CS 319 - Object-Oriented Software Engineering

Project Final Report

Medieval Tower Defense

<u>Group 1-A</u>

Buğra Aydın

Deniz Başaran

Mahir Özer

1. Introduction

1.1 Introduction

Medieval Tower Defense is a basic tower defense type video game we decided to develop. There are lots of different tower defense games with different objectives. The main purpose of tower defense games is to prevent enemies from reaching a certain point or destroying a base at that point, by marching along a predetermined path. In order to achieve this, the player sets up a defense of their own design by using the tools provided to them by the system. This is usually done by building towers with resources, customizing them in a way of their choosing according to their strategy.

The game we were influenced by is following game mods from the game Warcraft III.
https://www.epicwar.com/maps/159097/
http://www.moddb.com/mods/element-tower-defense/news/element-td-40-released
Medieval Tower Defense (MTD) will be developed with different features. In MTD we will add new types of waves like boss fights and various difficulty settings.
Classic tower defense game rules will apply. The player will face fifty waves of enemies with the defense they built and prevent them from reaching to the exit point. The player will be able to face the next wave after cleaning wave that is coming. The game is designed to be reusable, since the players will want to defeat the game and obtain a spot in the high score list.
The game will be a desktop application and it will be controlled by a mouse.

1.2 Game Overview

MTD is a medieval themed tower defense game. All tools, waves and bosses are from medieval times. Player faces a wave with towers built by spending resources provided to them at start. After killing each enemy, player gains different amount of resources and the player can choose to build more towers or upgrade towers they have already built. Goal of MTD is clearing all waves, killing bosses to face next wave and next boss until waves are finished.

1.2.1 Gameplay

Player will need a mouse to play the game. Keyboard will be available to perform some actions such as entering a nickname for high score screen when the game ends. Player will choose and place tower to the map with mouse. Towers can be placed on top of the areas which are made available to player. After given time finishes, waves starts to spawn and player's towers shots them and either kill them or fail to kill them and enemies will pass exit. There are 50 waves. Wave difficulty can be changed with difficulty settings. Player is allowed to miss 50 enemies but if that amount is reached, player will lose and start again from the beginning. Players will be able to customize tower according to their strategies and type of enemies.

1.2.2 Waves

Game progresses one wave at a time. Meaning second wave can not start before all enemies from first wave are killed or missed. A brief waiting time will be given to player between waves. A player who has established a fine defense which can hold on for several waves can use fast forward so that waiting time for those waves are shorter.

Information of each level are predefined in one of the classes.

1.2.3 Building and Customizing Towers

Player's defense are consist of towers they build. The player can place towers on areas they are allowed to and they will be able to customize them with game progress and resource gain.

1.2.4 Tower Types

There are different tower types and each type will have their own specifications. Tower specs are as follows.

- Slow Rating
- Range
- Damage (siege, infantry)
- Fire Rate

Each tower type will specialize in different ways. Tower types are a follows.

Tower name	Range/Attack Frequency	Attack Point/Damage Type
Arrow Tower	Medium/High	Low/single target
Canon Tower	High/Low	Medium/splash damage
Ice Tower	Medium/Medium	low/slow debuff
Oil Tower	Very Low/Low	High/area damage
Poison Tower	Medium/High	Medium/Victim bleeds
Arcane Tower	High/Medium	Very low/Armor reducement
Ballista Tower	Very High/very low	Very High/single target
Mortar Tower	High/low	Medium/single target

1.2.5 Enemies

Each wave will have types of enemies with their own specs.

- Speed
- Armor
- Health

1.2.6 Enemy Types

Invader Name	Speed	Health	Armor
Footman	Medium	Low	Low
Catapult	Low	Medium	Medium
Knight	Low	Medium	High

Light Cavalry	High	Low	Low
Elephant Rider	Low	High	Medium
Trojan Horse	Low	Very High	High
Chengiz Khan & riders	High	Very High	medium
Battering Ram	Low	High	Very High
Jester	Very High	Medium	Medium
Saint John’s Knigths	Low	High	Very High
Pope	Medium	Very High	High

2. Project Management Plan

2.1 Project Organization

We first thought on the idea of a Tower Defense game. Then with the instructor’s advices on the analysis and design, we wrote the analysis and design reports to organize the classes in a neat way to have an easy implementation.

2.2 Risk Analysis

Once generated, enemies are not deconstructed from the map, yet they only disappear from the current game window. This may lead to the performance issues when the enemies are accumulated in the map. Currently, stabilization may be needed, unless the game might consume much more memory than it should. Game size is not very large and not very likely to be large at the future.

3. Requirement Specificaton

3.1 Functional Requirements

3.1.1 Play Game

The game will start after play game button is pushed. The isometric game map that includes a path for AI invaders and free sites to build towers will be loaded. The AI invaders will be on the path to the final location and this final location is the keep that player has to protect with the towers built. In addition, any invader who manages to trespass the keep will cost the player a health point and player will have 50 health points at total. There will be various types of towers with different characteristics. Towers can be upgraded for better attack points, and better crowd control ability, also new towers can be purchased with sufficient amount of coin, which can be gathered by eliminating invaders with existing towers. Since Medieval-Tower-Defense is a wave-based game, there will be a number of 50 waves. At each 7 or 8 level player will face with a boss invader, which is considerably harder to beat, yet generates a lot of coins when defeated.

There will also be different types of invaders that have varying attributes of speed, health, and armor. There is another list available that shows the invader types. First 5 are minions and remainder 7 are boss units, which are more challenging special units.

3.1.2 Settings

Gameplay and music sounds are the features which can be set by the player. In each time when the game application is started, volume of the gameplay and music sounds, and visual quality are predefined in default settings. In order to change these default settings, user may enter the settings page from the main menu and adjust the volume with slide bar.

3.1.3 High Scores

One of the main objectives is to add High Scores bar user in which will be able to see the list of the highest scores, yet chart is accessible only if the user has internet connection. User's own score and other users with top scores will be visible in High Scores section. In this case, there will be a more competitive environment which means more game activity. The high score list will be implemented using a database. It is required to update and store this list.

3.1.4 Pause Button

The user can pause the game in progress any time desired by pressing pause button. Game time will be frozen as long as the pause button is pressed again. Button will be located at the top left corner during gameplay. If the player turns back to the main, game will be auto-paused, and player will have to click continue game in order to proceed more in the current game session. User will lose the achieved progress if application's closed during pause stance.

3.1.5 Help and Information

This section serves as a manual which represents the overall game concepts. The user can get information about: map pattern, game purpose, health, resource gathering, invader types/attributes, and tower types/attributes. Help document is accessed through the main menu, or anytime gameplay is in progress. There may also be a subsection in which game tips can be accessed.

Medieval Tower Defense game will have historical figures such as Chengiz Khan and St. John's Knights in it. Player's will be able to read information about historical characters and technologies from the information screen(tech tree).

3.1.6 Credits

The User may visualize the list of developers by pushing the credits bar of the main menu. Contact information for communicating the developers will be available as well.

Suggestions, comments, and bug reports may be sent directly to the developers with information available in this section.

3.1.7 Exit

Exit button is pretty self-explanatory. The application will be closed completely when the button’s pushed. Paused game session will be lost and settings will be set on the default mode in the next game launch of application.

3.2 Non-Functional Requirements

• Game graphics will be designed in order to be appealing. There will be no flickering and movement of the units and the projectiles will be smooth. Images of graphical objects will be chosen to give the players better gameplay experience.

• Control mechanisms of the game will have low response time, which enables users to play with minimal delay.

• The code will be efficient, allowing for no framerate drops during gameplay.

• The game will be readable, the player should understand what to do as long as they enter the game.

• The game art should not be too confusing to make it easier for the player to easily see what’s going on on the map.

3.3 Pseudo Functional Requirements

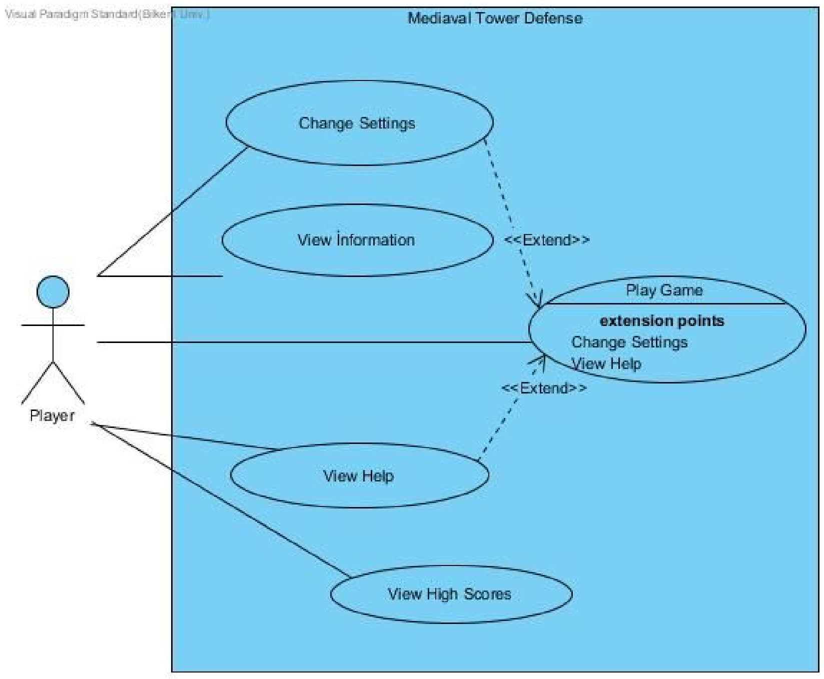
• The game will be implemented in Java.

• UI Design will be implemented with java Swing.

• Game art will be done using Adobe® Photoshop CC, Balzamiq.

4. System Model

This section provides information about the main use case model of MTD game, detailed use case explanations are included below.



4.1 Use Case Model

4.1.1 Play Game

Use case 1: Play Game

Actors involved: Player

Aim and System Response:

-Player aims to choose and place towers efficiently to not lose any lives. Losing a life means letting an enemy through, therefore also losing gold and reducing the chance of win.

-System keeps the score of the Player.

Entry condition: Player has opened the game and selected Play Game from the main menu.

Exit condition:

1. Player has won the game with or without achieving a high score, OR

2. Player has lost, i.e., letting through more enemies than his/her lives, OR

3. Player has exited the game from the in-game menu.

Preconditions: Player defined settings are used when starting a game if the player hasn't visited & changed the settings tab from the main menu, pre-defined settings are used when Play Game is selected.

Post-conditions: If the player's score is high enough to be on the leaderboard, it is added to the highest scores list.

Event Flow:

1. The player starts the game from the first level with the default life count.

2. Gameplay screen opens as a stack on main menu.

3. Player constructs his/her tower system in the given 30 second time before each wave appears. The player can also add towers when the wave is spawn.

Flow A:

4A. Player successfully finishes all the levels, i.e., not letting through more enemies than his given lives.

5A. The player's score is displayed on screen. If the player is among the top ten scorers, he enters his name to be added to the high scores list.

6A. Player returns to the main menu.

Flow B:

4B. Player lets through more enemies than his lives.

5B. The player's score is displayed on screen. If the player is among the top ten scorers, he enters his name to be added to the high scores list.

6B. Player returns to the main menu.

Flow C:

4C. Player pauses the game while playing, accessing the in-game menu.

5C. Player decides to quit from the in-game menu before finishing the game.

Flow D:

6D. Player decides to continue playing.(Back to step 2)

4.1.2 Change Settings

Use Case 2: Change Settings

Actors involved: Player

Aim and System Response:

-Players want to change the settings such as difficulty, sound volume, etc.

Pre-condition: Default game settings are given by the system each time the game is started.

Post-condition: Game settings are changed by the player.

Entry Condition: Player selects the “Settings” button from the main menu or from the in-game menu.

Exit Condition: Player selects “Back” to return to the menu

Event Flow:

1. Player clicks on the “Settings” button from the menu.

2. Settings panel opens as a stack on the main menu.

3. Game settings are displayed with interactable buttons.

Flow 1:

4A. Player changes settings.

5A. Player returns to the menu by pressing “Back.” The changes are saved.

Flow 2:

4B. Player returns to the menu by pressing “Back.” No new settings have been saved.

4.1.3 View Help

Use Case 3: View Help

Primary Actor: Player

Stakeholders and Interests:

- Player needs help in how to play the game.

- Player is given explanations on how certain aspects of the game work, the information is given in text format by the system.

Entry Condition: Player selects “View Help” from Main Menu or in-game Menu.

Exit Condition: Player selects “Back” to return to either main menu or the in-game menu.

Pre-conditions: Player is in the main menu or in-game menu.

Post-condition: -

Event Flow:

1. Player click on “Help” on the menu.

2 Help panel opens as a stack on the main menu panel .

3. A text explaining the controls and the aim of the game is shown.

4.1.4 View Credits

Use Case 4: View Credits

Primary Actor: Player

Aim and System Response:

- Player aims to get information about the developers.

- Name and contact information of the developers is displayed.

Pre-conditions: Player should be in the Main Menu .

Post-condition: -

Entry Condition: Player hits the “Credits” button from main menu.

Exit Condition: Player hits the “Back” button to return back to the main menu.

Event Flow:

1.Player clicks on the “Credits” button from the menu.

2 Credits panel opens as a stack on the main menu panel .

3.Information regarding the developers is displayed.

4.Player returns back to the menu by hitting the “Back” button.

4.1.5 View High Scores

Use Case 5: View High Scores

Primary Actor: Player

Aims and System Response:

-Player aims to see the leaderboard.

-The system shows the top ten scorers with their saved nicknames.

Pre-conditions: System should save the high scorers after every game instance is ended.

Post-condition: -

Entry Condition: Player hits the “High Scores” button from the main menu.

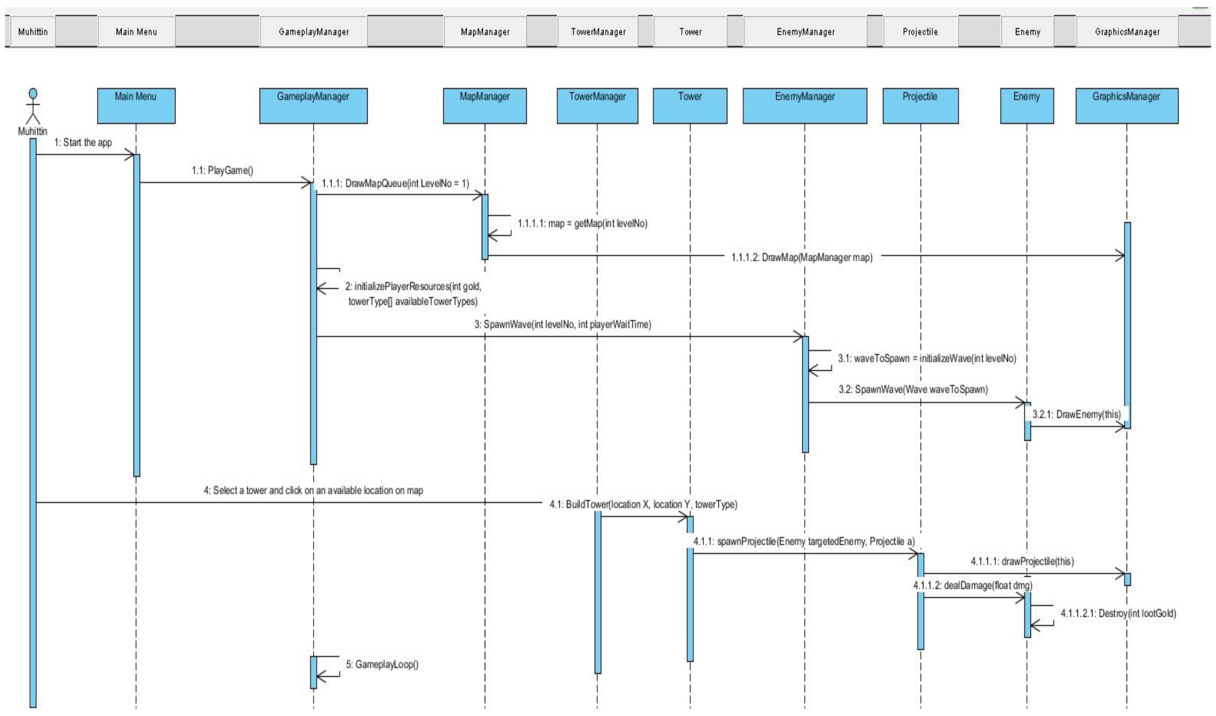
Exit Condition: Player hits the “Back” button to return back to the main menu.

Event Flow:

1. The player hits the “High Scores” button.
2. High Scores panel opens as a stack on main menu.
3. The system shows the nicknames of the top ten scorers.
4. The player hits the “Back” button when he is done examining the high scorers.

