This document presents the findings of our investigations into mutation testing and mocking, as detailed in parts 4 and 5 of this project.

In the mutation testing phase, we encountered 2 surviving mutation related to the nested loop boundaries in lines 42 and 43 of the original code:

```
42      for (int i = 0; gridSize; i++) {
43          for (int j = 0; j < gridSize; j++) {
```

We observed that after changing the gridSize our test wouldn't catch the error.

Initial attempts to address this by explicitly verifying the loop indices within the `gridSize` range resulted in the mutation being "KILLED" upon subsequent PIT runs. However, further analysis revealed the need for a more robust and reusable boundary check.

Therefore, we refactored the code by introducing the `isWithinAxis` and `isWithinBounds` methods:

```
42      for (int i = 0; isWithinAxis(i); i++) {
43          for (int j = 0; isWithinAxis(j); j++) {

54  private boolean isWithinAxis(int n) {
55      return n >= 0 && n < gridSize;
56  }

58  private boolean isWithinBounds(Cell cell) {
59      return isWithinAxis(cell.getX()) && isWithinAxis(cell.getY());
60  }
```

This refactoring effectively addressed the mutation issue, resulting in all relevant mutations being "KILLED" by the test suite.

The only exception observed was a "replaced boolean return with true for org/example/GameOfLife::isWithinAxis -> TIMED_OUT" mutation on line 55:

```
54  private boolean isWithinAxis(int n) {
55      return n >= 0 && n < gridSize;
56  }
```

This "TIMED_OUT" result is expected and not considered an issue, as the mutation effectively creates an infinite loop, causing the PIT analysis to exceed its time limit.

In the mocking section of this project, we focused on testing the `Ballot` class. The tests utilise Mockito to simulate `Voter` behavior, isolating the `Ballot`'s logic.

Specifically, the tests cover three scenarios:

- **testMajorityYes**: Verifies that when a majority of voters vote `YES`, the `Ballot` correctly determines the `Result` as `YES` and informs all voters of this

outcome.

- **testMajorityNo**: Verifies that when a majority of voters vote `NO`, the `Ballot` correctly determines the `Result` as `NO` and informs all voters of this outcome.
- **testDraw**: Verifies that when the number of `YES` and `NO` votes are equal, the `Ballot` correctly determines the `Result` as `DRAW` and informs all voters of this outcome.

The use of Mockito's `mock()` and `when()` methods allows us to control the `Voter`'s `vote()` method return value, ensuring comprehensive testing of different voting outcomes. The `verify()` method is used to assert that the `Ballot` correctly informs each `Voter` of the final `Result`.

These tests demonstrate the effective use of mocking to isolate and test the `Ballot` class's behavior, ensuring its robustness and correctness under various voting scenarios.