System Re-engineering

Assignment 3:

Re-Engineering AssertJ

Name: Shashidar Ette

User-id: se146

**NOTE:**

1. The code modified or updated during the course work are details ***Code Change*** boxes
2. Oututputs generated during analysis are available under [*dataFiles*]
3. Supplementary files along with the report are available under [*notesAndReports*] folder

Table of Contents

[Overview 4](#_Toc502958873)

[Initial Exploration 4](#_Toc502958874)

[History 4](#_Toc502958875)

[Technical background 4](#_Toc502958876)

[System perspective 5](#_Toc502958877)

[Metrics and Analysis 6](#_Toc502958878)

[Using Bash Scripts 7](#_Toc502958879)

[Using the .class files 8](#_Toc502958880)

[Junit – Code Coverage 10](#_Toc502958881)

[Repository log mining 11](#_Toc502958882)

[Trace Analysis 11](#_Toc502958883)

[The Big Picture 12](#_Toc502958884)

[Visualisation - Code City 12](#_Toc502958885)

[File Comparision for code duplication 14](#_Toc502958886)

[Class Diagram 17](#_Toc502958887)

[Code churn from release to release 19](#_Toc502958888)

[Reengineering 21](#_Toc502958889)

[Re-engineered Sections 23](#_Toc502958890)

[Conclusion 24](#_Toc502958891)

[References 25](#_Toc502958892)

[1. Costigliola, J. 25](#_Toc502958893)

[2. AssertJ: API changes review 25](#_Toc502958894)

Table of Figures

[Figure 1 Comparison between main source and JUnit source w.r.t number of classes/packages 6](#_Toc502958895)

[Figure 2 Project Dependencies in Eclipse 6](#_Toc502958896)

[Figure 3 Lines, LOC, Comments, % of Comments 7](#_Toc502958897)

[Figure 4 Candidate classes for Re-engineering from code documentation perspective 8](#_Toc502958898)

[Figure 5 Candidate methods (CC > 10) ordered by cyclometric complexity 9](#_Toc502958899)

[Figure 6 Candidate classes for Re-engineering based on number of methods 10](#_Toc502958900)

[Figure 7 Code coverage by JUnit tests 10](#_Toc502958901)

[Figure 8 Candidate classes for re-engineering based on code coverage 10](#_Toc502958902)

[Figure 9 Repository mining - code churn 11](#_Toc502958903)

[Figure 10 Trace Analysis outputs 12](#_Toc502958904)

[Figure 11 AbstractAssert superclass 18](#_Toc502958905)

[Figure 12 core.internal.Arrays - composed within arrays assertion classes 19](#_Toc502958906)

[Figure 13 GOD classes 19](#_Toc502958907)

[Figure 14 API changes viewer[2] 20](#_Toc502958908)

[Figure 15 Code churn from Release 2.8.0 to Release 3.9.0 20](#_Toc502958909)

# Overview

This document details the analysis of AssertJ core from system re-engineering perspective. The sections within the document begin with an introduction to the system and then dwelling into details of it with the help of static and dynamic analysis techniques covered as part of System Re-engineering module.

# Initial Exploration

As part of the initial exploration phase, I have primarily applied “Skim Documentation” pattern to understand the motivation behind the project and documentation available in terms of project overview, environment setup.

As part of “Chat with the Maintainers” pattern, I have also contacted with Joel Costigliola ([joel.costigliola@gmail.com](mailto:joel.costigliola@gmail.com)) for his views on the current state of the system and if he personally has any set of improvements or extended to be to the library. Incidentally, I got hold of the [Interview with Joel Costigliola, creator of AssertJ](http://www.mauricioaniche.com/2017/07/interview-with-joel-costigliola-assertj/) which sheds light on some of key information. In addition, there an active Google+ forum (Joined it) with discussions dated till last week <https://groups.google.com/forum/?fromgroups=#!forum/assertj> but mostly GitHub is used as discussion by contributors.

## History

* The motivation behind the library came from the limitation of Junit and assert instructions available with it.
* Initially Joel was contributing to [Fest Assert](https://github.com/alexruiz/fest-assert-2.x) created by [Alex Ruiz](https://github.com/alexruiz), but after some time upon request of Joel and Alex’s consent AssertJ got created as a fork for Fest Assert 2.x.
  + The main reasons to create a fork of Fest Assert were:
    - FEST 2.0 provided limited set of assertions than FEST 1.x
    - It was not open enough to users and contributors.
* Joel mentioned to keep AssertJ a more “community-driven” project.

## Technical background

* “AssertJ core is a Java library that provides a fluent interface for writing assertions. Its main goal is to improve test code readability and make maintenance of tests easier.” [[1](#_Costigliola,_J.)]
* AssertJ helps developers to create unit tests for complex objects including lists and strings.
* The main project website is here: <http://joel-costigliola.github.io/assertj/assertj-core.html>
* As a developer, the system can be fairly understood at high level via [ReadMe](https://github.com/joel-costigliola/assertj-core/blob/master/README.md)
* Quick start guide is available at: <http://joel-costigliola.github.io/assertj/assertj-core-quick-start.html>
* Github project is available here: <https://github.com/joel-costigliola/assertj-core>
  + Joel mentioned in the interview that he would like to move the git hub repository under AssertJ organization.
* AssertJ Core 3.9.0 is the latest released on 2nd Jan 2018. However it has different versions to support different versions of Java.
  + Assert 3.x requires Java 8 or higher
  + Assert 2.x requires Java 7 or higher
  + Assert 1.x required Java 6 or higher
* Support for [Andriod](http://joel-costigliola.github.io/assertj/assertj-core.html#android)
* Supports several modules listed below (with Start guides are available for each):
  + Guava
  + Joda Time
  + Database
  + Neo4J
  + Swing
* Support for several dependency/deployment tools
* Maven
* Gradle
* Ivy
* Groovy
* Scala as well
* Mocikto-core is used as mock library for creation of proxy objects during test execution
* It provides [Assertion Generator](http://joel-costigliola.github.io/assertj/assertj-assertions-generator.html) : it allows developers to create assertion for their own classes by “either through a simple [command line tool](http://joel-costigliola.github.io/assertj/assertj-assertions-generator.html#quickstart), a [Maven plugin](http://joel-costigliola.github.io/assertj/assertj-assertions-generator-maven-plugin.html) or a third-party [Gradle plugin](https://plugins.gradle.org/plugin/com.github.opengl-BOBO.assertjGen)”.

## System perspective

* Although the library is meant to be used for unit testing and write JUnit test cases. It a feature rich library.
* It has clearly documented guidelines for
  + Code of conduct
  + Contributor
  + Issue template
  + Pull request
* Active contributor list with contributor guidelines are available

<http://joel-costigliola.github.io/assertj/assertj-core.html#contributing>

<https://github.com/joel-costigliola/assertj-core/blob/master/CONTRIBUTING.md>

* The Code & Issues tracker is available on [Git Hub](https://github.com/joel-costigliola/assertj-core/issues?milestone=&q=is%3Aopen).

As on 24th December, it had:

* + 10 active pull requests
  + Regarding Issues - 59 open items - mostly defects, some of them are new feature, improvements, documentation items

Later I have applied “Read Code in One Hour” pattern to go through the source code at a high level.

