Stephen Dao, Danyaal Patel, Peter Salnikov, Brendan Saw

Project Phase 2 Implementation Group 20

Overall approach for implementation

At the beginning, we relied heavily on the UML diagram that we had created to form some sort of basis of how and what to implement. We got skeleton structures down, and decided to implement features later. After we initially implemented those, we started to go into the specifics. We would section off certain parts of the project to various team members, and have meetings every couple of days, to merge code together and resolve any problems with the implementation. Everyone used their own branches to make sure there would be no problems with the code, and generally different classes. When problems arised, we would sometimes have group coding sessions where one member shared their screen and the rest helped and discussed what to change. We used Jira as a "Team Tasks" platform to assign tasks to members and make deadlines. By having a structured schedule and clear assigned tasks, we had excellent communication and everyone knew what they were responsible for. Hence, every time we had a meeting, we found that team members had finished their tasks, and we were ready to merge and move on to the next steps.

Revised design choices

We ended up changing a lot of our initial design choices, as there were many features and little things that weren't initially considered in the first phase.

Characters

- NonStationaryCharacter:
    - Position coordinates split up into an x and y position rather than set to a Tile for easier updating
    - Getters for the x and y coordinates added for easier access in other classes
- MainCharacter:
    - Made into a singleton for easier access across classes
    - Handles movement related input for easier modification of position
    - Collision handling method checks for collision with a Reward object by checking if the Tile at the player coordinates contain a Reward
    - move() method calls Exit method to update whether the exit is closed or open
- Enemy:
    - Collision handling method checks for collision with MainCharacter
    - onPlayerEntered() method used to also call the endGame() method inside the Game class

BoardDesign

- Exit:

    - Added checkCheckpoints function that checks how many checkpoints are left; if it equals 0, then the exit sets its isOpen to true.
        - Additionally, it checked whether the mainCharacter was on the exit position tile and would trigger Game's endGame() if done so.
- Wall:
    - Added isOpen just to allow other classes to check whether this tile could be moved onto
    - isOpen set to false by default to disallow movement onto the tile

Grouped Board, Game and Tile into a single package called Core.

Core

- Board:
    - entryPos, exitPos ordered pairs to store coordinates instead of integer indices
    - added variables for dimensions
    - added 2D array board; store IDs of each tile which allows easy access to each tile through indexing using coordinates
    - Removed method createEntryAndExit(); Board class has access to those fields directly
    - Associated different types of Tile objects with an ID, which allows for easy creation of a board from an array of ints
    - Added method to generate Bonus rewards since the board stores the array of Tiles which makes it easier to edit each Tile
- Game:
    - Extends JavaFX Application to allow for usage of JavaFX
    - Added gameTicks
    - Added ticksElapsed counter, where a tick is 2 seconds
    - Allowed the user to pause by adding a pause boolean
    - Added an enemyArrayList that would enable us to iterate over every enemy that is inside the array list
    - Added public static void main inside game as it would need to call start()
    - Start function (called by launch in psvm)
        - Creates the window that we can visualize the game inside
        - Implemented gets input handling and passes it to characters to move accordingly
        - Implemented multiple groups to allow for different scenes (screens) and strict layout centering
        - Creates and displays the stats at the top of the board such as winStatus, Time, and Score.
        - Implemented dynamic sizing, where the board should adjust to the window size, and would try and center itself within the window

        - Created pausing, where pressing the ESC key on the keyboard will allow the user to stop the timer
    - Created a function to generateEnemies
    - Created a function to updateGame, when called it would call the enemies to move
    - Created a function to draw the main character on the board
    - Created a function to draw the enemies on the board since enemies are drawn separately from the player
    - Created a function to draw the board
    - Created a startTimer function that starts the game ticks
    -
- Tile:
    - Added isOpen variable that is true on default to allow the characters to walk on it
    - Did not implement the directional tiles/references to other tiles inside of the tile object since it would be easier to just access those using the x, y coordinates from another class, which tiles inside of a 2d array
    - Added a reward object inside of tile so that when the tile's hasReward is true, we could access the reward inside the tile
    - Inside the Tile constructor we added a parameter to construct it as either a checkpoint, punishment, bonus, or no input in parameters for it to just be a mundane tile with no special characteristics
    - Implemented getters for variables such as hasReward and the reward object
    - Implemented a removeReward for cases where the reward is collected or expires due to time constraints (bonus rewards)

