BILKENT UNIVERSITY

CS 319

Object Oriented Software Engineering

FINAL REPORT

31 March 2019

QUADRILLION

by The Group

Ana Peçini

Krisela Skënderi

Muhammad Hammad Malik

Ünsal Öztürk

Table of Contents

1. Introduction

We chose to implement the GUI and Controller classes using Eclipse IDE. Following we will provide details of the implementation progress divided into three parallel "implementation units" that we formed to assign by considering our design report and subsystem divisions , namely the main quadrillion functionality, the mode logic and the GUI classes. We used this method to separate the workload in such a way that every member could work on his task independently by following the naming conventions in the design report, and at the end it would be easy to assemble the individual parts into a single deliverable. Throughout the implementation stage so far we conducted 3 meetings to keep track of our individual progresses and inner deadlines, and discuss any issues we faced. Moreover, as for the previous stages and previous reports, we used Google Docs to keep track of our tasks and deadlines as well as share our work for the report.

1.1 Core Functionality

The core functionality of the game is fully implemented. This unit includes the QCoordinate, QPiece, QBoard, QGrid, QPieceFactory, QGridFactory, QBoardFactory, QGame, QGameFactory , QGameBuilder and QTimer classes. The code for all of these classes is already pushed to Github. The functionalities provided are:

- All the puzzle pieces and the 4-grid board configurations are available. A random quadrillion puzzle is generated every time the game is played .
- The pieces can be rotated, flipped, placed on the board and removed from the board (all the functionalities of QPiece) . There is no logical error in the placement of the pieces on the board: pieces cannot overlap, cannot be placed outside the boundary of the board and cannot be placed on top of blockers.
- The timer works correctly and the game is over when time runs out (indicated by a popup). The corresponding class is the QTimer class. The time PowerUp also works to increment the remaining time by a given number of minutes (amount likely to change after the next iteration).
- There are multiple themes available in the form of different images and colors for the puzzle pieces that can be incorporated into the game. The corresponding class is QThemeManager.

- Information about Player health is also updated when the game is lost. However, the main quadrillion game is still separate from the implementation of the modes, which means that the controllers that update the mode states according to a single game's outcome are not implemented yet.

1.2 Mode Logic

There are three modes in our Gazillion game: Treasure Mode, Level Mode and Ladder Mode. The state of the implementation for the classes included in this division is as follows:

- The implementation for the abstract class QMode is complete, as well as for the Level Mode and Ladder Mode classes that extend it.
- The implementation of the part of the Treasure Mode class that deals with the Final Game feature (treasure piece collection) is not complete yet.
- Player and Award classes are complete, but their functionalities still need to be connected to the individual mode classes so that in different modes the awards are evaluated differently. This part was intentionally left towards the end of the implementation as we still have to decide about the meaningful amount of awards based on how difficult the game will turn out to be. For now, the evaluateAwardForCurrentGame(), punishPlayer() and awardPlayer() methods in the abstract QMode class are the identical for every mode.

1.3 GUI classes

So far we focused more on the logic of the game and the modes, which is the most difficult and time consuming part of our project. Apart from the Main Menu panel itself, the other panels that should be accessed from the main menu are not implemented yet (Settings, Shop, Instructions and Credits). EventListeners and ActionListeners to connect the panels are not implemented yet as well.

However, these GUI components will be fairly easy to implement at the end because they are simply combinations of some other GUI classes which we did implement in order to be able to test the functionality of the main game. These implemented classes are: QFrame, QPanel, QImagePanel, QGazillionPanel (main panel for the core game that includes all other information panels), QPlayerInfoPanel, QUtilityPanel (for powerup information) and QPieceCollectionPanel (that holds all the unplaced pieces).

2. Design changes

There were not many major design changes since the design report. One of the important changes is the introduction of a simple message passing interface between the many threads of the program and the Observer-Observable interface. The reason behind this change is that when we came up with the design, we did not take into consideration the requirement of a message when the time in QTimer runs out. This event has to be communicated to the QGame instance which the QTimer lives in. This was achieved through having two interfaces, Observable and Observer, which follows the Observer pattern, but with a twist: The observers following an observable object have to be updated using a message wrapper, called a Message object, which encodes a message bit by bit, defined by constants. This makes sure that when the QTimer finishes in its own thread, a message is communicated to the QGame instance.

