Software Testing

Lab Assignments

March 31, 2025

Niels Van der Planken
s0191930@ad.ua.ac.be

Thomas Gueutal
s0195095@ad.ua.ac.be

Contents

1 Introduction

This report is a deliverable for one of the testing assignments for the course Software Testing (Master Computer Science 2001WETSWT) at the University of Antwerp in the academic year of 2024-2025.

As per the course description, "*The objective of the lab work of the Software Testing course is to help you learn how you can apply the various testing techniques and test design patterns as discussed during the lectures in practice. You will apply these techniques to a simple Pacman system written in Java. The amount of coding that needs to be done is relatively small: The focus is on testing.*".

This report was written in **Typst**.

2 Division of Tasks

The deliverables are as follows.

1. Code solutions to the tasks, located at **this** Github repo. Note that we add the source code at the end of an assignment as a separate release to the github repo.
2. This report, discussing the solutions and code.

We briefly discuss the division of tasks. We should strive for a half-half split of work between the two team members.

Member	Tasks
Niels	Exercises 1,2,4,5 (including report)
Thomas	Exercises 6,7,9 (including report)

Table 1: Division of tasks amongst team members.

3 Assignment 05 – Mutation Testing

We repeat a note made at the start of most assignment files.

Important Note: Create an archive for JPacman system after performing all the exercises that require modifications to the files. Submit it along with your report. Refer to the "Assignments General" document for introductory information.

"*Mutation testing is a technique to measure the quality of a test suite by assessing its fault detection capabilities. Mutation testing* ***starts with a green test suite****, i.e. a test suite in which all the tests pass. First, a faulty version of the software is created by introducing faults into the system (Mutation). This is done by applying a known transformation (Mutation Operator) on a certain part of the code. After generating the faulty version of the software (Mutant), it is passed onto the test suite. If there is an error or* ***failure*** *during the execution of the test suite, the mutant is marked as killed (****Killed Mutant****). If* ***all tests pass****, it means that the test suite could not catch the fault, and the mutant has survived (****Survived Mutant****).*"

"*Mutation testing allows software engineers to* ***monitor the fault detection capability of a test suite by means of mutation coverage*** *(see Equation 1). If the output of a mutant for all possible inputs is the same as the original program, it is called an equivalent mutant. It is not possible to create a test case that passes for the original program and fails for an equivalent mutant, because the equivalent mutant has the same semantics as the original program. A test suite is said to achieve* ***full mutation test adequacy whenever it can kill all the nonequivalent mutants****, thus reaching a mutation coverage of 100%. Such test suite is called a mutation-adequate test suite.*"

$$Mutation\ Coverage = \frac{Number\ of\ Killed\ Mutants}{Number\ of\ all\ non\text{-}Equivalent\ Mutants} \tag{1}$$

"*In this assignment we will use* ***LittleDarwin.*** *LittleDarwin is a mutation testing tool to provide mutation testing within a continuous integration environment. It is designed to have a loose coupling with the test infrastructure, instead relying on the build system to run the test suite.*"

3.1 Exercise 1

Question: Use LittleDarwin to analyse the original version of JPacMan. LittleDarwin and its manual can be seen and downloaded from https://github.com/aliparsai/LittleDarwin. Explain the results. Now, use LittleDarwin to analyse your version of JPacMan. What differences can you see?

3.1.1 LittleDarwin – Setup

Codeblock 1 specifies the expected file structure for running LittleDarwin, both in the case of the original and modified versions of the JPacman source code. **Assume that the current directory from which we run CLI commands contains the shown file structure**.

Codeblock 1 specifies the steps required to set up and run LittleDarwin for the original version of JPacman. Recall that the original version of JPacman was provided as a `JPackman.zip` on the blackboard page of this course. Now, suppose `JPackman.zip` has already been downloaded and is available in the current directory.

Note: LittleDarwin cautions that running it may unintentionally modify the source code that you run it against, if LittleDarwin itself contains a bug. Or, indeed, if you abort LittleDarwin prematurely whilst it is running, this may also result in unintentional changes to the analyzed source code. Thus, it is safest to have a backup of the source code you want to analyze with mutation testing. In our case, we can simply delete the original `JPacman` version after running LittleDarwin on it, and then start from a fresh copy by uncompressing `JPacman.zip` again.

Throughout the previous assignments we modified the `pom.xml` file. Note step `(2)` in Codeblock 1. We must overwrite the contents of the `pom.xml` file contained in `JPacman.zip` with our most up-to-date version as of this assignment, to ensure the project properly builds an runs.

```
# The assumed file structure.
├── JPacman.zip          # The original JPacman source code.
├── software-testing     # Our modified JPacman source code.
|    └── pom.xml
└── JPacman              # The uncompressed, original JPacman source code.
     ├── LittleDarwinResults/
     └── pom.xml

# The commands to run perform mutation testing for the original JPacman.
#
# (1) Set up JPacman.
$ unzip JPacman.zip
# (2) Manually overwrite `JPacman/pom.xml` by an up-to-date version.
# Here `software-testing` is our own, most recent JPacman version.
$ cp software-testing/pom.xml JPacman/pom.xml
# (3) Set up LittleDarwin and some dependencies.
$ python3 -m venv venv
$ source venv/bin/activate
(venv)$ pip3 install littledarwin setuptools
# (4) Run LittleDarwin for the original JPacman version.
(venv)$ python3 -m littledarwin -m -b \
    -p JPacman/src/main/java/jpacman/ \
    -t JPacman/ \
    -c mvn,clean,test
```

Codeblock 1: The commands use to set up and run LittleDarwin for the original JPacman version.

3.1.2 LittleDarwin – Execution

In this section we discuss running LittleDarwin on the original JPacman source code VS running it on our up-to-date version of the code.

