CS 319 - Object-Oriented Software Engineering

System Design Report

Iteration 2

Medieval Tower Defense

Group 1-A

Buğra Aydın

Berke Deniz Başaran

Mahir Özer

1.Introduction

1.1 Purpose of the System

Medieval Tower Defense is a 2-D tower defense game. Gameplay of MTD will be designed simple in order to have a wide range of player base, but still with its in-depth complexity with required strategies to defeat unique encounters will give players the pleasure of achievement. The concept of the game will be very simple, yet the player will have to generate different strategies in order to handle different enemies and bosses with different attributes. MTD will also have distinguishable content from other tower defense games as it will have unique boss encounters, difficulty levels and game mods. MTD's goal is to improve decision making, situation judging and sense of strategy while having a good time. Purpose of the Medieval Tower Defense Game is to offer a quality gameplay experience to the strategy game audience with an easy to use and control user interface and dynamic gameplay. Also, audience will be able to obtain historical information by while playing the game.

1.2 Design Goals

It is really important to decide on the design goals of the system in order to have a brief idea on the qualifications that the program contain. Due to that, while deciding on the design goals, we focused on our functional and nonfunctional requirements of the system. Important design goals of this system are explained below.

Performance Criteria:

Response Time and Throughput

Our system is designed to have a satisfying response time, since the game requires almost instant responses for user inputs in order to maintain a healthy game play. Also, system is designed to have a good throughput to respond to several user inputs and make corresponding changes in a fixed period of time. The system is going to be responsive and able to run with high performance. This is the most important design goal because performance of the game has a crucial role for users' excitement.

Memory

Since Medieval Tower Defense does not require a big memory space that should be considered, our main goal in this section was to have enough memory available for speed optimizations.

Dependability Criteria:

Reliability

System is designed to be bug-free and consistent in the boundary conditions. There should be no crashes and unintended actions such as glitches and bugs.

Cost Criteria:

Development Cost

Since Medieval Tower Defense is not a commercial product, cost of the initial system shouldn't be too expensive. In order to match with this design goal, images for the game elements will either be drawn by developers or be supplied from the existing free resources.

Deployment Cost

Since Medieval Tower Defense is not a commercial product, we decided to make our game free to install and use.

Maintenance Criteria:

Extensibility

It is important to add new content and features to keep the interest and excitement to a game to keep it alive. With our design perspective, it is possible to easily extend and add new features to the existing system such as new enemies, new tower types, new waves and different difficulty settings.

Modifiability

It is possible to modify our game content with our system design. Since our game might require balance updates for enemies and towers with nerfs and buffs, it was really important to consider modifiability.

Portability

Java is one of the programming languages which provide cross-platform portability. Java works on a portable Java Virtual Machine(JVM) that can be adapted to different machines. That makes our system to be possible to work on different platforms. So, our system is portable.

Readability

The system is designed to make the developer's life easier by making it simple to understand the system by reading the code. Object oriented software engineering perspective is really useful to met with this design goal since the real life objects, their attributes and their methods are represented accordingly in the sofware system.

End User Criteria:

Usability

Easiness in the usage is an important design goal in such games in terms of user's comfort. It makes the game more friendly and attractive. Therefore, the system will be designed such that user can easily interact with the system. User interface menu of the game will be as understandable as possible due to that. The game will have a help document in the main menu and in-game menu also, in order to provide ease of use for users. However, user friendliness of the system does not imply making the gameplay easier, which might make the player bored sooner than expected. Since our system is a game, it should provide good entertainment for the player. In order to provide the entertainment player should not have a difficulty in learning our system. In this respect, system will provide player friendly interfaces for menus, by which player will easily find desired operations, navigate through menus and perform the desired operations. User can easily access information about units from the information screen and develop strategies corresponding to those informations.

Design Goal Trade Offs:

Response Time and Throughput vs. Reusability

We are not considering to integrate any of our classes in any different game or any other projects, so reusability is not our main concern. Therefore, the classes will designed specifically for the tasks of our game so the code is not made more complex than necessary. This design approach will fortify the our most important design goal, which is response time and throughput.

Space vs. Speed

Our system is to be expected to be interactive, interesting and smooth. So, performance is our main aim since it is not a big project that needs to consider the memory efficiency. We may need to sacrifice memory in order to gain performance. For example, we store all waves, towers and enemies in our memory in order to gain performance, even though sacrificing memory is needed.

1.3 Definitions, acronyms and abbreviations

Cross-platform

Cross-platform refers to ability of software to run in same way on different platforms such as Microsoft Windows, Linux and Mac OS X.[1]

JRE

The Java Runtime Environment (JRE), also known as Java Runtime, is part of the Java Development Kit (JDK), a set of programming tools for developing Java applications.[2]

FPS

Abbreviation of “frames per second”. Fps represents the number of graphical layouts can be prepared by the system each second.[3]

1.4 References

[1] https://www.webopedia.com/TERM/C/cross_platform.html

[2] http://www.theserverside.com/definition/Java-Runtime-Environment-JRE

[3] https://www.computing.net/define/fps

2. Software Architecture

2.1. Overview

This section describes the proposed software architecture of the system.

The system proposed decompose the main system into a collection of subsystems. These subsystems will be much more maintainable. We tried to decompose the system in to subsystems while trying to obtain a Three-layered architectural style.

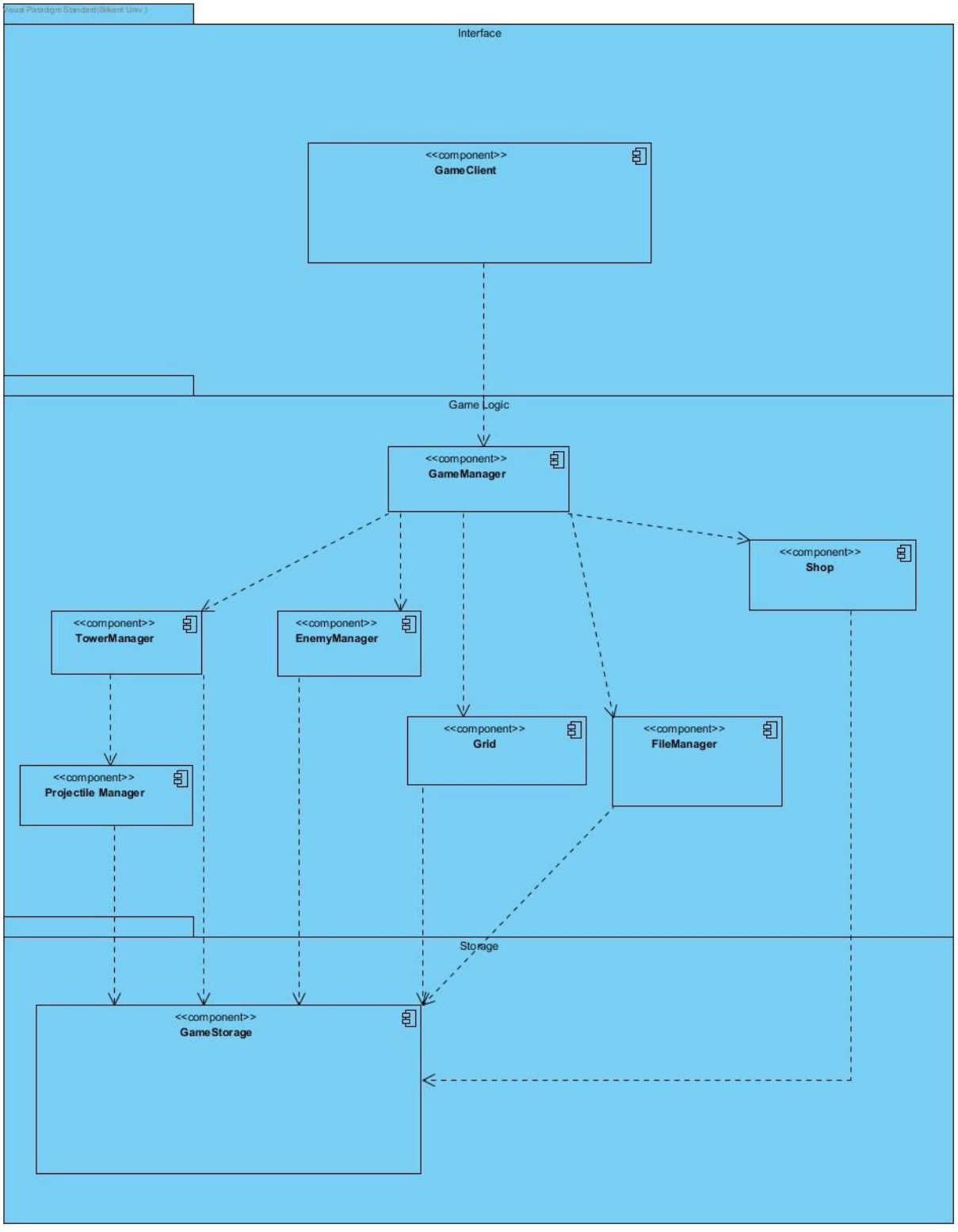
2.2. Subsystem Decomposition

This section decomposes the system into several subsystems. This provides us modifibility and extendibility.

We decided to use **three-tier architecture**. This architeture style consist three layers. In the top layer, there is user interface subsystem. In the mid layer, there is game logic subsystem. In the bottom layer, there are data storage subsystem. At total, three layers

are used in order to describe the workflow with less complexity and in order to meet the non-functional requirements. Layers in this system design are opaque, meaning each layer can only access and call operations from layers below.

