Phase 2 Report

Approach

Our initial approach to implementing the game was similar to our initial design in terms of the various components of the game. However, there were important areas of the game that were overlooked, for example the way we would draw our game onto the screen. As none of our group members had much experience with java libraries, we did some research to find Swing, a popular library to make desktop applications.

From here, we were able to start implementing the components of our game and further develop our design. To start, we used a Board class (consisting of a 2d array of Cells and more) to keep track of where the different components would appear on the screen. We used the factory method design pattern to create our game filled with Cell objects and Entity objects within Cells. Most of the components were extended from the Entity class or the Cell class, with the remaining classes providing support and functionality to these components.

Modifications

We found that our initial design was a great place to start, however we quickly realized that there were so many components that we had not thought of.

a) Class Diagram

Our first step in building the application was to create a game window, however we had not thought about visuals when we were designing the class diagram. As such, we created the GameHandler class to create the game loop with a Thread class. We found that having a class for each state of our game wasn't the best approach, as many of the states were overlapping. For example, playing/pausing/resetting the game requires an instance of the game to be running, while menu/lose/win states were more comparable to a "terminal" state. The change in between the states was handled by the GameHandler class, which was then visually updated by the UI class.

The UI class itself was also a class that we needed to add to our previous design. Even though some game states had overlapping functionalities, there still needed to be a unique visual output for each state. In addition, the UI class also displayed various HUD information such as the current player score and number of keys collected.

The Board class went through minor changes in the functions. Since the Board is in charge of updating the cells, we also added functions to give information on whether a Cell can be moved to or if it contains an Entity. We also added functions to insert a new type of Cell or insert Entities into the Cells. These new functions would return a value by calling a method from the Cell class, and so they required respective methods in the Cell class.

The Cell class went through minor changes as well, though it mainly had to do with the visual aspect. Our previous design had not taken into account the background of the game, and so we needed to add Cell tiles. Since the Cell class is abstract, we just needed to add individual draw methods to the corresponding Cell type. For example, the DoorCell had a unique drawing method because it could only be opened if the player had collected enough keys. To implement this, we added a Score class that could keep track of how many keys the player has, but also the current score of the player (determined by Coins and Diamonds). We also added a new type of Cell, GoalCell, that could be a clear landmark to the player and which triggers the win state once the player reaches the Cell.

The Entity class remained nearly the same, with additions to its subclasses. One key difference is that we needed a way to erase Entities both in our game and on the screen. Our solution was to create a new subclass, the NullEntity class, that would replace the old Entity's place in the Cell, and to draw the background over the previous image to make the object "disappear". We also added different reward classes, including the Key Class (used to unlock the Door), the Coin Class and the Diamond Class (points for player score). The Trap class remained unchanged, and the GroundItem Class remains consistent with our original design

The Player and Enemy classes needed an additional method in order to draw the character differently depending on the movement direction. Additionally, we hadn't added a way for the player to be controlled in our initial design, and so we needed to add the KeyControl class. This class was not only used for player movement, but also for pausing the game and navigating through the menus. To increase the difficulty of the game, we also wanted to create multiple enemies, and we added the EnemyCollection class to help update and keep track of multiple enemies.

Finally, we also added the Factory Method design pattern in order to set up the board. We wanted to have all of the positions randomized, however this introduced potential issues such as having keys/rewards spawn in unreachable locations or terrain that is severely unfavorable to the player. Our solution was to set up static locations of the terrain, and have the position of keys, coins, traps and diamonds randomized. We used the MapFactory to create different Cells, and used the ItemFactory class to create different Entities. The two factories are used within the MapBuilder class to create the board.

b) Use Cases

In general, the use cases didn't change significantly, and the general idea behind each use case is still apparent. However, the implementations introduced various details that are more specific than the design had originally described. For example, in our "tick forward" use case, we've now specified our FPS (frames per second) to 60. In our "Controls" use case, we changed the player movement speed from 1 cell per tick to 0.12 cells per tick. The use case for the Scoreboard sees minor adjustments as well, as we've introduced the concept of keys vs other rewards that grant points, and we've implemented our game in a way that shows both the scores and the keys that the player has collected. The pause use case has also been changed in that the user does not click on a pause menu, but rather uses specific keys.

We've also added more to the menu, including Credits, Exit Game, and a character selection option. Assuming that the program has successfully run, the use case for each of these options is to switch into the respective state if the option is selected by the player. The options can be selected by navigating the menu through the WASD keys.

