COM3523 Software Re-engineering

Individual Assignment

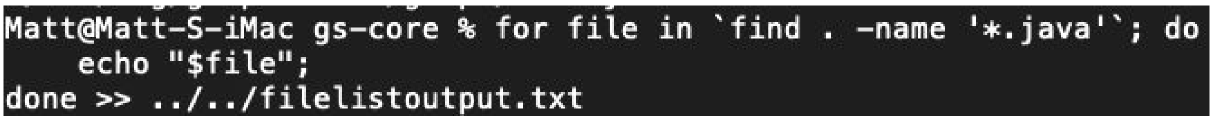
Matthew Sykes

aca17mts

Chapter 1: Analysis of the System

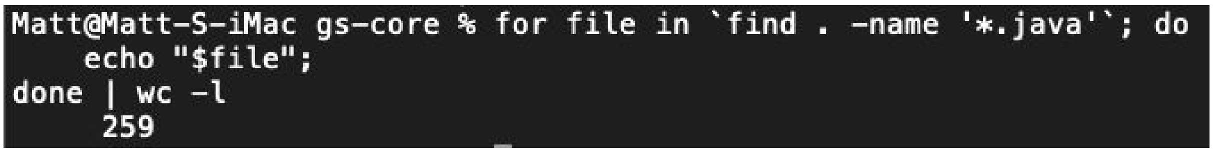
1.1 Bash Commands

Firstly, a set of Bash command to determine the source files of GraphStream were executed.



```
Matt@Matt-S-iMac gs-core % for file in `find . -name '*.java'`; do
    echo "$file";
done >> ../../filelistoutput.txt
```

Figure 1.1: Bash command to list Java files.



```
Matt@Matt-S-iMac gs-core % for file in `find . -name '*.java'`; do
    echo "$file";
done | wc -l
     259
```

Figure 1.2: Bash command which gives the number of Java files.

As shown in Figure 1.1 the list of Java source files are listed by using a for loop find command. Figure 1.2 then shows how when the 'wc -l' command is appended to the primary command, the console returns that there are 259 source files.

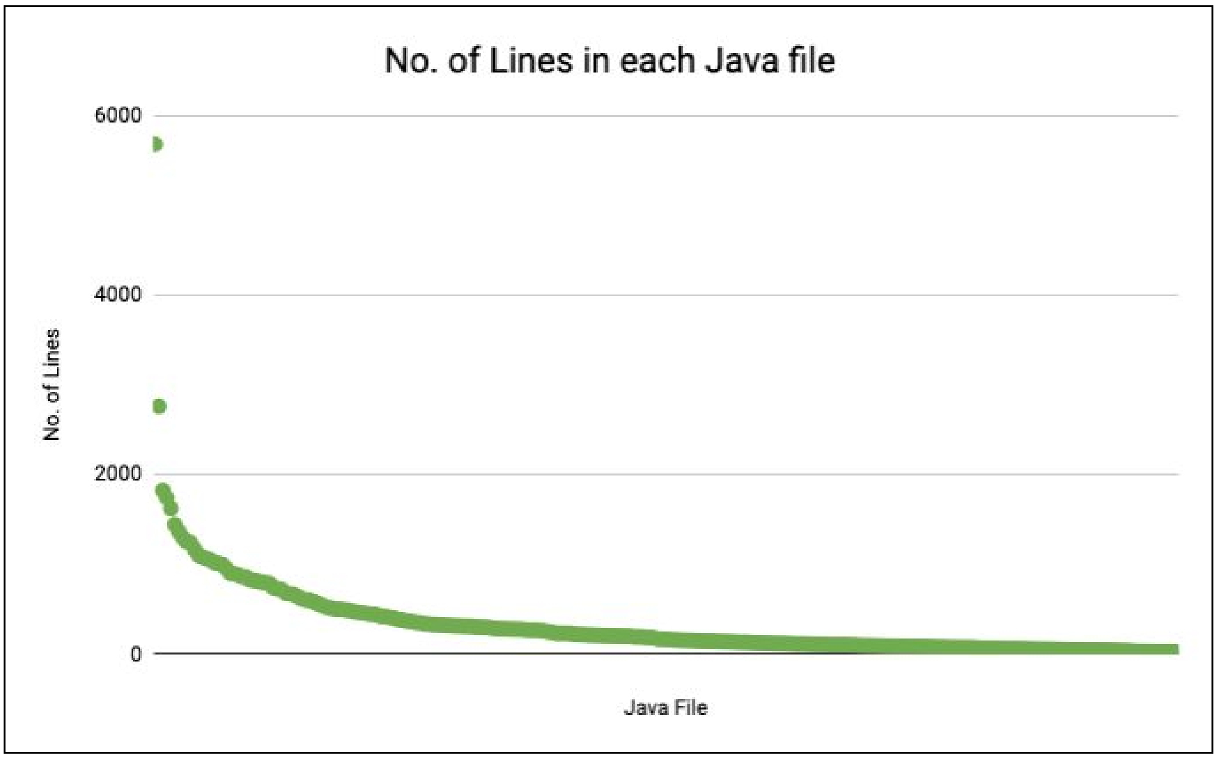
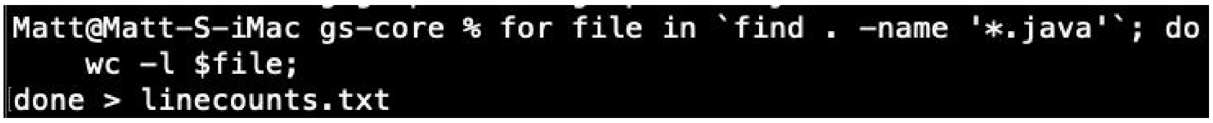


