Bilkent University

Department of Computer Engineering

CS-319 Object Oriented Software Engineering Project

Sword & Shield: A Space Adventure

System Design Report

Iteration II

Group 2D

Akın Berkay Bal

Eren Aslantürk

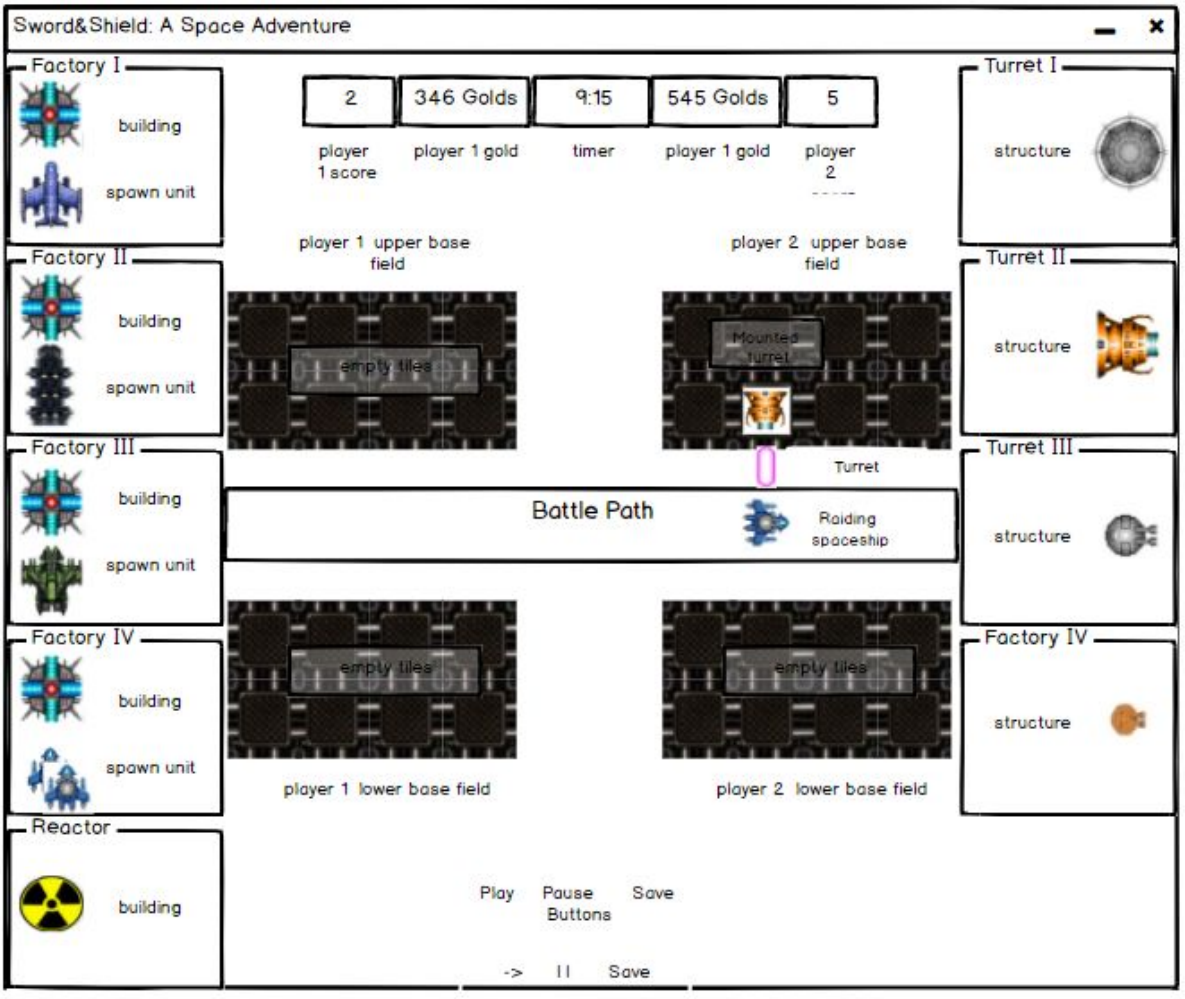
Mehmet Enes Keleş

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. A sample gameplay mockup is shown down below for reader to better understand the concepts in this report:



1.1 Purpose of the System

Sword & Shield: A Space Adventure is a 2 player top-down strategy/tower defense game that is mainly inspired from Bloons: Tower Defense but it 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. The most important one among 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 implement 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

