RE-ENGINEERING ASSERTJ

AlHasan AlSarraj

aa1013 - 179040072

Table of Contents

Initial Exploration

In the first contact with a system we tried to understand what is the system, what does the system do, how the system works and how is it structured.

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.
AssertJ core provides assertions for JDK standard types and can be used with JUnit or TestNG.
AssertJ is a Maven-buildable.

We investigated the system to gain a broad understanding of its **function**, **key components** and **architectural features**. We used some re-engineering patterns and we will explain how we applied these patterns and what we learnt about the system using these patterns.

A. Read all the Code in One Hour

The intent is to assess **the state of the system** "first impression of the **quality** of the source code" and describes the **functionality of the system**.

We used a generic checklist to review the kind of code we are dealing with.

Are we able to understand the code easily?	Yes
Is the code written following the coding standards/guidelines?	Yes
Do classes have a copyright header?	Yes
Do the classes do what the name of the class claims that they'll do?	Yes
Are classes too big? Do classes have too many methods?	Yes (not all)
Are methods too long?	Yes (not all)
Do methods have too many parameters?	No
Do classes and methods commented well?	Yes
Does the code compile without errors and run without exceptions?	Yes
Does the code compile without warnings?	No

Can we unit test / debug the code easily?	Yes
Do the methods do what the name of the method claims that they'll do?	Yes
Can we get an understanding of the desired behaviour just by doing quick scans through unit and acceptance tests?	Yes
Is this code introducing any new dependencies between classes/components/modules?	Yes
Does the system includes many test units and test cases?	Yes
Does the system includes interfaces and abstract classes and methods?	Yes

We can see that the system has a **Readable Code**, **Meaningful Identifiers** and **names**, Followed the **Style Guide**, tested using **test cases**, Described by **Comments**.

On the other hand, we can see that the system has some **Big Classes**, **Long Methods**, **Dependencies** between components and **compile with warning**.

From the previous checklist, we can:

- Filter out what seems important:
  - Split up God Classes
  - Eliminate Navigation code
- Learn the developers vocabulary
- Functional tests and unit tests convey important information about the functionality of a software system. They can help to verify whether the system is functioning as expected, which is very imported during reengineering

We can notice that the reengineering seems feasible, and while the system main function is to assert, we supposed that the most important classes and packages are where the system should assert and catch the exceptions and errors:

"org.assertj.core.api"	"org/assertj/core/api/Assertions"
"org.assertj.core.data"	"org/assertj/core/api/AbstractIterableAssert"
"org.assertj.core.error"	"org/assertj/core/api/AssertionsForClassTypes"
"org.assertj.core.api.exception"	

We need to investigate the test cases further using the tools of dynamic analysis.

B. Skim the Documentation

The intent is to Assess the relevance of the documentation “identify those parts of the documentation that might be helpful”

The list below summarizes those aspects of the system that seem interesting for our reengineering project. We matched this list against the documentation. meanwhile we made a crude assessment of how up to date the documentation seems:

Installation and build	In Starting Guide page, we can find how we can install the system using Maven.
System Features	In Features - Tutorial pages, we can find how to use the system and what features the system have.
Design Documentation	Not Exist.
Structure Documentation	Table of contents Not Exist.
Up to Date	Version Number: AssertJ fluent assertions **3.8.0** API “Up to Date”
System Functionality	In Overview and other pages we can find some Command Descriptions
Important Terms	In Index page, we can find most important terms

We can see that the documentation will be **useful**, user manual “Tutorial” and installation guide is **clear** and **up-to date**, system functionality **described** using **commands descriptions** and we have the **most important terms**.

C. Do a Mock Installation

The intent is to Check whether we have the necessary artefacts available by installing the system and recompiling the code.

We **Installed**, **Built** and **Recompiled** the system using Maven, no build or compile errors and exceptions, but warnings are existing.
The system is using JRE System Library [JavaSE-1.8] and Referenced Libraries. No need for a database and network toolkits.

D. Study the Exceptional Entities

The intent is to Identify potential design problems by collecting measurements and studying the exceptional values.

We implemented this pattern using tools “Static and Dynamic AnalysisTools” in next chapters, we looked for obvious anomalies “Big Classes, Big Methods and Standout metrics”, and we diagnosed potential problems to focus on.

E. Chat with Maintainers

The intent is to learn about the historical and political context of AssertJ project.

“Since we can’t talk to the maintainers, we implemented this pattern by:

- skimming over any issues on SourceForge and GitHub
- skimming over recent commit logs
- Using GitHub features (we implemented this pattern using code churn tools in next chapters)

And we found the following issues:

- **Write a complete assertions guide**
  “Users are often not aware of useful assertions, a guide showing best practice assertions would help a lot.**”**
- **Add extracting java 8 examples**

From this pattern, we found that the users asking for a complete guide and examples, and there are no open issues for the current version.

The version is 3.8.0 so they have 3 Major changes (API incompatible) and 8 Minor changes (functionality) and now there are no bugs for current version.

Metrics and Analysis

Our motivation in this section is to know what is the quality of the system, where are the obvious problems, what are the exceptional entities and How disciplined was the development process for the system.

A. Calculate LOC and Weighted Method Count for each class, and calculate LOC and Cyclomatic Complexity for each Method.

We used ClassMetric class from CodeAnalysisToolkit, and we amended the code, added some other code to get the results we need and we deleted the code we didn't use.

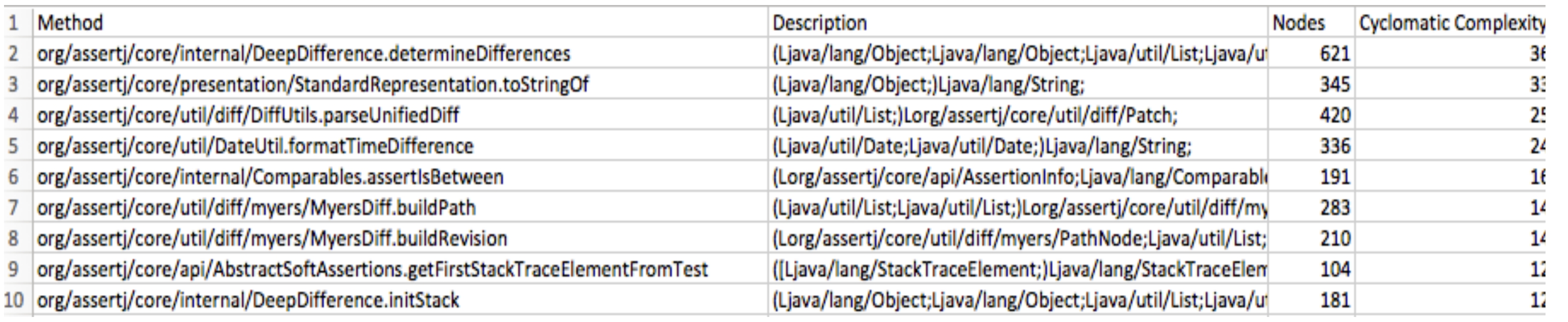
We read the classes from the path "assertj-core/target/classes/".
We stored the classes metrics in
"dataFiles/2A_ClassesAndMethodsMetrics/ClassesMetrics.csv".
We stored each method metrics in separate file in
"dataFiles/2A_ClassesAndMethodsMetrics/eachClassMethods/" folder.
Then we stored all method metrics in
"dataFiles/2A_ClassesAndMethodsMetrics/AllClassesMethods.csv"

We also used some Bash Scripts to get some metrics like LOC, Comments and Vocabulary count in each class and we stored the results in:
"analysisCode/2A_ClassesAndMethodsMetrics/CommandLineMetrics/"
"dataFiles/2A_ClassesAndMethodsMetrics/CommandLineMetrics/"

- **For the Methods**, we stored:
  - Method name
  - Description of the method (so we can know the different methods with the same names)
  - Number of Nodes in the CFG (LOC)
  - Cyclomatic Complexity of the methods

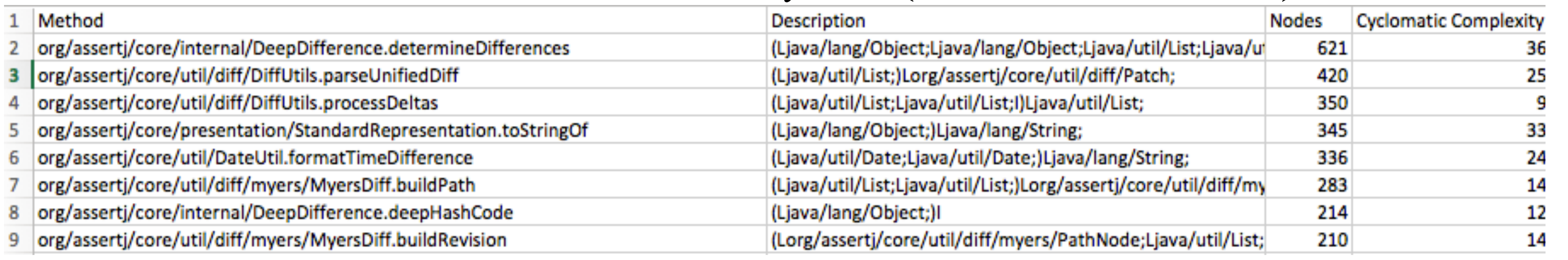
We used the same code in ClassMetrics (ASM4 and CFGExtractor) to get classes and methods nodes and names, and we just added some code to replace the "<init>" with the real name of the method when the method name is the same of the name of the class, and we separated the description of the method from the method name.

Some results for AllClassesMethods sorted by **Cyclomatic Complexity**:



1	Method	Description	Nodes	Cyclomatic Complexity
2	org/assertj/core/internal/DeepDifference.determineDifferences	(Ljava/lang/Object;Ljava/lang/Object;Ljava/util/List;Ljava/u	621	36
3	org/assertj/core/presentation/StandardRepresentation.toStringOf	(Ljava/lang/Object;)Ljava/lang/String;	345	33
4	org/assertj/core/util/diff/DiffUtils.parseUnifiedDiff	(Ljava/util/List;)Lorg/assertj/core/util/diff/Patch;	420	25
5	org/assertj/core/util/DateUtil.formatTimeDifference	(Ljava/util/Date;Ljava/util/Date;)Ljava/lang/String;	336	24
6	org/assertj/core/internal/Comparables.assertIsBetween	(Lorg/assertj/core/api/AssertionInfo;Ljava/lang/Comparabl	191	16
7	org/assertj/core/util/diff/myers/MyersDiff.buildPath	(Ljava/util/List;Ljava/util/List;)Lorg/assertj/core/util/diff/my	283	14
8	org/assertj/core/util/diff/myers/MyersDiff.buildRevision	(Lorg/assertj/core/util/diff/myers/PathNode;Ljava/util/List;	210	14
9	org/assertj/core/api/AbstractSoftAssertions.getFirstStackTraceElementFromTest	([Ljava/lang/StackTraceElement;)Ljava/lang/StackTraceElem	104	12
10	org/assertj/core/internal/DeepDifference.initStack	(Ljava/lang/Object;Ljava/lang/Object;Ljava/util/List;Ljava/u	181	12

Some results for ClassesMetrics sorted by **LOC (Nodes Number in CFG)**:

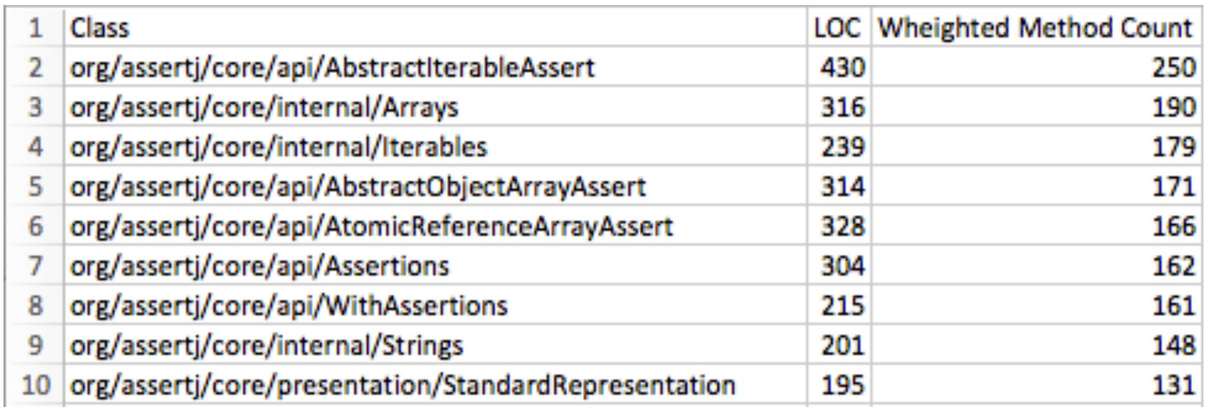


1	Method	Description	Nodes	Cyclomatic Complexity
2	org/assertj/core/internal/DeepDifference.determineDifferences	(Ljava/lang/Object;Ljava/lang/Object;Ljava/util/List;Ljava/u	621	36
3	org/assertj/core/util/diff/DiffUtils.parseUnifiedDiff	(Ljava/util/List;)Lorg/assertj/core/util/diff/Patch;	420	25
4	org/assertj/core/util/diff/DiffUtils.processDeltas	(Ljava/util/List;Ljava/util/List;I)Ljava/util/List;	350	9
5	org/assertj/core/presentation/StandardRepresentation.toStringOf	(Ljava/lang/Object;)Ljava/lang/String;	345	33
6	org/assertj/core/util/DateUtil.formatTimeDifference	(Ljava/util/Date;Ljava/util/Date;)Ljava/lang/String;	336	24
7	org/assertj/core/util/diff/myers/MyersDiff.buildPath	(Ljava/util/List;Ljava/util/List;)Lorg/assertj/core/util/diff/my	283	14
8	org/assertj/core/internal/DeepDifference.deepHashCode	(Ljava/lang/Object;)I	214	12
9	org/assertj/core/util/diff/myers/MyersDiff.buildRevision	(Lorg/assertj/core/util/diff/myers/PathNode;Ljava/util/List;	210	14

- **For the Classes**, we stored:
  - Class Name
  - LOC
  - Weighted Method Count

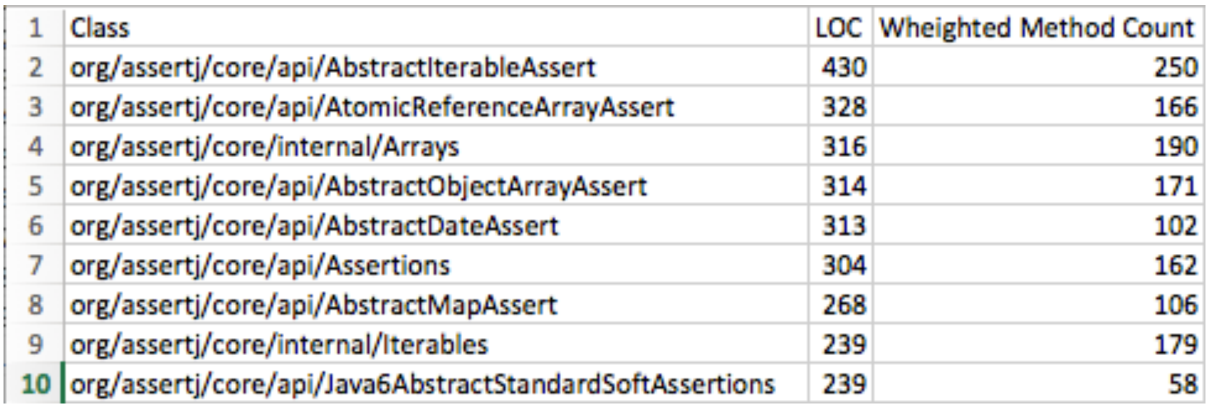
We used the same code in ClassMetrics to get the classes and extract each class name, then for each class we read the file of that class and count the number of lines using “count.readLine()”.
For the Weighted Method Count (which will offer details of how complex the classes are), we Sum of the complexity scores for each method (Method complexity is commonly Cyclomatic Complexity) “ClassCC”

Some results for ClassesMetrics sorted by **Weighted Method Count**:



1	Class	LOC	Wheighted Method Count
2	org/assertj/core/api/AbstractIterableAssert	430	250
3	org/assertj/core/internal/Arrays	316	190
4	org/assertj/core/internal/Iterables	239	179
5	org/assertj/core/api/AbstractObjectArrayAssert	314	171
6	org/assertj/core/api/AtomicReferenceArrayAssert	328	166
7	org/assertj/core/api/Assertions	304	162
8	org/assertj/core/api/WithAssertions	215	161
9	org/assertj/core/internal/Strings	201	148
10	org/assertj/core/presentation/StandardRepresentation	195	131

Some results for ClassesMetrics sorted by **LOC:**



1	Class	LOC	Wheighted Method Count
2	org/assertj/core/api/AbstractIterableAssert	430	250
3	org/assertj/core/api/AtomicReferenceArrayAssert	328	166
4	org/assertj/core/internal/Arrays	316	190
5	org/assertj/core/api/AbstractObjectArrayAssert	314	171
6	org/assertj/core/api/AbstractDateAssert	313	102
7	org/assertj/core/api/Assertions	304	162
8	org/assertj/core/api/AbstractMapAssert	268	106
9	org/assertj/core/internal/Iterables	239	179
10	org/assertj/core/api/Java6AbstractStandardSoftAssertions	239	58

B. For large Methods, employ slice-based metrics.

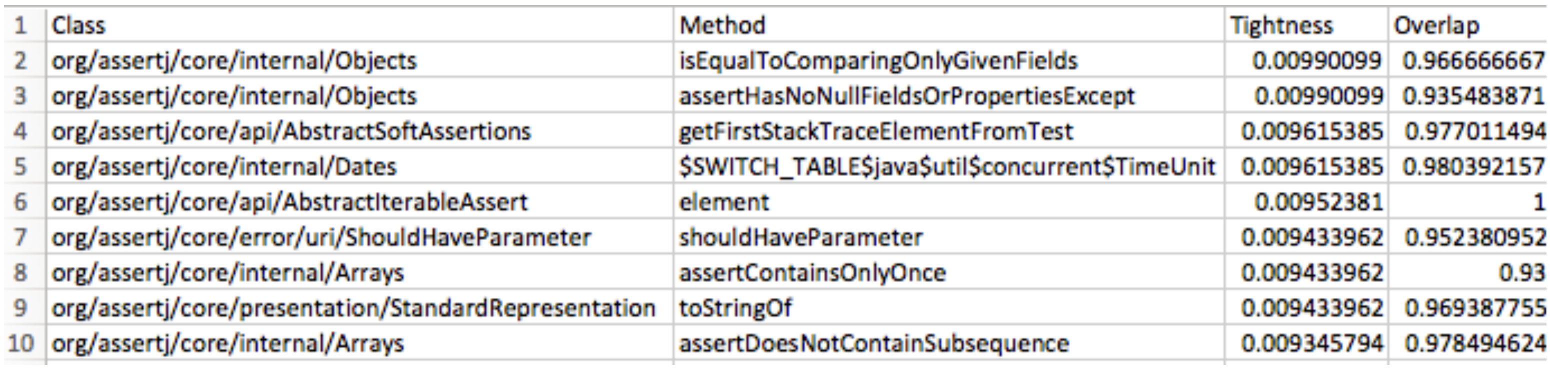
We computed Slice-Based Metrics "Tightness and Overlap" for large methods (With LOC > 100).
Tightness and Overlap will offer details of how cohesive the methods are.

We stored the result in "dataFiles/2B_SliceBasedMetrics/TightnessAndOverlap.csv" as:

- Class Name
- Method Name
- Tightness
- Overlap

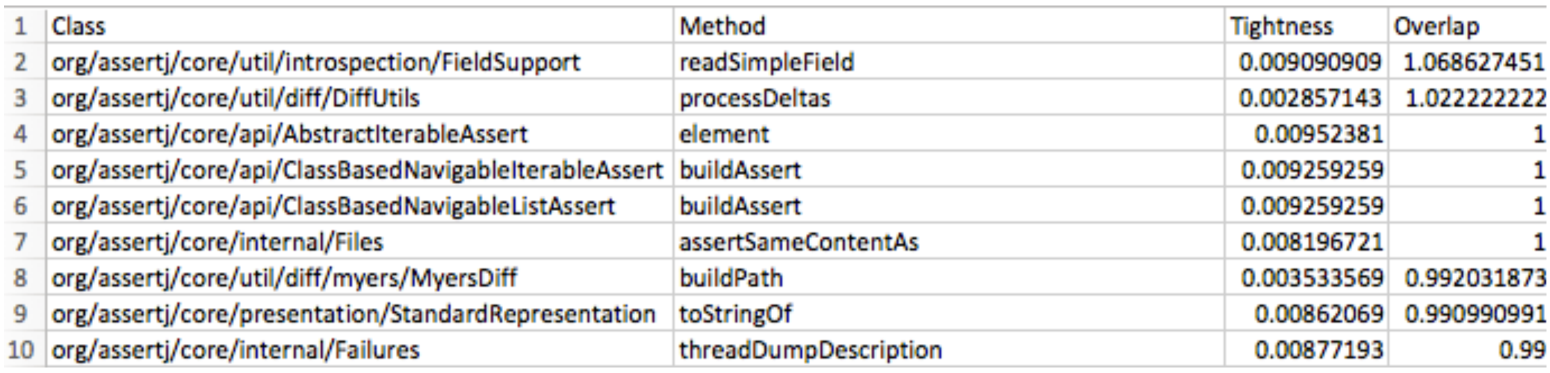
We used "computeTightness()" and "computeOverlap()" methods from ProgramDependenceGraph.java in ClassMetrics.java, but for computeTightness() we returned the size of inAllSlices collection of node divided by the number of nodes in the controlFlowGraph of the method.

Some results for TightnessAndOverlap.csv sorted by **Tightness:**



1	Class	Method	Tightness	Overlap
2	org/assertj/core/internal/Objects	isEqualToComparingOnlyGivenFields	0.00990099	0.966666667
3	org/assertj/core/internal/Objects	assertHasNoNullFieldsOrPropertiesExcept	0.00990099	0.935483871
4	org/assertj/core/api/AbstractSoftAssertions	getFirstStackTraceElementFromTest	0.009615385	0.977011494
5	org/assertj/core/internal/Dates	$SWITCH_TABLE$java$util$concurrent$TimeUnit	0.009615385	0.980392157
6	org/assertj/core/api/AbstractIterableAssert	element	0.00952381	1
7	org/assertj/core/error/uri/ShouldHaveParameter	shouldHaveParameter	0.009433962	0.952380952
8	org/assertj/core/internal/Arrays	assertContainsOnlyOnce	0.009433962	0.93
9	org/assertj/core/presentation/StandardRepresentation	toStringOf	0.009433962	0.969387755
10	org/assertj/core/internal/Arrays	assertDoesNotContainSubsequence	0.009345794	0.978494624

Some results for TightnessAndOverlap.csv sorted by **Overlap:**



1	Class	Method	Tightness	Overlap
2	org/assertj/core/util/introspection/FieldSupport	readSimpleField	0.009090909	1.068627451
3	org/assertj/core/util/diff/DiffUtils	processDeltas	0.002857143	1.022222222
4	org/assertj/core/api/AbstractIterableAssert	element	0.00952381	1
5	org/assertj/core/api/ClassBasedNavigableIterableAssert	buildAssert	0.009259259	1
6	org/assertj/core/api/ClassBasedNavigableListAssert	buildAssert	0.009259259	1
7	org/assertj/core/internal/Files	assertSameContentAs	0.008196721	1
8	org/assertj/core/util/diff/myers/MyersDiff	buildPath	0.003533569	0.992031873
9	org/assertj/core/presentation/StandardRepresentation	toStringOf	0.00862069	0.990990991
10	org/assertj/core/internal/Failures	threadDumpDescription	0.00877193	0.99

