Introduction

I set out to do some dynamic analysis on the org.apache.commons.net project, focusing more around the FTP class. I used Aspect Oriented Programming using a tool called AspectJ to inspect method / class executions and calls. I explain my approach to analysing the FTP class and what technologies I used. Afterwards I give some thoughts around the analysis of the system.

Implementation Details

For the dynamic analysis, I created a tool that could inspect source code during runtime execution to monitor what the system is doing by the user while recording important information regarding method / class executions and calls. I used a methodology called Aspect Oriented Programming and a tool called AspectJ that adheres to the Aspect Oriented Programming. The dynamic tool uses Java, that reads in the AspectJ logs, it then sorts out data and converts it the appropriate data structure to be used for Visualization graphs.

We inspect all methods that belong to the org.commons.apache.net that was executed from the following unit tests –

* FTPClientConfigFunctionalTest
* FTPClientConfigConfigTest
* FTPClientTest
* FTPCommandTest
* ListingFunctionalTest
* TestConnectTimeout

Just a side note the FTPClientConfigFunctionalTest and ListingFunctionTest unit tests did fail when I ran them. So the results that I got might not be 100% accurate when compared to a successful pass on those unit tests.

Those unit tests were recorded with our aspects that inserted the recorded data into a log file tmdcTrace.log which is all trace method calls and tmteTrace.log for all methods executed. They are appended with a numeric value that indicates the unit test that was run i.e. FTPClientConfigFunctionTest = 1. There is a total of 12 log files. 6 of which is tmdcTrace and 6 is tmteTrace.

The dynamic tool then merges all tmdcTrace.log files into one file called “totalTmdcTrace.log” and all tmteTrace.log files into “totalTmteTrace.log”. The totalTmteTrace.log is then read in by the tool, it then gets the average execution times for all methods, it also gets the average execution times for all classes, and it then gets converted into two csv files, one for method execution times and one for class execution times. The totalTmdcTrace.log is read in and all duplicate relationships are removed and then it gets converted into a digraph format ready for Graphviz to use.

The following tools were used for visualising the data we prepared using the dynamic tool.

* Classes called most frequently – Csv file, Excel Bar Graph
* Methods called most frequently - Csv file, Excel Bar Graph
* Classes required most of the time to execute - Csv file, Excel Bar Graph
* Methods required most of the time to execute - Csv file, Excel Bar Graph
* Which methods called each other most frequently – dot file, Graphviz

All data is prepared and converted to the appropriate data format for the graph tools.

How to run the tool.

In the “src” folder there should be the two Aspect compiled files and source files that was used for the assignment.

* TraceMethodDependencyCalls.java -> tmdcTracer.jar
* TraceMethodTimeExecution.java -> tmteTracer.jar

The following command prompts were used to compile both the tmdcTracer (for method calls) and tmteTracer (method/class execution) -

* ajc -cp ..\Libraries\aspectjrt.jar -outxml -outjar tmteTracer.jar TraceMethodTimeExecution.java
* ajc -cp ..\Libraries\aspectjrt.jar -outxml -outjar tmdcTracer.jar TraceMethodDependencyCalls.java

