SWE2721 Lab 13: Mutation Testing

To be completed with a lab partner

1 Objectives

1. Investigate the quality of existing test suites using Mutation testing
2. Analyze existing Testng test case effectiveness using the PIT Mutation testing tool
3. Critique the implementation of a method based upon it's ability to be easily tested.

2 Introduction

Mutation testing is an effective method that can be used to evaluate the effectiveness of unit test cases. The general idea is to modify the source code in small ways. These, so-called mutations, are based on well-defined mutation operators that either mimic typical user mistakes or force the creation of valuable tests. Figure 1 provides a graphical picture of the process.

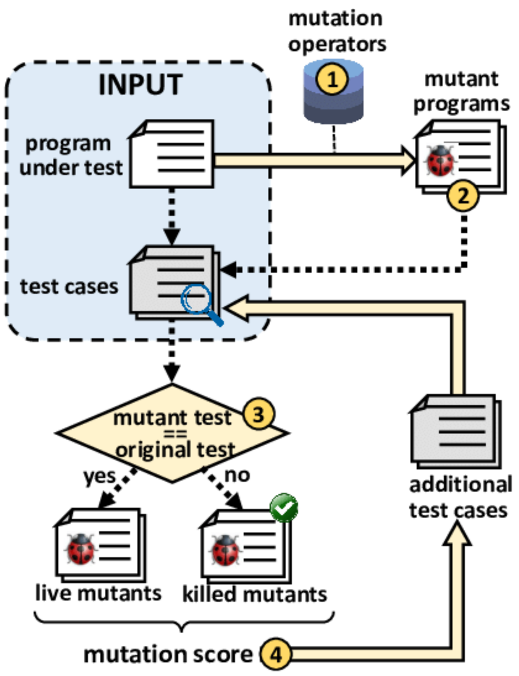


Figure 1 Mutation testing[1]

In this lab, you will use the PIT mutation tool to mutate the source code for a simple project. In doing so, you will see how small changes to the source code can go unnoticed by test suites even if full coverage is obtained.

[1] E. Guerra, J. Sánchez Cuadrado and J. de Lara, "Towards Effective Mutation Testing for ATL," 2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems (MODELS), Munich, Germany, 2019, pp. 78-88, doi: 10.1109/MODELS.2019.00-13. (https://www.researchgate.net/figure/Mutation-testing-process_fig1_337518980)

3 Lab Deliverables

Now that you have completed your lab assignment, tag your final version (which should be on the trunk) with the label LabSubmission1.0. Your code test cases should be thoroughly commented and build cleanly without warnings, and all code should pass the submitted tests.

In addition, submit one report in **pdf format**. The report should contain the following contents:

1. Title Page
   a. Names of all partners
   b. Date
   c. Assignment Title
   d. Repository Link
2. Introduction
   a. What are you trying to accomplish with this lab?
   b. This section shall be written IN YOUR OWN WORDS. DO NOT copy directly from the assignment.
3. Initial state analysis and discussion
   a. Include a screenshot of the code coverage summary for the initial tests
   b. If you were using code coverage as a basis for assessing the quality of these tests, explain whether this is a good quality test set or not. Justify your decision based upon the data shown in the coverage report.
   c. Include a screenshot of the initial Pit Test Coverage report, showing the base line coverage and mutation coverage for each of the classes that are under test. This is not the source code, but rather the summary screen.
   d. Based on your understanding of mutation testing, explain how good you feel the test cases are based upon the output of Pit Test.
4. Final State
   a. Include a summary showing the final code coverage for the project after you have added all of your tests
   b. Include a final screenshot for the Pit Test Coverage report after you have killed as many mutants as you feel you can.
5. Discussion and analysis
   a. For each mutant which is not killed off in your tests, explain whether the mutant is an equivalent mutant or a stubborn mutant. To do this, you can create a table showing the lines of code on the left with line numbers and mutants and then justifying whether the mutant is stubborn or equivalent and why.
6. Class design analysis
   a. For the CompletedCourse class, explain whether the construction of this class makes it easier or harder to mutation test. What did you have to do to achieve the highest mutation score possible for this class?
   b. For the Term class, explain whether the construction of this class makes it easier or harder to mutation test. What did you have to do to achieve the highest mutation score possible for this class?
   c. For the Transcript class, explain whether the construction of this class makes it easier or harder to mutation test. What did you have to do to achieve the highest mutation score possible for this class?
7. Things gone right / Things gone wrong
   a. This section shall discuss the things which went correctly with this experiment as well as the things which posed problems during this lab.
8. Conclusions
   a. What have you learned with this experience?
   b. What improvements can be made in this experience in the future?