Important insights are:

* The code is documented extensively.
* It is structured reasonably as a set of packages with most of the code available under ***core.a***pi package. There are a total of 21 packages.
* ***core.api*** package has abstract classes which act as a base for other concrete classes. Although the by names I sense some of the classes are related or might have duplicate code such as *Assertions.java* and *Java6Assersions.java* or Assertions.java and *AssertionsForClassTypes.java*
* ***core.api*** package is the biggest of the packages with 190+ java classes. Next section will further detail of analysis done to find some additional insights.
* JUnit test case set available is extensive with 2645 test classes under 119 packages with a total of 10K+ unit tests.
* In terms of ratio, for one class, there is an equivalent of ~4.6 test classes which is a good indicator.

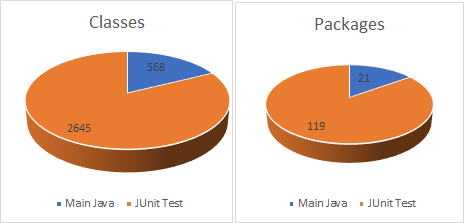


Figure 1 Comparison between main source and JUnit source w.r.t number of classes/packages

# Metrics and Analysis

To take the re-engineering to next level, I have followed “Mock Install”, “Study Exceptional Entities” and “Speculate about the design” patterns which led to some of the interesting insights.

As part of “Mock Install” step:

* AssertJ Core is a maven based project comes with required dependencies to build the project.
* From the pom.xml, the version of the *assertj-core* is *3.9.0* whereas *asssertj-parent-pom* is *2.1.6*
* Details of the dependencies are as below:

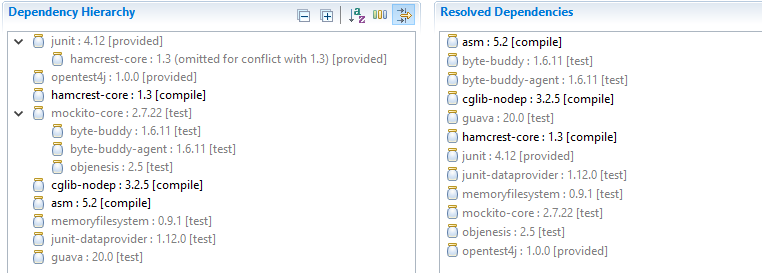


Figure 2 Project Dependencies in Eclipse

* The source code and project was imported in eclipse easily.
* It was successfully built without any build errors.
* Later I have executed JUnit test cases, there were total of *10861* test cases of which *91* were skipped with no failures and one of the tests in *Closeables\_closeQuietly\_Test* showed a warning.

As part of “Study Exceptional Entities” step:

* I have done static analysis of the source code in 2 ways i.e. Using Bash scripts and Analysing the .class files.

## Using Bash Scripts

* + Main motivation was to compute metrics such LOC, Comments and Vocabulary details of the code.
  + I have used the scripts available under [***analysisCode\src\main\scripts***] folder.

|  |
| --- |
| ***Code Change:*** I have found that *countComments.sh* has an issue does not consider all the comment lines. Especially for asserj-core system the comments are of the format  /\*  \*  \*…..  \* \*/  So I have committed a change in *countComments.sh*. |

* Generated outputs are saved into .csv files via bash shell and are available under ***[dataFiles\main\_code\_stats]*** and***[dataFiles\test\_code\_stats]***folder
* Each of the folder has *lines.csv*, *comments.csv* and *vocab.csv* file generated using *countLines.sh*, *countComments.sh* and *countVocab.sh* script files respectively.
* For main java source code, I have combined the outputs from lines.csv and comments.csv into the lines\_comments.xlsx.

Important insights:

* + There are total of 568 classes in main source code.
  + Percentage of comments against LOC

As mentioned in previous section, the code is documented extensively. In addition to other metrics. I have computed “% of comments” as below:

% of comments = number of lines in comments / total number of lines.

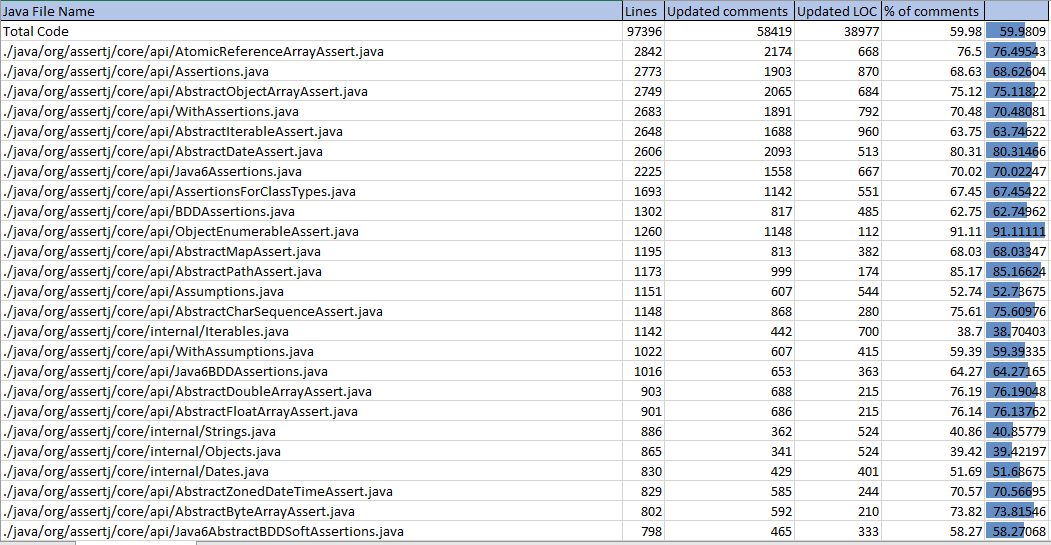


Figure 3 Lines, LOC, Comments, % of Comments

For a system as a whole, code documentation amounts to ~**60%** which is an interesting insight.



* + Of the total code, ***core.api*** package contains

189 classes, ~65% of total lines, 73% of total code documentation and ~50% of actual LOC.



* From system re-engineering perspective, the classes which have => 250 LOC but less than <= 25% documentation. There are about 6 classes which might need attention.

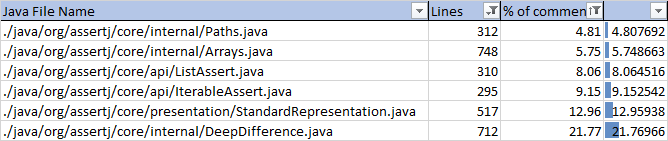


Figure 4 Candidate classes for Re-engineering from code documentation perspective

## Using the .class files

* + For additional metrics such as Cyclometric Complexity, Weighed Method count. I have used the ClassMetrics.java available under [analysisCode\src\main\java] folder.

|  |
| --- |
| ***Code Change:*** To make the analysis generic, I have extended the ClassMetrics.java to process the class files from a root directory. In addition to the extension, introduced new classes Metric.java, MethodInfo.Java and ClassInfo.java. |

