CMPT 276 Class Project: Phase 03 Report
Team: 06

Game title: Arcade Game
Goal: Collect the rewards, then reach the exit. Don't get caught, and don't let your score drop below zero.

---

1. Introduction

Our project is a 2D grid arcade game. The player starts at the left side of the board and tries to reach the exit on the right. To win, the player must collect all required rewards, avoid enemies and punishments, and then stand on the exit. There are also bonus rewards that are optional but give extra points. The top of the game window shows the score, how many required rewards are left, the elapsed time, and whether the game is RUNNING, WON, or LOST.

Phase 3 is about testing this game. In this phase we wrote JUnit tests for our code, looked at test coverage with JaCoCo, and made a few changes to the production code when tests revealed problems. All production classes are under src/main/java/com/project/team6 and all tests are under src/test/java/com/project/team6 in matching packages. The goal was to get a good mix of unit tests (for individual features) and integration tests (for interactions between parts of the system).

2. Unit tests and covered features

Most of our tests are unit tests that focus on one feature at a time. To make tests predictable, we use a helper class called TestBoards. It builds a fixed 7x7 board with walls on the border, a start cell at (0, 3), and an exit cell at (6, 3). Many tests use this board so that we do not depend on random maze layouts.

Movement rules are covered by BoardMoveTest and BoardStepTest. These tests check that moving into a wall is blocked, moving onto a floor tile works, and moving into a cell with an enemy gives a collision result. The tests use the MoveResult enum values MOVED, BLOCKED, and COLLISION to check the behaviour of Board.step.

Low level cell rules are tested in CellTest and CellEnterableRulesTest. They check which terrain types are walkable, that a cell can have at most one player and one enemy, and that enemies are not allowed to enter START or EXIT cells. These tests make sure the basic rules are correct before we look at more complex situations.

Enemy behaviour is tested in MovingEnemyTest and MovingEnemyCooldownTest. These tests verify that an enemy with a move period does not move every tick, and that when it moves it always chooses a step that reduces the number of up, down, left, or right moves needed to reach the player, while staying inside the board and on walkable terrain.

Board ticks are tested in BoardTickTest. One test moves the player off the start cell, places an enemy two cells away, calls tick, and confirms that the enemy reaches the player and the summary reports that the player was caught. The other test creates a bonus reward with a lifetime of one tick, calls tick, and checks that the bonus disappears from the cell.

Collectibles and scoring are tested in CollectibleAndScoreTest and ScoreboardTest. We place a regular reward, a bonus reward, and a punishment on the board, collect them, and then apply their values to the Scoreboard. The test checks that the final score and the remaining required count match what we expect. ScoreboardTest starts and stops the timer and checks that the formatted time string has the pattern m:ss.

Spawner and generator behaviour also have unit tests.

- SpawnerBonusTest and SpawnerBonusEdgeCasesTest check bonus configuration, delays, and lifetimes, and also that passing zero bonuses completely disables the feature.
- SpawnerPunishmentPathTest checks that punishments never break the path from start to exit and do not block access to regular rewards.

- SpawnerEnemyPlacementTest and SpawnerEnemyTest verify that enemies are not placed directly in front of the start and exit gates and that there is still a path from start to exit when enemy positions are treated as blocked cells.
- SpawnerRegularRewardsTest checks that the spawner throws an exception if we ask for more regular rewards than available free floor cells.

For terrain generation we use BoardGeneratorTest and GeneratorHelperTest. BoardGeneratorTest creates a random board and uses a helper breadth first search to confirm that the start cell can reach the exit using only walkable terrain. GeneratorHelperTest checks that perimeterWalls creates walls along the border and that toTerrainGrid correctly sets START and EXIT positions in the terrain grid.

Finally, runtime classes have unit tests too. GameStateTest checks that GameState starts in the RUNNING status and that calling setWon or setLost both change the status and stop the scoreboard timer, so that the elapsed time no longer increases.

3. Integration tests and interactions

We wrote two main integration tests to cover interactions between several components:

1. FullBuildIntegrationTest creates a complete game board with BoardGenerator and then uses Spawner to place regular rewards, bonus rewards, punishments, and enemies. It checks that the start and exit cells never hold any items, and that every enemy is at least three tiles away from both the start and the exit. This test touches the generator, board constructor, spawner logic, and cell behaviour all at once.