Because you are working as a group, only one report and code submission is necessary.

4 Understanding the System Under Test

For this lab, you are being provided with a complete implementation of a project as well as a set of test cases that are designed to verify them. The UML diagram for the code you are being provided with is shown in Figure 2.

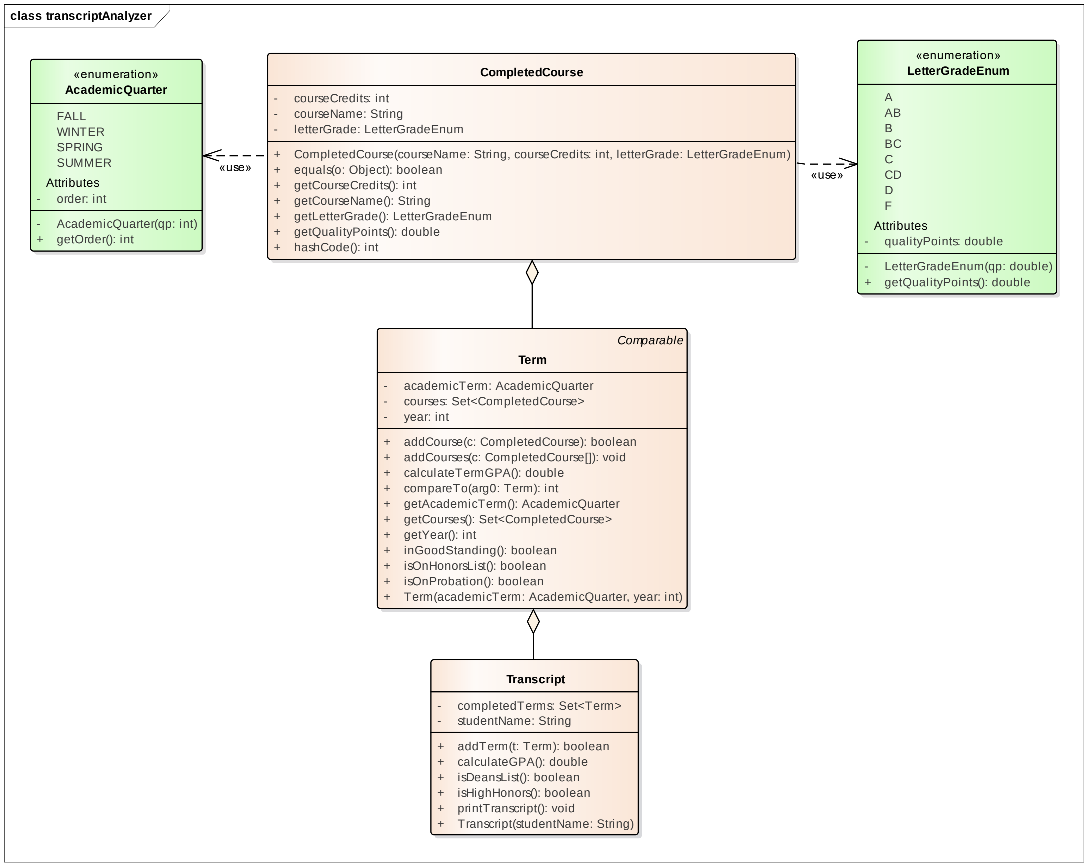

class transcriptAnalyzer
«enumeration»
AcademicQuarter
FALL
WINTER
SPRING
SUMMER
Attributes
- order: int
- AcademicQuarter(qp: int)
+ getOrder(): int
«use»
CompletedCourse
- courseCredits: int
- courseName: String
- letterGrade: LetterGradeEnum
+ CompletedCourse(courseName: String, courseCredits: int, letterGrade: LetterGradeEnum)
+ equals(o: Object): boolean
+ getCourseCredits(): int
+ getCourseName(): String
+ getLetterGrade(): LetterGradeEnum
+ getQualityPoints(): double
+ hashCode(): int
«use»
«enumeration»
LetterGradeEnum
A
AB
B
BC
C
CD
D
F
Attributes
- qualityPoints: double
- LetterGradeEnum(qp: double)
+ getQualityPoints(): double
Comparable
Term
- academicTerm: AcademicQuarter
- courses: Set<CompletedCourse>
- year: int
+ addCourse(c: CompletedCourse): boolean
+ addCourses(c: CompletedCourse[]): void
+ calculateTermGPA(): double
+ compareTo(arg0: Term): int
+ getAcademicTerm(): AcademicQuarter
+ getCourses(): Set<CompletedCourse>
+ getYear(): int
