Bilkent University
Dept. of Computer Engineering

Object Oriented Software Engineering Project

RoM: Redeemers of the Monarchy

Design Report

Project members: Efe Ulas Akay Seyitoglu, Erkam Berker Senol, Izel Gurbuz, Nashiha Ahmed
Course Instructor: Ugur Dogrusoz

Design Report (1-3)
November 12, 2016

This report is submitted to the Department of Computer Engineering of Bilkent University in partial fulfillment of the requirements of the Object Oriented Software Engineering Project, CS319

Contents

Design Report

1 Introduction

Our project is a "Tower Defense" type game. It will be a Java desktop application. Tower Defense is a strategy game. In Tower Defense games, players aim to defend their territories or possessions from enemies by destroying them before they reach the endpoint.

This design report is a kind of internal document, aimed to aid developers of the game. Specifically, this design report gives a description of the design of the game, which includes software architecture and subsystem services.

1.1 Purpose of the System

Redeemers of the Monarchy is a system designed to provide an easy-to-use and entertaining experience to the user. We intend to challenge the game player's strategic skills in this game of strategy. We aim to make the game fast and with as minimum bugs as possible. The game will have a simple, minimalistic user interface that is straightforward for the player to use.

1.2 Design Goals and Tradeoffs

1.2.1 Adaptability and Portability

The game will be programmed using Java. Java is a platform-independent language. Its platform-independence can allow it to work on all platforms that support Java, making it adaptable and portable to almost all platforms. Since our focus is to make the game a desktop application, it will not be available on the web or on mobile devices. However, we aim to code with the foresight to make the game available on the web and mobile devices.

1.2.2 Usability

The game will be easy-to-use through a simple and minimalistic user interface. We will use classic and straightforward user interface elements, keeping the design similar to existing games. The user interface elements will be consistent, as well. Input will be taken through classic keyboard shortcuts (arrow buttons, spacebar, enter key, esc, and "w, a, s, d OR i, k, j, l") and mouse-clicks. The tradeoff here is that the game will not be challenging or innovative in terms of UI and I/O; however, our main focus is to make the game challenging in its strategy- not challenging to use.

1.2.3 Reliability and Maintainability

To keep the system reliable with as minimum bugs, we will keep the system maintainable. We aim to do this by writing many tests. We may write test programs to test the code that we ourselves have not written to minimize programmer bias. We may do this by writing test programs for each other's code. We will also aim to keep the software and software logic simple, so that it is easy to maintain. We will keep our programs will documented so that we can review and develop each other's code. We aim to keep this design document simple and consistent as well so we are all "on the same page."

1.2.4 Extensibility

Because the game is written in Java and will be object-oriented, manipulations, development, and extensions to the program will be easy to implement. The object concerned will simply be tweaked. As explained in later sections, we will implement the game using the "Model, View, Controller" architecture. This will also benefit us in extensibility. For example, if we want to develop the model of the game, we may be able to do this without having to make tremendous changes to the view and controller.

1.2.5 High Performance and Efficiency

We aim to keep the framerate of our game around 30-40 frames per second. We also aim to keep a maximum of 2s response time. This is to ensure a smooth gameplay. Although we will try to avoid problems with high performance and efficiency completely, we aim to keep the user informed if a problem is encountered. For example, if the system is taking a long time to respond, we will display some sort of user interface that shows the user that the game is loading and has encountered a

problem. If the response time is too long, we will display a message to the user and restart the game.

1.2.6 Low cost

Our game is completely free since our game is mainly for this CS319 project. The tradeoffs of that is the game is like to not be maintained after the project is submitted. However, we aim to make the quality of the game as if it were to be purchased. We also aim to make the game such that it can be developed on in the future.

2 Software Architecture

2.1 Subsystem Decomposition

MVC ("Model, View, Controller") architecture will be used to design our system. The reason we chose MVC is because with MVC it is easier to simulate gaming experience into the software system compared to other architectures. As mentioned before, we also want to use MVC because it enables extensibility. MVC decouples data entity objects and data presentation. We can easily update the User Interface without greatly influencing the entity objects. Our system consists of three different subsystems namely the Model, View and Controller subsystems. These subsystems accomplish different tasks but they are dependent on each other. Relationships between them are depicted as in Figure 1.

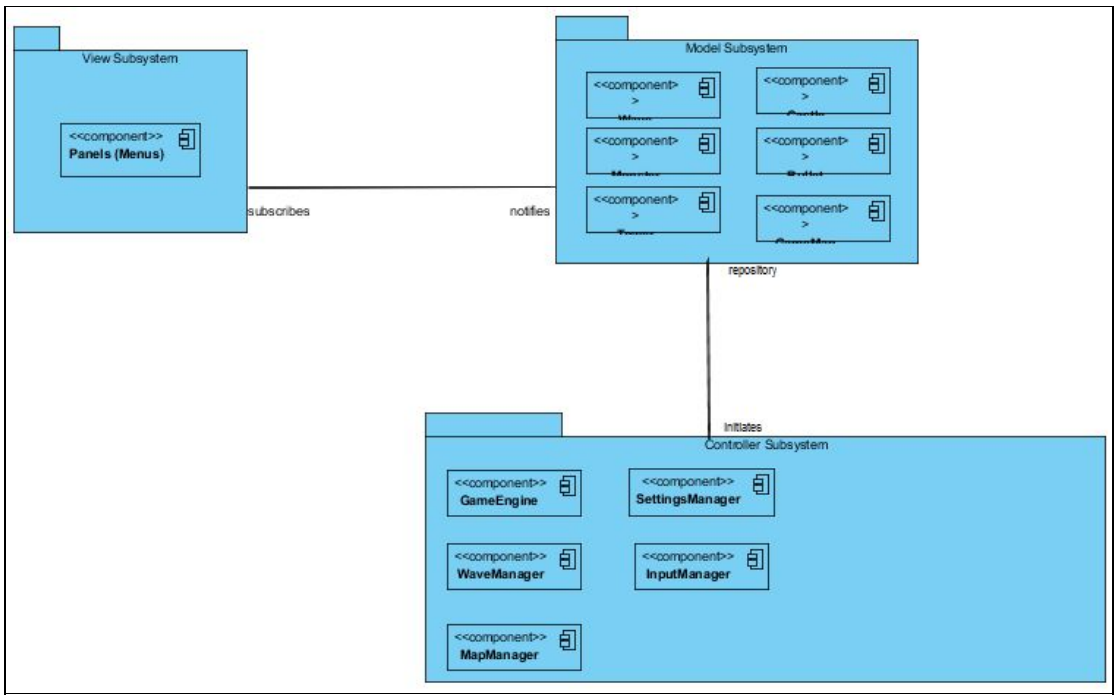

Figure 1-*Relationship between the subsystems*[1]

