Bilkent University

Department of Computer Engineering

CS-319 Object Oriented Software Engineering Project

Sword & Shield: A Space Adventure

System Design Report

Iteration I

Group 2D

Akın Berkay Bal

Eren Aslantürk

Mehmet Enes Keleş

Sadık Said Kasap

Table of Contents

1. Introduction

We are designing and implementing a game called Sword and Shield. It is a two dimensional top-down strategy/tower defense game inspired from Bloons: Tower Defense. The game will be playable by two players: the attacker and the defender. The map will consist of a base for attack units to spawn, a lane for them to walk and a lot of space for defense units to be located. There will be offensive and defensive units to be used during two different stages of the game. At the first stage, defensive player will buy and put his desired items on the map. During the second stage, attacker player will try to breach through the end of the map by spawning attack units in a limited time.

1.1 Purpose of the System

Sword & Shield: A Space Adventure is a 2 player top-down strategy/tower defense game that mainly inspired from Bloons: Tower Defense but has a different gameplay as there is one attacker and one defender instead of an AI attacker. As this is a 2 player game there are some design issues which we needed to come over. Most important one of them is how game flow will happen. Also there are a lot of dynamic elements like projectiles and spaceships in our game therefore it is possible to have performance and synchronization issues if it is not designed properly. It is not an impossible task but it is fairly challenging for us.

1.2 Design Goals

One of the most important parts of creating a system is design. Therefore design goals should be focused carefully. We have non-functional requirements of the system in our previous report but we aim to clarify them better.

Reliability

We want our game to be reliable, free from critical bugs and elements that can interfere with the game. It should not crash unless manipulated with an external tool.

Modifiability

The system should be modifiable. It means we should be able to change assets of the game or add units without redesigning the whole project from scratch. As we are using object oriented design we are expecting not to crash into such an issue.

Ease of Use

Game should be straightforward so that people don't need to bother with tutorials or reading a lot of stuff before being able to play, but rather they should be able to start and play immediately.

Maintainability

Even the billion dollar projects may have bugs, but important thing is that they should not be critical bugs and do not interfere with user experience, and most importantly they should be fixed quickly. Code is well divided to parts therefore it should be maintainable easily for future problems and bugs.

Responsiveness

We are planning our game to be 60 fps(frame per second), therefore it should be pretty responsive to changes in the screen. We are confident that it can be achieved.

Adaptability

As we are using Java to write code of the game, as long as people have Java Runtime Environment in their devices they will be able to play the game with ease.

1.2.1 Trade-Offs

Ease of Use vs Functionality

As we decided to give a more straightforward experience to the users we tried to make the game as lean as possible. We will give the player ease of use but to provide that we evaded adding too much features to the game.

