

Re-engineering AssertJ

Initial Exploration

For the initial part of this re-engineering task, the aim was to gain an overview of AssertJ, so I gathered information about the software's function, its 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 in a different git repository under the creator's name. The five different components are: **Guvana**, **Joda Time**, **Neo4J**, **DB** and **Swing** module.

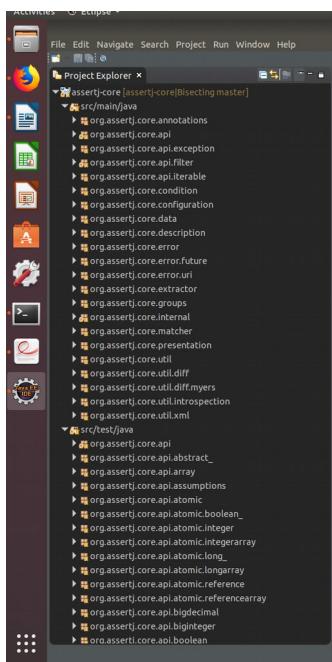
In addition, there is a feature called Assertion Generator, which allows developers to generate assertions code based on their own classes. This allows developers to write assertions specific to their domain model vocabulary.

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, 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 structure – 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.

```

/*
 * Software License, Version 2.0 (the "License"); you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 * http://bit.ly/1IZ2RqY
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */
package org.assertj.core.api;

import static org.assertj.core.api.Assertions.contentOf;

/**
 * Base class for all implementations of assertions for {@code CharSequence}.
 *
 * @param <SELF> the "self" type of this assertion class. Please read Quoet<SELF> href="http://bit.ly/1IZ2RqY" for more details.
 * @param <ACTUAL> the type of the "actual" value.
 */
public abstract class AbstractCharSequenceAssert<SELF extends AbstractCharSequenceAssert<SELF, ACTUAL>, ACTUAL extends CharSequence>
    extends AbstractAssert<SELF, ACTUAL> implements EnumerableAssert<SELF, Character> {
    @VisibleForTesting
    Strings strings = Strings.instance();
    ...
    public AbstractCharSequenceAssert(ACTUAL actual, Class<?> selfType) {
        super(actual, selfType);
    }
    ...
    /**
     * Verifies that the actual {@code CharSequence} is empty, i.e., it has a length of 0, or is {@code null}.
     *
     * <pre>
     * You do not want to accept a {@code null} value, use
     * <code>{@link org.assertj.core.api.AbstractCharSequenceAssert#isNullOrEmpty()}</code> instead.
     * </pre>
     * Both of these assertions will succeed:
     * <pre>
     * <code>java> String emptyString = "";
     * assertThat(emptyString).isNullOrEmpty();
     * </code>
     * <code>String nullString = null;
     * assertThat(nullString).isNullOrEmpty();</code>
     * </pre>
     * Whereas these assertions will fail:
     * <pre>
     * <code>java> assertThat("").isNullOrEmpty();
     * <code>assertThat("").isNullOrEmpty();</code>
     * </pre>
     * @throws AssertionError if the actual {@code CharSequence} has a non-zero length.
     */
    @Override
    public void isNullOrEmpty() {
        strings.assertNullOrEmpty(info, actual);
    }
}

```

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

```
ShouldBeComp... x ElementsShou... MessageForma... ShouldBeAft... BasicErrorMe... %s
20 * Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance
13 package org.assertj.core.error.future;
14
15 import java.util.concurrent.CompletableFuture;
16
17 import org.assertj.core.error.BasicErrorMessageFactory;
18 import org.assertj.core.error.ErrorMessageFactory;
19
20 public class ShouldBeCompletedExceptionally extends BasicErrorMessageFactory {
21
22     private static final String SHOULD_HAVE_COMPLETED_EXCEPTIONALLY =
23             "%nExpecting%n<%s> to be completed exceptionally" +
24             Message.NON_BLOCKING;
25
26     public static ErrorMessageFactory shouldHaveCompletedExceptionally(CompletableFuture<?> actual) {
27         return new ShouldBeCompletedExceptionally(actual);
28     }
29
30     private ShouldBeCompletedExceptionally(CompletableFuture<?> actual) {
31         super(SHOULD_HAVE_COMPLETED_EXCEPTIONALLY, actual);
32     }
33 }
```

| | |
|--|------------|
| core/api/AbstractITWithAbstractAsset.java | 2 + |
| core/asset/core/api/AbstractMapAsset.java | 184 ++++++ |
| core/api/AbstractITWithAnyAsset.java | 2 + |
| asset/core/api/AbstractITWithAsset.java | 67 ++++++ |
| src/main/java/org/assertj/core/api/Assertions.java | 4 + |
| asset/core/api/AbstractITWithAnyType.java | 4 + |
| src/main/java/org/assertj/core/api/Assertions.java | 2 + |
| src/main/java/org/assertj/core/api/Assertions.java | 8 + |
| src/main/java/org/assertj/core/api/MapAsset.java | 24 + |
| java/org/assertj/core/api/SoftProxies.java | 4 + |
| java/org/assertj/core/api/WithAssertions.java | 4 + |
| asset/core/error/ElementsShouldSatisfy.java | 7 + |
| error/ShouldHaveMessageFindingMatchRegex.java | 32 + |
| asset/core/error/ShouldHaveRootCause.java | 74 ++++++ |
| asset/core/error/ShouldNotBeNull.java | 23 + |
| core/error/ZippedElementsShouldSatisfy.java | 5 + |
| asset/core/error/ElementsFromMessages.java | 8 + |
| asset/core/error/ElementsFromMessages.java | 2 + |
| src/main/java/org/assertj/core/internal/Maps.java | 156 ++++++ |
| java/org/assertj/core/internal/Objects.java | 43 + |
| java/org/assertj/core/internal/Throwables.java | 44 +++++ |
| asset/core/util/DoubleComparator.java | 2 + |
| asset/core/util/FloatComparator.java | 2 + |
| asset/core/api/BDSDofAssertionsTest.java | 61 ++++++ |
| asset/core/api/MapAssetBaseTest.java | 12 + |
| asset/core/api/SoftAssertionsTest.java | 62 ++++++ |
| asset/core/api/SoftAssertionsTest.java | 30 + |
| core/error/MapAssetAnySatisfyTest.java | 35 + |
| MapAsset.containsEntriesOfTest.java | 53 + |
| Asset.containsOnlyKeysWithIterableTest.java | 52 +++++ |
| map/MapAsset.extractingFromEntriesTest.java | 96 ++++++ |
| throwableAsset.hasMessageFindingMatchTest.java | 40 +++++ |
| ThrowableAsset.hasRootCauseTest.java | 56 ++++++ |
| error/DescriptionFormatter.formatTest.java | 10 + |
| error/ElementsShouldSatisfy.createTest.java | 30 + |
| ElementsFromMessages.createTest.java | 25 + |
| error/ShouldNotCause.createTest.java | 43 + |
| error/ShouldNotCause.createTest.java | 104 ++++++ |
| asset/core/internal/MapsBaseTest.java | 21 + |
| asset/core/internal/ThrowablesBaseTest.java | 17 + |
| Maps.assetAllSatisfyingConsumerTest.java | 79 ++++++ |
| Maps.assetAnySatisfyingConsumerTest.java | 83 ++++++ |
| Maps.assetContainsOnlyKeyTest.java | 39 + |

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

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

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**.

```
ms853@xanthus: /s_home/ms853/Documents/software-reengineering/assignment3/assertj-core
File Edit View Search Terminal Tabs Help
ms853@xanthus: /s_home/ms853/Documents/software-reengineering/assignment3/assertj... x ms853@xanthus: /s_home/ms853/Documents/software-reengineering/assignment3/assertj... x + ▾
show bisect log.
git bisect run <cmd>...
  use <cmd>... to automatically bisect.

Please use "git help bisect" to get the full man page.
ms853@xanthus:~/Documents/software-reengineering/assignment3/assertj-core$ git log
commit f3bdf02953152da5121952088b4c358b501952 (HEAD -> master, origin/master, origin/HEAD)
Author: Pascal Schumacher <pascalschumacher@gmx.net>
Date:   Sun Dec 16 14:37:12 2018 +0100

  Update ByteBuddy to version 1.9.6
```