+ inGoodStanding(): boolean
+ isOnHonorsList(): boolean
+ isOnProbation(): boolean
+ Term(academicTerm: AcademicQuarter, year: int)
Transcript
- completedTerms: Set<Term>
- studentName: String
+ addTerm(t: Term): boolean
+ calculateGPA(): double
+ isDeansList(): boolean
+ isHighHonors(): boolean
+ printTranscript(): void
+ Transcript(studentName: String)

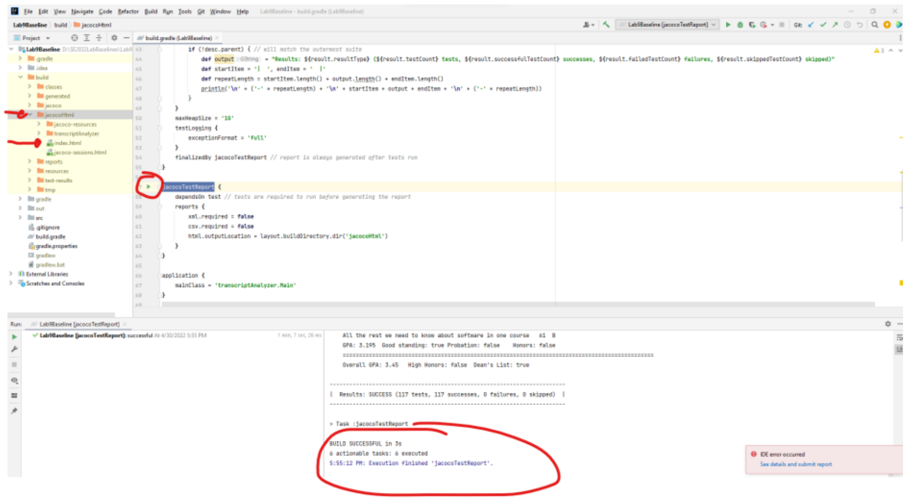
Figure 2 UML for the project

5 Obtaining an Initial Code Coverage Report from Jacoco

With PIT's new Gradle plugin working with the tool is very easy. We simply need to use Gradle to run out tests.

To begin, we want to establish a baseline of code coverage and verify that all tests pass. To do this, open the project in IntelliJ. After it has fully parsed and loaded, open up the build.gradle file. Scroll down to where you see the jacocoTestReport. You should be able to click on a green arrow to the left and run the tests and measure code coverage as is shown in Figure 3.

Once this has finished, you should be able to click right click on the index.html file in the jacocoHtml folder and open it in your browser. This will show you code coverage for the tests provided, as is shown in Figure 4. Keep a copy of the screenshot for your report. Look at the coverage and determine how good you feel the tests are that you are being provided with.