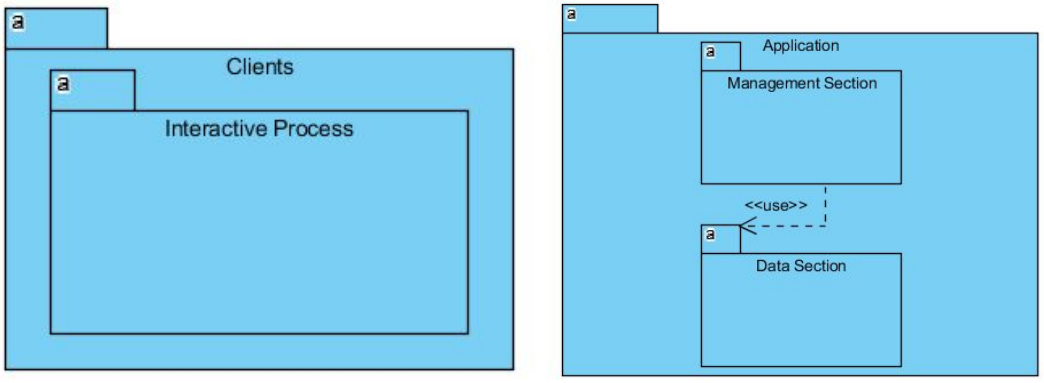
2. Software Architecture

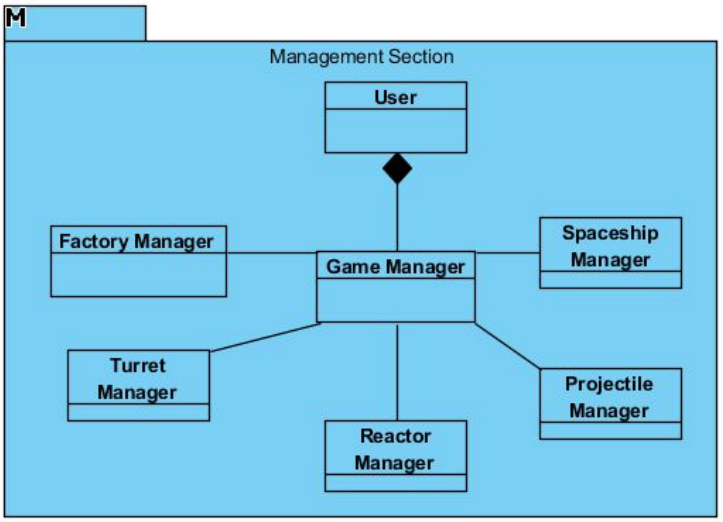
2.1 Overview

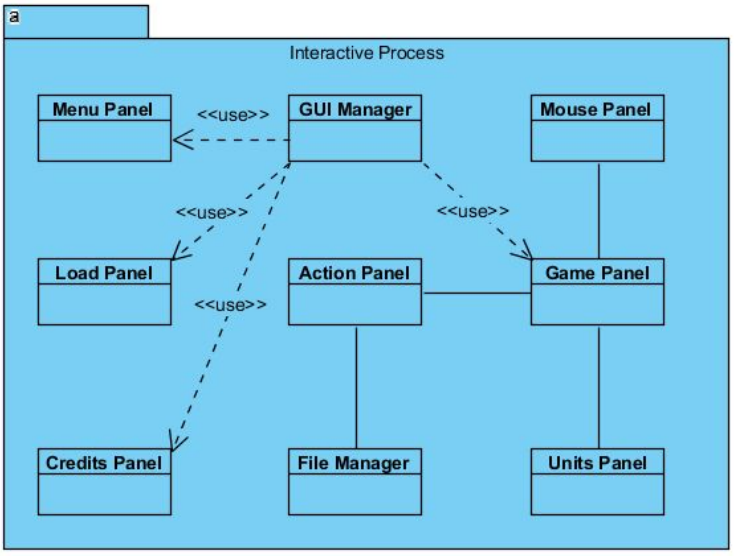
These sections below give details about software structure of our project application. Software system of the project depends on a trilayer format and it can be run on any kind of laptop or desktop. The requirements are a screen, a mouse and optimally a windows operating system.

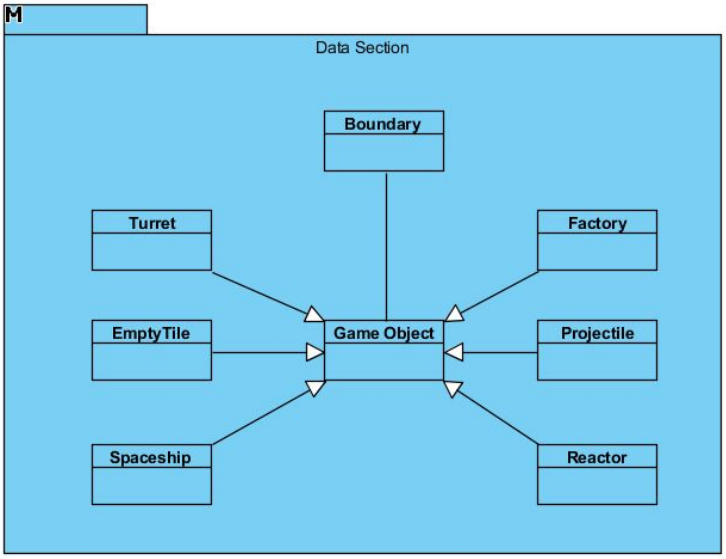
2.2 Subsystem Decomposition

The system of application based on trilayer format as mentioned. In this format, system interacts with client, updates status of the process and initiates system according to user. User is the obligatory element in the system as the activator of the application, an accessor to frames in the application and especially the visual game section. User is made of 2 essential parts as the attacker and the defender. With interactions of both of users, main game section progresses and application proceeds the situation and keeps datas need to be stored or gets rid of the unnecessary ones. Detailed visualization of the decomposition is as shown in the graphs below.




M
Management Section
User
Factory Manager
Game Manager
Spaceship Manager
Turret Manager
Projectile Manager
Reactor Manager


a
Interactive Process
Menu Panel
<<use>>
GUI Manager
Mouse Panel
<<use>>
<<use>>
Load Panel
Action Panel
Game Panel
<<use>>
Credits Panel
File Manager
Units Panel


M
Data Section
Boundary
Turret
Factory
EmptyTile
Game Object
Projectile
Spaceship
Reactor

2.3 Hardware/Software Mapping

Our project, Sword and Shield: A Space Adventure is a game based on java programming language. In java, the game uses file.io to interact with file type documents available in the project directory, backup the data from and reuse it in a local storage, audio libraries for song related features, java utilities for general purpose and the application's operational classes, awt, swing and applet for user interface focused classes. To be able to use java language, it is an obligatory to have the most recent version of JDK for best quality.

The game uses mouse hardware system input equipments. From traveling through in-app windows to interacting with game interface window such as purchasing turrets, playing/pausing the progress, placing entities onto the game field.

As the data of app able to backup, current status of the game such as scores, balance of both attacker and defender, and the last build up status of end users. These data will be backed up via text file available on the project file. The program will be able to read back the data from the text file to saved game progress on users' own will.

2.4 Persistent Data Management

The application aims to focus on simplicity and fertility of both design and data usage. According to this, we implement class relations and interactions onto optimality to make its readability and usability qualified as well as to make it client-programmer friendly. While sound files are also in wav format, icons and textures are in jpeg /png/gif format via game management; they are called back from the app through user interaction via gui.

2.5 Access Control and Security

For a secure and a trustable game experience, it is suggested to have a trustable antivirus and malware applications installed in the users' computer and allow firewall to use datas freely not to interrupt data processing. The application has no its own anti-malicious system and it is in user agreement before installing the game for precaution. Because the application uses local storage, user information backed up will be computer-wise. If users wishes to use same progress datas on another computer, they are available in the game file as the majority of games have.