[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.

```
1 countLines.sh
- #Iterate through the files and calculate the total number of lines
# for all java files in a given directory.
#Then write the output to a csv file.

echo "Class Name,Lines Of Code" > ../../../../../../dataFiles/LinesOfCode.csv" #write headers
for file in ./find .name*.java
do
    line=$(wc -l < $file)
    comment1=$(grep "\A" $file | wc -l)
    comment2=$(grep "\*" $file | wc -l)
    comment=$((comment1 + comment2))
    total=$((line - comment))
    echo "$file,$total"
done
echo "$file,$total" >> ../../../../../../dataFiles/LinesOfCode.csv" #write to csv.
```

[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 directory 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 written 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 dependency 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 tightness 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 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 functionality 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;)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 |
|--|-----|---|

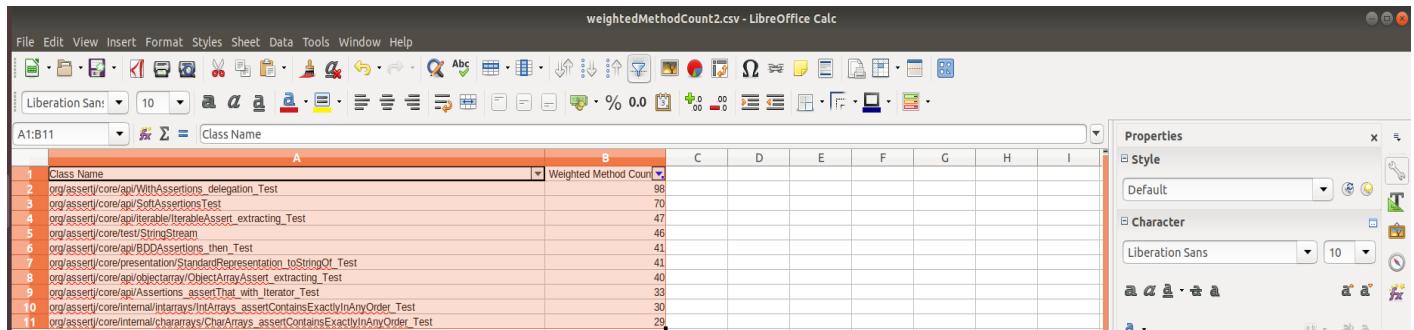
[figure 6 – Top 10 results from the classMetrics.csv, sorted based on the number of nodes.]

| 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]



| | Class Name | Weighted Method Count |
|----|--|-----------------------|
| 1 | org/assertj/core/api/WithAssertions_delegation_Test | 98 |
| 2 | org/assertj/core/api/SoftAssertionsTest | 70 |
| 3 | org/assertj/core/api/IterableAssert_extracting_Test | 47 |
| 4 | org/assertj/core/test/StringStream | 46 |
| 5 | org/assertj/core/api/BDDAssertions_then_Test | 41 |
| 6 | org/assertj/core/presentation/StandardRepresentation_toStringOf_Test | 41 |
| 7 | org/assertj/core/api/ObjectArrayAssert_extracting_Test | 40 |
| 8 | org/assertj/core/api/Assertions_assertThat_with_Iterator_Test | 33 |
| 9 | org/assertj/core/internal/intarrays/IntArrays_assertContainsExactlyInAnyOrder_Test | 30 |
| 10 | org/assertj/core/internal/chararrays/CharArrays_assertContainsExactlyInAnyOrder_Test | 29 |
| 11 | | |

[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 information. 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     Logger.getLogger("Tracing").log(
69         Level.INFO,
70         //sig.getDeclaringType().getSimpleName().toString() + " "
71         sourceName + " " + sig.getName()
72         + " " + paramSb.toString().replaceAll(","," ")
73         + " " + realStackDepth
74     );
75 }
76 }
77 }

```

Here you can see 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 indicate that 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 encompasses the dynamic analysis and it is scaled based on the following metrics: 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.csv";
58     fileName2 = "/s home/ms853/Documents/software-reengineering/assignment3/assignment3-ms853/dataFiles/classOccurrences.csv";
59     fileName3 = "/s home/ms853/Documents/software-reengineering/assignment3/assignment3-ms853/dataFiles/weightedMethodCount.csv";
60
61     BufferedReader reader;
62     BufferedReader reader2 = null;
63     BufferedReader reader3 = null;
64     //BufferedReader reader3 = null;
65     reader = new BufferedReader(new FileReader(fileName1));
66     String[] output;
67     String line = "";
68     String headerLine = "";
69     headerLine = reader.readLine(); //to skip the first header csv file.
70     System.out.println(headerLine);
71     while ((line = reader.readLine()) != null) {
72         output = line.split(",");
73         locMap.put(output[0], Integer.parseInt(output[1]));
74     }
75
76     reader.close();
77
78     reader2 = new BufferedReader(new FileReader(fileName2));
79     String[] output1;
80     String line1 = "";
81     String headerLine1 = "";
82     headerLine1 = reader2.readLine(); //to skip the first header csv file.
83     System.out.println(headerLine1);
84     while ((line1 = reader2.readLine()) != null) {
85         output1 = line1.split(",");
86         classOccursMap.put(output1[0], Integer.parseInt(output1[1].substring(1)));
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 }

```

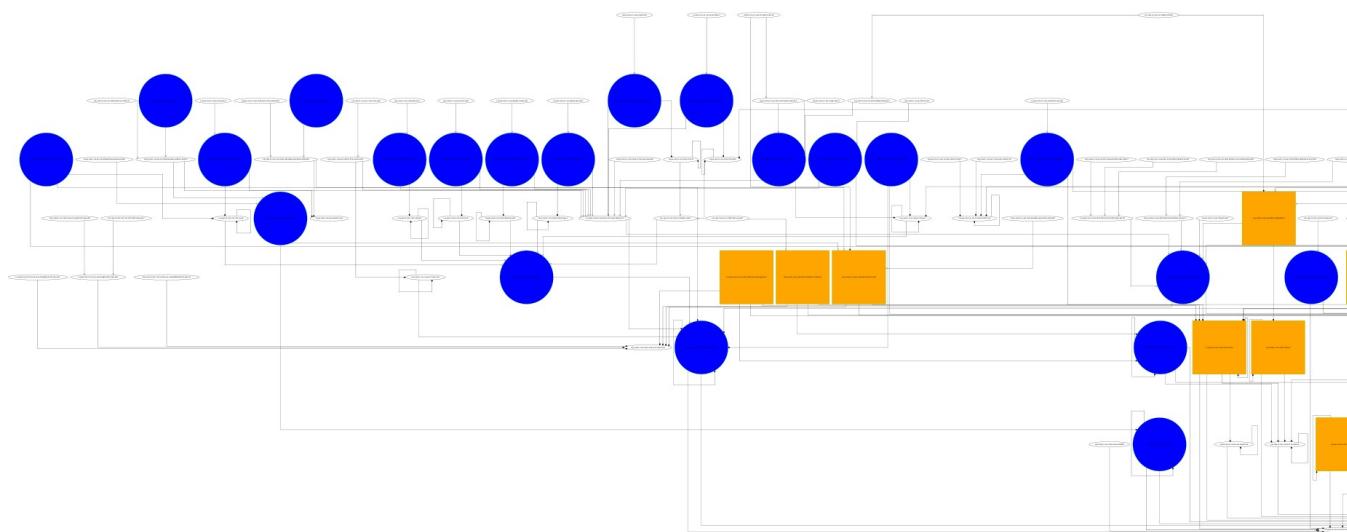
This code snippet is in the main method which where I read three different CSV files for my matrices, which populate my hashmap which I later on used to scale the diagram. Also it's in the same main method where I invoke the **writeDot()** method, which writes out the class diagram as a **dot file**.

```

43 * Firstly, I iterate through the HashMap and within that iteration,
44 * I iterate over the collection of dotGraphClasses.
45 * I then take the key type of the collection.
46 * If that the set of keys in the HashMap consists of that node.
47 * And I scale the class diagram according to the condition.
48 */
49 for(Entry<String, Integer> w : classOccursMap.entrySet()) {
50     for(String node : dotGraphClasses){
51         if(node.contains(w.getKey()) && (w.getValue() >= 500)) {
52             dotGraph.append(node+ "[shape = box height = 10 width = 10 fillcolor=purple style=filled];\n");
53         }
54     }
55 }
56
57 //Alter class diagram based on the lines of code.
58 for(Entry<String, Integer> lc : locMap.entrySet()) {
59     //some refactoring to the key so it can be compared with the node.
60     String className = org.assertj.core+ lc.getKey().substring(1, lc.getKey().length() -5).replaceAll("/", ".");
61
62     for(String node : dotGraphClasses){
63
64         // if lines of code is greater or equal to 100 then modify the classes.
65         if(node.contains(className) && (lc.getValue() >= 100)) {
66             dotGraph.append(node+ "[shape = circle height = 5 width = 5] [color=green fillcolor=blue style=filled];");
67             if(lc.getValue() >300) {
68                 dotGraph.append(node+ "[shape = doublecircle height = 7 width = 7] [color=black fillcolor=red style=filled];");
69             }
70         }
71     }
72 }
73
74
75 //This part of the code alters the structure of the class diagram based on the weighted
76 for(Entry<String, Integer> wc : wmcMap.entrySet()) {
77     //some refactoring to the key so it can be compared with the node.
78     String className = wc.getKey().replaceAll("/", ".");
79
80     for(String node : dotGraphClasses){
81
82         if(node.contains(className) && (wc.getValue() >= 100)) {
83             dotGraph.append(node+ "[shape = box height = 5 width = 5] [color=green fillcolor=orange style=filled];");
84             System.out.println("L000K ----- " + node);
85         }
86     }
87 }
88
89 }