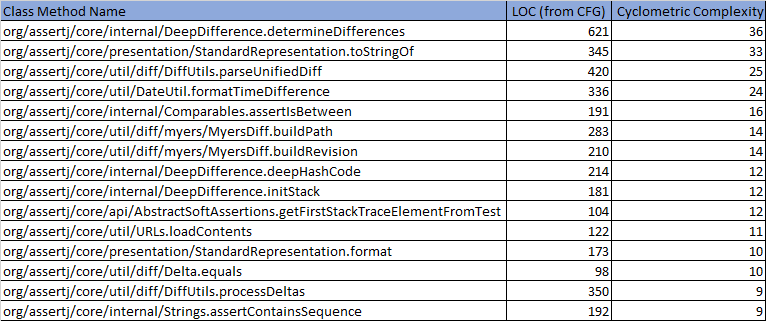
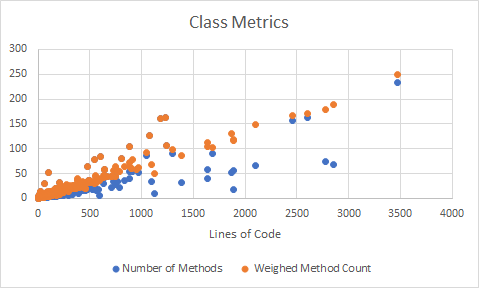
* + In addition to LOC and CC, computed method count and weighed method count of a class (sum of CC for all methods in a call).
  + The solution generates 2 outputs in CSV format
    - class\_metrics.csv – contains class level metrics
    - class\_method\_metrics.csv – contains class’s method metrics
* Important insights:
  + Cyclometric Complexity of all the methods is maximum 36 which implies the methods with CC above 10 might need refactoring with
  + CC – 11-20 - More complex, moderate risk
  + CC – 21-50 - Complex, high-risk program 

Figure 5 Candidate methods (CC > 10) ordered by cyclometric complexity

* + Maximum LOC (from CFG) of all the methods is 621 nodes which implies that methods of above LOC could be candidates for refactoring or re-engineering.
  + Classes:

Below chart is generated from ClassMetrics.xlsx (based on class\_metrics.csv)



From the above chart, its evident to find some of the candidates for re-engineering.

* + - about ~30 classes have more than 50 methods
    - about ~20 classes have above 1000 LOC

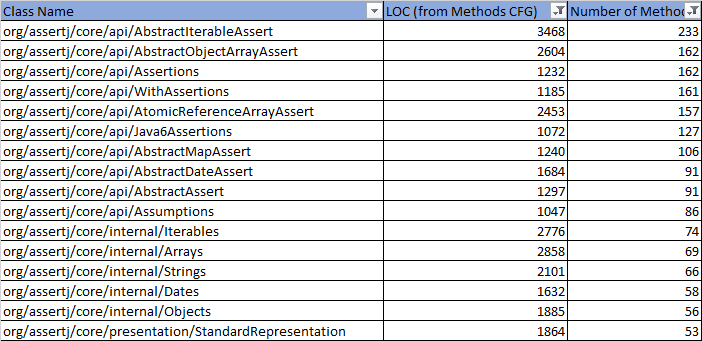


Figure 6 Candidate classes for Re-engineering based on number of methods

## Junit – Code Coverage

* + In the previous section, although it was conveyed that test case bed is extensive. But I wanted to capture the code coverage by JUnit test cases.
  + To do this analysis, I have used EclEmma plugin to calculate the code coverage. The results are available under [dataFiles\JUnitCodeCoverage] folder.
  + Overall coverage is 84% and Main source code coverage is 91%

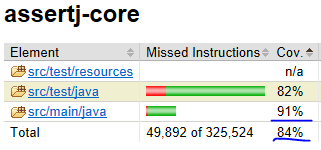


Figure 7 Code coverage by JUnit tests

* + If we look at the ***core.api*** package coverage, order by number of methods missed w.r.t coverage below list of classes will be candidates for re-engineering.

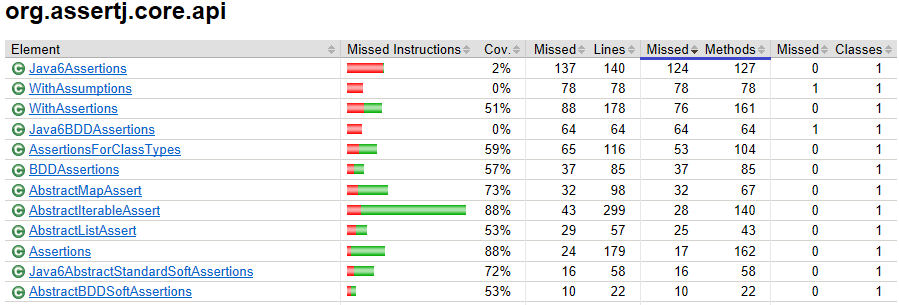


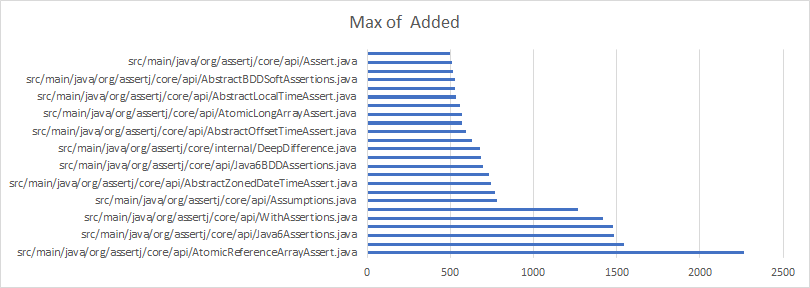
Figure 8 Candidate classes for re-engineering based on code coverage

## Repository log mining

Another aspect of an evolving system is the code churn. As part of “Learn from the Past” pattern, I have done repository mining of assert-core git repo. I have considered latest 1000 commits for analysis. The outputs generated are available under [**dataFiles\RepositoryMiningOutputs**] folder.

|  |
| --- |
| **Code Change:** Updated *rep-mining-churn.sh* as needed for assertj-core repo to check out revision 9d45e93. |

Insights: Below charts generated from diffs generated as part of code churn, clearly show either additions or deletion of code are mostly done in classes part of ***core.api*** package with few of them in *util* and *internal* packages.



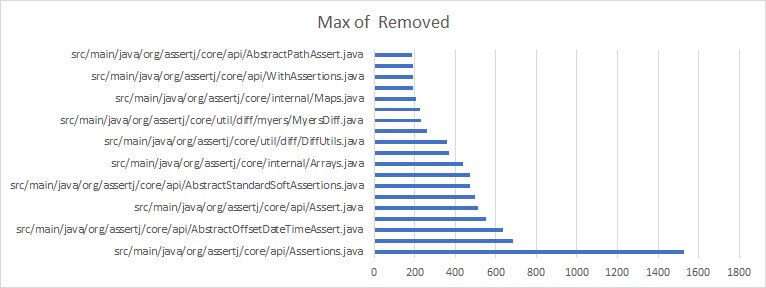


Figure 9 Repository mining - code churn

## Trace Analysis

Considering asserj-core is a library of assertions which is less intensive than an UI library such as jfreechart. I have used AspectJ weaving to analyse the traces for tests executed for ***core.api*** package.

|  |
| --- |
| **Code Change**:   * Aspect file ***AssertJAspect.aj*** added under [*analysisCode\src\main\java\aspects*]. This is referred from coursework 2 of System Re-engineering. Updated for AssertJ core library. * Added ***CoreApiTestSuite*** and ***CoreApiTestClass*** under [org.assertj.core.api] test class package. These classes are added to generate a JUnit executable jar. The generated jar *CoreApiTest.jar* is used along with Aspect.jar for trace recoding. |

Insights:

Initially the trace was generated with a filter as “org.assertj.core”, the output file was ~330MB which was difficult to analyse. To narrow down the trace, filter was changed to “org.assertj.core.api” which generated ~3MB output (available under dataFiles\TraceAnalysis). Based on the TraceAnalysis i.e. Number of occurrences of a class, the top 15 classes include:

* WritableAssertionInfo
* Assertions
* AssetionsForClassTYpes
* ListAssert
* Abstract\*Assert classes.