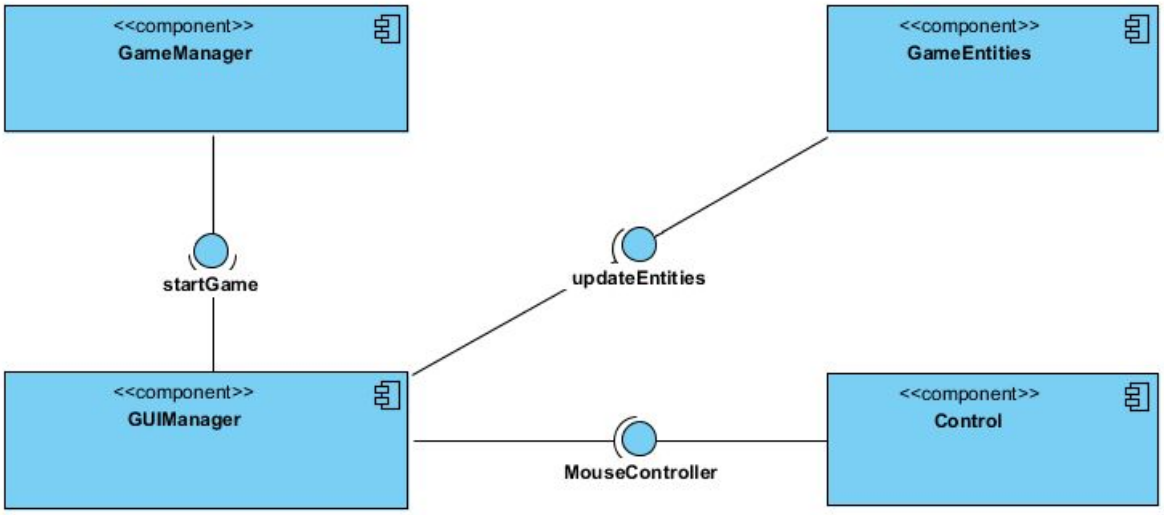
2.1 Overview

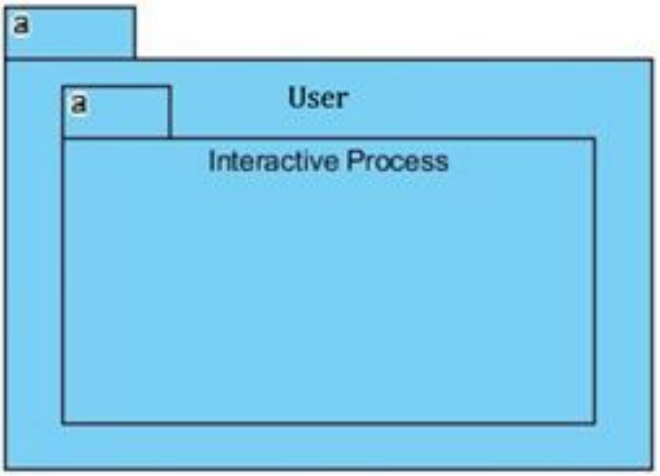
These sections below give details about software structure of our project application. Software system of the project depends on a trilayer format and it can be run on any kind of laptop or desktop. The requirements are a screen, a mouse and optimally a windows operating system.