```

[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 noticeable classe that appeared in the wmc metric for the test -classes, that class is:

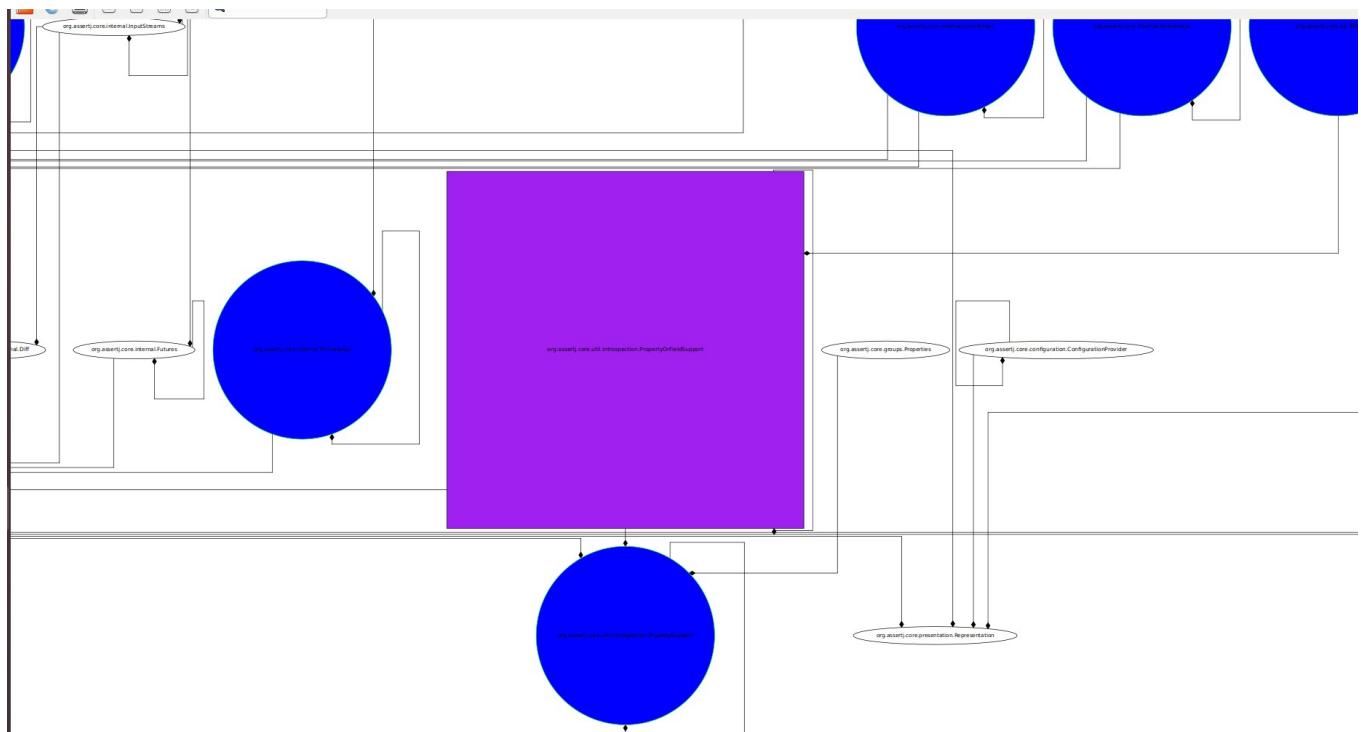
- `org.assertj.core.IterableAssert` (which is the class counterpart to the `IterableAssert_extacting_Test` class).

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):

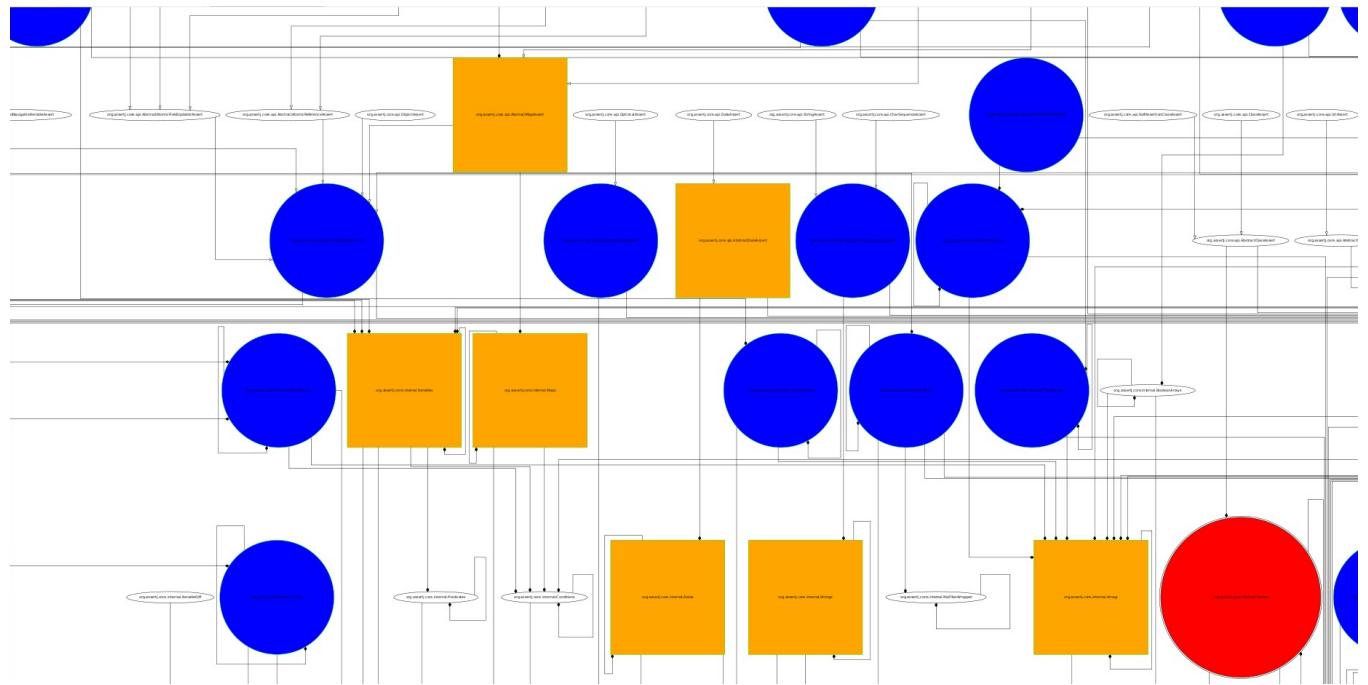
- org.assertj.core.api.AtomicReferenceArrayAssert
- **org.assertj.core.internal.Iterables**
- org.assertj.core.api.AbstractObjectArrayAssert
- **org.assertj.core.internal.DeepDifference** (A parent class that I will possibly look into to reduce the lines of code).
- org.assertj.core.internal.DeepDifference\$DualKey
- 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 noticeable classes that appeared in the wmc metrics are the same classes that have appeared to have a large number of LOC. This can possibly 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 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]



