Assignment 3 Report

Subject	CMPT 473

About Project

- Name → **Jaxen**
- GitHub Link → https://github.com/jaxen-xpath/jaxen
- OpenHub Link → https://www.openhub.net/p/jaxen
- SLOC → 32.9k (According to OpenHub)
- Evaluation platform → Linux was the platform we used to perform mutation testing on this project.
- Build time to compile and link an executable from source code. → **1.4s**
- Test suite infrastructure. → The project utilises JUnit being run with Maven.
- Number of tests & execution time for the test suite → **267** tests at **26.5** seconds

Mutation Analysis

Major

> The number of mutants generated

- 7066

> The number of mutants covered by the test suite

- 5033

> The number of mutants killed by the test suite

- 2577

> The number of live mutants

- 2456

> The overall mutation score / adequacy of the test suite

- Mutation Score = Number of mutants killed/ Number of mutants generated = 2577 / 7066 = 36.47%
- Not all of the mutations were tested and therefore could be killed. The score for all mutants that could be killed is :
  - Number of mutants killed / Number of mutants covered = 2577 / 5033 = 51.2%

PIT

> The reported line coverage (numerator and denominator)

- 3901/4877 → 80%

> The reported mutation coverage (numerator and denominator)

- 2174/4322 → 50%

Q/A

> Does killed + live = covered or ... = generated? Why or why not? What do the results tell you about your test suite?

- Killed + live for Major equals covered not generated. We initially assumed that the mutants that were generated but not covered were not worth testing by some metric evaluated by Major. Looking into one of the publications from the website confirms this as these mutants do not infect the program state from the view of Major and cannot be strongly killed.

> How do your results from Major and PIT compare? Are the results consistent? Can you determine why?

- If we compare the results from Major and PIT we can see that the results were not consistent. PIT generated 4322 mutants, killed 2174 of them, achieving an overall mutation score of 50%. Major on the other hand, generated 7066 mutants, 2577 of which were killed and 2456 were still alive, achieving an overall mutation score of approximately 36.5%; however, if we calculate the mutation score using the 5033 mutants that were covered by the test suite in the denominator, we obtain a more comparable 51.2% mutation score.
- By the above results, we can clearly see that the difference in the mutation scores is due to the number of mutants generated and covered by the two testing frameworks. Major did generate more mutants than PIT; however, the percentage of which were covered by the test suite showed that the generated mutants were of lower quality and many were not able to be used by the test suite. PIT on the other hand, generated less, but better mutants, allowing more of the mutants to be used in the test suite, and achieving a comparable mutation score.

> Which system did you find easier to use (both integrate and interpret) and why?

- In terms of the level of difficulty regarding integration, PIT was clearly the easier framework to integrate. Using Maven as our build tool, all we had to do was copy and paste a plugin that was provided in the PIT quick start guide and make a few changes to the version numbers in the pom.xml file. Compared to the PIT framework, the Major framework only supports Ant as a build system, requires old Java projects to function properly, and forces the user to use the custom compiler and Ant tool that comes with Major.
- In terms of interpretation, the test results generated by the PIT framework continue to be more user friendly than what is generated by Major. When a PIT test is run, reports are generated for each individual class, the mutants and their status are placed to the left of the original code, and at the bottom of each report is a list of tests used on the mutants; whereas the test results created by the Major framework are separated into many files. The mutants created are put in the mutants.log file, the status of the mutants are put in the **killed.csv** file, and once the user understands the syntax in the mutants.log file, they can use the information to locate where the mutant is used in the original code. Once someone learns to understand the files produced by the Major framework, interpretation is not an issue, but overall, the test results produced by the Major framework is not as straightforward as that produced by PIT.
- Support was also a concern when using Major, the only easily found documentation on the tool is from the manual on the website and related publications. Almost no other support or documentation exists for the tool including troubleshooting help. PIT has more comprehensive documentation including troubleshooting information and has a community around it as help on forums from other users of the tool can be found outside of the main website.

> Does the test suite exhibit weaknesses? How can it be improved?

