Phase 2 Report

Group Members:
Evan Sarkozi
Kevin Park
Harry Nguyen
Daniel Wang

Our Overall Approach

Our overall approach was to start from the foundation up, prioritizing what would have the greatest ripple effect over the course of the project:

- First, we imported all our required libraries, and ensured a proper test application could compile and run (being fully integrated with Maven).
- Next, we made the base classes for our system (Board and Entity specifically), and focused our efforts on 1) making sure they met all requirements under the hood, and 2) having a means of seeing/checking that it worked correctly.
- Once that foundation was in place, we added subclasses like Obstacle, Enemy, Reward, etcetera, and tried to match the UML diagram to the best of our ability.
- At this point, we worked on the UI, and the algorithm for Enemy movement. Once this was done, we had everything we needed to have the complete lifecycle of the program.
- Tying it all together, we set up state transitions between game start, finish, pauses, etcetera, and set up the triggers for each state (Enemy instances can trigger transition to lose condition, Exit instance triggers win condition, etc.)
- Now that the game could successfully be run from start to finish, and be replayed without issue, we spent some time polishing the game mechanics, adding some new features, and otherwise tidying it up a bit before the deadline.

Modifications From Phase 1

Modification	Justification
Score and rewards collected are kept in player class, rather than in UI class.	A better place for it (i.e., why would game logic depend on values kept in the UI? The UI should probably report game values, not store them)
Bonus rewards do not extend upon *regular rewards*, *enemy* does not extend from *punishment*, and *rewards* and *punishments* do not extend from a "*ScoreModifier*" class.	Most likely an oversight, but was ultimately not that detrimental to the final product. Additionally, we had trouble with changing the Sprite of inherited objects, so it would have caused some trouble.
Added two interfaces that were absent in our initial UML diagram.	Allowed for common rendering/input functionality without requiring a common ancestor (e.g., a game

Phase 2 Report

Group Members:
Evan Sarkozi
Kevin Park
Harry Nguyen
Daniel Wang

---

	entity and a textbox are both rendered, but don't share much in common other than that!)
Introduced a "*DrawableEntity*" class that extends from "*Entity*" class.	There may have been more abstract entities that did not get rendered (e.g., a pathfinding node, or an invisible wall).
Leaderboard is absent from the game (for now).	Time constraints + leaderboard was of medium/low priority as per our use cases.
Only one map size (for now).	Time constraints + not necessarily required
More than one *entity* can occupy a cell at a time (i.e an *enemy* can be on top of a *punishment*).	Done to better fit the specification, which stated that enemies passing over obstacles should not consume them - this implicitly means 2+ entities can be on the same tile at a given time.
Punishments and *rewards* do not have setter methods.	The punishment and reward amount were able to be initialized in their respective constructors, so changing them did not matter all that much.
Punishments and *rewards* do not have getter methods	We were able to allow the player's score to be changed directly from the *punishment* or *reward* class using a method called "OnEntityWasTouched," negating the need for this.
Two types of *enemies* (fast and slow)	Provided more gameplay variety while still meeting specifications.

Division of Labour

Group Member	Role/Responsibilities
Evan Sarkozi	• Core Architecture • Game Mechanics • Some UI ○ Button functionality

Phase 2 Report

Group Members:
Evan Sarkozi
Kevin Park
Harry Nguyen
Daniel Wang

Kevin Park	● UI ○ Layouted elements in UI ○ Implemented system to display game time
Harry Nguyen	● UI ○ Win screen ○ Lose screen ○ Foundation for score/time table ● Some Game Mechanics ○ Set foundation for punishment and reward entities
Daniel Wang	● Path Finding Algorithm ● Leaderboard Implementation

---

External Libraries

For this project, we used the external library libGDX, which provided us with our file I/O, rendering, window management, and input handling solutions. It was recommended most consistently in a search for a good Java game development framework, and has a proven track record of fully fledged commercial games which use it. As libGDX is a comprehensive suite that wrangles all the components required for a game (visuals - in-game and UI, sound, interaction with OS, etcetera.), we did not require any other frameworks to complete this iteration of the software.

Code Quality Control

- Components strived to be as decoupled as possible, with most serving only one function, and only interacting with one or two related ones at most. Additionally, an emphasis was made on hiding methods and data from other components which did not need to access it. This allowed our code to be more of a lasagna than a loose pile of spaghetti (though the layers of lasagna may have some spaghetti baked in).

Phase 2 Report

Group Members:
Evan Sarkozi
Kevin Park
Harry Nguyen
Daniel Wang

- Appropriate and descriptive comments, as well as JavaDoc documentation, were included in the source code to provide adequate explanation of classes, methods, and fields. Unfortunately, there are a few classes that, while relatively straightforward, are still missing proper JavaDocs and comments. Were there to be a later deadline, this would be a priority to fix.
- For much of the code, Hungarian Notation prefixes are used to better clarify between instance/static members and functions.

Design Patterns

In our code, we used several design patterns which helped us out a lot.
A few of the ones incorporated were:

- Singletons (Game, Input/Render Handler)
- Push-Based Observers (IRenderable, IInputListener being observers for render and input events)
- Prototype-esque pattern for MazeBuilder (patterns are supplied at runtime, which it uses to build into its product by cloning them into the grid).

Challenges

- Figuring out how to use LibGDX (the library we chose), and reading its documentation. Much of the documentation for libGDX - while extensive - is lacking in proper explanation, so resorting to forum crawling was a common occurrence.
- Communicating with other groupmates, and ensuring everyone was on the same page. Fortunately, division of work was done well enough that merge conflicts never arose. There were, however, instances where people were unsure of what they should be doing - these were easily resolved, though, as we simply needed to consult our diagrams and use cases to determine what was left to do before we had a product that met specification.
- Time constraints - many assignments and midterm exams took time away from phase 2.
- Becoming familiar with Git (i.e merge conflicts, learning how to properly push, pull, stash, etcetera.)