CMPT 276 Class Project: Phase 04 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. The Game

Our project is a 2D grid-based arcade game. The player starts on 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 tile. A heads-up display at the top shows the current score, how many required rewards are left, the elapsed time, and whether the game is RUNNING, WON, or LOST. When the player loses, an explosion appears on the board at the player position to show where the game ended.

We tried to stay close to the original plan from Phase 1. The main ideas stayed the same. We still have a single player character, moving enemies that chase the player, regular rewards, optional bonus rewards, and negative punishment items. The player still moves one tile at a time using the keyboard. There is still a clear path from a start gate on the west edge to an exit gate on the east edge.

Some details of the implementation changed as we moved through the phases. At first, we thought we would hard code one level layout. Later we added a BoardGenerator that can build different terrain layouts. It supports a random barrier mode, a provided list of barriers, and a text file map. Right now, the main entry point uses the random barrier mode because it gives more replay value. We also added a Spawner that places regular rewards, bonus rewards, punishments, and enemies while keeping the path from start to exit safe.

The final version of the game has more safety checks than we first planned. For example, enemies are not allowed to stand directly in front of the start and exit gates, and punishments are not allowed to block required rewards or the exit. Bonus rewards appear in waves and disappear after a lifetime, which makes the game feel more dynamic. These rules came from later design decisions and from the tests we wrote in Phase 3.

The most important lessons from building the game are about separation of concerns and testing. Keeping model, controller, and view classes separate made it easier to add tests for the model, and it also made it easier to change the graphics without touching the rules. Writing tests showed us several corner cases that we did not think about at first, such as what happens when enemies or punishments block important paths. Fixing those bugs made the game more consistent and fairer.

2. Tutorial and Main Scenarios

We created a separate tutorial for this phase. It is included in our repository and linked from the README. The tutorial explains the controls, the different items, and shows screenshots of common situations. It also gives a short demo of a full run from start to finish. Anyone who wants step by step instructions can follow that tutorial document instead of this brief summary.

<u>Here we give a short overview of how to play:</u>

[1] Use the arrow keys or W, A, S, D to move one tile at a time.
[2] You cannot move into walls or barriers or outside the board.
[3] Collect all regular rewards to reduce the "Required left" counter.
[4] Bonus rewards give extra points but appear only for a short time.
[5] Punishments reduce your score and are not required to win.
[6] Enemies move every few ticks and try to move closer to you.
[7] If an enemy reaches your tile, you lose.
[8] You win when "Required left" is zero, your score is not negative, and you stand on the exit tile.

These are the main scenarios we highlight in the tutorial: exploring the level, collecting required rewards, making choices about chasing bonus rewards, avoiding punishments and enemies, and finally reaching the exit while keeping the score above zero.

3. Build Automation and Artifacts

We use Maven to build, run, test, and package the game. The pom.xml file includes plugins that support compilation, running tests, coverage, and running the game. For Phase 4, we also make sure that users can create a JAR file and generate Javadocs.

From the *team6game* directory, users can run:

- **mvn clean compile exec:java**
  This compiles the code and starts the game window.
- **mvn clean test**
  This compiles the code and runs all JUnit tests from src/test/java. JaCoCo runs during this phase and creates a coverage report at target/site/jacoco/index.html.
- **mvn clean package**
  This creates a runnable JAR file of the game in the *target* directory. Users can run it with **java -jar target/team6game-1.0-SNAPSHOT.jar** as long as they have Java 17 installed.
- **mvn javadoc:javadoc**
  This generates HTML documentation from our Javadoc comments and places it under target/site/apidocs.

The README file explains these commands and shows how to open the coverage and Javadoc reports. It also explains how to switch between image-based rendering and symbol-based rendering in App.java. We describe where the built JAR and Javadocs live so that users can find them easily. As an additional artifact, we also provide a short tutorial/demo video that shows the main features of the game and a short sample playthrough.

Here is the tutorial video link that you can watch to see how the game is played:

[1] Losing scenario: https://www.youtube.com/watch?v=Gktyp7l3N6c
[2] Winning scenario: https://www.youtube.com/shorts/2SCb2KY5Y0M

4. Lessons Learned and Conclusions

Across all four phases we moved from planning, to implementation, to testing, and finally to polishing and documenting the game. During this process we made several changes that improved the design while still staying close to our original UML plan. We kept the Model, View, Controller separation strict, and we separated the board utilities from board generation and item spawning. GameState and Scoreboard were moved out of the board internals, so they only track status and time. In the App class we now let the client control settings such as board size, the number of each item type, enemy speed, and barrier percentage. We also support random placement of barriers, punishments, and enemies while checking that the player can still reach all required rewards and the exit. These changes make the game more flexible and closer to a real reusable library, not just a single hard coded level.

Looking back, a few other lessons stand out.

First, a clear design at the start made later phases easier. Because we knew we wanted a model that did not depend on Swing, we could test most of the logic without worrying about the user interface. This helped a lot in Phase 3 and Phase 4.

Second, writing tests early would have saved even more time. Many bugs in enemy placement, bonus timing, and path safety only appeared when we started writing tests that tried strange scenarios. Once we had those tests, we were more confident when we changed the code.

Third, good build automation helped keep the project under control. Using Maven to compile, run, test, and package the game means that anyone can clone the repository and follow a few simple commands. Adding JAR creation, Javadocs, and coverage reports made the project feel more like a complete software product and not just a single class assignment.

Finally, we learned how to work as a team on a code base that grows over time. Communication, small commits, and clear division of work made a big difference. Each phase built on the previous one, and by the end we had a game that is fun to play, has automated tests, and can be built and shared using standard tools.