2.1.1 Model Subsystem

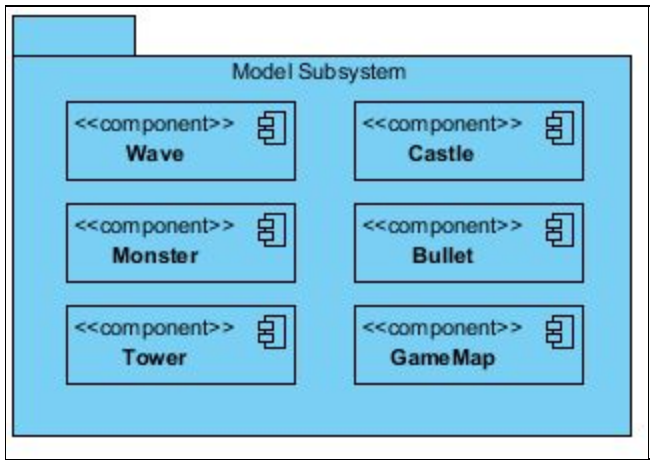

Figure 2 - *Model Subsystem*

In typical MVC patterns, the model is the core unit of the program. The data accesses or entity objects are in the Model subsystem. Model subsystem in our project will follow the traditional way of model subsystems to form the structure of the game.[3]

2.1.2 View Subsystem

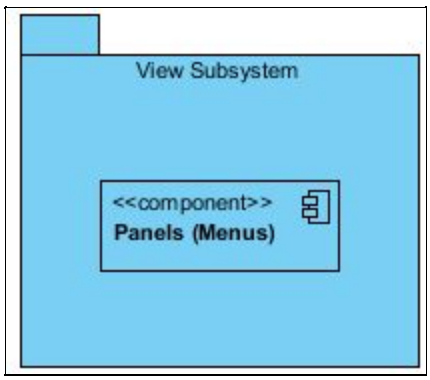

Figure 3 - *View Subsystem*

The View system represents data. It has the boundary objects, which will be the User Interface with which the user interacts. In the View subsystem, the menu interactions between the player and the game will be handled. Meaning, when the player pauses the game or enters the game at the first time the View subsystem will be notified. Furthermore at the end of the game, when the user is asked to provide information to be entered to the high-score list this sub-system will be notified. The View will have one frame and multiple panels. Thus, we will use the CardLayout in Java. Java Swing will be used for our User Interface.

2.1.3 Controller Subsystem

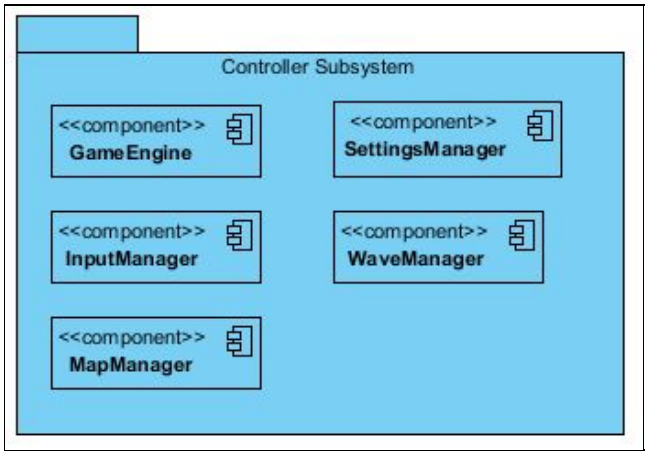

Figure 4 - *Controller Subsystem*

The Controller subsystem pieces together the Model and the View. It handles the user interactions with the boundary objects and notifies the entity objects as needed. As it is the case for programs that follow the MVC architecture, our system will also use the controller subsystem to dictate the control flow.[2] The actions that the player takes during the game will be managed by this component of our program.

2.2 Hardware/Software Mapping

The project will be implemented using Java and we will use JDK 8. The required hardwares for inputs are keyboard and mouse. Keyboard will be used for shortcuts and typing the name at high score table. Specifically, the arrow buttons (up, down, left, and right) and "w, a, s, d" or "i, k, j, l" can be used to navigate through selections. Enter button and spacebar selects an option. The mouse can be used to navigate through the application. A double left-click selects the user interface object it clicks over. The high score and latest settings will be saved in a .txt file in the memory. For sounds and music .MP3 files will be used and for images .jpeg files will be used, so the operating system should support .txt, .jpg/jpeg and .MP3 files. The computer does not need internet connection to run the game.

2.3 Persistent Data Management

We will not hold a database. The game's data, such as the pictures, settings, and high score list will be saved on files. These files will be on the user's hard drive and loaded when needed by the game. Sound files will be stored as .MP3 files and images are stored in .jpg/.jpeg formats. The data will be managed by the "Manager" classes.

2.4 Access Control and Security

The player will not need to sign up or in to play our game. Therefore, there will not be a password or identity check. The reason of this kind of implementation is that the player will only have access to the desktop application. Thus, access control and security is not a major concern. Many of our methods and classes will be made private to disable users from manipulating our code. Access to the files is only managed by our program, specifically the "Game Engine Class" and some "Manager" classes.

2.5 Boundary Conditions

When the user opens the .exe file (assuming the user had already installed the game to the computer), the system presents the main menu. Options given by the main menu include: Play, Help, Settings, High Scores, Tutorial, and Exit. In order for the game to start, the user needs to select play. When play is selected, the game engine class will respond by starting the game, which calls the necessary methods from other classes to start the game (MapManager loads the map). The game is paused if the player clicks the pause button during the game. Player is presented with options (Resume, Help, Settings, High Scores, and Exit.) Resume game returns back to the game. Exit terminates the game. The game displays high score at the end of every game, whether won or lost (a player can get a score even though he/she had not won). The high score table is only updated if the player has a high score above the tenth score in the high score table. The game will return to the main menu after the high score table is shown. If the game shuts down before the high-score of the player is entered, for whatever the reason may be, the high score is not saved.

