

Exercise 3.1:

- 1) **Ease of use.** PIT has an eclipse plugin for easy integration to Eclipse but Javalanche is not integrated with Eclipse. PIT has a much better concise, complete and helpful documentation than Javalanche. PIT is highly active and regularly maintained while Javalanche has only two committers on GitHub and was last updated four years ago. PIT supports build tools such as Maven, Gradle and Ant which can easily be configured for the codebase and integrated into continuous builds processes. Particularly, the plugin installation in Maven is an easy one-step installation process to have PIT up and running. On the other hand, Javalanche requires shell scripts and multiple Ant calls for installation, mutation testing and reporting, and will require more development effort to integrate into a continuous builds environment. It appears that Javalanche was developed for an academic research project while PIT is more critically developed to address mutation testing in a more practical sense.
- 2) **Mutation operators by default.**
 - a. The *activated* Mutator operators for PIT are: conditionals boundary mutator, increments mutator, invert negative mutator, math mutator, negative conditionals mutator, return values mutator, void method calls mutator.
 - b. The *deactivated* Mutator operators for PIT are: constructor calls mutator, inline constant mutator, non-void method calls mutator, remove conditionals mutator, experimental member variable mutator, experimental switch mutator.
 - c. The Mutator operators for Javalanche are: Replace constants, negate jump, arithmetic replace, remove call, replace variable, absolute value, unary operator.
- 3) **Mutation testing strategy and effectiveness.** Javalanche supports the problem of equivalent mutation most mutation tools don't by analysis of dynamic invariants. It looks to be built for the purposes of efficiency by manipulating bytecode directly as a key strategy. Even though, PIT doesn't support equivalent mutation it focuses more on the performance and easy of use. It also uses bytecode manipulation and supports JUnit as well as most Mock frameworks such as JMock, EasyMock, Mockito, PowerMock and JMockit. Javalanche is not known to support any mock frameworks.

I would select to use the PIT mutation tool over Javalanche because it appears more active, stable, well documented, and supported in development communities. It is faster and has an overall better user experience.

Exercise 3.2

Total PIT mutations: 33
Mutants Killed: 30
Mutants Alive: 3

Classes	Mutation Coverage	Percentage (%)
---------	-------------------	----------------

DefaultGameFactory.java	7/7	100
Board.java	7/7	100
Game.java	6/7	86
PointManager.java	7/7	86
Sprite.java	3/5	60

Exercise 3.3

Game.java

[204](#) replaced return of integer sized value with (x == 0 ? 1 : 0) : NO_COVERAGE

The mutant in Line 204 is still alive because no test case exists in the unit test suite that kills the RETURN_VALS_MUTATOR.

Sprite.java

[82](#) mutated return of Object value for org/jpacman/framework/model/Sprite::getSpriteType to (if (x != null) null else throw new RuntimeException) : NO_COVERAGE

[87](#) mutated return of Object value for org/jpacman/framework/model/Sprite::toString to (if (x != null) null else throw new RuntimeException) : NO_COVERAGE

The mutant in Line 82 and 87 is still alive because no test case exists in the unit test suite that kills the RETURN_VALS_MUTATOR.

The mutations not killed by the test suite are weaknesses of the test suite because there have no tests that exercised the line of code where the mutation was created. Additionally, there was no difference in mutation coverage results observed when correct test cases are added and the mutants were killed.

The mutation score of the test suite is the # of killed mutants / (total number of mutants – equivalent mutants). But, since there are no known equivalent mutants in this test run it is the # of killed mutants/total number of mutants which 91%.

Exercise 3.4

Total PIT mutations: 33

Mutants Killed: 33

Mutants Alive: 0

Classes	Mutation Coverage	Percentage (%)
DefaultGameFactory.java	7/7	100
Board.java	7/7	100
Game.java	7/7	100
PointManager.java	7/7	100
Sprite.java	5/5	100

Code Snippets

SpriteTest.java
<pre>@Test public void testGetSpriteType() { assertEquals(SpriteType.OTHER, john.getSpriteType()); }</pre>
SpriteTest.java
<pre>@Test public void testToString() { assertEquals("OTHER occupying [0,0]", john.toString()); }</pre>
GameTest.java
<pre>@Test public void testWon() throws FactoryException { Game g = makePlay("P.. "); g.movePlayer(Direction.RIGHT); assertFalse(g.won()); g.movePlayer(Direction.RIGHT); assertTrue(g.won()); }</pre>