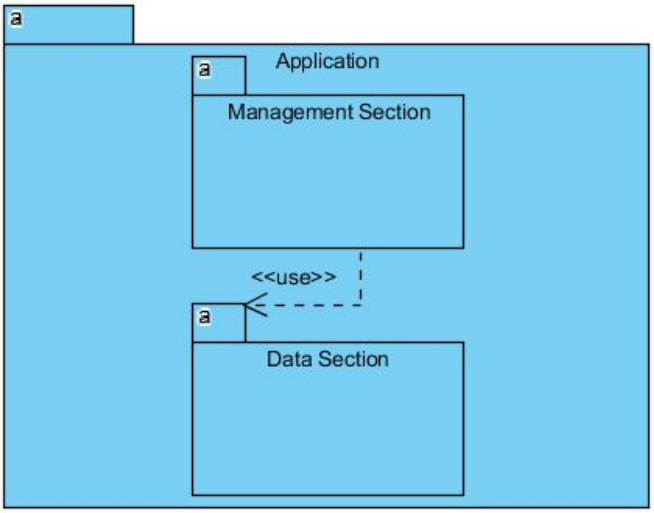
2.2 Subsystem Decomposition

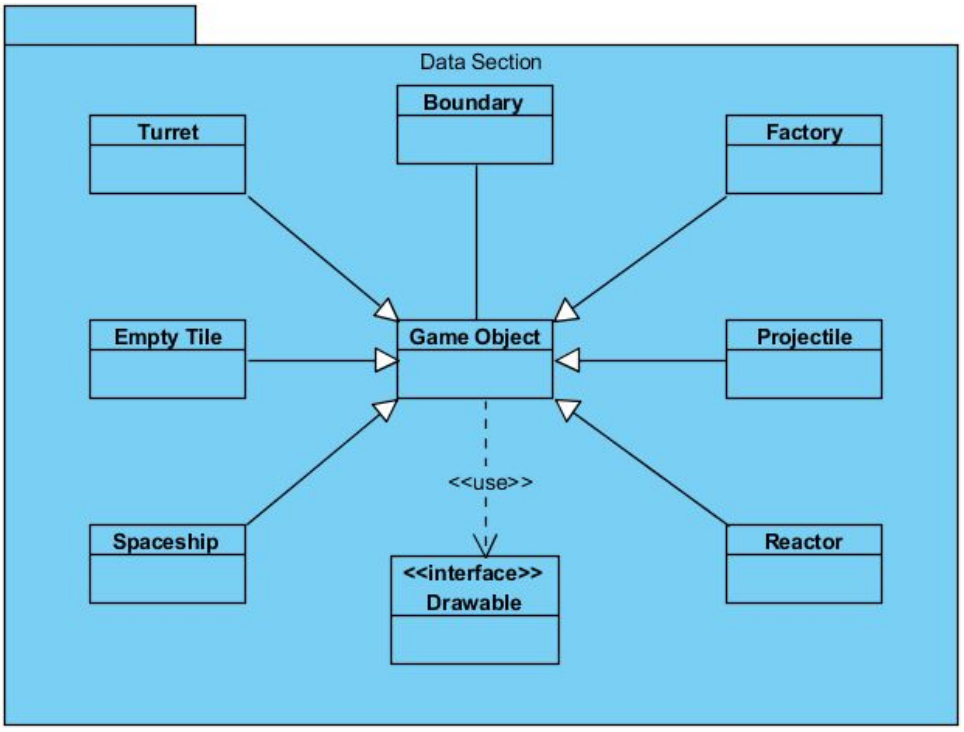
The system of application based on trilayer format as mentioned. In this format, system interacts with user, updates status of the process and initiates system according to user. User is the obligatory element in the system as the activator of the application, an accessor to frames in the application 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

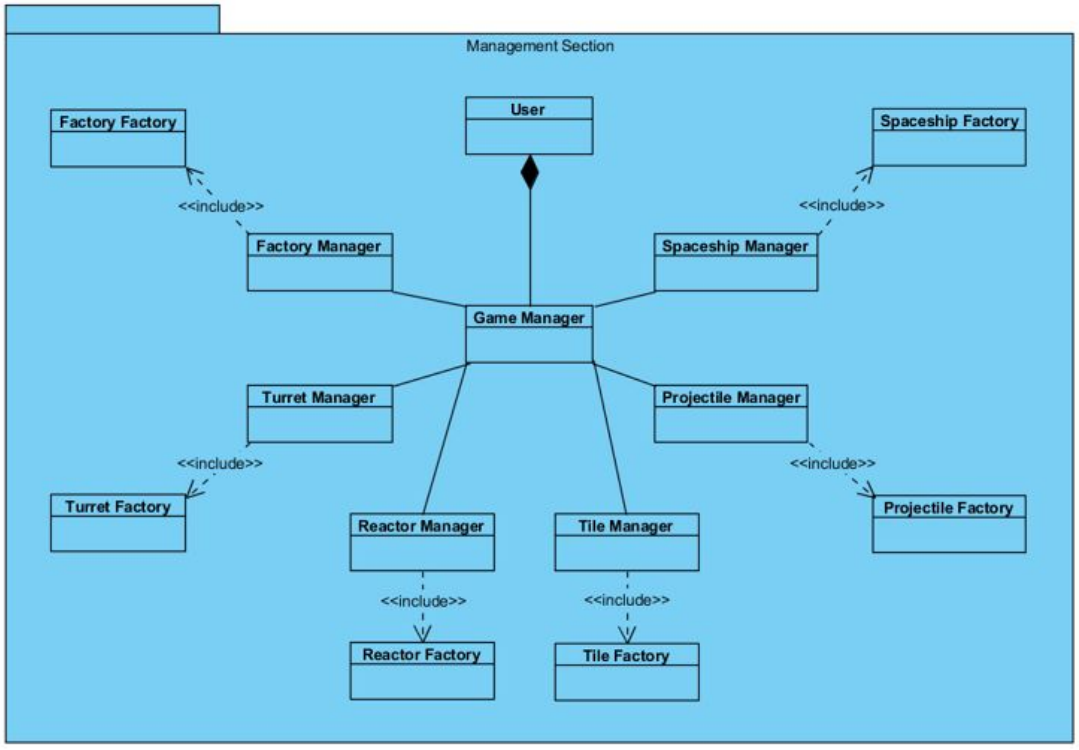
application proceeds the situation and keeps data need to be stored or gets rid of the unnecessary ones. Detailed visualization of the decomposition is as shown in the graphs below.