TestConnectTimeout class

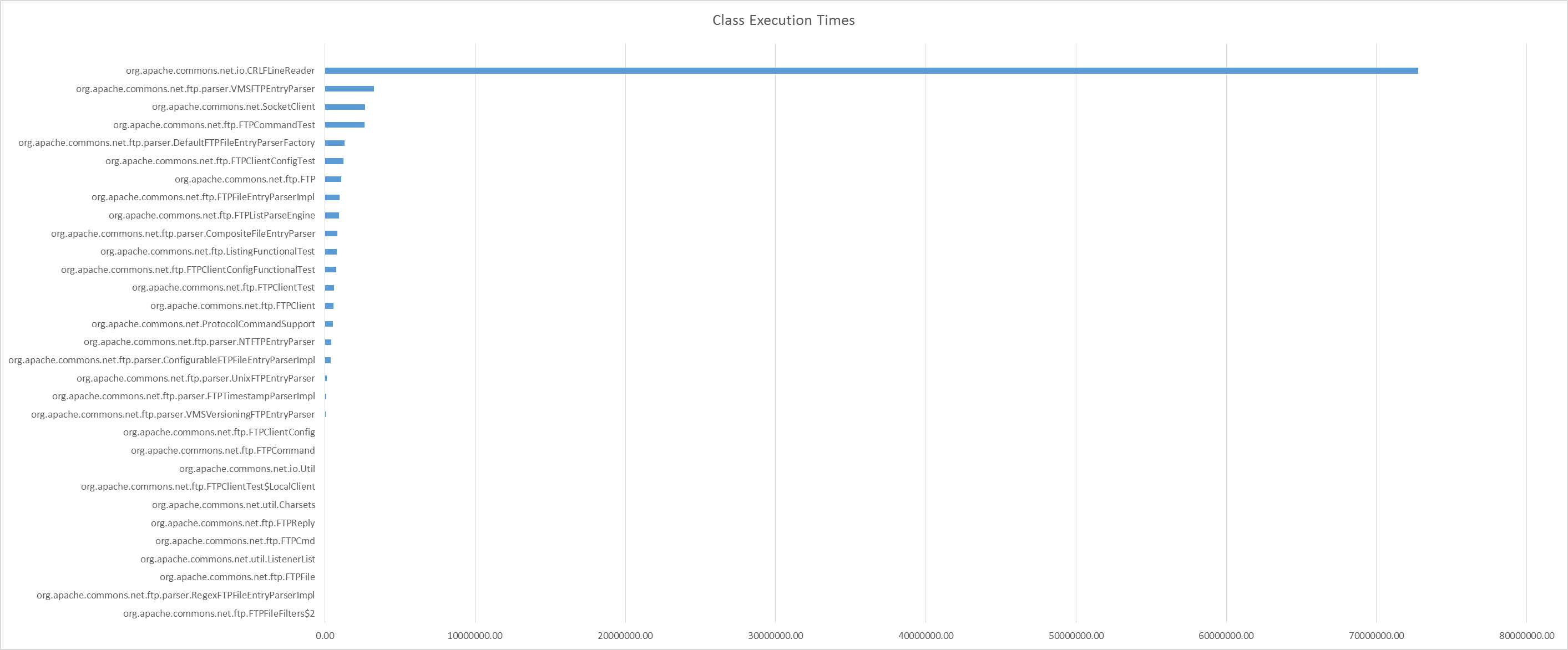
testConnectTimeout method

The final out graphs.

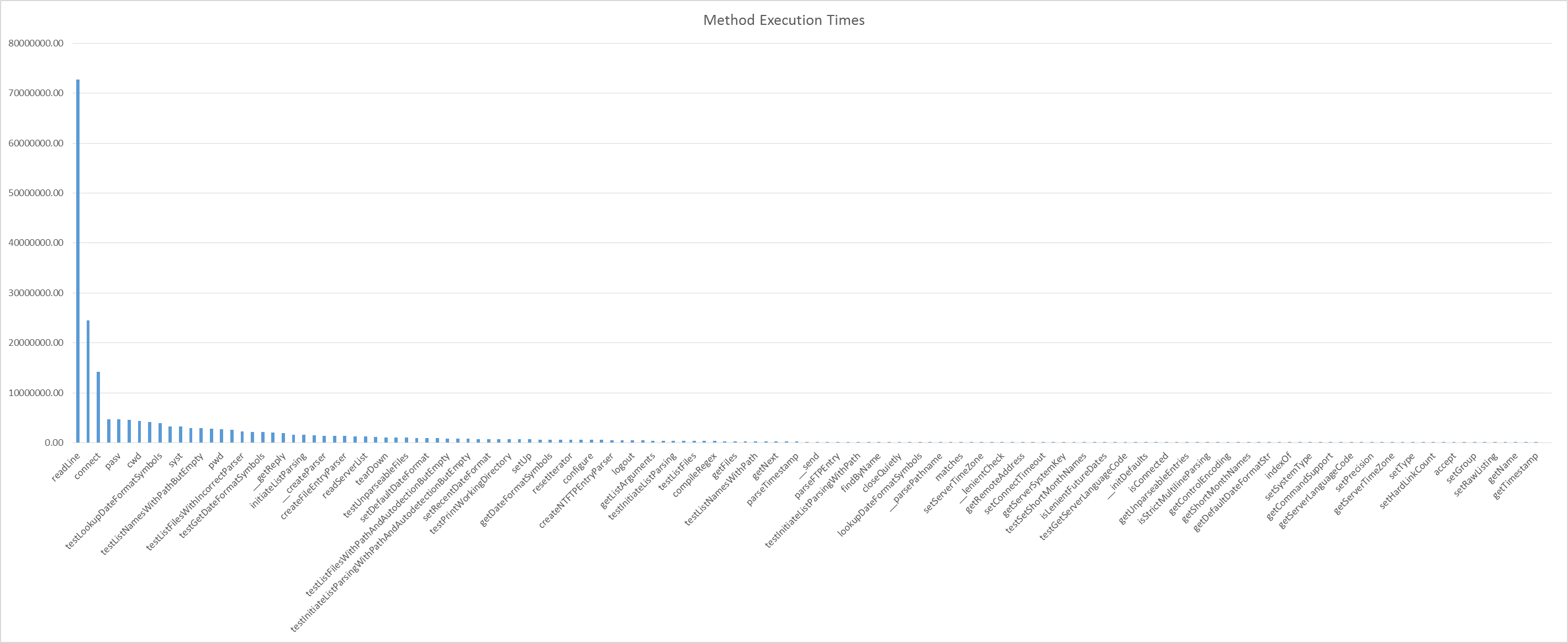
Please note that for graphs “Class Execution Times” and “Method Execution Times” the class “TestConnectTimeout” and method “testConnectTimeout()” were taken away from these graphs respectively since it distorted the graph output and was related to the actual unit test class and method.