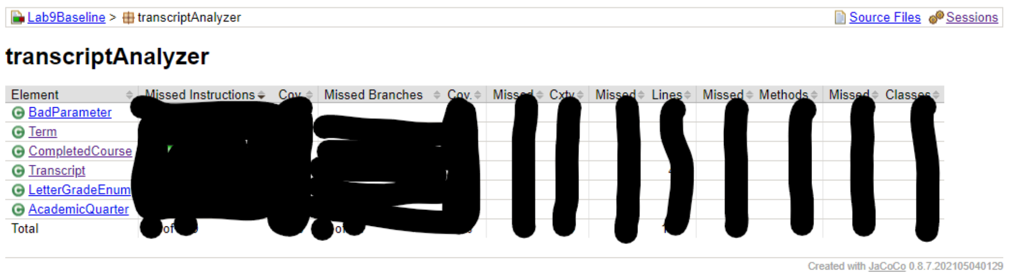


```
        if (!desc.parent) { // will match the outermost suite
            def output = "Results: ${result.resultType} (${result.testCount} tests, ${result.successfulTestCount} successes, ${result.failedTestCount} failures, ${result.skippedTestCount} skipped)"
            def startItem = '|  ', endItem = '  |'
            def repeatLength = startItem.length() + output.length() + endItem.length()
            println('\n' + ('-' * repeatLength) + '\n' + startItem + output + endItem + '\n' + ('-' * repeatLength))
        }
    }
    maxHeapSize = '1G'
    testLogging {
        exceptionFormat = 'full'
    }
    finalizedBy jacocoTestReport // report is always generated after tests run
}

jacocoTestReport {
    dependsOn test // tests are required to run before generating the report
    reports {
        xml.required = false
        csv.required = false
        html.outputLocation = layout.buildDirectory.dir('jacocoHtml')
    }
}

application {
    mainClass = 'transcriptAnalyzer.Main'
}
```

```
    All the rest we need to know about software in one course   61  8
    GPA: 3.195  Good standing: true Probation: false    Honors: false
    ==========================================================================================
    Overall GPA: 3.45   High Honors: false  Dean's List: true

-------------------------------------------------------------------------
|  Results: SUCCESS (117 tests, 117 successes, 0 failures, 0 skipped)  |
-------------------------------------------------------------------------

> Task :jacocoTestReport

BUILD SUCCESSFUL in 3s
6 actionable tasks: 6 executed
5:55:12 PM: Execution finished 'jacocoTestReport'.
```

Lab9Baseline [jacocoTestReport]
Lab9Baseline [jacocoTestReport]: successful At 4/30/2022 5:55 PM
IDE error occurred
See details and submit report

Figure 3 Running the tests and obtaining code coverage information using Gradle settings.



Lab9Baseline > transcriptAnalyzer Source Files Sessions

transcriptAnalyzer

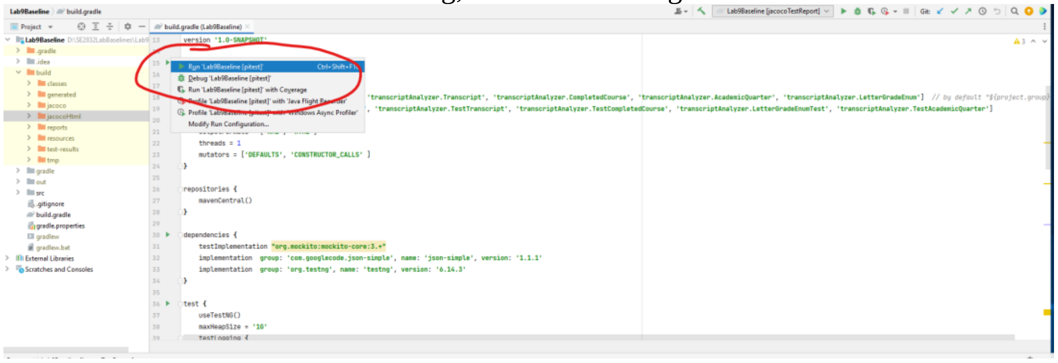
Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
BadParameter	[redacted]	[redacted]	[redacted]	[redacted]	[redacted]	[redacted]	[redacted]	[redacted]	[redacted]	[redacted]	[redacted]	[redacted]
Term	[redacted]	[redacted]	[redacted]	[redacted]	[redacted]	[redacted]	[redacted]	[redacted]	[redacted]	[redacted]	[redacted]	[redacted]
CompletedCourse	[redacted]	[redacted]	[redacted]	[redacted]	[redacted]	[redacted]	[redacted]	[redacted]	[redacted]	[redacted]	[redacted]	[redacted]
Transcript	[redacted]	[redacted]	[redacted]	[redacted]	[redacted]	[redacted]	[redacted]	[redacted]	[redacted]	[redacted]	[redacted]	[redacted]
LetterGradeEnum	[redacted]	[redacted]	[redacted]	[redacted]	[redacted]	[redacted]	[redacted]	[redacted]	[redacted]	[redacted]	[redacted]	[redacted]
AcademicQuarter	[redacted]	[redacted]	[redacted]	[redacted]	[redacted]	[redacted]	[redacted]	[redacted]	[redacted]	[redacted]	[redacted]	[redacted]
Total	[redacted]	[redacted]	[redacted]	[redacted]	[redacted]	[redacted]	[redacted]	[redacted]	[redacted]	[redacted]	[redacted]	[redacted]