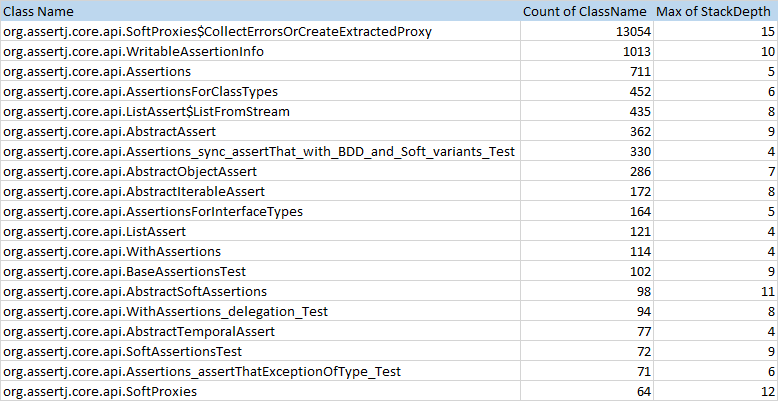


Figure 10 Trace Analysis outputs

With regards to Stack Depth, considering the trace is filtered for ***core.api*** the maximum call depth of 15 is a good indicator and not an alarming one.

# The Big Picture

As part of this section, I have applied “Learn from the Past” “Speculate about Design”, “Compare Code Mechanically”, “Visualize Code as Dot plots” patterns with help of different visualization and analysis techniques to understand the system in detail in turn find out candidates for re-engineering.

It covers insights from below items:

* Code City visualizing software
* File comparison to find out the duplication
* Release to Release comparison analysis
* Class diagram

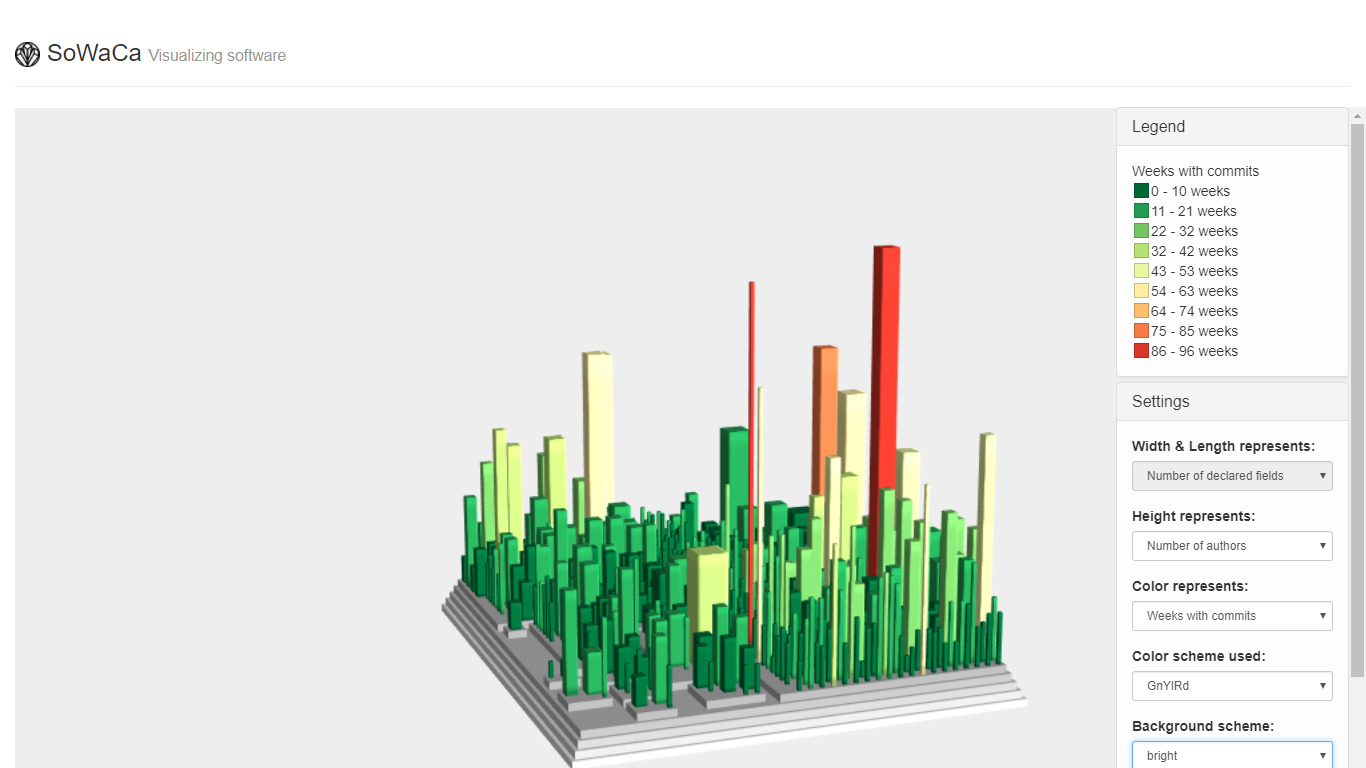
## Visualisation - Code City

To understand more about the system and to apply “Learn from the Past”, I have taken the code from original asserj-core repository @ 9d45e93. Used the Code City visualization software from Eclipse to generate additional insights based on the repository log from git.

Below is the visualization of main source code java with color representing the number of weeks with commits and height representing number of authors. Notably below entities stand-out:

* AbstractIterableAssert.java – 88 weeks with commits & 32 authors
* Assertions.java – 96 weeks with commits & 30 authors
* AbstractObjectArrayAssert.java - 79 weeks with commits & 23 authors

Other entities are relative to them.