There are four graphs that were used to depict the following -

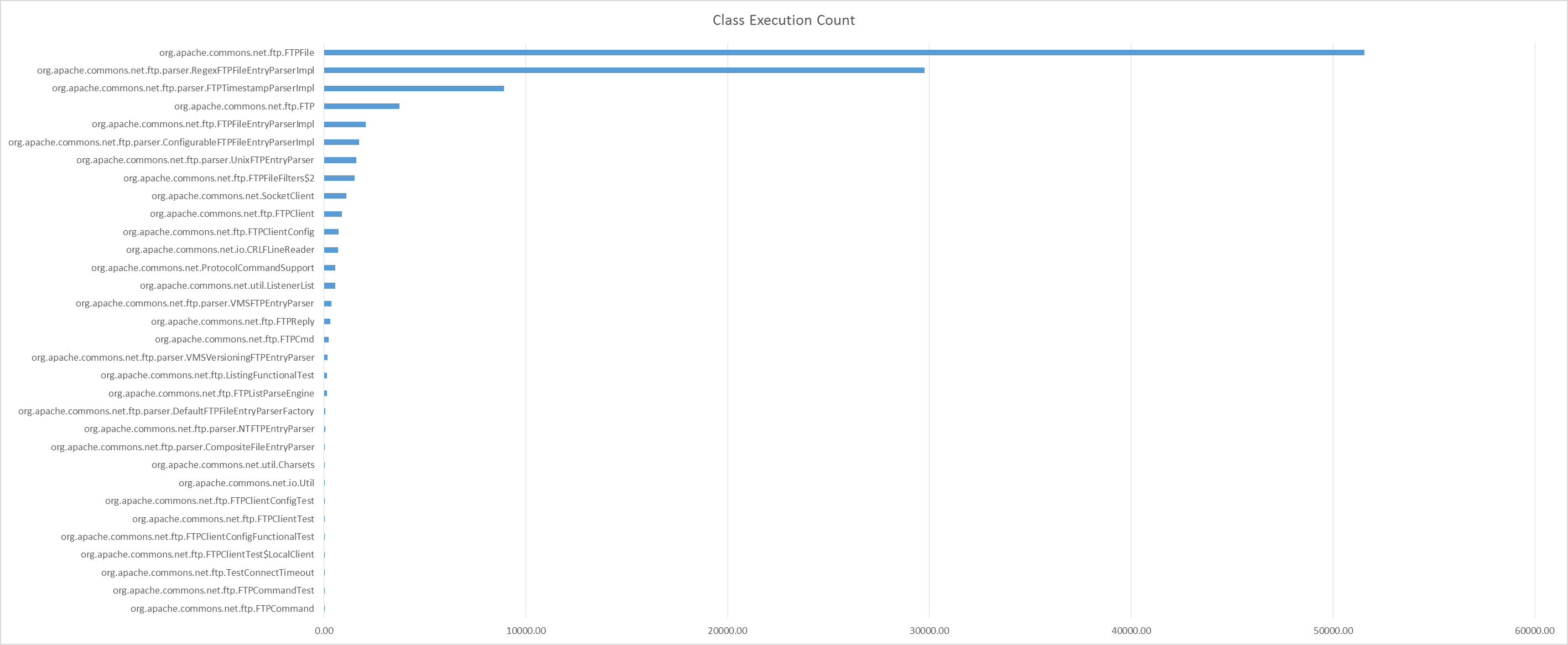
* Method Execution Times (The time it took for a method to execute in nano seconds)
* Class Execution Times (The time it took for a class to execute in nano seconds)
* Method Execution Count (The amount of times a method was called/executed)
* Class Execution Count (The amount of times a method was called/executed)