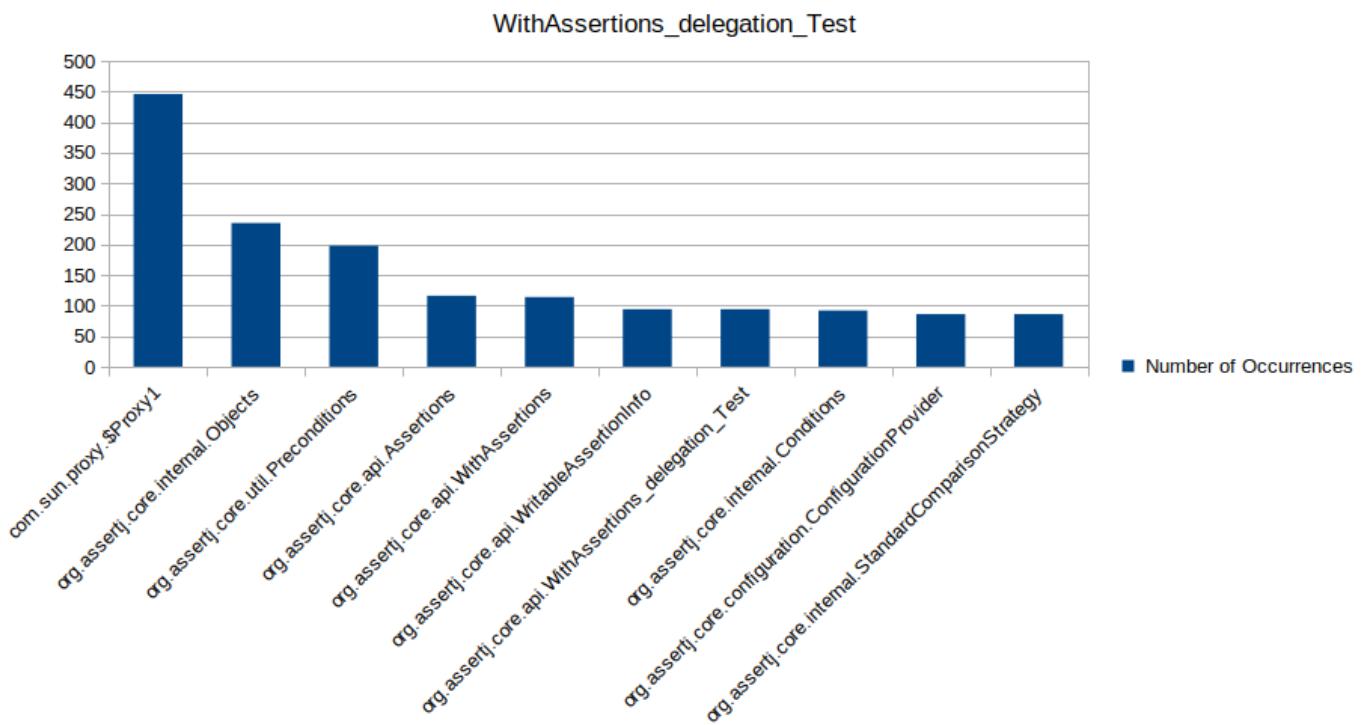
[Figure 12 – This part of the class diagram shows all the complex classes colour coded in orange. The threshold set for the weighted method count are for classes with values exceeding **100**. Once again, the classes that appear here where there is a high concentration of inheritance and associations 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, who 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.

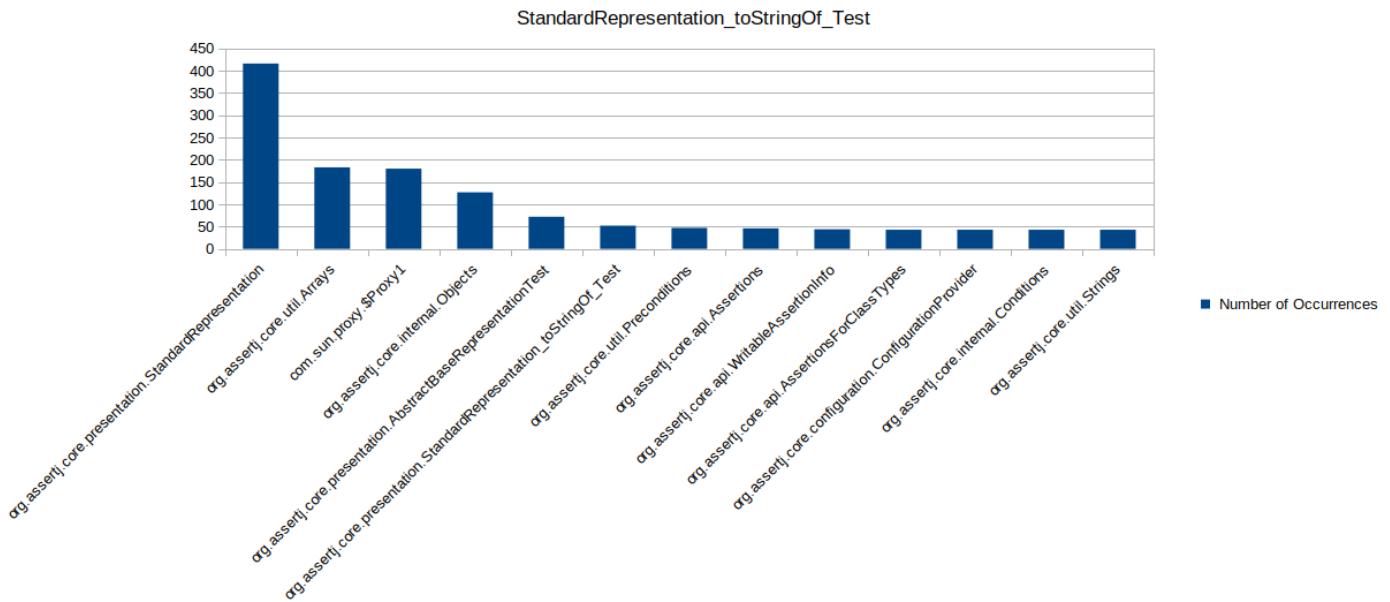
The Big Picture

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.

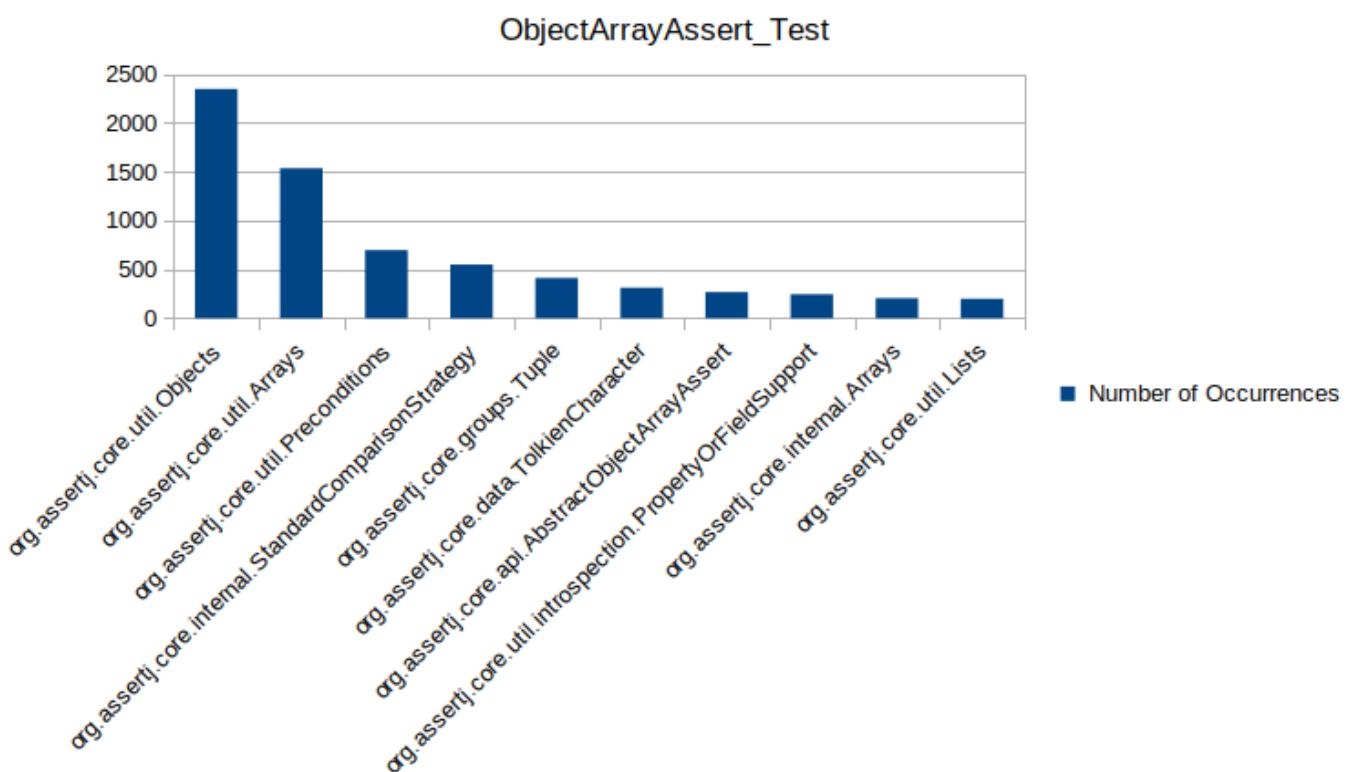
To generate visualizations, I started by reading the data from the 5 trace-files which I used as part of the dynamic analysis. The 5 trace files, which are files listed up which generated based on the test classes, which are counterparts of the re-engineering I wrote a new class called **ParseTraceToCSV.java**, 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 written to a csv file corresponding to each trace-file.