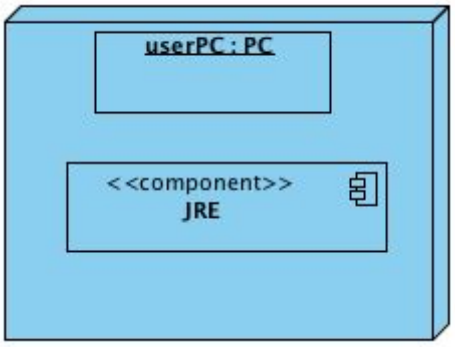
If there is any case of modifying the existing code for the reason of performance improvement, this decomposition will allow us to navigate our path and solve the bugs or fix the working code with better efficiency. Otherwise, there will be louse couplings, and without the usage of diagrams at hand, code will not be maintainable.



High-level structure of the program is shown with a component diagram which uses 3-Layers. Control flow is from up layer, which contains main control structures, to the down layer, which contains the collection of the game storage objects such as Projectile, Tower, Enemy, and Traps.

2.3. Hardware/Software Mapping

First of all, Medieval Tower Defence will not use any external hardware component in order to run. Our game is written in Java, hence it will be using Java Runtime Environment (JRE) while running. Game will be running in RAM but since it is similar to an Indie game, very basic components will be enough in order to run smoothly. Also, Medieval Tower Defence runs using only mouse. Only interesting thing is that we will be using txt files to save game. This means we will be using a small amount of Hard Drive too, additional to the size of the game. A simple deployment diagram is added below which focuses on giving a high-level view of the single component we use.



2.4. Persistent Data Management

Medieval Tower Defense will store the high scores in a .text file. When the player clicks on the High Scores tab from the provided menu, 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.

2.5. Access Control and Security

In order to access the game, downloading the .jar file is enough so there won't be any special acces control cases. Playing the game won't require any internet connection.

There won’t be any security checkings going on with the system since the game requires no personal information except a nickname, therefore a security issue is not the case.

2.6. Global Software Control

Event-driven Control

In our design, we are using the event-driven control for our game logic and updates. This software control design makes it simple to structure the classes and events. It makes it possible to centralize a main updateObjects method inside the GameManager class that waits for an external event. This external event in our case is time. This updateObject method executes ten times in a second by providing a timer event. However, this control mechanism has a trade-off. Even though it makes the update methode simple and centralized, implemantion of multi-step sequences are much more difficult for developers to implement.

2.7. Boundary Conditions

Start-up

Medieval Tower Defense does not require any setup but installation of the .jar file. Start-up condition will start as the .jar file is opened by the user.

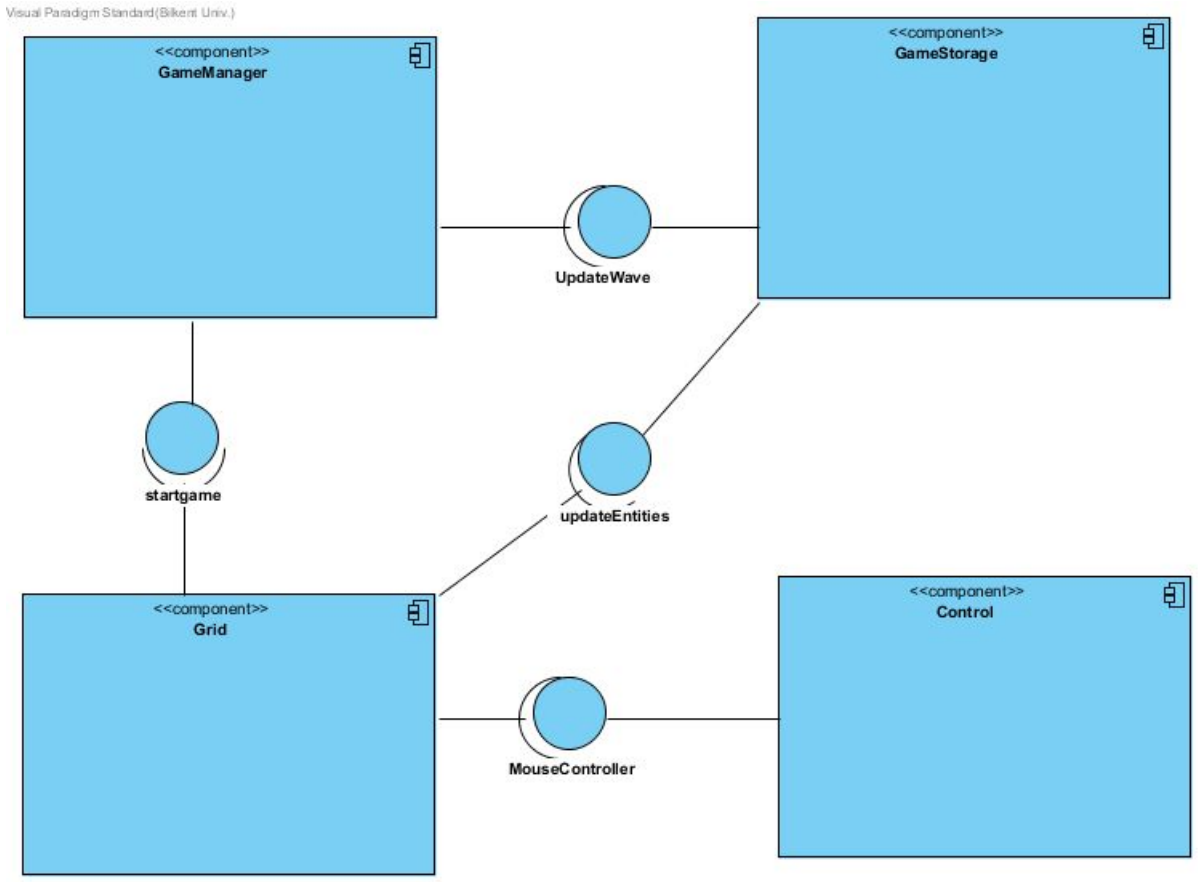
Shutdown

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.

Exception Handling

Even though we will try our bests to reduce the amounts of errors as much as possible, if an error occurs during the gameplay, the system will provide user and explationary error message. If an error occurs while loading images or sounds, game will not be able to start.

3. Subsystem Services

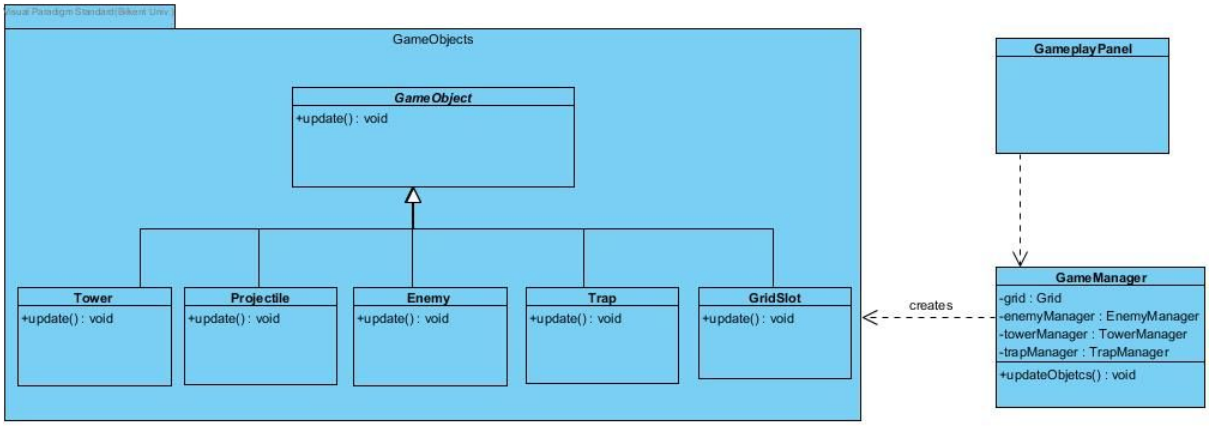


3.1 Design Patterns Used

Behavioral Patterns

Façade

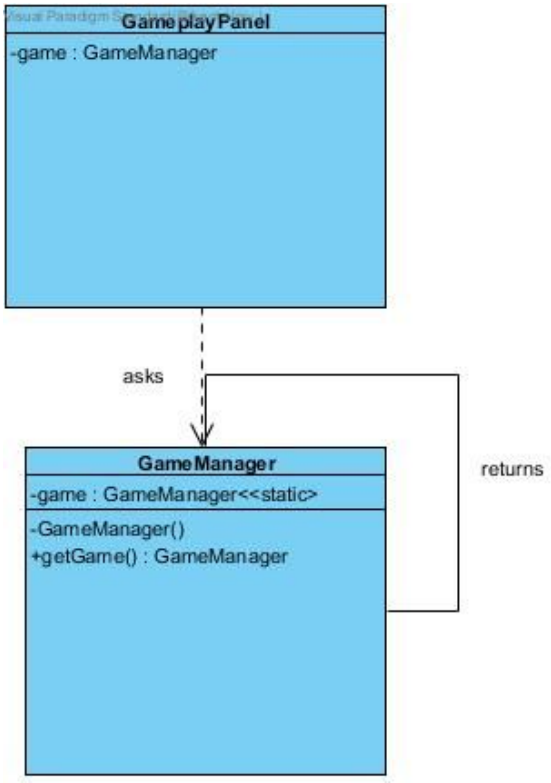
In our design, we decided to use façade design pattern to provide an interface for the GameplayPanel class that user is accessing the system from. GameManager class is the façade class that calls complex update method for each game object in the current game and provide a simplified updateObject method for the client. By making this pattern, we delegate calls to the update methods of the GameObjects from GameplayPanel class and make it much more easier and flexible to make changes in any game object.