The same pattern was also used in the UI when it comes to updating the label which displays the remaining time for the game. In particular, the QUtilityPanel was reworked such that it now implements the new Observer interface. To provide higher level functionality and to ensure proper encapsulation and abstraction, the QGame class now also implements Observable (it is both Observable and an Observer), and redirects the observation requests to the QTimer instance. The QUtilityPanel class now observes the QGame instance for the time remaining.

Another slight change in the design is the overriding of the hashCode() method of the QCoordinate object. This was deemed to be necessary because we used HashMap instances in the implementation, and to run equality checks on the set of keys, which happen to be QCoordinate instances, and in order to do that, one has to provide the hashing scheme for the key of the entry. The implementation is a simple prime-number based hashing.

Throughout the implementation process, there also were several changes in class attributes as well. QFrame no longer stores the panel hierarchy as a list, and it does not do explicit indexing for keeping a collection of panels, but rather, all panels are now kept as separate fields in the QFrame. Several fields regarding interactions with the Controller components were added to the QGazillionPanel, such as a “lock” variable, which is modified upon receiving a signal from the QTimer thread of the QGame instance: if the player runs out of time, the game “locks” the UI and lets the player know that time has run out.

The last major design change was the introduction of the QSoundLoader class. The purpose of this class is to load sounds into the game, and allow the functionalities of playing those sounds at whenever desired by the application programmer. It follows a singleton pattern for multi-threaded access to the sound playing service: methods are synchronized, the constructor is private, and an instance of the class is exposed outside via a getInstance() method. Whenever a sound is requested, the QSoundLoader instance launches a new thread and plays the desired sound in that thread. The drawback of this method is that one can not stop a sound once it has started playing, however, this allows more than one sound to be played at the same time.

3. Future Implementation Goals

The drawback of the current implementation is that the logic of the modes and the core functionality of the game is not entirely connected. In the future we will be properly connecting everything so that the code is coherent.

Also, in the current version of the code, since there is a disconnect between the modes and the core functionality, mode specific file persistence has not yet been implemented, i.e. the QDiskPersistable interface and the QFileManager utility class are currently not deployed, and placeholder IO operations are present throughout the implementation: QThemeManager loads only one default theme, QSoundManager was hard-coded to read the audio from disk, QPlayer is persisted to disk in a specific program-defined format instead of XML, and QSettingsManager does not use any form of persistent format at all, given the lack of sound assets.

The lack of assets for the game is also another pressing issue for the continuation of the implementation. We do not currently have good backgrounds, images for themes, music, tiles for the treasure mode map, icons for player information, etc. which is undesirable for the final product.

In the future, we will be addressing these implementation goals by holding serious meetings on gluing the modes and the core functionality, rehauling disk persistence in the implementation by actually following the design, and using proper channels to obtain better assets for the game.

4. Lessons learned and Reflections

We were not able to provide the whole implementation of the game, but we have provided the core functionality of a quadrillion game. Our first challenge was to find a good design for the game logic. During the first iteration, we first learned how to make UML diagrams such as use- case, activity, state and sequence and apply the concept of the diagrams to our application and our source code. We have some experience in class diagram from CS 102 course, but we went over the details for each diagram type more thoroughly, because in this case we had a lot more classes and functionalities. We found the diagrams helpful, especially the class diagram because our actual implementation was not far from the conceptual design during our analysis and design phase.

We started writing the source code by following the documentation and notation of objects and methods written on the report. We practiced on using the MVC framework in order to have modularity in the code and decouple the major components. In this way, it was also easier to work in parallel and divide tasks, since each member of the group was working on separate components (UI, logic and controller). We had some difficulties during the user interface implementation regarding the layout of the frames and panels. Since the application we are developing is a game, it needs to have appealing graphics and display so we tried to make the GUI components fit good in the frames. For this reason, we used Eclipse WindowBuilder to build the main panels.

We also gained experience in using Github for sharing the source code and merging the project between members of the group. Moreover, we developed soft skills, by keeping regular group meetings, setting inner deadlines and balancing the workload.