Figure 1.3: Scatter graph showing the number of lines for each Java file in GraphStream.

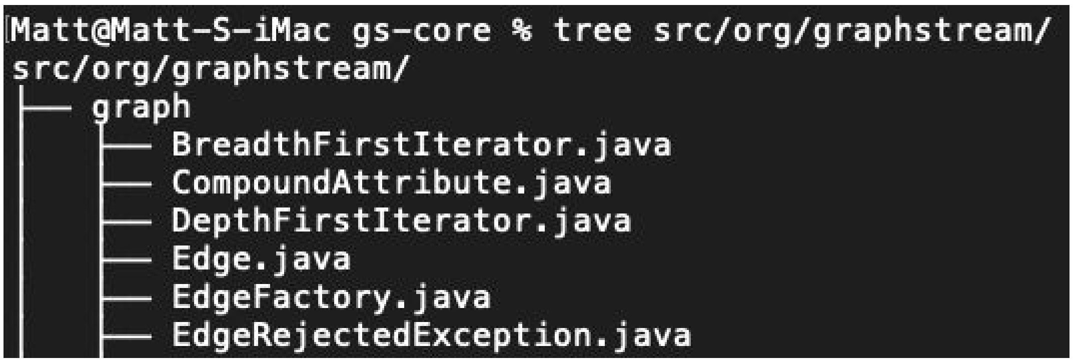


```
Matt@Matt-S-iMac gs-core % for file in `find . -name '*.java'`; do
    wc -l $file;
done > linecounts.txt
```

Figure 1.4: Bash command for listing every Java file line count.

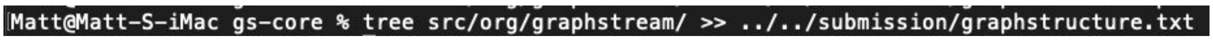
To analyse the number of lines per file, the bash command 'wc -l' was added to the existing for loop-based find command. The output was saved to a text file and then copied to a spreadsheet. Using the spreadsheet software, a scatter graph was produced shown in the figure above. The graph and text output both show that there are two candidates for potential 'God' classes: 'PajekParser.java' and 'StyleSheetToeknManager.java' with 5680 and 2760 lines of code each respectively which makes them unusual. 15 further files had more than 1000 lines of code.

To determine the exact structure of the GraphSteam project, a tree command was executed to output a tree data structure:



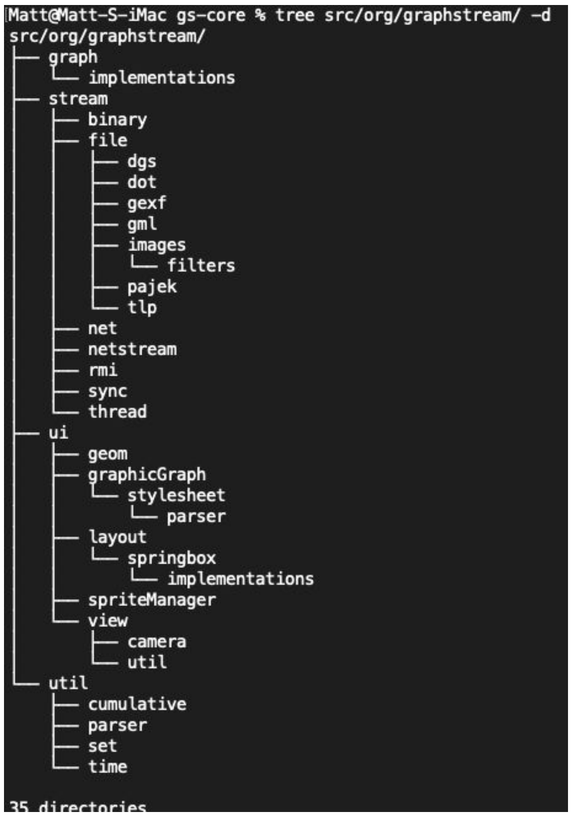
```
Matt@Matt-S-iMac gs-core % tree src/org/graphstream/
src/org/graphstream/
├── graph
│   ├── BreadthFirstIterator.java
│   ├── CompoundAttribute.java
│   ├── DepthFirstIterator.java
│   ├── Edge.java
│   ├── EdgeFactory.java
│   ├── EdgeRejectedException.java
```