Creational Patterns

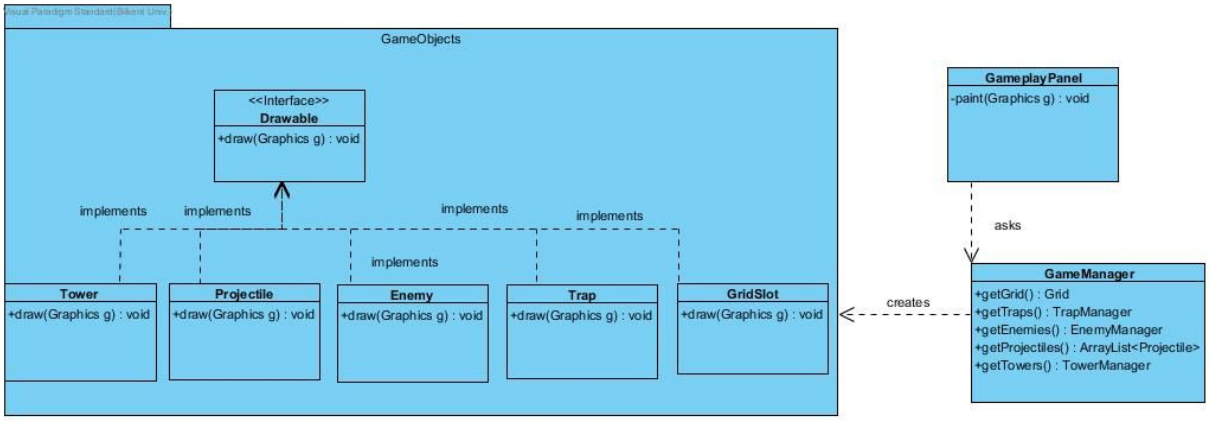
Singleton Design Pattern

In our design, we decided to use singleton design pattern to make sure that only a single instance of game manager is up unless the old one is nullified and replaced. In this pattern, GameManager class has a private static GameManager attribute, called game. In its publicly exposed getGame method, if the game is null, it creates an object and assigns it to the game attribute. If the game is not null and there is a concurrent game, it just returns game. By doing this pattern, we make sure that classes using an instance of GameManager such as GameplayManager can only create a single game at a time.

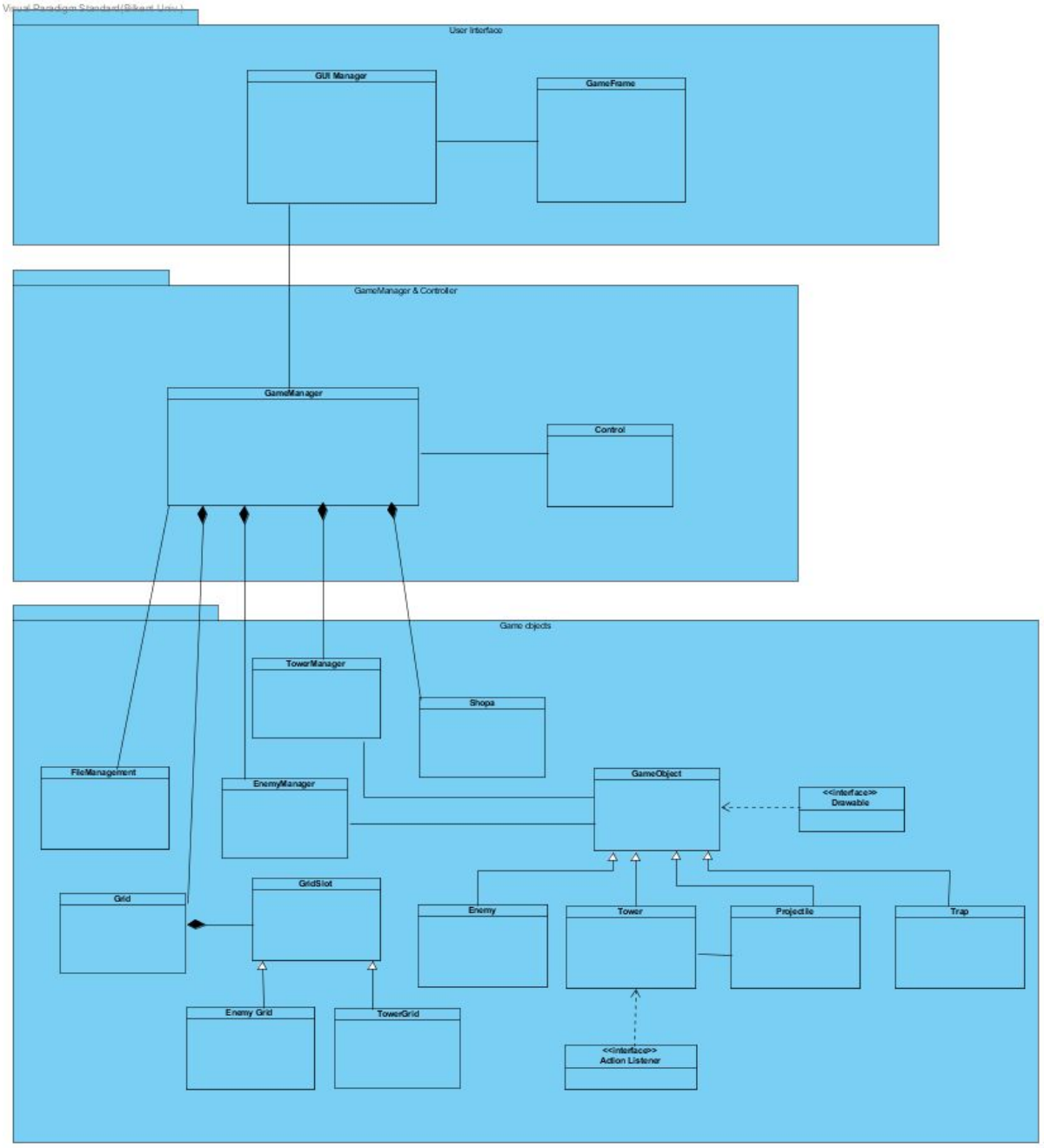


Abstract Factory Design Pattern

In our design, we decided to use abstract factory pattern to be able to create game objects without exposing them to outside. Concrete classes Tower, Projectile, Enemy, Trap and GridSlot implements the Drawable interface, which has a void draw method that takes Graphic object as its parameter. Those classes define their draw method according to their own specialities, such as different image file paths. GameManager is the factory class that creates game objects in corresponding forms with its corresponding getter methods. GameplayPanel is the client object that has a method paint, that calls draw method on each object after asking for those object from the factory class GameManager. GameplayPanel can only access those game objects using factory class GameManager.



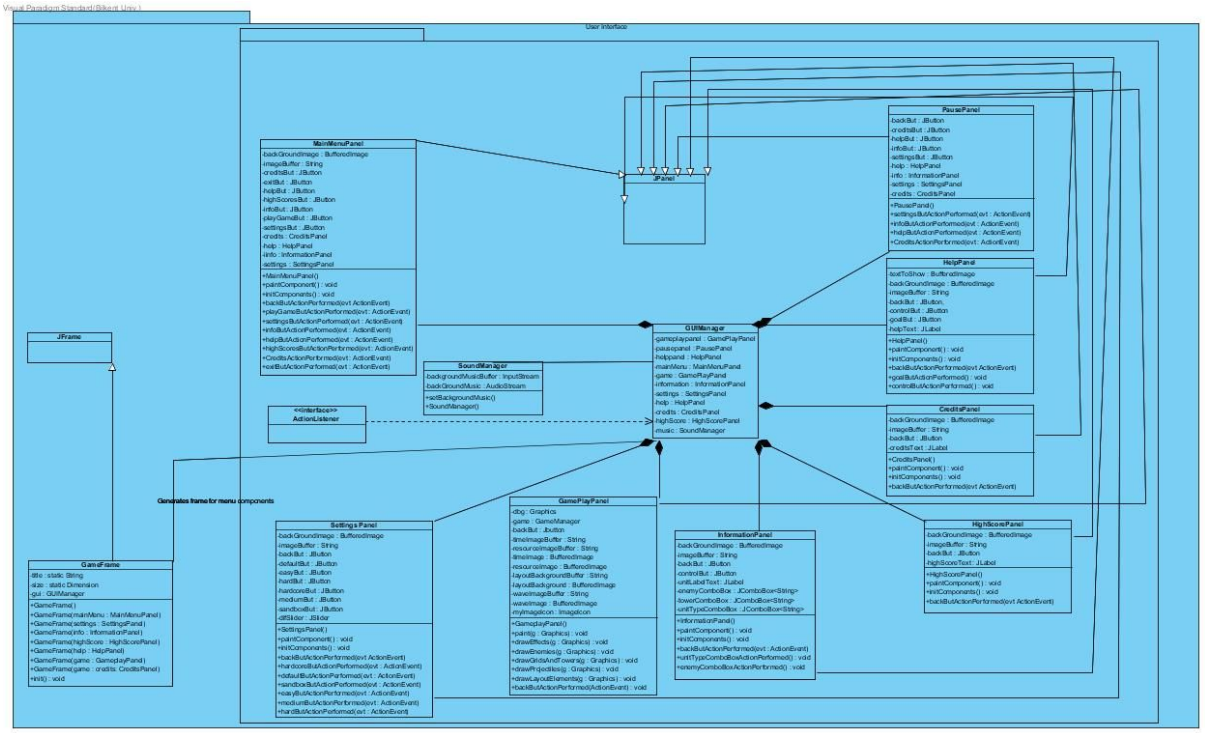
3.2 High Level Design



Game manager is the controller class updates all the model classes(game objects such as towers, enemies etc.) Game manager holds the manager classes as controller and updates the game objects through the manager classes which hold the game objects. The gameplay panel is the “View” in the MVC design. Game manager updates the view(GUI Manager&GameplayPanel) as well.

3.3 Low Level Design

3.3.1 User Interface Subsystem



GameplayPanel Class

Attributes

private GameManager game: A reference to the controller class gameManager.

private JButton backBut: A back button to back out of the game.

