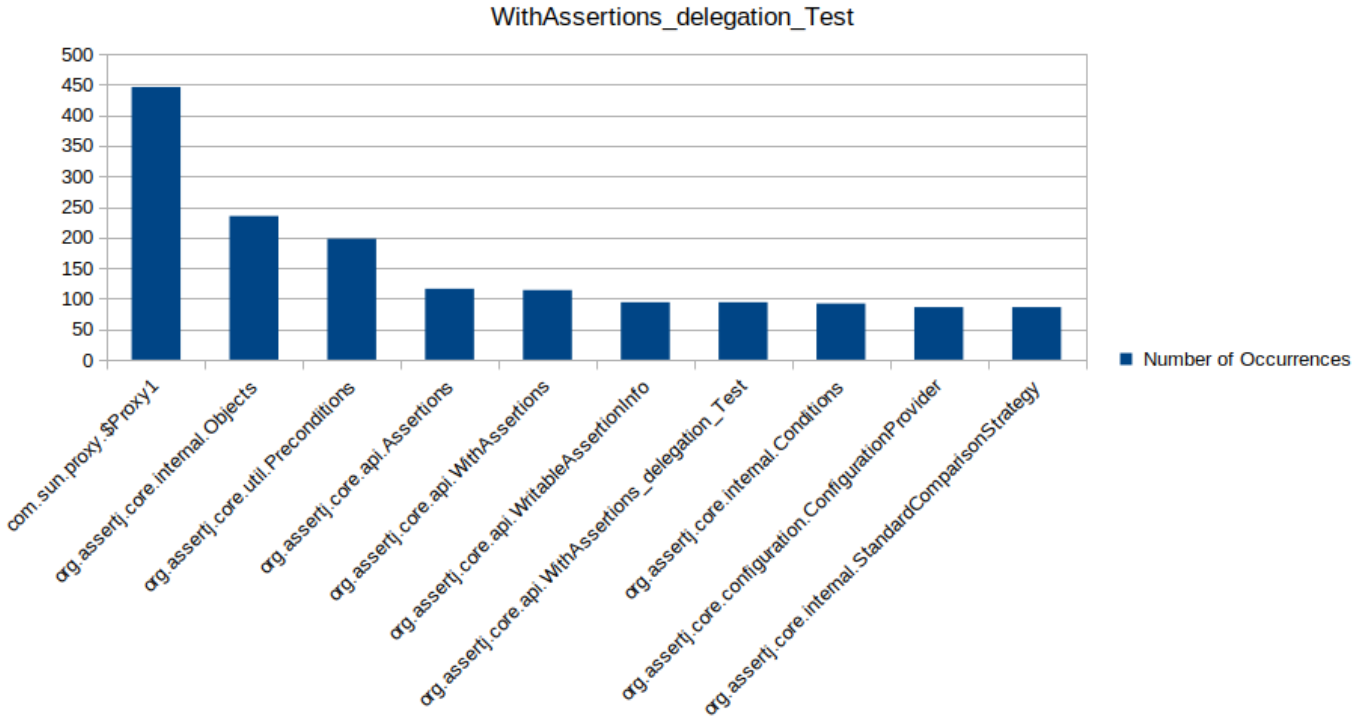
[Figure 12 – This part of the class diagram shows all the complex classes colour coded in orange. The threshold set for the weighted method count are for classes with values exceeding **100**. Once again, the classes that appear here where there is a high consentration of inheritance and associations between different classes are: Iterables, Arrays and Strings. Likewise, these classes are in the 10 classes that I identified as having the largest weighted method count values in the entire AssertJ project. However, since they have a high associations and multiple inheritances I need to take heed if I attempt to re-engineer any of those classes, as the behaviour of AssertJ can change drastically.]

In contrast, to the other classes like - Assertions, StandardRepresentations, Java6Assertions and DeepDifference, wh o have much higher complexity values are located in places where there is a low density of inheritance. This means modifying will not have a major impact on the behaviour of the other classes. However, by reducing the complexity of these classes will increase the maintainability of the software.