Figure 1.5: Tree command.



```
Matt@Matt-S-iMac gs-core % tree src/org/graphstream/ >> ../../submission/graphstructure.txt
```

Figure 1.6: Output to text file command.



```
Matt@Matt-S-iMac gs-core % tree src/org/graphstream/ -d
src/org/graphstream/
├── graph
│   └── implementations
├── stream
│   ├── binary
│   ├── file
│   │   ├── dgs
│   │   ├── dot
│   │   ├── gexf
│   │   ├── gml
│   │   ├── images
│   │   │   └── filters
│   │   ├── pajek
│   │   └── tlp
│   ├── net
│   ├── netstream
│   ├── rmi
│   ├── sync
│   └── thread
├── ui
│   ├── geom
│   ├── graphicGraph
│   │   └── stylesheet
│   │       └── parser
│   ├── layout
│   │   └── springbox
│   │       └── implementations
│   ├── spriteManager
│   └── view
│       ├── camera
│       └── util
└── util
    ├── cumulative
    ├── parser
    ├── set
    └── time

35 directories
```

Figure 1.7: Outputted tree structure when tree command was executed.

The full output with all the files in ‘src’ was saved to a text file named ‘graph structure.txt’.

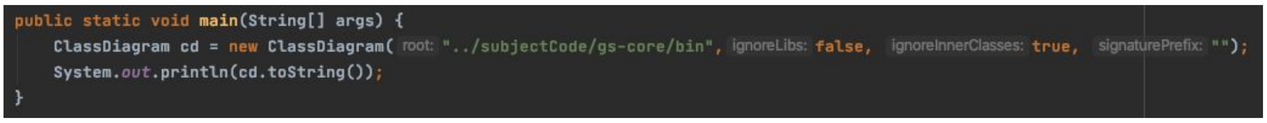
The graph data structure shows how the files in the project are organised into sub-directories, the deepest level of sub-directories is four. There were 35 directories in total.

The main class in GraphStream is Timeline.java within the stream package, this class implements the Source and Replayable Java interfaces. The location of this class is src/org/stream.

1.2 Static Analysis

For the Static Analysis, a 'fat Jar' was created so the Java dependencies can be added in bulk to the classpath. The command "mvn clean package" was executed using the pom.xml plugin code; this created a 'fat Jar' in the target directory.

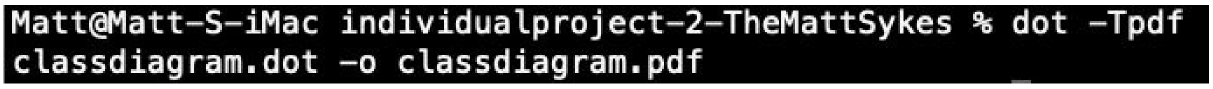
The 'fat Jar' was then added to the analysisCode project so that it could be referenced when creating a class diagram.



```java
public static void main(String[] args) {
    ClassDiagram cd = new ClassDiagram( root: "../subjectCode/gs-core/bin", ignoreLibs: false,  ignoreInnerClasses: true,  signaturePrefix: "");
    System.out.println(cd.toString());
}
```

Figure 1.8: Main class added to ClassDiagram to execute the code.

By adding a main method in ClassDiagram.java, the class was instantiated to generate a class diagram with no signaturePrefix set. The result was outputted to the console and then saved to a dot file.



```
Matt@Matt-S-iMac individualproject-2-TheMattSykes % dot -Tpdf
classdiagram.dot -o classdiagram.pdf
```

Figure 1.9: Dot command to generate a PDF of the graph.