2.6 Global Software Control

All classes and interactions must be synchronous with each other and toward user. There should be an arranged latency and caller shouldn't block other classes' calls. Therefore we have managers to keep classes in order and cut-and-dried bounds in order to prevent possible bugs and unexpected result as well as readability of code in simplicity.

2.7 Boundary Conditions

In order to install the game application, it is enough to download game files from the website it is available and unzip onto the desired location. The next step is autorunning the exe file located in the game directory. It is suggested to have a shortcut for an easy access. The last and most important step is enjoying the game, suggested by 9 of 10 game developers.

For possible bug and crashes issues, it is enough to closing the app and opening again. The users can continue from most recent backed up game progress. In case, there is an issue with sound playing. It just requires to check whether the sound file is in the sound files directory (in case user mistakenly moves them while installing or wants to store it to somewhere with their own will).

3. Subsystem Services

This section provides information about the design patterns used in our design and detailed structure of our subsystems.

3.1. Design Patterns

3.1.1 Trilayer Architectural Design

Layering is commonly used in many modern systems such as TCP/IP, Operating Systems, subway networks in big cities etc. Inspired from these examples we separated our design into three functional components. These components are called GUI Subsystem, Logic Subsystem and Entity Subsystem. Our subsystems communicate each other in many cases but they do not share any functionality. This compact design of components prevents confusion and

reduces the complexity of developing our software while offering smooth and fast user experience.

3.1.2 Façade Pattern

Façade Pattern contains a class that works as an interface for the user by providing suitable methods to manipulate all of the subsystem. This kind of approach hides complexities of the underlying design and makes the system more understandable and user-friendly. In order to make our code more organized and hence maintainable we chose to use this approach in our design.

Our GUI Subsystem and Logic Subsystem designs have Façade Pattern. GUIManager and GameManager classes consist of methods that abstract other subsystem classes while providing solid usability.

3.1.3 Singleton Pattern

Singleton pattern makes sure there is only one instance of a class at a time while creating a new instance of that class. This design pattern will be widely used in GUI and Logic subsystems to make our code more reliable. For most of the classes in these two subsystems, we don't need more than one instance running at a time. Opposite situation can give birth to many mistakes.

3.2 GUI Subsystem

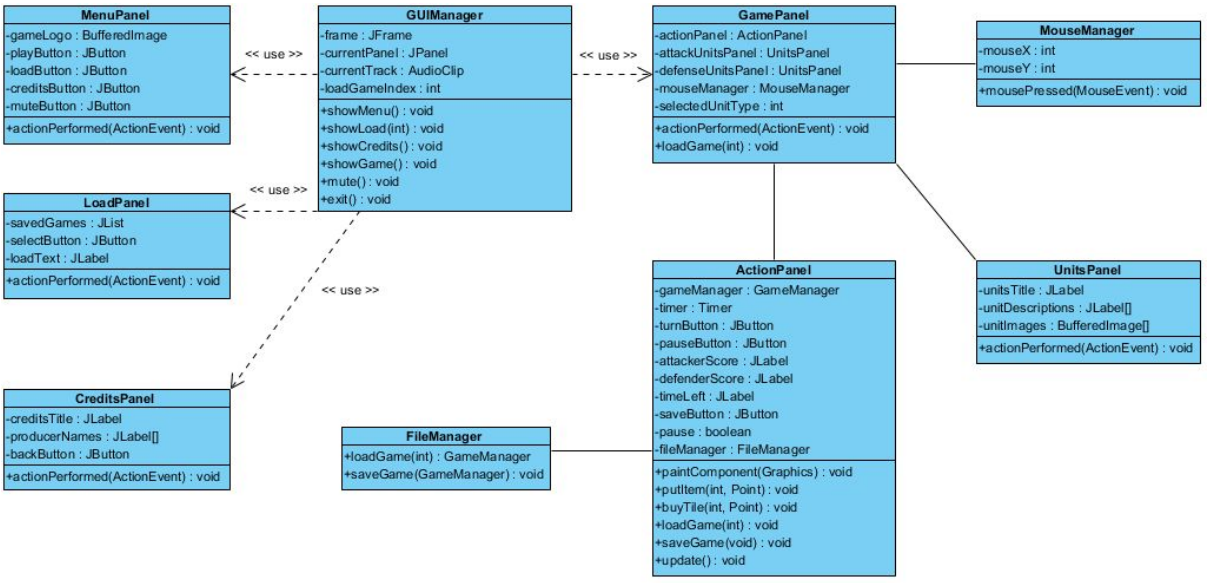
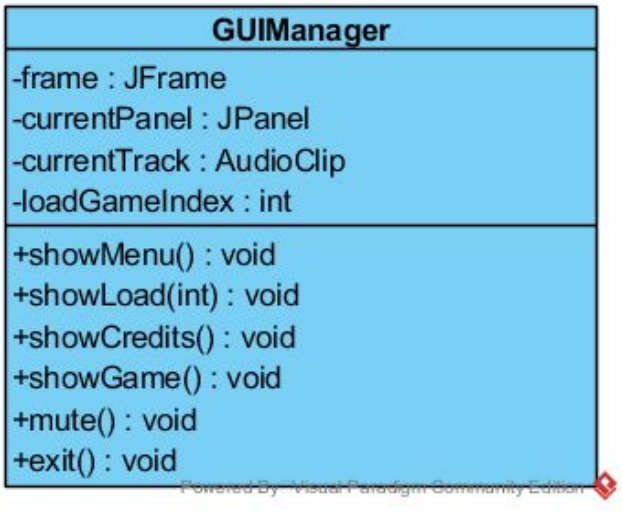