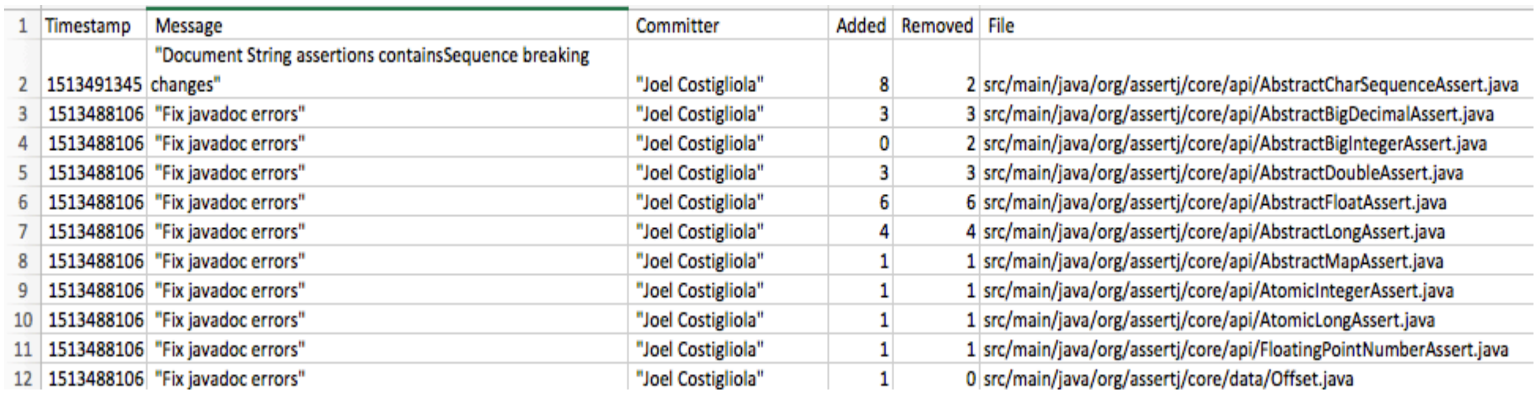
C. Analyses the Repository Log

We used "rep-mining-churn" in scripts folder to get all the changes on the repository.
Our intent is to know the total number of added or changed lines of code because code that changes often pre-release is more likely to contain faults post-release.

We stored the result in "dataFiles/2C_RepositoryLog/RepositoryLog.csv" as:

- Added
- Removed
- File

Some results for RepositoryLog.csv



1	Timestamp	Message	Committer	Added	Removed	File
2	1513491345	"Document String assertions containsSequence breaking changes"	"Joel Costigliola"	8	2	src/main/java/org/assertj/core/api/AbstractCharSequenceAssert.java
3	1513488106	"Fix javadoc errors"	"Joel Costigliola"	3	3	src/main/java/org/assertj/core/api/AbstractBigDecimalAssert.java
4	1513488106	"Fix javadoc errors"	"Joel Costigliola"	0	2	src/main/java/org/assertj/core/api/AbstractBigIntegerAssert.java
5	1513488106	"Fix javadoc errors"	"Joel Costigliola"	3	3	src/main/java/org/assertj/core/api/AbstractDoubleAssert.java
6	1513488106	"Fix javadoc errors"	"Joel Costigliola"	6	6	src/main/java/org/assertj/core/api/AbstractFloatAssert.java
7	1513488106	"Fix javadoc errors"	"Joel Costigliola"	4	4	src/main/java/org/assertj/core/api/AbstractLongAssert.java
8	1513488106	"Fix javadoc errors"	"Joel Costigliola"	1	1	src/main/java/org/assertj/core/api/AbstractMapAssert.java
9	1513488106	"Fix javadoc errors"	"Joel Costigliola"	1	1	src/main/java/org/assertj/core/api/AtomicIntegerAssert.java
10	1513488106	"Fix javadoc errors"	"Joel Costigliola"	1	1	src/main/java/org/assertj/core/api/AtomicLongAssert.java
11	1513488106	"Fix javadoc errors"	"Joel Costigliola"	1	1	src/main/java/org/assertj/core/api/FloatingPointNumberAssert.java
12	1513488106	"Fix javadoc errors"	"Joel Costigliola"	1	0	src/main/java/org/assertj/core/data/Offset.java

D. Use dynamic Analysis to highlight important classes

Our intent is to Monitor the system while it executes.

We choose some test cases from different folders. Then for each test case, we created Trace AspectJ file.

In all of our aspects, we defined a logger object. This uses the built-in java logging framework to record messages. In the constructor (Trace(){…}), we set this logger up. We added a "FileHandler" object to the logger, so that any messages can be written out to a file (in our case we defined logs files as csv files for each test case

Then we defined a "pointcut". This tells the aspect weaver which parts of the code we are interested in.

*pointcut traceMethods() : (**execution(* *(..))**&& **within(org.assertj..*)** && **!cflow(within(Trace))**);*

In our case:

- **execution(* *(..))** tells it to look at the execution of every method from every class, for any pattern of parameters.
- **within(org.assertj..*)** tells it to look into "org.assertj" packages only, because when we run the test using java agent we use **Junit, and we don't need to record the executed methods of JUnit packages.**
- **!cflow(within(Trace))** tells it to ignore anything that happens within the current Trace aspect, so we are only concentrating on the running program.

The code in the blocks starting with the keyword before() and with the keyword after() defines what should happen before and after any of the point cuts defined. The code has aspects to certain details about the current point in the execution (accessed through the object thisJoinPointStaticPart). This code takes the relevant information from that object, and sends it to the logger defined above. So, for every method, it records its details along with which method called it.

In before() block we added the code to get the class name, method name and parameters.
For the stack depth, we created a counter which will increase by 1 before every method execution, and will decrease by 1 when the method returns a value:

*after() **returning**: traceMethods(){ i--; }*

Then we compile the core of our dynamic analysis tools and producing the .jar files from .java files “analysisCode/2D_DynamicAnalysisTool/”

```
ajc -cp /Users/AlHasanAlSarraj/aspectj1.8/lib/aspectjrt.jar -outxml -
outjar Aspect.jar Trace.java
```

Then we run the tests and stored the results in csv log files: “dataFiles/2D_DynamicAnalysisImportantClasses/TracesLogFiles”

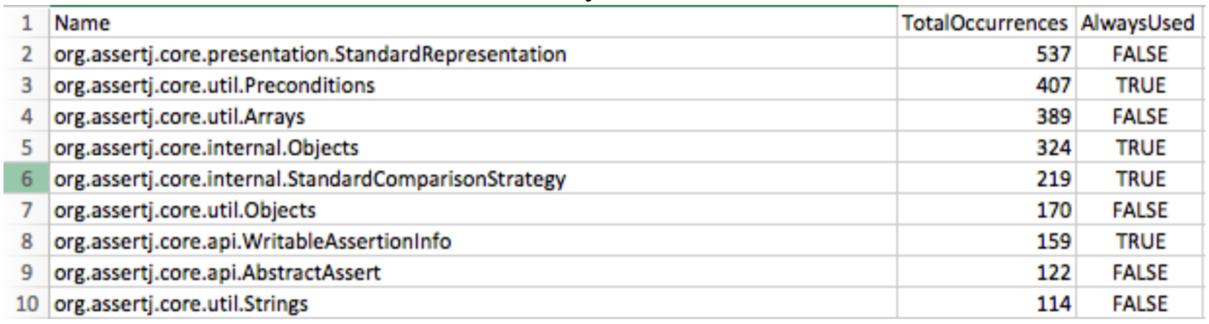
Then for each class we stored the total occurrence in the traces files and if the class is used in all traces files. And the same thing for the methods.
We stored the result in “dataFiles/2D_DynamicAnalysisImportantClasses/ClassesAndMethodsCSVFiles”

For the TotalOccurrence:
We created HashMap object to store each class with the total occur in all traces files.
We started with null HashMap, and for every class from every trace file we check if it exists in the HashMap, if not we add it and make the total occur = 1, else we increase the total occur by 1.

For AlwaysUsed:
We read the name of classes or methods from 5 traces file in sets, then we took the intersection of all these sets using reatinAll(). So, we got a set with names repeated in all trace files.

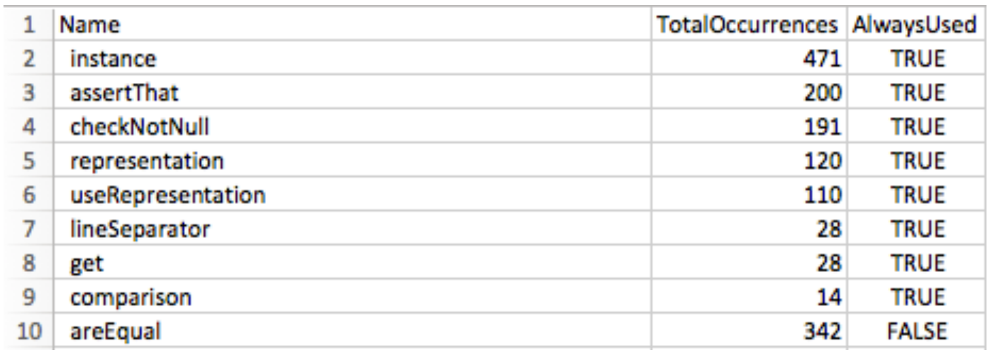
Then we produced execution phases for each test case.

Some results for classes.csv ordered by Total Occurrences:



Name	TotalOccurrences	AlwaysUsed
org.assertj.core.presentation.StandardRepresentation	537	FALSE
org.assertj.core.util.Preconditions	407	TRUE
org.assertj.core.util.Arrays	389	FALSE
org.assertj.core.internal.Objects	324	TRUE
org.assertj.core.internal.StandardComparisonStrategy	219	TRUE
org.assertj.core.util.Objects	170	FALSE
org.assertj.core.api.WritableAssertionInfo	159	TRUE
org.assertj.core.api.AbstractAssert	122	FALSE
org.assertj.core.util.Strings	114	FALSE

Some results for methods.csv ordered by Total Occurrences:



Name	TotalOccurrences	AlwaysUsed
instance	471	TRUE
assertThat	200	TRUE
checkNotNull	191	TRUE
representation	120	TRUE
useRepresentation	110	TRUE
lineSeparator	28	TRUE
get	28	TRUE
comparison	14	TRUE
areEqual	342	FALSE

Conclusion:

From the static and dynamic analysis, we found that the system has **593 Class**, **6326 Method, 20180 Line Of Code.**

We found that there are many big classes and big methods (Weakness Point), also there are some complex classes and complex methods with cyclomatic complexity between 21 – 50 which put the system in high risk (Weakness Point).

We found from slice-based methods that the elements are closely related within the module. (Strength Point)

We also found that some classes change a lot of times, and we know that Code that changes often pre-release is more likely to contain faults post-release.

Then we found the most important classes and methods in the system by using the dynamic analysis techniques. And they are the classes and methods which are using in each test case and each run of the system.

By mixing all the steps above we can find where the real weakness and real strength of the system, and we will explain that in the section 4 of this assignment.

The notable classes and methods:

Classes

- org/assertj/core/api/AbstractIterableAssert
- org/assertj/core/api/Assertions
- org/assertj/core/api/AbstractAssert
- org/assertj/core/api/Assertions
- org/assertj/core/api/AbstractIterableAssert
- org/assertj/core/api/AssertionsForClassTypes
-

Methods

- org/assertj/core/api/Assertions.assertThat
- org/assertj/core/api/AbstractAssert.describedAs
- org/assertj/core/api/AbstractAssert.isEqualTo

Note That ClassMetrics.java will do the most of second question so it will take a lot of time to be executed.