For the commands to run LittleDarwin for the original JPacman version see Codeblock 1, and see Codeblock 2 for the commands to do so for our modified JPacman version. When executing LittleDarwin, its results appear in the `<build>/LittleDarwinResults/` directory, where `<build>` is the directory passed to LittleDarwin via the `-t` option.

See Figure 2 for the mutation coverage results for the **original JPacman version**: $57,9\%$ mutation coverage.

See Figure 3 for the mutation coverage results for our **modified JPacman version**: $67,6\%$ mutation coverage.

Even though the modified version has been extended with new features and functions, leading to a larger number of to-mutate branches, its mutation coverage is still $\approx 10\%$ higher because the test suite has also been extended to match.

The majority of lacking mutation coverage stems from UI classes and Model-View-Controller (MVC) classes that were untouched by the assignments so far. Notably, the classes `controller/BoardViewer.java`, `controller/ImageFactory.java`, `controller/Pacman.java`, `controller/PacmanUI.java` and `controller/RandomMonsterMover.java` account for $\frac{70}{100}$ of mistakenly survived mutants.

Many other classes were changed during the assignments, with a focus on applying various testing techniques to them. This led to an improved mutation coverage for the majority of the lower level classes, and is subsequently the source of the improved mutation coverage.

```
# The assumed file structure.
└── software-testing    # Our modified JPacman source code.
    ├── LittleDarwinResults/
    └── pom.xml

# The commands to run perform mutation testing for our modified JPacman.
#
# (1) Set up LittleDarwin and some dependencies
$ python3 -m venv venv
$ source venv/bin/activate
(venv)$ pip3 install littledarwin setuptools
# (2) Run LittleDarwin for the modified JPacman version
(venv)$ python3 -m littledarwin -m -b \
    -p software-testing/src/main/java/jpacman/ \
    -t software-testing/ \
    -c mvn,clean,test
```

Codeblock 2: The commands use to set up and run LittleDarwin for our modified JPacman version.

3.1.3 LittleDarwin – Problems

In this section we list any difficulties we encountered with using the LittleDarwin tool specifically.

First. For each mutant LittleDarwin generates a `.txt` build output file and a `.java` mutant of the source code. The exact mutation listed at the top of the `.java` file is sometimes incorrect. More specifically, LittleDarwin has problems dealing with multi-line conditionals. See **Table 2** for an example.

<table>
<tr>
<td>The mutation listed by LittleDarwin. at the top of the <code>.java</code> mutant file only lists the first line of the multi-line condition. But, the first line does not contain the mutation!

<pre>/* LittleDarwin generated order-1
mutant
mutant type:
ConditionalOperatorReplacement
----> before: if (bimg == null
----> after: if (bimg == null
----> line number in original
file: 168
----> mutated node: 785
*/</pre></td>
<td>The original source code, before the mutation.

<pre>if (bimg == null
     || bimg.getWidth() != w
     || bimg.getHeight() != h) {
    ...
}</pre>
The source code mutated by LittleDarwin: replace the second <code>||</code> operator by <code>&&</code>.

<pre>if (bimg == null
     || bimg.getWidth() != w
     && bimg.getHeight() != h) {
    ...
}</pre></td>
</tr>
</table>

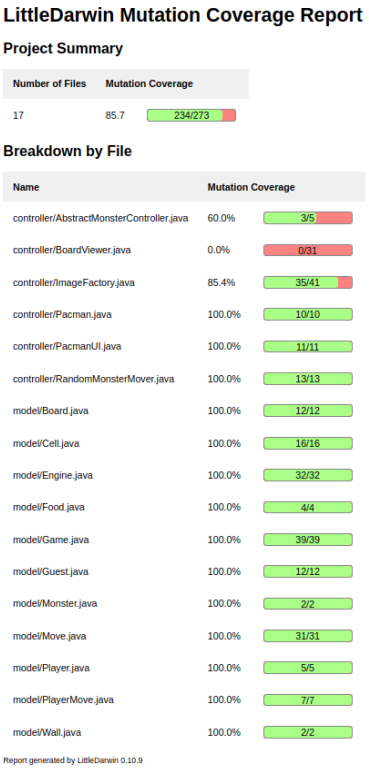
Table 2: LittleDarwin has trouble correctly reporting (displaying) mutations to multi-line conditionals.

3.1.4 LittleDarwin – Headless GUI

This section may be ignored, it provides background information.

We add to this discussion by mentioning the **effect of running the GUI in headless** mode when performing the mutation testing. We do not recommend using `xvfb` for the mutation testing. When we compare the LittleDarwin mutation coverage results when running **with** `xvfb` in Figure 1 VS when **not** running with `xvfb` in Figure 2, we see that `xvfb` massively increases the mutation coverage. Naturally, we would want that using or not using `xvfb` would not affect the mutation coverage at all. Thus, at least for this project, **running with the GUI enabled – i.e. not using `xvfb` – seems essential** because otherwise many mutants mistakenly survive.

More specifically, on Linux the tool `xvfb` can be used to run the GUI in headless mode, which avoids actually rendering the GUI. This can significantly speed up the testing suite, since mutation testing must run the entire suite for every mutant it generates. Note step `(4)` in Codeblock 1. On Linux, an alternate command can be passed for the `-c` option, to disable the GUI visualizations and speed up execution. Instead of `-c mvn,clean,test` specify `-c xvfb-run,mvn,clean,test` iff. the package `xvfb` has been installed, for example through `sudo apt-get install xvfb`.