4.2 Dynamic Models

4.2.1 Start Game


Muhittin
Main Menu
GameplayManager
MapManager
TowerManager
Tower
EnemyManager
Projectile
Enemy
GraphicsManager
1: Start the app
1.1: PlayGame()
1.1.1: DrawMapQueue(int LevelNo = 1)
1.1.1.1: map = getMap(int levelNo)
1.1.1.2: DrawMap(MapManager map)
2: initializePlayerResources(int gold, towerType[] availableTowerTypes)
3: SpawnWave(int levelNo, int playerWaitTime)
3.1: waveToSpawn = initializeWave(int levelNo)
3.2: SpawnWave(Wave waveToSpawn)
3.2.1: DrawEnemy(this)
4: Select a tower and click on an available location on map
4.1: BuildTower(location X, location Y, towerType)
4.1.1: spawnProjectile(Enemy targetedEnemy, Projectile a)
4.1.1.1: drawProjectile(this)
4.1.1.2: dealDamage(float dmg)
4.1.1.2.1: Destroy(int lootGold)
5: GameplayLoop()

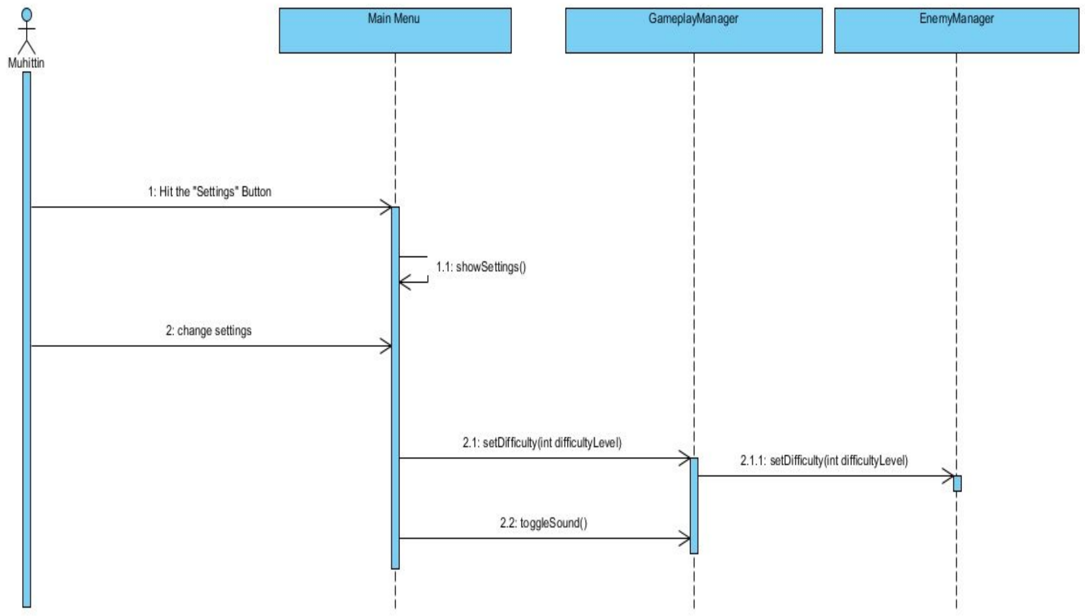
Event Flow:

Player Muhittin clicks the Play Game button to start playing. Then the MapManager system draws the pre-defined level and the pathway for enemies, then MapManager calls the GraphicsManager to display the graphics objects for these elements. Then the system gives the instructions and waits 30 seconds for the player to build towers. The player can then build the available towers with his resources.

EnemyManager creates the Enemy instances, each enemy instance calls the GraphicsManager to be drawn on the map. In the same manner, each tower instance calls the GraphicsManager to be drawn. For each enemy destroyed, the player receives resources to build/upgrade new towers. The amount of resources given to the player depends on the attributes of the enemy that is destroyed. Once all the enemies of a particular wave is either destroyed by the player or has crossed the end of the pathway, the system then goes into the gameplay loop for the next wave and again waits 30 seconds for the player to prepare new towers.

Once the player wins or loses the game, the system asks for the nickname of the player if his score is in the top 10 among all scores so far.

4.2.2 Change Settings


Muhittin
Main Menu
GameplayManager
EnemyManager
1: Hit the "Settings" Button
1.1: showSettings()
2: change settings
2.1: setDifficulty(int difficultyLevel)
2.1.1: setDifficulty(int difficultyLevel)
2.2: toggleSound()

Muhittin wants to change the settings. He clicks on the settings from the main menu. The system displays the settings screen. Once Muhittin is done playing with the settings, the changes are saved on the GameplayManager object.

4.2.3 View Help

Event Flow:

Muhittin wants to get information about the game before starting to play. He clicks on the “Help” button. The system then displays info on most of the gameplay elements.

4.2.4 View Credits

Event Flow:

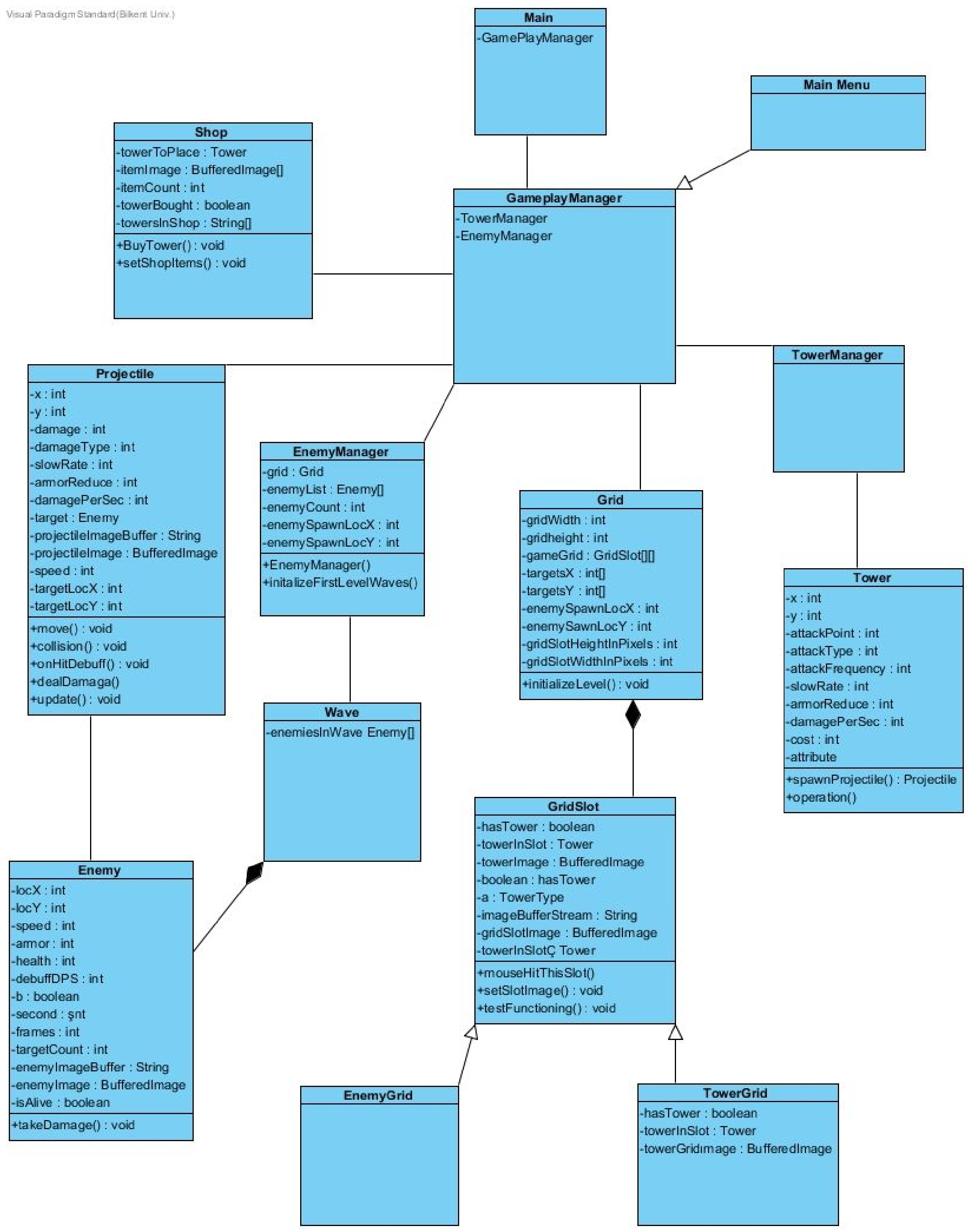
Here Muhittin wants to get information about the developers. The system displays the names and contact info about the developers.

4.2.5 View High Scores

Event Flow:

Here Muhittin wants to see the highest scorers. He clicks on the “High scores” button. The system gets the high scores from the saved location and displays them on screen.

4.3 Class Diagram



5. User Interface

5.1 Navigational Path

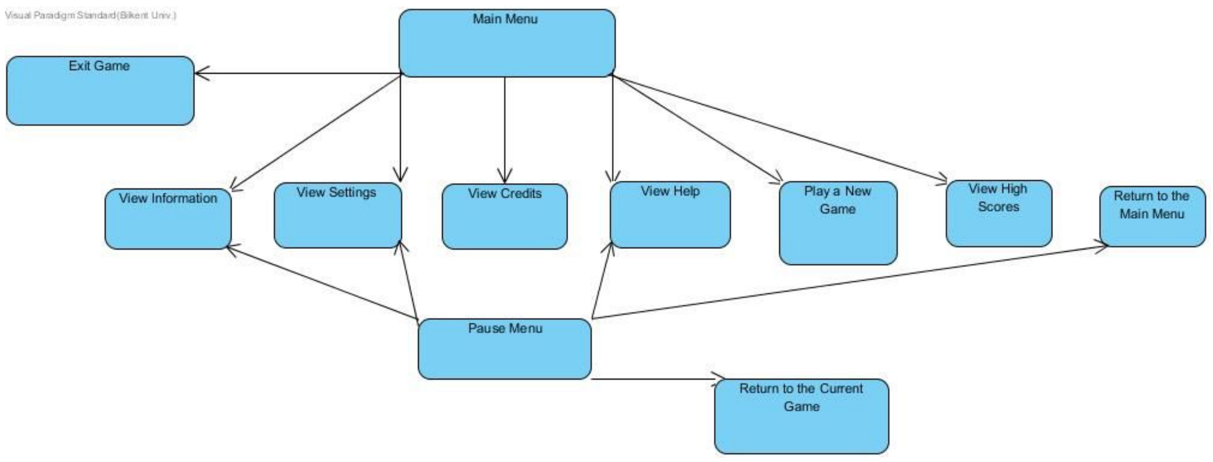


Figure 5.1 Illustrates the Navigational Path of Medieval Tower Defense

5.2 Screenshots

5.2.1 Main Menu

When the game is started, player sees the main menu screen. Main menu contains six options as buttons and an exit icon. Player can choose between playing the game, change settings, get information about the units in the game, get help, see his/her and other player's high scores, see credits and exit the game. (Figure 5.2.1)

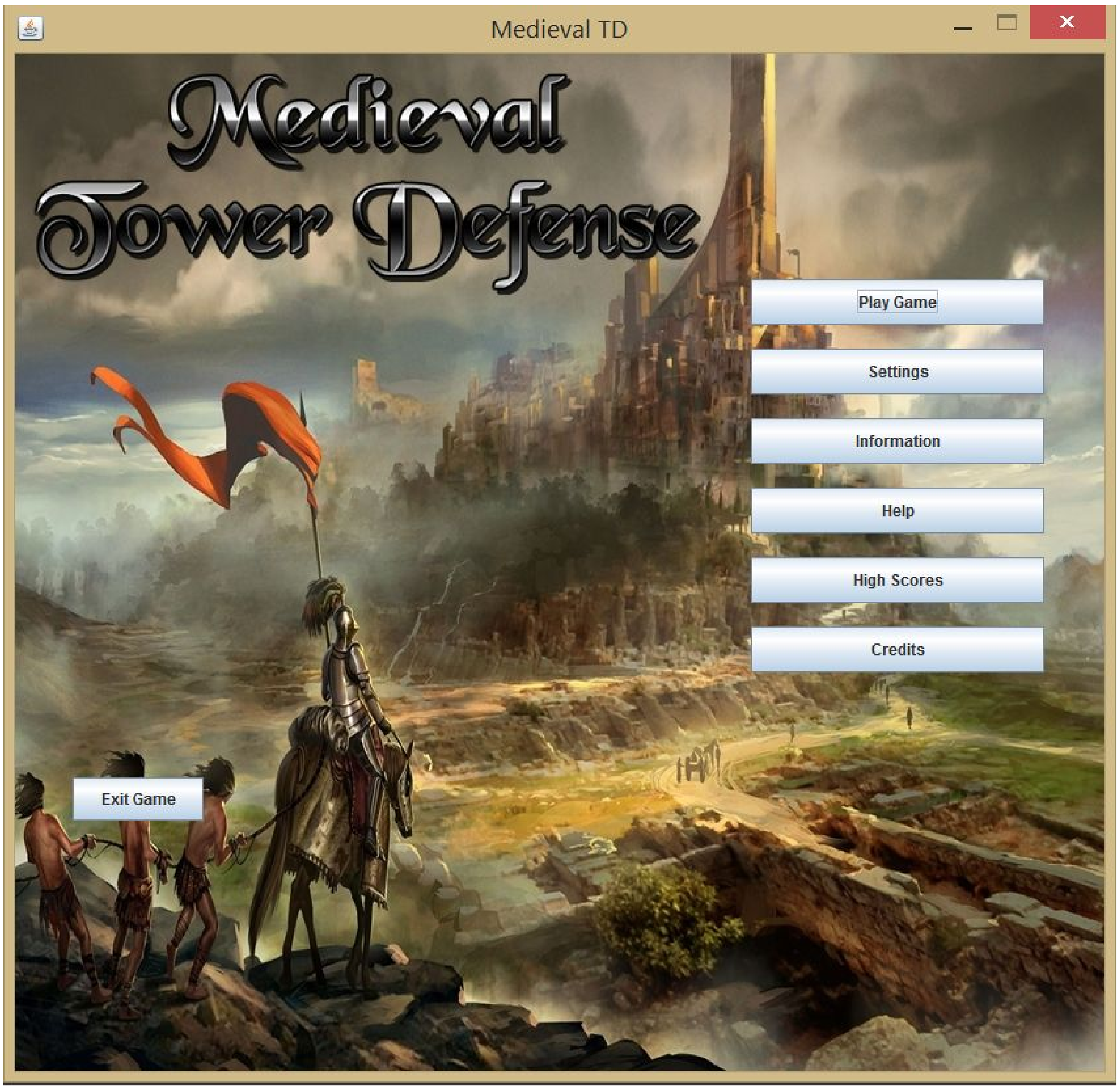


Figure 5.2.1 A screenshot of the main menu

- Play Game:

If the user selects the play game option, game starts with the selected settings. (Figure 5.2.1.1)

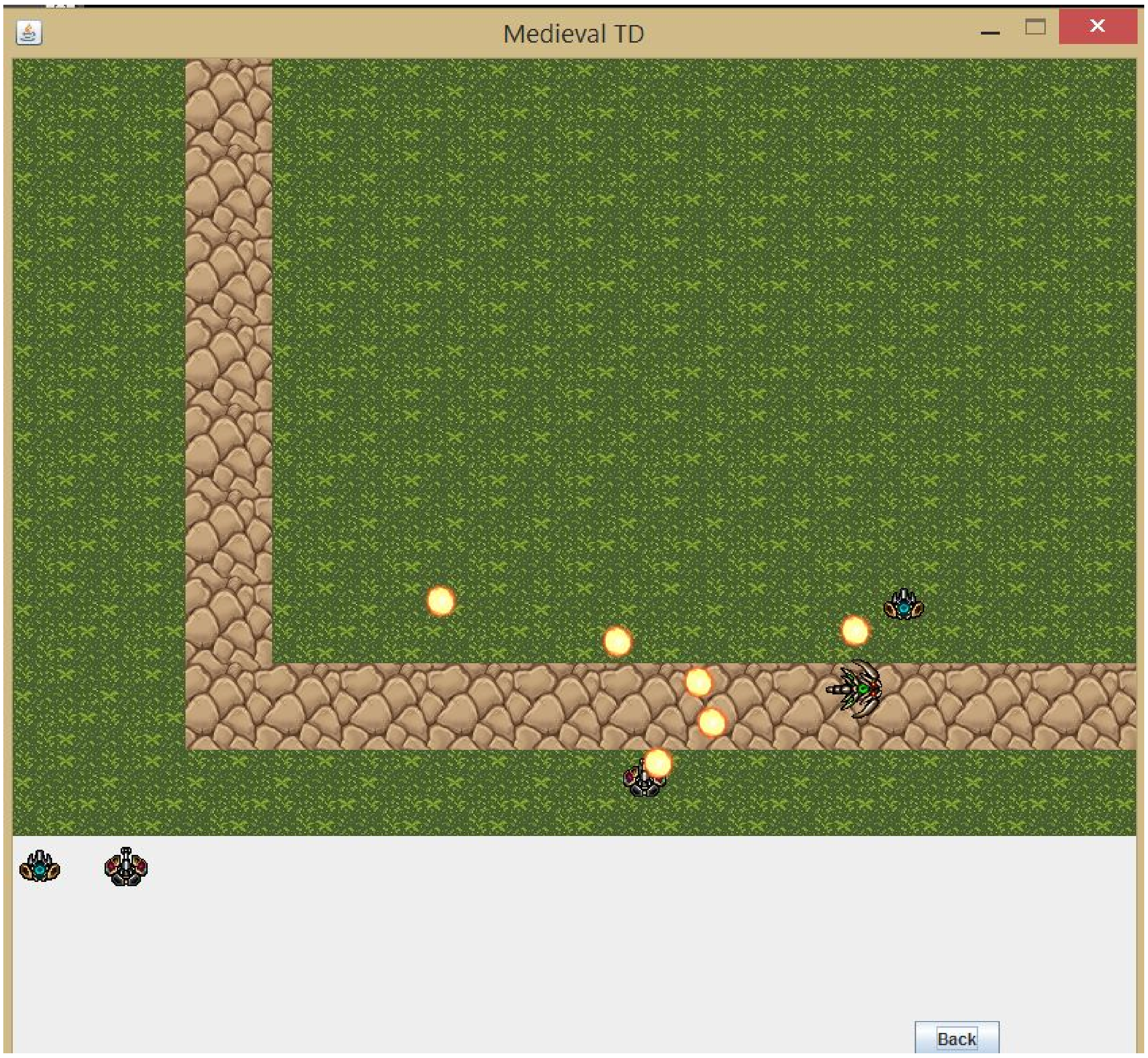


Figure 5.2.1.1: A screenshot from the game

- Settings:

When the player selects the settings button, a screen containing several options appears on the screen. (Figure 5.2.1.2) These settings can be changed to the desired levels by the player. Automatically, default settings are selected. Players can change game difficulty and turn the volume up and down. Players is also able to reset to default settings.