5. User’s guide

5.1 System Requirements and installation

Depending on the type of installation, the user may be required to have the latest version of JRE and/or the JDK installed on their machine. If the user does not wish to acquire the source code for the installation/is merely consuming the product, then it is sufficient for the user to acquire the .jar file and then run it on their machine. In the case that the user wants to compile the code from ground up/take active part in the development of the game, the user should have the latest version of JDK installed on their machine. From this point, the user may download the source code from the repository and then compile the code, either using the terminal or an IDE that the user is familiar with. We recommend using the community edition of IntellIj to compile the project into a .jar file, along with the resources. The user then may access the application by running the .jar file.

System Requirements:

- 512 MB RAM
- 256 MB Disk Space
- JDK/JRE installation

5.2 How to Play

Running the .jar file will launch a menu as in Figure 1, from which one may select among different options. Clicking on the “PLAY” button will allow the player to choose a mode among the three game modes. Another panel will show up with three options to choose between the modes.

The player can choose to go to Settings by pressing the button “SETTINGS”, where he can change the volume or the themes. If the player presses “INSTRUCTIONS” he is redirected to a panel containing image with the instructions on how to play the game and game rules. If the player presses “SHOP” he is redirected to a panel where ha can buy different themes or power-ups using coins. The player can also press the “CREDITS” button where a panel containing the group information is displayed.

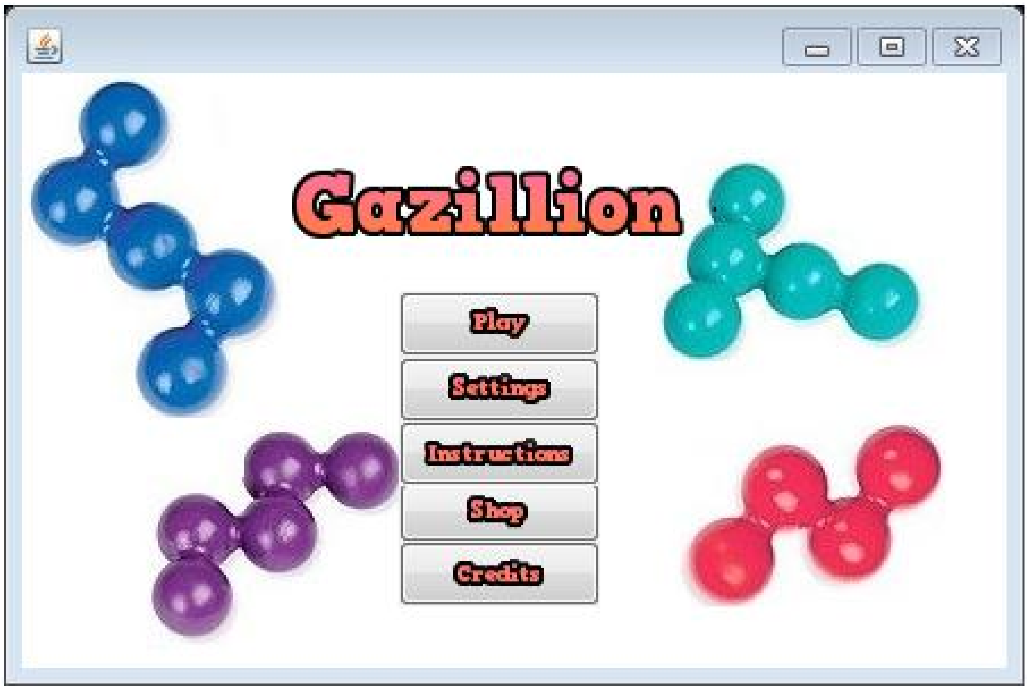
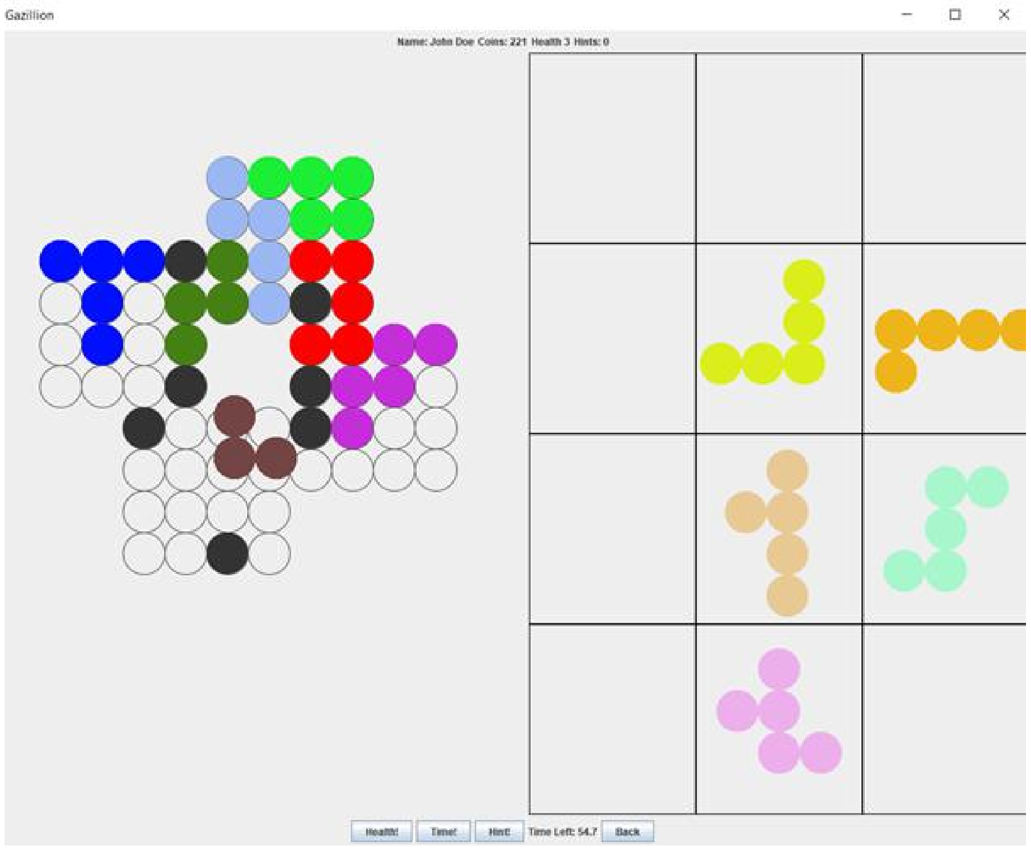