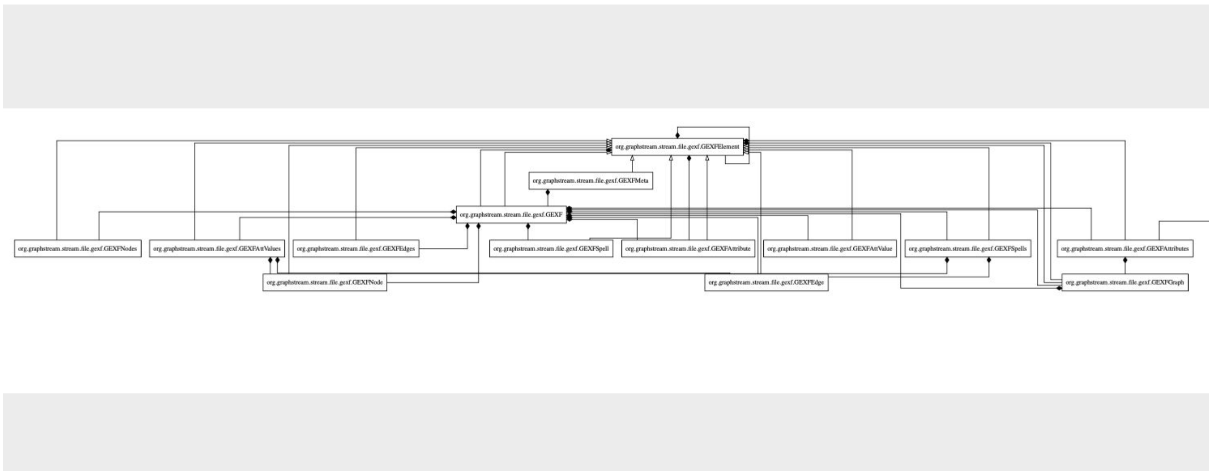
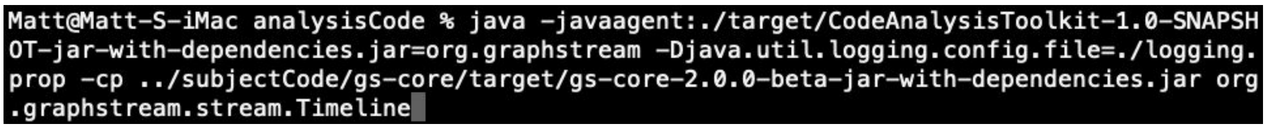

org.graphstream.stream.file.gexf.GEXFElement
org.graphstream.stream.file.gexf.GEXFMeta
org.graphstream.stream.file.gexf.GEXF
org.graphstream.stream.file.gexf.GEXFNodes
org.graphstream.stream.file.gexf.GEXFAttValues
org.graphstream.stream.file.gexf.GEXFEdges
org.graphstream.stream.file.gexf.GEXFSpell
org.graphstream.stream.file.gexf.GEXFAttribute
org.graphstream.stream.file.gexf.GEXFAttValue
org.graphstream.stream.file.gexf.GEXFSpells
org.graphstream.stream.file.gexf.GEXFAttributes
org.graphstream.stream.file.gexf.GEXFNode
org.graphstream.stream.file.gexf.GEXFEdge
org.graphstream.stream.file.gexf.GEXFGraph

Figure 1.10: Sample of outputted PDF graph.

From this, using GraphViz the class diagram was converted to a more readable diagram in PDF format.

1.3 Dynamic Analysis



```
Matt@Matt-S-iMac analysisCode % java -javaagent:./target/CodeAnalysisToolkit-1.0-SNAPSH
OT-jar-with-dependencies.jar=org.graphstream -Djava.util.logging.config.file=./logging.
prop -cp ../subjectCode/gs-core/target/gs-core-2.0.0-beta-jar-with-dependencies.jar org
.graphstream.stream.Timeline
```

Figure 1.11: Bash command to execute Java agent on Timeline.java.

For the dynamic analysis, ‘mvn package’ was executed in gs-core to generate another fat Jar containing the required dependencies. To obtain a trace file the command in the figure above was executed using the logging Java utilities file and ‘logging.prop’, the trace output was then stored in ‘logged.log’. The class ‘Timeline.java’ was chosen as the file to trace as this class is the project's main class. Performing ‘wc -l’ showed that the log file contained 690 lines.