String timeImageBuffer: The system directory to read the image to indicate time.

String resourceImageBuffer: The system directory to read the image to indicate the resource type.

BufferedImage timeImage: An image to indicate time.(A clock image)

BufferedImage resourceImage: The image to indicate the resource type.(A gold coin image)

String layoutBackgroundBuffer: The system directory to read the image for the background of the layer.

BufferedImage layoutBackground: An image for the background of the layer.

String waveImageBuffer: The system directory to read the image to display the wave.(A face image)

BufferedImage waveImage: An image to display the wave.(A face image)

Methods

public void paint(Graphics g): Paint method to draw all the game objects.

drawEffects(Graphics g): The method to draw the impact effects for projectiles. Called in the paint method of this class.

drawGridsAndTowers(Graphics g): The method to draw the map and the towers.(i.e. to draw the gridSlots and the towers on top of the gridSlots. Called in the paint method of this class.

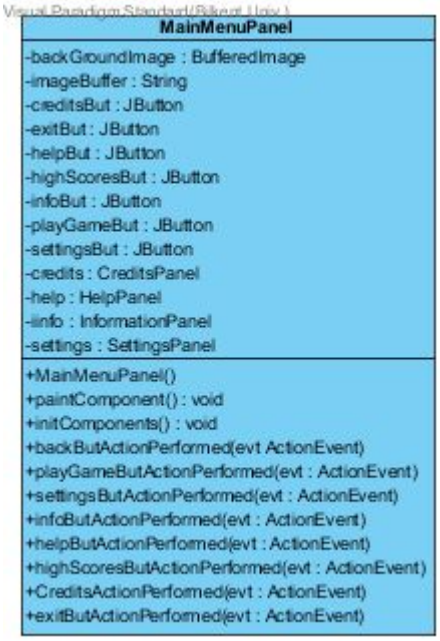
drawEnemies(Graphics g): The method to draw every enemy instance on the grid. Called in the paint method of this class.

drawProjectiles(Graphics g): The method to draw every projectile on the grid. Called in the paint method of this class.

game.getShop().draw(Graphics g): The method to draw the shop. Called in the paint method of this class.

drawLayoutElements(Graphics g): The method to draw the layout elements like gold, time, current wave etc. Called in the paint method of this class.

MainMenuPanel Class



BufferedImage backgroundImage : This is an image which is used to display the background.

String imageBuffer : This String is used in printing some writings on the main menu, such as “Medieval Tower Defence”.

CreditsPanel credits : This panel displays credits. Is redirected to by credits button.

HelpPanel help : This panel displays help. Is redirected to by help button.

InformationPanel info : This panel displays information. Is redirected by info button.

SettingsPanel settings : This panel displays settings. Is redirected by settings button.

private javax.swing.JButton Credits : This is the Button used in display of Main Menu which will show credits when activated. Developers of the game will be shown when clicked.

private javax.swing.JButton exitBut : This is the Button used in display of Main Menu which will exit the game when clicked.

private javax.swing.JButton helpBut : This is the Button used in display of Main Menu which will display help menu when clicked. Help menu enables user to gain information about the game and learn their options.

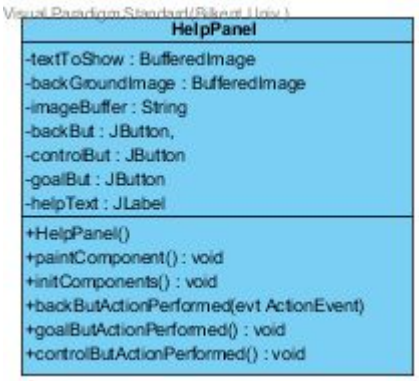
private javax.swing.JButton highScoresBut : This is the Button used in display of Main Menu which will display help menu when clicked. User can see their high scores from here.

private javax.swing.JButton infoBut : This is the Button used in display of Main Menu which will display information about the game when clicked. Difference from help button is that this button will show information about hardware components, not in-game help.

private javax.swing.JButton playGameBut : This is the Button used in display of Main Menu which will start the game when clicked.

private javax.swing.JButton settingsBut : This is the Button used in display of Main Menu which will redirect user to Settings.

HelpPanel Class



BufferedImage backgroundImage: The image to display in the background.

String textToShow: The help text that player can read from this panel.

String imageBuffer: The system directory to read the image of Help Panel Button.

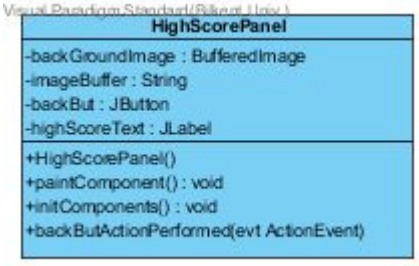
private javax.swing.JButton backBut: In HelpPanel, a JButton instance is called and it's modified to a button with "to control page" functionality

private javax.swing.JButton controlBut: In HelpPanel, a JButton instance is called and it's modified to a button with "to control page" functionality

private javax.swing.JButton goalBut: In HelpPanel, a JButton instance is called and it's modified to a button with "to goal page" functionality

private javax.swing.JLabel helpText: In HelpPanel, a JLabel instance is called and it's modified to a Label with "display informing text" functionality

HighScorePanel Class



BufferedImage backgroundImage: The image to display in the background of HighScorePanel.

String imageBuffer : Address of image to be used is defined with this String attribute.

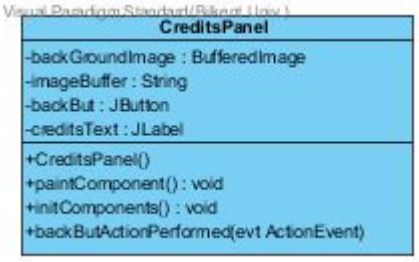
private javax.swing.JButton backBut : In HighScorePanel, a JButton instance is called and it's modified to a button with "to back page" functionality

private javax.swing.JButton controlBut : In HighScorePanel, a JButton instance is called and it's modified to a button with "go to control panel" functionality

private javax.swing.JButton goalBut : In HighScorePanel, a JButton instance is called and it's modified to a button with "go to goal panel" functionality

private javax.swing.JLabel helpText : In HighScorePanel, a JButton instance is called and it's modified to a button with "to help page" functionality

CreditsPanel Class



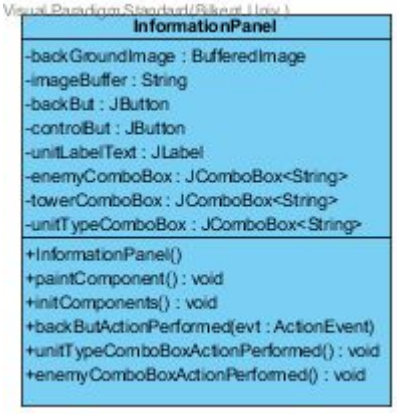
BufferedImage backgroundImage: The image to display in the background of CreditsPanel.

String imageBuffer: This is a String which will be used to print some writings on Credits Panel, specifically developer's names.

private javax.swing.JButton backBut: This is a Button displayed in Credits panel which will be used to go back to main menu.

private javax.swing.JLabel creditsText: This is the panel where String imageBuffer will be printed.

InformationPanel Class

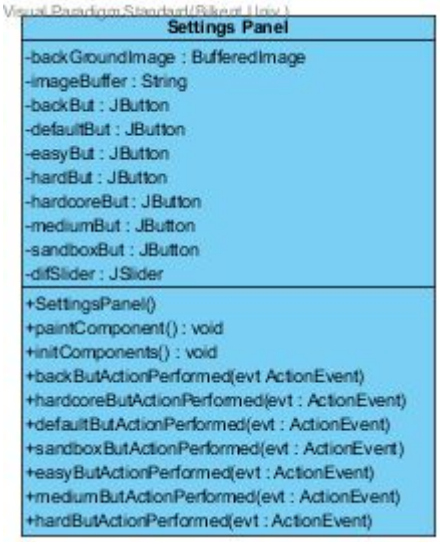


String unitLabel ：This string value notifies the player about the index of panel to be selected.

BufferedImage backgroundImage : The actual image to be taken with String attribute imageBuffer.

String imageBuffer ：This string holds the address of image to be used as backgroudImage.

SettingsPanel Class



BufferedImage backgroundImage: The image to display in the background of SettingsPanel.

String imageBuffer: The directory to read the image to display in the CreditsPanel.

private javax.swing.JButton backBut: In settings, a JButton instance is called and it's modified to a button with "to back page" functionality

private javax.swing.JButton defaultBut: Another JButton instance is called and it's modified to a button which sets difficulty to default.

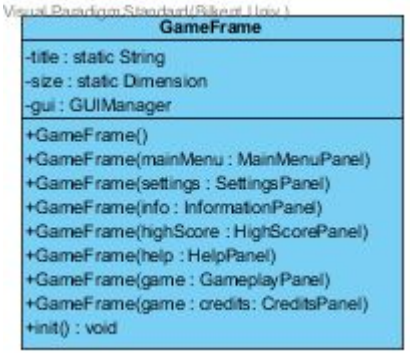
private javax.swing.JSlider difSlider: A JSlider instance is called so that music of the game can be set easily with a slider.

private javax.swing.JButton easyBut: Another JButton instance is called and it's modified to a button which sets difficulty to default.

private javax.swing.JButton hardBut: Another JButton instance is called and it's modified to a button which sets difficulty to hard.

private javax.swing.JButton hardcoreBut: Another JButton instance is called and it's modified to a button which sets difficulty to hardcore, which is the hardest difficulty.
private javax.swing.JButton mediumBut: Another JButton instance is called and it's modified to a button which sets difficulty to medium.
private javax.swing.JButton sandboxBut : Another JButton instance is called and it's modified to a button which sets difficulty to sandbox, which is easiest difficulty.