LittleDarwin Mutation Coverage Report

Project Summary

Number of Files	Mutation Coverage
17	85.7 234/273

Breakdown by File

Name	Mutation Coverage
controller/AbstractMonsterController.java	60.0% 3/5
controller/BoardViewer.java	0.0% 0/31
controller/ImageFactory.java	85.4% 35/41
controller/Pacman.java	100.0% 10/10
controller/PacmanUI.java	100.0% 11/11
controller/RandomMonsterMover.java	100.0% 13/13
model/Board.java	100.0% 12/12
model/Cell.java	100.0% 16/16
model/Engine.java	100.0% 32/32
model/Food.java	100.0% 4/4
model/Game.java	100.0% 39/39
model/Guest.java	100.0% 12/12
model/Monster.java	100.0% 2/2
model/Move.java	100.0% 31/31
model/Player.java	100.0% 5/5
model/PlayerMove.java	100.0% 7/7
model/Wall.java	100.0% 2/2

Report generated by LittleDarwin 0.10.9

Figure 1: The LittleDarwin coverage report for the **original** JPacman version, when called **with** `xvfb-run` to run the GUI in headless mode. See Section 3.1.4 for context.

LittleDarwin Mutation Coverage Report

Project Summary

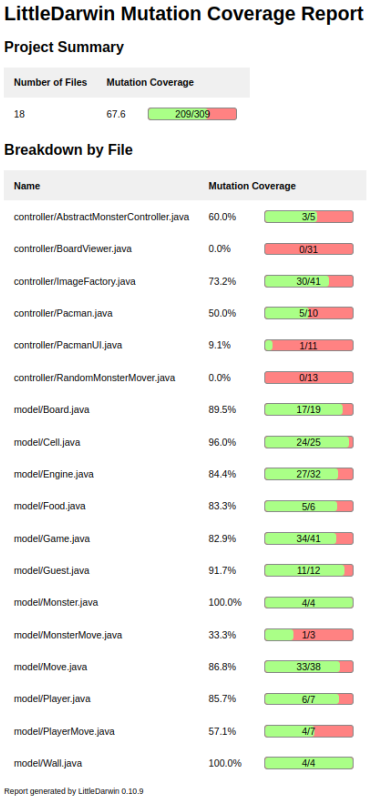
Number of Files	Mutation Coverage
17	57.9 158/273

Breakdown by File

Name	Mutation Coverage
controller/AbstractMonsterController.java	60.0% 3/5
controller/BoardViewer.java	0.0% 0/31
controller/ImageFactory.java	73.2% 30/41
controller/Pacman.java	50.0% 5/10
controller/PacmanUI.java	9.1% 1/11
controller/RandomMonsterMover.java	0.0% 0/13
model/Board.java	83.3% 10/12
model/Cell.java	75.0% 12/16
model/Engine.java	65.6% 21/32
model/Food.java	75.0% 3/4
model/Game.java	79.5% 31/39
model/Guest.java	91.7% 11/12
model/Monster.java	0.0% 0/2
model/Move.java	80.6% 25/31
model/Player.java	20.0% 1/5
model/PlayerMove.java	42.9% 3/7
model/Wall.java	100.0% 2/2

Report generated by LittleDarwin 0.10.9

Figure 2: The LittleDarwin coverage report for the **original** JPacman version, when called **without** `xvfb-run`, so that the GUI is actually enabled during testing.



LittleDarwin Mutation Coverage Report

Project Summary

Number of Files	Mutation Coverage
18	67.6 209/309

Breakdown by File

Name	Mutation Coverage
controller/AbstractMonsterController.java	60.0% 3/5
controller/BoardViewer.java	0.0% 0/31
controller/ImageFactory.java	73.2% 30/41
controller/Pacman.java	50.0% 5/10
controller/PacmanUI.java	9.1% 1/11
controller/RandomMonsterMover.java	0.0% 0/13
model/Board.java	89.5% 17/19
model/Cell.java	96.0% 24/25
model/Engine.java	84.4% 27/32
model/Food.java	83.3% 5/6
model/Game.java	82.9% 34/41
model/Guest.java	91.7% 11/12
model/Monster.java	100.0% 4/4
model/MonsterMove.java	33.3% 1/3
model/Move.java	86.8% 33/38
model/Player.java	85.7% 6/7
model/PlayerMove.java	57.1% 4/7
model/Wall.java	100.0% 4/4

Report generated by LittleDarwin 0.10.9

Figure 3: The LittleDarwin coverage report for our **modified** JPacman version.

3.2 Exercise 2

Question: Repeat the previous exercise, but this time use branch coverage (with JaCoCo) instead of mutation coverage. Look for a class with 100% branch coverage and less than 100% mutation coverage. Why did this happen? Look for similar examples (interesting differences between the coverages of the two techniques) and explain the difference between two results.

3.2.1 Branch VS Mutation Coverage

We simply execute `mvn clean site` for the original and modified versions of JPacman. We note the high level branch coverage. We focus on the coverage of `jpacman.model` over the coverage of `jpacman.controller`, because the assignments until now involved classes in `jpacman.model` only.

- The **original** JPacman version has 53% branch coverage, given total a cyclomatic complexity (Cxty) of $\frac{307}{550}$ missed paths. See Figure 5.
- The **modified** JPacman version has 58% branch coverage, given a total cyclomatic complexity (Cxty) of $\frac{317}{608}$ missed paths. See Figure 6.

There is no class with 100% branch coverage in our testing suite. Still, we provide an answer in the spirit of the question.

Branch coverage means that for each branch – literally a branching instruction in the machine code, a jump instruction – the testing suite contains at least one test that *takes* that branch and at least one test that *does not* take that branch if possible. In other words, all execution paths through the code are covered by at least one test.

Mutation coverage means that the for different ways to modify source code, for example replacing conditional (branching) operator $\leq$ by $<$, the testing suite contains at least one test that *fails* due to the modification. In other words, if the programmer were to incorrectly use $<$ instead of $\leq$ when implementing the given (branching) condition, then the testing suite would catch that fault and fail some test.