JRE must be installed to run the game. If the platform the application is on does not support Java, an error message will pop up to the user. If the system takes more than 2s to load when the application is running, the user will be notified. There will be an internal loading bar. If the system takes longer than 1 minute, the application will be restarted. If the user runs into an error (any kind of run-time error during the game), an error message will pop up, and the user will be requested to restart the app. If the game is interrupted by power loss or forced termination, the game will start from the beginning when the user restarts the application. If the user tries to open the application while the game is already running, an error message will pop up and will request the user to terminate the current game before opening the application again. If any kind of internal material that the game needs are lacking, the user will also be notified.

3 Subsystem Services

Model-view-controller design pattern will be used in the implementation. The system will be decomposed into three subsystems. In this section each subsystem will be described.

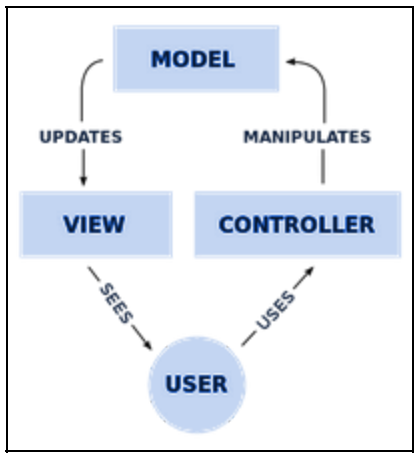

Figure 6 *- MVC Diagram [3]*

3.1 Model Services

The data of the objects are encapsulated in model subsystem, it responds to requests from the view subsystem about its current state and model can be updated with controller. Model can change with the effect from other sources such as user inputs or game operations. Controller is responsible for stimulating model service. Model updates its data with its own update methods which is triggered by controller. Furthermore model has connections with both the view and the controller as it is seen from figure 1.

3.2 View Services

The view sub-system contains all the panels that is in the interaction with the user. It is used to display the changes in the models. Model notifies the view (since they are in connection as in figure 1) . According to the given notifications the view updates to the user.

3.3 Controller Services

In order for the whole sub-system to work, the entity objects and other important classes need to be managed. This management is done through Controller -Manager- classes. For example, at the start of the game, the map will be loaded safely. In order to safely load the game the MapManager class manages the loading of the map. Furthermore, we have designed our controller sub-system in such a way that all our objects are controlled through such manager classes. These manager classes also interact with the user interface subsystem since the states of the objects are reflected on the user-interface throughout the game.

4 Low-Level Design

4.1 Object Design Trade Offs

4.1.1 Performance vs. Security

The trade-off between performance vs. Security is that increase in one of them leads to a decrease in the other. Meaning, as we wish to increase the performance, there occurs security leaks etc. Because of the constraints of this project and its purpose being just for this course, performance is preferred over security. A better running program is more preferable in our case. Due to the fact that our data is kept in a file system, our data may not be secure. While this (data being kept at the file system) ensures faster access to relevant data, images, or sound files, it in turn causes security problems. More security ,by applying safer methods to protect our data, would mean a decrease in performance since the system will be required to overcome that security appliance every time the system is run.

4.1.2 Space/ Memory vs Object Oriented Design

Since we are using object oriented design, our program takes up more memory than a non object oriented design. However, our object oriented design comes with man pros, some of which include extensibility and modifiability.

4.1.3 Decentralized Design

In our object design, the software control is decentralized. This means that the control is spread out over multiple objects. While this ensures better performance, avoiding performance "bottleneck," it is more difficult to create a change in control. The control is spread out.

4.1.4 MVC Architecture vs Extensibility

MVC design works well for our object design; however, if the game is further developed and becomes more complex(for example, dealing with more complex data or logic) MVC may no longer be a good solution to deal with a high level of complexity in our gaming. Our game is simple enough that MVC does not over-complicate the design, but it may limit the extensibility and developability of the game.

4.2 Final Object Design

4.2.1 Singleton Pattern

In order to avoid ambiguity about the interaction of objects. The *singleton design pattern* is used. The class map manager has this feature since via map manager everything is updated visually. Thus, there is a manager object within the map manager that coordinates all these events.[1]

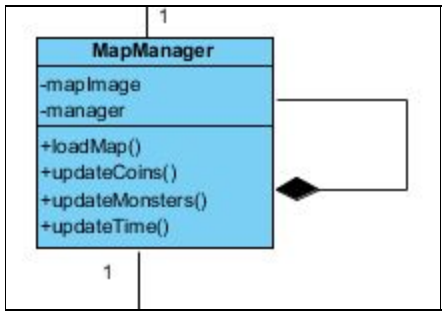

Figure 7 *-Singleton Design Pattern*

Via the utilization of this pattern the system operates more efficiently. Therefore the game logic is reflected on the system with ease.

4.2.2 Facade Pattern

Since our systems are the Model, View, and Controller subsystems there are some classes that implement the Facade Pattern so that the subsystems that communicate with each other. Our Object Design implements a Facade Pattern. For example, The Game Engine in the Controller subsystem communicates with the User Interface subsystem. The GameMap and GameObject classes also act as Facade classes because they communicate with the Game Engine and the Game Map Manager classes which are essential control classes.[1]