Created with JaCoCo 0.8.7.202105040129

Figure 4 Sample Jacoco code coverage report.

6 Determining the Initial Mutation testing State

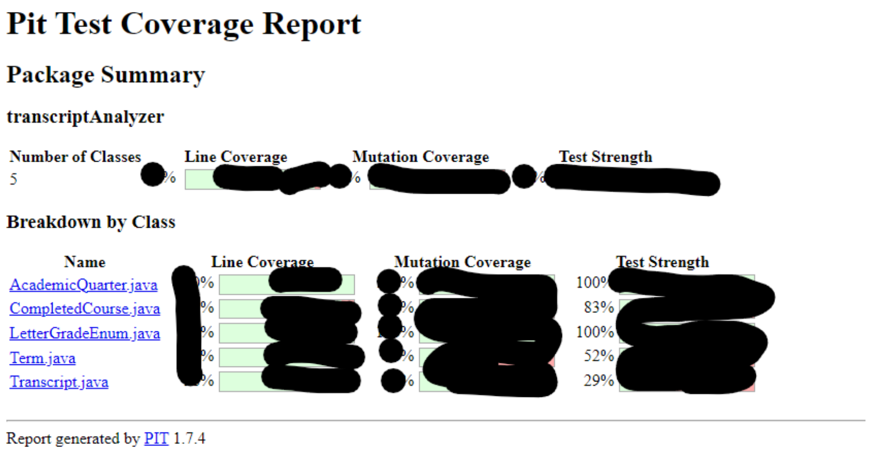
Next, we want to run Pit Test on the baseline test cases that have been provided. To do this, go to the Pit Test entry in the build.gradle file. There should be a green triangle to the left of this entry. Click on it and you should start the mutation tests running, as is shown in Figure 5.


Lab9Baseline
build.gradle
Project
build.gradle (Lab9Baseline)
Lab9Baseline [jacocoTestReport]
Git:
Run 'Lab9Baseline [pitest]' Ctrl+Shift+F10
Debug 'Lab9Baseline [pitest]'
Run 'Lab9Baseline [pitest]' with Coverage
Profile 'Lab9Baseline [pitest]' with 'Java Flight Recorder'
Profile 'Lab9Baseline [pitest]' with 'Windows Async Profiler'
Modify Run Configuration...
.gradle
.idea
build
classes
generated
jacoco
jacocoHtml
reports
resources
test-results
tmp
gradle
out
src
.gitignore
build.gradle
gradle.properties
gradlew
gradlew.bat
External Libraries
Scratches and Consoles

```
version '1.0-SNAPSHOT'

 'transcriptAnalyzer.Transcript', 'transcriptAnalyzer.CompletedCourse', 'transcriptAnalyzer.AcademicQuarter', 'transcriptAnalyzer.LetterGradeEnum'] // by default "${project.group}
', 'transcriptAnalyzer.TestTranscript', 'transcriptAnalyzer.TestCompletedCourse', 'transcriptAnalyzer.LetterGradeEnumTest', 'transcriptAnalyzer.TestAcademicQuarter']

    threads = 1
    mutators = ['DEFAULTS', 'CONSTRUCTOR_CALLS' ]
}

repositories {
    mavenCentral()
}

dependencies {
    testImplementation "org.mockito:mockito-core:3.+"
    implementation group: 'com.googlecode.json-simple', name: 'json-simple', version: '1.1.1'
    implementation group: 'org.testng', name: 'testng', version: '6.14.3'
}

test {
    useTestNG()
    maxHeapSize = '1G'
    testLogging {
```