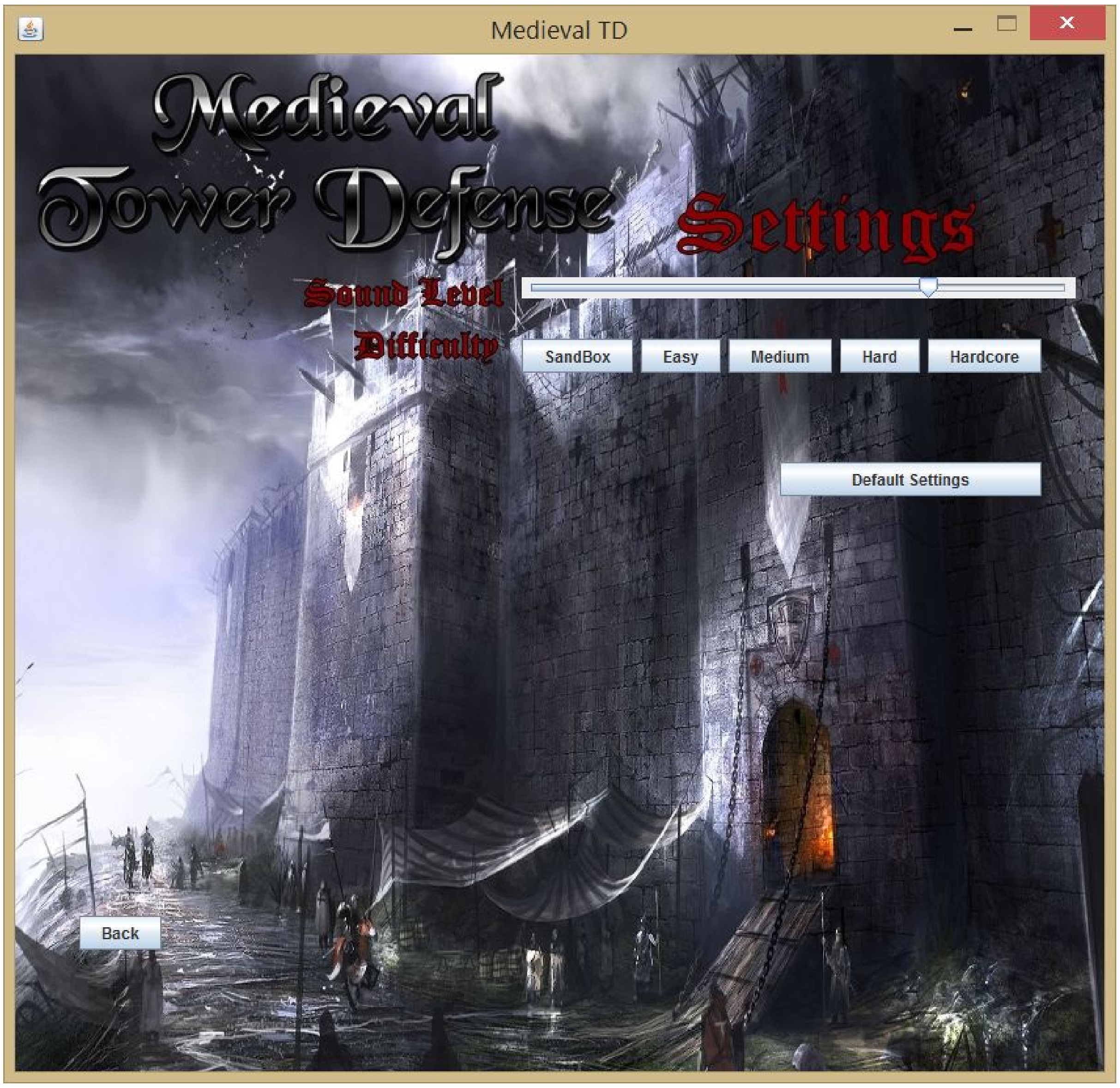
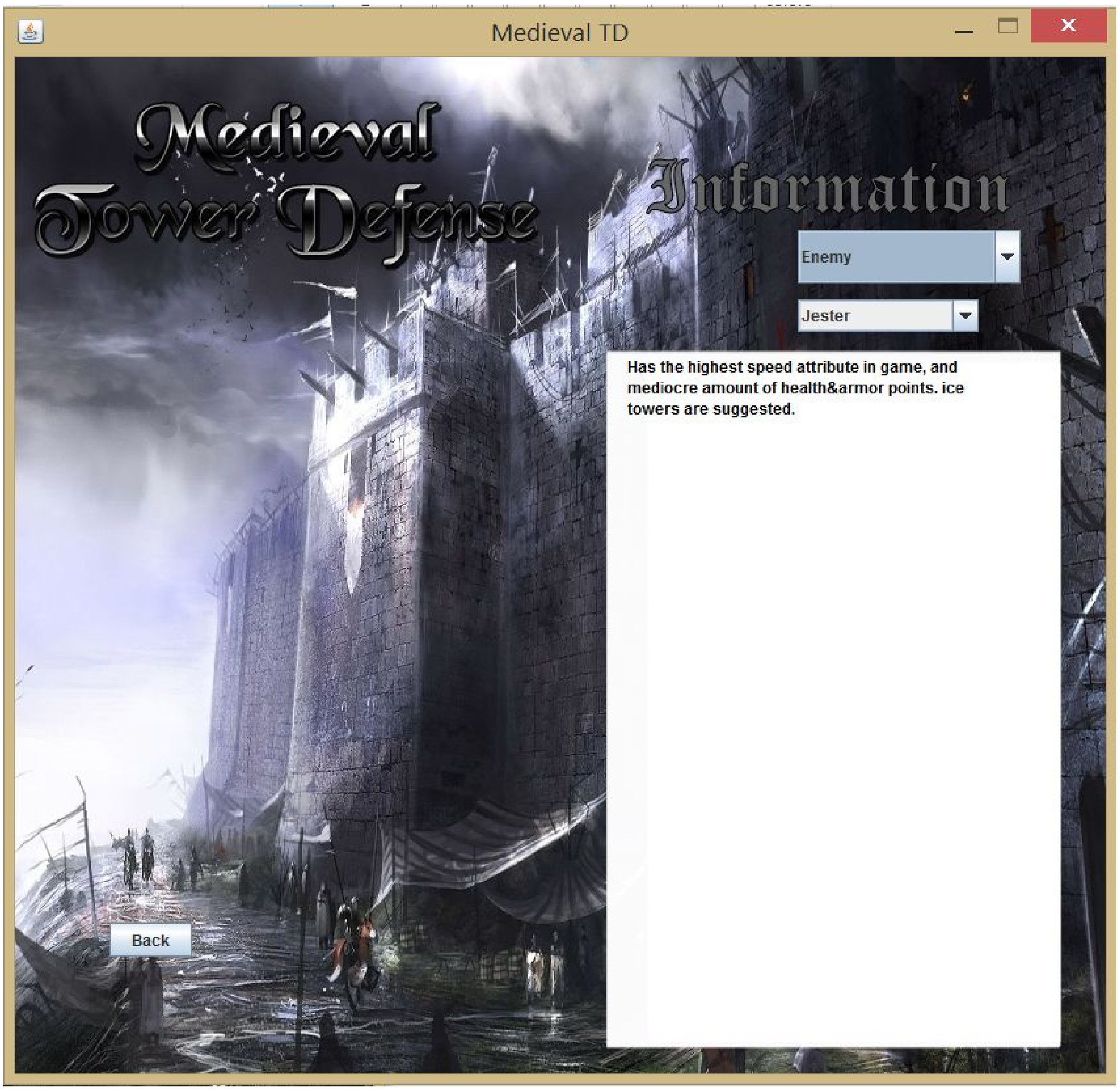


Figure 5.2.1.2: A screenshot of the settings screen

- Information:

When the player selects the information button, a screen containing the tech-tree of the units in the game appears.(Figure 5.2.1.3) The player can read about the attributes of the enemies, towers, waves and gather information about their strategies in this window.

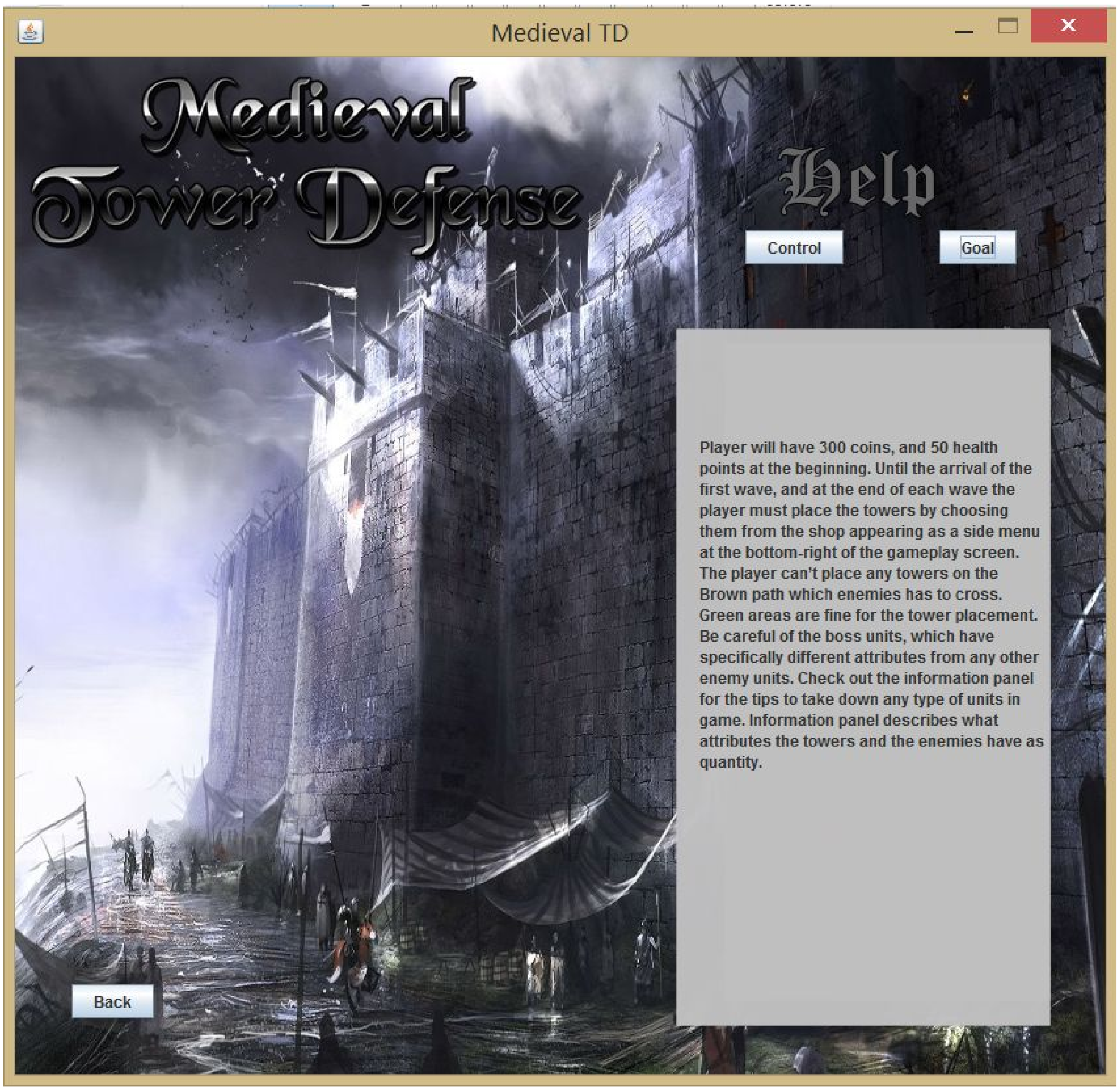


Has the highest speed attribute in game, and mediocre amount of health&armor points. ice towers are suggested.

Figure 5.2.1.3: A screenshot of the information screen

- Help:

When the player selects the help button, a screen containing the controls, basic concepts and aims of the game appears. (Figure 5.2.1.4) The player can read about the controls and how to play the game in this screen.

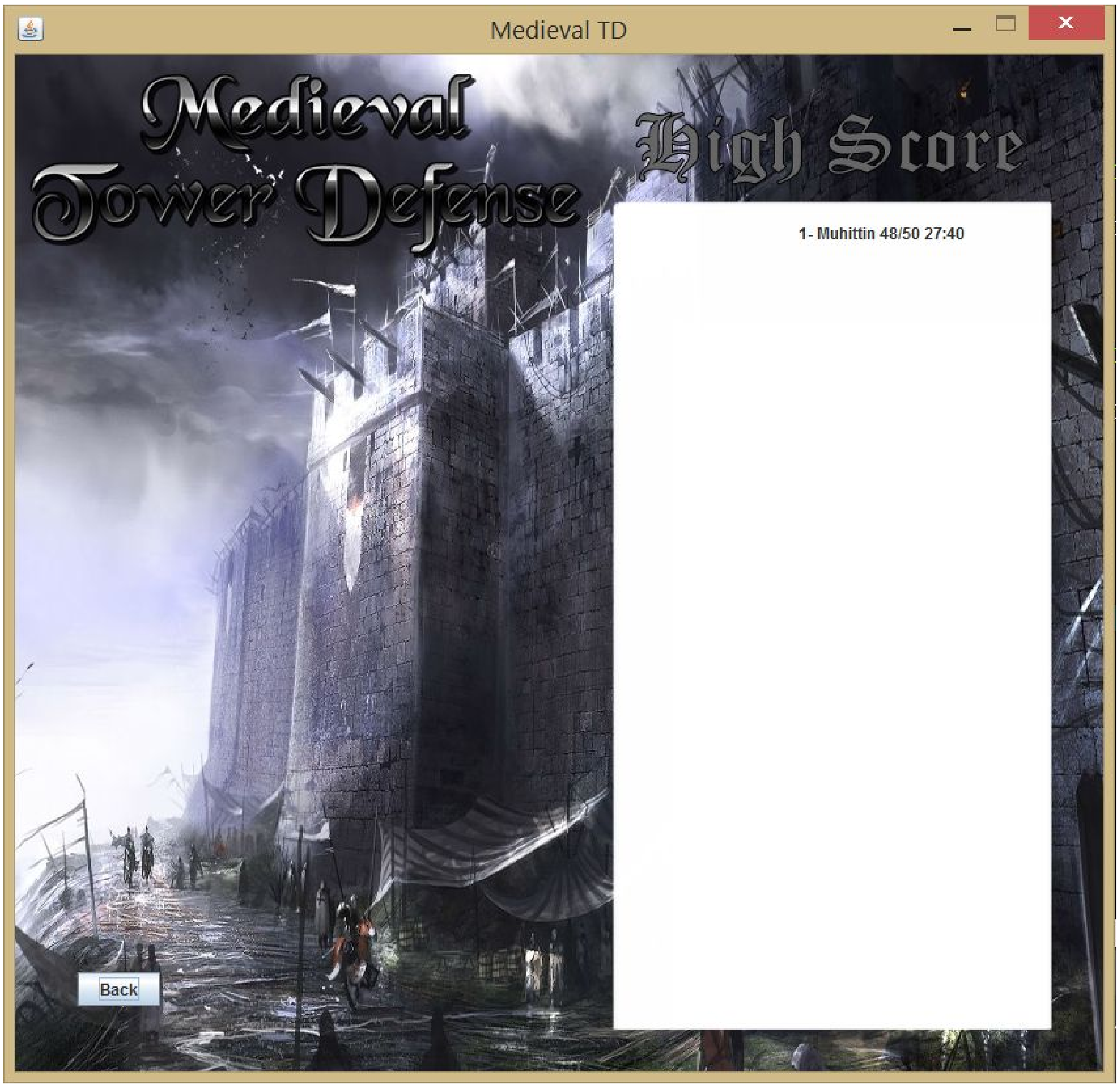


Player will have 300 coins, and 50 health points at the beginning. Until the arrival of the first wave, and at the end of each wave the player must place the towers by choosing them from the shop appearing as a side menu at the bottom-right of the gameplay screen. The player can't place any towers on the Brown path which enemies has to cross. Green areas are fine for the tower placement. Be careful of the boss units, which have specifically different attributes from any other enemy units. Check out the information panel for the tips to take down any type of units in game. Information panel describes what attributes the towers and the enemies have as quantity.