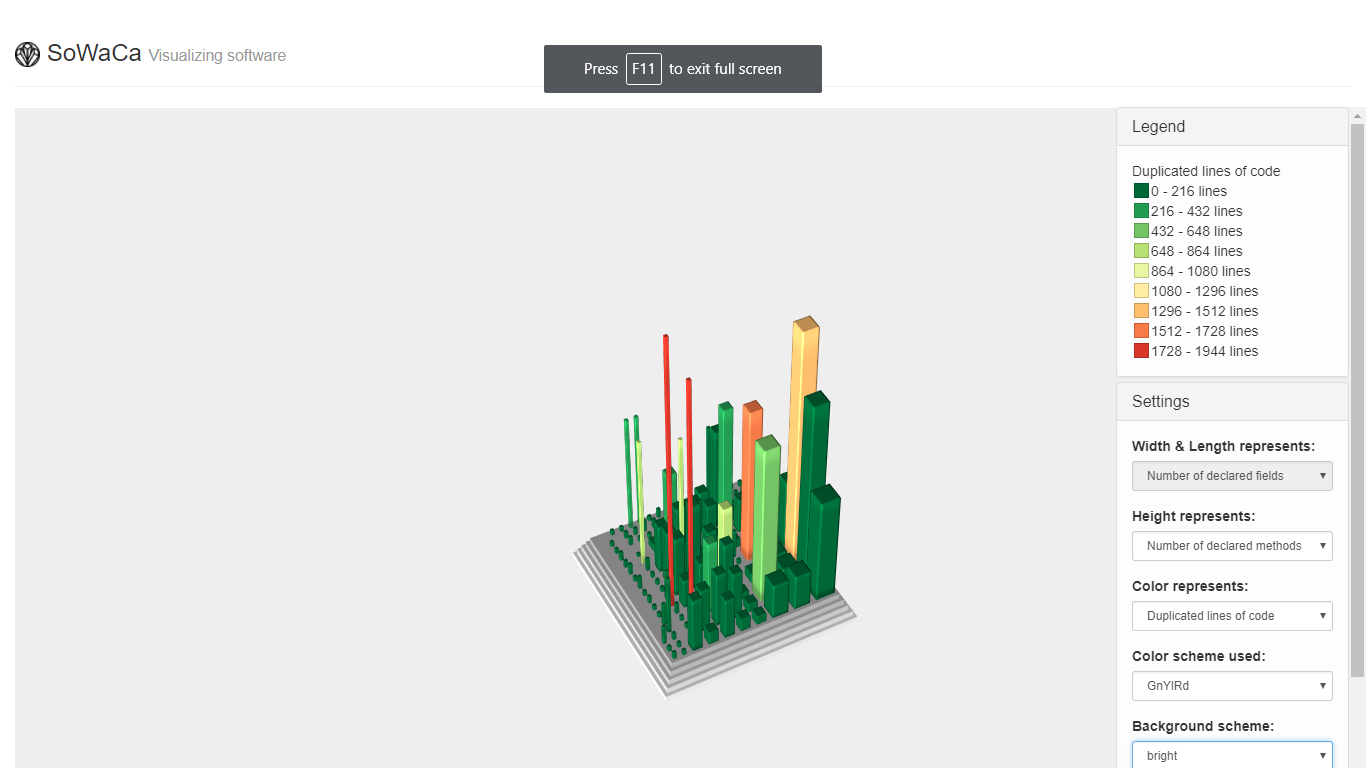


AbstractObjectArrayAssert

Assertions

AbstractIIterableAseert

Below is the visualization of only core.api package with color showing duplicated lines of code and height showing number of declared methods. Interestingly the same entities standout similar to above visualization.



Assertions & WithAssertions

AbstractIIterableAseert

## File Comparision for code duplication

As part “Visualize Code as Dot plots”, I have used the file comparison code available under *analysisCode* and R-Studio to generate the dot plots using fileComparisonScripts.R script.

To make the comparison structured, I have considered the code base at package level instead of whole code together.

* Package Core.API

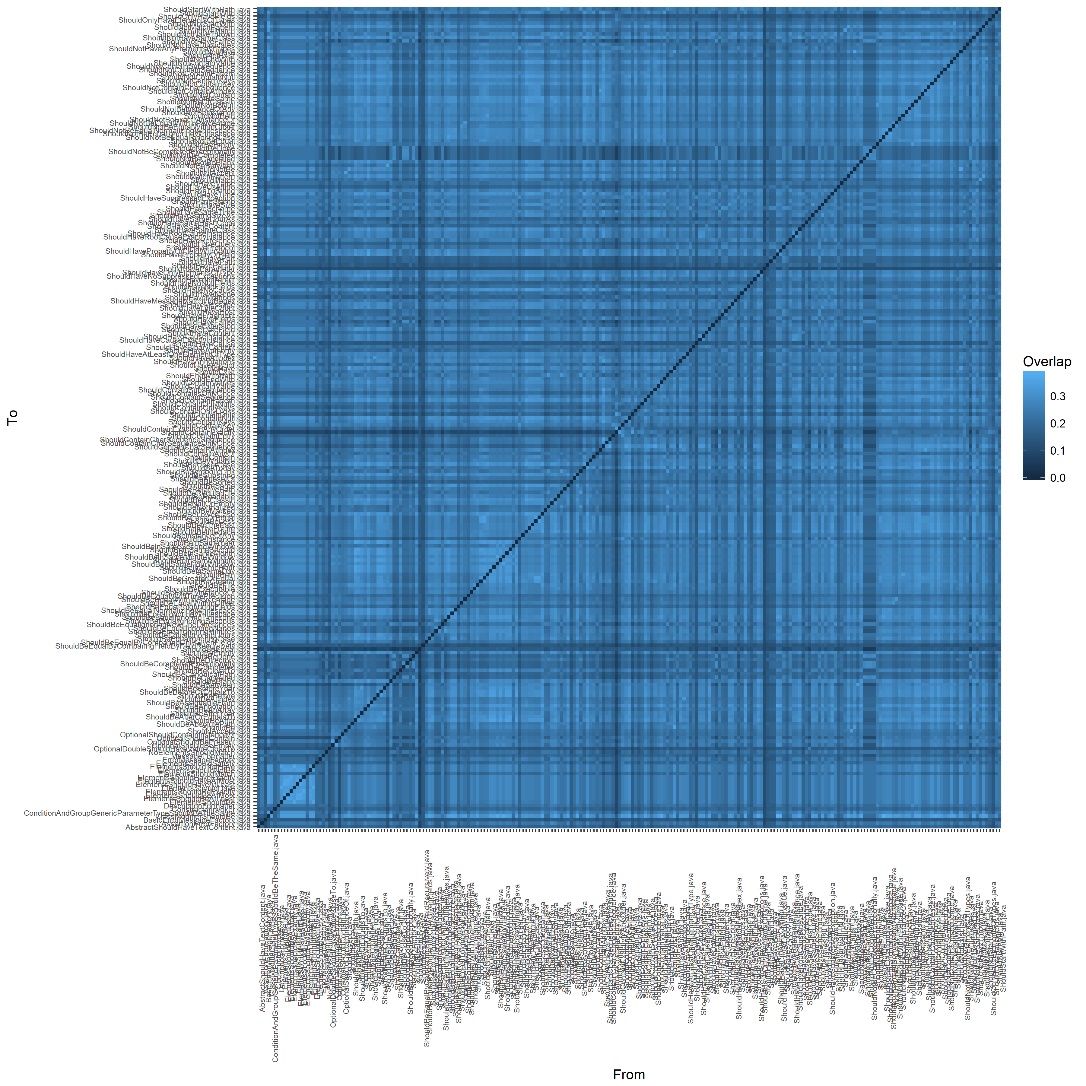
From the below dot plot it is evident that there is duplication of code from bright blue spots. One of the duplication is the documentation of the code which is repeated in a uniform structure. But there are couple of areas which stand-out in the dot plot. These areas are zoomed out with areas 1 and 2 below:

|  |  |
| --- | --- |
| s | **1** |
| **2** | In the 1st zoomed-in plot,  the classes CharArrayAssert.java, CharacterAssert.java, ByteAssert.java. ByteArrayAssert.java, BooleanAsserrt.java BooleanArrayAssert.java are related classes and have duplicate code.  In addition BDDAssertions and BDDSoftAssertions also related & have duplicate code.  In the 2nd zoomed-in plot, the classes of the pattern Atomic\*.java are related and have duplicate code thus showing bright blue spots. |

* Package Error

The dot plot for this package shows lot of duplication. Since most of the classes in this package have similar classes names and functions within each of the classes follow uniform naming convention.

There are group of classes which are particularly related are:

* Element\*.java
* ShoudlBe\*.java
* ShouldContain\*.java
* ShouldHave\*.java etc.
* 

For other significant packages, the duplication arises due to classes inherited from same base class whereas package util has fairly distinct code since each of the classes have separate responsibility.

|  |  |
| --- | --- |
| Package Representation: | * Package Util |

## Class Diagram

For Class Diagram solution, I have extended the Class Metric solution.

|  |
| --- |
| **Code Change**: In addition to the existing metrics, I have added *AttributeCount*, *FieldCount* and computed metric for each class *DepthOfInheritance*.  *DepthOfInheritance* is computed as number of levels of inheritance until the Superclass is java.object class.    In terms of class generation in the dot file, important notations are:   * Number of methods in a class is represented by width of the box shape * LOC in a class is represented by height of the box shape * DepthOfIneritance is represented by text font size of the class, so higher the depth larger the font size |

Based on the updated solution, the output generated is available as EnhancedClassMetrics.csv under [*dataFiles*] folder. The dot file and pdf generated using dot utility are available under [*dataFiles\classDiagramSolution*] folder.