Consequently, a class could have 100% branch coverage but not 100% mutation coverage because these two approaches evaluate different subjects (criteria).

- **Branch coverage evaluates** the quality – the lack of faults – of the **source code**, by covering all execution paths through it by implementing an extensive testing suite.
- **Mutation coverage evaluates** the quality of the **testing suite**, by checking how many modifications – how many intentionally introduced faults – go unnoticed by the testing suite.

Some choices of test data in a given test case may not trigger a possible fault, but still cover all branches. Let us refer to the branch coverage example in the Test Design slides of this course, see Codeblock 3. Suppose we choose the following test cases, as was done in the course slides.

- x = null; y = 5
- x = [2,3,5]; y = 3
- x = [2,3,5]; y = 25

Together, these three cases reach 100% branch coverage. But, the fault is not found! Namely, $i > 0$ should actually be $i \geq 0$. So, the mutant that mutates the incorrect condition $i > 0$ into the correct condition $i \geq 0$ that actually matches the specification, would survive. Thus, the mutation coverage is **below** 100%, because a surviving mutant exists! We claim that the testing suite consisting of the three cases given above inadequately validates the specification. An additional test case is needed to reveal the fault and kill all mutants.

- x = [2,3,5]; y = 2.

```java
/**
* Find last index of element
* @param x array to search
* @param y element to look for
* @return last index of y in x, if absent -1
* @throws NullPointerException if x is null
*/
public static int findLast(int [] x, int y)
{
    for (int i=x.length-1; i>0; i--)
        if (x[i] == y)
            return i;
    return -1;
}
```

Codeblock 3: The branch coverage example in the Test Design ppt of the course slides.

3.2.2 Examples

In this subsection we discuss notable examples where branching coverage (Figure 5, Figure 6) and mutation coverage (Figure 2, Figure 3) differ. We focus on our modified version of JPacman, since our test suite is by design of the assignments more complete that of the original JPacman version.

The class `controller.BoardViewer` has 85% branch coverage given a cyclomatic complexity (Cxty) of $\frac{3}{25}$ missed paths and a mutation coverage of 0% given $\frac{0}{31}$ killed mutants. We attribute this to a lack of specific *BoardViewer* class test cases, see Figure 4; this class is only tested by proxy of other test cases that depend on a *BoardViewer* object indirectly. Specialized *BoardViewer* tests are required to cover faults between *BoardViewer* specification and implementation.

We find similar results for other `jpacman.controller` classes due to this lack of dedicated test cases. Figure 4 shows that only the `controller/ImageFactoryTest.java` and `PacmanTest.java` files are dedicated `jpacman.controller` test files.

The `jpacman.model` classes have been the focus of the assignments. Consequently, we find that their mutation coverage numbers are considerably higher, see Table 3. Notably, the branch coverage is often lower than the mutation coverage. The lacking branch coverage can be attributed to the testing suite never trying to fail the pre- and post-condition assertions. Java assert statements still count as branches, but mostly have basic conditions such as a call to an invariant method like `assert invariant()`, which the mutation testing can not easily mutate. So, if there are many simple pre- and post-condition asserts, then mutation testing can generate relatively fewer mutants than the number of branches that branch coverage detects.

Class	Branch Cov.	Mutation Cov.
Game	59%	82, 9%
Engine	64%	84, 4%
Move	63%	86, 8%
Board	67%	89, 5%
Cell	60%	96, 0%
Player	50%	85, 7%
Guest	53%	91, 7%
PlayerMove	50%	57, 1%
Food	50%	83, 3%
MonsterMove	50%	33, 3%
Monster	50%	100%
Wall	50%	100%
MovingGuest	n.a.	n.a.

Table 3: A comparison of `jpacman.model` branch and mutation coverage. We reference Figure 6 and Figure 3 for the coverage percentages.

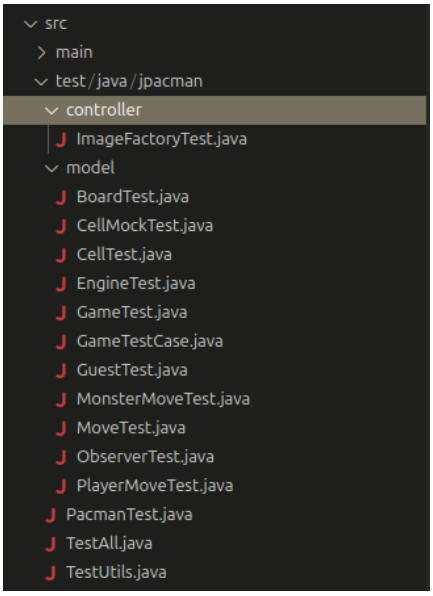


Figure 4: The test directory of our modified JPacman project.