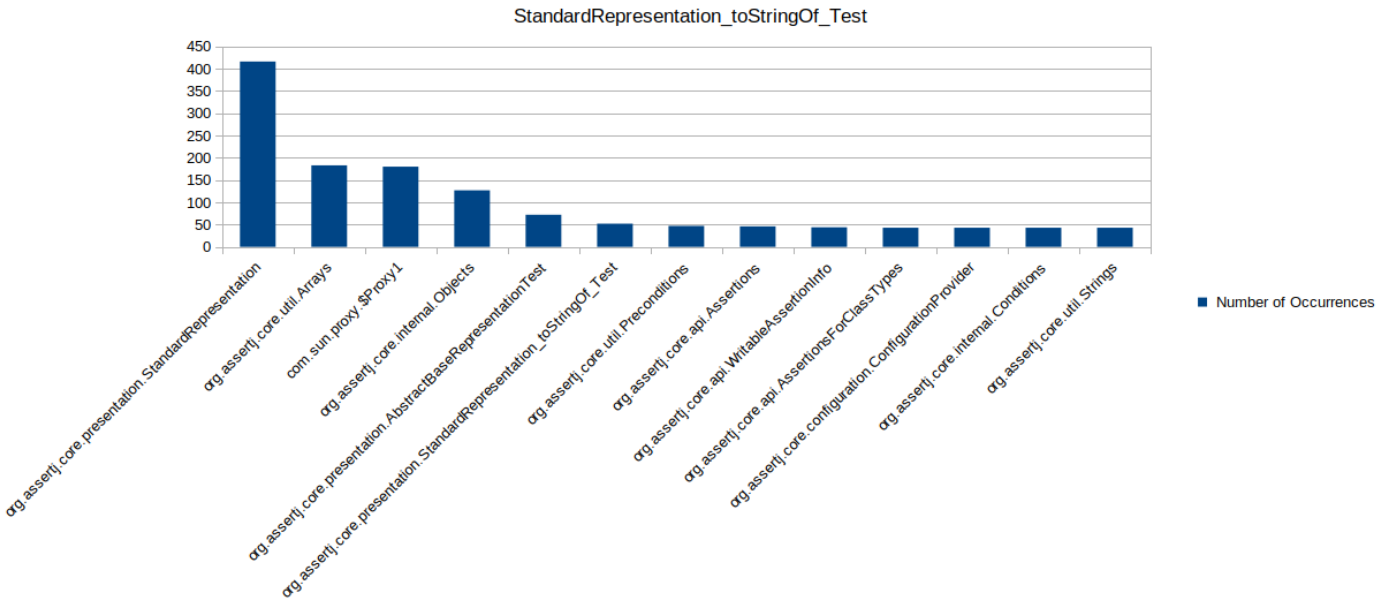
The Big Picture

Based on the static and dynamic analysis I produced some visualizations such as: time-series line graphs and bar charts and a dynamic class diagram to highlighting the results I have obtained about the behaviour of the AssertJ-core system.