As per “Speculate about Design” and “Study the Exceptional Entities” patterns, from the class diagram generated

* The classes look highly related showing cohesion
* AbstractAssert stands out as an important class as being the superclass of most of the Assertion classes.

In the second image, core.api.internal.Arrays shown as part of Arrays assertion classes. In the third image, the 4 classes below due to their LOC and number of methods stand out in the class diagram as GOD classes.

- org.assertj.core.api.AbstractObjectArrayAssert

- org.assertj.core.api.AtomicReferenceArrayAssert

- org.assertj.core.api.AbstractIterableAssert

- org.assertj.core.internal.Iterables

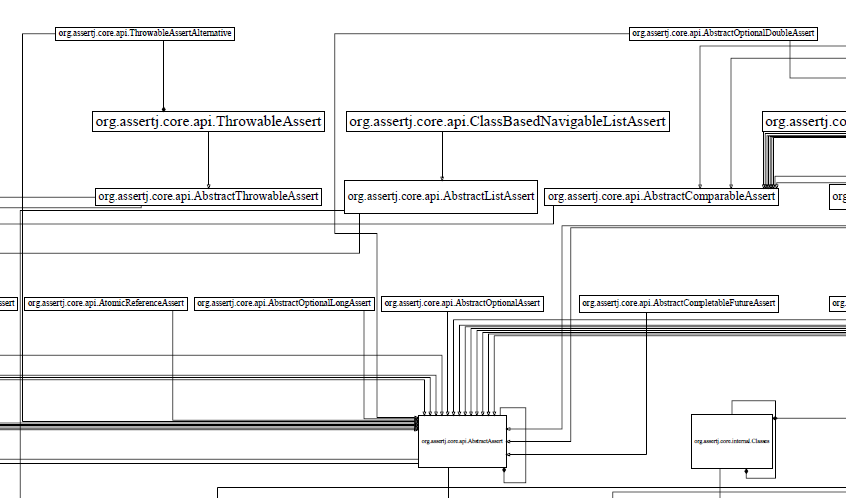


Figure 11 AbstractAssert superclass

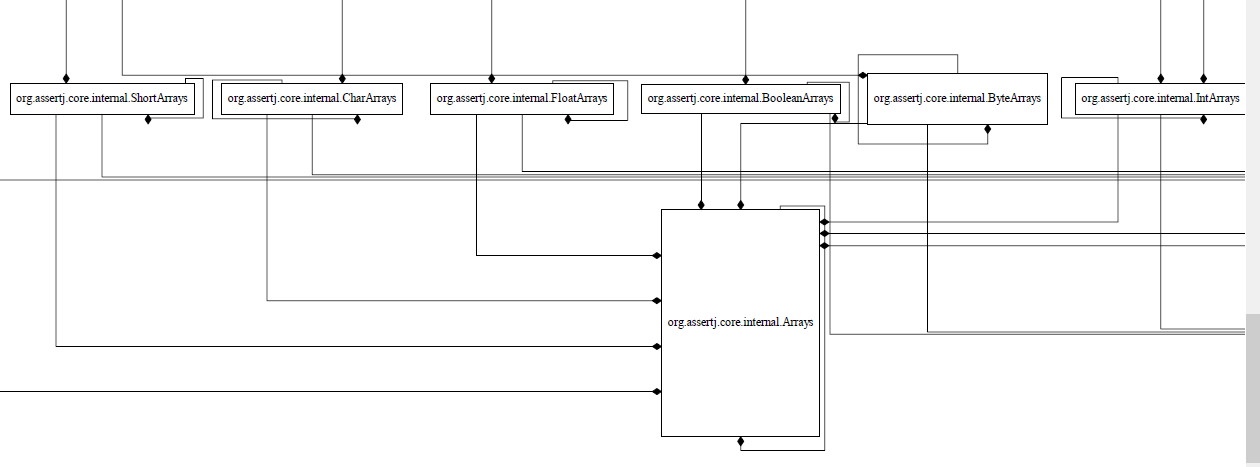


Figure 12 core.internal.Arrays - composed within arrays assertion classes

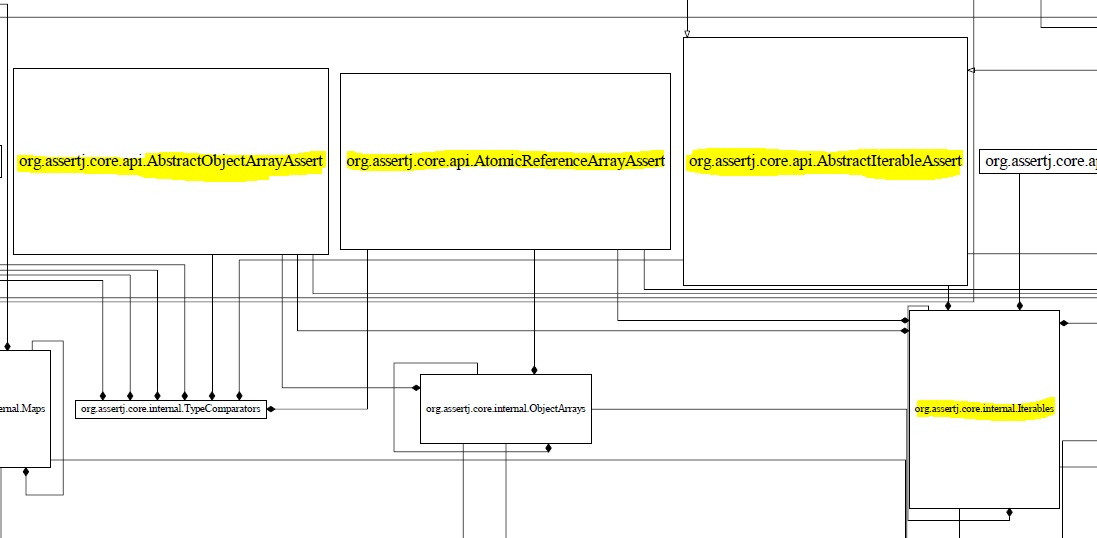


Figure 13 GOD classes

## Code churn from release to release

AssertJ core as a library had many releases since its inception. There has been total of 37 releases from release 1.0.0 in March 2013 to the latest release 3.9.0 dated 2nd Jan 2018. One of the important aspect is the code churn from release to release.

Incidentally I found an interesting [link](https://github.com/joel-costigliola/assertj-core/issues/1061) on git hub of the system, which discussed API tracking from release to release. More information can be found at this link: <https://abi-laboratory.pro/java/tracker/timeline/assertj-core/>

Image [[2](#_AssertJ:_API_changes)] in the next page shows the code churn from release 3.1.0 till 3.9.0. The image shown the code churn in terms of methods added and removed. However I was interested in code churn from release 2.8.0 (supports Java 7) to release 3.9.0 (supports Java 8). I have considered the code from this tag: <https://github.com/joel-costigliola/assertj-core/releases/tag/assertj-core-2.8.0> and generated class metrics and class method metrics as detailed in [Using the .class files](#_Using_the_.class) section.

