CS 409 Software Architecture and Design
Assignment 1
Fraser Beaton

Overview of Problem

The aim of this assignment was build a program which used static and or dynamic analysis that works on java systems. In order to achieve this the use of the framework Javaparser was required. The program I decided to build was a code/test coverage tool which could return useful statistics on how much of the code was covered and which lines were executed. The process of achieving this is mainly to take a source code file and parse it, inserting statements (in the case of my program method calls) at each possible line of execution, where you would then recompile the code and execute the said statements. An important challenge in going with this approach is that when you insert statements, that you maintain the original semantics of the code such that it will compile and not affect the original source code. A trickier alternative approach would be to use bytecode instrumentation where you can then modify the bytecode of the classes before they are loaded into the JVM.

Overview of Design

As stated above my chosen problem was to create a code coverage tool, when building this my design was spilt up key areas into different classes in order to take advantage off OOP and keep the code more maintainable. I have a Parser class which does most of the work in running the main program for inputting chosen files to be parsed along with using the visitor pattern in Javaparser in order to parse different parts of the source code that have been input. A challenge I had with Javaparser was trying to edit blocks of code within say an if statement, to insert the method calls for gaining code coverage stats, is that it would alter the semantics of the code by adding curly braces around each expression I was trying to parse, this in turn would make parts of the code inaccessible and thus not execute later on when being recompiled. To overcome this, I made use of making a recursive method which took advantage of the method declarator visit method where you could edit the block within a statement by looking at the children nodes and applying the recursive function to instrument the code. The method call I add to each line adds information about the line executed to an array list. As I was using an array list I decided to make a Line class which would mean I could create a line object and add that too the list. The line object being added to list would be made up of the file name of the source file being executed, the line number being executed and a Boolean line executed statement being set to false when initially added and then set to true when executed. The last part of the design was to have a class which would deal the logging of execution and correlating statistics on the tests that take place.

Implementation

The program I have produced does a good job of parsing a wide range of source code, the main program when run takes in a path to a folder where you have the java files you wish to parse, this allows the user not have to parse each file individually as when in a directory the program will just grab all java files and parse them. The main statements that are handled are If, while, do while, switch, for loops, for each loops, expressions and try catch blocks. All of these are handled in different ways by java parser, in my implementation I had to create separate methods in order to parse them properly, this all handled recursively via method which modifies the block of a statement, where it looks in an if else statement for all the listed above statements to then parse. I feel this wasn't the best way this could have been done and if had more time would definitely refactor to more clean solution. Due the approach I took for this task when code is parsed, it adds lines that have been parsed to a list, so when the input file is parsed the program will stop running and when you go to recompile your newly parsed code we lose all the lines that were added to the array list so I couldn't actually analyse what lines of code were executed. To overcome this issue, I decided to use serialisation in order to save the state of the list when parsing is complete and deserialise when the user is ready to recompile the parsed code, thus allowing the program to use the list of possible executable lines to set to true of executed. When Keeping track of which lines have been executed in a particular file, I added in a hash map which would then take a file name as a key and have a incremental counter for each line executed. A feature that I wanted to add but ran out of time was to add in a method call into test files which I use to display the statistics automatically, as at this time I need to add it in manually to the test case which isn't ideal.

Results and evaluation

When testing the system, it handles all the main common statements you could come across, for testing I used examples which had a bit of everything including examples with multiple layers of nesting. Once the program has parsed the users chosen files, the user should just need to run the program of Junit test cases as usual where upon completion the system will provided a line by line print out in the console such that it will show what line number was executed along with the file it was executed in, allowing the user to perhaps follow the execution path of the program their running. Once the test has finished executing in the the program will print out some simple statistics, such as how many lines were executed in each file along with the percentage of code that was executed. Overall I feel it works fairly well and is a good example of how Javaparser can be used to perform dynamic analysis on your source code. Although I do feel that design does slightly bring down the general program from a user's perspective, as in order to perform the analysis on the code it needs to have access to classes in my code coverage program. Another way in which this could be improved is to add in some automation of by using javac commands to automatically run detected Junit test files which would save having to parse and then run the recompiled code separately.