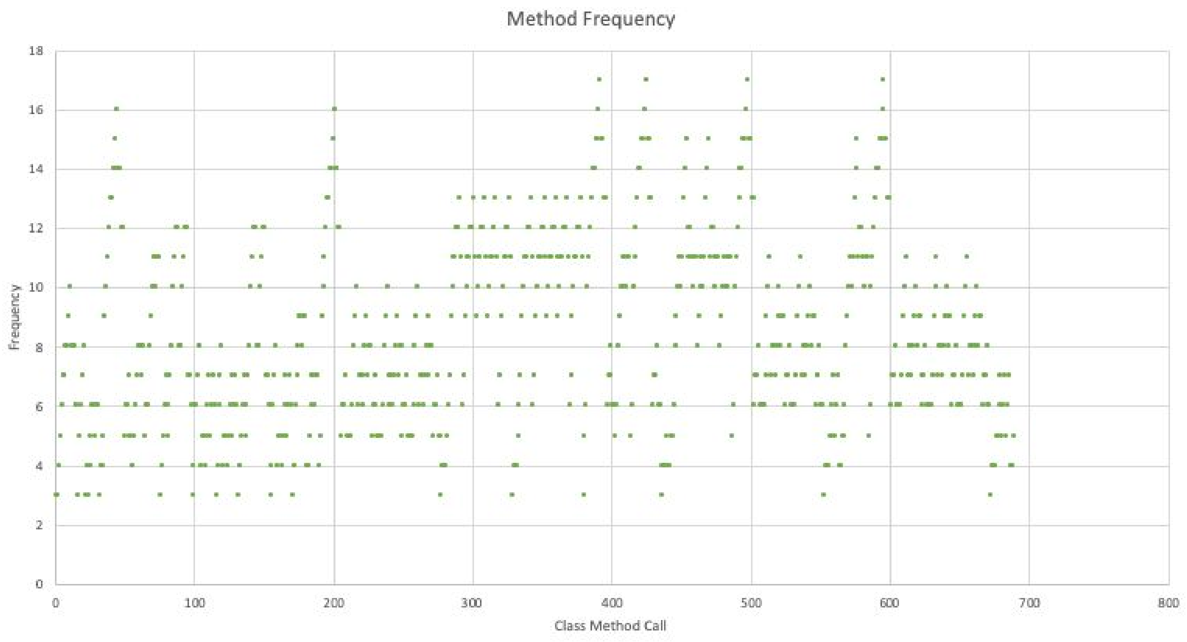
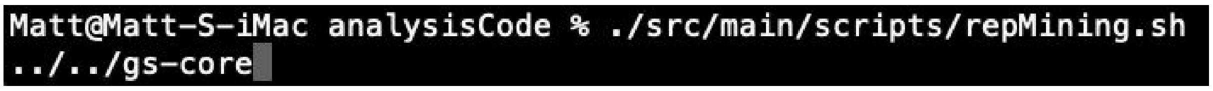


Figure 1.12: Scatter graph for method frequency.

From the log file, the values were extracted using comma-separated values into spreadsheet software; using this software a scatter graph was created shown in the figure above. From this, it can be extrapolated that the highest number of calls for any particular function was 17, and 4 methods reached this maximum amount.

1.4 Version Repository Analysis



```
Matt@Matt-S-iMac analysisCode % ./src/main/scripts/repMining.sh
../../gs-core
```

Figure 1.13: Command for running repMining shell script on gs-core.

To perform a version repository analysis GraphStream's repository was separately cloned from GitHub then checked out to version 807bea7. Next, the repMining.sh shell script was executed using this repository to generate a CSV file containing all the version repository changes.

The CSV file was then imported into spreadsheet software where a pivot table was created. From this, the file names were set to the row values, the committer to the column values and a count of the timestamps to the values section. A bar chart graph from this data was then extrapolated to examine which areas of the code were modified by version control the most.

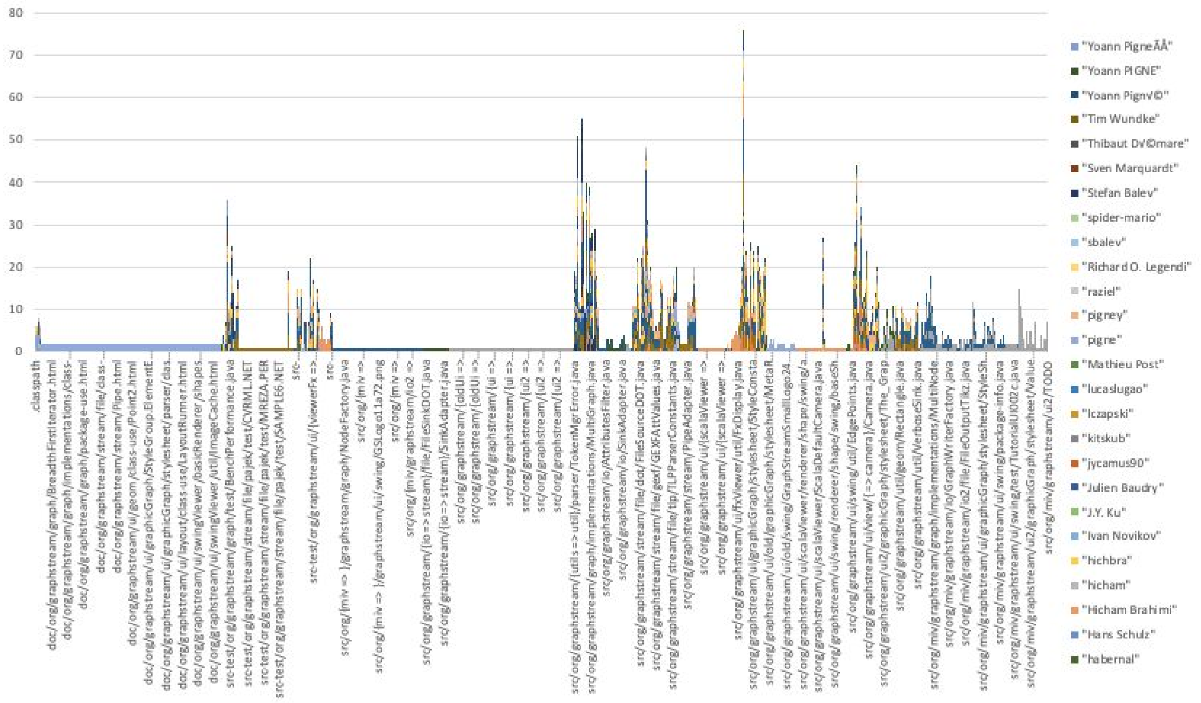
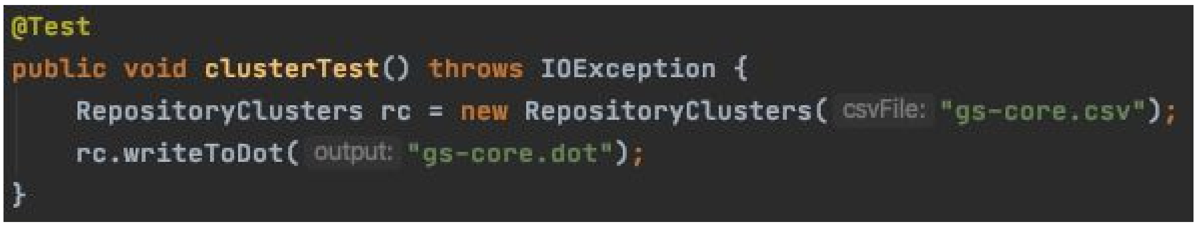