Noticibly between Release 2.8.0 and Release 3.9.0 –

* There were 82 new classes added and about 1300 new methods
* About 20 K lines of new source code added (approx. 60% being code documentation)
* Majority changes happened in classes in ***core.api*** package which in-line with earlier analysis.

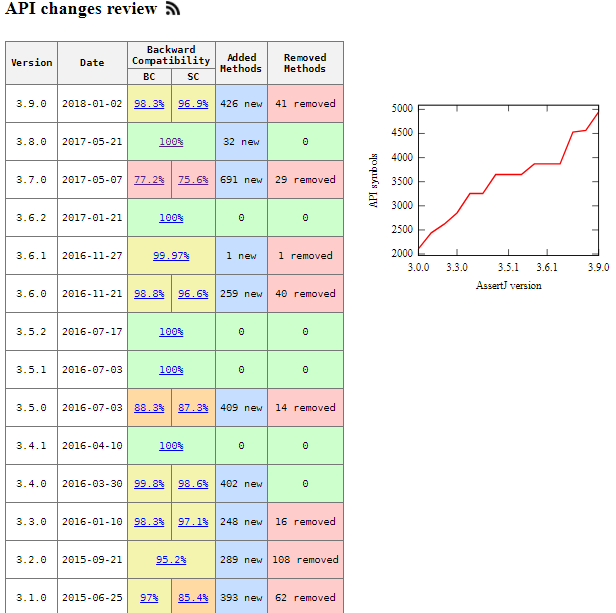


Figure 14 API changes viewer [[2](#_AssertJ:_API_changes)]

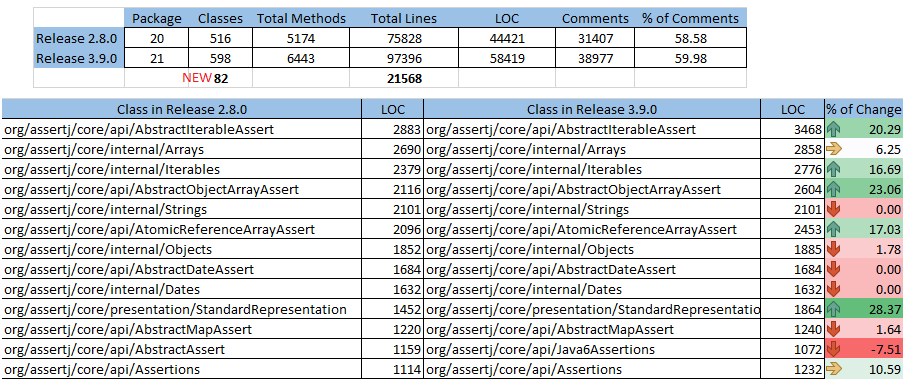


Figure 15 Code churn from Release 2.8.0 to Release 3.9.0

# Reengineering

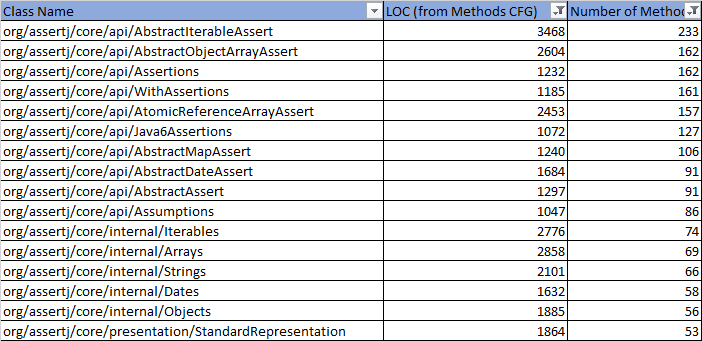
This section reviews the AssertJ core system with respect to re-engineering opportunities and listing the candidates of it. Based on the analyses done in previous section, highlights are:

* AssertJ Core although started by Joel Costigliola, its community-driven project.
* The project has a strong coding and contributor guidelines. With a total of 119 authors, 37 releases (since 2013) and 2000 commits the assertion library is feature rich with a detailed documentation.
* The main source code and test code are structured into several packages.

The list of re-engineering opportunities is as below:

* GOD classes

Due to extensive code documentation, some of the classes appear larger in size. The actual LOC is at most 960. The classes which have methods [more than 50](#_Using_the_.class) are direct for re-engineering.



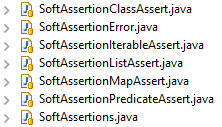
* Duplicate Code

As part of visual analysis, it’s evident that there is a duplication of code between several related classes. We can employ automatic comparison or use “Compare Code Mechanically” to refactor out duplicate code.

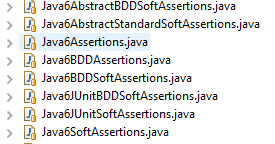
* Move behaviour close to data

During the static analysis of the code and skim through found instances where a code or a function could be moved into respective class.

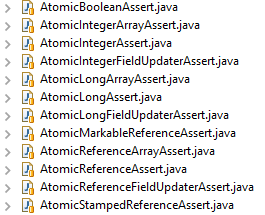
* ***core.api*** package is the largest with 180+ classes within it. The package can be further broken down in more structured way. The proposed changes are as follows:
  + SoftAssertions is a crucial feature of AssertJ core which allows to capture output of all the assertions of testing an object instead of returning after first assert failed. So the below classes and related can be considered to be part of a sub-package ***core.api.soft***



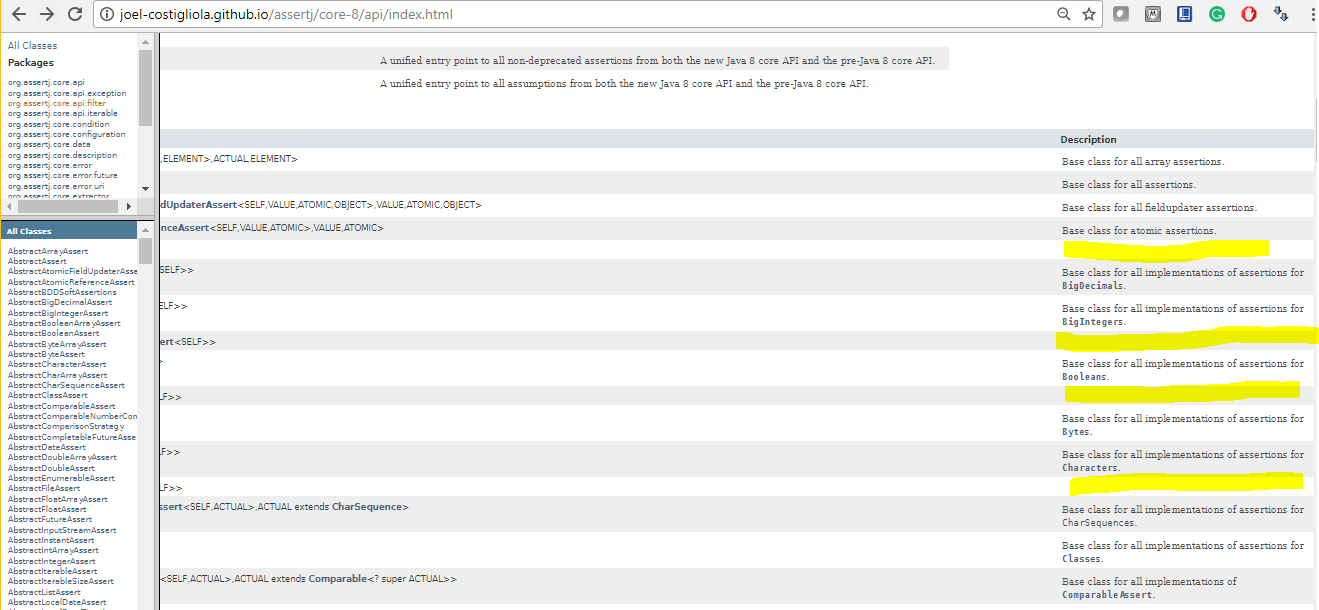
* + There are assertion classes compatible with Andriod. These classes can also be considered to be part of a sub-package ***core.api.andriod***