The Big Picture

Our intent is to provide visual explanations of the system, show the strength and weakness points and to give the big picture.

We created two versions of the visualizations, 1st one will visualize all the classes and methods in the system, the 2nd one will visualize only the important classes and methods.

From the Dynamic analysis techniques, we highlighted the most important classes and methods by filtering which are used by running the test cases of the system.
We created a java class to filter each of the following steps and take only the important classes from that file.
"analysisCode/src/main/java/ExtractImportantClasses"

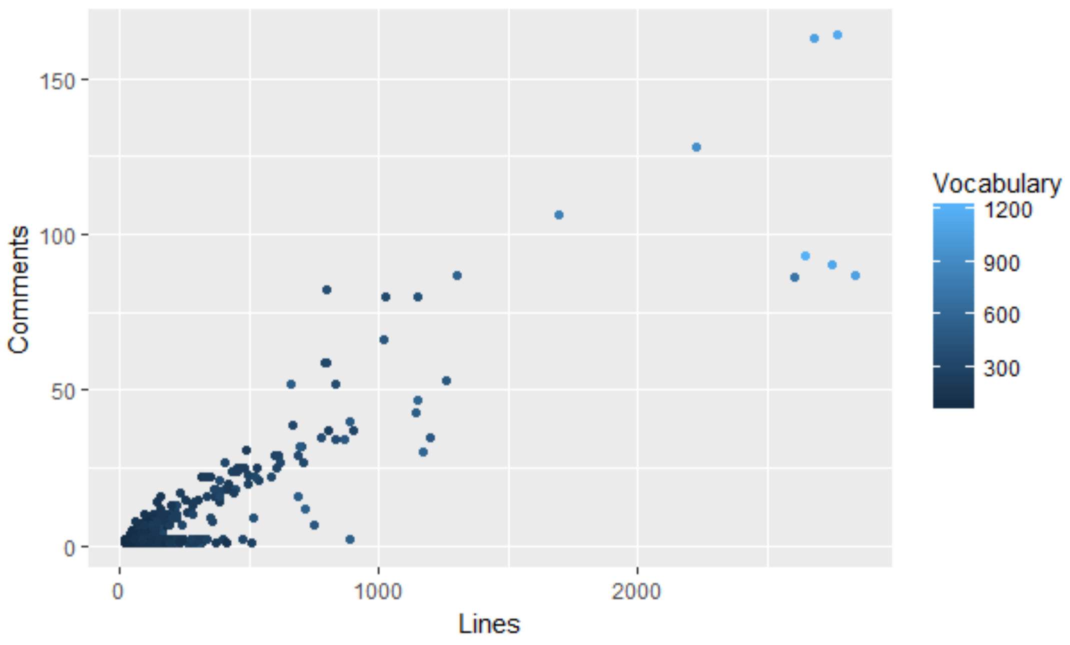
A. Visualize metrics and relationships between metrics.

"3A_MetricsAndRelations/"
We went through 4 steps to visualize metrics:

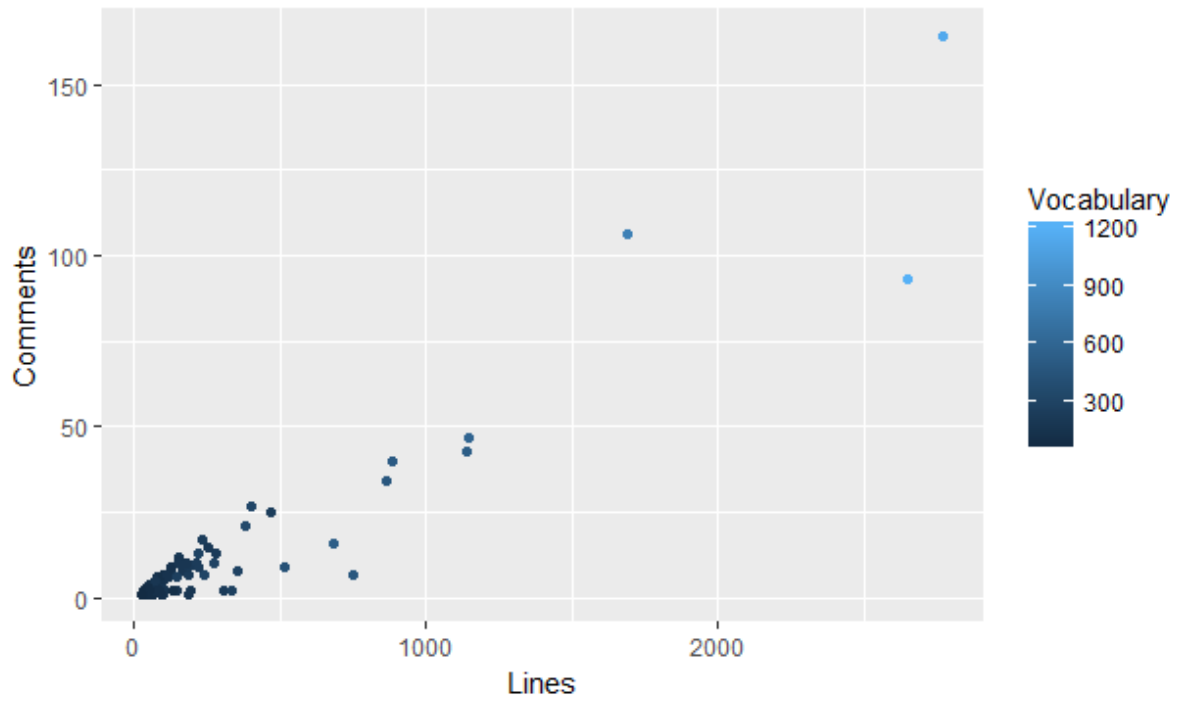
- Bash Commands
- Classes and Methods Metrics
- Execution Phases
- Repository Log (Code Churn)

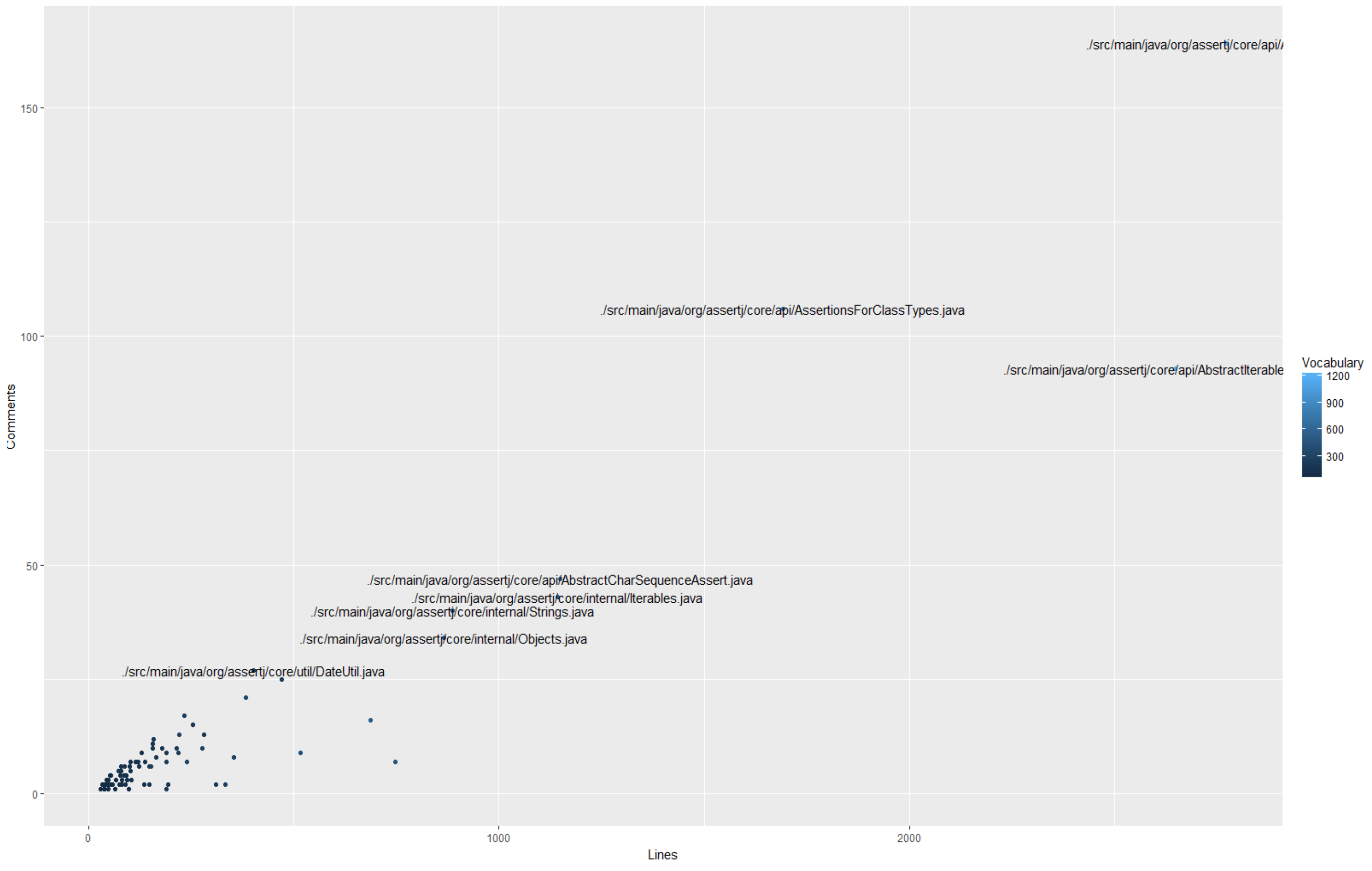
- **Bash Commands**

We calculated the lines, comments and vocabularies of each file, then we visualized them using R for all classes of the system
"dataFiles/3A_MetricsAndRelations/CommandLineMetrics/AllClasses"



Then we filter the classes in the files by checking if the class used in the test cases, and exclude the classes which are not important "dataFiles/3A_MetricsAndRelations/CommandLineMetrics/ImportantClasses"





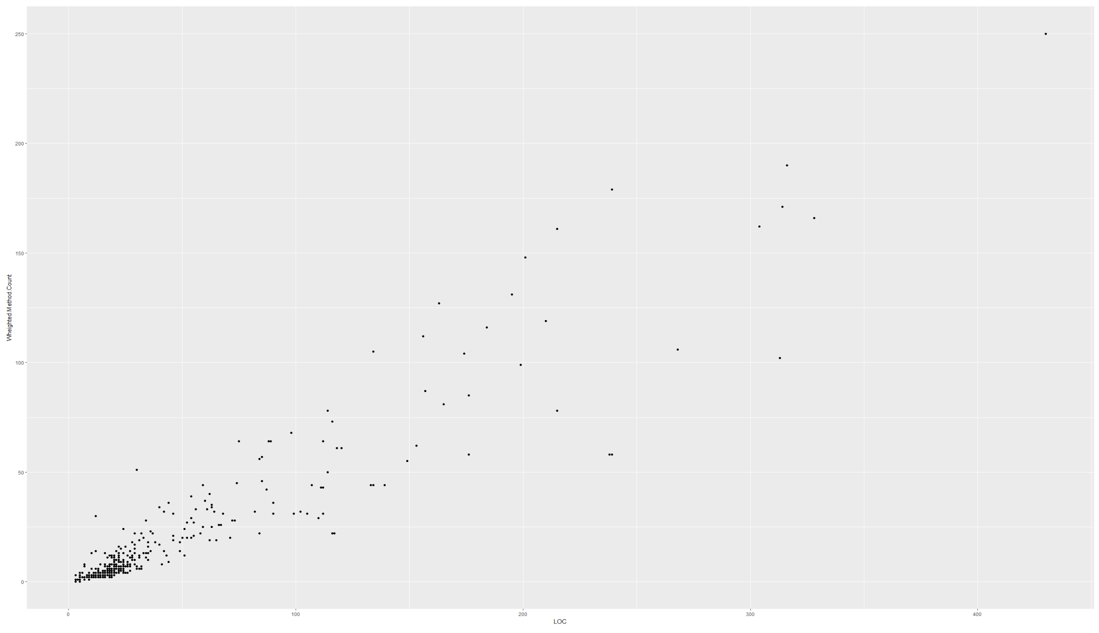
From bash command visualizations, we can find what are the big classes and we can understand if these classes are big because they include a big amount of comments or big amount of codes.