Figure 3.2.1 (Overall GUI Subsystem)

GUI Subsystem shows the user the state of the game and also lets user interact with program. It is the top layer of the program which has the elements that user can interact. All in game actions will be showed in this layer

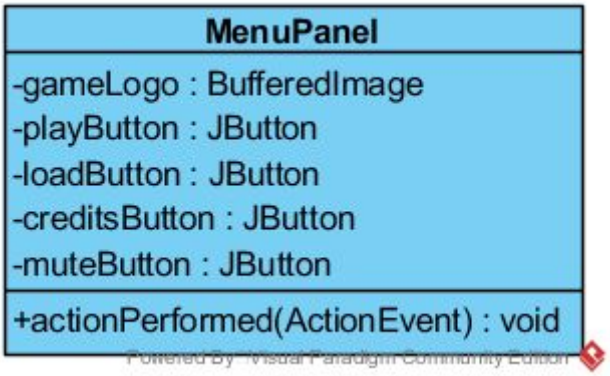
GUIManager



It has a frame that will show the panels. Also it has “currentPanel” which will show the panel that is being used at that moment. It has a currentTrack object for music. Also “loadGameIndex” for loading the game.

Its methods are all void. showMenu() changes panel to MenuPanel, “showLoad()” changes panel to “LoadPanel”, “showCredits()” changes panel to “CreditsPanel”. “showGame()” goes to the GamePanel. It has an actionPerformed(ActionEvent) method that is void to detect inputs.

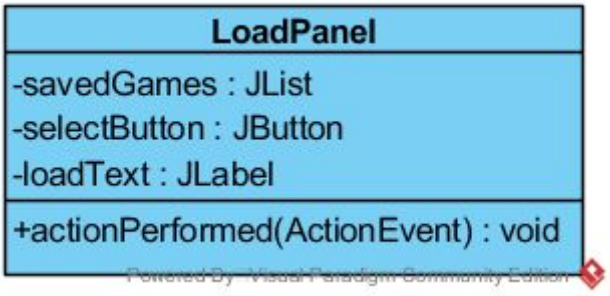
MenuPanel



It has gameLogo which is a BufferedImage. It has a playButton which takes you to GamePanel, loadButton which takes you to LoadPanel, creditsButton which takes you to CreditsPanel, muteButton which silences all the audio.

It has an actionPerformed(ActionEvent) method that is void to detect inputs.

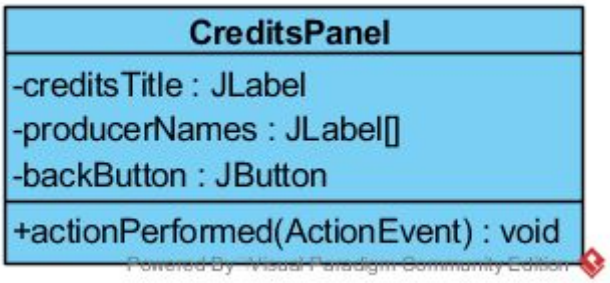
LoadPanel



LoadPanel has a savedGames which is a list that contains the games that are saved, a button which is named selectButton to select the game that desired to be loaded and a ”loadText“ which is a label that is written “Load”.

It has an “actionPerformed(ActionEvent)” method that is void to detect inputs.

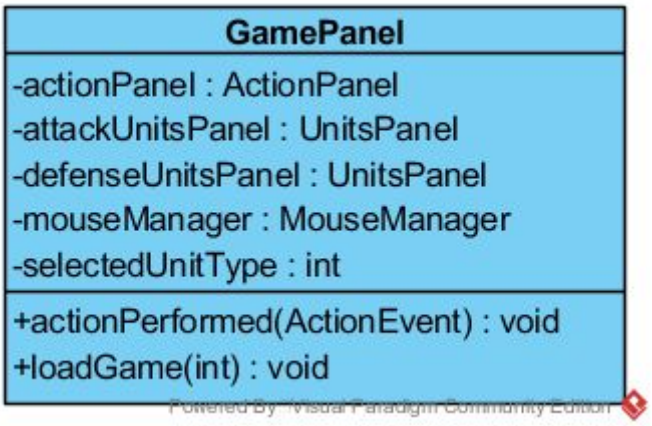
CreditsPanel



CreditsPanel has “creditsTitle” which is a label which shows its title, “producerNames” which is a label array that shows the names of the producers and a “backButton” to go back to the main menu.

It has an “actionPerformed(ActionEvent)” method that is void to detect inputs.