4.3 Packages

4.3.1 Model Package

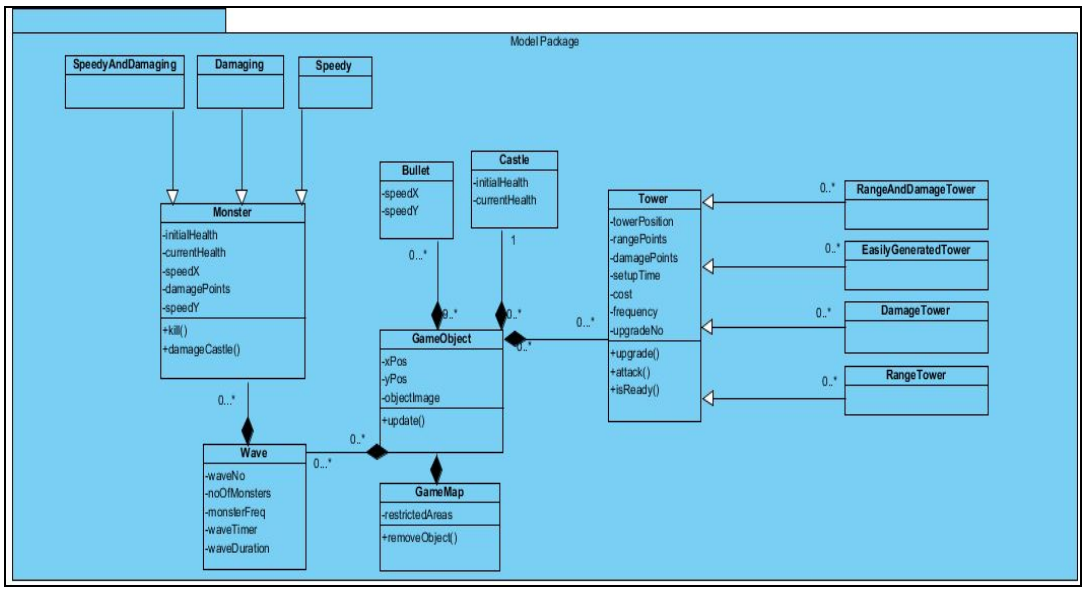

Figure 8 *-Model Package*

Model package is important since it contains the entity objects inside of the system. GameObject class is the class that interacts with almost all the entity objects. Key components such as: Tower, Monster, Castle & Bullet are included in this package. These entity objects interact with the controllers to apply MVC idea better throughout the game.

4.3.2 View Package

View package always makes use of controllers services. According to an event that the controller handles, the view of the game is updated. This package contains the following panels. Their features are explained below in section 4.4.1:

- MainFrame
- MainMenuPanel
- CreditsPanel
- HighScorePanel
- GamePanel
- PausePanel
- SettingsPanel
- TutorialPanel

They also interact with model since game logic influences the output (shown via UI components).

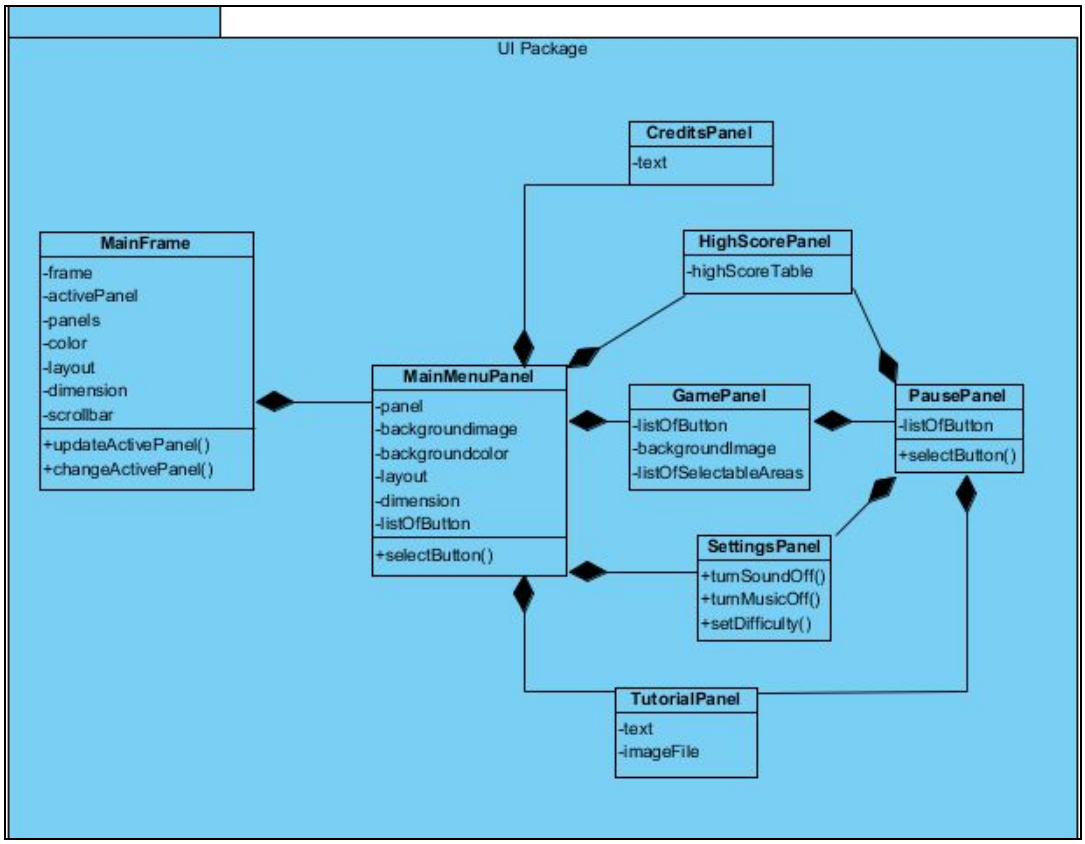

***Figure 9**- UI package*

4.3.3 Controller Package

Controller package provides control services for the subsystems. In this package there are manager classes and game engine. Game engine is the top class in the hierarchy in this package. The game engine class uses the other manager classes. The manager classes are:

- Input Manager: helps engine to get inputs from the user.
- Wave Manager: helps engine to deal with monster waves.
- Map Manager: helps engine with map operations such as adding a monster or collision checking.
- Storage Manager: deals with saving the high scores of the game.
- Settings Manager: used to change the sound/music level of the game and to change the difficulty.

The controller classes manage the changes on model classes. There are only one instance for each controller classes in the system. Thus, singleton pattern will be used for controller package. Controller package also provides notifications to user interface package.