TileAction

- Reward interface:
    - Removed used and scoreChange
- Punishment:
    - Added a negative scoreIncreaseValue
- Checkpoint:
    - Added a scoreIncreaseValue
    - Added a static checkpointsLeft variable since each Checkpoint object shares the same total number of Checkpoints and can easily decrement this value themselves
    - Whenever constructor is called, checkpointsLeft is increased by 1

Management process

We used Atlassian's Jira to manage tasks and understand what everyone was doing in the week. Using this software was effective since we learned to divide large tasks into smaller subtasks for each person to handle. While Jira is typically used with an Agile SDLC, we chose not to opt by this methodology, since our schedules did not permit such actions, though the board

was used in a Kanban-style manner, with users continually moving tasks to completion and new tasks being added.

In order to assign tasks to specific people, we coordinated weekly or biweekly meetings to talk about what has been implemented. Additionally, in these meetings we would merge our branches and resolve conflicts. It was important that we were all together when merging so we could give input as to how to merge our branches - notably the ideas to keep between the two branches. Additionally, we would spend more time on the weekend implementing code when we were in our calls to bounce ideas back one another, since we wouldn't have time to meet on the weekdays.

External libraries used

The external library that we used for this project was JavaFX, which was used for drawing up the board. JavaFX was a big part of our project, with many import statements linked to that. JavaFX was implemented in the Game file of the Board package, and we used a Stage from JavaFX to have our application window, using the start() method and launch from main. Using a combination of Group nodes, AnchorPane, and BorderPane, our window and board came to life. AnchorPane allowed us to make an offset grid with tiles and Rectangles. BorderPane was used for the overall structure of our window, and The actual Board was implemented in the center of BorderPane. Group was used to make sure the Board was centered, and adjusted to any resizing of windows. SetAlignment and setCenter were also used to reinforce the strict layout. We implemented separate scenes for the pause menu and the game, by using Scene. Timeline, Duration, and ActionEvent were used for the Timer that we implemented at the top. KeyFrame was also used to update this timer. EventHandler was used to listen for key inputs and releases while calling the appropriate methods. JavaFX Scene was also used to paint tiles different colours, and have shapes such as rectangles, along with text displayed for the various stats at the top of the screen. BorderPane also allowed us to use the top of the screen to display the stats.

Measures took to enhance quality of code

Added JavaDocs and JUnit testing early. Comments to allow other people in the group to understand what we implemented and why. Following our modularized UI diagram already gave us a higher quality of code since we need to ensure low coupling, and we made sure to break down larger functions into multiple small functions to enable better legibility and clarity.

Biggest challenges

A major challenge that we had at the beginning was getting started. We had broken down subtasks and knew what we should be implemented, though without some sort of initial skeleton for our code and classes, we didn't know what other people in the group were implementing, and how to get classes to interact with one another. This was resolved by eventually creating the

skeleton, and after that we were finally able to more easily assume the implementation features that others had done.

Integrating our individual work was a major challenge at the beginning. Merging everyone's branches into master would cause conflicts, crashes, and bugs. To take care of this, instead of having all the individuals work on properly fixing the errors, we acted efficiently and had one person share their screen, so that everyone could discuss and work together to solve the problem at hand. Other times, there were problems that were not related to merging, but rather bugs or errors in our application. This was dealt with in a similar manner to ensure that everyone is on the same page. We remedied this by merging branches during our group meetings; the amount of work done each week allowed for such breaks, and this way everybody kno

Sometimes, it was a matter of trusting that everyone completed their respective tasks for the next meeting, since there was often no way to tell what everybody was working on without daily communication. This issue was mitigated by mandating at least one weekly meeting, every weekend, with more if needed. This provided a soft deadline for work to be completed by, and ensured a good communication flow between our group. We also realized that many methods, fields, and classes had to be modified upon implementation time. In hindsight, this should be expected, as porting an abstract concept into the concrete immediately revealed what would and would not work. This was simply solved by more quick sketch-up design work, and keeping a log on everything that we changed to make writing up the report easier.

Another big challenge was when the merging of different branches would create errors and crashes. To take care of this, instead of having all the individuals work on properly fixing the errors, we acted efficiently and had one person share their screen, so that everyone could discuss and work together to solve the problem at hand. Other times, there were problems that were not related to merging, but rather bugs or errors in our application. This was dealt with in a similar manner to ensure that everyone is on the same page.