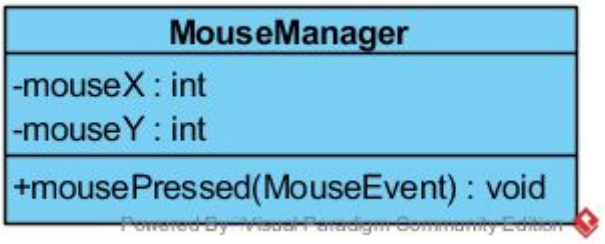
GamePanel



GamePanel has an actionPanel which is an ActionPanel which will be explained after this. It has attackUnitsPanel and defenseUnitsPanel which are UnitsPanel type. They will show the respected units on their side of the panel. It has a mouseManager which is a MouseManager to control the user interaction with the program. It also has a selectedUNitType which is an integer.

It has an actionPerformed(ActionEvent) method that is void to detect inputs. Also it has loadGame(int) which is a void method to load the game with the given integer value.

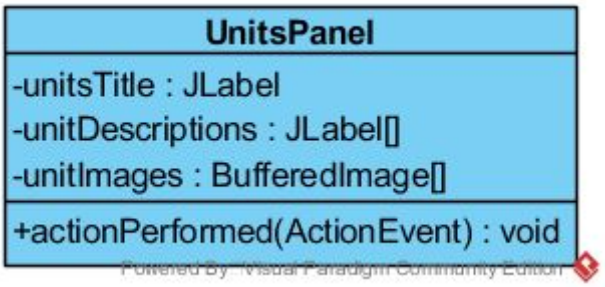
MouseManager



It takes the mouseX and mouseY as integers which represents the Mouse's coordinates on program's panel.

It has mousePressed(MouseEvent) which is a void method to detect mouse inputs into the game.

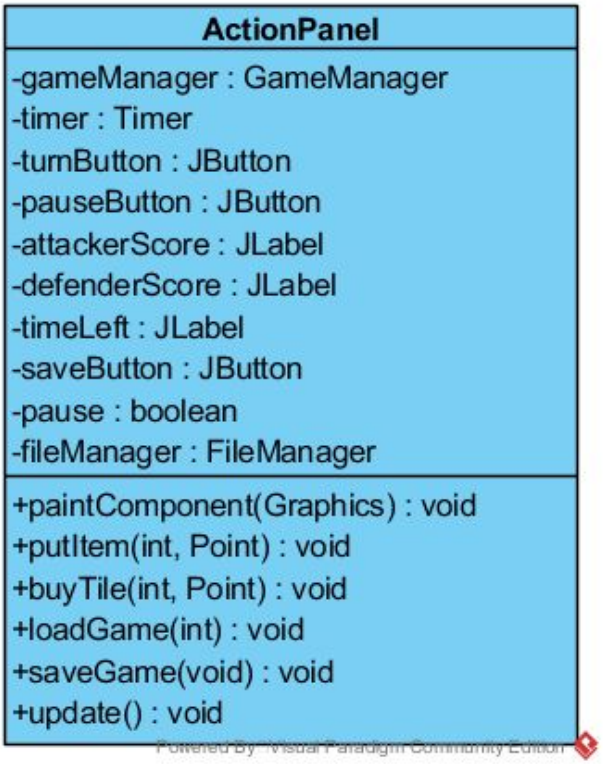
UnitsPanel



It will be used for both attackers and defenders. It has unitsTitle which holds the name of the units, unitDescriptions which a label array that shows the descriptions for the units in a detailed way and unitImages which is BufferedImage type that shows the images of fthe units so that user can see them more clearly while choosing.

It has an actionPerformed(ActionEvent) method that is void to detect inputs.

