Simon Fraser University
CMPT 276 : Introduction to Software Engineering

Phase 2 Project Report

Implementation Phase

Group 19

Arda Cifci, Nayeong Lee, Ellie Neufeld, Cole Thacker

Overall Approach for Implementation

Our group's overall approach for the build phase of the project was to separate the implementation into three major stages. At each major stage we would then further break down what we would need to accomplish by reviewing our four phase one design documents. After discussing which tasks should be completed in the current stage we would then add them into a comprehensive to-do list. Each member would then pick tasks to work on individually, or if the task was complicated, work on it together. By utilizing the to-do list, we were able to keep track of the progress we were making and made delegating tasks to the group members straightforward.

The first stage of our implementation phase was to 'set the scene' of our project by adding all the building blocks our game would need before we start implementing the core aspects of our game. These building blocks included obtaining or creating all the required resources (sprites, music, etc), creating the maven project folder with the pom.xml file, creating skeletal class files from our UML diagram, and implementing some of the simple classes such as our Music class. Our second stage was implementing the core aspects of the game. This would include the game's overall logic, such as updating the game's state or moving the player, and the user interface with all of its features. Our group knew that this stage would take the majority of our time thus we made sure to allocate enough time to finish this stage properly.

The third and final stage of our implementation was primarily bug fixing, code fine tuning, and commenting. After all the program's logic and UI have been implemented, we took time in this third stage to fine tune the numbers in our program such as entity position or score as after seeing the completed game we can better place things in correct positions with less guesswork. Although we knew we were going to do unit testing in the next phase of the project, we decided that fixing the noticeable bugs in this phase of the project would help reduce the problems we would face in the next phase. Overall, building this phase in a three stage process guided us into implementing the game in a more structured, goal-oriented manner.

Adjustments and Modifications to the Initial Design

While constructing the game, we realized we needed some more classes that we hadn't originally anticipated. We added a class for each screen the user sees, as well as the class BoardSpriteManager to manage the sprite images. These classes were added to make it easier for us to paint images onto the user's screen. We added a Music class to play music while the game runs to make the game more enjoyable for the user. We added a Direction enum to be used by our game loop, and for our Input Handler class, which acted as more of a utility to classes.

MapReader allows us to flexibly design maps, instead of having them be in main memory, we can store the layout of the map and read from disk, populating the screen as needed, which greatly reduces overhead in terms of lines of code. It also allows more variety for the user/player to enjoy, so instead of working on a singular map, they can experience many. MapReader also vastly improves the extensibility of our system, as there are no hard coded blockers or doors, everything is dynamically created as the map is read in.

The majority of the rest of the new classes came with implementing the user interface via Java Swing, which was expected as all project members were new to the Java Swing framework, which was expected, and did not cause us to radially diverge from our original design plan These additions were more of an extension of the original design as opposed to a redesign. This constituted more of ‘at the keyboard’ level of design as the two developers in charge of the UI became more familiar with Java Swing and how it worked.

We removed MenuCommand class, as the original use case was for using keys such as escape to implement pausing, however this was handled by adding buttons to the screen, making it much more apparent to the user how to pause the game. Timer observer was removed as we only had a singular observer that the timer would have updated(GameLoop) therefore having a game timer be a part of the GameState class was more efficient as it allowed us to integrate the two more easily. The class Door was removed when we realized that we could implement doors more easily without having its own class, reducing the number of entity classes to integrate.

We also had to extend from abstract classes in Java Swing, and implement interfaces, such as GameLoop which extended JPanel for the UI and also implemented Runnable to be able to be threaded. Input Handler also implemented KeyListener to be able to consume the signals emitted by the ‘WASD’ keys. The implemented and extended classes did add some overhead, however as development progressed we were able to hone in on some essential required functionality.

Management Process, Roles, Responsibilities

To ensure everyone is on the same page, we primarily followed the management process that we established in phase one. We had regular meetings on Wednesdays and Fridays on a weekly basis through Zoom. There, we checked in on our progress, raised and solved any technical issues, and shared new ideas. Moreover, we adopted scrum-like daily meetings on Discord, where we updated each other on our work and informed any changes and issues that need to be addressed immediately. Members also communicated on Discord when needed in an ad-hoc manner to address issues that arose on the fly and render help to each other when needed. This allowed other members to be apprised of issues and have insight into issues and solutions to those issues.