GameFrame Class



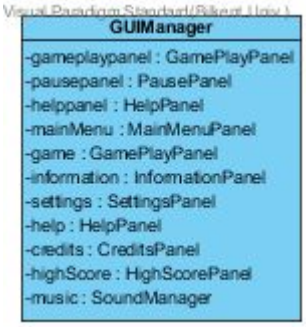
MainMenuPanel mainMenu: The main menu to access other menus and gameplay.
GameplayPanel game: The GameplayPanel where the actual gameplay occurs, accessed via the main menu.
InformationPanel information: This is the Panel which will be used to display information of the game.
SettingsPanel settings: This is the Panel which will be used to display settings of the game.
HelpPanel help: This is the Panel which will be used to display help of the game.
CreditsPanel credits: This is the Panel which will be used to display credits of the game.
HighScorePanel highScore: This is the Panel which will be used to display high scores of the game.
SoundManager music: Game's bakground music is instantiated with this attribute.
static String title: Game's title is set with this variable.
static Dimension size: Gameplay screen's dimensions are set with this attribute.

SoundsManager Class

public InputStream backgroundMusicBuffer : This instance calls an input stream so that music file can be taken and music chosen will be ready to play.

public AudioStream backGroundMusic: Once the input .wav file is received, this attribute will call AudioStream class to start the music.

GUIManagerClass



GameplayPanel game: Instance of GamePlayPanel is created and managed by this class.

InformationPanel information: Instance of InformationPanel is created and managed by this class.

HelpPanel help : Instance of HelpPanel is created and managed by this class.

MainMenuPanel mainMenu : Instance of MainMenuPanel is created and managed by this class.

GamePlayPanel game : Instance of GamePlayPanel is created and managed by this class.

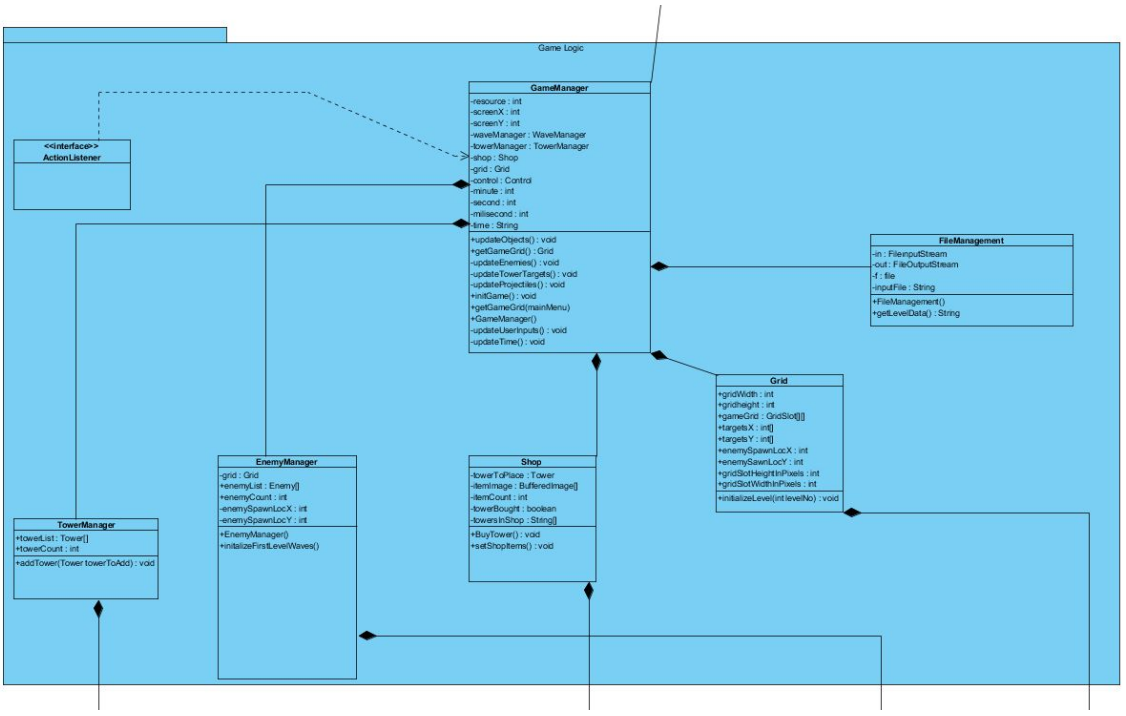
SettingsPanel settings : Instance of SettingsPanel is created and managed by this class.

CreditsPanel credits : Instance of CreditsPanel is created and managed by this class.

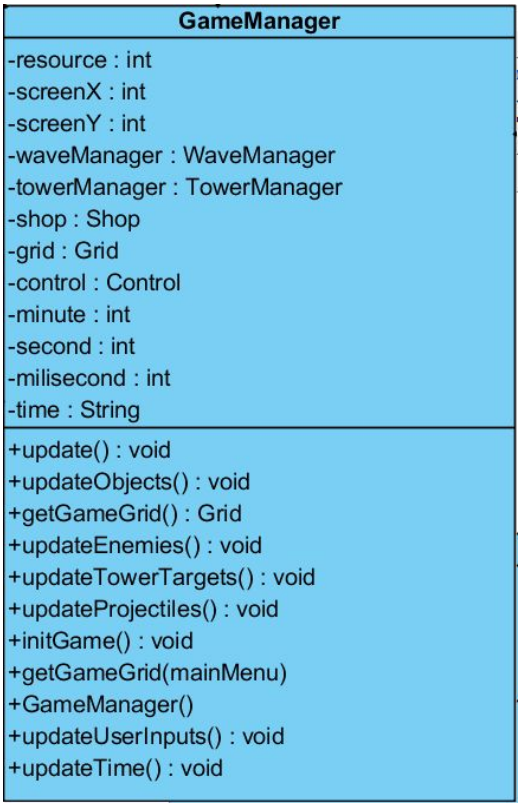
SoundManager music : Instance of SoundManager is created and managed by this class.

HighScorePanel highScore: Instance of HighScorePanel is created and managed by this class.

3.3.2 Game Logic Subsystem



GameManager Class



Architectural Style Description: GameManager is the controller class in the MVC. It updates and notifies the view class GameplayPanel.(View class that draws everything) GameManager also updates and notifies many model classes like towers, shops, enemies etc.

Attributes

private int playerGold : Gold integer attribute indicates how many towers the player can buy. It's default value is 300 as been set in the constructor of GameManager class. ıt decreases when player buys new tower

private int screenX : This integer attribute is used to set the width of game window. The information for calculation of the width pixels is taken from the Grid class, which contains the exclusive attributes of *gridWidth* and *gridSlotWidthInPixels.*

private int screenY : Being counterpart of *screenX,* this attribute is for setting the height of game window. The information for *screenY* is also taken from Grid class.

private WaveManager waveManager : There is a single instance of WaveManager. Adopting Singleton pattern, waveManager organizes the type and frequency of individual enemies and keeps them in structure of wave arrays.

private TowerManager towerManager : towerManager is another singleton class instance. It manages the functonalities of built towers, and new towers are added to the array in towerManager class.

private Shop shop : This attribute contains the functionality of an interface that shows the icon of the tower types. By clicking these tower icons with the help of *control* class, player can choose the tower to buy with mouse.

private Grid grid : *grid* attribute is defined for the game map, and contains a two dimensional array of *gridSlot*, which is the smallest particle of *grid* and has two inherited classes called EnemyGrid, and TowerGrid. Input files that contain information for different map patterns can command *grid* to generate different maps.

private Control control : control, which is an instance of control class which extends the MouseAdapter. It is modified for the MTDefense game specifically and it receives mouse events.

private int minute : This is an integer attribute to hold the in-game time as minute.

private int second : Another integer to hold the in-game time as second.

private int milisecond : Yet another integer to hold the in-game time as milisecond.

private String time : This attribute is for demonstrating how much time passed since the user started gameplay. It gives time as minute, second, and millisecond elapsed.

Methods

public GameManager() : Being the constructor of this class, *GameManager* serves as the game engine, which performs main functionality of game. It sets time attributes to initial 0, sets player gold to 300, defines shop, control, grid, enemyManager, and towerManage attributes as new, sets the screenX and screenY with the values retrieved from *grid* class, and initiates updateObjects method as the last step.

public void updateObjects() : This method ensures that game objects and their functionalities are updated frequently. There is a *Timer* called as an instance of swing library that ensures the delay of objects with proper frequency. A local delay attribute is set to 100 so that game is updated approximately 10 times per second, which is the proper value. *updateTime, updateEnemies, updateTowerTargets, updateUserInputs, updateProjectiles* methods are callen one by one so that these methods will consequently and correspondingly be updated.

private void updateTime() : This method has the functionality of an ordinary chronometer. It updates time as milliseconds. Milliseconds are later converted to seconds, and minutes are later converted to minutes. As last step, minute and second values are shown as time.

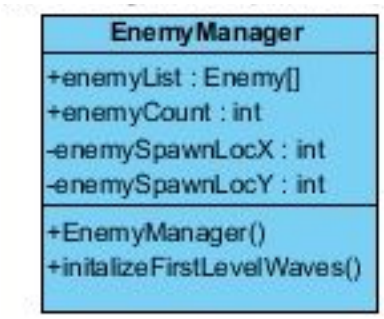
private void updateEnemies() : This update method is responsible of ensuring that enemies are moving around the game map. *move* method in *Enemy* class is callen so that each Enemy in *enemyManager*'s *enemyList* array changes location.

private void updateTowerTargets() : Target of each tower in *towerManager*'s *towerList* array is updated with this function. If an enemy is in range of tower, and if tower doesn't have any target at that time, tower's *setTarget* function will be callen so it will activate tower's shooting functionality as result. Else if tower does have target or enemy is out of range, tower will not be activated and target will be cleared from queue.

private void updateProjectiles() throws Throwable : Projectiles are updated with update function of projectile which is callen by Tower class and for each tower in *towerManager* class' *towerList* array. For each tower, there is a default arrayList of projectiles is defined. Until enemy dies, new projectile is defined in the array list. Once the instantiated and thrown projectile collides with the enemy, projectile is removed from the array list and new projectile is added to the list.

private void updateUserInputs() : This function allows user to interact with game map and shop menu. If the mouse points and clicks to one of the tower icons and if the player has sufficient amount of money for the pointed icon, *buyTower()* function will be activated, which allows player to place towers to a certain location specified as *TowerGrid*. The bought tower will be added to *towerList* array in *TowerManager* class.