- With both tools, only roughly 50% of the mutants generated and could be killed were killed. While a number of live mutants may be equivalent to the original code, it's difficult to believe that all of the live mutants are equivalent. So a large number of mutants were not detected by the test suite and that shows a weakness with the suite.

> Does the test suite exhibit strengths? How do you recognize them?

- Strengths can be found in the test suite by examining the results given by the mutation testing tools. PIT also records line coverage of the test suite and found that Jaxen has 80% coverage which is good. Also some packages and classes have better mutation coverage than others. Looking at the PIT report, the function package has great mutation coverage as 360 mutants were generated for it and 93% of the mutants were killed by the test suite.

> Contrast the costs and benefits of mutation analysis and testing versus what you might expect from other techniques.

- One of the clear costs of mutation testing from this assignment is the time commitment needed to leave your system running and wait for results while performing mutation testing. And not all mutations are equally useful to analyze. The description of the individual analysis portion of this assignment has us examining a subset of mutants to determine if any of them are equivalent to the original code. Equivalent mutants do not contribute to the mutation score in a meaningful way as they cannot be killed. There are also other types of mutants such as trivial mutants that will fail for most test cases. One example of a generated trivial mutant was a variable meant to be used as an array index being initialized to -1.

> What was easy? What was difficult?

- From the start, choosing the right project to use in this assignment was relatively difficult. Some projects were less than 10,000 lines of code while others were way over 10,000. Some projects use Gradle instead of Ant which is the only build system Major supports or Maven which has a means of generating an Ant build file. There were also concerns with PIT and how it required a 'green suite' that required all tests to be passing. There were a number of projects we looked at that were compatible with Major but had failing tests or projects with functional test suites that did not work well with Major. Finding a proper project that could satisfy the assignment and tool criterias that also ran within a reasonable amount of time was quite a challenge.
- Once a project was chosen, came the task of integrating Major and PIT. Major had some obstacles to get the project working properly. Jaxen is a maven project so a generated ant build file needed to be created which then needed to be modified in specific ways for Major to run the tests correctly.
- Pit on the other hand, was quite simple to integrate. All we had to do was copy and paste the plugin provided in the PIT quickstart guide into the pom.xml file and make a few changes to the version numbers. Once those changes were made, we let PIT run for a bit more than half an hour and out came the results.

> What obstacles did you face in applying mutation analysis to a real world project, and how did you overcome them?

- These tools were not easy to integrate with the existing project. Both PIT and Major use their own set of custom tools that require changing configurations of the project under test. Even after getting everything working, there may be other obstacles waiting ahead. A different project that we tried using Major on threw a fatal runtime exception that prevented mutation testing from completing. Even with Jaxen, external files were used for testing that the Major compiler could not properly detect. PIT also had issues with the classpath supplied from the base project. All of these issues were overcome with trial and error that eventually let the mutation testing proceed without any other visible problems.

> Do you have any other interesting insights or opinions on the experience?

- Mutation testing using the Major framework is overly time consuming. Although we only had 32.9k lines of code in our project, mutation analysis with Major took over 13 hours. This assignment also shows how troublesome it can be to integrate useful development tools with your own projects as can be seen from the section above.

Individual Analysis

Harman Singh - 303126898

Analysis → isSublang method

For individual analysis, I choose **isSublang** method of **LangFunction**

```
private static boolean isSublang(String sublang, String lang)
    {
        if(sublang.equalsIgnoreCase(lang))
        {
            return true;
        }
        int ll = lang.length();
        return
            sublang.length() > ll &&
            sublang.charAt(ll) == '-' &&
            sublang.substring(0, ll).equalsIgnoreCase(lang);
    }
```

💡 COR → Conditional Operator Replacement

💡 LVR → Literal Value Replacement

💡 ROR → Relational Operator Replacement

1. Log →

   ```
   1435:COR:sublang.equalsIgnoreCase(lang):TRUE:org.jaxen.function.LangFunction@isSublang:182:sublang.equalsIgnoreCase(lang) |
   ```