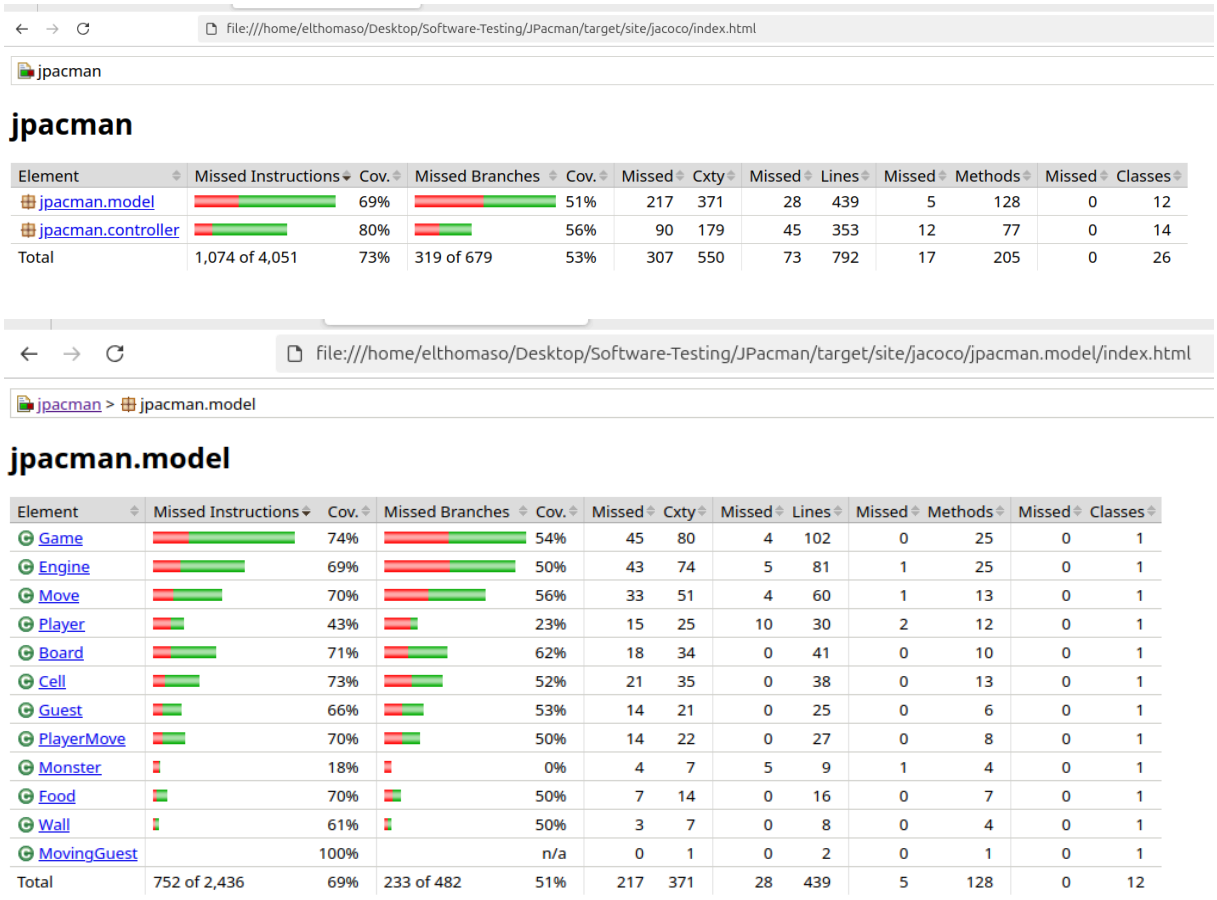


Figure 5: The JaCoCo coverage for the **original** JPacman version.

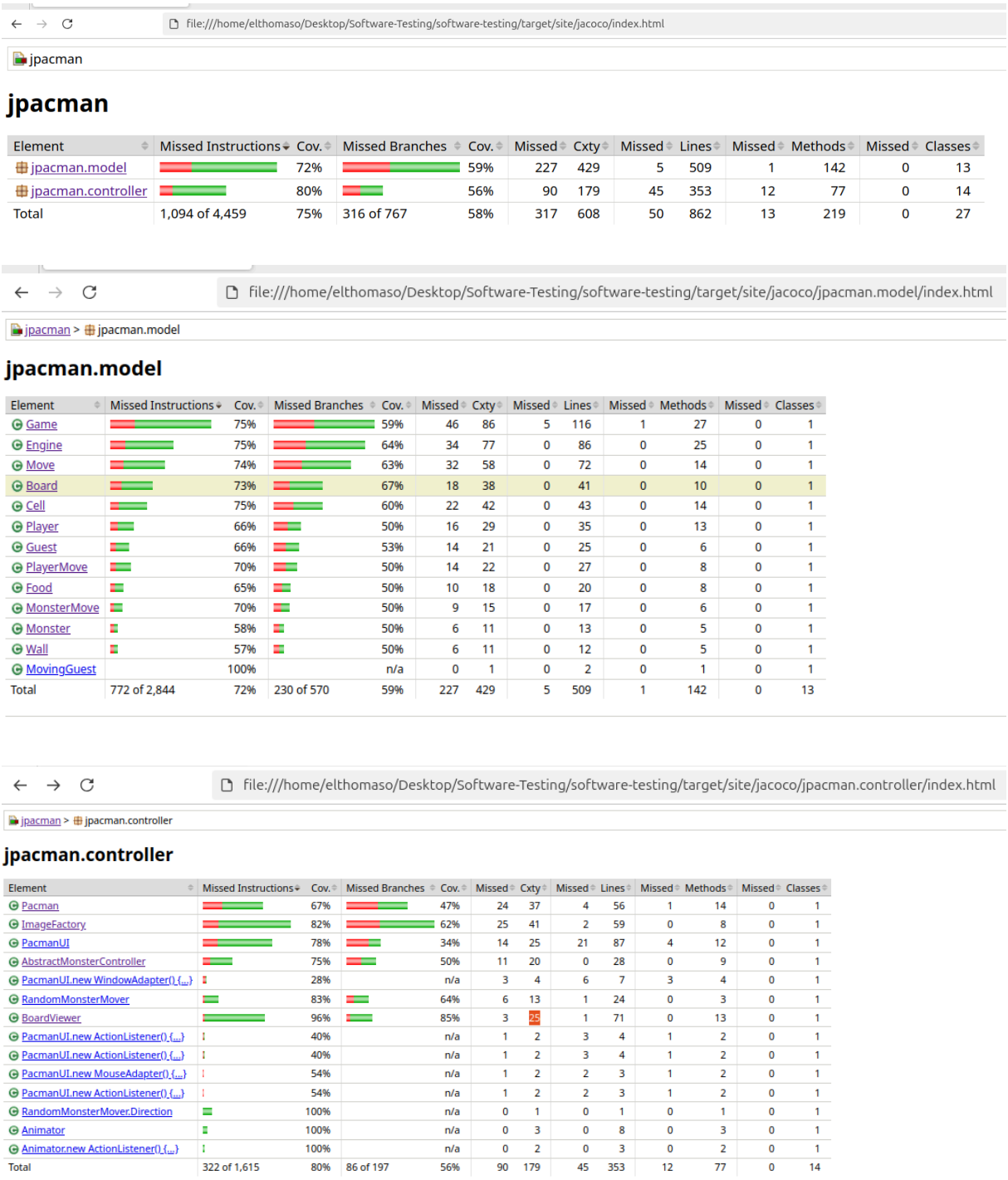


Figure 6: The JaCoCo coverage for the **modified** JPacman version.