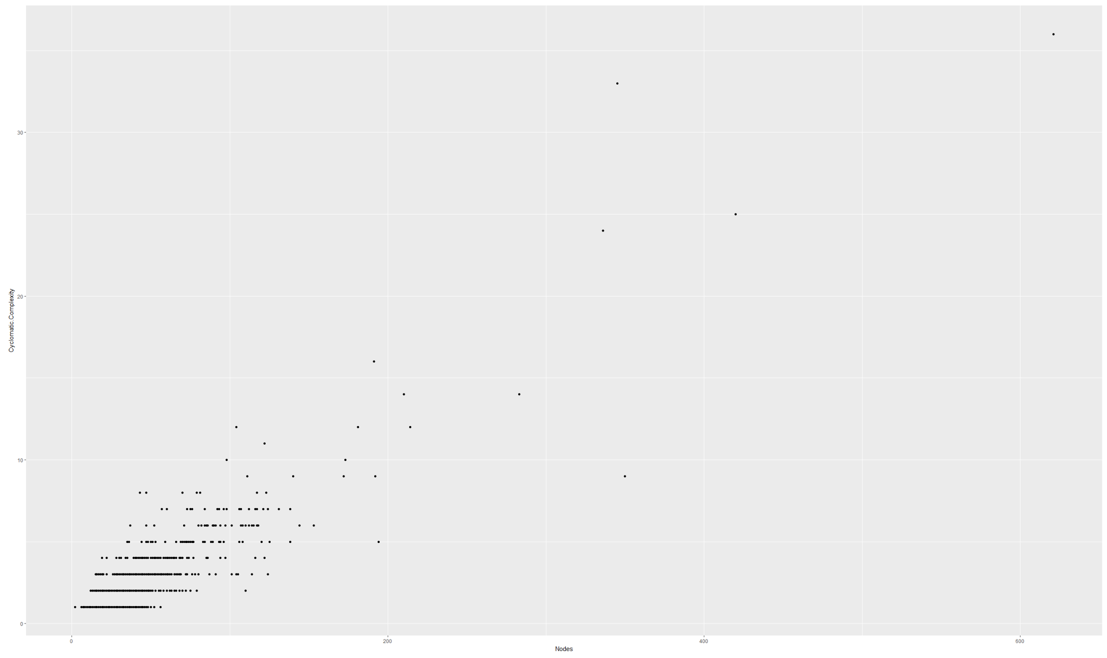
Also, we can know what classes have big amount of vocabularies and we can mark this classes as very important classes, and these important classes are:

- org/assertj/core/api/Assertions
- org/assertj/core/api/AbstractIterableAssert
- org/assertj/core/api/AssertionsForClassTypes

- **Classes and Methods Metrics**

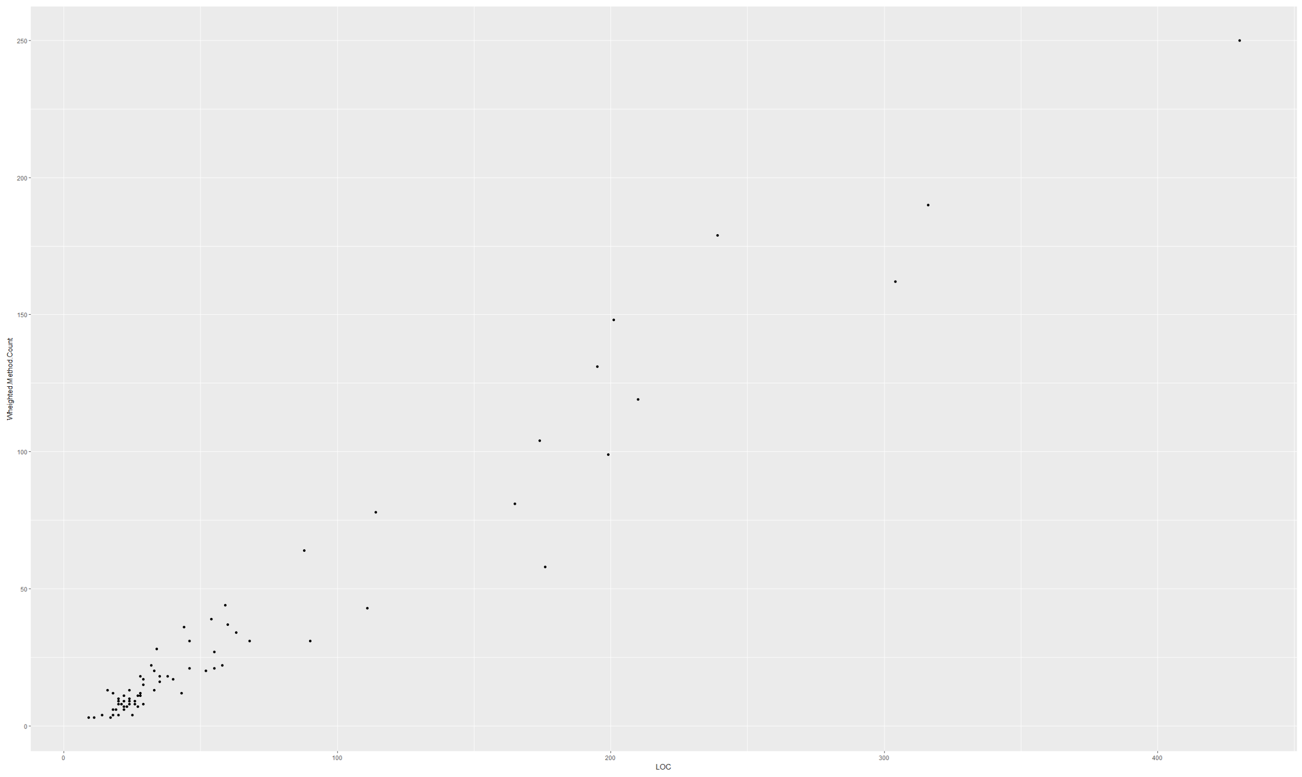
We visualized the metrics files from the static analysis section, and we got the following images for all classes and methods:
"dataFiles/3A_MetricsAndRelations/ClassesMetricsLOCAndWMC/AllClasses"