Figure 5.2.1.4: A screenshot of the help screen

- High Scores:

When the player selects the high scores button, a screen that shows the top 10 high scores of the game appears. (Figure 5.2.1.5) These top 10 high scores are arranged according to the wave progress of the players. It priorities the wave progress, but if two players' wave progress is same, then it chooses the one with less time spent.

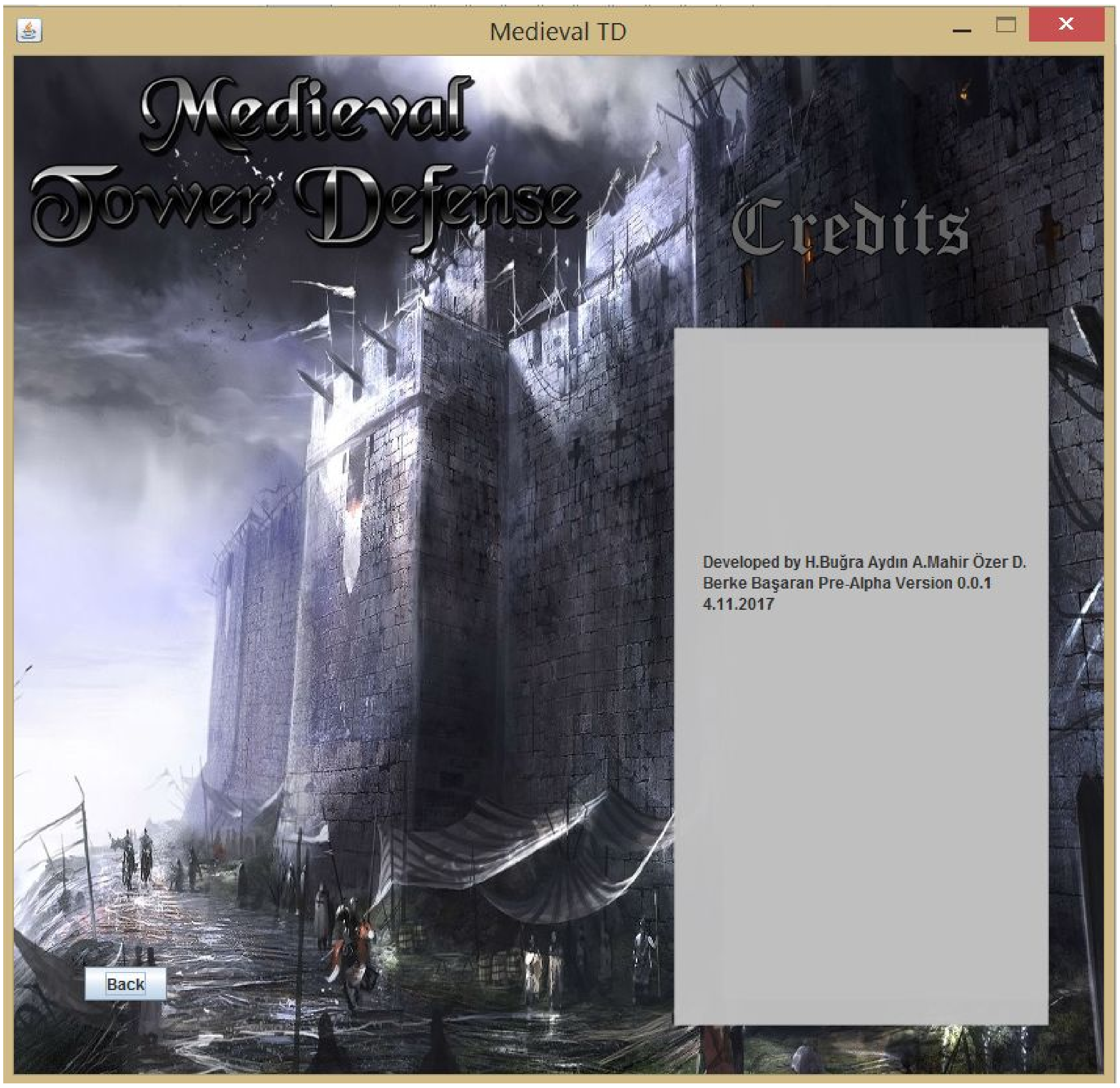


1- Muhittin 48/50 27:40

Figure 5.2.1.5: A screenshot of the highscores screen

- Credits:

When the player selects the credits button, a screen containing the information about the developers appears (Figure 5.2.1.6)



Developed by H.Buğra Aydın A.Mahir Özer D. Berke Başaran Pre-Alpha Version 0.0.1 4.11.2017

Figure 5.2.1.6: A screenshot of the credits screen

6. Design

6.1. Overview

When designing the software architecture, our main goal was to put the methods in their respective classes, aiming to have minimum redirections in between the classes & design the layer interactions neat and no more complex than it needs to be, in order to keep the software straightforward and easily updatable.

6.2 Subsystem Decomposition

In order to organize the system and make the implementation more manageable when working together, we tried to have a lesser number of Manager classes. In terms of performance, this decision doesn't affect the running time of the game.

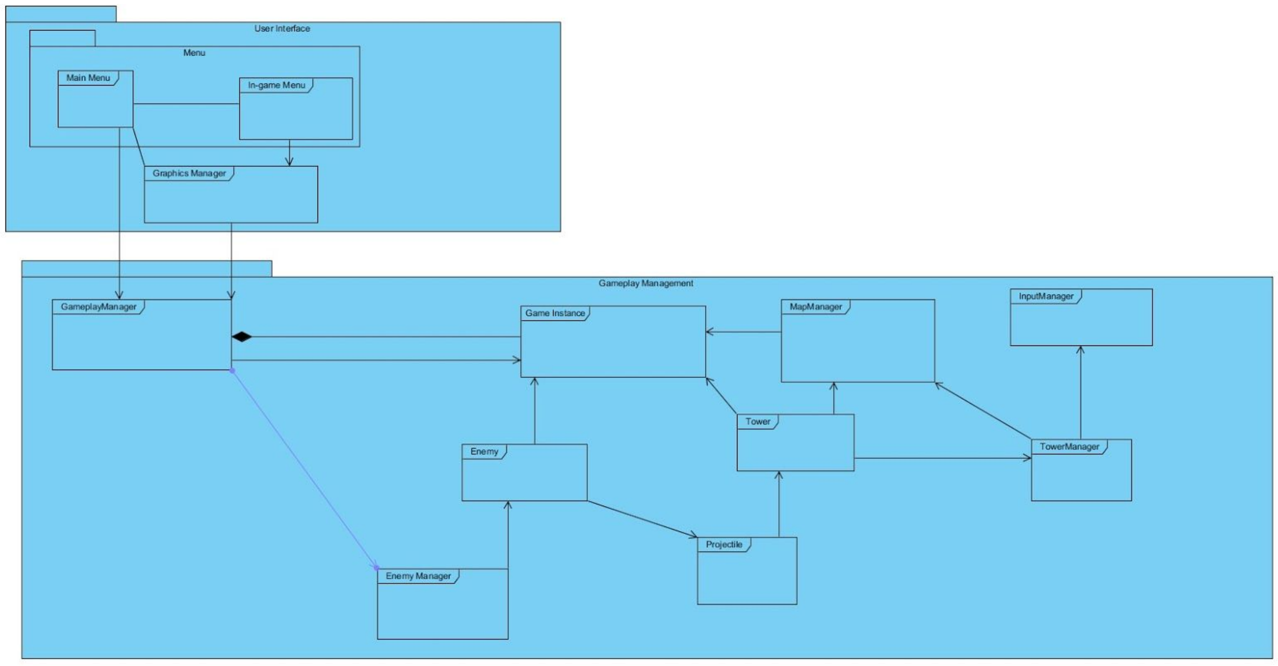


Figure 6.2.1: UI and Gameplay Layer interactions(Basic)

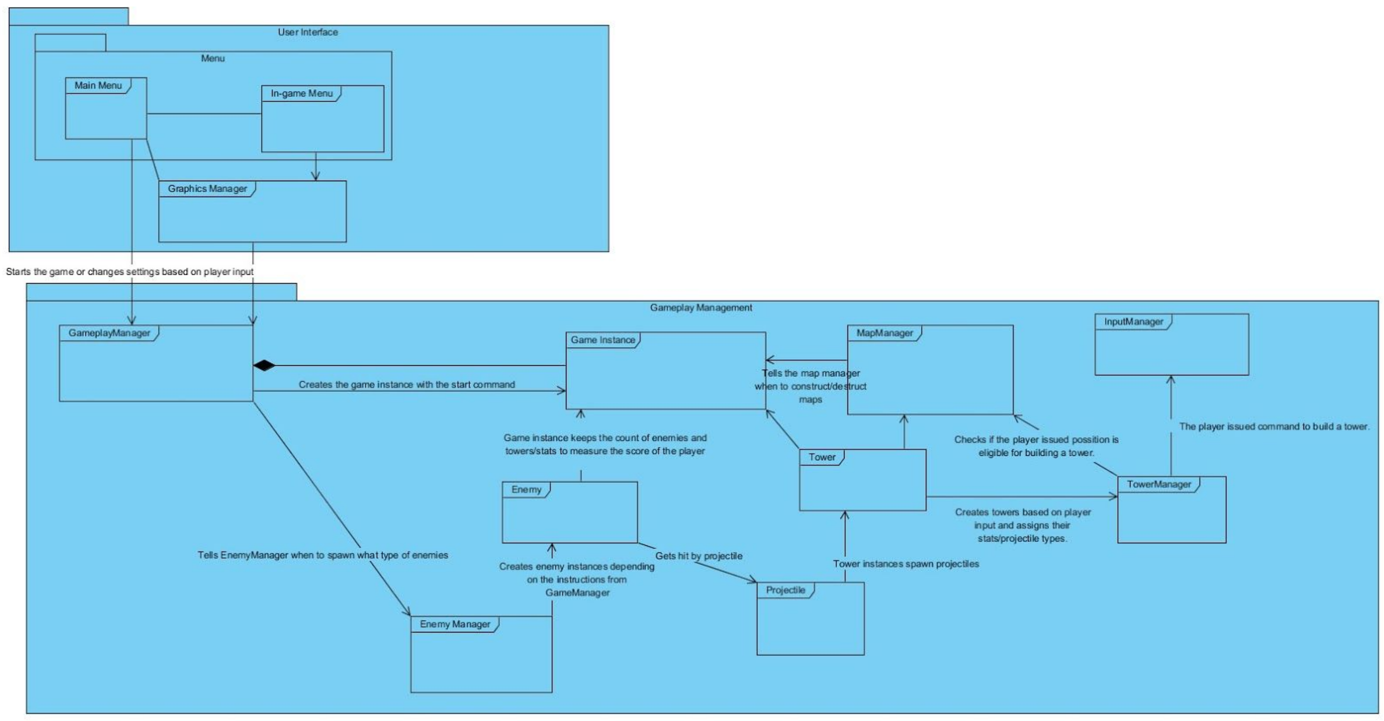

User Interface
Menu
Main Menu
In-game Menu
Graphics Manager
Starts the game or changes settings based on player input
Gameplay Management
GameplayManager
Game Instance
Creates the game instance with the start command
MapManager
Tells the map manager when to construct/destruct maps
InputManager
The player issued command to build a tower.
Game instance keeps the count of enemies and towers/stats to measure the score of the player
Tower
Checks if the player issued possition is eligible for building a tower.
TowerManager
Enemy
Creates towers based on player input and assigns their stats/projectile types.
Tells EnemyManager when to spawn what type of enemies
Creates enemy instances depending on the instructions from GameManager
Gets hit by projectile
Tower instances spawn projectiles
Projectile
Enemy Manager

Figure 6.2.2:UI and Gameplay Layer interactions(Detailed)

6.3 Main Architectural Styles

6.3.1 Layers

In the system design of Medieval Tower Defense, we have composed the system from two different layers, one of which is the User Interface layer, handling functionalities such as settings, starting the game, quitting the game, viewing highscores/help/instructions panels. The other layer handles all of the gameplay, in this layer, the “GameplayManager” holds all the smaller components such as the “TowerManager,” “EnemyManager,” “InputManager,” “SoundManager,” etc. Each of these smaller layers handle relevant game objects, therefore handling all of the objects.

6.3.2 Model View Controller

In this architectural design approach, the user interacts with the system using our controllers, such as listeners. Controllers manipulates the model of the game by making corresponding changes. Then, the “model” updates the view of the application, which can be gameplay screen, settings screen, information screen, help screen, high score screen, and credits screen. User sees this updates and the cycle continues.

The underlying objects of our classes are accessed and controlled by our manager classes such as GameplayManager(controller). GUIManager class makes the desired changes to the view (view) and classes such as Tower and Invader corresponds to the model part of this structure.

6.4 Hardware / Software Mapping

During the implementation of this project, latest version of JDK will be used. As hardware configuration, a mouse and optionally a keyboard is required. The game is really small in sizes and requires really low CPU power, so it will be possible to run it on almost any PC with java installed. Java's portability will be useful in this stage since it will help us to code it only once and it will be converted to corresponding forms in java virtual machine.

6.5 Persistent Data Management

Medieval Tower Defense will store the high scores in a .text file. When the player clicks on the HighScores tab, the game will read from the text file using Java's File I/O libraries. New high scores are written on the .txt file when a player finishes a game with a high score and enters their name.

6.6 Access Control and Security

In order to access the game, downloading the .jar file is enough. There won't be any security checkings going on with the system. But the game requires no personal information except a nickname, therefore a security issue is not the case.

6.7 Boundary Conditions

Initialization

Medieval Tower Defense does not require any installation. Game will be in the .jar format.

Termination

The user will be able to terminate the game by clicking the 'back' icon in the main menu. During the game, pausing the game will provide this button also.

Error

If an error occurs during the gameplay, the data will be lost. If an error occurs while loading images or sounds, game will not be able to start.

7. Subsystem Services

This section provides the detailed information about the information of subsystems.

7.1 Design Patterns

Façade Design Pattern:

Façade design involves a single class that provides easy to use and understand methods required by the user, and abstracts the underlying system classes and their methods. Façade class can be considered as an interface for the client in order to acces the system. It is useful because it hides the underlying complexities and makes it easy to reflect any changes in the system.

In this design, façade pattern was used in order to connect different classes in a single manager class. For example, our GUIManager class is a Façade class that communicates with other subsystems and abstracts other UI classes.

7.2 Graphical User Interface Subsystem

UI subsystem makes the users able to iterate through the contents of the Medieval Tower Defense game and providing them graphical contents. It also manages the sections of the main menu. It also alerts the game manager object according to the choices of the user (such as settings).