EnemyManager Class



Attributes

public Enemy[] enemyList: The list of enemies that are on the map. The controller class GameManager has an instance to this class to update the enemyList.

public int enemyCount: The number of enemies currently on the map.

int enemySpawnLocX: The X location of where the enemies spawn on the grid.

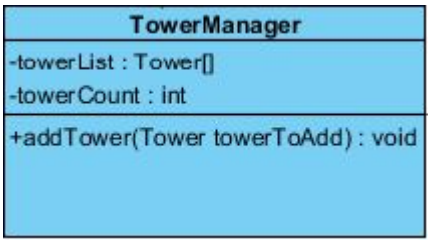
int enemySpawnLocY: The Y location of where the enemies spawn on the grid.

Methods

public EnemyManager(int levelNo, int gridSizeX, int gridSizeY):

public void initializeLevelWaves(int levelNo): Spawns different types of enemies depending on the integer levelNo, passing the spawned enemies the attributes of this class enemySpawnLocX and enemySpawnLocY.

TowerManager Class



Attributes

public Tower[ ] towerList : An array of towers defined in this class so that new towers can be added and they can be managed through the singleton class of TowerManager which's called in gameManager. The array can be used to validate targets, and control projectiles thrown by the array elements as towers.

public int towerCount : Each time a new tower is created in map and added to the *towerList* array, towerCount is increased. This attribute can be used for traversing through the elements of *towerList* array and make various checks in GameManager class.

Methods

public TowerManager() : TowerManager is initated in the GameManager as a single instance. Once initiated, the *towerList* will be instantiated beside it as well with a predefined size.

public void addTower(Tower towerToAdd) : When player buys new towers, this function will be called and a new instance of tower will be added to the array of *towerList.* Attribute of *towerCount* will also be increased by 1.

Grid Class

Attributes

public GridSlot[][] gameGrid: The grid is basically an array of gridSlots on which the player can build towers on, and some slots are unbuildable, they are just pathways for enemies to cross. This variable holds the gridslots on the map. The size of this array is gridWidth*gridHeight.

public int gridSlotWidthInPixels: The width of a grid slot in pixels, i.e. i.e. how much space a single grid slot takes on the screens X axis.

public int gridSlotHeightInPixels: The height of a grid slot in pixels, i.e. how much space a single grid slot takes on the screens Y axis.

public int gridWidth: The length of the gridSlot[][] array's width.

public int gridHeight: The length of the gridSlot[][] array's height.

int enemySpawnLocX: Where the enemies are spawned on the map, in X axis.

int enemyspawnLocY: Where the enemies are spawned on the map, in Y axis.

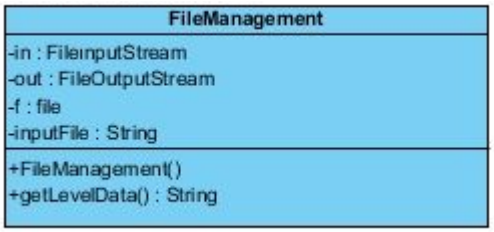
public int[] targetsX: The X axis locations of the targets on this grid. The enemies are designed to move to these targets.

public int[] targetsY: The X axis locations of the targets on this grid. The enemies are designed to move to these targets.

Methods

initializeLevel(int level): It was our decision to make the map composed of gridSlots. This way we can create new levels and get them working just with a few lines of code and a few images.

FileManagement Class



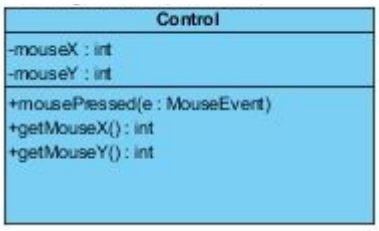
Attributes

private FileInputStream in: This instance is for taking a file, which's address is defined via inputFile String as an input.
private FileOutputStream out: It's used for writing data to a speified file instance.

Methods

public FileManagement(): This class' constructor is called in the gamemanager. Specifically, FileManagement will be used for getting the map credentials from input file.
public String getLevelData(): Specifications of a level will be stored as String with this attribute.

Control Class



Attributes

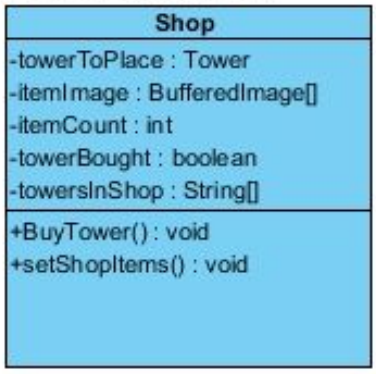
private int mouseX: This integer attribute will be specifying mouse pointer's x location. Will be used in the mousePressed method.
private int mouseY: This integer attribute will be specifying mouse pointer's y location. Will be used in the mousePressed method.

Methods

public void mousePressed(MouseEvent e): With the help of a mouse event listener, this method will handle mouse pointing operations.
public int getMouseX(): This method will retrieve the current x location of mouse pointer.

public int getMouseY(): This method will retrieve the current y location of mouse pointer.

Shop Class



Attributes

private int itemCount : Number of the items in shop are specified with this attribute. At total, it's 8. It's used in the *setShopItems* method, in order to be used for the proper imaging of shop items by setting the true boundary .

private BufferedImage[ ] itemImage : This array is used to behold the images for the icons of different towers. It's initiated in the *setShopItems* method so that icons will be placed in a proper manner.

private Tower towerToPlace : A tower instance is called as attribute so there can be new tower instances generated in game with shop's *buyTower* function.

private boolean towerBought : Initially set as false, this boolean attribute returns true once any condition in *buyTower* function is fulfilled.

private String backgroundImageBuffer : String value which gives the address of the background image to be used for shop menu.

private BufferedImage backgroundImage : When address for background image is obtained, the image will be ready to use for decorating shop menu.

Methods

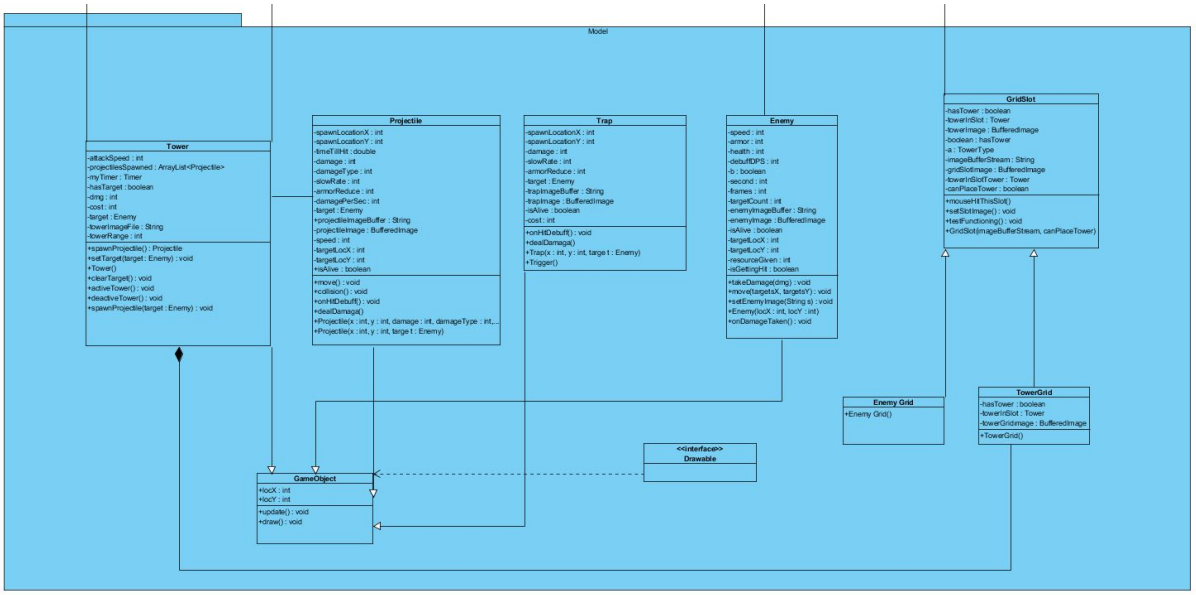
public Shop() : The constructor of shop class initiates the number of items as 8, sets fixed background image, and finally arranges the place of tower icons

public void buyTower(int mouseX, int mouseY, int playerGold) : This function allows player to purchase different types of towers under certain conditions. First condtition checks if mouse icon points to the desired tower icon, and a nested condition checks if player has sufficient money or not. There are 8 checks at total.

public void setShopItems() : This function sets an image for each tower to be bought through the shop menu. These images are stored in itemImage array. There are 8 number of image settings.

public void draw(Graphics g) : Once the tower icons are stored in itemImage array, draw function does the final touch by generating the pixels fittingly for shop menu.

3.3.3 Game Objects Subsystem



Game entity objects were shown in this diagram, and so the mechanisms between these entity objects. Links that come out of the package to the top symbolize the passage to the control objects. Inheritence to the “GameObject” class is the key point to catch in this diagram.