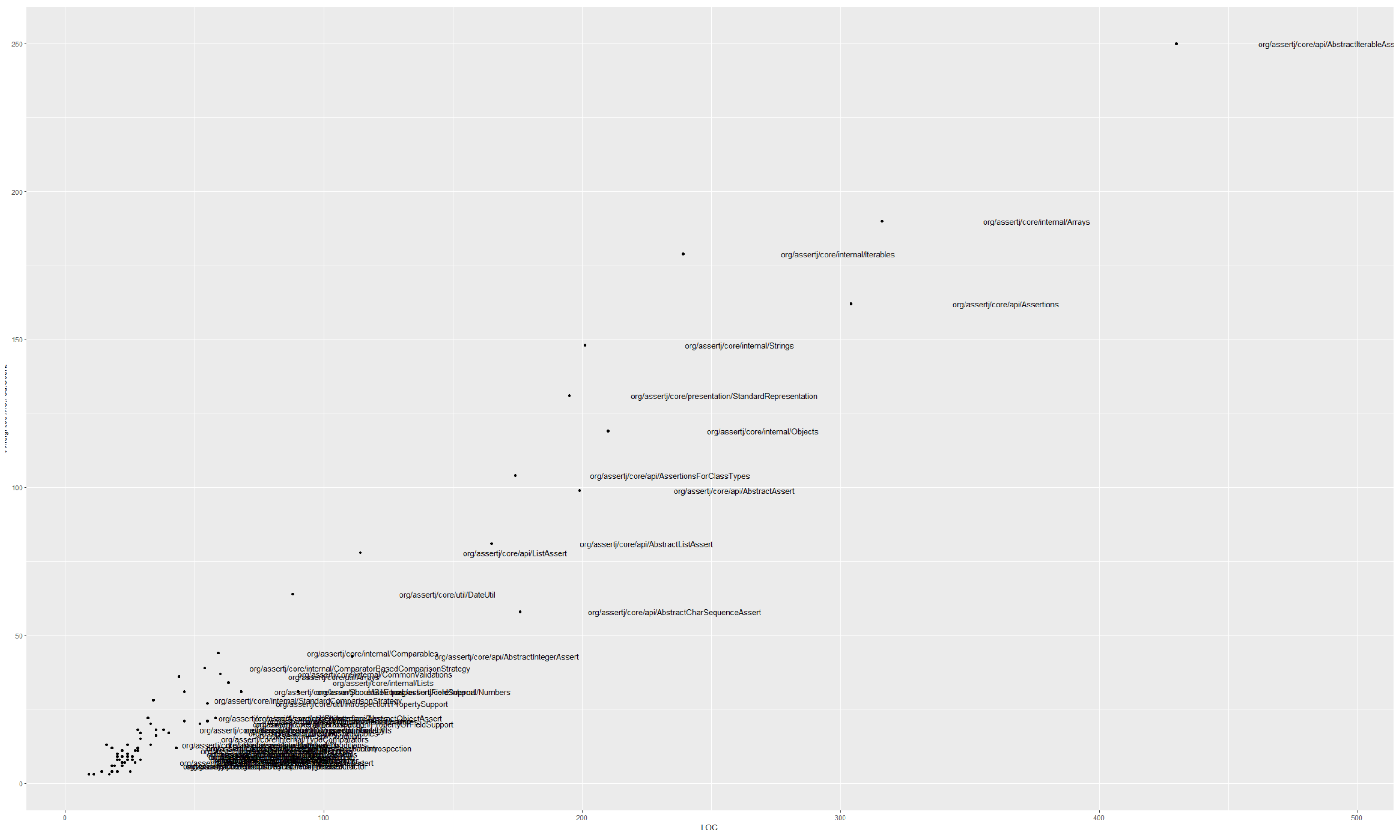




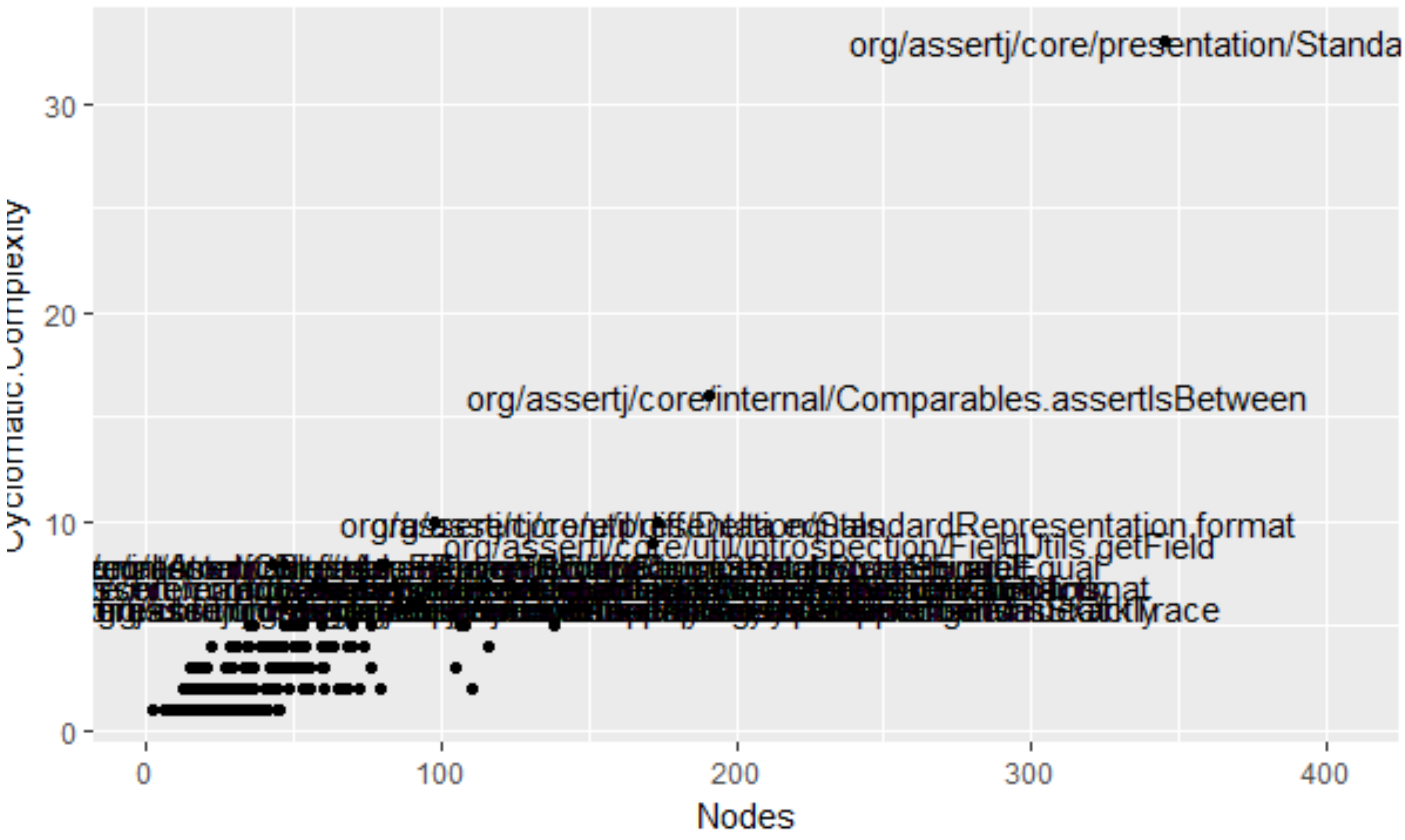
Then we filter the classes and methods in the same way, and we got only the important classes to get the following visualizations:
"dataFiles/3A_MetricsAndRelations/ClassesMetricsLOCAndWMC/ImportantClasses"

Classes:





Methods:



From the previous visualizations, we can find what the complex classes and methods are, and what classes and methods are big and complex in the same time.

We can highlight the complex and big classes and methods as very important classes to study and re-engineering, and these classes and methods are:

Classes

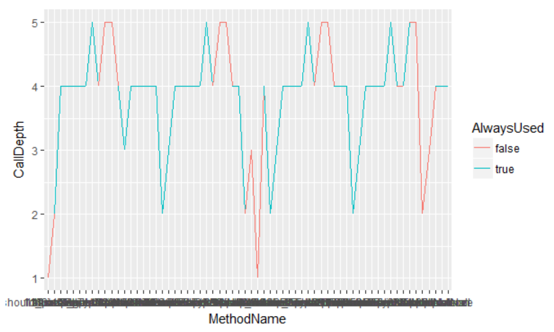
- org/assertj/core/api/AbstractIterableAssert
- org/assertj/core/api/Assertions
- org/assertj/core/api/AssertionsForClassTypes

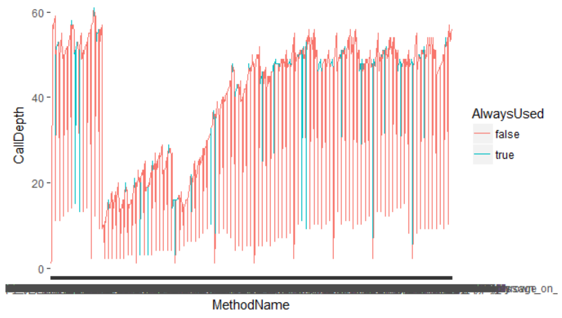
Methods

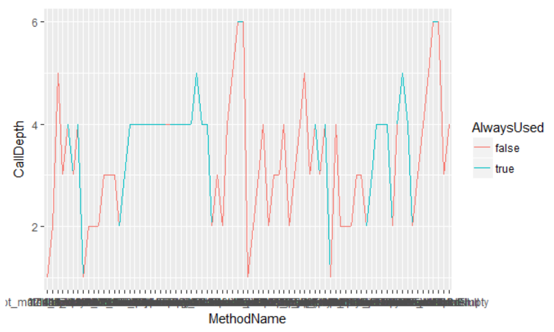
- org/assertj/core/api/Assertions.assertThat
- org/assertj/core/api/AbstractAssert.describedAs
- org/assertj/core/api/AbstractAssert.isEqualTo

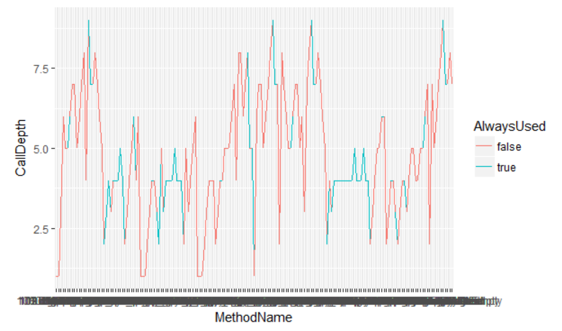
- **Execution Phases**

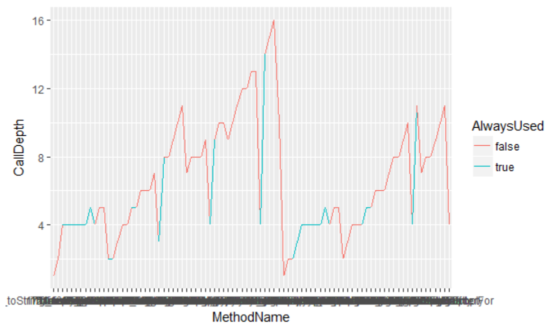
We read the traces log files and produce csv files with 3 columns, Method Name, Call Depth and AlwaysUsed. but we add a number for each method name so we can make different to the execution of two methods in different times.
“dataFiles/3A_MetricsAndRelations/ExecutionPhases/1_ExecutionPhasesByTest”

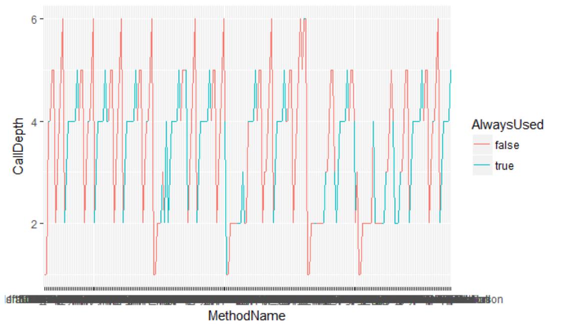


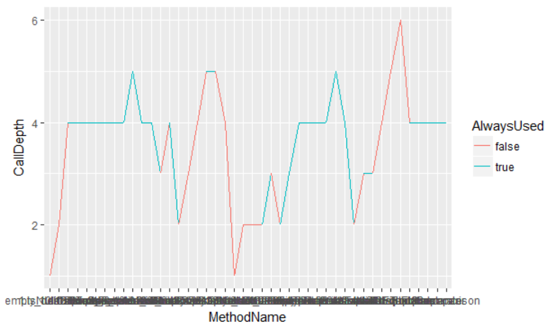


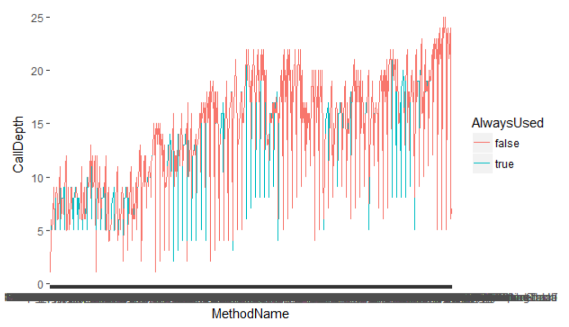


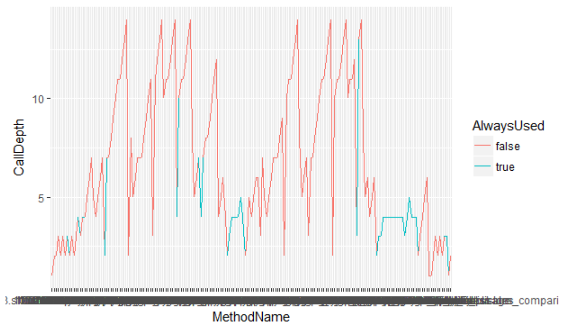


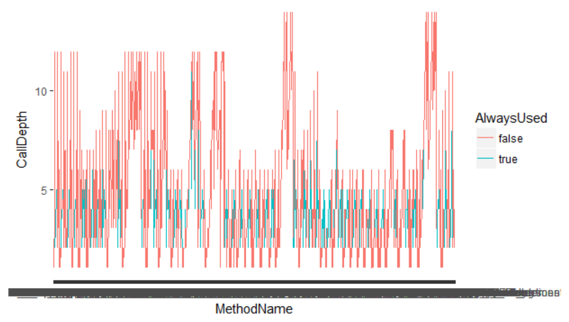












we can find that the system can go through many different phases depending on the run-time data and data state “variable values”.

On the other hand, we can find that the system can go through many different paths in the control flow diagram depending as well on the run-time data and data state. Methods called in some executions may not be called in other executions and classes used in one execution may not be used in some other executions.

We can understand how the system works by looking at the similarity of methods executions in the charts, and we can understand the dependencies of methods on each other by looking on the call depth.

We can understand the structure and the design from the execution phases and class diagram, and we can notice that the system depends on the same methods in every execution significantly, and the system distinguish the types of chart by using a small amount of different methods.

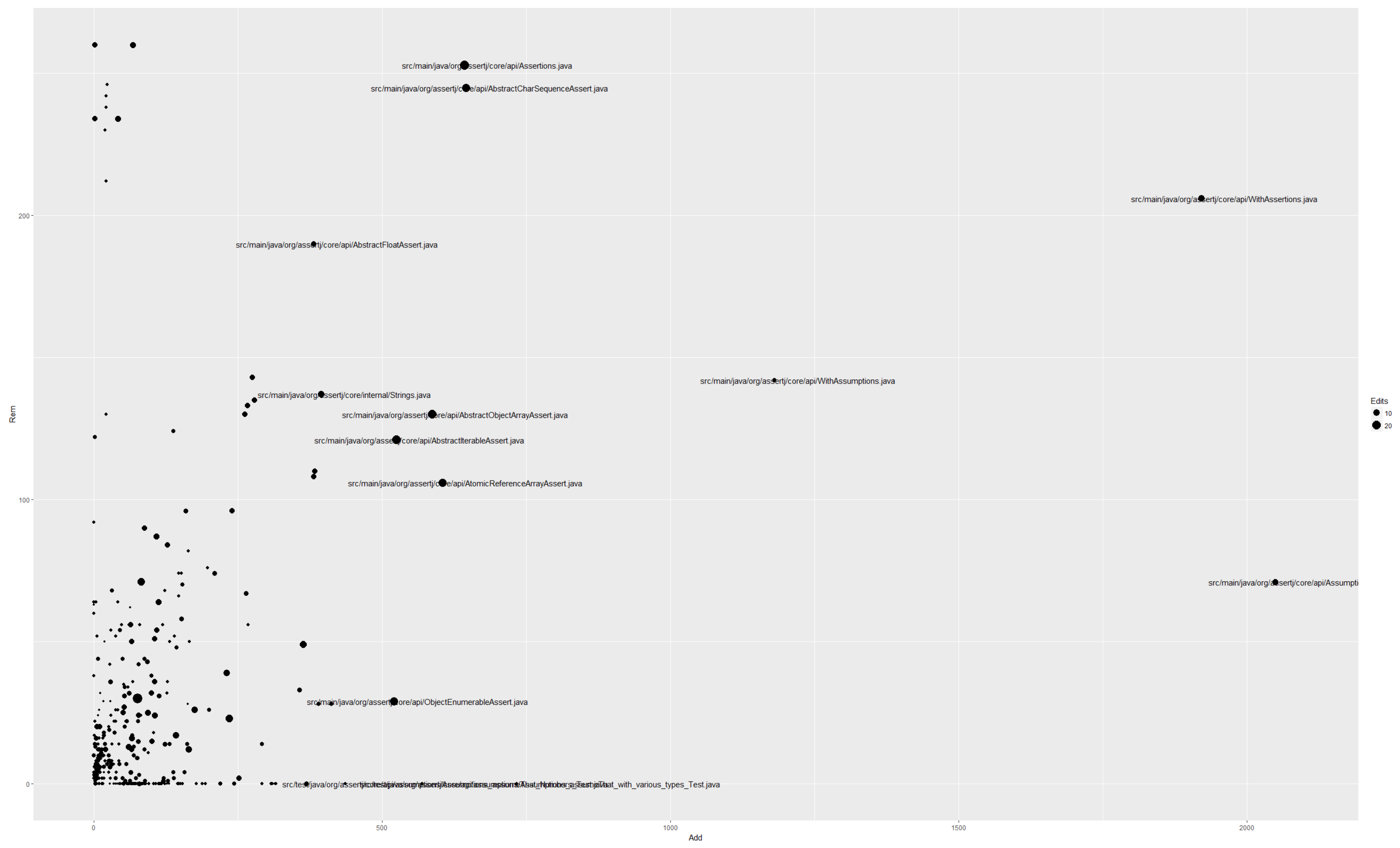
Also, we can see what the main phases are during the execution.