3.3 Exercise 4

> **Question:** Find a killed mutant. Explain why, where, and how it was killed?

See Figure 7 for the mutation report for the `jpacman.controller.Pacman` class. Mutant 16 is listed as killed.

We inspect the mutated source code in `16.java` for the exact mutation, see Codeblock 5. A single operator was replaced: `theViewer != null` was changed to `theViewer == null` inside the *Pacman* class invariant method, see Codeblock 4.

The related maven build output found in `16.txt` reveals which test case failed in response to the mutation, killing the mutant. The singular test case in the *PacmanTest* class located at `src/test/java/jpacman/PacmanTest.java` is the only failing test case. Its setup method invokes

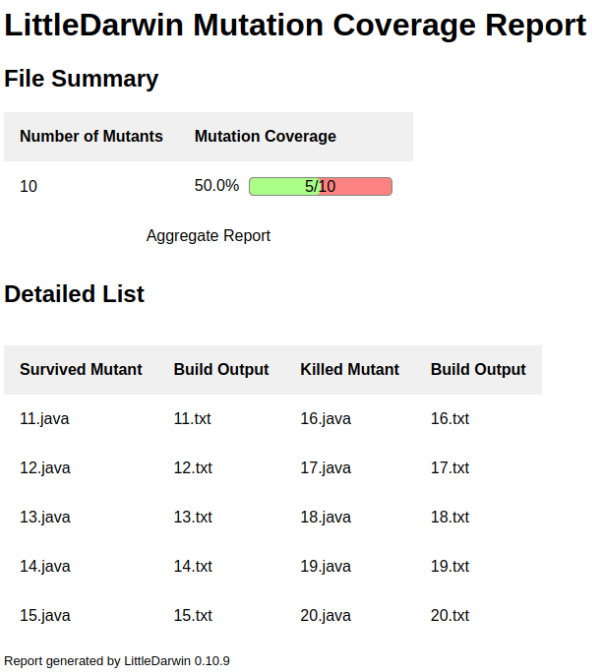
the *Pacman* class constructor, which obviously asserts that the `Pacman.invariant()` method holds as a post-condition. The mutation causes this assertion to fail, since `theViewer` was defined to not be `null` in the test case.

```
// The original invariant method.
protected boolean invariant() {
    return theEngine != null && monsterTicker != null && theViewer != null;
}

// The mutated invariant method.
protected boolean invariant() {
    return theEngine != null && monsterTicker != null && theViewer == null;
}
```

Codeblock 4: The original and mutated versions of the `jpacman.controller.Pacman` class invariant method.

```
/* LittleDarwin generated order-1 mutant
mutant type: RelationalOperatorReplacement
----> before:         return theEngine != null && monsterTicker != null &&
theViewer != null;
----> after:         return theEngine != null && monsterTicker != null &&
theViewer == null;
----> line number in original file: 88
----> mutated node: 551

*/
```

Codeblock 5: The mutation applied in killed mutant `16.java` listed in Figure 7.



LittleDarwin Mutation Coverage Report

File Summary

Number of Mutants	Mutation Coverage
10	50.0% 5/10

Aggregate Report

Detailed List

Survived Mutant	Build Output	Killed Mutant	Build Output
11.java	11.txt	16.java	16.txt
12.java	12.txt	17.java	17.txt
13.java	13.txt	18.java	18.txt
14.java	14.txt	19.java	19.txt
15.java	15.txt	20.java	20.txt

Report generated by LittleDarwin 0.10.9

Figure 7: The LittleDarwin mutation coverage report for `jpacman.controller.Pacman`.

3.4 Exercise 5

Question: Take a survived mutant for class Engine, and write a test that kills it. Repeat the process until all survived mutants from the Engine class are covered. Rerun LittleDarwin to confirm.

Note that you can run LittleDarwin for specific classes/files. This drastically cuts down on run time.

Naturally, we will solve this question for our modified version of JPacman, **not** for the original (unmodified) version of JPacman.

We did not get the `--whitelist` CLI option of LittleDarwin to work. So, we have to run the entire mutation suite to produce the mutation report for *Engine*.

Figure 8 presents the initial LittleDarwin mutation coverage results of `jpacman.model.Engine`, before we extend the test suite to solve this question. There are still five surviving mutants, which we should kill by adding test cases or extending existing cases.

Note that the *Engine* class encodes a state machine. The functions that implement (represent) the state machine events are the interface for affecting state changes on the state machine. The event functions implicitly implement a state-event transition table. Thus, by definition, the event functions check in which (origin) state the state machine currently is before they decide which (target) state they should transition to. This means that it is impossible to bring the state machine into an inconsistent or invalid state configuration, assuming the event functions are correctly implemented, unless the *Engine* class explicitly provides a public or protected interface to do so.