Figure 1.14: Bar graph showing most frequently altered Java files in GraphStream.

From the graph in the figure above it can be determined that GraphicGraph.java featured the most changes in the repository. Graph.java and AbstractGraph.java also featured a considerable emphasis when changes were being made.

For the converted CSV spreadsheet, a new column was added named 'Bug?' which was designed to list whether the commit message for each file contained the word "bug" if so, a Boolean True would be outputted (Using "=ISNUMBER(SEARCH("fix",<cell>))"). Then using "=COUNTIF(G2:G8424, TRUE)" the number of cells that were true was calculated. The result of this was 421, which shows that at least 421 repository commits were fixes.



```java
@Test
public void clusterTest() throws IOException {
    RepositoryClusters rc = new RepositoryClusters( csvFile: "gs-core.csv");
    rc.writeToDot( output: "gs-core.dot");
}
```

Figure 1.15: Test method/case for running RepositoryClusters code.

Next, running RepositoryClusters on the CSV file produced a dot file where each node is a file if two nodes are connected both have been edited according to the git log.

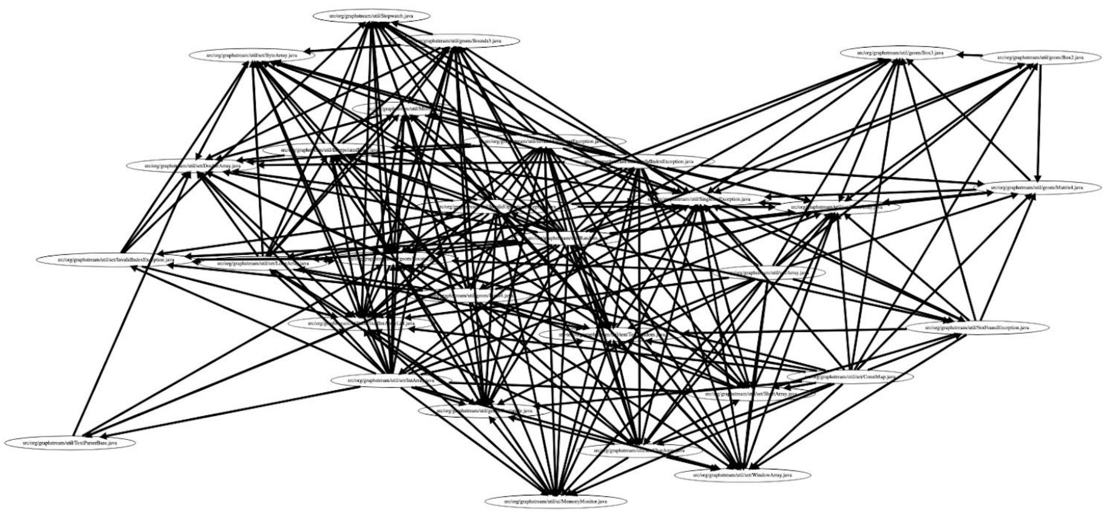


Figure 1.16: Sample of outputted PDF graph.