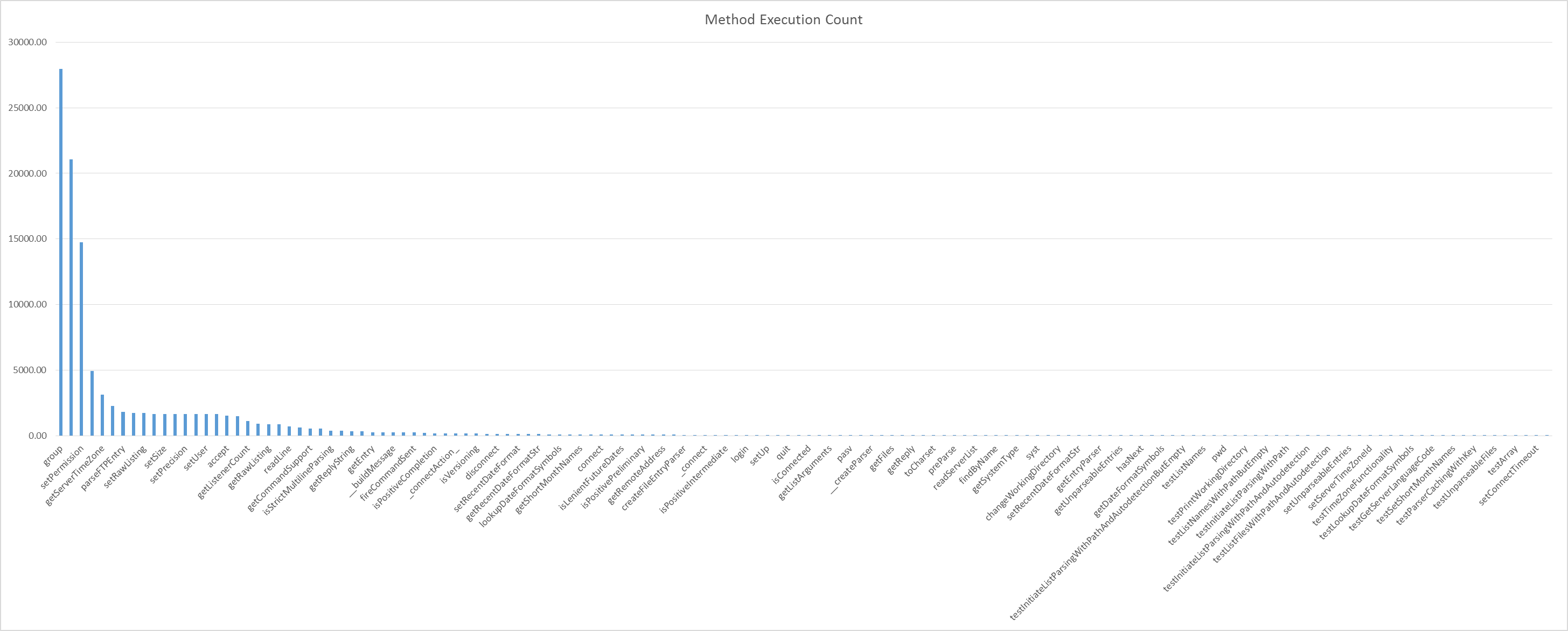


**Graph 1.1 – Class Execution Times**

****

**Graph 1.2 – Method Execution Times**

**Graph 1.3 – Class Execution Count**

**Graph 1.4 – Method Execution Count**

**Tdfg**

Graph Analysis

For Graph 1.1 – Class Execution Times

The classes with the highest executions times are –

Graph 1.2 – Method Execution Times

Graph 1.3 – Class Execution Count

Graph 1.4 – Method Execution Count