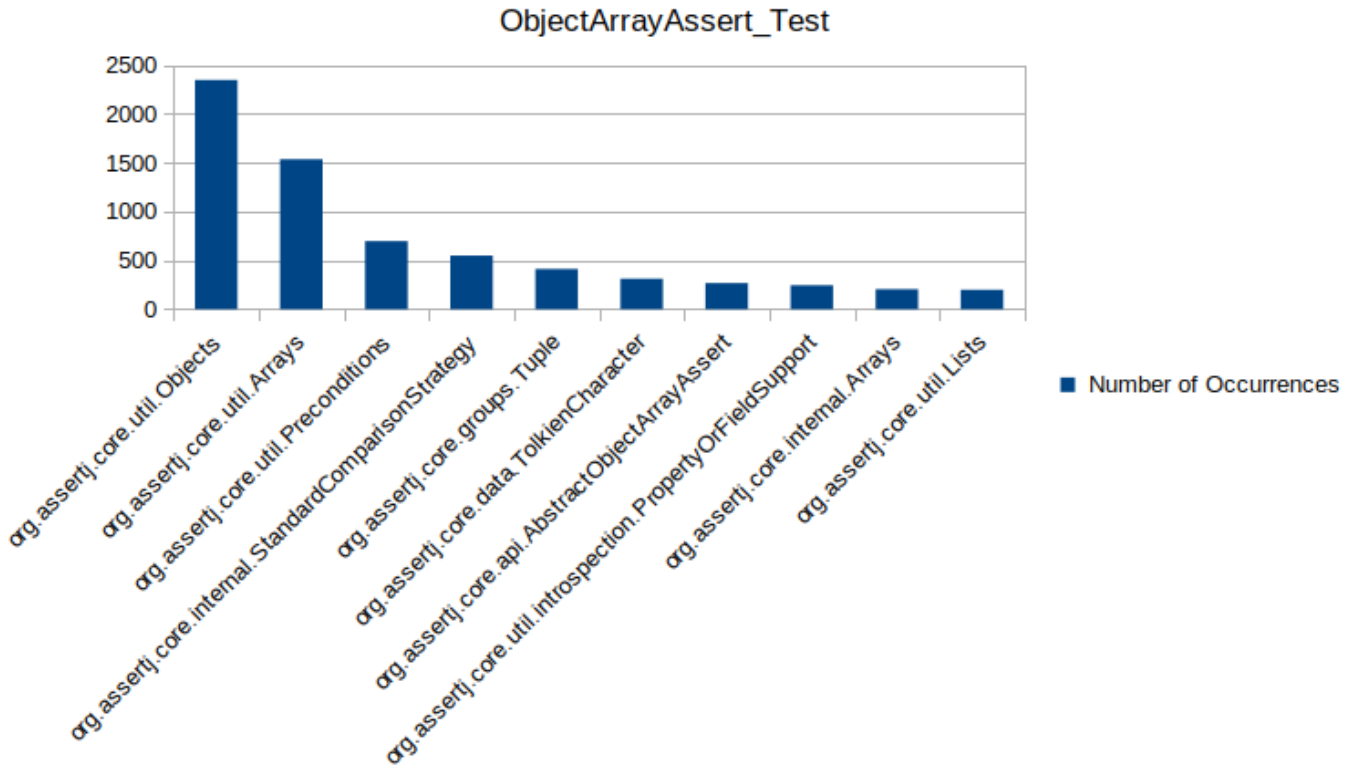
To generate visualizations, I started by reading the data from the 5 trace-files which I used as part of the dynamic analysis. The 5 trace files, which are files listed up which generated based on the test classes, which are counterparts of the re-engineering I wrote a new class called **ParseTraceToCSV.java,**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



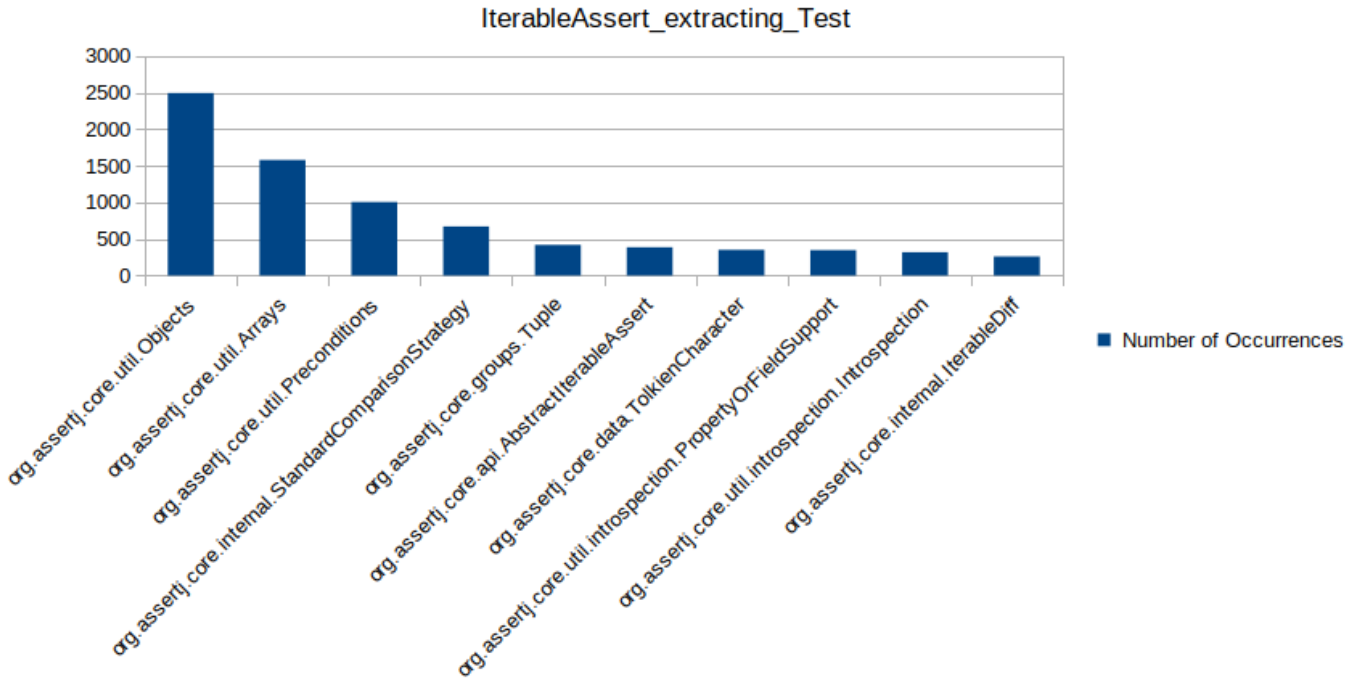
[Figure 13 – This is a bar chart shows the 10 classes that have been frequently executed by this test file. Here the class that are executed the most: Objects, Preconditions and Assertions and WithAssertions. Assertions in this case the second most executed class yet I know that based on my previous analysis that it has a high wmc.]