Further, note that to kill a mutant, we should write a test case that given specific test data produces an unexpected (incorrect) result when the specific mutation is applied. So, to kill mutants that mutate a class' invariant method, i.e. `Engine.invariant()`, we must be able to invalidate the class state somehow, and then assert that the invariant agrees that the state is invalid. But, by definition, a class invariant must hold immediately after the constructor finishes, and from then onwards the invariant must also hold before and after every *Engine* method invocation. Since the *Engine* class does not define special functions that allow us to invalidate the invariant manually, for testing purposes, it is impossible to invalidate the invariant in our test suite.

Let us now list for each of the five surviving mutants (33, 34, 36, 37 and 47) the exact mutation.

Mutant 33 correctly displays the mutation.
We can not kill this mutant. It modifies a conditional operator of the `Engine.invariant()` method, and *Engine* does not provide an interface to invalidate the invariant.

```
// Mutant 33.
/* LittleDarwin generated order-1 mutant
mutant type: ConditionalOperatorReplacement
----> before:         return oneStateOnly && theGame != null;
----> after:         return oneStateOnly || theGame != null;
----> line number in original file: 99
----> mutated node: 677
*/
```

Mutant 34 does not display the mutation correctly. It actually replaces `... && !(inStartingState() ...)` by `... || !(inStartingState() ...)` in `Engine.invariant()`. We can **not** kill this mutant. It modifies a conditional operator of the `Engine.invariant()` method, and *Engine* does not provide an interface to invalidate the invariant.

```
/* LittleDarwin generated order-1 mutant
mutant type: ConditionalOperatorReplacement
----> before:         boolean oneStateOnly = inStartingState()
----> after:         boolean oneStateOnly = inStartingState()
----> line number in original file: 92
----> mutated node: 1246
*/
```

Mutant 36 does not display the mutation correctly. It actually replaces `... && inDiedState() ...` by `... || inDiedState() ...`. We can **not** kill this mutant. It modifies a conditional operator of the `Engine.invariant()` method, and *Engine* does not provide an interface to invalidate the invariant.

```
/* LittleDarwin generated order-1 mutant
mutant type: ConditionalOperatorReplacement
----> before:         && !(inStartingState() && inPlayingState() &&
inHaltedState()
----> after:         && !(inStartingState() && inPlayingState() &&
inHaltedState()
----> line number in original file: 97
----> mutated node: 1503
*/
```

Mutant 37 correctly displays the mutation. We can **not** kill this mutant. It modifies a conditional operator of the `Engine.invariant()` method, and *Engine* does not provide an interface to invalidate the invariant.

```
/* LittleDarwin generated order-1 mutant
mutant type: ConditionalOperatorReplacement
----> before:         && !(inStartingState() && inPlayingState() &&
inHaltedState()
----> after:         && !(inStartingState() && inPlayingState() ||
inHaltedState()
----> line number in original file: 97
----> mutated node: 1529
*/
```

Mutant 47 correctly displays the mutation. We **can** kill this mutant. It modifies a conditional operator of the `Engine.inStartingState()` method. If we construct a *Game* that has a map with zero *Food*, then the player wins by default, i.e. `Engine.getGame().playerWon() == true`, and at the same time the *Engine* is in the starting state as normal. As a result of the mutation, passing this *Game* to the *Engine* constructor will **not** cause an *AssertionError* to be thrown because the mutation `starting && !(theGame.playerDied() && theGame.playerWon())` erroneously allows the *Engine* to be in this dual state configuration.

The test that kills his mutant essentially asserts that in normal circumstances, such a *Game* with zero *Food* must lead to an *AssertionError* when passed to an *Engine* constructor, see **Codeblock 6**.

```
/* LittleDarwin generated order-1 mutant
mutant type: ConditionalOperatorReplacement
----> before:         return starting && !(theGame.playerDied() ||
```

```
theGame.playerWon());
----> after:         return starting && !(theGame.playerDied() &&
theGame.playerWon());
----> line number in original file: 39
----> mutated node: 1225

*/
```

```
/** Guard against an Engine.invariant mutation. */
@Test public void testMutatedInStartingState() {
    String[] NO_FOOD_MAP = new String[]{
        "P"
    };
    Game noFoodGame = new Game(NO_FOOD_MAP);

    assertTrue(throwsAssertion(() -> {
        Engine noFoodEngine = new Engine(noFoodGame);
    }));
}
```

Codeblock 6: The test that kills mutant 47. The *Engine* constructor throws an *AssertionError* because the *Game* is both starting AND won at the beginning, which the mutation erroneously allows!

LittleDarwin Mutation Coverage Report

File Summary

Number of Mutants	Mutation Coverage
32	84.4% 27/32

Aggregate Report

Detailed List

Survived Mutant	Build Output	Killed Mutant	Build Output
33.java	33.txt	35.java	35.txt
34.java	34.txt	38.java	38.txt
36.java	36.txt	39.java	39.txt
37.java	37.txt	40.java	40.txt
47.java	47.txt	41.java	41.txt
		42.java	42.txt
		43.java	43.txt
		44.java	44.txt
		45.java	45.txt
		46.java	46.txt
		48.java	48.txt
		49.java	49.txt
		50.java	50.txt
		51.java	51.txt
		52.java	52.txt
		53.java	53.txt
		54.java	54.txt
		55.java	55.txt
		56.java	56.txt
		57.java	57.txt
		58.java	58.txt
		59.java	59.txt
		60.java	60.txt
		61.java	61.txt
		62.java	62.txt
		63.java	63.txt
		64.java	64.txt

Report generated by LittleDarwin 0.10.9

Figure 8: The initial LittleDarwin mutation report for `jpacman.model.Engine`, **before** we extend the test suite.

3.5 Exercise 6

> **Question:** Can you find an example of an equivalent mutant? How do you know (proof) it is equivalent?