Trap Class

Attributes

private String trapImageBuffer: Location of the image to be buffered must be identified first, and identification if made with String defined as this attribute.

public BufferedImage trapImage: When identified with trapImageBufferImageBuffer, this attribute will allow printing of image for projectile instance.

private int spawnLocationX : With this attribute, x coordinate of the trap instance will be defined. It will be modified with location of tower which hosts to the multitude of projectiles.

private int spawnLocationY : This attribute is the scalar counterpart of spawnLocationX.

private double timeTillHit : This attribute is defined according to the basic calculation of travel speed in physics. Without it, projectile will move in an imbalanced manner and suddenly reach to the target.

private int damage : Each trap has damage and differently inherited projectiles will have different damage attributes.

private int slowRate : For certain trap types, there is a slowRate defined which reduces the movement speed of targetted enemy.

private int armorReduce : There is also attribute for armor reducement defined for specific traps. The projectiles with armor reducement will debuff the enemy, making it more vulnurable to any attack.

private int damagePerSec : This attribute stands for another characteristic debuff element, which is bleed debuff. Targets will lose health for each second if damagePerSec is defined

private Enemy target : For a trap , target means the final destination. Projectiles move according to the location of **target** dynamically, and causes different effects on target such as health loss, or attribute debuff.

private int speed : This attribute defines how speed the trap is. Each characteristic projectile has different speed.

public boolean isAlive : Traps have a life span. When the projectile instance is called with enemy detection, this span starts and lasts until the collision with enemy. *isAlive* attribute returns true when projectile initiated, it returns false when collision occures. According to the value of *isAlive*, projectile is painted on map or not.

Methods

public void onHitDebuff(): According to the debuff values defined for projectile, onHitDebuff composes the debuff effects on the target.

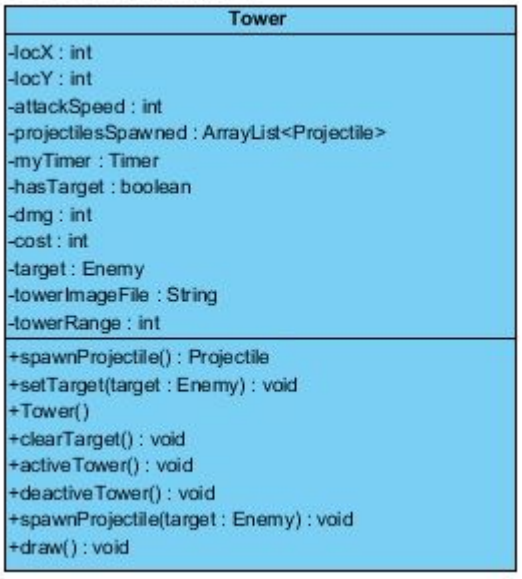
public void update(): This function updates the location of projectile instance. As each time passes, the distance between target's scalar coordinate and projectile's scalar coordinate is decreased. Denoted as speed, this value is finally multiplied with timeTillHit attribute and added to the current location of Projectile so that location update may occur.

public void Trigger(): Serves beneath the update function with the condition of location intervention that's between target and trap. this function notifies the call of onHitDebuff.

public void dealDamage(): Once it's declared with condition that projectile has arrived to target, the target loses health according to the damage attribute of the projectile instance.

public Projectile(int x, int y, Enemy target): One of the two constructors. This constructor only takes the x and y coordinates, and the targetted enemy as three parameters. In addition, projectile's image is set with a throw-catch statement.

Tower Class



Attributes

private Enemy target : Each tower has a specified target and this specification is made according to the tower's range. Once the target is dead, Tower chooses a new target. Target's condition and avaibility in tower's range sets the boolean value of hasTarget.

private ArrayList<Projectile> projectilesSpawned : Towers are constructed with an array list of projectiles so that enemies can be dealt with. Once enemy is spotted, new instances of projectiles are deposited in the projectilesSpawned array list. When enemy is dead, the projectile instances will be unloaded from array list.

private String towerImageFile : There is a string value to be used for defining the address of image to be used for specific instance of tower.

private boolean hasTarget : In default, hasTarget for each tower is defined as false. For it to be true, there must be an enemy in the interval of tower range. The attribute hasTarget is used for a check to update tower targets. If tower already has a target and hasTarget is true, tower can't attack to the upcoming enemies and has to deal with the target at hand first.

private Timer myTimer : Callen from swing library, this instance serves as helper for *activateTower* function. It uses the delay value of 750 as parameter so that frequency will be harmonic with the timing functions set in the other classes of the game.

private int locX : Just like the other game objects, tower class also has the scalar x coordinate.

private int locY : Just like the other game objects, tower class also has the scalar y coordinate.

private int towerRange : Tower range is defined differently for each inherited tower class. Bigger this attribute is, larger the attack radius of a tower.

private int cost : Each different tower type has different cost. This attribute defines the tower's availability to player. If cost of tower exceeds player's money, player can't buy the tower from shop

private int attackSpeed : How fast a tower can attack is defined by this attribute.

private int dmg : Tower's damage output is defined with this attribute alongside the damage output of the projectile initialized by it.

Methods

public Tower() : Once a tower object is called, an arraylist of projectiles is initialized with it. This array list will provide a projectile shoot or reload mechanism to the tower.

public void setTarget(Enemy target) : Towers can check the distance between their target and their own location. If this distance is not greater than or equal to the range issued to the tower, tower will set target and initiate projectile objects for it as setTarget suggests.

public void clearTarget() : When enemy is out of range of the tower, target at hand will be offload from tower's focus. This function's availability is determined by the condition statement in *setTarget* function.

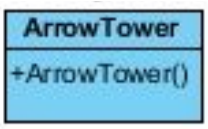
public void activateTower() : Once the target is set with *setTarget* method, tower will spawn projectile upon target at hand with function *spawnProjectile.* This function is called according to the conditioning in *setTarget*. The timer attribute is declared in it with the delay integer parameter of 750 so that game's functioning will not be disturbed.

public void deactivateTower() : This is an auxiliary function to be used in *clearTarget* function. It basically stops the timer, and defined in terms of making the design neater.

public void spawnProjectile(Enemy target) : If *activeTower* function is called, this function will also be available cooperatively. It creates new instances of projectile class to be added to the *projectilesSpawned* array so that tower can consequently aim and shoot the target.

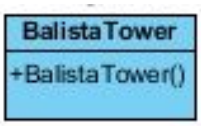
public void draw() : This function draws towers to the game map with corresponding coordinates and according to the image retrieved with address which is stored in. towerImageFile string.

ArrowTower Class



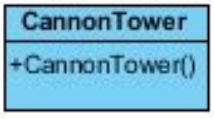
public ArrowTower() : The constructor of this class, which is inherited by Tower class. The proper image and attributes will be specified with the constructor.

BallistaTower Class



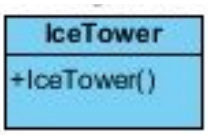
public BallistaTower() : The constructor of this class, which is inherited by Tower class. The proper image and attributes will be specified with the constructor.

CannonTower Class



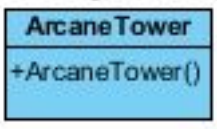
public CannonTower() : The constructor of this class, which is inherited by Tower class. The proper image and attributes will be specified with the constructor.

IceTower Class



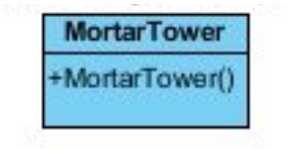
public IceTower() : The constructor of this class, which is inherited by Tower class. The proper image and attributes will be specified with the constructor.

ArcaneTower Class



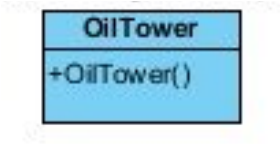
public ArcaneTower() : The constructor of this class, which is inherited by Tower class. The proper image and attributes will be specified with the constructor.

MortarTower Class



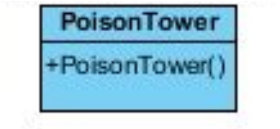
public MortarTower() : The constructor of this class, which is inherited by Tower class. The proper image and attributes will be specified with the constructor.

OilTower Class



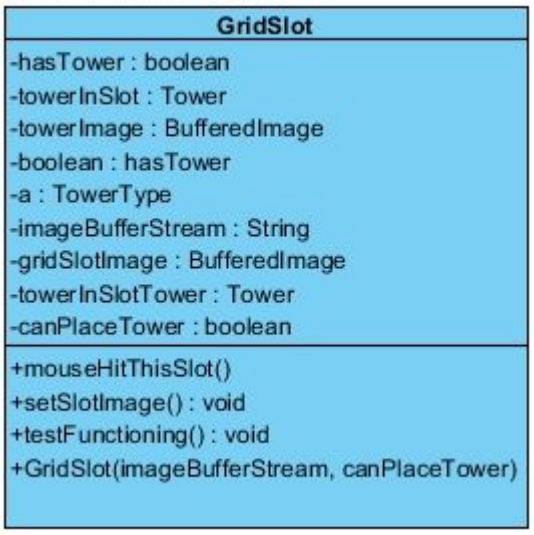
public OilTower() : The constructor of this class, which is inherited by Tower class. The proper image and attributes will be specified with the constructor.

PoisonTower Class



public PoisonTower() : The constructor of this class, which is inherited by Tower class. The proper image and attributes will be specified with the constructor.

GridSlot Class



Attributes

public boolean canPlaceTower: Whether the player can place a tower on this gridSlot. If this boolean is false, then this grid is for enemies to move across.

public boolean hasTower: Whether this gridSlot has a tower that's been built by the player.

String towerGridBufferStream: This string holds the image location(system directory) of this gridSlot.

String towerImageBufferStream: This string holds the image location(system directory) of the tower on this gridSlot.

public BufferedImage towerImage: The image of the tower on this gridSlot that's read from the bufferStream.

public BufferedImage gridSlotImage: The image of this gridSlot that's read from the bufferStream.

private Tower towerInThisSlot: The tower class instance that's built on this gridSlot.