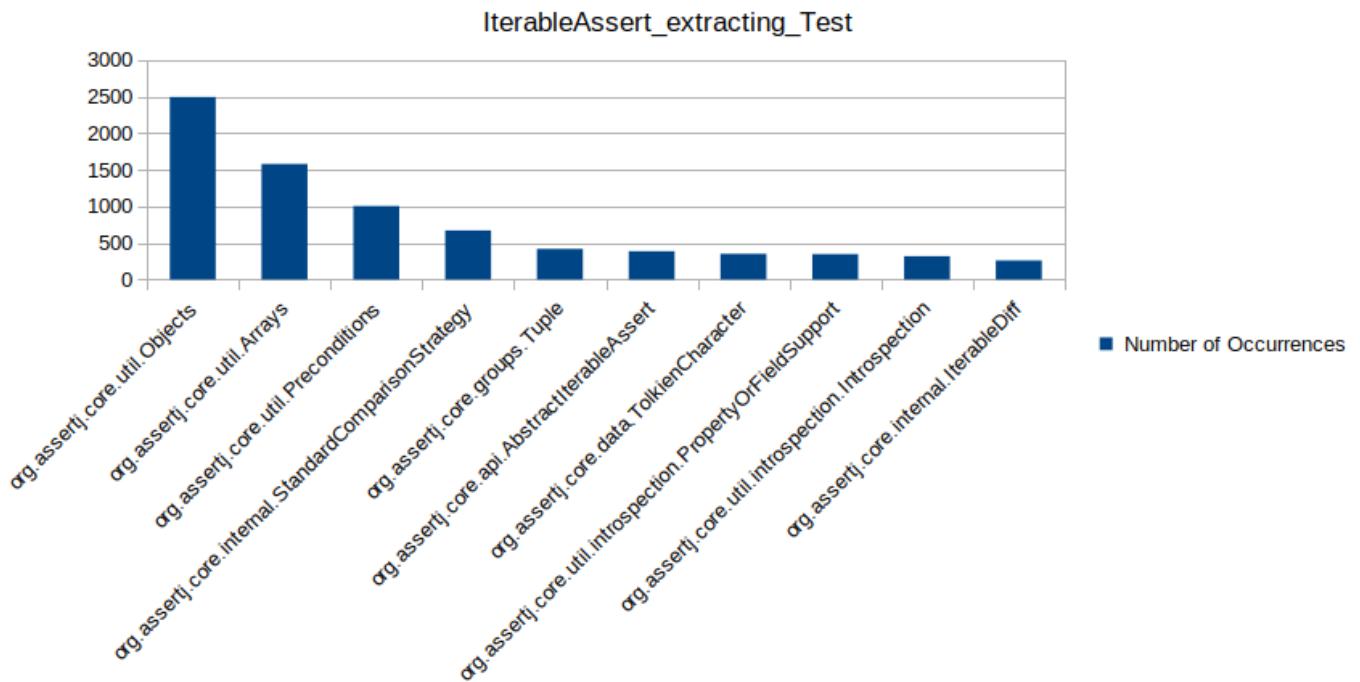
[Figure 13 – This is a bar chart shows the 10 classes that have been frequently executed by this test file. Here the class that are executed the most: Objects, Preconditions and Assertions and WithAssertions. Assertions in this case the second most executed class yet I know that based on my previous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



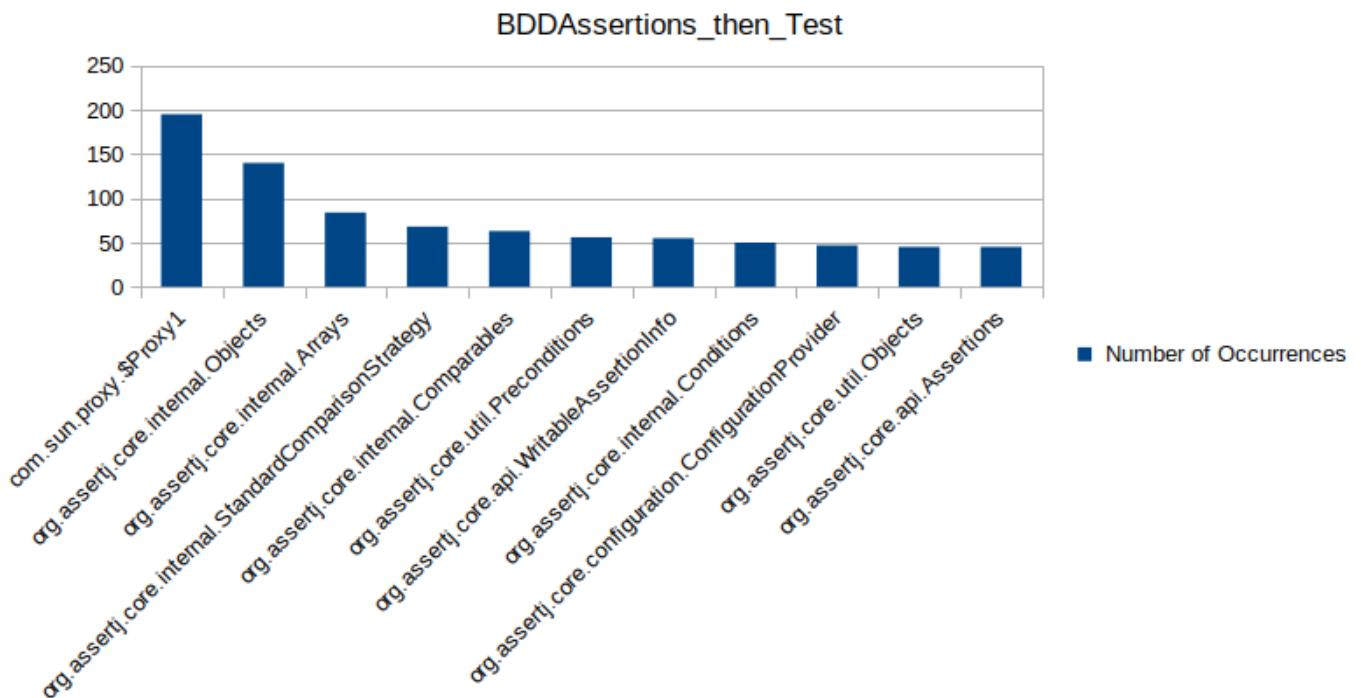
[figure 14 - The highest class that is executed by this test-class is StandardRepresentation (over 400 times). Likewise, Arrays class which is the second highest class executed also has a has a wmc value greater or equal to a 100.]