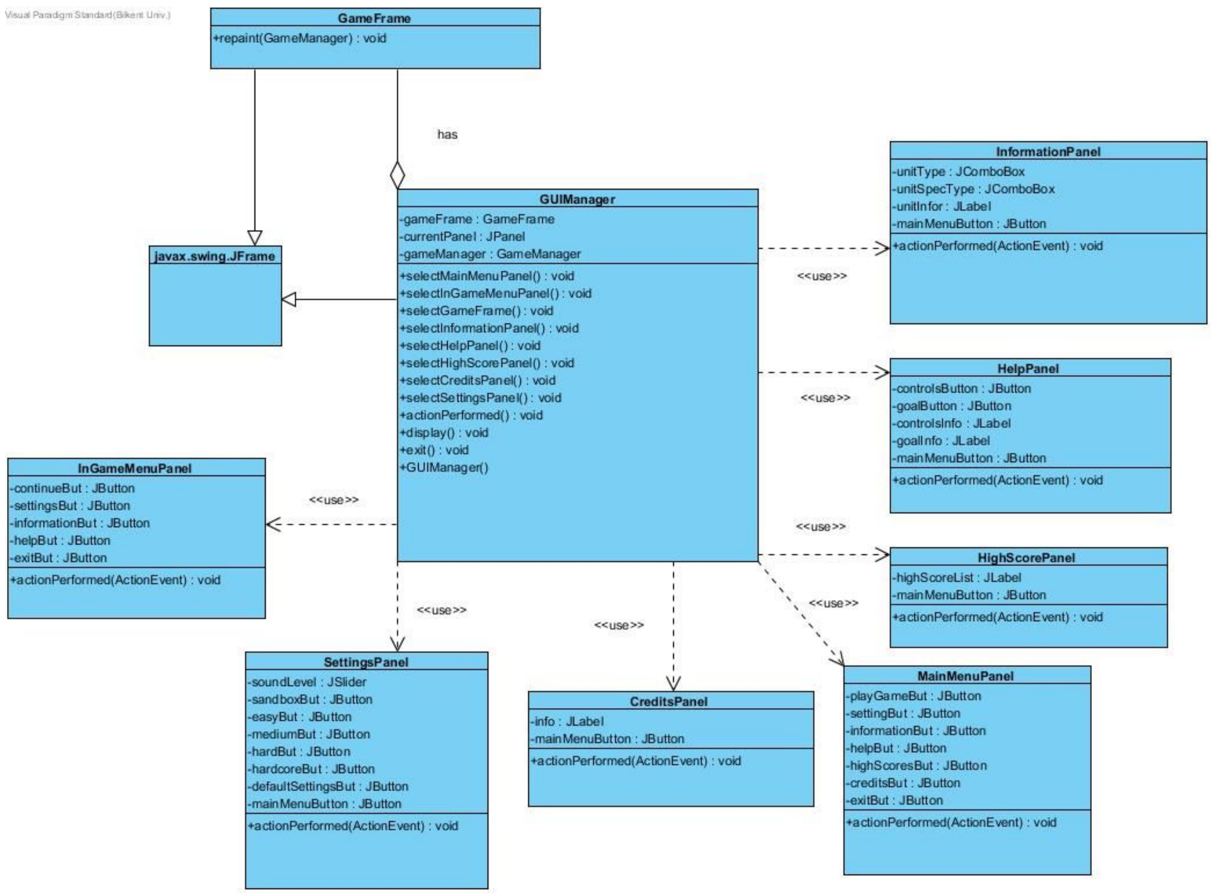

Visual Paradigm Standard(Bilkent Univ.)
GameFrame
+repaint(GameManager) : void
has
javax.swing.JFrame
GUIManager
-gameFrame : GameFrame
-currentPanel : JPanel
-gameManager : GameManager
+selectMainMenuPanel() : void
+selectInGameMenuPanel() : void
+selectGameFrame() : void
+selectInformationPanel() : void
+selectHelpPanel() : void
+selectHighScorePanel() : void
+selectCreditsPanel() : void
+selectSettingsPanel() : void
+actionPerformed() : void
+display() : void
+exit() : void
+GUIManager()
InformationPanel
-unitType : JComboBox
-unitSpecType : JComboBox
-unitInfor : JLabel
-mainMenuButton : JButton
+actionPerformed(ActionEvent) : void
<<use>>
HelpPanel
-controlsButton : JButton
-goalButton : JButton
-controlsInfo : JLabel
-goalInfo : JLabel
-mainMenuButton : JButton
+actionPerformed(ActionEvent) : void
<<use>>
InGameMenuPanel
-continueBut : JButton
-settingsBut : JButton
-informationBut : JButton
-helpBut : JButton
-exitBut : JButton
+actionPerformed(ActionEvent) : void
<<use>>
<<use>>
HighScorePanel
-highScoreList : JLabel
-mainMenuButton : JButton
+actionPerformed(ActionEvent) : void
<<use>>
<<use>>
<<use>>
SettingsPanel
-soundLevel : JSlider
-sandboxBut : JButton
-easyBut : JButton
-mediumBut : JButton
-hardBut : JButton
-hardcoreBut : JButton
-defaultSettingsBut : JButton
-mainMenuButton : JButton
+actionPerformed(ActionEvent) : void
CreditsPanel
-info : JLabel
-mainMenuButton : JButton
+actionPerformed(ActionEvent) : void
MainMenuPanel
-playGameBut : JButton
-settingBut : JButton
-informationBut : JButton
-helpBut : JButton
-highScoresBut : JButton
-creditsBut : JButton
-exitBut : JButton
+actionPerformed(ActionEvent) : void

Figure 3.2.1 (User Interface Subsystem)

GUIManager Class

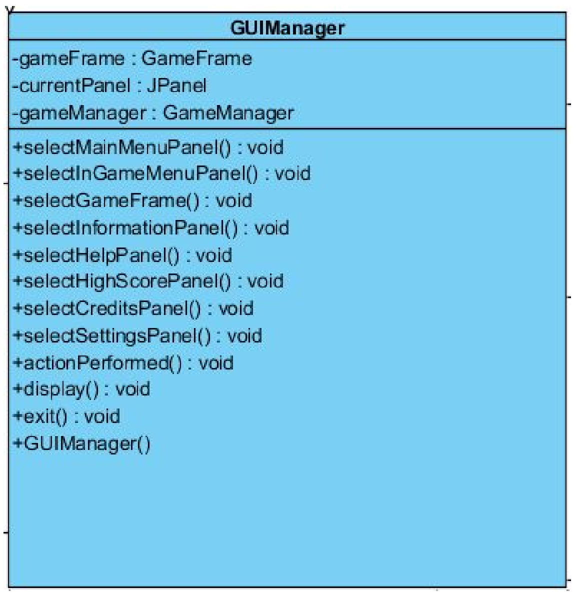

GUIManager
-gameFrame : GameFrame
-currentPanel : JPanel
-gameManager : GameManager
+selectMainMenuPanel() : void
+selectInGameMenuPanel() : void
+selectGameFrame() : void
+selectInformationPanel() : void
+selectHelpPanel() : void
+selectHighScorePanel() : void
+selectCreditsPanel() : void
+selectSettingsPanel() : void
+actionPerformed() : void
+display() : void
+exit() : void
+GUIManager()

Figure 3.2.2 (GUIManager class)

Attributes:

private GameFrame gameFrame: This is the frame where the game is displayed

private JPanel currentPanel: This panel is the panel which the current selection is shown

private GameManager gameManager: This is the object which connects the game and the gui features. GUI gets updated from this object.

Constructors:

public GUIManager: Initilizes panels such as help panel, information panel and others. Also initilizes settings.

Methods:

public void selectMainMenuPanel(): This method changes the current view to the main menu screen.

public void selectInGameMenuPanel(): This method changes the current view to the in game menu screen.

public void selectGameFrame(): This method changes the current view to the game screen.

public void selectInformationPanel(): This method changes the current view to the information screen.

public void selectHelpPanel(): This method changes the current view to the help screen.

public void selectHighScorePanel(): This method changes the current view to the high score screen.

public void selectCreditsPanel(): This method changes the current view to the credits screen.

public void selectSettingsPanel(): This method changes the current view to the settings screen.

public void actionPerformed(): This method makes it possible to communicate with the user, interpreting his/her actions.

public void display(): This method is used by select methods above to display the current view.

public void exit(): This method terminates the game when called.

MainMenuPanel Class

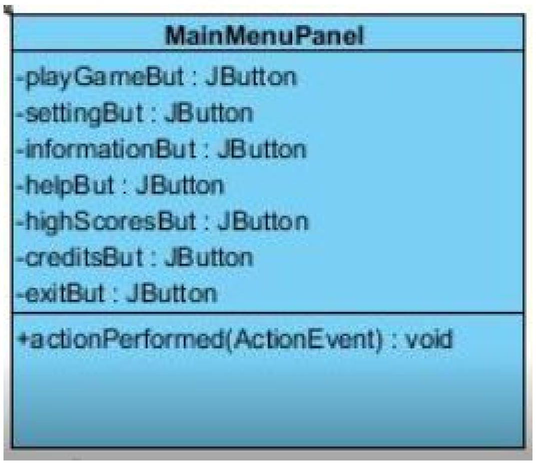


Figure 3.2.3 (MainMenuPanel Class)

Attributes:

private JButton playGameBut: This attribute provides a button to the screen. When this button is selected by the user, game starts by selecting the game frame.

private JButton settingBut: This attribute provides a button to the screen. When this button is selected by the user, settings screen appears by selecting the settings panel.

private JButton helpBut: This attribute provides a button to the screen. When this button is selected by the user, help screen appears by selecting the help panel.

private JButton informationBut: This attribute provides a button to the screen. When this button is selected by the user, information screen appears by selecting the information panel.

private JButton helpBut: This attribute provides a button to the screen. When this button is selected by the user, help screen appears by selecting the help panel.

private JButton highScoresBut: This attribute provides a button to the screen. When this button is selected by the user, high scores screen appears by selecting the high scores panel.

private JButton creditsBut: This attribute provides a button to the screen. When this button is selected by the user, credits screen appears by selecting the credits panel.

private JButton exitBut: This attribute provides a button to the screen. When this button is selected by the user, exit() method of the GUIManager is called and the game ends.

Methods:

public void actionPerformed(ActionEvent): This method gets alerted when user selects a button and makes acts in the program corresponding to the selection.

GameFrame Class

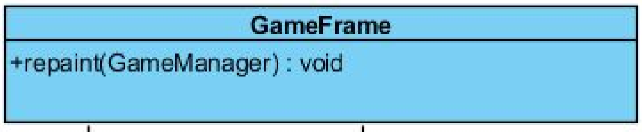


Figure 3.2.4 (GameFrame Class)

Methods:

public void repaint(GameManager): If the selectGameFrame method is called on the GUIManager class, GameFrame keeps painting the game objects on their corresponding positions. This class uses the gameManager object from the GUIFrame, taking arguments from it.

InGameMenuPanel Class

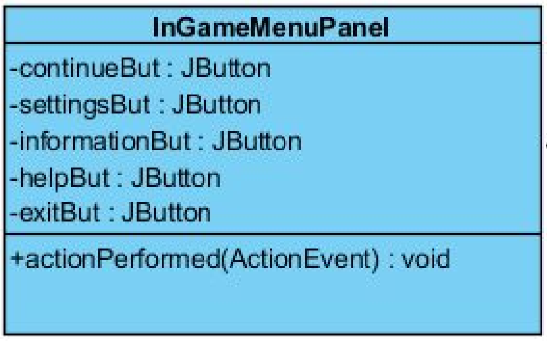


Figure 3.2.5 (InGameMenuPanel Class)

Attributes:

private JButton continueBut: This attribute provides a button to the screen. When this button is selected by the user, game frame is selected and game continues.

private JButton settingsBut: This attribute provides a button to the screen. When this button is selected by the user, settings panel is selected and settings screen is provided to the user.

private JButton informationBut: This attribute provides a button to the screen. When this button is selected by the user, information panel is selected and information screen is provided to the user.

private JButton helpBut: This attribute provides a button to the screen. When this button is selected by the user, help panel is selected and help screen is provided to the user.

private JButton exitBut: This attribute provides a button to the screen. When this button is selected by the user, exit() method of the GUIManager is called and the program terminates.

Methods:

public void actionPerformed(ActionEvent): This method gets alerted when user selects a button and makes acts in the program corresponding to the selection.

SettingsPanel Class

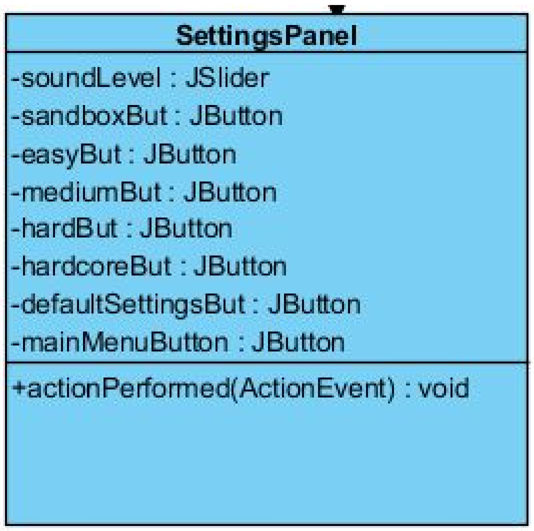


Figure 3.2.6 (SettingsPanel Class)

Attributes:

private JSlider soundLevel: This attribute provides a JSlider to the settings panel. By using it, user will be able to change the sound level to his/her desires.

private JButton sandboxBut: This attribute provides a button to the settings panel. When selected, calls the corresponding method from the gameplaymanager and sets the difficulty level to sandbox.

private JButton easyBut: This attribute provides a button to the settings panel. When selected, calls the corresponding method from the gameplaymanager and sets the difficulty level to easy.

private JButton mediumboxBut: This attribute provides a button to the settings panel. When selected, calls the corresponding method from the gameplaymanager and sets the difficulty level to medium.

private JButton hardBut: This attribute provides a button to the settings panel. When selected, calls the corresponding method from the gameplaymanager and sets the difficulty level to hard.

private JButton hardcoreBut: This attribute provides a button to the settings panel. When selected, calls the corresponding method from the gameplaymanager and sets the difficulty level to hardcore.

private JButton defaultSettingsBut: This attribute provides a button to the settings panel. When selected, calls the corresponding method from the gameplaymanager and sets the current settings back to default.

private JButton mainMenuButton: This attribute provides a button to the settings panel. When selected, calls the selectMainMenu method of the GUIManager and changes the view.

Methods:

public void actionPerformed(ActionEvent): This method gets alerted when user selects a button and makes acts in the program corresponding to the selection.

CreditsPanel Class

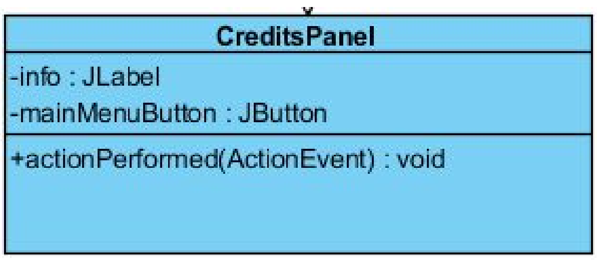

CreditsPanel
-info : JLabel
-mainMenuButton : JButton
+actionPerformed(ActionEvent) : void

Figure 3.2.7 (CreditsPanel Class)

Attributes:

private JLabel info: This attribute provides a text label to the credits panel. It contains the credits as text format.

private JButton mainMenuButton: This attribute provides a button to the credits panel. When selected, calls the selectMainMenu method of the GUIManager and changes the view.

Methods:

public void actionPerformed(ActionEvent): This method gets alerted when user selects a button and makes acts in the program corresponding to the selection.

HighScorePanel Class

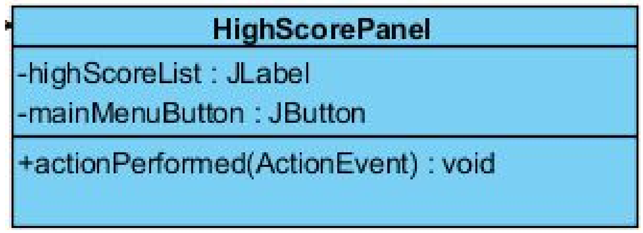

HighScorePanel
-highScoreList : JLabel
-mainMenuButton : JButton
+actionPerformed(ActionEvent) : void

Figure 3.2.8 (HighScorePanel Class)

Attributes:

private JLabel highScoreList: This attribute provides a text label to the highScorePanel. It contains the high score information.

private JButton mainMenuButton: This attribute provides a button to the high score panel. When selected, calls the selectMainMenu method of the GUIManager and changes the view.

Methods:

public void actionPerformed(ActionEvent): This method gets alerted when user selects a button and makes acts in the program corresponding to the selection.

HelpPanel Class

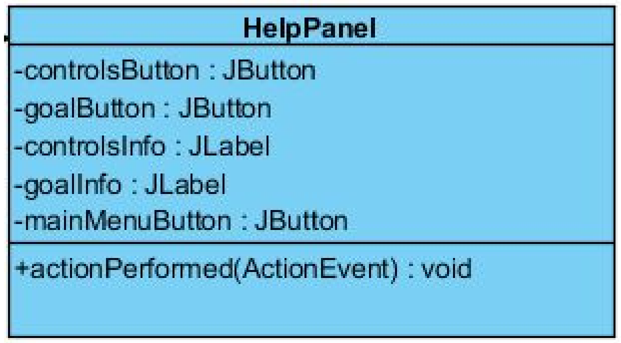

HelpPanel
-controlsButton : JButton
-goalButton : JButton
-controlsInfo : JLabel
-goalInfo : JLabel
-mainMenuButton : JButton
+actionPerformed(ActionEvent) : void

Figure 3.2.9 (HelpPanel Class)

Attributes:

private JButton controlsButton: This attribute provides a button to the help panel. When selected, changes the information displayed on the screen from goal info to controls info.

private JButton goalButton: This attribute provides a button to the help panel. When selected, changes the information displayed on the screen from controls info to goal info.

private JLabel controlsInfo: This attribute provides a JLabel to the help panel. It contains the corresponding controls information about the game.

private JLabel goalInfo: This attribute provides a JLabel to the help panel. It contains the corresponding information about the aims of the game.

private JButton mainMenuButton: This attribute provides a button to the help panel. When selected, calls the selectMainMenu method of the GUIManager and changes the view.