ActionPanel



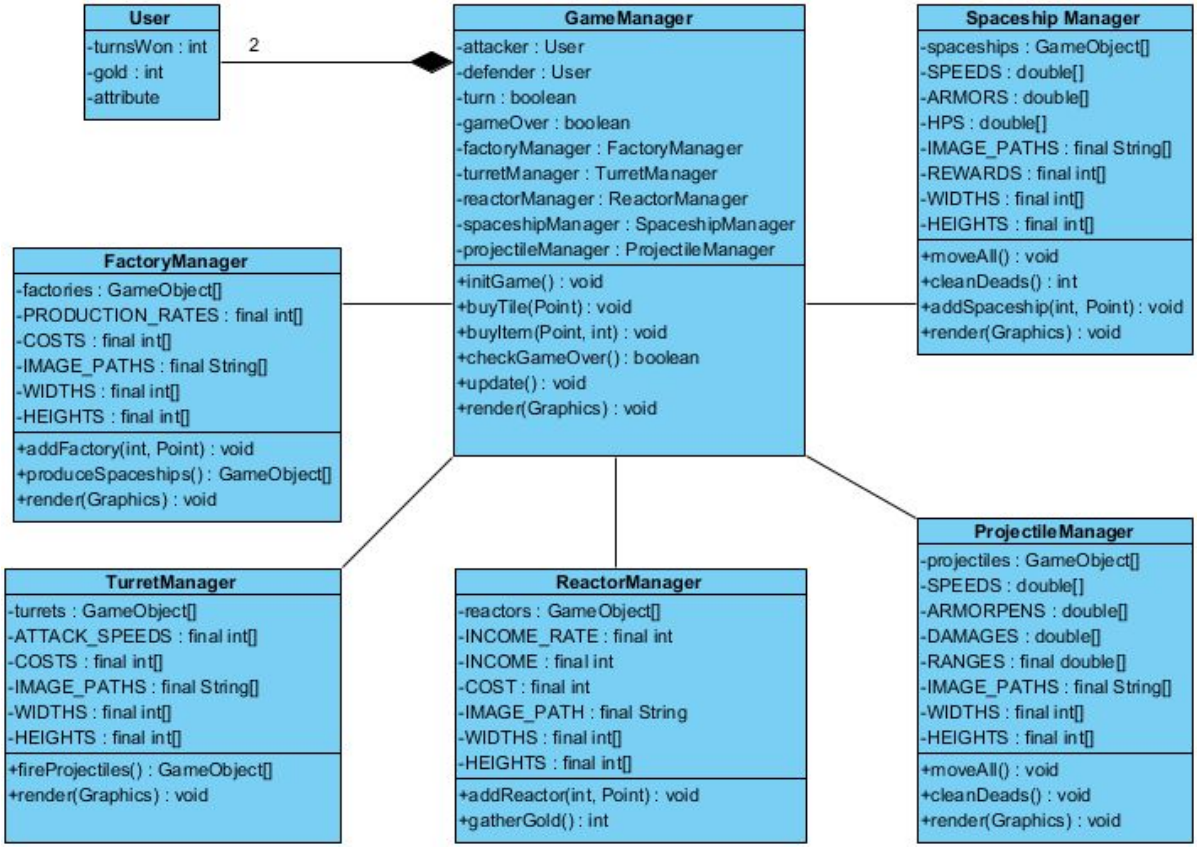
One of the most important panels in here as it has gameManager as a GameManager object, timer that will be used for timing issues and shows the time to use to players. It has turnButton which gives turn to other player before the time runs out, pauseButton that calls pause which is boolean and pauses game until it is clicked again. attackerScore and defenderScore which are labels that shows individual users scores in-game. There is a timeLeft label which works with timer to determine time until the turn ends. There is a saveButton to save the game. Also there is a fileManager, which is a FileManager object to interact with the files.

There is 6 void methods. paintComponent(Graphics) paints the components. putItem(int, Point) is used to put an object to desired place on board. buyTile(int, Point) is used to buy tiles. loadGame(int) loads the game that's integer value is given. saveGame(void) saves the game in its current state. update() used to update all data in the game, including drawings and values.

FileManager

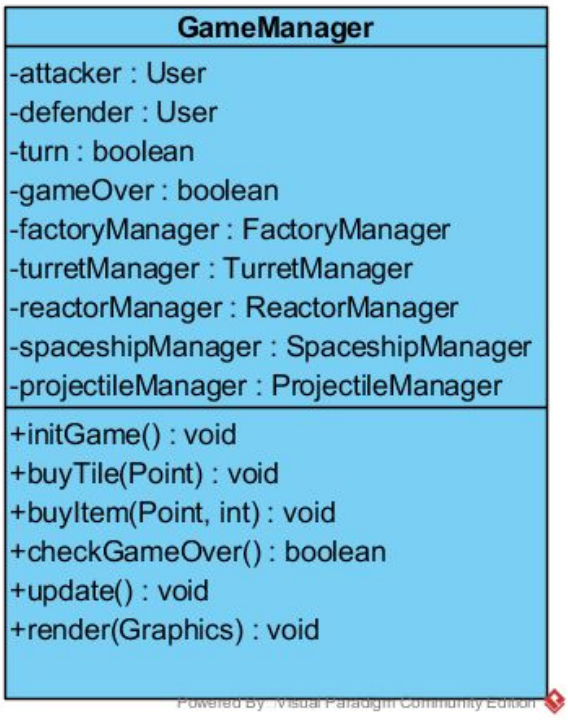
FileManager is used to load and save the game. It has 2 methods. loadGame(int) is a GameManager object that loads the game that's integer value is given. saveGame(GameManager) is a void method that saves the game by taking the data from GameManager.

3.3 Logic Subsystem



Logic Subsystem keeps the current game state. It is also a intermediate layer between GUI Subsystem and Entity Subsystem which provides an interface to the user to play the game. All mechanics in the game are implemented in this layer. Note that every class on this layer except User will use Singleton Pattern.

GameManager

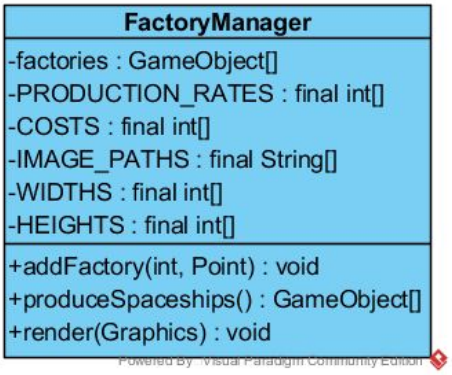


GameManager class will keep two User objects representing the attacker and the defender. It will keep the current turn and a variable named gameOver that tells if the game is over. Also it will contain instances of manager classes of every unit type. initGame method will be used to initially create the required items on the predefined game setup. buyTile method will find the emptyTile object corresponding to the given point and buy it if the user has enough money and the emptyTile is not restricted for that user. buyItem method will take location and type as parameters and will put the item of that type into the given location using the corresponding manager object. update method will trigger all game managers to iterate the game state. For example it will call moveAll method of spaceShip manager, fireProjectiles method of turretManager etc. render method calls render methods of all manager objects to display game entities. checkGameOver will update gameOver variable if all turns are over.