[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

```

10  public double[][] fileComparison(boolean proportional){
11      double[][] fileCompare = new double[files.size()][files.size()];
12      for(int i = 0; i<files.size(); i++){
13          for(int j = i+1; j<files.size(); j++){
14              double score;
15              if(i == j)
16                  score = 0;
17              else
18                  FileComparator fc = new FileComparator(files.get(i),files.get(j));
19                  score = fc.coarseCompare(proportional);
20              }
21              fileCompare[i][j] = score;
22              fileCompare[j][i] = score;
23          }
24      }
25      return fileCompare;
26  }
27
28  public List<File> getFiles(){
29      return files;
30  }
31
32 }

```

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 directory and compare one file against the other. Below is the implementation of the class FileComparator class.

```

1  ClassMetrics...  DuplicateDet...  FileComparat...  JavaAsserti...  StandardRep...  IterableAsse...
2
3  try {
4      BufferedReader br = new BufferedReader(new FileReader(file));
5      String line = br.readLine();
6      while(line != null) {
7          line.trim();
8          fromFile.add(line);
9          line = br.readLine();
10     }
11     br.close();
12 } catch (FileNotFoundException e) {
13     e.printStackTrace();
14 } catch (IOException e) {
15     e.printStackTrace();
16 }
17
18 public boolean[][] detailedCompare() {
19     List<String> fromFile = new ArrayList<String>();
20     List<String> toFile = new ArrayList<String>();
21     extractStrings(fromFile, from);
22     extractStrings(toFile, to);
23     boolean match = new boolean[fromFile.size()][toFile.size()];
24     for(int i = 0; i<fromFile.size(); i++){
25         for(int j = 0; j<toFile.size(); j++){
26             boolean match = fromFile.get(i).equals(toFile.get(j));
27             if(fromFile.get(i).length()==0 || fromFile.get(i).equals(" ")|| fromFile.get(i).equals("\n"))
28                 match = false;
29         }
30     }
31     return matrix;
32 }
33
34 }

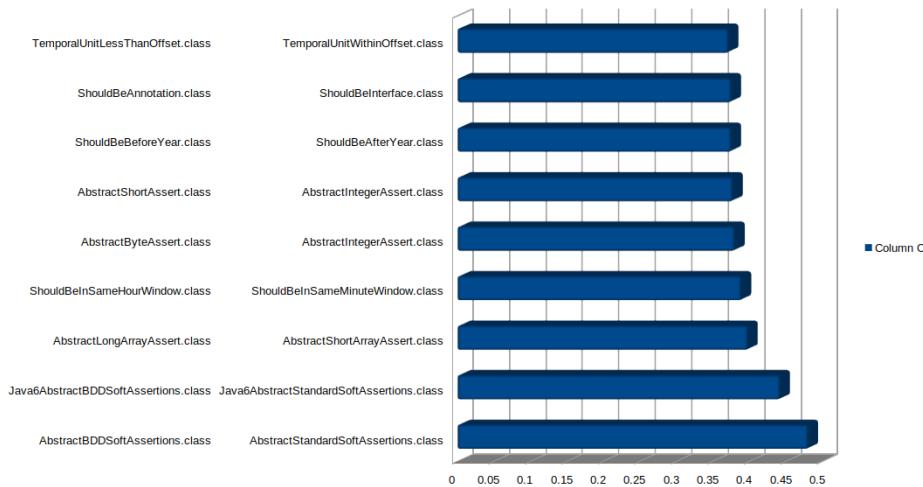
```

```

dynamicAnaly...  DuplicateDet...  TablePrint...  ParseTraceTo...  JavaAsserti...  StandardRep...
1  public static void printRelations(double[][] scores, File target, List<File> files) throws IOException {
2      FileWriter fw = new FileWriter(target);
3      CSVPrinter csvPrinter = new CSVPrinter(fw, CSVFormat.EXCEL);
4      List<String> record = new ArrayList<String>();
5      record.add("From");
6      record.add("To");
7      record.add("Score");
8      csvPrinter.printRecord(record);
9      for(int i = 0; i<scores.length; i++){
10         double[] toScores = scores[i];
11         for(int j = i; j<toScores.length; j++){
12             if(toScores[j] > 0.10){
13                 record = new ArrayList<String>();
14                 record.add(files.get(i).getFileName());
15                 record.add(files.get(j).getFileName());
16                 record.add(Double.toString(toScores[j]));
17                 csvPrinter.printRecord(record);
18             }
19             record = new ArrayList<String>();
20             record.add(files.get(i).getFileName());
21             record.add(files.get(j).getFileName());
22             record.add(Double.toString(toScores[j]));
23             csvPrinter.printRecord(record);
24         }
25     }
26     csvPrinter.close();
27 }

```

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 overlap 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:

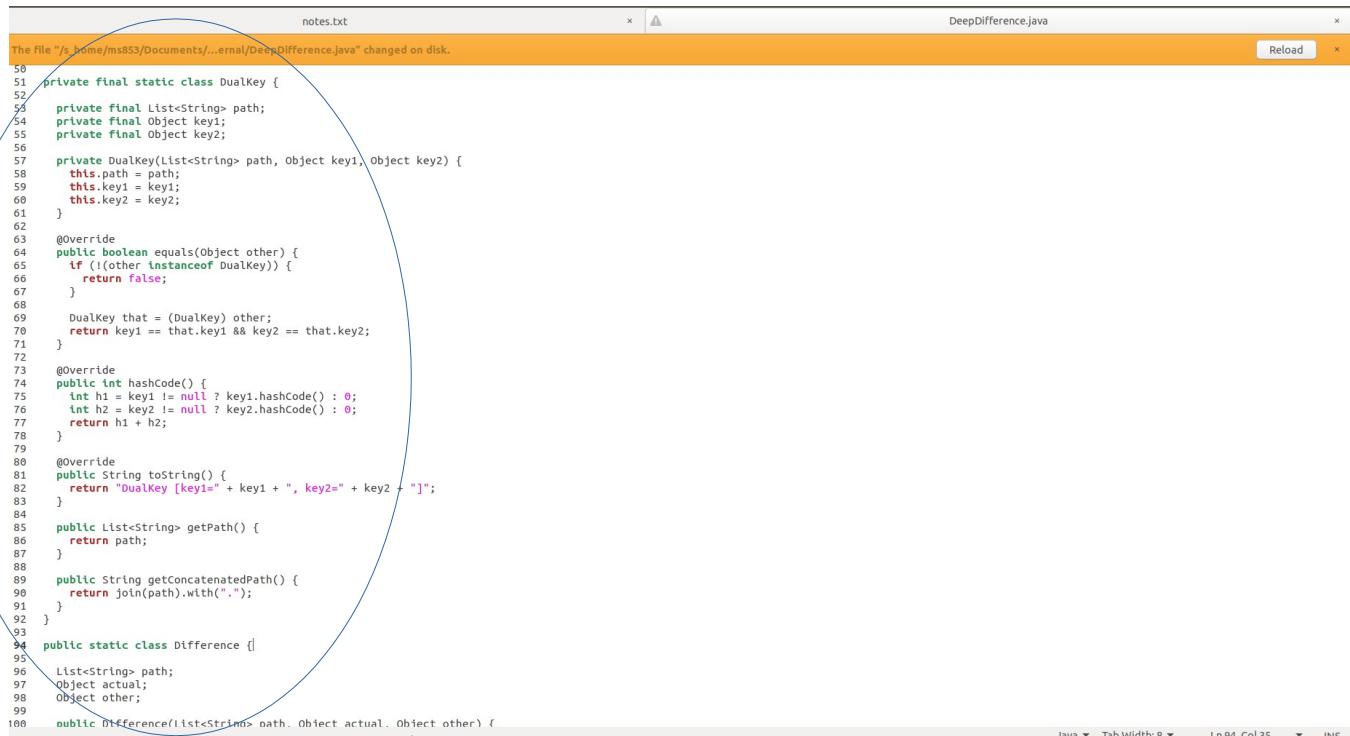
- `org.assertj.core.internal.DeepDifference` (I will refactor the code such that there will be two classes which will reduce the lines of code, and split the responsibilities of the class).
- `org/assertj/core/presentation/StandardRepresentation` (For that I will refactor the `toStringOf` method in such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