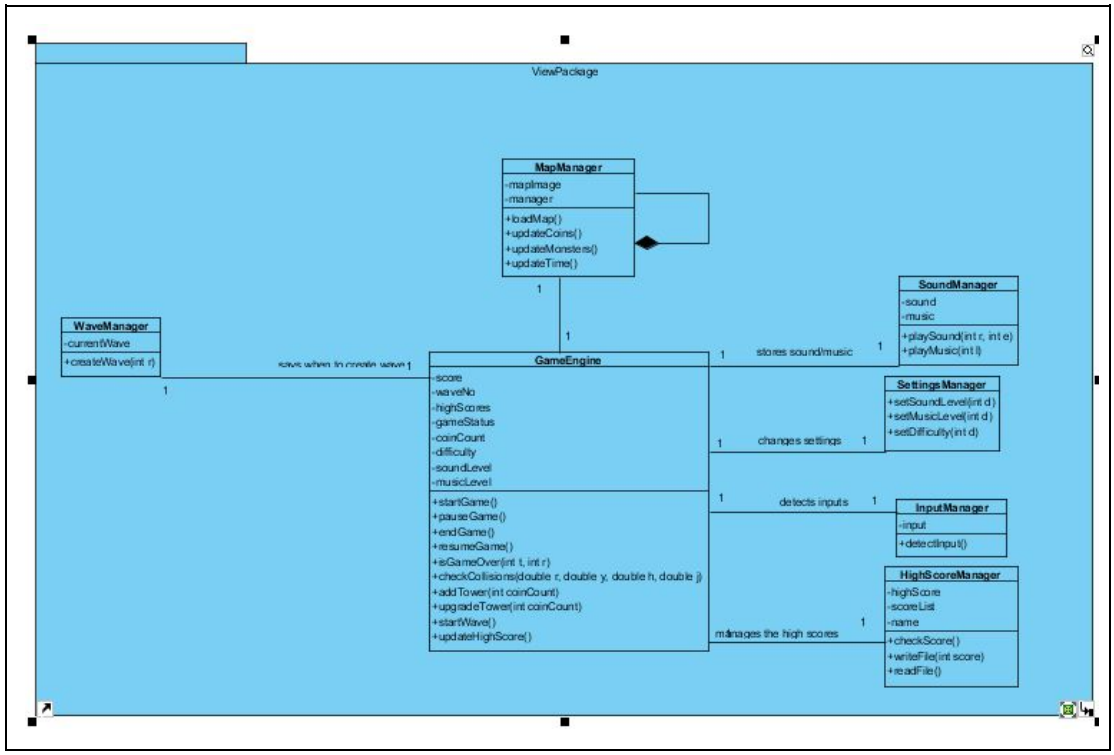

***Figure 10**-Controller Package*

4.4 Class Interfaces

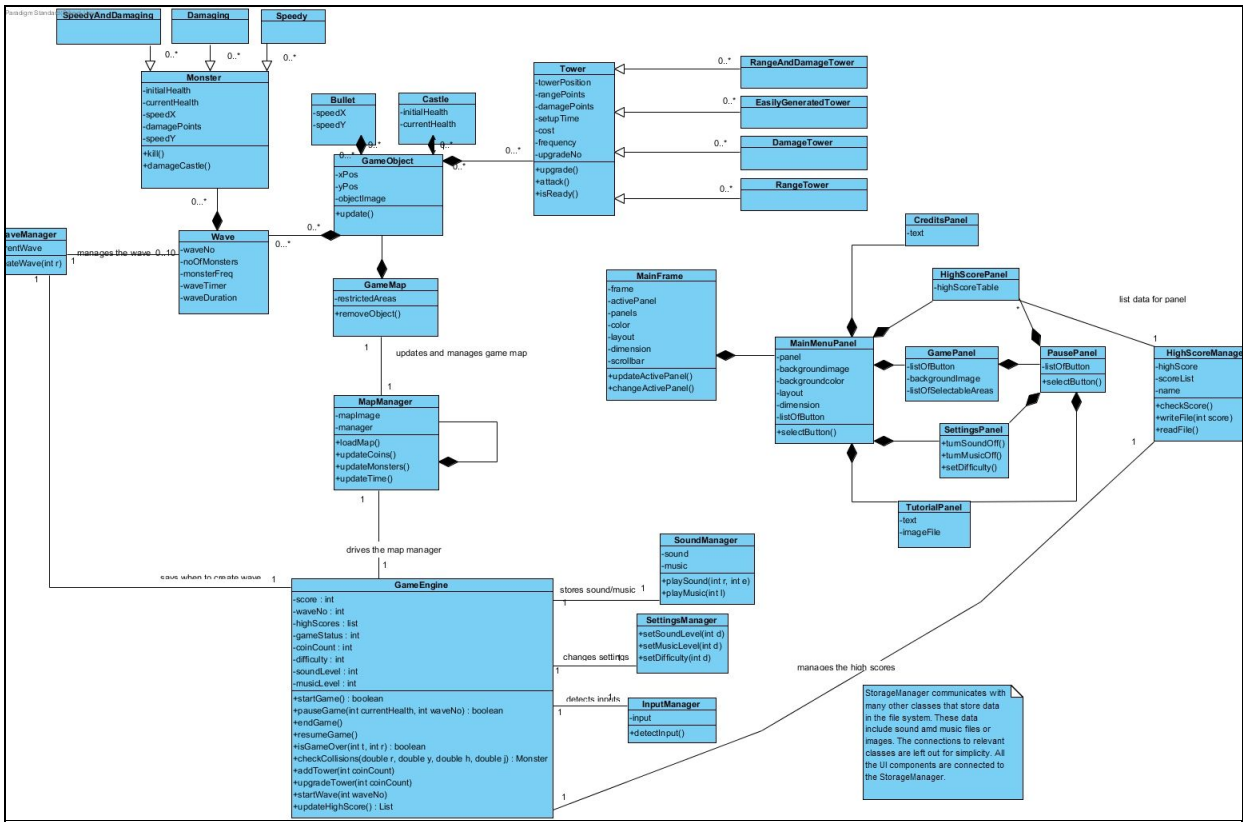

Figure 11*-Final Class Diagram*

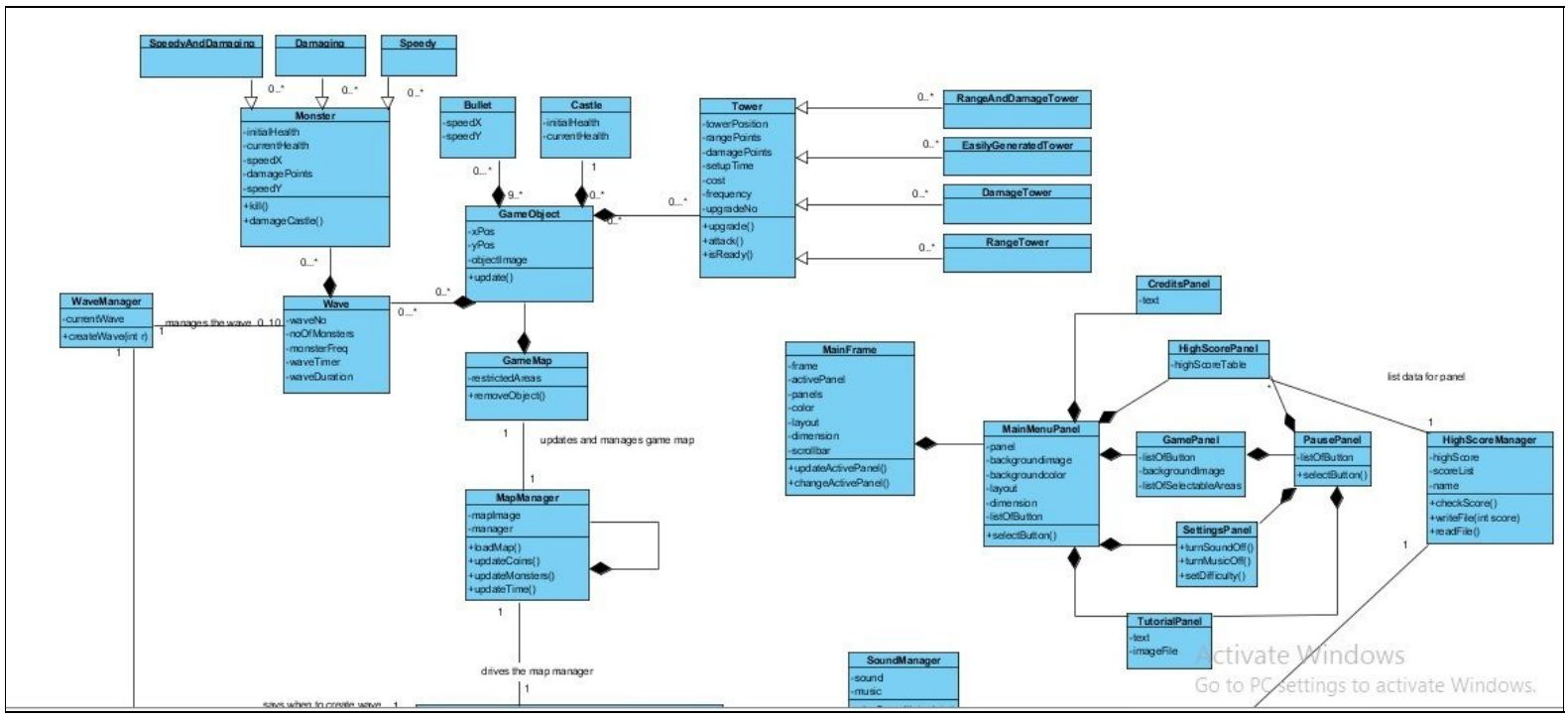

Figure 11A*- Part 1*

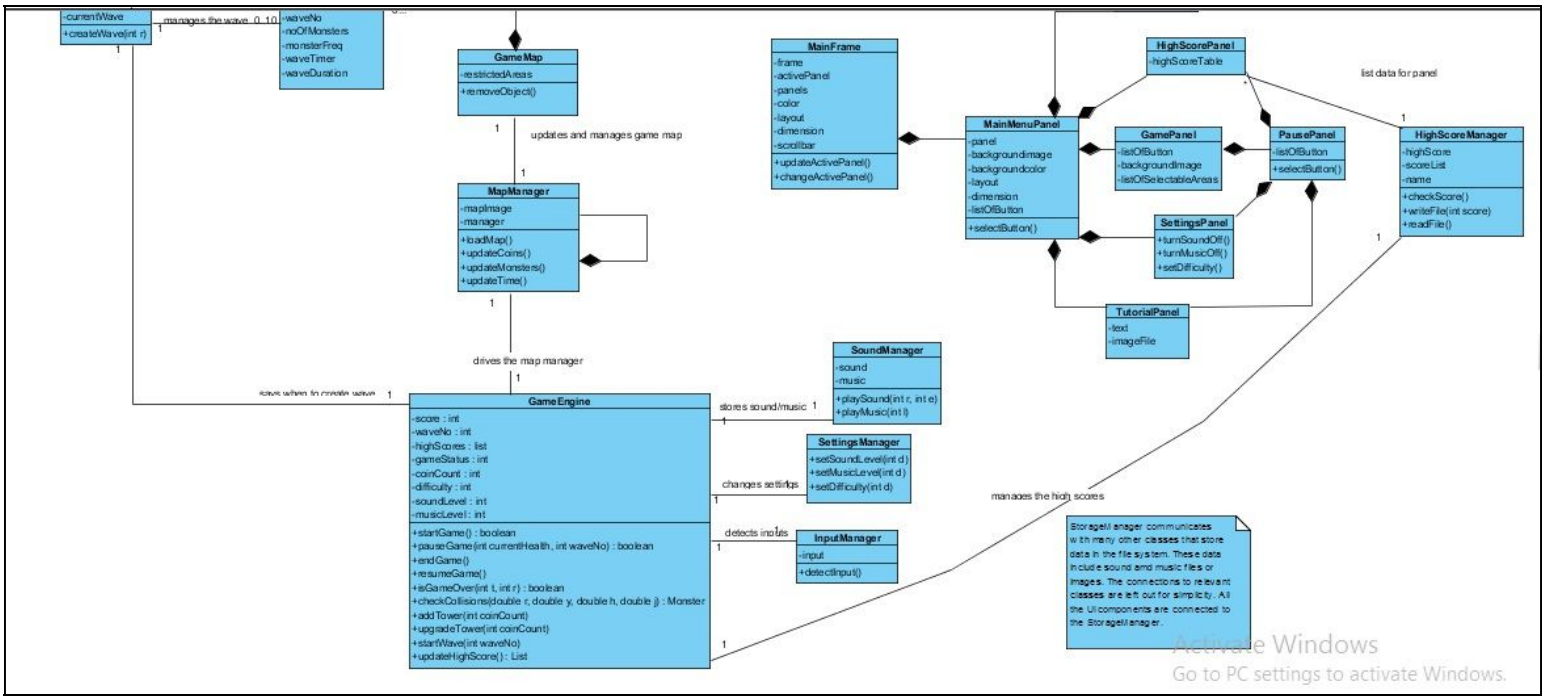

***Figure 11B**- Part 2*

MainFrame Class

As the name suggests this class is the main frame of our game. It directs which panel is active and which panel is not active. There exists a composition relationship between MainMenuPanel and MainFrame. Meaning, they cannot exist without each other.

Attributes of Main Frame

private JPanel activePanel: Denotes panel that is active. It s important to actively control this panel since many events will be occurring during run-time on this panel.

private Color color: Denotes the main colour of the game. Background color of the main frame can be modified via this attribute.

Methods of Main Frame Class

public void updateActivePanel(): Updates the current panel of the on-going game.

public void changeActivePanel(): Changes the actively used panel.

MainMenuPanel Class

This class is a central view component since it interacts with a lot of classes. This class has a duty to coordinate the main menu panel. Main menu panel is the first panel that the user encounters.

Attributes of MainMenuPanel Class

private Image backgroundImage: This image is the background of the main menu. When the user initially opens up the game, a main menu with this background will pop up.

private JButton[] listOfButton: There will be an array of buttons stored at this class. Each array element denotes an array in the main menu.

private Color backgroundColor: Denotes the background color of the main menu.
private FlowLayout layout: Our layout for this game will be of type FlowLayout since it provides flexibility.

Methods of MainMenuPanel Class

public void selectButton(): Sets the visibility of all buttons to true in the listOfButton array.

CreditsPanel Class

This class forms the credits page of or game. It is accessed via the MaiMenuPanel thus, there is a composition relationship between each other.

Attributes of CreditsPanel Class

private JTextField text: Contains the text to be shown to the user.

HighScorePanel Class

This class is the high score page of our game. It is accessed via the MaiMenuPanel thus, there exists a composition relationship.

Attributes of HighScorePanel Class

private JTable highScoreTable: Contains the high score table.

GamePanel Class

This class is where the game takes place. Moving of monsters, building of towers etc. are shown via this panel.