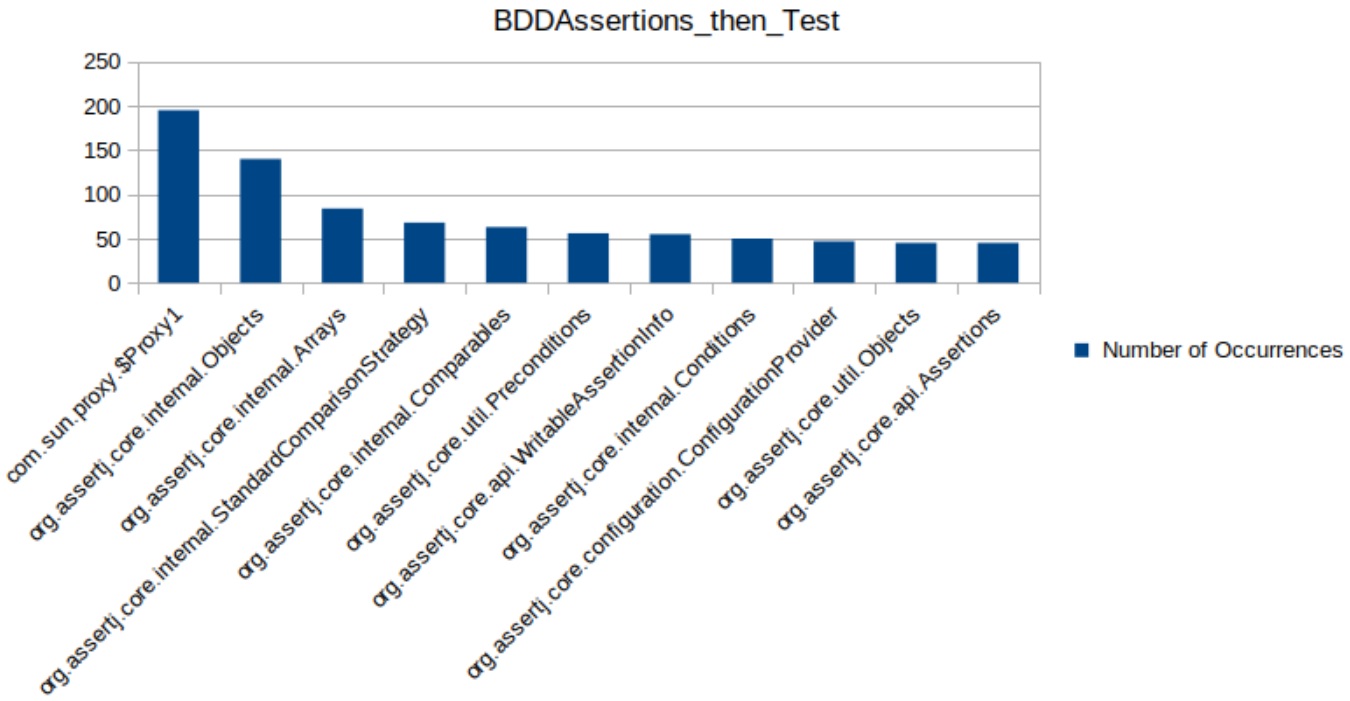
[figure 14 - The highest class that is executed by this test-class is StandardRepresentation (over 400 times). Likewise, Arrays class which is the second highest class executed also has a has a wmc value greater or equal to a 100.]



