Overall Approach to Implementing Game

Our game is a 2D top-down RPG where the player has the objective of reaching the endpoint having collected all the keys. The player must make his way to the endpoint while avoiding moving enemies and hazards. To increase the player's chances of winning, he can collect rewards that increase his health, score or freeze the enemies. The game was implemented using java and maven for package management and version control.

Our initial approach was to divide the work based on our UML class diagram from phase 1, into even related sections. For example, we planned to have one member tackle the implementation of the hazards and another the rewards. However, once we decided on our library for the GUI (Swing), it became clear that there were quite a few revisions we had to make (listed below). To account for the new dependencies and added complexity we shifted our delegation to have members who would implement the necessary GUI components and game logic while the rest of the group worked on our original classes.

Adjustments & Modifications

We had to make certain revisions to our classes and their elements because of many unforeseen dependencies. Below is a compiled list (in no particular order) of our revisions and why they were necessary:

Main

This class was added as our driver class, and serves to populate our game and run it (along with our TestLevel class, these two essentially replace our original board class), as well as provide utility to some of our other classes:

- Defines dimensions for the board and its tile grid.
- Creates objects and instantiates them for all the elements in the board.
- Defines starting positions for dynamic elements (Player and enemies) of the board.
- Thread management.
- Updates the GUI.

TestLevel

TestLevel is our factory which populates the board with all of the static elements i.e. barriers, hazards and rewards. We added several utility functions to help with populating the elements.

- Contains methods for populating the board with the various static elements.

Window

This class was added to make and configure the JFrame for our window class

- Sets dimensions for the game board window.
- Set default operations for the window.

Bomb

This is an abstract class that is extended by other specific types of bombs to give different details. In our original plan we had a Hazards class but we decided to split it up into our static hazards (Bombs), and our dynamic hazards (Enemies).

Enemy

Our enemy class no longer implements a hazard interface like we originally planned, since we decided that our dynamic enemies function fundamentally differently from our static hazards, which implement a Bomb interface.

- Contains logic for enemy movement.
- Contains logic for enemy collision detection with barriers .

Health & HealthBar

These two classes were added to take care of the logic surrounding the Player's life count and to update the graphics in the GUI.

- Health contains static values for Player life count.
- HealthBar contains update methods to update the Players health in the GUI.

Player

Our player class was updated so that it could handle all it's required collision detection (e.g. with barriers, with enemies, with hazards, etc…).

Key

We added a class to implement Java's KeyListener library to read keyboard events.

GameOver/Pause/MainMenu

We decided to divide our menu class into three separate classes for each individual menu. Each menu takes care of drawing itself and pausing or running the main thread at the appropriate times.

Score

Our score class displays and updates the score tracker at the top of the window as the player progresses and the player gains/losses points.

Images

This class was added to handle all of our texture resources and to load them into our game.

Project Management & Division of Roles

Initially all team members would push their modules to personal branches on the GitLab project, after which the modules would be combined on a single branch. After this point all team members would work off the same branch. Division of roles follow below:

- Ritvik
    - Implementing the timer and thread functionality in the Main class.
    - Implementing Key class.
    - Implementing Enemy movement logic.
    - Implementing the Swing framework for the main board.
    - Implementing a framework for loading in textures to our game.
- Yifan
    - Organizing file/directory structure.
    - Implementing the Freeze reward class.
    - Implementing GameOver/Pause/MainMenu/MapChoice classes.
    - Implementing BGM/Boom/FreezeSound/KeySound/Losing/Winning classes.
    - Implementing Map class.
- Karan
    - Gathering resource files i.e. textures for our game
    - Implementing HealthBomb and ScoreBomb classes.
    - Implementing KeyReward and HealthReward classes.
- Milan
    - Implementing collision detection and onHit methods.
    - Implementing TestLevel class.
    - Implementing Tile generic class.
    - Implementing Health and HealthBar classes.
    - Implementing KeyBar class.
    - Implementing EndPoint class.
    - Implementing variable/list population in Main class.

External Libraries

For our GUI we decided to use Java's Swing libraries. We settled on using Swing as it is very well documented, albeit not quite as modern as some other GUI toolkits, but seemed to fit well as not all of our group members were familiar with Java so we opted for an easier to learn and well documented toolkit over a more modern and perhaps robust toolkit.

Code Enhancement

To ensure the quality of our code we added well placed comments, used threads where possible to boost efficiency and used Maven to manage our packages.

Challenges
The biggest challenges we faced were getting familiar with git and version control, learning some of Swing's functionality and concurrency when running threads. Particularly with our pause menu which needed to stop the running game thread and resume it when the player decided to continue. Fortunately, most of our issues were possible to overcome with external resources i.e. StackOverflow or documentation, although we did hit a few blocks where we decided it was best to change our method of implementation either due to time constraints or because there was a simpler way to achieve functionally the same results.