1. **Introduction**

Mutation testing is a way of testing where bugs (mutations) are seeded into your program and then the tests are run. If the tests fail then the mutations are killed. If not then the mutations are alive. The quality of your tests can be gauged from the percentage of mutations killed.

The purpose of this lab is to explain why mutation testing is important and should be used as an addition to other software testing methods, and introduce one of the mutation testing tools PIT.

**Analysis Tool**

The tool used in this lab is PIT mutation testing system which has a plugin for Eclipse. It applies a configurable set of mutation operators (or mutators) to the byte code generated by compiling your code.

**System Under Test**

You will be amending a test suite for MinBinaryHeap.java class, a minimum binary heap implementation.

2. **Tasks**

**Task 1: Playing Code Defenders online (~30 minutes)**

To get the general idea of how the Pitest plugin works you will first play a game online.

Go to [http://code-defenders.org/](http://code-defenders.org/) and create a user (takes less than a minute). The TA will create a battleground where half of the students play as defenders and the other half as attackers.

To join the game, go to games -> Open games, scroll down to battlegrounds. The TA will tell you the game ID and you can join as a defender or an attacker.

One of the deskmates should choose the defender role and the other one the attacker role. If you prefer to do your homework alone choose the defender role.

**Attacker:** You will start the game by seeding a bug in the original code. A bug can be one small change at a time (e.g. changing a variable name, changing one operator).

For example, the conditional boundary operator is changed:

```java
52    public void goDown() {
53        if (currentFloor > 0)
54            currentFloor--;  
55    }
```

**Defender:** Your task is to write unit tests to kill the mutants generated by the attackers. When there are no mutants alive, write a test that will kill a possible future mutant. Write at least 1, but not more than 2 assertions, no loops, no new methods, no calls to System.*.

An example of a test:

```java
import org.junit.*;
import static org.junit.Assert.*;

public class TestElevator {
    @Test(timeout = 4000)
    public void test() throws Throwable {
        Elevator e = new Elevator(10, 2);
        e.addRiders(1);
        assertEquals(1, e.getNumRiders());
    }
}
```

To see the changes made by other students simply refresh the page.
Equivalent mutants

It is possible to create a mutant which is identical in functionality to the code, so no test can pass and fail on the mutated class.

For example, the following functions are identical in behavior, they are equivalent:

```java
public void addRiders(int numEntering) {
    if (numRiders + numEntering <= capacity) {
        numRiders = numRiders + numEntering;
    } else {
        numRiders = capacity;
    }
}
```

```java
public void addRiders(int numEntering) {
    if (numRiders + numEntering > capacity) {
        numRiders = capacity;
    } else {
        numRiders = numRiders + numEntering;
    }
}
```

If a defender believes that an attacker's mutant is equivalent, they can click the "Claim Equivalent" button on the mutant. After this, the attacker will see that their mutant was marked as equivalent. If the mutant is equivalent, they should accept it as equivalent.

However, if the mutant isn't equivalent, the attacker can prove that it isn't by writing a test which kills it.

Rules and more information about the game: http://code-defenders.org/help
Task 2: Tool Setup

For Eclipse:
Set up Eclipse IDE, install PITEST plugin from here: http://marketplace.eclipse.org/content/pitclipse and import MinBinaryHeap from the course wiki page: https://courses.cs.ut.ee/2017/SWT2017/spring/Main/LabsPracticeSessions

Use the default settings as shown below:
Window -> Preferences -> Pitest

Default Mutators

For IntelliJ:
Set up IntelliJ IDE and install PIT plugin from https://plugins.jetbrains.com/plugin/7119-pit-mutation-testing-idea-plugin

File -> Settings -> Plugins -> PIT mutation testing Idea plugin and import MinBinaryHeap from the course wiki page: https://courses.cs.ut.ee/2017/SWT2017/spring/Main/LabsPracticeSessions.

Next, move tests to the same folder as main class (src/main). Add a new Run Configuration

Run -> Edit Configurations -> Add new configuration (green plus button)

Source directory should be the same as were the classes are. Click Apply and Ok.

PS! If you for some reason get an exception when running tests (java.lang.NoClassDefFoundError: org/hamcrest/SelfDescribing) then add Hamcrest library to the classpath as shown here (underlined lines):
**Task 3: Amending the test suite**

You are given a test suite with 71% mutation coverage and 94% line coverage. Your task is to write more tests to kill most of the remaining mutants and look for bugs (here we mean the bugs that were in the code before the mutation testing started).

PIT tool can be launched:
Run As -&gt; PIT Mutation Test (**Eclipse**) or
Run PIT (or whatever you named in the set up for **IntelliJ**).

Result should be displayed in the bottom side of IDE.
For **Eclipse** in the **PIT Mutations** tab, coverage report is in **PIT Summary** tab
-&gt; MinBinayHeap.java
For **IntelliJ**, click Open report in browser for mutation coverage report.

*Light green* shows line coverage; *dark green* shows mutation coverage. *Light pink* shows lack of line coverage; *dark pink* shows lack of mutation coverage.

**The goal** is to kill mutants and fix the bugs. Once you find a bug, fix it and continue to look for more bugs.

**For this lab you should submit a ZIP folder containing the following:**

1. **PDF report including:**
   - A list of found bugs and their fixes with explanations.
   - A list of the added test cases.
   - PIT mutation coverage and line coverage statistics
   - Feedback on the lab assignment
   - Bonus: A list of equivalent mutants with explanations (if found)

2. **Fixed code** and amended test suite

Anything that is not mentioned in the file will not be reckoned during the grading process.
3. Grading

You can get up to 10 points for this lab, plus up to 1 bonus point.

The grading is as follows:

1) **1 point for in-lab activity**
   You will be required to demonstrate your progress to the TA during lab time.

2) **Up to 1 point for the list of found bugs and their fixes**
   You will get maximum points for finding two or more bugs with your added test cases. For each listed bug, clearly state what has been corrected, in which code lines the corrections were made, and which test case found the bug.

3) **Up to 1 point for the list of added test cases**
   You will get maximum points for listing all test cases that you added to the test suite. Follow Lab 2 instructions on how to list test cases.

4) **Up to 5 points for improving the test suite**
   You will get points for reporting the mutant and line coverage statistics. You may get up to 5 points for the achieved mutation coverage as follows:
   - >90% killed mutants – 5p
   - >85-90% killed mutants – 4p
   - >80-85% killed mutants – 3p
   - >75-80% killed mutants – 2p
   - 70-75% killed mutants – 1p

5) **Up to 1 point for the fixed code and amended test suite**
   You will get maximum points for including the code of the corrected program and the code of the (amended) test suite.
   IMPORTANT: If the program or test code is missing, or the lab supervisors cannot execute your code, all points you received under point 4 will be cancelled.

6) **Up to 1 point for feedback**
   Try to be constructive and comment on the lab: was is (too) easy, (too) difficult, overall useful? How much time did you spend? How do you think it can be improved? Please be explicit and explain your answers.

7) **Up to 1 bonus point**
   There may be some cases where PIT has generated equivalent mutants. If you find any, point them out and shortly describe why they are equivalent. You get 0.5 points for each correctly identified equivalent mutant but not more than 1 bonus point in total (even if you find more than 2 equivalent mutants).
Mutants are called equivalent when they are behaviorally equivalent to the original program (i.e., the program to which the mutation operation was applied).