[figure 15 – similarly to the other test files, this ObjectArrayAssert_Test class frequently executes Object.]



[figure 16 – This test class as you can see invokes the **AbstractIterableAssert** which based on my metrics in the static analysis to have a wmc of 250 which is the highest.]



[figure – 17 – This test class similar to the others invokes the Objects calls frequently (145 times) which is considerably less than the other test-classes that tend to execute the Object class more. Additionally, it executes Arrays class in the iterables package. That class is considered based on my static analysis to be very complex and is a potential candidate for code refactoring.]

File Comparison and Code Duplication

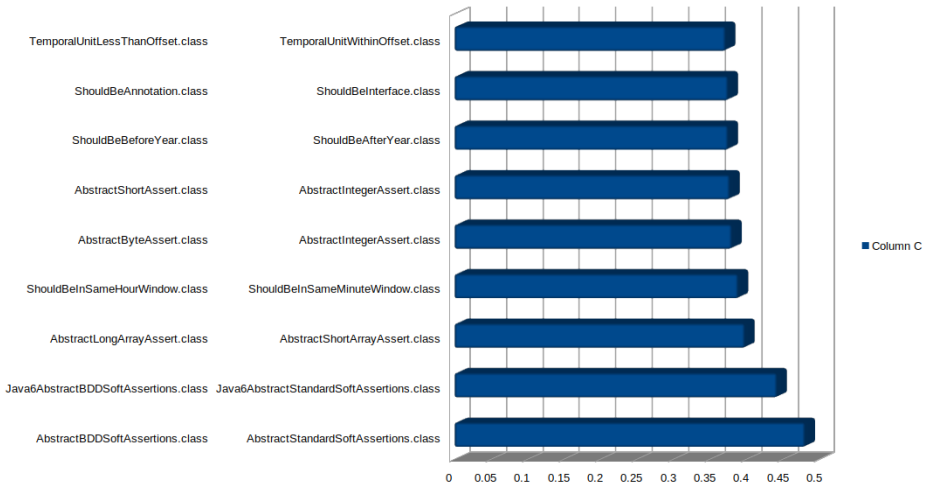
```
public double[][]fileComparison(boolean proportional){
    double[][] fileCompare = new double[files.size()][files.size()];
    for(int i = 0; i<files.size(); i++){
        for(int j = i+1; j<files.size(); j++){
            double score;
            if(i == j)
                score = 0;
            else{
                FileComparator fc = new FileComparator(files.get(i),files.get(j));
                score = fc.coarseCompare(proportional);
            }
            fileCompare[i][j] = score;
            fileCompare[j][i] = score;
        }
    }
    return fileCompare;
}

public List<File> getFiles(){
    return files;
}

}
```

This piece of code is that of the method fileComparison found in the duplicate detector class. The method is an algorithm that returns a 2-dimensional array. The method will iterate through the list of all the files in a given direcory and compare one file against the other. Below is the implementation of the class FileComparitor class.

```
try {
    BufferedReader br = new BufferedReader(new FileReader(file));
    String line = br.readLine();
    while(line != null) {
        line.trim();
        fromFile.add(line);
        line = br.readLine();
    }
    br.close();
} catch (FileNotFoundException e) {
    e.printStackTrace();
} catch (IOException e) {
    e.printStackTrace();
}
}

public boolean[][] detailedCompare(){
    List<String> fromFile = new ArrayList<String>();
    List<String> toFile = new ArrayList<String>();
    extractStrings(fromFile, from);
    extractStrings(toFile, to);
    boolean[][] matrix = new boolean[fromFile.size()][toFile.size()];
    for(int i = 0; i< fromFile.size(); i++){
        for(int j = 0; j<toFile.size(); j++){
            boolean match = fromFile.get(i).equals(toFile.get(j));
            if(fromFile.get(i).length()==0 || fromFile.get(i).equals("{")||fromFile.get(i).equals("}"))
                match = false;
            matrix[i][j]=match;
        }
    }
    return matrix;
}
}
```

```
public static void printRelations(double[][] scores, File target, List<File> files) throws IOException {
    FileWriter fw = new FileWriter(target);
    CSVPrinter csvPrinter = new CSVPrinter(fw, CSVFormat.EXCEL);
    List<String> record = new ArrayList<String>();
    record.add("From");
    record.add("To");
    record.add("Overlap");
    csvPrinter.printRecord(record);
    for(int i = 0; i<scores.length; i++){

        double[] toScores = scores[i];
        for(int j = i; j<toScores.length;j++){
            if(toScores[j] > 0.18){
            record = new ArrayList<String>();
            record.add(files.get(i).getName());
            record.add(files.get(j).getName());
            record.add(Double.toString(toScores[j]));
            csvPrinter.printRecord(record);
            }
//              record = new ArrayList<String>();
//              record.add(files.get(j).getName());
//              record.add(files.get(i).getName());
//              record.add(Double.toString(toScores[j]));
//              csvPrinter.printRecord(record);
        }

    }
    csvPrinter.close();
}
```

