CMPT 276 Group 2 | Phase 2 Report

This version was added for the midway-progress deadline.

Overall Approach & Plan

Our overall approach to Phase 2 for our project focuses on creating a functional and maintainable implementation of our game in Java and JavaFX. We aim to create a solid foundation for the game's logic with a focus on our core gameplay mechanics in the first milestone. To establish a clean architecture based on our designs and packages from Phase 1, we will keep our logic decoupled from rendering. The core game classes are written independently of JavaFX, while the rendering layer observes entity states and draws them on the screen.

For the **midway-progress deadline**, we aim to have a working prototype of the game with the following completed:

- Maven project setup fully configured with a *pom.xml* file
- Core packages (*map*, *reward*, *entity*, *core*, & *behaviour*) implemented with working constructors and movement logic
- Basic game mechanics, including:
    - Player movement logic using directional input
    - Entity interactions and their consequences (enemy and player collisions)
    - A minimal working, prototype pathfinding strategy for mobile enemy movement that pursues the player
    - **Roughly 50% of our entire codebase should be implemented**.
- Sufficient code documentation with JavaDoc comments for all implemented classes

For the **final phase 2 deadline**, our game should be fully implemented. After the midway-progress deadline, we will focus on integrating and rendering the user interface with the following completed:

- Integrated JavaFX rendering with completed game logic
- Fully implemented scoring system, win/loss conditions, and game/board loading
- Fine-tuned pathfinding strategy for mobile enemy movement

Adjustments & Modifications

- Type Direction was not defined in UML - now defined in game.core as *enum Direction*, needed by game.core.Game and game.entity (might also be needed by game.behaviour).
- Encapsulation issues in game.core.Game (dot rule/law of demeters), as a result:
    - Added method *collidesWith(Entity e)* in game.entity.Entity
- Added the following methods to game.map.Map for convenience: *isBlocked(), isPassable(), setTile(), createBorder(),* and exit/entry point setters, and overloaded *getTile(int, int)* for cleaner code.
- Added *getPosition()* to game.map.Position that returns a copy rather than the Position object itself, preventing issues where multiple entities share the same position reference.

- Adjusted *isBlocked()* in game.map.Map to treat out-of-bounds positions as blocked walls, making the code more defensive so we don't have to use *inBounds()* before *isBlocked()* every time.
- Changed *totalRewards* and *totalCollected* to be static fields since they need to track all rewards across the game (global tracking).

Management Process & Division of Responsibilities
Throughout implementation we will commit and communicate our respective work often to avoid merge conflicts. We aim to review each other's code after another member commits to provide feedback, look over changes, and resolve issues as early as possible. We used Discord to communicate and a shared document to track tasks and progress towards the midway deadline.

John - setup Maven project structure, implement *game.map* & *game.reward*
Lex - implement *game.core*
Sydney - implement *game.entity*, write-up the initial components of the report for milestone 1
Diar - implement *game.behaviour*

After the midway-progress deadline, we will decide how to further configure the work in rendering and finalizing the complete game implementation.

External Libraries
JavaFX: we chose this library since it is one of the most widely used Java event-driven GUI systems, and its simplicity and support for modular design will help us build a clean 2D interface while keeping our game's logic and rendering layers separate.

Measures for Quality Code
To maintain high code quality, we emphasized clarity, modularity, and consistency throughout implementation. We followed strong object-oriented design principles including encapsulation and inheritance to reduce redundancy and simplify maintenance. Each class consists of descriptive JavaDoc comments and consistent formatting. We carefully reviewed classes to follow the Law of Demeter/dot product rule to reduce coupling.

Challenges (*so far*)
- Maintaining clear and frequent communication with **all** team members.
- Managing classes to have high cohesion and low coupling while also interacting with other packages to meet functionality requirements.
- Maintaining efficient code with low redundancy.