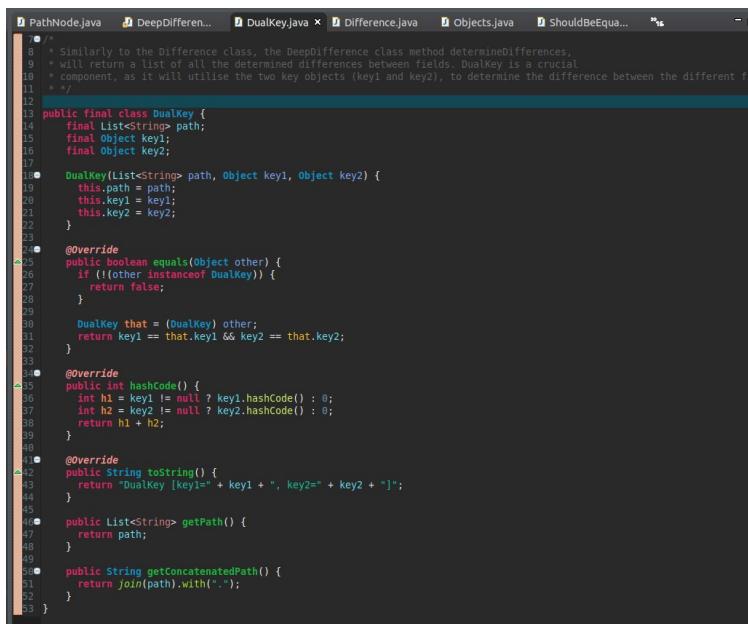
```

notes.txt
DeepDifference.java

58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
779
780
781
782
783
784
785
786
787
788
789
789
790
791
792
793
794
795
796
797
797
798
799
800
801
802
803
804
805
806
807
808
809
809
810
811
812
813
814
815
816
817
817
818
819
819
820
821
822
823
823
824
825
825
826
827
827
828
828
829
829
830
830
831
831
832
832
833
833
834
834
835
835
836
836
837
837
838
838
839
839
840
840
841
841
842
842
843
843
844
844
845
845
846
846
847
847
848
848
849
849
850
850
851
851
852
852
853
853
854
854
855
855
856
856
857
857
858
858
859
859
860
860
861
861
862
862
863
863
864
864
865
865
866
866
867
867
868
868
869
869
870
870
871
871
872
872
873
873
874
874
875
875
876
876
877
877
878
878
879
879
880
880
881
881
882
882
883
883
884
884
885
885
886
886
887
887
888
888
889
889
890
890
891
891
892
892
893
893
894
894
895
895
896
896
897
897
898
898
899
899
900
900
901
901
902
902
903
903
904
904
905
905
906
906
907
907
908
908
909
909
910
910
911
911
912
912
913
913
914
914
915
915
916
916
917
917
918
918
919
919
920
920
921
921
922
922
923
923
924
924
925
925
926
926
927
927
928
928
929
929
930
930
931
931
932
932
933
933
934
934
935
935
936
936
937
937
938
938
939
939
940
940
941
941
942
942
943
943
944
944
945
945
946
946
947
947
948
948
949
949
950
950
951
951
952
952
953
953
954
954
955
955
956
956
957
957
958
958
959
959
960
960
961
961
962
962
963
963
964
964
965
965
966
966
967
967
968
968
969
969
970
970
971
971
972
972
973
973
974
974
975
975
976
976
977
977
978
978
979
979
980
980
981
981
982
982
983
983
984
984
985
985
986
986
987
987
988
988
989
989
990
990
991
991
992
992
993
993
994
994
995
995
996
996
997
997
998
998
999
999
1000
1000
1001
1001
1002
1002
1003
1003
1004
1004
1005
1005
1006
1006
1007
1007
1008
1008
1009
1009
1010
1010
1011
1011
1012
1012
1013
1013
1014
1014
1015
1015
1016
1016
1017
1017
1018
1018
1019
1019
1020
1020
1021
1021
1022
1022
1023
1023
1024
1024
1025
1025
1026
1026
1027
1027
1028
1028
1029
1029
1030
1030
1031
1031
1032
1032
1033
1033
1034
1034
1035
1035
1036
1036
1037
1037
1038
1038
1039
1039
1040
1040
1041
1041
1042
1042
1043
1043
1044
1044
1045
1045
1046
1046
1047
1047
1048
1048
1049
1049
1050
1050
1051
1051
1052
1052
1053
1053
1054
1054
1055
1055
1056
1056
1057
1057
1058
1058
1059
1059
1060
1060
1061
1061
1062
1062
1063
1063
1064
1064
1065
1065
1066
1066
1067
1067
1068
1068
1069
1069
1070
1070
1071
1071
1072
1072
1073
1073
1074
1074
1075
1075
1076
1076
1077
1077
1078
1078
1079
1079
1080
1080
1081
1081
1082
1082
1083
1083
1084
1084
1085
1085
1086
1086
1087
1087
1088
1088
1089
1089
1090
1090
1091
1091
1092
1092
1093
1093
1094
1094
1095
1095
1096
1096
1097
1097
1098
1098
1099
1099
1100
1100
1101
1101
1102
1102
1103
1103
1104
1104
1105
1105
1106
1106
1107
1107
1108
1108
1109
1109
1110
1110
1111
1111
1112
1112
1113
1113
1114
1114
1115
1115
1116
1116
1117
1117
1118
1118
1119
1119
1120
1120
1121
1121
1122
1122
1123
1123
1124
1124
1125
1125
1126
1126
1127
1127
1128
1128
1129
1129
1130
1130
1131
1131
1132
1132
1133
1133
1134
1134
1135
1135
1136
1136
1137
1137
1138
1138
1139
1139
1140
1140
1141
1141
1142
1142
1143
1143
1144
1144
1145
1145
1146
1146
1147
1147
1148
1148
1149
1149
1150
1150
1151
1151
1152
1152
1153
1153
1154
1154
1155
1155
1156
1156
1157
1157
1158
1158
1159
1159
1160
1160
1161
1161
1162
1162
1163
1163
1164
1164
1165
1165
1166
1166
1167
1167
1168
1168
1169
1169
1170
1170
1171
1171
1172
1172
1173
1173
1174
1174
1175
1175
1176
1176
1177
1177
1178
1178
1179
1179
1180
1180
1181
1181
1182
1182
1183
1183
1184
1184
1185
1185
1186
1186
1187
1187
1188
1188
1189
1189
1190
1190
1191
1191
1192
1192
1193
1193
1194
1194
1195
1195
1196
1196
1197
1197
1198
1198
1199
1199
1200
1200
1201
1201
1202
1202
1203
1203
1204
1204
1205
1205
1206
1206
1207
1207
1208
1208
1209
1209
1210
1210
1211
1211
1212
1212
1213
1213
1214
1214
1215
1215
1216
1216
1217
1217
1218
1218
1219
1219
1220
1220
1221
1221
1222
1222
1223
1223
1224
1224
1225
1225
1226
1226
1227
1227
1228
1228
1229
1229
1230
1230
1231
1231
1232
1232
1233
1233
1234
1234
1235
1235
1236
1236
1237
1237
1238
1238
1239
1239
1240
1240
1241
1241
1242
1242
1243
1243
1244
1244
1245
1245
1246
1246
1247
1247
1248
1248
1249
1249
1250
1250
1251
1251
1252
1252
1253
1253
1254
1254
1255
1255
1256
1256
1257
1257
1258
1258
1259
1259
1260
1260
1261
1261
1262
1262
1263
1263
1264
1264
1265
1265
1266
1266
1267
1267
1268
1268
1269
1269
1270
1270
1271
1271
1272
1272
1273
1273
1274
1274
1275
1275
1276
1276
1277
1277
1278
1278
1279
1279
1280
1280
1281
1281
1282
1282
1283
1283
1284
1284
1285
1285
1286
1286
1287
1287
1288
1288
1289
1289
1290
1290
1291
1291
1292
1292
1293
1293
1294
1294
1295
1295
1296
1296
1297
1297
1298
1298
1299
1299
1300
1300
1301
1301
1302
1302
1303
1303
1304
1304
1305
1305
1306
1306
1307
1307
1308
1308
1309
1309
1310
1310
1311
1311
1312
1312
1313
1313
1314
1314
1315
1315
1316
1316
1317
1317
1318
1318
1319
1319
1320
1320
1321
1321
1322
1322
1323
1323
1324
1324
1325
1325
1326
1326
1327
1327
1328
1328
1329
1329
1330
1330
1331
1331
1332
1332
1333
1333
1334
1334
1335
1335
1336
1336
1337
1337
1338
1338
1339
1339
1340
1340
1341
1341
1342
1342
1343
1343
1344
1344
1345
1345
1346
1346
1347
1347
1348
1348
1349
1349
1350
1350
1351
1351
1352
1352
1353
1353
1354
1354
1355
1355
1356
1356
1357
1357
1358
1358
1359
1359
1360
1360
1361
1361
1362
1362
1363
1363
1364
1364
1365
1365
1366
1366
1367
1367
1368
1368
1369
1369
1370
1370
1371
1371
1372
1372
1373
1373
1374
1374
1375
1375
1376
1376
1377
1377
1378
1378
1379
1379
1380
1380
1381
1381
1382
1382
1383
1383
1384
1384
1385
1385
1386
1386
1387
1387
1388
1388
1389
1389
1390
1390
1391
1391
1392
1392
1393
1393
1394
1394
1395
1395
1396
1396
1397
1397
1398
1398
1399
1399
1400
1400
1401
1401
1402
1402
1403
1403
1404
1404
1405
1405
1406
1406
1407
1407
1408
1408
1409
1409
1410
1410
1411
1411
1412
1412
1413
1413
1414
1414
1415
1415
1416
1416
1417
1417
1418
1418
1419
1419
1420
1420
1421
1421
1422
1422
1423
1423
1424
1424
1425
1425
1426
1426
1427
1427
1428
1428
1429
1429
1430
1430
1431
1431
1432
1432
1433
1433
1434
1434
1435
1435
1436
1436
1437
1437
1438
1438
1439
1439
1440
1440
1441
1441
1442
1442
1443
1443
1444
1444
1445
1445
1446
1446
1447
1447
1448
1448
1449
1449
1450
1450
1451
1451
1452
1452
1453
1453
1454
1454
1455
1455
1456
1456
1457
1457
1458
1458
1459
1459
1460
1460
1461
1461
1462
1462
1463
1463
1464
1464
1465
1465
1466
1466
1467
1467
1468
1468
1469
1469
1470
1470
1471
1471
1472
1472
1473
1473
1474
1474
1475
1475
1476
1476
1477
1477
1478
1478
1479
1479
1480
1480
1481
1481
1482
1482
1483
1483
1484
1484
1485
1485
1486
1486
1487
1487
1488
1488
1489
1489
1490
1490
1491
1491
1492
1492
1493
1493
1494
1494
1495
1495
1496
1496
1497
1497
1498
1498
1499
1499
1500
1500
1501
1501
1502
1502
1503
1503
1504
1504
1505
1505
1506
1506
1507
1507
1508
1508
1509
1509
1510
1510
1511
1511
1512
1512
1513
1513
1514
1514
1515
1515
1516
1516
1517
1517
1518
1518
1519
1519
1520
1520
1521
1521
1522
1522
1523
1523
1524
1524
1525
1525
1526
1526
1527
1527
1528
1528
1529
1529
1530
1530
1531
1531
1532
1532
1533
1533
1534
1534
1535
1535
1536
1536
1537
1537
1538
1538
1539
1539
1540
1540
1541
1541
1542
1542
1543
1543
1544
1544
1545
1545
1546
1546
1547
1547
1548
1548
1549
1549
1550
1550
1551
1551
1552
1552
1553
1553
1554
1554
1555
1555
1556
1556
1557
1557
1558
1558
1559
1559
1560
1560
1561
1561
1562
1562
1563
1563
1564
1564
1565
1565
1566
1566
1567
1567
1568
1568
1569
1569
1570
1570
1571
1571
1572
1572
1573
1573
1574
1574
1575
1575
1576
1576
1577
1577
1578
1578
1579
1579
1580
1580
1581
1581
1582
1582
1583
1583
1584
1584
1585
1585
1586
1586
1587
1587
1588
1588
1589
1589
1590
1590
1591
1591
1592
1592
1593
1593
1594
1594
1595
1595
1596
1596
1597
1597
1598
1598
1599
1599
1600
1600
1601
1601
1602
1602
1603
1603
1604
1604
1605
1605
1606
1606
1607
1607
1608
1608
1609
1609
1610
1610
1611
1611
1612
1612
1613
1613
1614
1614
1615
1615
1616
1616
1617
1617
1618
1618
1619
1619
1620
1620
1621
1621
1622
1622
1623
1623
1624
1624
1625
1625
1626
1626
1627
16
```