Methods

public GridSlot(String imageBufferStream, boolean canPlaceTower): Sets whether this slot is for the enemies to move across or for the player to build a tower on.(i.e. canPlaceTower boolean) Also sets the system directory String to read the image file for this gridSlot.(Grass.png, Rock.png etc.)

public boolean mouseHitThisSlot(boolean isInBuyMode, Tower towerToPlace, int x, int y): Whenever the clicks on this gridSlot, an action is determined based on the current state of the attributes of this class. For example if the player is in buy mode, and canPlaceTower is true, and hasTower is false, a tower is placed on this gridSlot.

public void setSlotImage(String s): Set the directory to read the image file to display this gridSlot.

EnemyGrid Class

public EnemyGrid(): When constructed, TowerGrid will call super class and set canPlaceTower to be true so that player can place tower to the grid specified.

TowerGrid Class

public TowerGrid(): When constructed, TowerGrid will call super class and set canPlaceTower to be true so that player can place tower to the grid specified.

Enemy Class

Attributes

private ImageIcon impactImageIcon: This variable is to hold the impact effect image sequence on the enemy. The impact effect happens after a projectile hits this certain enemy. There are many ways to hold an image sequence, so the type of this variable can be changed easily if seen fit. This variable is changed by the projectile on onDamageTaken() function so that only 1 effect is active at a time on a particular enemy instance. This is a design decision to make the game look more readable. This attribute is only used when isGettingHit boolean is true.

private String enemyImageBuffer: This variable holds where the image of this enemy instance exists in the system explorer. The file at this location is read to get the image.

private BufferedImage enemyImage: The image of this enemy. “enemyImageBuffer” variable is read as an image and that image is assigned to this variable.

private int armor: The armor of an enemy instance, used when calculating the damage taken per hit.

private int health: The health points of an enemy instance. isAlive is set to false when the health goes below 0. And any references to this object is deleted so Java's garbage collector can destroy this enemy instance from the memory.

private int resourceGiven: Resource given to the player when this enemy instance is killed.

private int locX: The X location of this enemy instance on the screen(the Grid).

private int locY: The Y location of this enemy instance on the screen(the Grid).

private int targetLocX: The X location of where this enemy instance is moving to on the map at the current time.

private int targetLocY: The Y location of where this enemy instance is moving to on the map at the current time.

private int speed: The movement speed of an enemy instance, used by the move() and update() functions.

Methods

public Enemy(int locX, int locY): Constructor determining where the enemy is spawned on the map as locX and locY locations.

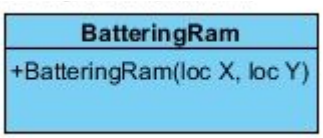
public void setEnemyImage(String s): Sets the location(system directory) of the image of this enemy instance.

public boolean isGettingHit(): Whether this enemy is getting hit at the current time. The damage is dealt when a projectile hits this enemy instance. But the isGettingHit boolean is turned false when the projectile impact animation is over.

public void move(int[] targetsX, int[] targetsY): The function to move the enemy to it's next target. When a target is reached, targetCount is incremented and this enemy instance starts moving to the next target.

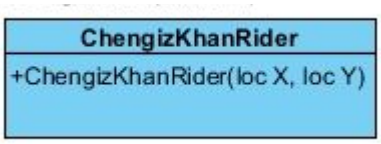
public void onDamageTaken(int dmg): When a projectile hits this unit and calls dealDamage() which calls this function on the enemy instance, this function decrements the health by the damage amount, determines if isAlive is still true after the damage, and starts the onHitImpact animation.

BatteringRam Class



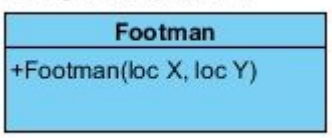
public BatteringRam(int locX, int locY): The constructor of this class, which is inherited by Enemy class. The proper image and attributes will be specified with the constructor.

ChengizKhanRider Class



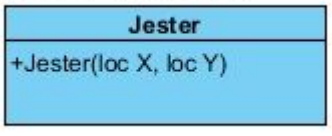
public ChengizKhanRider(int locX, int locY): The constructor of this class, which is inherited by Enemy class. The proper image and attributes will be specified with the constructor.

Footman Class



public Footman(int locX, int locY): The constructor of this class, which is inherited by Enemy class. The proper image and attributes will be specified with the constructor.

Jester Class



public Jester(int locX, int locY): The constructor of this class, which is inherited by Enemy class. The proper image and attributes will be specified with the constructor.

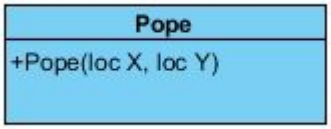
Knight Class

public Knight(int locX, int locY): The constructor of this class, which is inherited by Enemy class. The proper image and attributes will be specified with the constructor.

LightCavalry Class

public LightCavalry(int locX, int locY): The constructor of this class, which is inherited by Enemy class. The proper image and attributes will be specified with the constructor.

Pope Class



public Pope(int locX, int locY): The constructor of this class, which is inherited by Enemy class. The proper image and attributes will be specified with the constructor.

SaintJohnsKnight Class

public SaintJohnsKnight(int locX, int locY): The constructor of this class, which is inherited by Enemy class. The proper image and attributes will be specified with the constructor.

TrojanHorse Class

Projectile Class

Attributes

private String projectileImageBuffer : Location of the image to be buffered must be identified first, and identification if made with String defined as this attribute.

public BufferedImage projectileImage : When identified with projectImageBuffer, this attribute will allow printing of image for projectile instance.

private int spawnLocationX : With this attribute, x coordinate of the projectile instance will be defined. It will be modified with location of tower which hosts to the multitude of projectiles.

private int spawnLocationY : This attribute is the scalar counterpart of spawnLocationX.

private int locX : Defined for general purpose, this attribute is mainly used for the update function of Projectile class. Update function ensures that projectile moves in right direction.

private int locY : Another attribute to be used as the scalar counterpart. Same as locX, this is used mainly for *update* function.

private double timeTillHit : This attribute is defined according to the basic calculation of travel speed in physics. Without it, projectile will move in an imbalanced manner and suddenly reach to the target.

private int damage : Each projectile has damage and differently inherited projectiles will have different damage attributes.

private int damageType : Denoted a an integer value, this attribute defined the kind of damage that projectile does. It can be ice, fire, magical, or bleeding.

private int slowRate : For certain projectile types, there is a slowRate defined which reduces the movement speed of targetted enemy.

private int armorReduce : There is also attribute for armor reducement defined for specific projectiles. The projectiles with armor reducement will debuff the enemy, making it more vulnurable to any attack.

private int damagePerSec : This attribute stands for another characteristic debuff element, which is bleed debuff. Targets will lose health for each second if damagePerSec is defined

private Enemy target : For a projectile, target means the final destination. Projectiles move according to the location of **target** dynamically, and causes different effects on target such as health loss, or attribute debuff.

private int speed : This attribute defines how speed the projectile is. Each characteristic projectile has different speed.

private int targetLocX : Target enemy of the projectile has an location of x coordinate and targetLocX allows us to dentify this location.

private int targetLocY : Target enemy of the projectile also has location of y coordinate and targetLocY allows us to dentify this location.

public boolean isAlive : Projectiles have a life span. When the projectile instance is called with enemy detection, this span starts and lasts until the collision with enemy. *isAlive* attribute returns true when projectile initiated, it returns false when collision occures. According to the value of *isAlive*, projectile is painted on map or not.

Methods

public void onHitDebuff(): According to the debuff values defined for projectile, onHitDebuff composes the debuff effects on the target.

public void update(): This function updates the location of projectile instance. As each time passes, the distance between target's scalar coordinate and projectile's scalar coordinate is decreased. Denoted as speed, this value is finally multiplied with timeTillHit attribute and added to the current location of Projectile so that location update may occur.

public void collision(): Serves beneath the update function with the condition of location intervention that's between target and projectile. this function notifies the call of onHitDebuff.

public void dealDamage(): Once it's declared with condition that projectile has arrived to target, the target loses health according to the damage attribute of the projectile instance.

public Projectile(int x, int y, Enemy target): One of the two constructors. This constructor only takes the x and y coordinates, and the targetted enemy as three parameters. In addition, projectile's image is set with a throw-catch statement.

public Projectile(int x,int y,int damage,int damageType,int slowRate,int armorReduce,int damagePerSec,Enemy target, String projectileImageBuffer): Last constructor to remain. This projectile constructor ultimately defines all the attributes of a certain projectile element. Just like the other constructor, it sets the image for projectile instance.
public TrojanHorse(int locX, int locY): The constructor of this class, which is inherited by Enemy class. The proper image and attributes will be specified with the constructor.

4. Improvement Summary

In the first iteration, there were no diagrams for architectural patterns planned. In the current iteration, the main functionality is demonstrated with two diagrams. There is another component diagram implemented with "lollipop" convention. These is component diagram that shows the relationships in high level scheme as three layers. And deployment diagram that shows hardware-software mapping. With new classes added to the code, their functionality is integrated to the existing functionality. Control structures were changed. For instance, main control mechanism is now a class called "GameManager". "GUIManager" classes' functionality is improved, and now design at hand is more likely to follow the conventions of Object-oriented software engineering. A new class "gameObject" implementing drawable interface is defined and Enemy, Projectile, Tower, and Grid were redefined under the GameObject class, being inherited by it. A new game object "traps" defined just as promised. Traps are planned to be like consumable objects, and new traps shall be once collision occures. In the updated iteration, functionings of **singleton** and **façade patterns** were extended and new classes façade and singleton classes were defined in the control hierarchy. In addition, **abstract factory design pattern** is used for game objects. Factory will be helpful to create new instances of these objects.