User

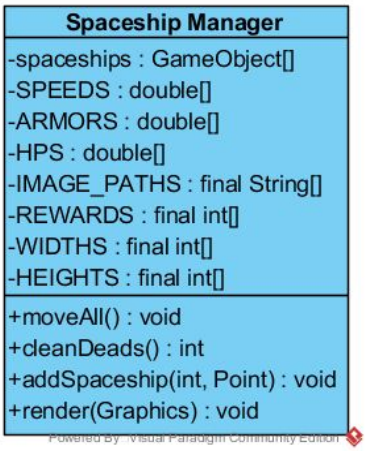
User class holds information simple information about the users such as how many turns they won so far and how much gold they have to put more items on the map.

FactoryManager



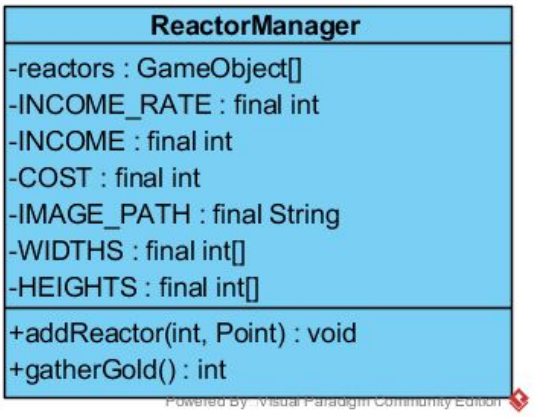
FactoryManager class will handle all the factories that are present on the map. It includes constant values regarding properties of factories for different types. addFactory method add a factory of given type to specified location. produceSpaceships method will create new Spaceship instances for all factories depending on their timing. render method is used to draw all factories on the screen.

SpaceshipManager



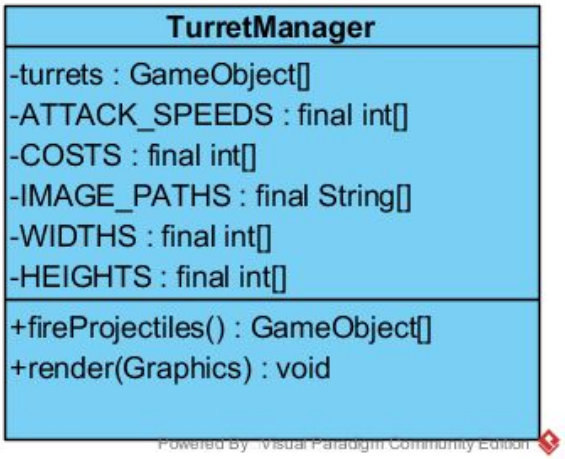
SpaceshipManager class holds all Spaceship objects on the map and constants used for creating a new spaceship. moveAll method moves all spaceships on the map. cleanDeads method removes spaceships which are killed or exceeded map boundaries. render method is used to draw all spaceships on the map.

ReactorManager



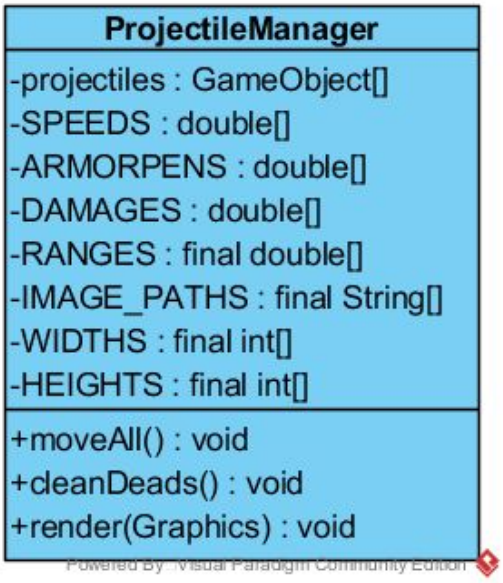
ReactorManager class holds all Reactor objects on the map and constants regarding reactors such as INCOME, INCOME_RATE etc. addReactor method creates a new Reactor on the specified location. gatherGold methods harvests all reactors and returns total income that will later be added to the attacker user. render method is used to draw all reactors on the screen.

TurretManager



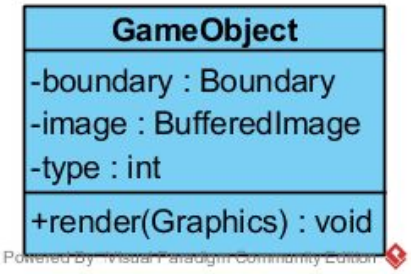
TurretManager class handles all operations related to turrets. It holds all Turret objects that are present in the game. Also it includes constants related to turret features such as ATTACK_SPEEDS, COSTS etc. addTurret method creates a new Turret of the given type on the specified location. fireProjectiles method iterates all turrets and fires projectiles depending on corresponding turrets' timing. render method is used to draw all turrets on the screen.