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

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



```

1  /*
2  *  Similarly to the Difference class, the DeepDifference class method determineDifferences,
3  *  will return a list of all the determined differences between fields. DualKey is a crucial
4  *  component, as it will utilise the two key objects (key1 and key2), to determine the difference between the different fi
5  */
6
7  public final class DualKey {
8
9     final List<String> path;
10    final Object key1;
11    final Object key2;
12
13    DualKey(List<String> path, Object key1, Object key2) {
14        this.path = path;
15        this.key1 = key1;
16        this.key2 = key2;
17    }
18
19    @Override
20    public boolean equals(Object other) {
21        if (!(other instanceof DualKey)) {
22            return false;
23        }
24
25        DualKey that = (DualKey) other;
26        return key1 == that.key1 && key2 == that.key2;
27    }
28
29    @Override
30    public int hashCode() {
31        int h1 = key1 != null ? key1.hashCode() : 0;
32        int h2 = key2 != null ? key2.hashCode() : 0;
33        return h1 + h2;
34    }
35
36    @Override
37    public String toString() {
38        return "DualKey [key1=" + key1 + ", key2=" + key2 + "]";
39    }
40
41    public List<String> getPath() {
42        return path;
43    }
44
45    public String getConcatenatedPath() {
46        return join(path).with(".");
47    }
48
49 }
50
51
52 }
53

```

What was achieved in this re-engineering task?

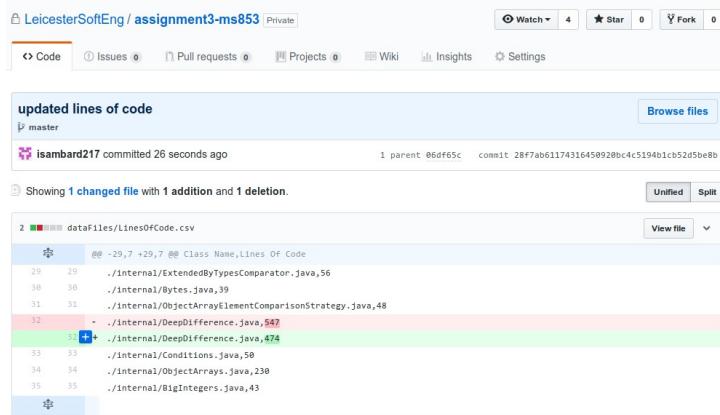
In this re-engineering task I have been able to split up the responsibilities of a **God Class** which is **DeepDifference.java**. I have done this by extracting the internal classes and making them into smaller classes(OORP, p.263). That way I was able to

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:



updated lines of code

isambard217 committed 26 seconds ago

1 parent 06df65c commit 28f7ab61174316459920bc4c5194b1cb52d5be8b

Showing 1 changed file with 1 addition and 1 deletion.

2 datafiles/LinesOfCode.csv

| | | |
|----|----|---|
| 29 | 29 | @@ -29,7 +29,7 @@ Class Name,Lines Of Code |
| 30 | 30 | ./Internal/ExtendedByTypesComparator.java,56 |
| 31 | 31 | ./Internal/Bytes.java,39 |
| 32 | 32 | ./Internal/ObjectArrayListElementComparisonStrategy.java,48 |
| 33 | 33 | - ./Internal/DeepDifference.java,547 |
| 34 | 34 | + ./Internal/DeepDifference.java,474 |
| 35 | 35 | ./Internal/Conditions.java,50 |
| | | ./Internal/ObjectArrays.java,230 |
| | | ./Internal/BigIntegers.java,43 |

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/AspectOrientedProgramming.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).