Methods:

public void actionPerformed(ActionEvent): This method gets alerted when user selects a button and makes acts in the program corresponding to the selection.

InformationPanel Class

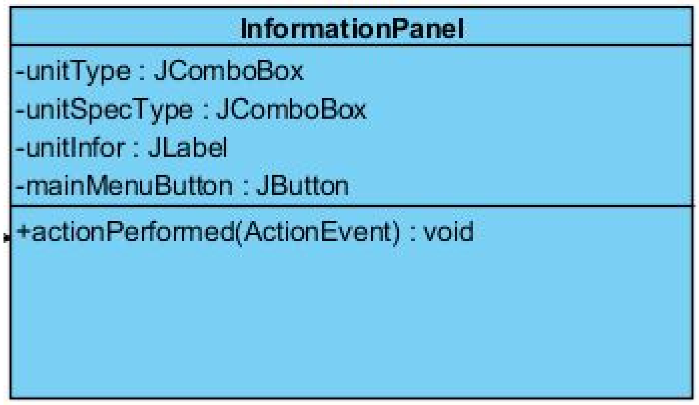

InformationPanel
-unitType : JComboBox
-unitSpecType : JComboBox
-unitInfor : JLabel
-mainMenuButton : JButton
+actionPerformed(ActionEvent) : void

Figure 3.2.10 (InformationPanel Class)

Attributes:

private JComboBox unitType: This attribute provides a combo box to the information panel. User can select the unit type from this combo box (such as enemies, bosses, towers). This selection changes the corresponding options in the unitSpecType combo box.

private JComboBox unitSpecType: This attribute provides a combo box to the information panel. User can select the units' specific type from this combo box (for example for the enemy selection from unitType, the options avaible in the unitSpecType combo box will be Chengiz Khan, Knight, Trojan Horse etc) This selection changes the corresponding information in the unitInfor label.

private JLabel unitInfor: This attribute provides a text label to the information panel. According to the user's selections, corresponding information about the specific unit appears here.

private JButton mainMenuButton: This attribute provides a button to the information panel. When selected, calls the selectMainMenu method of the GUIManager and changes the view.

Methods:

public void actionPerformed(ActionEvent): This method gets alerted when user selects a button and makes acts in the program corresponding to the selection.

7.3 Gameplay Subsystem Interface

In this subsystem, we have the facade class "Map" which handles the communication between the "Grid", the "Player", and all instances of "Enemy" and "Tower" classes. Since this class has the reference to the "Grid" as an attribute, it also has access to all of the "gridSlot"s, and therefore all of the objects on those "gridSlot"s, which makes it easier for us to manipulate objects without getting redirected too much in between classes/interfaces.

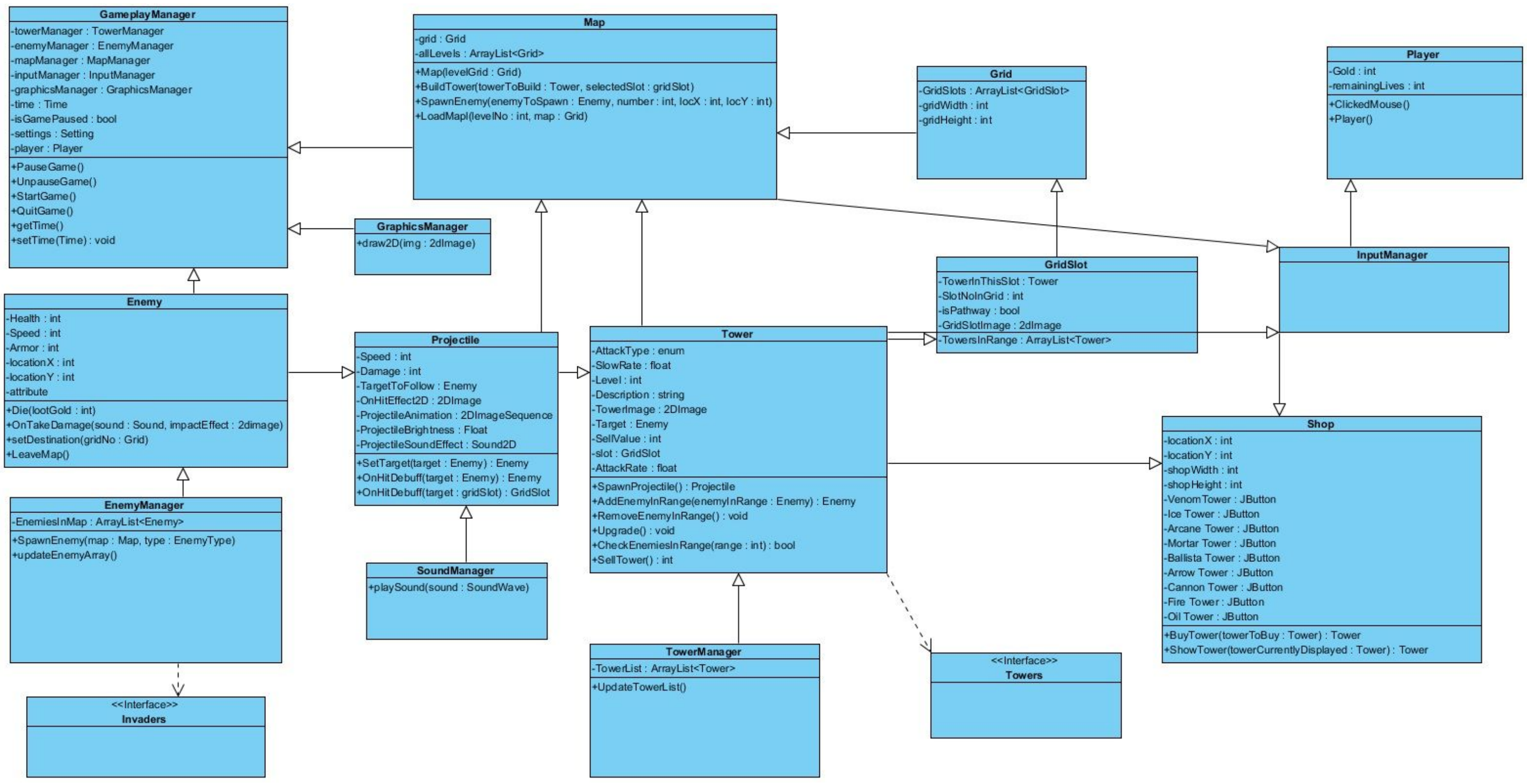


Figure 7.3.1 Gameplay Objects Subsystem

GameplayManager Class

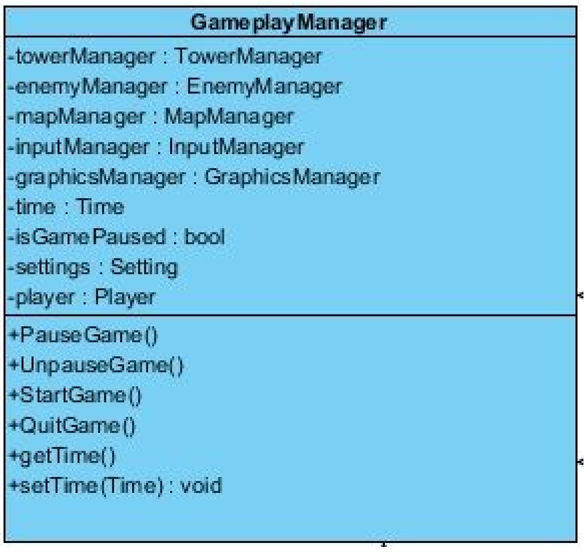


Figure 7.3.2 (GameplayManager Class)

Attribute:

private TowerManager towerManager: A reference to the TowerManager that holds all the tower instances in an array.

private EnemyManager enemyManager: A reference to the EnemyManager that holds all the enemy instances in an array.

private MapManager mapManager: A reference to the MapManager that holds a reference to the map.

private InputManager inputManager: Listens for mouse click events from the player.

private GraphicsManager graphicsManager: Draws all the 2D images and animations of enemies, and towers. Also visualizes the movement of enemies.

public Time time: This value keeps a track of time in order to calculate the score of the player at the end of the game.

public bool isGamePaused: Whether the game is paused.

private Setting settings: A reference to the class where the settings ‘sound volume, difficulty(sandbox, medium, easy, hard, hardcore)’ are kept.

Methods:

public PauseGame(): Game may be paused anytime by the player if Pause Break button is pushed. When paused, player will be directed to in-game menu.

public UnpauseGame(): If player is in in-game menu, or if the game is paused, player may unpause the game by hitting the button Pause Break once again.

public StartGame(): New game sequence will be initiated once StartGame button is hit.

public QuitGame(): Current game progress may be abondened by hitting the QuitGame button, which is available at in-game menu

public getTime(): Calculated time can be retrieved with getTime method.

public void setTime(): Time is calculated with this method. It measures the player's speed of destroying the enemy wave. Quicker the player destroys the wave, better the score will be.

Map Class

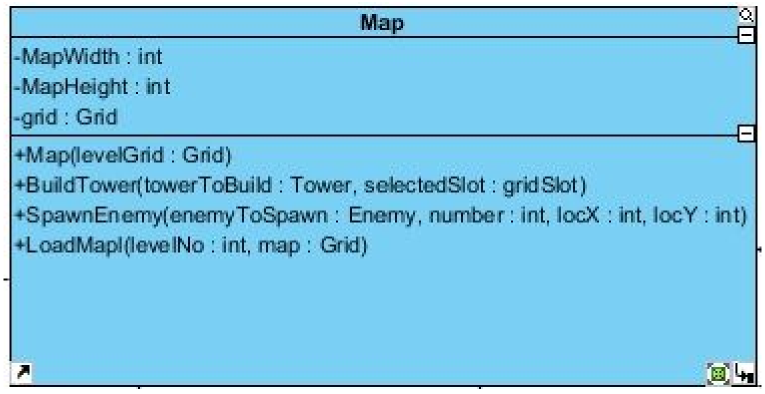

Map
-MapWidth : int
-MapHeight : int
-grid : Grid
+Map(levelGrid : Grid)
+BuildTower(towerToBuild : Tower, selectedSlot : gridSlot)
+SpawnEnemy(enemyToSpawn : Enemy, number : int, locX : int, locY : int)
+LoadMapl(levelNo : int, map : Grid)

Figure 7.3.3 (GameplayManager Class)

Constructors:

public Map(levelGrid : Grid): Constructs a game map with the given "Grid" object. Since the graphics are just seamless images of each grid slot piled up together, no further work is needed on this class for the map construction, that is handled on the "Grid" object.

Methods:

public Tower BuildTower(towerToBuild : Tower, selectedSlot : GridSlot): When the player selects a tower from the shop and clicks on the game map, this function is called to build a tower on the selected "gridSlot," if the selected gridSlot is available. The availability of the gridSlot is checked by first looking at "isPathway" boolean value of the gridSlot object. If the gridSlot is not a pathway, and if it has no tower object on it, then the tower is built, otherwise the player click is dismissed.

private Map LoadMap(levelNo : int): Can load a map by the level number.

public Enemy SpawnEnemy(enemyToSpawn : Enemy, number : int, locX : int, locY : int): Spawns an enemy

GraphicsManager Class

Attributes:

draw2D(img : 2dImage): This method draws the image that is passed to it by parameter. Since GraphicsManager has access to the GameManager, it can draw the needed objects. For instance, all the towers on the map are stored in the TowerManager, and the GameManager has access to the TowerManager, therefore the GraphicsManager can also access all the individual towers and draw them.

Grid Class

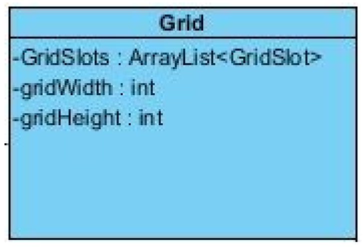


Figure 7.3.4 (Grid Class)

Attributes:

GridSlots: All the slots in the grid stored in the Grid Class. There is only one “Grid” class instance for each level/map.

gridWidth: Stores the width of the grid.

gridHeight: Stores the height of the grid.

GridSlot Class

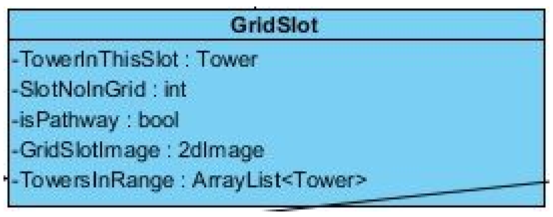


Figure 7.3.5 (GridSlot Class)

Attributes

private Tower TowerInThisSlot: Stores the tower in this slot, if this attribute is NULL and "isPathway" is "FALSE," then the player can build a tower on this "gridSlot."

private int SlotNoInGrid: The slot number that this gridSlot corresponds to in the whole Grid.

private bool isPathway: If this boolean is true, than this gridSlot is where the enemies can move to, and the player cannot build a tower on this slot. If this boolean is false, the player can build a tower on this slot and the enemies cannot move through this gridSlot.

private 2dImage GridSlotImage: The 2d image to draw for a particular gridSlot instance. These images are seamless so that the same type of floors in the game work together.

private ArrayList<Tower> TowersInRange: This attribute holds the towers that can attack the enemies in this gridSlot.

Tower Class

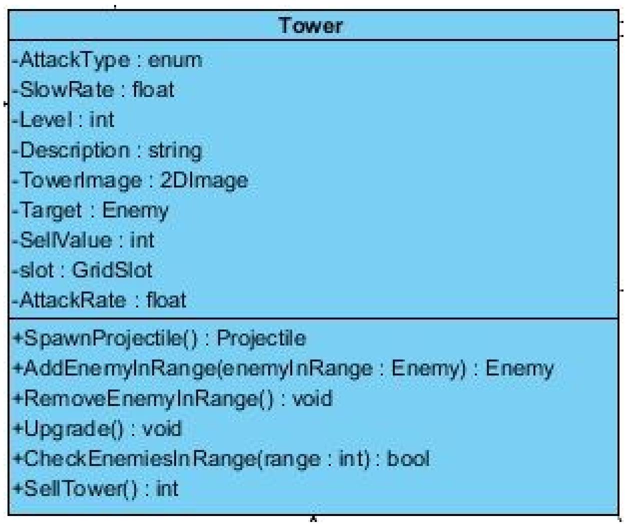


Figure 7.3.6 (Tower Class)

(Moved Function) CheckEnemiesInRange(range : int) was a function of the "Tower" class, but we decided to move it to the gridSlot class for performance reasons, here's a short explanation:
"CheckEnemiesInRange" is a function that is called every frame for each tower, which can result in framerate drops because of the CPU time when there are many towers. So the CPU time is the downside of placing this function on this class. The good side is to this method is that towers can take float values as range and the game would benefit a bit more variety.

Our decision was to define the tower ranges as integers which correspond to gridSlots. So that a tower can hit the 8 “gridSlots” around it, or 16, 24 and so on… For this way of handling the targeting, when each tower is built, or upgraded in range, it checks all the gridSlots in range and adds a reference of itself to the “TowersInRange” attribute of the corresponding gridSlots.(notice that this action is called only 1 time either when a tower is built or upgraded in range) The “gridSlots” that have the boolean value “isPathway” set to true, check for the enemies inside them and sometimes apply debuffs like slow etc. After doing this check, the gridSlot sends references of those enemies each tower in the array “TowersInRange.” And that is how the targets are given to the towers.

Attributes:

private enum AttackType: There are attack type enumerations such as splash, single target, orb effect.

private float SlowRate: If this tower instance doesn’t have the ability to slow enemies, the slowRate is 0, if it has, than the slow rate is more than 0.

private int level: The level of the tower, can be upgraded.

private string description: The description of the tower, giving short info for a quick read.

private 2dImage TowerImage: The image of this tower, GraphicsManager gets a reference to this image via the “GameManager->TowerManager->ArrayList<Towers>->this tower instance” route.

public Enemy Target: The target that this tower instance attacks. This reference is passed to the projectiles spawned by this tower.

private int sellValue: The sell value of this, usually half of the buy value.

public int slot: A reference to the slot that this tower instance is built onto.

private float AttackRate: The attack speed(projectile spawned per second) value of this tower instance.

Methods:

private SpawnProjectile(): As soon as the “Target” attribute of this class changes from “NULL” to a valid object, the SpawnProjectile() function starts getting called every x seconds depending on the AttackRate float value.

public AddEnemyInRange(enemyInRange : Enemy): Sets the “Target” attribute of this class to the parameter “enemyInRange.”