Here in the printRelations method is where the results of the comparison will be printed to the csv file. I specified a threshold such that the only classes with a comparison score (toScore[j]) is greater than 0.18.



[figure 18 – Here are the results of the of my code duplication analysis. The following diagram the overlap value between the two files, which suggests that the greater the overlap value, the higher the code duplication between those classes. In particular, the overlab value for code duplicate between Java6Assertions sub classes – Java6AbstractBDDSoftAssertions.class and Java6AbstractStandardSoftAssertions.class is 44% which pretty, as it is not ideal and the aim is to have that value as low as possible.]

Re-engineering

Overall, the analysis I have deployed in this re-engineering project were: static analysis and dynamic analysis. I conducted static analysis to asses structure of the source code syntax, without execution the program. The metrics I obtained from the static analysis gave me an insight to what particular classes are complex (based on the weighted method count) and how cohesive the class methods based on the methods that had large number of statements that exceeded or was equal to a 100. I achieved this by calling the computeTightness on every method in the AssertJ project. The method was called on the program dependency graph which consists of a combination of the control dependence with the dataflow between statements. This as a result gave me an insight into the dataflow within the program and most importantly which classes are considered to be complex.

Afterwards, I employed some dynamic analysis, whereby using aspectJ tools to log class executions and method calls invoked the 5 test-classes. Based on the results of the tracing, I was able to identify if there was some correlation between the metrics I obtained from the static analysis. Further, when it came to the visualization of the metrics, I employed some graph visualisations and the scaling the class diagram to illustrate the behaviour between the classes and the classes which I have identified to be complex.

So as a result the following candidates I have selected to be re-engineered are:

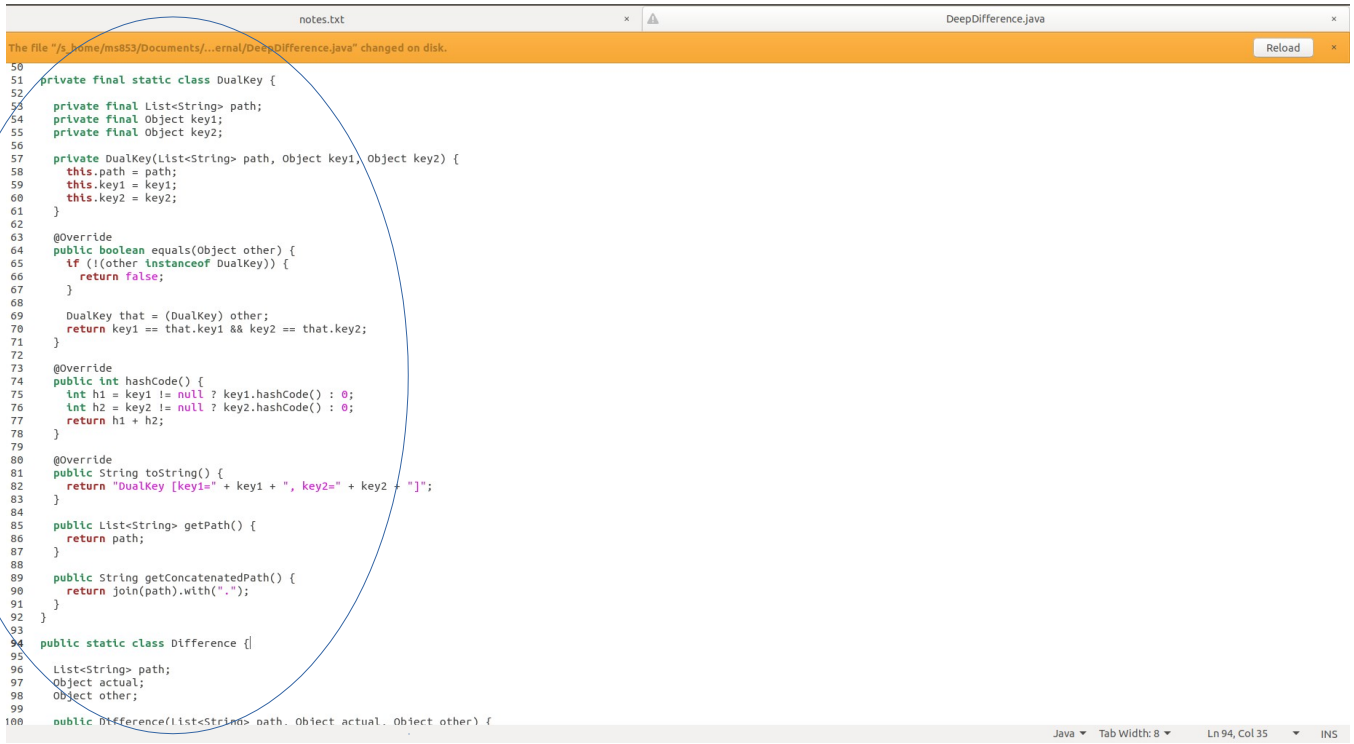
- org.assertj.core.internal.DeepDifference (I will refactor the code such that there will be two classes which will reduce the lines of code, and split the responsibilies of the class).
- **org/assertj/core/presentation/StandardRepresentation (For that I will refactor the toStringOf method in such a way that it)**