* + There are assertion classes to support Atomic objects. These classes can also be considered to be part of a sub-package ***core.api.atomic***

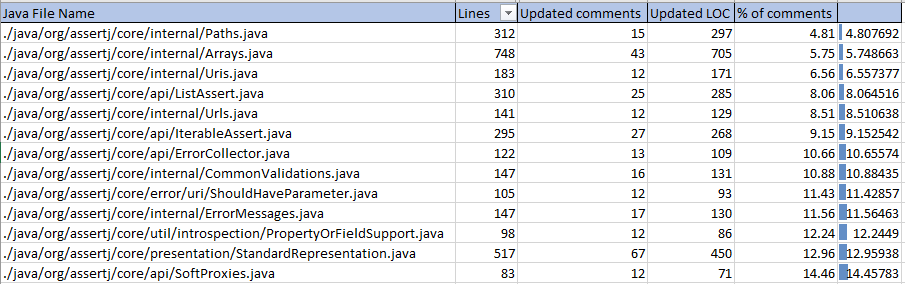


* + If needed to restructure bit more, all the date and time can be considered to be part of sub-package ***core.api.time***
  + As a re-engineering note, we can communicate the developers about the new packages of these assertions so that migration can be done.
* Code documentation has been extensive. But there are lot of classes which have class documentation header or description missing. This can be considered as part of re-engineering, it is crucial for a new developer to understand what they are meant for. This is evident from the java doc page as well refer: <http://joel-costigliola.github.io/assertj/core-8/api/index.html> .

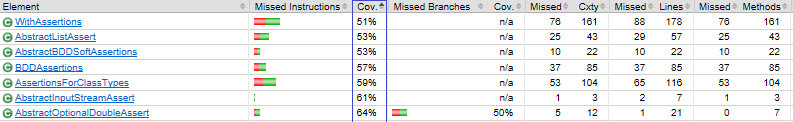


In addition, there are some classes which lack code documentation. These classes can also be considered for review and add documentation where required.

For example: Arrays.java lacks code documentation for most of the functions.



* Unit test case set is extensive, but there are classes which have code coverage less than 65%. These can be considered for re-engineering to increase JUnit test base.



* Architectural enhancements

During the static analysis of the code and skim through found instances where a group of classes structured hierarchically. Additional analysis could find out instances where the overall structure of the API can be improved.

# Re-engineered Sections

From the identified candidates for re-engineering, the process of re-engineering is done in an iterative way. As per “Tests! Your Life Insurance” pattern, every change done to the system while re-engineering should be followed by extending the test base and make sure there are no failures.

Since Assertj-core has an extensive unit test base. It’s easier to refactor and test the modified code. However, the code base is large, it’s easier to start with smaller changes than get into bigger ones. i.e. to follow “Refactor to Understand” pattern.

As part of the engineering process, I have used Extract Class, Extract Method, Encapsulate Field, Move method refactor techniques.

Some of the re-engineering completed as part of the course-work are detailed below.

1. Split GOD classes and remove duplication of code: I have considered below classes for Split and remove duplicate code.

core\api\Assertions.javacore\api\AssertionsForClassTypes.javacore\api\AssertionsForClassTypes.java

Have 900+ lines of common code. All the code is set of utility functions which are duplicated in each class, hence it is been moved to new AssertionBase class. This splits the GOD classes and reduces code duplication.

|  |
| --- |
| **Code Change:**  a596425 :  **Re-engineering - AssertionsBase introduced**  - Split GOD classes - Duplication of code in Assertions.java,  AssertionsForClassTypes.java, Java6Assertions.java  8e5b87a  **Re-engineering - AssertionsBase update**  remove duplication of code from:  core\api\Assertions.java  core\api\AssertionsForClassTypes.java  core\api\AssertionsForClassTypes.java |

1. Added a contract for ArrayAssert so that all sub-classes implement required operations

|  |
| --- |
| **Code Change:** 7dd971b  **Re-engineering - design update - updated the contract of an ArrayAssert class** |

1. Moved SoftAssertions related assertion classes to a new package core.api.soft

|  |
| --- |
| **Code Change:** 715b3eb |

1. Moved Atomic\* related assertion classes to a new package core.api.atomic

|  |
| --- |
| **Code Change:** d3a0fc5 |

1. Moved Date and Time-related assertion classes to a new package core.api.time

|  |
| --- |
| **Code Change:** 763753f |

1. Moved Android related assertion classes to a new package core.api.android

|  |
| --- |
| **Code Change:** ad67d0c |

1. Moved method close to data for couple of instances

|  |
| --- |
| **Code Change:** 01b00fd  **Re-engineering - Move code close to data**  From [DescriptionFormatter.java](https://github.com/LeicesterSoftEng/assignment-3-shashidarette/commit/01b00fd98284c44ab9a0a482d261e821897cac5d#diff-ea1c987380d7d7cf5a10dbec62ce7be5) to [Description.java](https://github.com/LeicesterSoftEng/assignment-3-shashidarette/commit/01b00fd98284c44ab9a0a482d261e821897cac5d#diff-8b5ad2cc65c409609bc573f262416d5d)  From [StandardRepresentation.java](https://github.com/LeicesterSoftEng/assignment-3-shashidarette/commit/01b00fd98284c44ab9a0a482d261e821897cac5d#diff-b2bfaaab61b821ddcc5bc669f73b8cf1) to [PredicateDescription.java](https://github.com/LeicesterSoftEng/assignment-3-shashidarette/commit/01b00fd98284c44ab9a0a482d261e821897cac5d#diff-4775a8517438bf3d04e4ecec81a82aea) |

1. Re-engineering - Moved method closed to data - 12a2821

- Moved Date parse related functions from AbstractDateAssert to DateUtil class

9. Re-engineering - Code documentation & Move to a package - d24c46d

# Conclusion

Analysis of AssertJ core from system re-engineering perspective has given very valuable insights. Each of the object-oriented re-engineering pattern needed application of different analysis tool/techniques such as AspectJ, Bash Scripts, R Studio & R scripts, Code City, Ecl Emma, Excel, ASM and more importantly Java programming language.

Although some of the insights found during analysis felt simple to considered for re-engineering. During actual refactoring of the concerned areas seemed difficult so candidates considered earlier had to left to be as it. Overall the analysis findings enabled to re-engineer the system and in turn improve it.

# References

### Costigliola, J.

Costigliola, J. (2018). AssertJ / Fluent assertions for java. [online] Joel-costigliola.github.io. Available at: http://joel-costigliola.github.io/assertj/assertj-core.html [Accessed 23 Dec. 2017].

### AssertJ: API changes review

Abi-laboratory.pro. (2018). AssertJ: API changes review . [online] Available at: https://abi-laboratory.pro/java/tracker/timeline/assertj-core/ [Accessed 3 Jan. 2018].