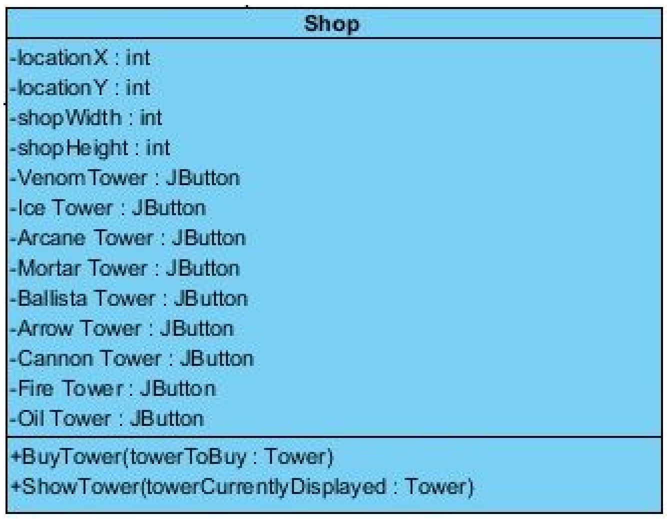
public RemoveEnemyInRange(): Sets the “Target” attribute of this class to “NULL.”

private Upgrade(): Upgrades the tower, increasing its’s attack rate, damage, and for some upgrades it’s range as well.

private SellTower(): Sells this tower and destroys this tower instance.

Deleted

Shop Class



Shop

-locationX : int
-locationY : int
-shopWidth : int
-shopHeight : int
-VenomTower : JButton
-Ice Tower : JButton
-Arcane Tower : JButton
-Mortar Tower : JButton
-Ballista Tower : JButton
-Arrow Tower : JButton
-Cannon Tower : JButton
-Fire Tower : JButton
-Oil Tower : JButton

+BuyTower(towerToBuy : Tower)
+ShowTower(towerCurrentlyDisplayed : Tower)

Figure 7.3.7 (Shop Class)

Attributes

Tower Types

private JButton Arrow Tower: If the player has enough gold, when clicked, this button gives the player the ability to place an “Arrow Tower” on a “GridSlot” the is not an “isPathway”.

private JButton Cannon Tower: If the player has enough gold, when clicked, this button gives the player the ability to place an “Cannon Tower” on a “GridSlot” the is not an “isPathway”.

private JButton Ice Tower: If the player has enough gold, when clicked, this button gives the player the ability to place an “Ice Tower” on a “GridSlot” the is not an “isPathway”.

private JButton Fire Tower: If the player has enough gold, when clicked, this button gives the player the ability to place an “Fire Tower” on a “GridSlot” the is not an “isPathway”.

private JButton Oil Tower: If the player has enough gold, when clicked, this button gives the player the ability to place an “Oil Tower” on a “GridSlot” the is not an “isPathway”.

private JButton Poison Tower: If the player has enough gold, when clicked, this button gives the player the ability to place an “Poison Tower” on a “GridSlot” the is not an “isPathway”.

private JButton Arcane Tower: If the player has enough gold, when clicked, this button gives the player the ability to place an “Arcane Tower” on a “GridSlot” the is not an “isPathway”.

private JButton Ballista Tower: If the player has enough gold, when clicked, this button gives the player the ability to place an “Ballista Tower” on a “GridSlot” the is not an “isPathway”.

private JButton Mortar Tower: If the player has enough gold, when clicked, this button gives the player the ability to place an “Mortar Tower” on a “GridSlot” the is not an “isPathway”.

Methods

public BuyTower(towerToBuy : Tower): Player will choose the tower to be bought via the side menu. towerToBuy will be the tower that’s been chosen by player. If player has enough gold, player will be able to click on tower and place it to a specified grid location. Yet if player lack of gold, BuyTower will not be activated and player will be warned instead.

public ShowTower(towerCurrentlyDisplayed : Tower):
The behaviours that player can apply are buy tower, and sell tower. These are defined in functions called BuyTower(). There is also a constructor that defines the overall Player object and sets default attributes to this object, such as life count, gold, and username. BuyTower() function allows player to purchase one of the towers and place it to a specified grid. Player will be able to choose a tower from the shop menu, in which all towers and their costs are displayed. Once the player has clicked a tower that he is able to buy, player will click again onto a gridSlot and BuyTower() function will be concluded. If the player clicks on a tower that he is unable to buy, there will be a pop-up informing the player about that matter.

Player Class

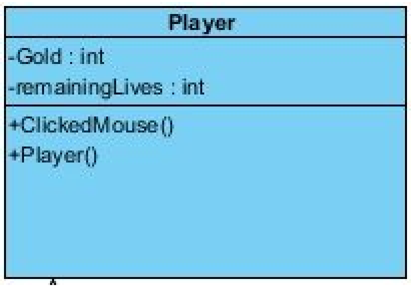


Figure 7.3.8 (Player Class)

Attributes:

private int Gold: The amount of gold that the player has which is used for buying new towers. For each enemy eliminated, this attribute increases.

private int remainingLives: The remaining lives of the player,

Constructors:

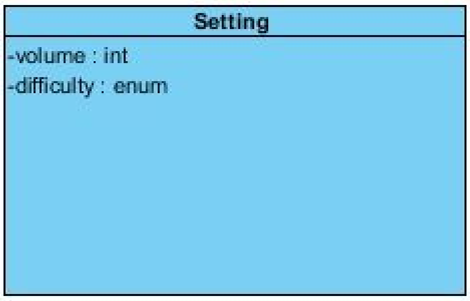
public Player(): Sets the default values for the “remainingLives” and the “gold.”

Methods:

public void ClickedMouse(): Method that is executed when the player has hit the left mouse button.

There is player class which describes the certain behaviour and attributes of the user. Attributes that are declared will also describe the player’s momentary statistics. As ordered, the attributes are username, gold, and remaining lives. Gold is an integer value and the currency of player. It allows player to buy new towers, and it’s gathered by player as the invaders are vanquished. Remaining lives is another integer value which determines the condition of player, by each trespassing enemy the remaining lives of the play drops by 1. If it drops down to 0, the player will lose the game.

Setting Class



Attributes:
public int volume: Keeps the volume proportion as an integer over 100 max value.
private enum difficulty: Difficulty types are one of the 5 enumerated types: “sandbox, easy, medium, hard, hardcore.”

Projectile Class

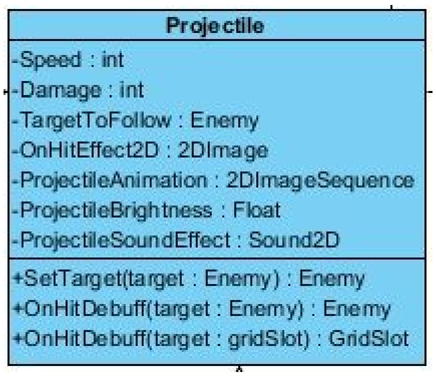

Projectile
-Speed : int
-Damage : int
-TargetToFollow : Enemy
-OnHitEffect2D : 2DImage
-ProjectileAnimation : 2DImageSequence
-ProjectileBrightness : Float
-ProjectileSoundEffect : Sound2D
+SetTarget(target : Enemy) : Enemy
+OnHitDebuff(target : Enemy) : Enemy
+OnHitDebuff(target : gridSlot) : GridSlot

Figure 7.3.9 (Projectile Class)

Attributes:

private int Speed: This attribute indicates the speed of projectile. Each tower has different projectile speed and so different attack frequency.
private int Damage: There is also damage attribute which remarks the damage output of the projectile.

private Enemy TargetToFollow: Projectile's target is identified with TargetToFollow attribute. Each tower can target one enemy at a time.

public 2DImage OnHitEffect2D: When enemies got hit by projectile, there will be a special effect generated and this effect is denoted with OnHitEffect2D attribute.

public 2DImageSequence ProjectileAnimation: This image sequence is drawn by the graphicsManager.

public float ProjectileBrightness: This value is used when drawing the projectile animation and it determines the brightness of the projectile. The brightness usually increases when the power of the projectile is stronger.

public Sound2D ProjectileSoundEffect: The sound effect to be played when the projectile hits an enemy.

Methods:
public Enemy SetTarget: Sets the "TargetToFollow" attribute of this class. This method is called by the tower instance that has spawned this class.

private Enemy OnHitDebuff: This is the method for applying the minus armor, slowRate orb effects on the enemies that are hit by this projectile instance.

private GridSlot OnHitDebuff: This is the method for applying the minus armor, slowRate orb effects on a particular gridSlot so that all the enemies that are in that slot are affected..

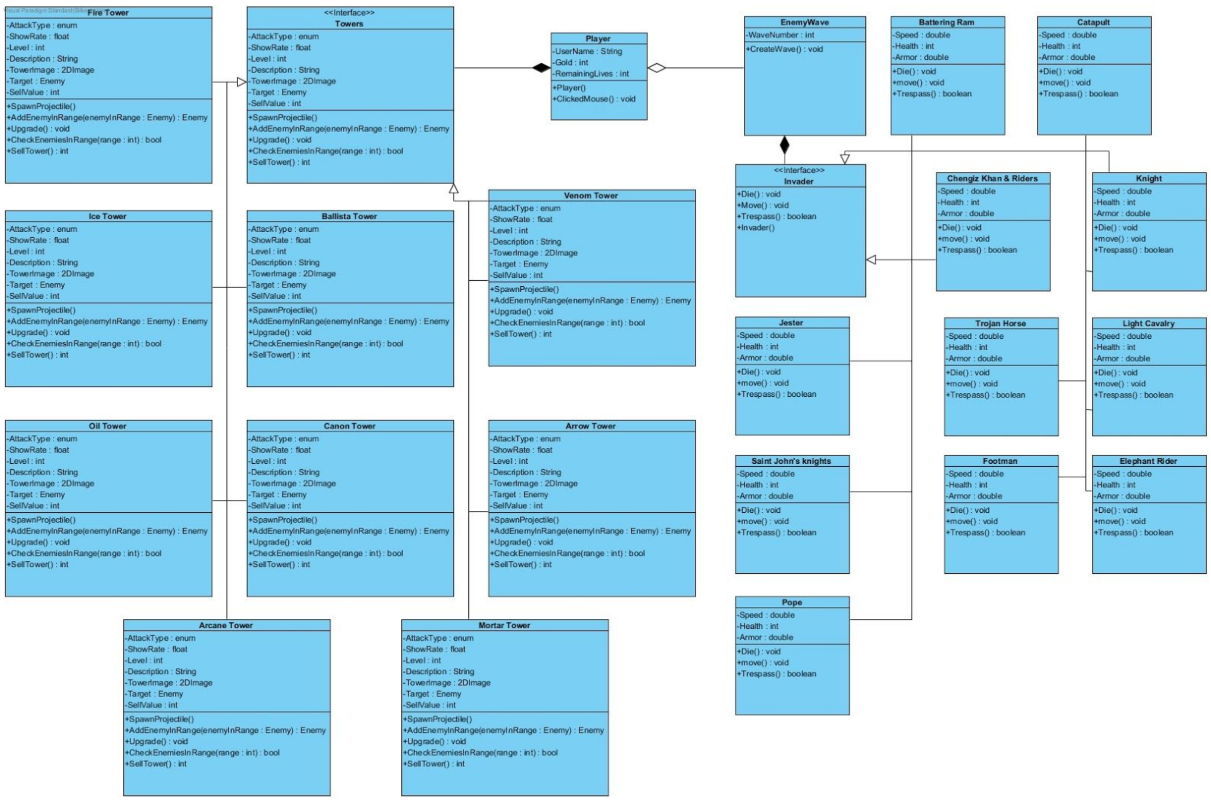


Figure 7.3.10 (Polymorphism Diagram)

Towers<parent>

Towers class will serve as an interface for the specialized tower classes. The concepts of inheritence and polymorphism will take place so that there will be towers with different characteristics and different attributes. Yet at the end, behaviours and attribute types of all towers will be same. Attributes specified until now are Description, Damage, Cost, and **UpgradeLevel.** Description is a string variable that describes the characteristic of the tower that player points on. It helps player to choose the most fitting tower for the situation that player needs to handle. Damage is the raw power that towers beholds It is identified as a double variable, because the damage will vary according to the armor of the target invader. Cost variable will be on charge when the player decides to buy a new tower. Towers which have the higher capability of crowd control or the ones which can take individual targets easily will be more expensive, so buying a new tower can be a tactical aspect of the game. Upgrade level is the last attribute to be represented and it will deny player from upgrading the tower forever. Towers that are cheaper will have more upgrade levels, yet upgrading them will cost less when compared to more expensive towers.

In the current state, towers have two behaviours and these are Shoot(), and UpgradeLevel(). Both of them have the return type of void. Shoot() function acts differently for each specialized tower. This function certainly describes the characteristic of each specialized tower. The splash damage made, speed and armor debuffs that’s been caused, raw damage output, and damage frequency are all handled in this function for each different polymorphic tower class. Last function, UpgradeLevel() basically handles the constraint of upgrade limit specified for different types of towers. It permits or doesn’t let player to upgrade the tower that’s been chosen.

Methods:

public Projectile SpawnProjectile(): This function acts differently for each specialized tower. Certainly, it describes the characteristic of each specialized tower. The splash damage made, speed and armor debuffs that’s been caused, raw damage output, and damage frequency are all handled in this function for each different polymorphic tower class.

public bool CheckEnemiesInRange(int range): Serving as an auxiliary function, CheckEnemiesInRange will function as the sensor of tower. The integer parameter denoted as range will check the enemies around in each time interval.

public Enemy AddEnemyInRange(Enemy enemyInRange): In case of an enemy detection, AddEnemyInRange will notify tower about enemy activity, and will let tower to activate the function responsible for taking down the invaders.

public void Upgrade(): It basically handles the constraint of upgrade limit specified for different types of towers. It permits or doesn’t let player to upgrade the tower that’s been chosen.

public int SellTower(): When player decides to sell the chosen tower, SellTower function returns the amount of gold that’s specified for the tower to be sold. If tower has been upgraded previously, player will receive more gold by selling it.

Attributes:

private enum AttackType: Attack type may be splash damage or single target.

private float SlowRate: Slow rate to be applied to the enemy.

private int Level: This attribute will indicate the current level of tower

private String Description: Each tower will have a description indicating its certain characteristics with a brief description

private 2DImage: There will be an image display of each tower. Every tower will have different image.

private Enemy: This attribute will identify the enemy targeted by the tower.

private Int SellValue: Sell value remarks the chosen tower's value in case of sell consideration

Classes derived from Towers Parent

ArrowTower Class:

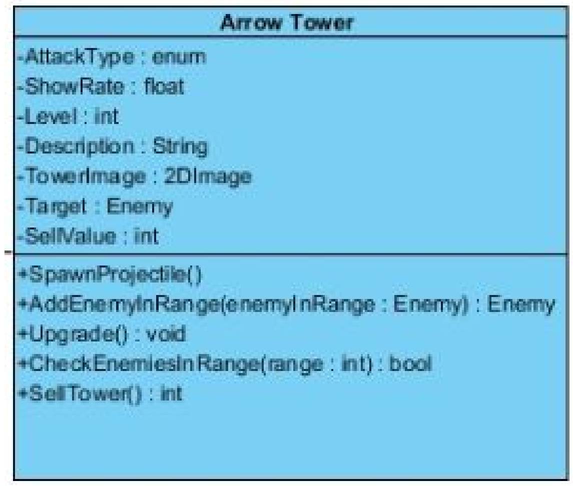


Figure 7.3.11 (Polymorphism Diagram)

Constructor:

public ArrowTower: The constructor of this class. The proper image and attributes will be specified with the constructor.

CanonTower Class:

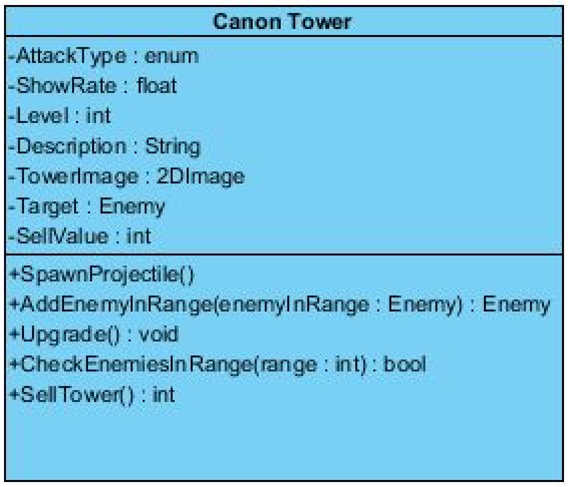


Figure 7.3.12 (Polymorphism Diagram)

Constructor:

public CanonTower: The constructor of this class. The proper image and attributes will be specified with the constructor.