Projectile Manager



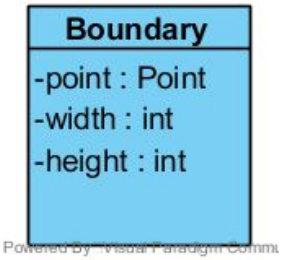
ProjectileManager class holds all projectiles that are present in the game. moveAll method moves all projectiles on the map towards their targets. cleanDeads method removes projectiles that are out of range or hit to target. render method is used to draw all projectiles on the screen.

GameObject Class



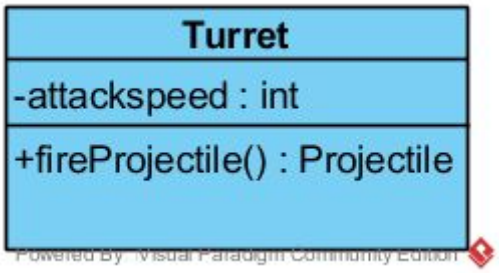
“Gameobject” will be instantiated after users decide the locating towers and factories. Also, “GameObject” will be instantiated in game. GameObject class have three attributes. “boundary” object will use the determining size of the object and position the object. “image” object will use for storing object image. “Type” will use for determine object type. All fundamental object classes will use “GameObject” class as a parent class; since, they all need boundary object for determining their size and position, image for drawing process. Because of that “GameObject” has all of these attributes. “render(Graphics)” method will be used to draw the object onto screen.

Boundary Class



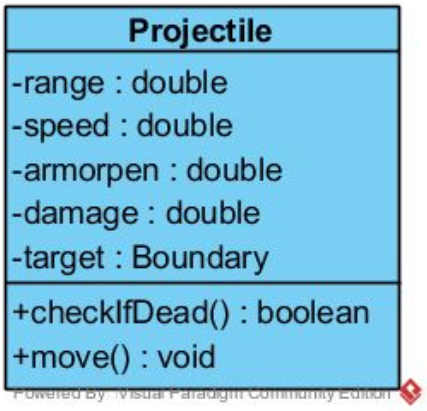
“Boundary” class has three attributes. “Point” stores the information of the x and y coordinates of the object. Width and height attributes store the dimensions of the object.

Turret Class



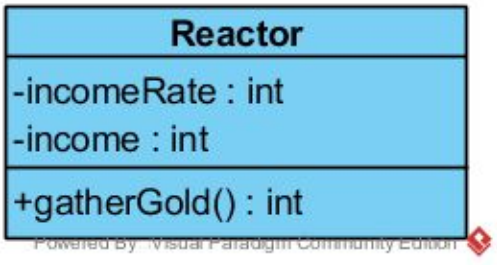
“TurretClass” has one addition to “GameObject” class attributes. In “attackspeed”, the system stores the information of turret fire in how many seconds it projectiles. “fireProjectile()” will create the projectile and return it as an object.

Projectile Class



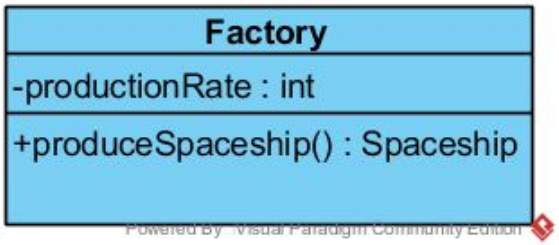
"Range" will store the maximum distance that projectile can advance. "speed" stores the speed of the project tile. "armorpen" will be used in the calculation of damage that it creates, when projectile hits the spaceship. "damage" stores the damage amount when projectile hits the spaceship. "target" will store position of the target spaceship. "checkIfDead()" method will check the projectile is out of the range or hits any spaceship; it returns true or false. "move()" method moves the project tile with respect to speed.

Reactor Class



"Reactor" class has two additional attributes to "GameObject" class attributes. "income" stores an int value for how much reactor object create gold. "incomeRate" stores the int value that we will use determining time that "gatherGold()" called. "gatherGold()" method increases the gold amount.

Factory Class

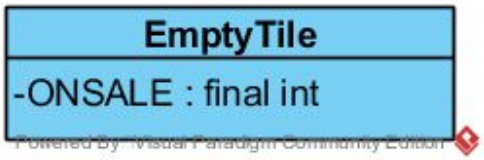


"productionRate" stores the int value that we will use to determine the time when "produceSpaceship()" method called. "produceSpaceship()" method creates spaceship when it called.

Spaceship Class

"Spaceship" class has 5 additions to "GameObject" class' attributes. "speed" attribute stores the speed of the spaceship. "target" attribute stores the target of the spaceship. "armor" will store armor value of spaceship when projectile hits the spaceship armor will decrease. "hp" will store health of the spaceship as an int value. Also this value decreases when the projectile hits the spaceship when spaceship destroyed by turret defender will gold. "reward" will store this gold value. "checkIfDead()" controls the object and if hp of spaceship less than it will return true, otherwise it will return false. "move()" method moves the spaceship with respect to speed.

EmptyTile Class



"EmptyTile" has one addition to "GameObject" class' attributes. "ONSALE" can have five different possible values. One of the them is if a tile which belongs to attacker side and it is able to be sold by the user. The other one is a tile which belongs to defender side and the tile cannot be sold...