Re-engineering Attempt

Re-engineering attempt(1)

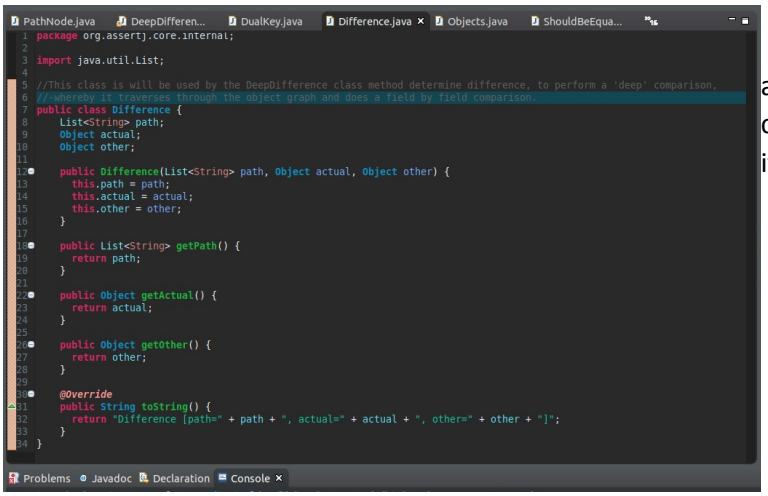
For the first re-engineering attempt I sought out to decrease the lines of code as well as the complexity of the DeepDifference class located in the internal package of the Assertj project.

Firstly, analyised the class and I discovered that the class had internal classes which were: 'DualKey' and 'Difference'. Both classes were declared with the following modifiers: ***'public final static class'*** **(refer to figure 20).** I felt that it was important to enforce some abstraction to the DeepDifference class such that it does not contain the internal classes but rather utilizes those classes as external classes.



[Figure 20 – The DeepDifference Class be seen having two external calles which can be seen in the image above.]

