Phase 2 Report

Implementation Approach

Our team met weekly to plan, discuss, and code together. We also coded separately in our own free time and consulted each other when necessary. At the start of Phase 2, we looked at the objects in our class diagram from Phase 1 and assigned those objects to team members according to interest and availability. We used IntelliJ as our IDE for consistency. To manage our code, we named Git branches after ourselves and regularly added code, committed with brief messages, and pushed to our branches before merging with master.

The game app itself uses the State design pattern with menu, instruction, game, and win/lose end states. The game loops through “ticks”. In each tick, a user input is accepted via keyboard to interact with the app, and the app checks if a state change is required. In the game state, each tick also manages Entity classes by appropriately moving Characters, collecting items, and making bonus rewards appear or disappear. For project structure, we also chose to have “Grim\src” as our sources root for the sake of naming consistency, and this still allows “Grim\src\main\java” to maintain source code as per Phase 2 requirements.

Adjustments & Modifications

Note: “Game” object refers to the overall game app (including all states). However, the word “game” in “GameState” refers to the actual part of the app where the user controls a character, collects rewards, and experiences interactive gameplay.

Phase 1	Phase 2	Reason
We had only 3 States: MenuState, GameState, and EndState.	We now have 5 states: MenuState, GameState, LoseEndState, WinEndState, and InstructionsState.	We split EndState into Win and Lose to reflect its more specific versions, and added an InstructionsState to teach the user how to play.
The Game object contained all existing States as fields.	Game now contains a reference to a single current State instance. Game changes States when appropriate.	This strategy aligns more with the State design pattern taught in class, and prevents us from having to store all our States simultaneously in Game.
Game contained most of the current in-game information (e.g. bool “isOver” and a Controller object to manage Entity’s).	We moved that in-game information to the GameState object.	Better separation of concerns: Game object focuses on managing ticks and State switches, while GameState manages current in-game information.
State tick() methods had void return values.	State tick() methods now return true if we want to end the current state, and false otherwise.	This allows the Game object to know exactly when to change states.

Position class which store the x, y coordinates of an entities	Use java.awt.Point to represent coordinates of entities	It has everything we intended to implement in Position class.
Items (regular rewards, bonus rewards, and punishments) were originally initialized in the Controller method initItem().	initItem() was split into initRegs and initPuns. Meanwhile, bonus rewards are now initialized using manageBonusRewards().	Regulars and punishments are all spawned at the beginning of the game, but bonus rewards are spawned randomly throughout the game and with a maximum of three per game. Bonus rewards therefore require extra checks and unique functionality.
Board and Cell originally were handling everything related to positioning	Cell helps with GUI and map generation. Board helps to maintain walls of the map.	The positioning is handled in the Controller as this decision decreased coupling. GUI external library required the classes to be present.

Management and Responsibilities:

We worked as a flexible team meeting every week to discuss designs, plan schedules, and code together. All team members supported each other with advice and practiced effective communication by sharing any thoughts, concerns, or ideas. The general division of roles and responsibilities is as follows:

	Code	Other
Adrienne	Game object (excl. run and render), States (excl. InstructionState), Controller object, Command enum	Report (excl. external libraries)
Sarah	Graphics, Display, start and stop thread methods in Game, Board and Cells, utilities for text file loading, miscellaneous tweaks	Textures, board layout
Wai Lee	Entities object, UserInput, run and render method in Game object, Collision between objects, InstructionState	Report (External libraries used)
Nikita	Review and minor edits	Review of deliverables

External Libraries Used:

- **java.awt.event.KeyEvent & java.awt.event.KeyListener:** For implementing UserInput: KeyAdapter might have been a better option as it is designed for javax.Swing object but keyListener is easier to implement and it should be enough considering we only have a single Jframe.

- **java.awt.Point:** To represent the position entities on the map.
- **java.awt.image.BufferedImage:** Represent image of any object that would be rendered.
- **java.awt.Rectangle:** To represent bound of on-screen objects. Rectangle.intersect() method is very handy for collision checking.
- **java.util.concurrent.ConcurrentHashMap:** To store the object's position on the board. Used concurrentHashMap instead of hashMap to avoid runtime error related to threads.
- **javax.swing.JFrame & java.awt.Canvas & java.awt.Dimension:** To get the display window and canvas
- **java.awt.Graphic:** Where the images are rendered.