Attributes of GamePanel Class

private double[][] listOfSelectableAreas: This attribute denotes the listOfSelectable areas. This is a 2-D array since the first element of the array denotes the center point of the area in which a tower can be place whereas the second element denotes the radius of the oval. (Selected areas will be in the form of oval).
private Image backgroundImage: Denotes the background image of the game arena.
private JButton[] listOfButton: List of buttons that indicates what kind of tower will be placed when the cursor is on the selected areas.

PausePanel Class

Is the panel that shows up when pause button is selected.

Attributes of PausePanel Class

private JButton[] listOfButton: Denotes the available list of available buttons when the pause menu is selected.

Methods of PausePanel Class

public void selectButton(): Sets the visibility of all buttons to true in the listOfButton array true.

SettingsPanel Class

Is the panel that shows up when the user wishes to change the settings.

Methods of SettingsPanel Class

public void turnSoundOff(): turns the sound off.

public void turnSoundOn(): turns the sound on.

public void setDifficulty(): sets the difficulty of the game.

TutorialPanel Class

Is the panel that shows up when the user wishes to see the tutorial.

Attributes of the TutorialPanel Class

private JTextArea text: text that contains the tutorial.

private Image imageFile: provides visualization to the tutorial

4.4.2 Controller Classes

Game Engine Class

The GameEngine class is the class with the main control of the game. It has the most essential game logic methods. The ones with interesting behavior will be described in the following sections. Methods are attributes that are simple and obvious by name will not be explained.

Methods of the GameEngine Class

public boolean startGame():

This method as the name suggests starts the game. When the user clicks the play button in the main menu, the system runs this method. This method in return calls essential methods from other classes and instantiates all the necessary objects needed to start the game. For example, it calls loadMap() which creates a Map object. It also calls the startWave() method, so that the monsters start entering the path on the map. The start game instantiates the GamePanel as well which is the UI representation of the game. It returns true if the game started.

public void pauseGame():