The dot file was then converted to a pdf file so that the graph can be graphically represented.

Chapter 2: Assessment and Critique

During the analysis using bash commands, it was established that there were several God class candidates within GraphStream. Upon closer inspection of PajekParser, the class contains a colossal amount of methods most of which return boolean values from an integer input. Judging from the majority of the method names being similar to ‘jj_2_23’ without any documentation for each of these methods, there is a strong chance many of the methods are computer/automatically generated - which could make for a difficult task to split them up. A more optimal method may be to shrink the functions into a smaller number of methods.

The PajekParser also makes poor use of interface design with PajekParserConstants being used to store a considerable number of constant values. Interfaces are used better in cases such as FileSource which defines a range of empty bodied methods.

There are many cases of good documentation using JavaDoc for example in FileSink and GraphFactory where each method is clearly documented; however, in some cases such as NetStreamUtils there are a large number of undocumented methods leading to possible issues during any form of re-engineering/maintenance.

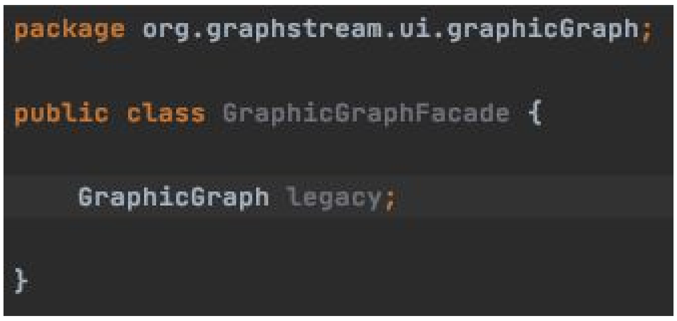
Another case of poor interface use is Sink, where the interface extends AttributeSink and ElementSink but contains no code of its own.

GraphicGraph is another example of a God class in GraphStream, with 1368 lines as shown from the Bash commands and a very large number of methods indicating a large WMC; this is also the class that featured the most changes in the git repository according to the version repository analysis.

There is generally a good use of encapsulation throughout the application, with classes, variables and methods being appropriately encapsulated; GraphicGraph is an example of this with private and protected functions such as Logger and StyleSheet. However, GraphicGraph is also an example of high SLOC with attributeChanged containing 33 lines which is above 10.

Chapter 3: Re-engineering the System

To re-engineer GraphStream, the strategy employed was to focus on the God class GraphicGraph, as this appears to be a fairly crucial class within the system. Although PajekParser is almost 5x larger, the analysis indicated there was a large amount of automatic code with poor documentation; therefore, GraphicGraph would make a more appropriate choice considering its importance.



```
package org.graphstream.ui.graphicGraph;

public class GraphicGraphFacade {

    GraphicGraph legacy;

}
```

Figure 3.1: Empty facade class with legacy instance variable.

The first stage to split up GraphicGraph was to create a reference copy of the class as a backup which can be useful in case errors occur and to use as a control. Next, a facade class was created called GraphicGraphFacade with a single data-member GraphicGraph.

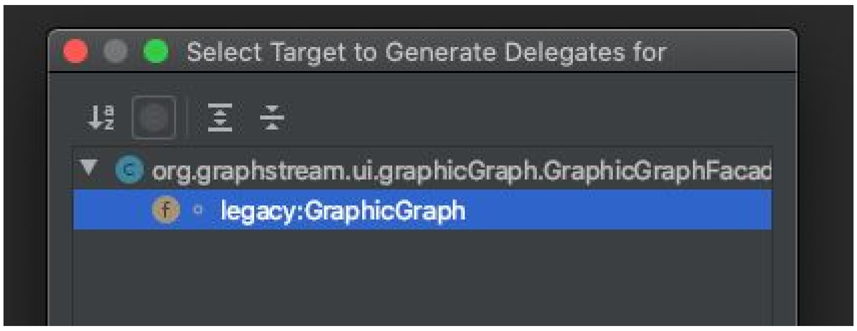
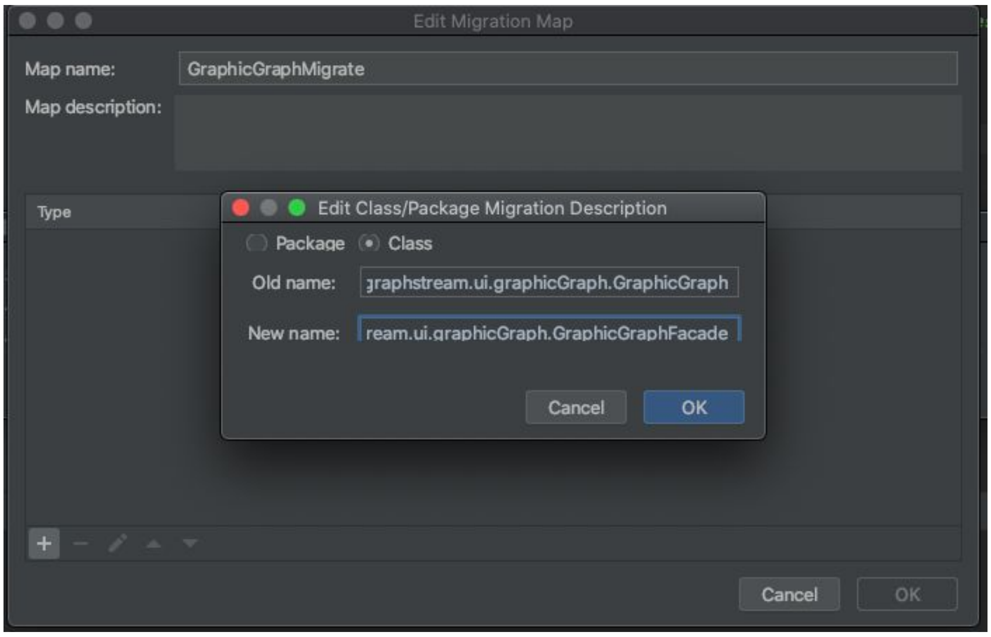

