The MazeGame Phase 2 Report

Overall Approach

Our team adapts the Waterfall model in the development process of The MazeGame. The development flows and iterates through the following phases: communication, planning, modeling, construction. The team first understands the problem and identifies the game's sets of requirements through the communication phase. The team scheduled meetings and set deadlines for the completion of the expected milestones. The MazeGame is further modeled based on the existing UML class diagram and use-cases of the essential classes. After the requirements and solutions to the problems are finalized, the team builds the game by creating the classes using our UML diagrams. The construction of the game also builds automation using Apache Maven for better project management, building, and future testing. The team members then further run tests and give constructive feedback for improvements, ensuring the game's smoothness.

The MazeGame in phase 2 is redesigned based on the game proposal in phase 1, where the inheritance hierarchy of the classes is more distinct compared to what we had from before. The game's design is carefully measured according to the five quality factors of FURPS by Hewlett Packard. The features are further refined with measures of various scenario cases. While keeping the arcade game's essential functionality, the unnecessary features are omitted along in the design stage in phase 2. The MazeGame is implemented using creational design patterns by creating singleton and factory methods to make the design more flexible. Singleton is used for ensuring certain classes - Board, BoardState, ScoreBoard, and TimeCounter - are globally accessible and have one instance per game. Additionally, we also used the factory method for object creations relying on the classes' inheritance property.

For the frame design, we decided to use a grid layout because it fits with how we visualize the Cell class's implementations. Each panel on the frame is the representation of the Cells, i.e. if a Cell contains a Player the corresponding panel will also show the player's image.

The MazeGame Phase 2 Report

Adjustments and Modifications to the Initial Design

The following information are the adjustment and modifications we have made to the design along with justification:

Adjustments and Modifications	Justifications
Applied singleton design pattern to Board, BoardState, ScoreBoard, TimeCounter classes	To ensure these classes only have one instance per game and are globally accessible to other object classes.
Created the abstract class movingCharacter to implement movement of Player and Enemy instead of directly implementing move method inside Player class.	To replace explicit constructor calls with dynamic dispatch. This case we use overriding to change the position of the player and enemies.
Incorporated getters/setters inside Board class that obtains/set attributes of other objects.	Ensure encapsulation to hide implementation details from users.
Reward, Punishment, Cell, MovingCharacter classes inherit from abstract Position class. The Position class is separated from the Cell class, and Cell checks the condition of different class objects at different positions.	Ensure Position is designed to be subclassed and cannot be instantiated. Enables more flexible condition checking for objects at different cell positions.
Instead of creating barrier/wall classes, we Implemented wall/barrier classes inside boardstate class by setting its cell condition.	Eliminate redundant implementation of classes.
Added GUI screen classes to display ending scenarios for players.	To enhance user interface by altering the design to user-centric design that potentially enhances user's visual perception of the game.
Incorporated generic algorithms (java generic methods) for rewards and punishments inside the BoardState class	Eliminate casts and stronger type check at compile time
Incorporated GUI libraries for graphical display and player interactions with the game.	GUI components were disregarded in the previous UML class diagram due to insufficient knowledge of the GUI libraries.

Randomly generated walls, rewards and punishments at every initialize instead of loading pre-build maze every initialize.	Every game would be distinct and there would be no need of adding a pre-build maze to play in a different maze
Added methods that allow the Enemy class to interact with reward and punishment.	This is the solution for when there is more than 1 object in a cell, so if it occurs that the player is going to a cell that contains both Enemy and Reward. Player would still get caught by the Enemy and game over.
Instead of a button to pause, we use KeyListener that responds to the “ESC” key for pausing the game.	In order to avoid any focus on buttons and mainly focus on KeyListener.

Management Process

During phase 2, the management process involved short meetings during which we discussed what we accomplished and set goals/objectives for our next meeting. Furthermore, we had some longer work meetings for implementing classes that are more complicated like BoardState so all of us get a great foundation of how the objects will interact on it. Our division of roles and responsibilities was flexible. We worked together to implement classes that were commonly used, picked classes to work on individually based on each of our skills and interests. We also created branches so that we could work in parallel and check each other's work while improving the quality of code.

External Libraries

Library	**Reason**
java.util	- java.util.Random for generating random coordinates and probabilities for spawning rewards, barriers, and punishments - java.util.ArrayList for keeping track of rewards and punishments
java.io	- java.io.File for reading the images for player, reward, bonus reward, punishment, and enemies - java.io.IOException for catching file reading exceptions
java.awt	- java.awt.KeyListener for receiving player input (movement) - java.awt.KeyEvent for parsing player input (movement)

	- java.awt.Image for scaling images
javax.swing	- javax.swing.* for creating GUI components using JPanels, JLabels and JFrames and displaying the player, rewards, bonus reward, punishments and enemies - javax.imageio.ImageIO for reading the images for player, reward, bonus reward, punishment and enemies

Measures for Enhancing Code Quality

To enhance the quality of our code we used inheritance, singletons, encapsulation, and modular design. First, we had Player and Enemy inherit methods from MovingCharacter which reduced the amount of code we had to write. Second, we used singletons to ensure that Board, BoardState, ScoreBoard, TimeCounter, and Player only have one instance. This also lets us avoid having to pass in these objects as parameters which increases the readability of our code. Third, we encapsulated our code using access modifiers, getters, and setters which increases the maintainability of our code. Finally, we made our design modular by dividing the game into components. An example of this would be how we divided Board and BoardState such that Board focused on the GUI while BoardState focused towards the functionality or object interaction which is significant as it makes our code easier to understand and maintain.

Challenges

During phase 2, the main challenges that our team faced were mainly related to the lack of experience and familiarity. We started by implementing classes according to our UML class diagram in phase 1. Due to the lack of feedback and requirement deficiencies from phase 1, we discovered that many classes and methods had to be changed and reconsidered. When we tried to implement each class, such as creating the game board, the lack of familiarity with Java, Maven, and GUI libraries caused us to spend plenty of time learning from online tutorials and reading documentation. We have realized that the initial design requires further polish to create a working game that runs smoothly. After discovering the core issues with our initial design from phase 1, we have managed to communicate and redesign the game through narrowing down the specifications and detailed requirements.