This method pauses the game. When the user clicks the pause button during a game, this method is called. The pause method essentially freezes the game. Variables such as castle state, monster states, tower positions, coin count, and game time are all saved. The pause method calls the relevant UI class, PausePanel, which shows up on the MainFrame. From the pause game, the user can run the SettingsManager and the SettingsPanel or view tutorial through the TutorialPanel, depending on which button the player selects. If the player clicks on the resume button, the game will resume with all the states that the game had before it was paused. The player can also quit the game after pause game.

public void endGame():

This method ends the game. No states or variables are saved. Once the game ends, if another game starts, the game with a "clean-slate." The endGame method is run when the user successfully or unsuccessfully finishes the game. The high score is calculated and the high score table is updated.

public boolean isGameOver(int currentHealth, int waveNo):

This method checks whether the game is over by checking the castle's currentHealth and the waveNo. The the 10th wave is finished and/or castle's currentHealth is 0, the game is over.

public Monster checkCollision (double mX, double mY, double bX, double bY):

This method checks if the bullet or bullets have hit any monster. This is done by checking the final x and y positions of both the bullet and the monster. If there are the same, then the monster has been hit. If the bullets have hit monsters, the monster which has been hit is returned.

pubilc void addTower(int cointCount):

A tower is only added if the user has enough coins to buy the specified tower. The tower is selected by clicking on the tower image in the GamePanel. The tower is added on of the emty spots to the specified x and y coordinates on the game map. The method returns the tower that was added. upgradeTower() works in a similar way.

public void startWave(int waveNo):

The startWave method starts a wave in the game. At the beginning of this method, the waveNo in the Wave class should be zero. The startWave calls the createWave method in the WaveManager class and updates the game map and UI components accordingly.

public List updateHighScore():

The updateHighScore method updates the high score table when the game is ended. The updateHighScore method sends the high score and name of the user to the HighScoreManager class, which the user inputs after the game ends on the HighScorePanel. The updateHighScore method checks if the game has ended by calling the isGameOver method. It returns a list with the name and high score of the player.