   - Explanation → Mutation operator used is **COR**. It changed condition **if(sublang.equalsIgnoreCase(lang))** to **true**
   - Result → Status of mutant was **FAIL** therefore mutant was killed
2. Log →

   ```
   1436:COR:sublang.equalsIgnoreCase(lang):FALSE:org.jaxen.function.LangFunction@isSublang:182:sublang.equalsIgnoreCase(lang)
   ```

   - Explanation → Mutation operator used is **COR**. It changed condition **if(sublang.equalsIgnoreCase(lang))** to **false**
   - Result → Status of mutant was **FAIL** therefore mutant was killed
3. Log →

   ```
   1436:COR:sublang.equalsIgnoreCase(lang):FALSE:org.jaxen.function.LangFunction@isSublang:182:sublang.equalsIgnoreCase(lang)
   ```

- Explanation → Mutation operator used is **COR**. It changed condition **if(sublang.equalsIgnoreCase(lang))** to **false**
- Result → Status of mutant was **FAIL** therefore mutant was killed

4. Log →

```
1437:LVR:TRUE:FALSE:org.jaxen.function.LangFunction@isSublang:184:true |==> false
```

- Explanation → Mutation operator used is **LVR**. It changed return value from **true** to **false**
- Result → Status of mutant was **FAIL** therefore mutant was killed

5. Log →

```
1438:ROR:>(int,int):!(int,int):org.jaxen.function.LangFunction@isSublang:188:sublang.length() > ll |==> sublang.length() !=
```

- Explanation → Mutation operator used is **ROR**. It changed condition **if(sublang.length() > ll)** to **if(sublang.length() ≠ ll)**
- Result → Status of mutant was **EXC** therefore mutant was killed

6. Log →

```
1439:ROR:>(int,int):>=(int,int):org.jaxen.function.LangFunction@isSublang:188:sublang.length() > ll |==> sublang.length() >
```

- Explanation → Mutation operator used is **ROR**. It changed condition **if(sublang.length() > ll)** to **if(sublang.length() ≥ ll)**
- Result → Status of mutant was **EXC** therefore mutant was killed

7. Log →

```
1440:ROR:>(int,int):FALSE(int,int):org.jaxen.function.LangFunction@isSublang:188:sublang.length() > ll |==> false
```

- Explanation → Mutation operator used is **ROR**. It changed condition **sublang.length() > ll** to **false**
- Result → Status of mutant was **FAIL** therefore mutant was killed

8. Log →

```
1441:ROR:==(int,int):<=(int,int):org.jaxen.function.LangFunction@isSublang:189:sublang.charAt(ll) == '-' |==> sublang.charA
```

- Explanation → Mutation operator used is **ROR**. It changed condition **sublang.charAt(ll) == '-'** to **sublang.charAt(ll) <= '-'**
- Result → Status of mutant was **LIVE** therefore mutant was not killed

9. Log →

```
1442:ROR:==(int,int):>=(int,int):org.jaxen.function.LangFunction@isSublang:189:sublang.charAt(ll) == '-' |==> sublang.charA
```

- Explanation → Mutation operator used is **ROR**. It changed condition **sublang.charAt(ll) == '-'** to **sublang.charAt(ll) >= '-'**
- Result → Status of mutant was **FAIL** therefore mutant was killed

10. Log →

```
1443:ROR:==(int,int):FALSE(int,int):org.jaxen.function.LangFunction@isSublang:189:sublang.charAt(ll) == '-' |==> false
```

- Explanation → Mutation operator used is **ROR**. It changed condition **sublang.charAt(ll) == '-'** to **false**

- Result → Status of mutant was **FAIL** therefore mutant was killed

Number of Mutants killed → 9

Number of mutants live → 1

Mutation score = 9/10

Mutant 8 was not killed. It checks for last character of string. Since mutant replace **==** with **≤**, any string with character whose **ASCII** value is less that of '-'(45) will pass the test. Therefore having test string whose last character has **ASCII** value more than 45 will kill the mutant.