The most important main phases from my point of view are:

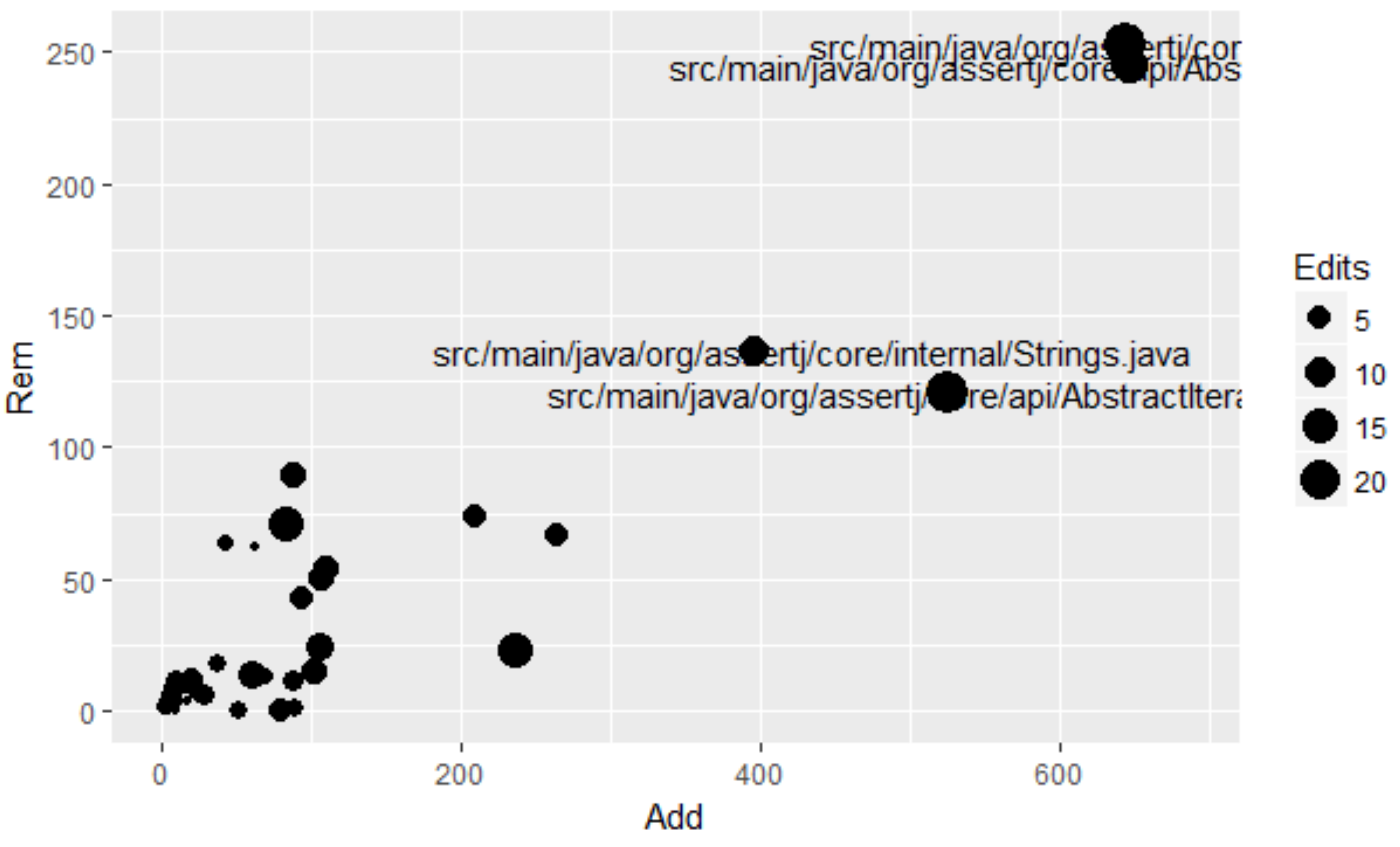
- assertThat
- lineSeparator
- comparison
- representation
- checkNotNull

- **Repository Log (Code Churn)**

We highlighted all classes with big total number of added or changed lines of code because code that changes often pre-release is more likely to contain faults post-release.
“dataFiles/3A_MetricsAndRelations/RepositoryLog/AllChanges”



Then we visualized only important classes from the dynamic analysis
“ dataFiles/3A_MetricsAndRelations/RepositoryLog/ImportantChanges”



We can find that the classes with top 10 total number of added or changed lines are:

- org/assertj/core/api/AbstractCharSequenceAssert
- org/assertj/core/internal/Comparables
- org/assertj/core/api/AbstractIterableAssert
- org/assertj/core/internal/Iterables
- org/assertj/core/api/AbstractCharSequenceAssert
- org/assertj/core/api/AbstractIntegerAssert
- org/assertj/core/api/AbstractIterableAssert
- org/assertj/core/api/AbstractObjectAssert
- org/assertj/core/api/Assertions
- org/assertj/core/error/ShouldBeEqual

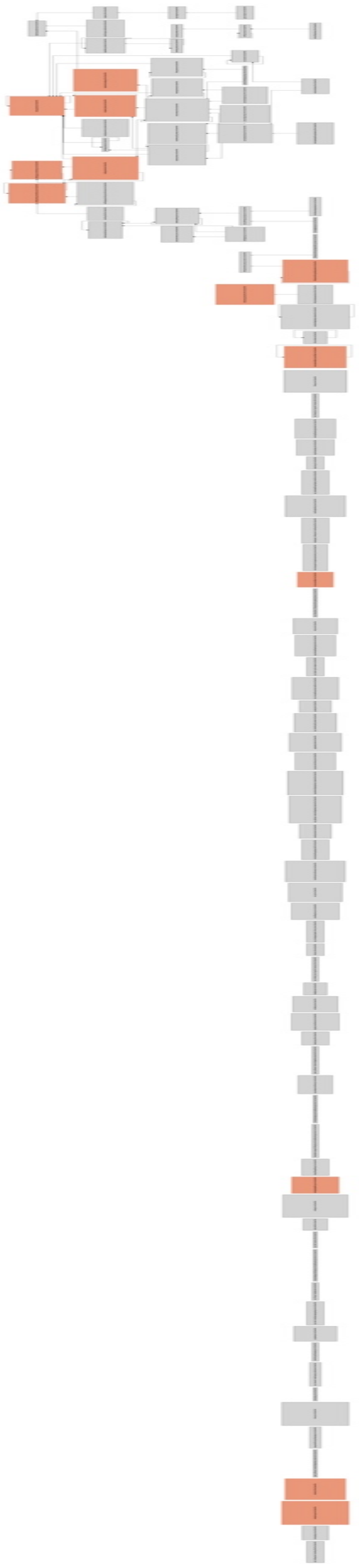
B. Visualise the Class Structure

We used ClassDiagram.java to visualize the class diagram for all classes in the system
"analysisCode/src/test/java/ClassDiagram.java"
"dataFiles/3B_ClassDiagram/1- AllClasses"

Then we generated a class diagram for the traces file classes, so we took all classes files from the system, then we check if the class in the traces classes set, then we found the inheritance and association relations using getSuperclass() and getDeclaredFields() and added the classes if they are in the traces classes set.

After adding the classes and relations we gave the classes different sizes depending on the occurrence in the traces files, we found how many different "TotalOccurrence" numbers in the classes file we created. and put all the TotalOccurrence numbers in sorted set, then we divided 5 on that size of that set, and for each class we multiply the result of dividing by the index of the TotalOccurrence belonging to that class. So, we found different size of each class, and the sizes are between 0 and 5.

Then we add a colour for each class used in each test case by checking the alwaysUsed coloumn.
"analysisCode/src/main/java/ClassDiagram.java"
Below, we can find the class diagram:
"dataFiles/3B_ClassDiagram/2- ImportantClasses"

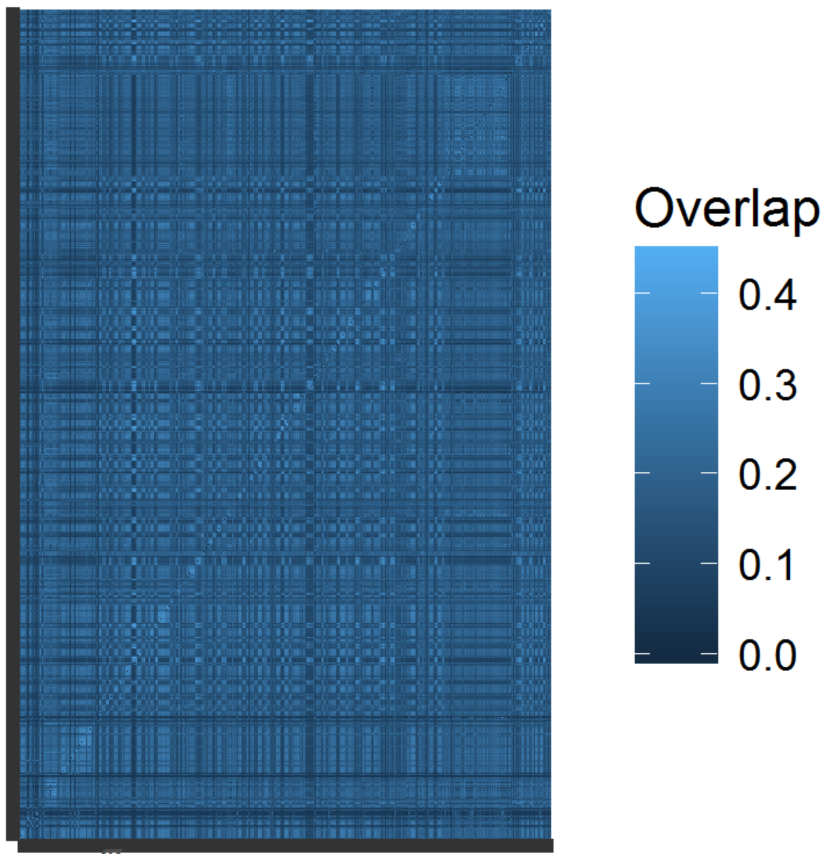


From the class diagram, we can find the classes which always used by the dynamic analysis test cases, and we can find the classes which occur in each run of the system, and these classes are:

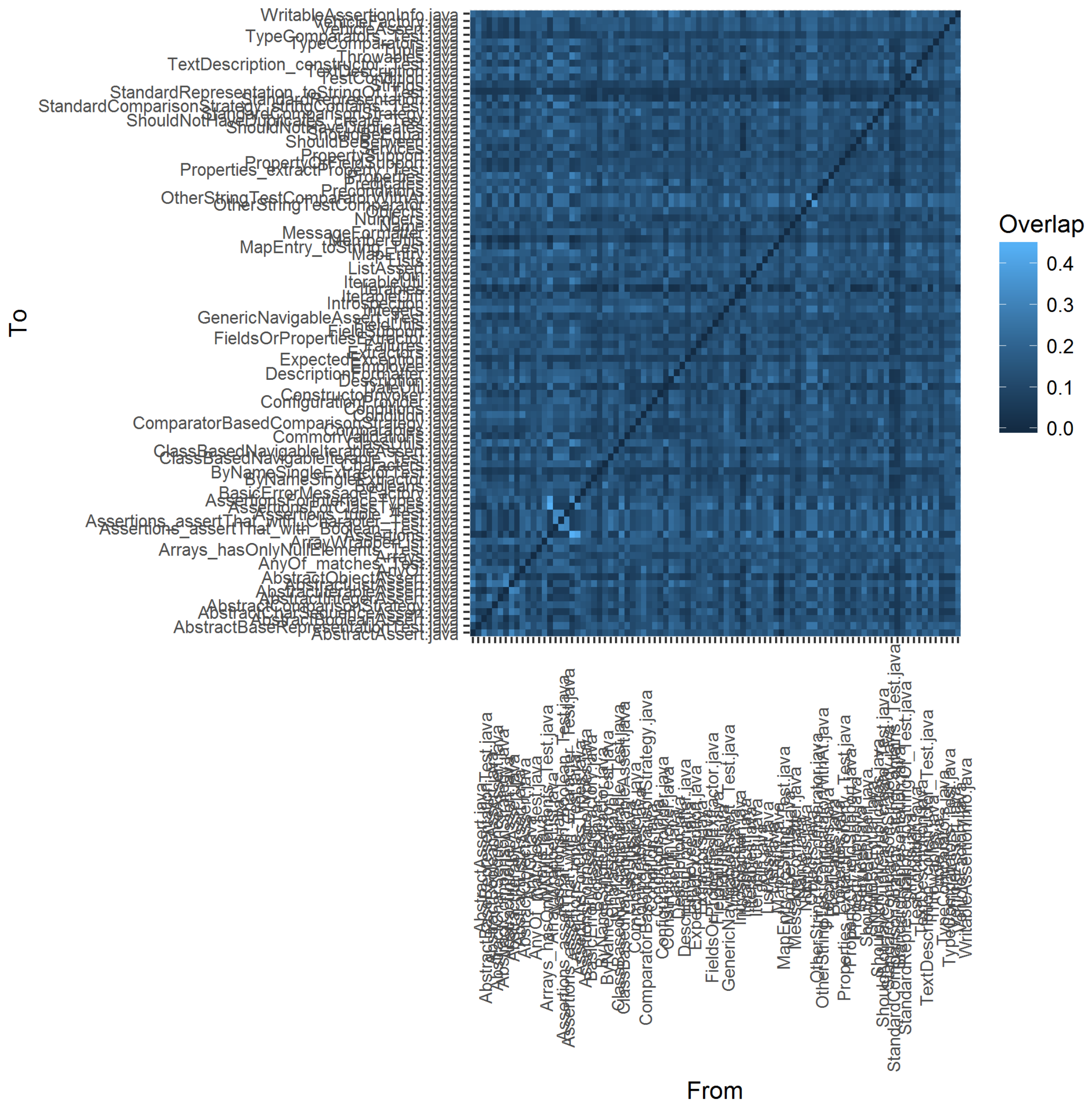
- org.assertj.core.api.Assertions
- org.assertj.core.api.AssertionsForClassTypes
- org.assertj.core.internal.Failures
- org.assertj.core.internal.Conditions
- org.assertj.core.api.WritableAssertionInfo

C. Code Duplication

We visualized the code duplication for all classes in the system as follow:
"dataFiles/3C_CodeDuplication/1- AllClasses"



Then we visualized the code duplication for only important classes using the dynamic analysis as we said before.
"dataFiles/3C_CodeDuplication /2- ImportantClasses"



We can find that the system has some duplication, and the maximum overlap is 0.4 and the duplication codes are in the files below:

- Assertions - AssertionsForClassTypes
- Assertions - AssertionsForInterfaceTypes

From code duplication analysis, we can find that the system doesn't have many duplication code and we don't have to consider code duplication in the system re-engineering

Reengineering

Our intent is to set out which areas of the system could be candidates for re-engineering using the previous analysis techniques and tools.

From the initial exploration of the system we found that there are important and big classes by applying “Skim the Code in one hour” pattern. and these classes are:

org/assertj/core/api/Assertions
org/assertj/core/api/AbstractIterableAssert
org/assertj/core/api/AssertionsForClassTypes

Also, we found that the complete guide with examples is missing from applying “Skim the documentation and Chat with Maintainers” patterns

Then from applying static and dynamic analysis we calculated some metrics to study the weakness and strength of the system, and we found the most important classes and methods of the system:

org/assertj/core/api/AbstractIterableAssert
org/assertj/core/api/Assertions
org/assertj/core/api/AbstractAssert
org/assertj/core/api/AssertionsForClassTypes

Then we created some visuals using the metrics and relationships between metrics and we created class diagram and duplication diagram and we found the important classes using these visuals and they are:

org/assertj/core/api/AbstractCharSequenceAssert
org/assertj/core/internal/Comparables
org/assertj/core/api/AbstractIterableAssert
org/assertj/core/internal/Iterables
org/assertj/core/api/AbstractCharSequenceAssert
org/assertj/core/api/AbstractIntegerAssert
org/assertj/core/api/AbstractIterableAssert
org/assertj/core/api/AbstractObjectAssert
org/assertj/core/api/Assertions
org/assertj/core/api/AssertionsForClassTypes

We found that the classes below are common in all metrics previous steps and from the statistics we believed they are the most important classes and they are the candidates for re-engineering:

org/assertj/core/api/Assertions
org/assertj/core/api/AbstractIterableAssert
org/assertj/core/api/AssertionsForClassTypes
org/assertj/core/api/AbstractAssert

We found that the above classes are having the same problem which is they are all God Classes because they have lots of methods.
So, we focused in our reengineering project on splitting up the God Classes.

We applied the reengineering to:

org/assertj/core/api/AssertionsForClassTypes

Because AssertionsForClassTypes class have:

- 174 LOC
- 104 Weighted Method Count
- 88 Total Occurrence in our runs of Test Cases
- Always Used in all our runs of Test Cases
- Changed 5 times in the last 150 commits to repository logs (264 lines added and 67 lines removed)

We should also note that there are some methods in other classes have a cyclomatic complexity between 21 – 50 which put the system in high risk and these methods should be reengineered but we will not go through that in this project.

Reengineering – Refactoring

Our intent is to get rid of **assertionsForClassTypes** class and move the behaviors close to the data by putting everything where it belong.

First, we created Façade class "AssertionsFacade" to isolate what we need to change, then we pointed all of the client classes to this Façade class. So, whatever comes to this Façade class should go where we want to go.

Example of the first commit:

```
package org.assertj.core.api;
import org.assertj.core.util.CheckReturnValue;

public class AssertionsFacade {
   static AssertionsForClassTypes toBeReengineered;

       public static AbstractDoubleAssert<?> assertThat(double actual) {
              AbstractDoubleAssert<?> x = toBeReengineered.assertThat(actual);
              return x;
       }

       public static AbstractDoubleAssert<?> assertThat(Double actual) {
              AbstractDoubleAssert<?> x = toBeReengineered.assertThat(actual);
              return x;
       }
}
```

"Façade Class gave us the control of which functionality going where"

Then we changed the client class "**Assertions**", we linked it directly to AssertionsFacade class instead of assertionForClassTypes class.

```
public class Assertions {

  static AssertionsFacade system;
.
.
.   }
```

Then, in the second commit we moved each function to its origin, for example for "**BigDecimals**", we moved the following method from assertionForClassTypes class to org.assertj.core.internal.BigDecimals:

```
public static AbstractBigDecimalAssert<?> assertThat(BigDecimal actual) {
           return new BigDecimalAssert(actual);
         }
```

We did the same thing for the following classes:

- Booleans
- BooleansArrays
- Bytes
- BytesArrays
- Characters
- CharactersArrays
- Classes
- Doubles
- Floats
- Integers
- DoublesArrays
- FloatsArrays
- IntArrays
- …

And we changed the client class "Assertions" to call the methods using the above classed instead of assertionForClassTypes class.

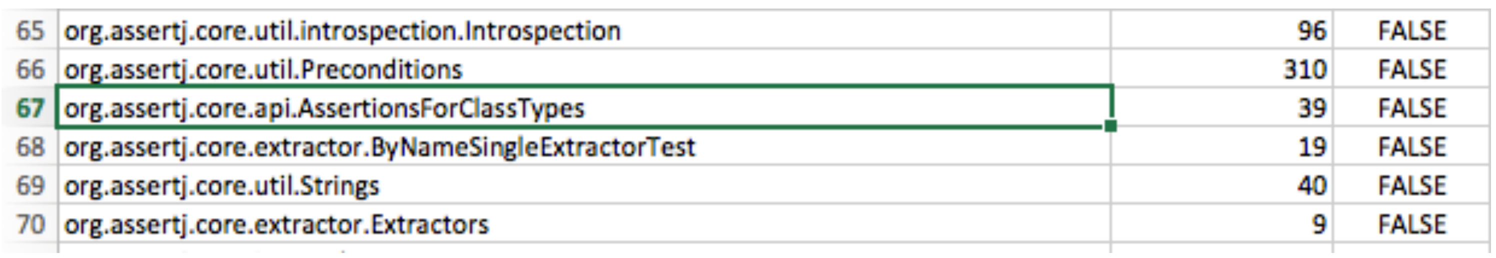
We used the test cases from Assertj project to ensure that our refactoring didn't break anything. Examples of the test cases we used:

/assertj-core/src/test/java/org/assertj/core/api/Assertions_assertThat_with_Double_Test.java
/assertj-core/src/test/java/org/assertj/core/api/Assertions_assertThat_with_Float_Test.java
/assertj-core/src/test/java/org/assertj/core/api/Assertions_assertThat_with_Integer_Test.java
/assertj-core/src/test/java/org/assertj/core/api/BooleanArrayAssertBaseTest.java
. . .

Then we cleaned the project and built it up to ensure that our changes didn't break anything, then we run all of test cases successfully.

After that, we did some analysis from the previous chapters to check out how our re-engineering process went. And the results for AssertionsForClassTypes class are:
"dataFiles/5_Reengineering/"

- 112 LOC
- 73 Weighted Method Count
- 39 Total Occurrence in our runs of Test Cases
- **Not always Used** in all our runs of Test Cases



65	org.assertj.core.util.introspection.Introspection	96	FALSE
66	org.assertj.core.util.Preconditions	310	FALSE
67	org.assertj.core.api.AssertionsForClassTypes	39	FALSE
68	org.assertj.core.extractor.ByNameSingleExtractorTest	19	FALSE
69	org.assertj.core.util.Strings	40	FALSE
70	org.assertj.core.extractor.Extractors	9	FALSE

From the previous statistics, we can found that we split up the God Class AssertionsForClassTypes class and we reduced the dependencies between classes by ending the need of linking to AssertionsForClassTypes class from other classes.

We can do the same steps for other God Classes to improve the system, but here we just applied the reengineering as example to one God Class as requested in the assignment.