Input Manager Class

Input Manager is the controller class for inputs of the game. It manages the inputs from user and notifies Game Engine Class.

Methods of the Input Manager Class:

public void detectInput()

This method understands the input coming from the user. It notifies the game engine class. Mouse clicks and keyboard shortcuts are taken in this class.

High Score Manager Class

High Score Manager is responsible for reading and writing data to local storage for high score table.

Methods of the High Score Manager Class:

private void writeFile(int score):

At the end of the game, game engine sends the score of the game to high score manager class and the class checks the .txt file if the score is in top 10, if it is the high score file is updated with the new score/name.

private List<String> readFile():
This method is used to read the high score .txt file.

Settings Manager Class:
Settings Manager is responsible for changing the settings of the game that are changed by user. The class is responsible for changing the difficulty of the game and changing the sound/music level of the game.

Methods of the Settings Manager Class
private void setDifficulty(int difficulty):
This method is used to change the difficulty of the game, when user changes the difficulty this method notifies the game engine class to make necessary changes according to game logic.
private void setSoundLevel(int soundLevel):
This method is used to set the sound level of the game.
private void setMusicLevel(int musicLevel):
This method is used to set the music level of the game.

Wave Manager Class
This class is responsible of creating and managing the monster waves according to the notification from game engine class.

Attributes of the Wave Manager Class
currentWave: This property holds the value of current wave.

Methods of the Wave Manager Class
private void createWave(int difficulty):
When the game engine notifies wave manager class to create a new wave, according to the current wave and the difficulty of the game. Then it creates a wave of monsters, increments the currentWave and then notifies the game engine.

Map Manager Class

Attributes of the Map Manager Class
mapImage: This property holds the image of the map.

Methods of the Map Manager Class
private void loadMap():
When this method is called, the map image is loaded to the frame.
private int updateCoins():
The coins of the user is updated according to the changes on the map, this method handles the situation.
private void updateMonsters():
When new monsters are added to the game, game engine notifies map manager to add the

monsters to the map. This method adds the monsters to the game map.

Sound Manager Class

This class stores the paths for sound/music files and plays them when game engine notifies.

Attributes of the Sound Manager Class

private void playSound(int id, int soundLevel):

This method is called by game engine class with specific things (i.e monster died, tower created etc.) to make the proper sound effect.

private void playMusic(int musicLevel):

This method is used to play the music sound.

4.4.3 Model Classes

Monster Class

Monster class is the blueprint for generating monster objects. Meaning, even though there are varying number of monsters, each monster has common characteristics. To elaborate on these characteristics there are three classes that extend from the class Monster.

Monster Class Attributes

private int initialHealth: Denotes the initial health of the monster.

private int currentHealth: Denotes the current health of the monster.

private int speedX: Denotes the speed of the monster.

private int damagePoints: Denotes the damage the monster causes to the castle.

Monster Class Methods

public int kill(): Takes away the monster from the game map (destroys monster).

public int damageCastlel(): reduces the damage point of the monster from the castle health.

Wave Class

Wave Class Attributes

private int waveNo: denotes which wave the game is currently in.

private int noOfMonsters: is the attribute that demonstrates how many monsters there are in a given wave.

private int monsterFreq: denotes the monster frequency per minute.

private Time waveTimer: is the timer that coordinates the game

private int waveDuration: denotes the duration of the game in seconds.

GameMap Class

GameMap Attributes

private double[][][] restrictedAreas: denotes the restricted areas on the map. First & second element denotes the center location of the circle whereas the third element shows the radius of the circle.

GameMap methods

public void removeObject: Removes any object from the GameMap.

Bullet Class

Attributes of Bullet Class

private int speedX: denotes the X coordinate the bullet is headed.

private int speedY: denotes the Y coordinate the bullet is headed.

Castle Class

Attributes of the Castle Class

private int initialHealth: Denotes the initial health of the castle.

private int currentHealth: Denotes the current health of the castle.

Tower Class

Tower class is a blueprint for towers which destroys monsters.

Tower Class Attributes

private int[][] towerPosition: Denotes the position of the tower. (x coordinate as the first element, y coordinate as the second element)

private int rangePoints: Denotes the radius of the circle that the tower will be able to attack.

private int damagePoints: indicates the damage point caused by the tower to the monster.

private Time setupTime: denotes the setup time to build the tower.

private int cost: denotes the cost of the tower to be build.

private int frequency: denotes the frequency (shot per second) that the tower fires.

private int updateNo: denotes which upgrade the tower is currently in.

Tower Class Methods

public void upgrade(): upgrades the tower so that the tower has better capabilities.

public int attack(): returns the damage points the tower possesses.

public boolean isReady(): returns whether the tower is ready to attack (a tower may not be ready to attack since the setup time might not have elapsed)

GameObject Class

Game Object is a blueprint for all the objects in the game.

Attributes of GameObject Class

public int xPos: denotes the x position of the game object.

public int yPos: denotes the y position of the game object.
public Image objectImage: is the image of the object

Attributes of GameObject Method

public void update(): updates the game object.

5 References

[1] "CS 319 Object-Oriented Software Engineering." Accessed November 11, 2016. http://www.cs.bilkent.edu.tr/~ugur/teaching/cs319/.

[2] Object-Oriented Software Engineering, Using UML, Patterns, and Java, 2nd Edition, by Bernd Bruegge and Allen H. Dutoit, Prentice-Hall, 2004, ISBN: 0-13-047110-0.

[3] MVC Process, Viewed 12 November 2016, <https://upload.wikimedia.org/wikipedia/commons/thumb/a/a0/MVC-Process.svg/2000px-MVC-Process.svg.png >.