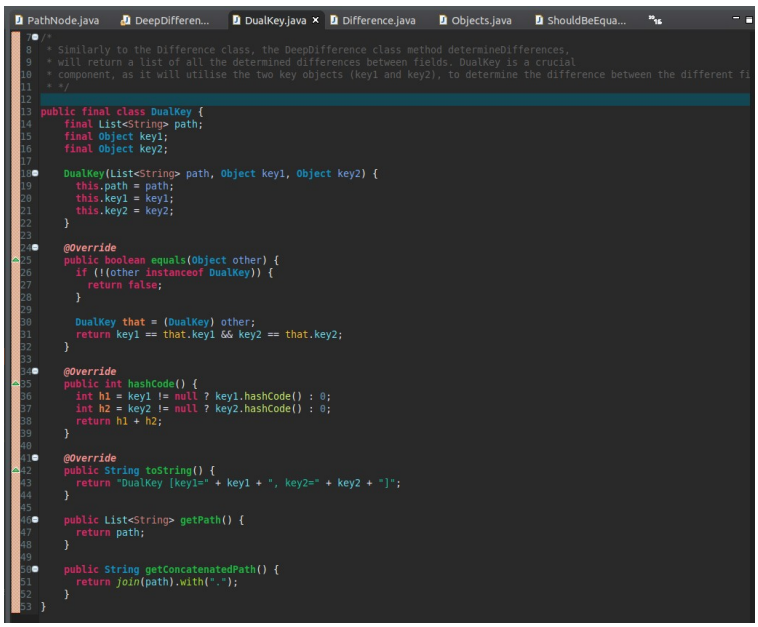
Then I made separate class files for those classes and made I changed the modifiers such that other classes in the AssertJ program especially the Objects class can access the data members of the class and invoke the required method to determine the differences. Then I resolved any compile errors that came as a result of this re-engineering task.



Here is the Difference Class and I have added comments so that there is more clarity on the purpose of the class and how it will be used.

Similarly, I did the same for DualKey, whereby I have stated how the its data members will interact with the DeepDifference class.

Here is a picture of the DualKey.java class located in the internal package:



What was achieved in this re-engineering task?

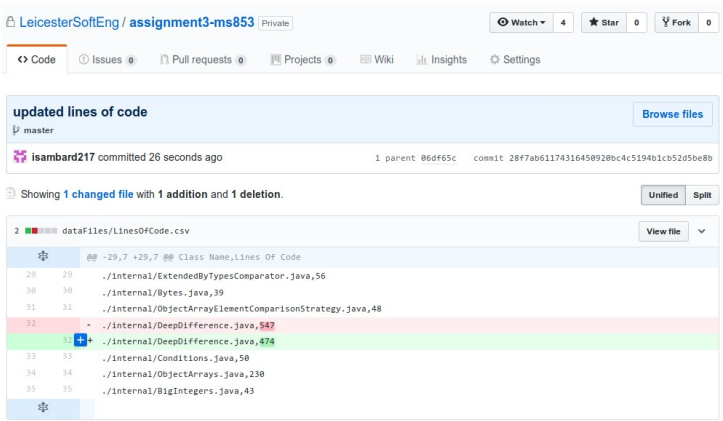
In this re-engineering task I have able to split up the responsiblities of a **God Class** which is **DeepDifference.java**. I have did this by extracting the internal classes and making them into smaller classes(OORP, p.263). That way I was able to

The problem of God Classes in AssertJ and the implications it has on the maintainability of the software

By assuming too many responsibilities, a god class monopolizes control of an application. Evolution of the application is difficult because nearly every change touches this class, and affects multiple responsibilities. This is the issue with assertJ as there is another God Class called object whereby every change made to the code structure of the application can have negative implications on the behaviour of that class and the system.

Furthermore, the lines of code for DeepDifference went down from 547 to 474. As a result, of the code refactoring I have successfully altered the lines of code. So the main objective of the re-engineering task has been achieved.

The following change can be seen here:



In conclusion, the re-engineering task which I have performed has introduced some efficiency to the project. For e.g. by splitting up the DeepDifference class I was able to change the behaviour of the project and reduce the lines of code as a whole for DeepDifference. I have gained not only an understanding of the software and its behaviour but I was able to identify potential weaknesses of the system which needed to be addressed and refactored.

Bibliography

GitHub issues forum for assert-core. Available at: https://github.com/joel-costigliola/assertj-core/issues (Accessed: 18th December 2018).

Cyclomatic Complexity. Available at: http://www.arisa.se/compendium/node96.html (Accessed: 20th December 2018)

Weighted Method Count. Available at: http://www.arisa.se/compendium/node96.html (Accessed 20th December 2018).

Rojas, J. and Walkinshaw, N. (2018). *Measuring Software Quality*. (Accessed: 15th January 2019).

Aspect Oriented Programming. Available at:

https://flowframework.readthedocs.io/en/stable/TheDefinitiveGuide/PartIII/AspectOrientedProgrammi

ng.html (Accessed: 8 th December 2018).

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).

Walkinshaw, N. (2013). *Reverse-Engineering Software Behavior*. Leicester: Elsevier IncU, pp.14, 23, 31, 34, 44, 46, 51. (Accessed: 19th January 2019).

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).