At the beginning of each phase, we held additional in-person meetings at the school library to solidify our direction in the game and work out some of the logic. Such in-person meetings helped develop a collaborative and supportive environment among team members. We separated the build implementation into three major stages, each in which we further broke down. In addition to weekly meetings, our team members utilized a comprehensive to-do list where we put tasks to be completed. Each member picked tasks to work on individually, and by crossing it off the list, everyone was able to keep track of the progress and be aware of remaining tasks to be completed.

During the implementation of the game, all team members contributed to each part of the implementation (UI, gamelogic, etc), but we each focused in different areas. Ellie and Cole worked predominantly on game logic, developing sprite moving algorithms and overall mechanics, while Arda and Nayeong concentrated on UI and asset creation, developing an interface that is single themed, useful, and reliable to all users.

Ultimately, our means of communication and assigned roles increased productivity and efficiency of phase 2.

External Libraries Used

Our group did not use any external libraries in the creation of our game. We solely used internal libraries that were included in the Java Development Kit (JDK) and our own code. For example we used javax.swing and java.awt to create our user interface and its features, and java.util.Arraylist to keep track and store our entities. We also took advantage of internal libraries such as stream() to abstract away manipulation of collections regarding filtering and sorting.

Use of no outside libraries in our game logic/rendering was a challenge, however it allowed us to understand how our system worked and allowed us to build in a manner that we could extend easily, and reduce confusion. We used build automation utilities to help abstract away some of the more difficult details of building our project.

Measures to Enhance Code Quality

In consideration of code quality, our group adopted a coding style guide to enhance readability of our code. Where we could, we attempted to reduce the amount of repeated code, following the Don't Repeat Yourself principle, and also the Single Responsibility Principle. At times there is repeated code, however extracting that functionality into a separate function reduced readability and it was useful to include those sections, i.e. calculating what tiles entities are located at. Using the Java native fluent API Stream() greatly reduced the amount of code for filtering and sorting.

Following on from the Single Responsibility Principle we tried to decompose functionality into different classes, attempting to decouple functionality between the data models of the entities and associated entity classes, and board and associated board classes. Any reading of image files regarding entities that move were held in those classes, since those classes know about their own movement and what should be displayed upon render, and all other static non-moving entities or BoardTiles were encapsulated in another class to handle image reading since those displayed images do not change.

Our convention was captured by a reformatting tool that would be run after large functionalities were implemented to help with readability to automate refactoring that did not impact functionality, but instead the form of the code. If other group members needed to understand the code, then the code was forced to appear in an expected and easily digestible manner.

We respected encapsulation, where the board stored the width and height in columns and rows of a matrix, whereas the entities had X/Y pixel coordinates, requiring translation between the two, but hid the underlying implementations from each of the two classes. Data fields of classes were accessed through getters or setters, preventing accidental state

changes or references. We also used inheritance to group together the functionality of an abstract entity class to allow extension to concrete classes, and a Movable interface to allow for interface segregation when entities had to be able to be moved.

Biggest Challenges Faced

During the implementation phase of the project, we faced several small challenges. However there were two bigger challenges we faced that stood out from the rest. The first big challenge we faced was maintaining our daily scrum meetings on discord. Because everyone has their own schedule, which often changes, it was rather difficult to bring everyone together daily to update each other on what they have completed and what else needed to be done. At one point in our implementation we had two group members double book a task because of a missed scrum meeting. This ended up costing us some time and solidified our desire to find a simple solution. Our solution to this problem was letting the group members who knew they were going to miss the meeting be able to write what they intended to say at the meeting and post it in our discord meeting chat. This ensured that everyone got to tell what they needed to say and stopped our double booking problem from recurring.

The second big challenge we faced in the implementation phase was creating the code for the enemy player tracking logic. Our group knew that implementing the enemy player tracking was going to be the hardest part of the game's implementation in terms of logic difficulty. We initially wanted to use a proper path search algorithm like A* but we could not get it to work properly for our implementation. Thus instead we settled on creating our own simple yet effective pathing algorithm. This came with its own set of challenges and required many hours to figure out properly.