IceTower Class:

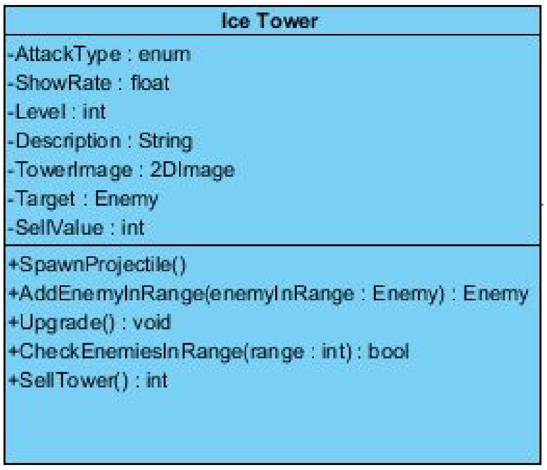


Figure 7.3.13 (Polymorphism Diagram)

Constructor:

public IceTower: The constructor of this class. The proper image and attributes will be specified with the constructor.

FireTower Class:

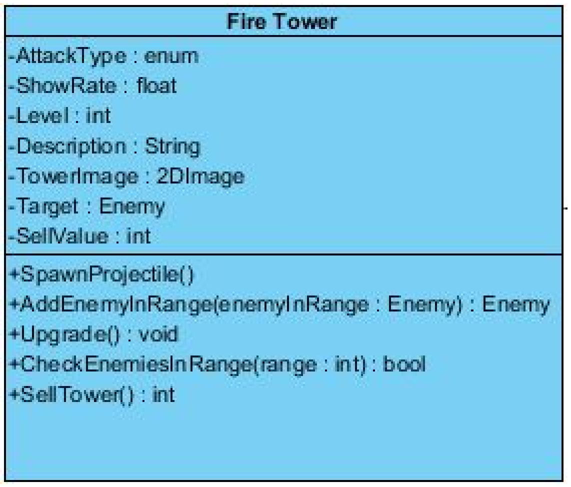


Figure 7.3.14 (Polymorphism Diagram)

Constructor:

public FireTower: The constructor of this class. The proper image and attributes will be specified with the constructor.

OilTower Class:

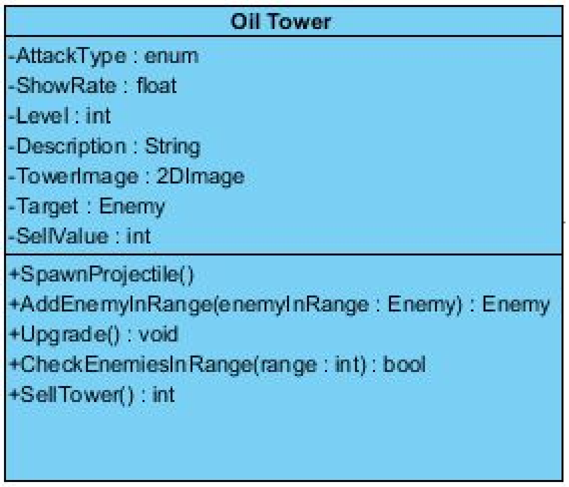


Figure 7.3.15 (Polymorphism Diagram)

Constructor:

public OilTower: The constructor of this class. The proper image and attributes will be specified with the constructor.

PoisonTower Class:

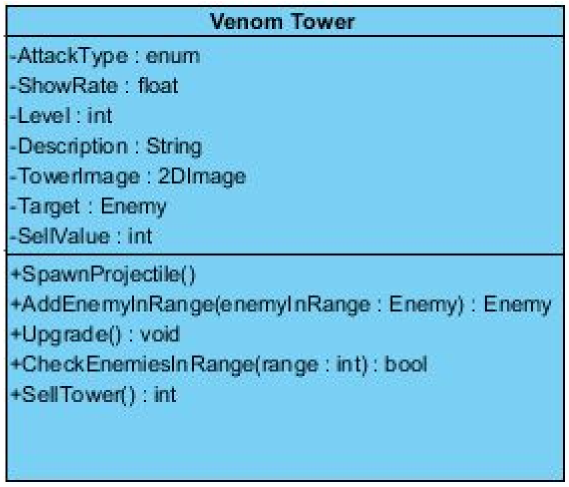


Figure 7.3.16 (Polymorphism Diagram)

Constructor:

public PoisonTower: The constructor of this class. The proper image and attributes will be specified with the constructor.

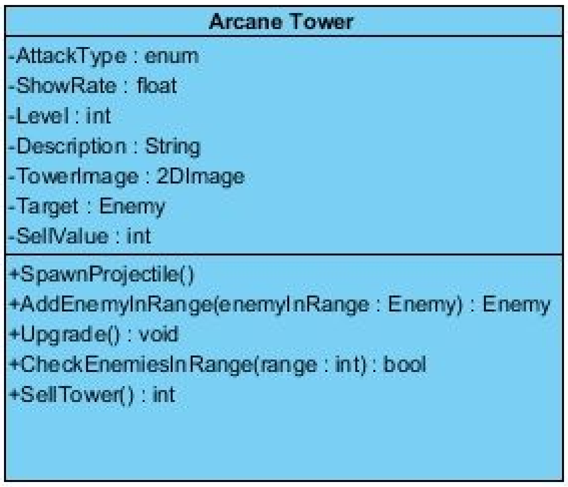


Figure 7.3.17 (Polymorphism Diagram)

Constructor:

public ArcaneTower: The constructor of this class. The proper image and attributes will be specified with the constructor.

BallistaTower Class:

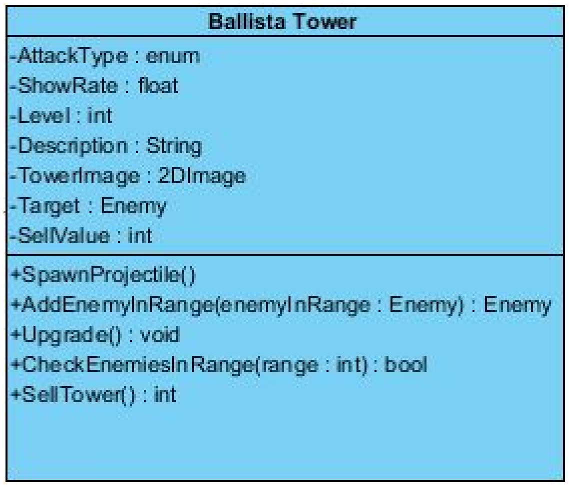


Figure 7.3.18 (Polymorphism Diagram)

Constructor:

public BallistaTower: The constructor of this class. The proper image and attributes will be specified with the constructor.

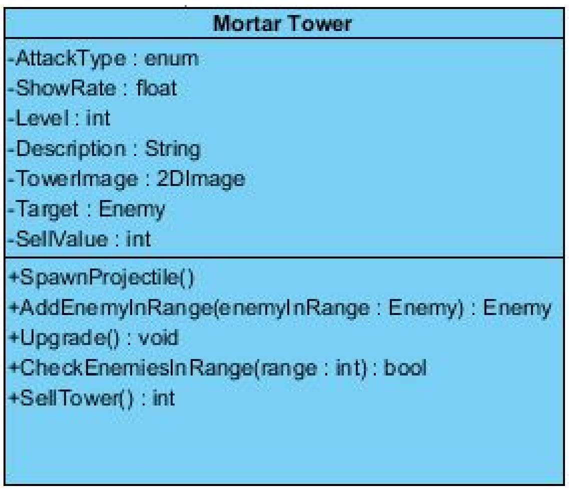


Figure 7.3.19 (Polymorphism Diagram)

MortarTower Class:

Constructor:

public MortarTower: The constructor of this class. The proper image and attributes will be specified with the constructor.

Invader

Invaders are individual unit types that assemble the body of an enemy wave. Invader class serves as parent of polymorphic enemy types that have different attributes. Main goal of an enemy invader is to survive the defense mechanisms that's been set by player and move along to the player castle. Attributes identified for the Invader class are speed, health, and armor. As default, all of these attributes are set in the constructor of different enemy types. Speed is an integer value and it declares how fast the invader is. Health is another integer value and it's the measure of how much can invader endure to the damage input. Once the invader loses all health points, player will earn gold. Armor is a double type attribute and it reduces the damage input with the multiplier identified for each different invader.

Methods:

public void Die(): It informs the invader's progress to the player. If Die function is been activated in the case of health bar drops zero, player will receive money in conclusion.

public void Move(): Another void function Move() will command invader to move on as long as the invader is not dead. In each frame, invader will move from one place to another according to the speed value.

public bool Trespass(): Trespass() will be active if invader manages to cross aside the player keep. In conclusion the player's health bar will be reduced by one.

Attributes:

private double Speed: Speed attribute will differ for any type of enemy. It basically describes how speed an enemy is.

private int Health: Health value shows how much damage an enemy can endure.

private double Armor: Armor will split the damage input of an enemy individual and more armor means less health points to be reduced.

Classes Derived From Invader Parent

Catapult

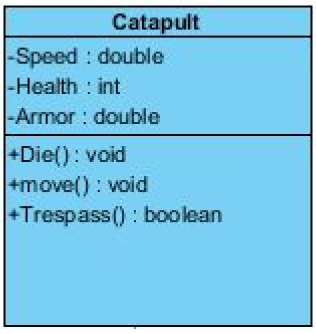


Figure 7.3.20 (Polymorphism Diagram)

Constructor:

public Catapult(): The constructor of this class. The proper image and attributes will be specified with the constructor.

Chengiz Khan & Riders

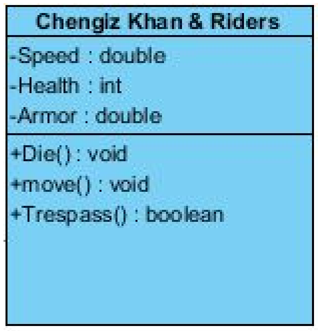


Figure 7.3.21 (Polymorphism Diagram)

Constructor:

public ChengizKhanAndRiders(): The constructor of this class. The proper image and attributes will be specified with the constructor.

Footman

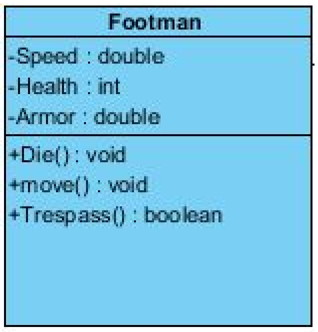


Figure 7.3.22 (Polymorphism Diagram)

Constructor:

public Footman(): The constructor of this class. The proper image and attributes will be specified with the constructor.

Jester

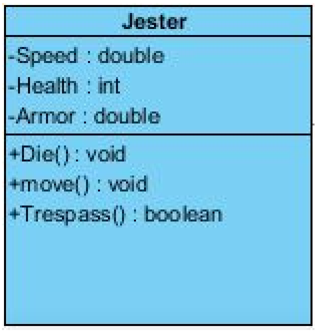


Figure 7.3.23 (Polymorphism Diagram)

Constructor:

public Jester(): The constructor of this class. The proper image and attributes will be specified with the constructor.

Knight

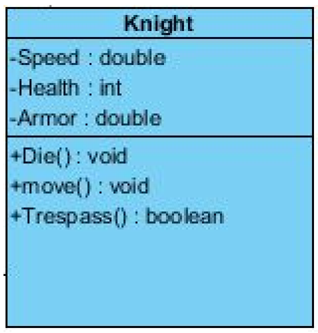


Figure 7.3.24 (Polymorphism Diagram)

Constructor:

public Knight(): The constructor of this class. The proper image and attributes will be specified with the constructor.

Light Cavalry

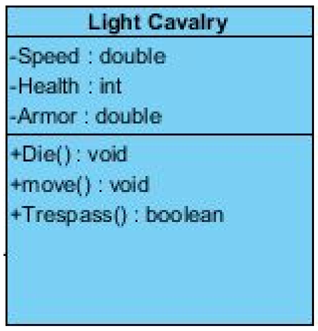


Figure 7.3.25 (Polymorphism Diagram)

Constructor:

public LightCavalry(): The constructor of this class. The proper image and attributes will be specified with the constructor.

Pope

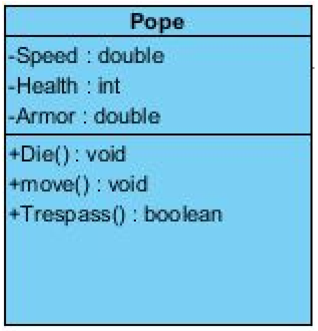


Figure 7.3.26 (Polymorphism Diagram)

Constructor:

public Pope(): The constructor of this class. The proper image and attributes will be specified with the constructor.

Battering Ram

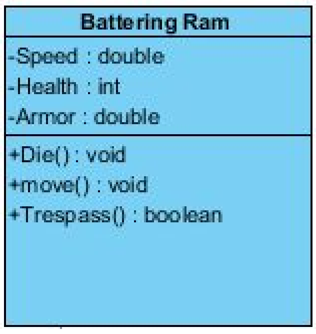


Figure 7.3.27 (Polymorphism Diagram)

Constructor:

public BatteringRam(): The constructor of this class. The proper image and attributes will be specified with the constructor.

Trojan Horse

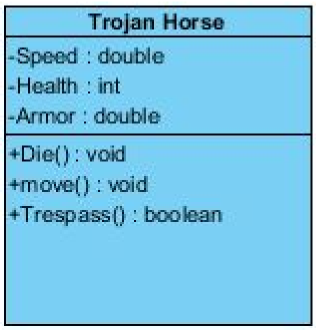


Figure 7.3.28 (Polymorphism Diagram)

Constructor:

public TrojanHorse(): The constructor of this class. The proper image and attributes will be specified with the constructor.

Saint John's Knights

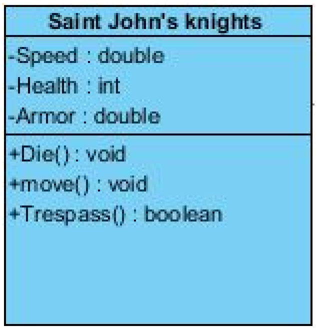


Figure 7.3.29 (Polymorphism Diagram)

Constructor:

public SaintJohnKnights(): The constructor of this class. The proper image and attributes will be specified with the constructor.

EnemyWave

The invaders will come as divisions, and these divisions are called as enemy waves. There will be a certain number of waves and this number will be recorded in EnemyWave() class, as WaveNumber attribute.

Methods:

public void CreateWave: This method acts if all individual invaders are dead and WaveNumber constraint allows to new waves be generated.

9. Conclusion

The design goals and methodologies were dictated our first final iteration and the plans which were described in the first analysis and design reports are followed. In the current stage, the application is capable of performing the essential functionality of a tower defense game. Enemies follow a certain path, and the towers can be built to custom grid locations. Different enemy and tower types were implemented with parent-child relationship and behaviour of each different type is introduced in inherited classes. There is one level at the current stage of the game, and yet more levels still to be implemented with the usage of enemy wave classes. This was eventually the case with the system design at hand.

The UML diagrams were helpful in terms of visualization of the design at hand. The usage of use-case diagrams, sequence diagrams, and scenarios were insightful. They have identified the systems to improve and the systems to fix. These diagrams were implemented as been thought by the instructor.

The design goals, which assemble the non-functional requirements section were user-friendliness, portability, good documentation, and high performance in this project. Relationship between the subsystems are created according to the relationships defined in the analysis and design sections. Subsystems were designed to be comprehensive, and well-maintainable.Component management, and session management were handled with the classes which are specifically created. The main system, is connected with the composition of transparent waves in which each layer can call operations from any layer above or below. The goal with such a terminology is the runtime efficiency. The subsystems are high-coupled more than high-cohered. This being one of the architectural styles, another architectural style been used is Model view controller. The model counterpart is mainly responsible for the knowledge and commands of different application domain. The view counterpart is for the visualization of these application domains, and the controller counterpart is responsible with the management of interactions among the domains.

In case of dynamic modeling, parallelism is not yet introduced to the system design, but it may be also included at future according to the suggested criteria. Among the types of software control, event-driven is the most convenient one for this software developed. Because in game menu or in UI interface sections, events flow according to the player actions. The player actions are validated with the action listeners which are declared in GUI classes. The interactions between subcomponents of GUI system are handled by GUI manager class. The subcomponents handle the basic functionalities of the UI sections which are information, help, high scores, and credits. This class handles the other classes as Façade design pattern suggests. Global resource handling is made as control list access. The towers, and enemies are added one by one to the lists.

10. Test Plan

10.1. Requirements/specifications-based system level test cases

In the current state, the full functionality is not reached, and design goals are still the concern. However, the trace algorithm for the Enemies work fine, for future, it must work for the maps that have different pattern. Towers shall also be placed to proper locations, they can't be placed on the area specified as road grid. For the newly added towers, there must be enough room in the shop menu, and the towers in the shop menu shall obey to the actionlistener commands.

11. References

https://sites.google.com/site/knightsvszombies/
http://www.cs.bilkent.edu.tr/~ugur/teaching/cs319/proj/11_2C/analysis.doc
https://docs.oracle.com/javase/tutorial/uiswing/index.html