Figure 1. Main Menu Panel

After starting the game the user will be presented a panel like the following:



Name: John Doe Coins: 221 Health 3 Hints: 0

Time Left: 54.7

Figure 2. QGame Panel

The player may select a piece from the right-hand side 4x3 grid by clicking on the desired piece. Once the player clicks on the piece with a left click, the piece is “attached” to the mouse pointer, and the player does not need to drag the piece around by holding the left mouse button. The player can place the piece on the board by moving the piece over the board and pressing left click on the mouse. If the location is suitable, the piece is placed. If the location of the piece is not suitable, depending on the type of placement the player attempted, the piece is either returned back to the grid, or nothing happens. If the player attempts to place the piece on the board such that center coordinate of the piece is out of bounds of the board, it is returned back to its slot and the piece is deselected. Otherwise, nothing happens, and the player may still place the piece somewhere else. The player may rotate the piece clockwise or counter-clockwise using the mouse wheel. Scrolling the mouse wheel up results in a counter-clockwise rotation of the currently selected piece, and scrolling down results in a clockwise rotation. The player may also flip the selected piece along the horizontal axis using the right mouse button.

The player can remove a piece from the board by clicking on that particular piece when the player does not have any pieces selected. If the player wishes to remove a piece from the board while the player is holding a piece, the player should first return the piece to its corresponding slot by clicking out of bounds of the board, and then the player may remove a piece.

At any time, the player may use any of the powerups located at the bottom of the screen. Clicking on the “Health!” button will increase the player’s health, clicking on the “Time!” button will give the user more time to finish the level, and clicking on “Hint!” will give the user a hint in Treasure Mode.

The player has to complete the level before the timer ends, or the player loses the game. The current time remaining is displayed at the bottom of the screen.