We identified an example of an equivalent mutant in the `jpacman.model.Direction` class. The mutant was introduced by *LittleDarwin* through a transformation of a logical condition:
Original Code:

```
return dx == other.dx && dy == other.dy;
```

Mutated Code:

```
return dx == other.dx && !(dy != other.dy);
```

This mutation modifies the second part of the condition from `dy == other.dy` to `!(dy != other.dy)`. Despite the syntactic change, this mutation does not alter the program's semantics. Logically, the expressions `a == b` and `! (a != b)` are equivalent for all values of `a` and `b`. Therefore, the mutated code will behave identically to the original across all possible inputs.

We verified the equivalence by attempting to construct test cases with various combinations of `dx` and `dy` values, including edge cases such as zero, negative values, and large integers. In all scenarios, the test suite produced the same result for both the original and mutated versions of the code.

Because it is impossible to write a test that fails for the mutant while passing for the original (or vice versa), we can conclude that this mutant is **semantically equivalent** to the original code. As such, it cannot be killed by any test case and is classified as an equivalent mutant.

3.6 Exercise 7

> **Question:** Make a mutation-adequate test suite for the model package of JPacMan. How many tests did you have to write? (Count them!) How many equivalent mutants did you find? Explain why each of them is equivalent.
>
> *Note that you can run LittleDarwin for specific classes/files. This drastically cuts down on run time.*

A test suite is **mutation-adequate** when:

- It kills all non-equivalent mutants
- Mutation coverage = 100% (except for equivalent mutants)

This means that our task is:

- To run mutation testing on the `model` package
- To write or improve tests until all mutants are killed (except for the ones we can prove are equivalent)
- To count and document our test cases and equivalent mutants.

To start this exercise, we first ran *LittleDarwin* on the `model` package using following code:

```
(venv)$ python3 -m littledarwin -m -b \
  -p software-testing/src/main/java/jpacman/model/ \
  -t software-testing/ \
  -c mvn,clean,test
```

For each `java` file, a list is given below of the mutations that survived:

1. `Board.java` : 2 survived
    - The first mutant that survived is the following:

```
  /* LittleDarwin generated order-1 mutant
mutant type: ConditionalOperatorReplacement
----> before:                 result = result &&
c.getBoard().equals(this);
----> after:                result = result ||
c.getBoard().equals(this);
----> line number in original file: 67
----> mutated node: 918

*/
```

This is not an equivalent mutation since the logical AND operator is switched to the logical OR operation. The effect of this mutation is that the result becomes true more often since only one of the two variables needs to be true. This is a killable mutation. The test itself was not written due to time shortage but would have looked like this:

We set the result initially to `false` and `c.getBoard().equals(this)` is `true`. In the original code, the result remains false; in the mutated version, it incorrectly becomes true, therefor successfully killing the mutant.

```
// result starts as false
    boolean result = false;
    // if mutated, result may become true wrongly
    result = result && c.getBoard().equals(board2); // should remain false

    assertFalse(result); // fails if mutation is present
```

1. `Cell.java` : 1 survived
2. `Engine.java` : 5 survived
3. `Food.java` : 1 survived
4. `Game.java` : 7 survived
5. `Guest.java` : 1 survived
6. `Monster.java` : 0 survived
7. `MonsterMove.java` : 2 survived
8. `Move.java` : 5 survived
9. `Player.java` : 1 survived
10. `PlayerMove.java` : 3 survived
11. `Wall.java` : 0 survived

3.7 Exercise 9

Question: What are the upsides and downsides of mutation testing? Explain your argument.

Mutation testing is a powerful technique for evaluating the quality of a test suite by introducing small, controlled faults (mutants) into the code and observing whether the test suite detects them. While it offers strong insights into test effectiveness, it also comes with certain limitations.

3.7.1 Upsides

1. **High precision in Test Evaluation**
   Mutation testing goes beyond traditional code coverage metrics (like line or branch coverage) by actually evaluating whether the tests can detect behavioral changes. It identifies weak spots in the test suite that might be masked by high coverage numbers.

2. **Improves fault detection capability**
   Since mutation testing simulates actual programming errors (e.g., replacing `==` with `!=`, or `+` with `-`), it provides direct feedback on whether a test suite can catch realistic bugs.
3. **Encourages better test design**
   To achieve high mutation coverage, developers must write more thorough, specification-driven tests, which often leads to a deeper understanding of the software and improved robustness.
4. **Helps identify equivalent code**
   In cases where mutants survive but are equivalent to the original code, this can point to areas where code can be simplified or clarified.

3.7.2 Downsides

1. **High computational cost**
   Each mutant requires a full run of the test suite, leading to significant time and resource consumption, especially in large codebases with many tests. This can be partially mitigated by targeting specific classes or files.
2. **Equivalent mutants are hard to detect**
   Determining whether a surviving mutant is genuinely undetectable (equivalent) or just untested is undecidable in general and requires manual analysis, which is time-consuming.
3. **False sense of security with survived mutants**
   Some mutants may survive not because they are equivalent, but because the tests are not comprehensive. Differentiating between these cases requires careful inspection.
4. **Tool limitations and reporting issues**
   Mutation tools like *LittleDarwin* may occasionally misreport the location or nature of a mutation, especially with complex or multiline conditions, which adds noise to the results and increases analysis effort.
5. **Increase code complexity.**
   Some mutants survive because their mutations are difficult to test in a testing suite. For example, some mutant can mutate the conditional operators involved in a class' invariant method. For the test suite to be able to guard against such faults, the class in question needs to expose an interface that allows the class' invariant to be manually invalidated, so that the test suite may incorporate a test that must fail because it intentionally invalidates the class invariant.

3.7.3 Conclusion

Despite its limitations, mutation testing remains one of the most rigorous methods available for evaluating the fault detection power of a test suite. It complements traditional coverage metrics and should be considered an advanced tool for teams aiming to improve test quality, especially when applied selectively to critical components.

References