Challenges

One of the biggest challenges that we faced was getting the game to reset in all scenarios. There were various components that we had to change that would allow us to get the game to reset because of how they were implemented in the first place. After figuring out and resolving the issues, we learned to not make the same mistakes again. When we first implemented the game, we didn't have a title screen. Then when we made the title screen, the map was being built when we were in the title screen, but this caused the title screen to have lots of lag, so we only let the game be built in the play state. This fixed the lag issue. Then when we wanted to reset the game, the map was not being built properly as it was missing keys, and the player could walk through the walls, and it couldn't pick up anything. There were many issues to solve. This is one of those things that is learned with experience, as when we were originally building certain assets, we were not concerned with the fact that we had to reset them at some point. Our first thought was just to start a new game loop, we soon found out this was not a good solution so we decided to create a reset game function that would handle the objects that need to be recreated. The first issue we solved was that we were using the same player object on the map before the rest, so the new map wasn't fully connected to the player, it was only resetting its position. We solved this by creating a new player each time we reset the game. We also had some issues with the door and goal cells that were solved the same way.

So now that the player wasn't walking through the walls anymore, we still needed to find a way to reset all the objects, this consisted of a complete rework of when we built the game. Since earlier we only built the game in the play state, we needed to undo this change so we could reset the game from the lose state and the play state. To fix the title screen lag we only built the game when we entered the game. The reason this took so long to figure out was for two reasons. Firstly, we thought that we could solve the issue by just creating new objects because somewhere we were reusing an existing object without resetting it.. The other reason was that the person resetting the map was not the person who made the map in the first place and was not fully aware of the code base. Eventually the issue was found but it could have gone a lot faster if there was more communication about the issue. We realized our communication was an issue, which is why we started implementing pair programming. However, if we did this for the entire project, it would have slowed down the overall efficiency because we would need to constantly look at each other's code.

Quality of Code

To maintain good quality of code, our group first started designing the general structure in phase 1 (class diagrams). Our goal was to separate components by their function and create relationships between these components in order to create a clear and consistent code. This included using abstract classes to generalize objects that had shared attributes and functions, enumeration "classes" to make directions clear, and creating relationships between classes that would prevent violations of Demeter's law (one dot rule). Using our design, we implemented our code using the previous strategies that we came up with together. We also implemented additional strategies, such as the factory method design pattern to insert the Entities into the game.

Another great way we used to improve the quality of our code was to use pair programming. This allowed us to make less mistakes when first writing the code and made it easier to bounce ideas off of each other to allow for correct implementation the first time. A couple of times there were pairs of us who programmed together a couple times to implement some key changes when implementation was critical as well as during testing. They also pair programmed when they wanted to go and redo a few things, this was to ensure that it was implemented properly the second time so that we didn't have to go back and change it again. This was especially useful when coding different components that worked together closely, such as the GoalCell and the winstate.

The main portion of time that we spent improving our code had nothing to do with the functionality of the code at all. Documentation is an extremely important part of the quality of code, code without documentation even if it is perfect is often still extremely hard to read and extremely time consuming. The communication issues we mentioned earlier likely could have been solved faster with better documentation. For example, if there was a note in the documentation about how the map is only built when in the play state, it may have been picked up sooner. So, after this we really wanted to focus on the documentation and provide clear communication for our work. This prompted us to use a Discord channel to update each other on various changes and what we were currently working on. Finally, we made sure that we had Javadoc documentation at the top of every function and class, which allowed our code to be much more readable and usable, especially for those who didn't write the code themselves.

Division of Roles

Our plan at the start was to make a list of classes that we wanted to create in the order that we wanted to create them in. This was important as some components needed to be implemented before the others. Then, when we had time to work on them we would text the group chat to let each other know, so that we weren't working on the same things at the same time and doing the same work twice. This worked out very well. There was some disparity in how much work was done at the start due to this method, but as we became more accustomed to the strategy, we were able to communicate better by the end and produced more even workloads. This also allowed all members to have flexibility of their time to get things done when they can and not hold anyone back from doing their part. Near the end when we had just a few things left to complete, we divided the remaining tasks amongst the group so that we knew what we needed to get done. This helped us finish the code and refine the game before our deadline.

Libraries

- Swing
  - JPanel
  - JFrame
- AWT
  - BufferedImage
  - event.KeyEvent/KeyListener
  - Colour
  - Font
  - Graphics2D

The two main libraries we used were the Swing library and the AWT library. We chose these libraries due to their popularity and compatibility with each other. Both libraries were essential to reading, updating and outputting various visuals and images.