Select Target to Generate Delegates for
org.graphstream.ui.graphicGraph.GraphicGraphFacad
legacy:GraphicGraph

Figure 3.2: Selecting targets for generating delegates in IntelliJ.

Using the IntelliJ IDE, delegate methods are automatically created for all the methods in GraphicGraph.


Edit Migration Map

Map name: GraphicGraphMigrate

Map description:

Edit Class/Package Migration Description

Package [✓ Class]

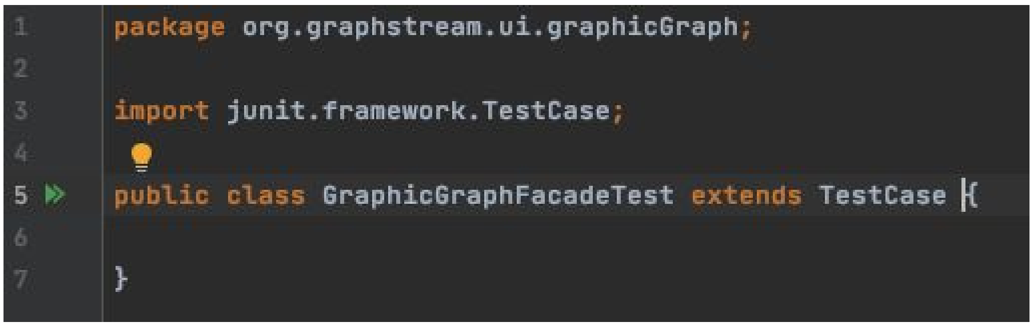
Old name: ɟraphstream.ui.graphicGraph.GraphicGraph

New name: ream.ui.graphicGraph.GraphicGraphFacade

Figure 3.3: Creating a new migration map in IntelliJ.

Next, to replace the GraphicGraph references with references to GraphicGraphFacade IntelliJ’s migration tool was used by creating a new migration map and executing the migration using that map.

Upon trying to build the project to examine whether the migration was fully successful, IntelliJ reported 69 errors. The first set of errors were corrected by adding a constructor and adding the correct inheritance and interface classes. This decreased the error count to 46. 35 of these errors related to initialiser variables not existing in the facade class, this was easily corrected by adding these variables. After adding all the initialiser variables to the facade class constructor and auto-creating missing methods using IntelliJ, the rest of the errors were corrected.


```java
package org.graphstream.ui.graphicGraph;

import junit.framework.TestCase;

public class GraphicGraphFacadeTest extends TestCase {

}
```

Figure 3.4: Empty test class for testing the facade class.

The next stage was to create a test harness, by using IntelliJ an empty test case was automatically created for the facade class.

To look for appropriate test cases, testing methods from TestGraphicGraph.java were examined. Within this class a testing method named basicTest() which tests the class on its own, appeared to be the most appropriate test to conduct for the facade.

The majority of the issues in this class were caused by implementing the Graph interface, resulting in a very large amount of lines being taken up by override functions and Javadoc from the interface. However, considering the size it would be less practical to attempt minimalising the interface here.

To begin splitting up the class, the addEdge method appeared to be used frequently, by extracting the largest polymorphic version of this method to another class named EdgeAdder, the LOC count decreased to 1338.

Next, the positionSprite method was extracted to another class named PositionSprite which decreased the LOC count further to 1301.

Finally, the computeBounds method was extracted which calculates the high and low values. This class was more involved as it affected a range of variables. However, by using a mixture of arrays and extracting other classes, the LOC count was decreased to 1229.