2. GameControllerEndStatesTest focuses on the controller and end game rules. It builds a small board using BoardGenerator, then creates a Scoreboard, GameState, Spawner, GamePanel, and GameController. To test the private method evaluateEndStates, it uses reflection. One test simulates a win by marking all required rewards as collected and moving the player to the exit. After calling evaluateEndStates, it checks that the state is WON and that there is no explosion position on the board. The other test simulates a loss by keeping at least one required reward, making sure the player is not at the exit, and reducing the score below zero. After evaluation, the state should be LOST and the explosion position should match the player position. This confirms the main win and loss conditions.

4. Test automation and build process

All tests use JUnit 5. Our pom.xml declares the JUnit Jupiter dependencies and configures the maven-surefire-plugin so that mvn test compiles and runs the whole test cases automatically. We also added the jacoco-maven-plugin which attaches the coverage agent during tests and then creates an HTML report in target/site/jacoco/index.html.

The project can be built and tested from the team6game directory on the terminal using the following Maven commands:

- **mvn clean compile exec:java** to compile the code and run the game.
- **mvn clean test** to build the project and run all tests.

These steps are described in our updated README so that anyone can follow them without extra setup.

5. Test quality and coverage

To keep test quality reasonable, we tried to follow a few simple rules. Each test class focuses on one area of the code, for example board movement or spawner logic. Test method names describe what they check. Inside each test we try to keep a clear structure: set up the data, call the method under test, and then assert the results.

We also tried to avoid tests that sometimes pass and sometimes fail for no clear reason. Most tests use the fixed 7x7 board to control the layout. Only a few tests use random generation, and those tests check general properties such as reachability, not exact positions. For time-based tests, such as ScoreboardTest and GameStateTest, we

only check the format of the result or that the time stops changing after the game ends, instead of checking exact numbers of milliseconds.

The JaCoCo report shows which parts of the code are executed when the test set runs. The tests currently run code in the main model packages, including the classes that handle the board, cells, positions, enemies, collectibles, and runtime logic, along with helper classes used for board generation and path checking. Some other parts of the project are not executed by the automated tests, such as the App main class, the keyboard bindings in GameController, most of the drawing code in GamePanel, and the BarrierMode.TEXT path that loads the map file from maps/level1.txt. These behaviours are checked by running the application and trying them out in the game.

6. Findings and changes

Across all three phases we made several changes to the game and learned a lot about our own design. In Phase 1 we focused on planning. We decided early to separate the game into a model, a controller, and a view. We also planned the main rules, such as having a single start gate on the left, an exit gate on the right, and different kinds of items on the board. This planning phase helped us avoid mixing user interface code with game logic and made the later phases easier, because we already knew which classes should be responsible for which tasks.

In Phase 2 we implemented most of the game logic. We created Board, Cell, Player, MovingEnemy, Spawner, BoardGenerator, Scoreboard, GameState, and the Swing user interface classes. During this phase we adjusted a lot of details to make the game feel better. For example, we tuned the tick speed, the enemy move period, the number of rewards and punishments, and the random barrier density. We also added quality of life features such as the HUD strip with score and time, image-based rendering in GamePanel, and the explosion effect when the player loses. The choice to keep model code separate from the view let us change the look of the game without touching core logic.

Phase 3 focused on testing, but the work there also pushed us to improve earlier design decisions. The enemy placement tests showed that our first approach could block the only path between start and exit, so we updated Spawner.spawnEnemies to always keep a valid path. Punishment tests led us to keep important paths and regular rewards reachable. Bonus tests helped us clean up the timing and quota rules, and runtime tests led to better coordination between GameState and Scoreboard so that the timer stops when the game ends. These changes improved both gameplay and the internal structure of the code.

We also learned that some parts of a game are more natural to test manually. Drawing code in GamePanel, keyboard bindings in GameController, and different render modes are easier to check by actually running the game and playing a few rounds. Because our core logic is in the model layer, we can safely change the user interface in the future while keeping the same rules underneath.

Overall, the combination of design work in Phase 1, implementation work in Phase 2, and testing work in Phase 3 led to a more stable and understandable game. We now have 34 passing JUnit tests that can be run automatically with Maven, and a game that is more consistent and fair to play.