Figure 5 Starting mutation tests running

When the mutation tests are finished, click on the reports directory in the build folder. You should see a report folder for pit test and a timestamp of when you ran pit test. From there, open the index.html file. You should be able to link to a coverage report, as is shown in Figure 6. Look at the coverage report, and based upon the mutation coverage and mutation scores, determine how good the test cases are for the repository. The line coverage is essentially statement coverage, while the mutation coverage and test strength are based on mutations. Save a copy of this info for your report.



Pit Test Coverage Report

Package Summary

transcriptAnalyzer

Number of Classes	Line Coverage	Mutation Coverage	Test Strength
5	[redacted]	[redacted]	[redacted]

Breakdown by Class

Name	Line Coverage	Mutation Coverage	Test Strength
AcademicQuarter.java	[redacted]	[redacted]	100 [redacted]
CompletedCourse.java	[redacted]	[redacted]	83% [redacted]
LetterGradeEnum.java	[redacted]	[redacted]	100% [redacted]
Term.java	[redacted]	[redacted]	52% [redacted]
Transcript.java	[redacted]	[redacted]	29% [redacted]

Report generated by PIT 1.7.4

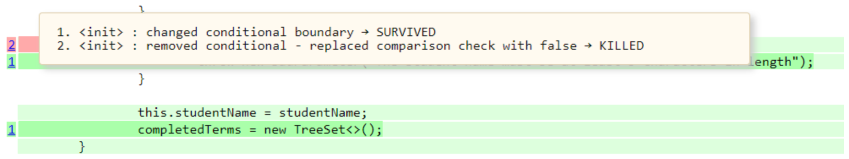
Figure 6 Pit Test Coverage Report

7 Improving the test cases

As has been previously stated, your job in this lab is to try to improve the test harness as is provided using mutation testing to detect mutations which are not killed off by the existing test cases. You should make no changes to the production code, only to the test cases. Continually commit and push code as you make your way through the lab.

To begin with, look at the code coverage. In general, the coverage should be pretty good. There may be some lines which are not covered. A line which is not covered will show up in the report as a pink line without any numbers to the left. Look at the test cases and the code. Without making any changes to the source code for the project and only changing the test cases, can you make the code which is not covered code go away? This may or may not be possible. (Hint: A line of code may or may not be reachable based on how the code is constructed.) There may or may not be a good assertion that detects the mutation that is surviving within the test cases.

Once you have done this, examine each of the mutations that is not killed by the test suite, as is shown in Figure 7. Look at how the method is being tested. Can you create test cases that will kill off the mutants within your source code? Ideally, we would like to see 100% of the mutations killed off, but this may not be possible due to equivalent mutants and stubborn mutants. There should be only a small number of these in the code, but you will need to use critical thinking to truly identify them. If you need an explanation for a mutation, you can visit the site https://pitest.org/quickstart/mutators/ where the mutations are explained in detail.



```
1. <init> : changed conditional boundary → SURVIVED
2. <init> : removed conditional - replaced comparison check with false → KILLED
```

```
2
1                                                      ength");
                }

                this.studentName = studentName;
1               completedTerms = new TreeSet<>();
        }
```

Figure 7 Example live mutation. The top line is pink because one of the two mutations applied to it has not yet been killed by the test suite. The 1 next to the green line at the bottom indicates this mutation was killed off.

Continue adding tests to your test suite, attempting to kill off each mutant. At some point you may not be able to kill off any more mutants. When this is the case, carefully look at the remaining mutants, and determine if they are stubborn or equivalent mutants. Identify this for the report.