CMPT 276 Group 18 Phase 4 Report

Rodrigo Arce Diaz, Caleb Hengeveld, Alicia Lam, Shiqi Zhong

<u>The Game</u>

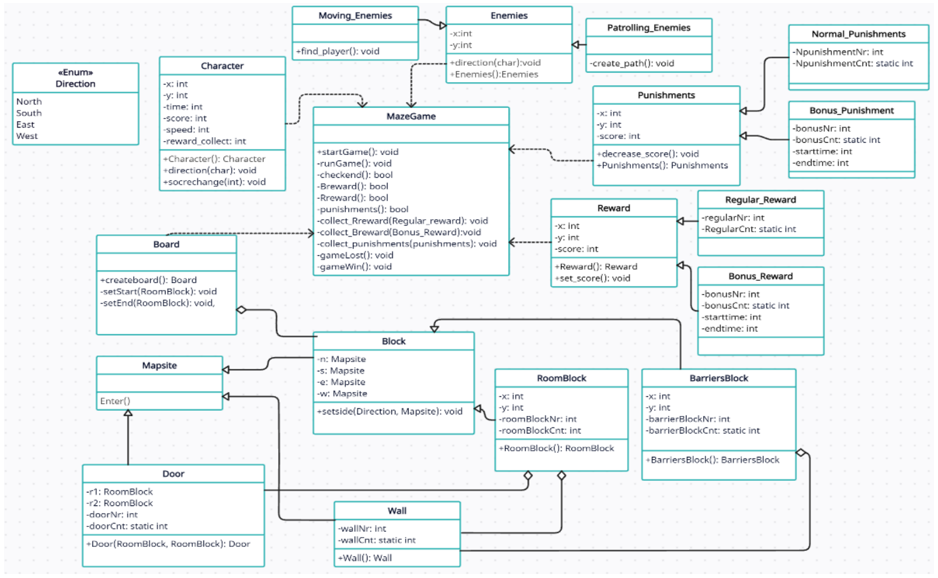
Summary:

In *Jurassic Meteor*, Rock the caveman takes on the perilous mission of ensuring the extinction of the dinosaurs. Rock must race across the Chicxulub Impactor disarming bombs that the dinosaurs have placed to stop the meteor. Throughout his journey, Rock will encounter many prehistoric foes and delicious but potentially dangerous dinosaur eggs. Will Rock be able to stop the dinosaurs and ensure the future of humanity?

In our game, the player plays as a caveman that traverses through a maze. The goal of the player is to collect all the rewards, which are bombs, while trying to avoid being caught by the moving dinosaurs and keeping their score above 0. Bombs provide a score of 10, and all the bombs on the map need to be collected before the player can win the game. Dinosaur eggs are timed bonus rewards, which are not required to be collected, but can add extra points to the player's score.

Entities that can end the game are t-rexes and pterodactyls. Pterodactyls are able to fly over the walls of the maze, however, they do not follow the player and travel only in either an x or y direction. T-rexes follow the player. They can see over the walls to the player but are not smart enough to traverse the maze. The player has to strategize on how to traverse the maze without having the T-rexes on their tail, as it is very difficult to win once the T-rex is on the same path. Static punishments consist of baby dinosaurs that decrease the collected score of the player. Additionally, there are timed bonus punishments, which are aliens that show up on the board at specific intervals of time.

Changes in our Plan & Justifications:
This is the class diagram we had when we started the project.



Because our UML class diagram was created prior to knowledge of our UI framework, we needed to create additional classes to display the game in a window. We added GameScreen, HelpScreen, MainMenuScreen, EndScreen, and ReadyScreen, to contain the attributes and methods of that screen. These classes would extend the ScreenAdapter class and override the render method, allowing the objects such as buttons and the background to be drawn. Additionally, we added draw methods and TextureRegion attributes to the game object classes. These would allow us to draw the blocks, entities, punishments, and rewards onto the screen.

Originally, the MazeGame class contained all methods that would be needed to run the game. We moved these methods into the GameScreen class, and changed MazeGame to contain the shared objects (textures, font, camera, etc.) between the screens. This object would only be instantiated once, at the start of the application, and would be passed to the other screens as they are instantiated.

We also modified the structure of the board objects as well. Instead of thinking of each cell as a room, we decided to treat each cell as a block at a specific coordinate. This would mean that we would not need to keep the Door class, and we could remove the attributes keeping blocks above, below, left, and right of the generic RoomBlock. This made the game much more simpler, as we would not need to set all four sides of each room and just access them through their respective coordinates.

We decided to move Mapsite into Block and Block was made abstract, as Mapsite and Block would now be functioning essentially as the same default board object class. This would make it easier to modularize the subclasses of Block, as these classes would need to override the enter method, and they would also inherit any shared methods/attributes such as draw. This would allow the draw method in our board class, containing all the Blocks, to be able to draw 2D-array of blocks regardless of whether the block was a Wall, RoomBlock, or BarrierBlock.

Additionally, many of the method headers in our UML class had to be changed as our design plan evolved throughout the project. This is due to many methods needing more information to function properly in the system that we had originally expected.

Lessons Learned:

We made many mistakes and learned many lessons over the course of this project. From a technical standpoint, many of us had not extensively used Maven, JUnit, JaCoco, or even Git, and we learned how to use them for external library management, testing, test analysis, and group work, respectively. With respect to project management, we learned how to work in a group with flexible milestones and responsibilities, and learned how thorough design and planning is key to avoiding future headaches

We learned the most about project management during Phase 2, specifically about dividing work and communication. Initially, it was relatively easy to divide the work; each group member would implement a number of the classes in the UML class diagram, and there was no overlap. However, once we started working on the user interface, the tasks were not easily separable from one another and were all in the same few files, inevitably leading to merge conflicts. Furthermore, problems with the initial UML diagram classes were discovered during this stage of Phase 2, and tight schedules and ever-changing requirements meant that we could not each stick to our own corner. To solve this, we communicated frequently in our Discord group chat to re-evaluate milestones and tasks every day, to determine who could work on what when, and were careful to notify everybody when we had pushed new changes to the remote repository.

In Phase 3, we struggled with unit testing classes that used LibGDX; documentation was sparse, online anecdotes were contradictory, and we ultimately were forced to only test limited functionality of our game window classes integrated with other class objects by using runnables. Most online sources concluded that purely unit testing LibGDX applications that used textures, shaders, and other OpenGL methods (which we used in abundance) was not currently supported. This problem was in large part due to our decision of which external libraries to use happened over the course of a few days in the back half of Phase 2, when we were in a hurry and didn't have time to research the library beyond how it served our immediate requirements. Had we begun researching external libraries for UI implementation in the Design phase, we could have factored in its testability when choosing an external library, and sidestepped much of our difficulties with Phase 3.

Another problem that could have been avoided by better planning was the haphazard, on-the-fly nature of the design of our screen classes. Because we were not familiar with UI implementation frameworks, we could not design related classes during the design phase. Once we did pick an external UI implementation library, however, we learned it on the fly while creating the user interface classes. As a result, the screen classes (especially GameScreen) were disorganized, riddled with code smells, and not easy to test, and we spent much of Phase 3 separating the game logic from the UI and reorganizing the code. If we had managed our time in Phase 2 better, we could have briefly redesigned our UML diagram to include the new classes and eliminate most of their problems before Phase 3 even started. In retrospect, both of these roadblocks could have been avoided by better and more frequent design assessments.

Tutorial

For our Tutorial we provide a video highlighting all the features of the game:

https://drive.google.com/file/d/1hlNQjlI5rIi8IMVj6dTT106g71w2itI_/view?usp=sharing