Describe the measures you took to enhance the quality of your code:

We paid attention in class and used design principles and design patterns as suggested in lecture. For example, we focused on abstraction and separation of concerns by having multiple modules and classes (instead of all fields and methods in a single class or file), making our code easier to understand. We also used the State design pattern as mentioned above, allowing us to easily keep track of and utilize our current state. Each State implementation also naturally used the Singleton design pattern so that we didn't have to make multiple instances when one would suffice. We encapsulated our code by hiding data and behavior whenever possible, so that any personal characteristics and functionality of the Main Character would be privately owned by and solely accessible through the Main Character to prevent accidental manipulation of data. We took advantage of polymorphism through interface implementation; for example, all rewards and punishments share similar fields and methods such as positions and render methods. So we made an abstract Item class and implemented it with RegularReward, BonusReward, and Punishment classes. These measures prevent redundant code, let us change characteristics of several items in one place, and make debugging easier since we only have one class to check for related problems.

Outside of class, we did extensive research into existing libraries and methods, observing effective user experience design from other games and watching tutorials to discover any efficient logic or tools we could use. We also practiced active communication over Discord so that everyone was on the same wavelength.

Within our code, proper documentation and comments were utilized as time allowed, closely following the Javadoc guidelines from class. As for Git, we used branches during development to try out our code before merging. We also added descriptive commit comments to explain code updates. These measures allowed our team to understand each other's code to prevent and recover easily from merge conflicts.

Finally, our IDE of choice was IntelliJ, which allowed us to catch syntax errors quickly, build and run our code with Maven, understand project structure at a glance, and even use Git without having to use a command prompt. IntelliJ improved our efficiency so we could work with our code efficiently without breaking concentration flow, ultimately improving code quality.

Biggest Challenges:

- Researching and learning game design
    - We had never made a game like this in any language before let alone Java, so it was difficult figuring out what libraries, tools, and overall game logic to use. We

had to research and make so many decisions on everything from small details like user input style (keyboard versus mouse) to large components of the game like how to keep track of entity positions, switch between states, render all images, and search for specific entities efficiently. These details were not all taught in class and required much time and effort to figure out. We did look towards lecture slides for inspiration, but there was an overwhelming amount of information (design principles, patterns, styles, and extraneous information) to sift through, analyze, and determine relevance. For example, we were introduced to various creational, behavioural, and structural design patterns and had to analyze the diagrams and code of each pattern to see if they were applicable to our game. Online, there were also many tutorials for slightly similar games, which helped with the basic set up but also added extra problems by being out-of-date, or inapplicable.

- Staying up-to-date with Git and preventing merge conflicts
    - Working in a team requires us to frequently update our code so that we are all on the same page. However, it took some time for us to become familiar with creating Git branches, pulling code from other branches, and merging safely. Sometimes, we would be working on the same file simultaneously which could cause complications if our edited versions don't mesh well. So we had to practice checking out development branches, pulling frequently to update our code before pushing to prevent merge conflicts and help everyone stay up-to-date. We also used regular communication in Discord, pinging team members to ask questions and keep track of progress. We also quickly divided roles and responsibilities between team members to minimize situations where multiple members were working on the same file.
- Overall time crunch
    - The time required to do the following is not short: figure out game logic, reconfigure design, set up and attend meetings, implement code, test new functions, update branches, resolve merge conflicts, attend lectures, search for and watch tutorials, fix OS-related problems, make assets, ask questions, wait for answers, and all details in between. Add in non-276 obligations (other classes, volunteering, work, moving, and personal mental and physical health time), and our available time quickly decreases. We worked very hard to commit to regular meetings and coding so that we could meet project requirements.
- Gameplay
    - User experience is very important! We wanted to make winning a bit of a challenge so the user has fun, but also easy enough that the user doesn't get frustrated. This required us to continuously tweak character speeds, wall widths, doorway sizes, and reward/punishment spawn locations to strike a comfortable balance in difficulty. We are excited to continue experimenting with balance in our testing phase. We also want the game to be pleasing aesthetically, so our artist personally drew and designed our textures, including animations to match on-screen movements. These efforts took some extra time but were worth it. We were also faced with the challenge of accessibility in gameplay, and ended up using different colours and shapes to represent different items.