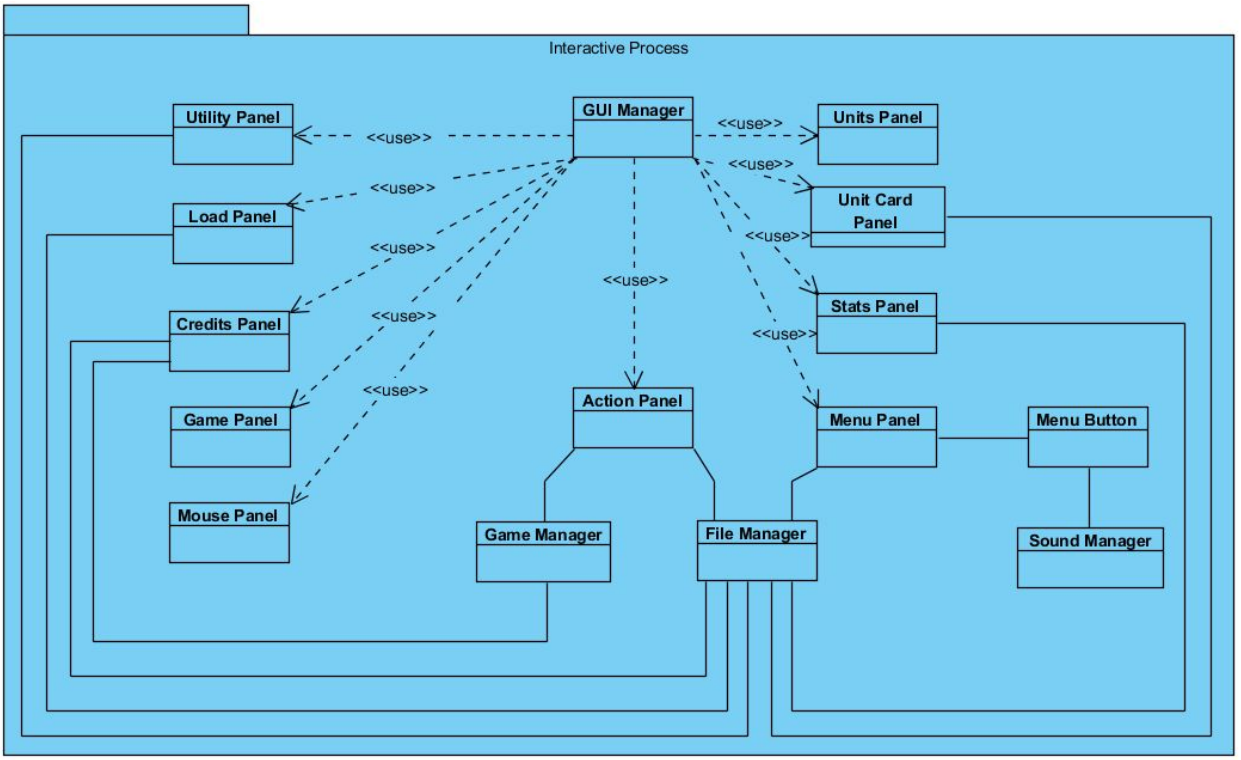





a
Application
a
Management Section
<<use>>
a
Data Section


Data Section
Boundary
Turret
Factory
Empty Tile
Game Object
Projectile
<<use>>
Spaceship
<<interface>>
Drawable
Reactor


Management Section
User
Factory Factory
Spaceship Factory
<<include>>
<<include>>
Factory Manager
Spaceship Manager
Game Manager
Turret Manager
Projectile Manager
<<include>>
<<include>>
Turret Factory
Projectile Factory
Reactor Manager
Tile Manager
<<include>>
<<include>>
Reactor Factory
Tile Factory


Interactive Process
GUI Manager
Utility Panel
Units Panel
Load Panel
Unit Card Panel
Credits Panel
Stats Panel
Game Panel
Action Panel
Menu Panel
Menu Button
Mouse Panel
Game Manager
File Manager
Sound Manager
<<use>>

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 equipment. 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 user 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

data 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 wish to use same progress data 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 auto-running the executable 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 moves them by mistake while installing or wants to store them 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

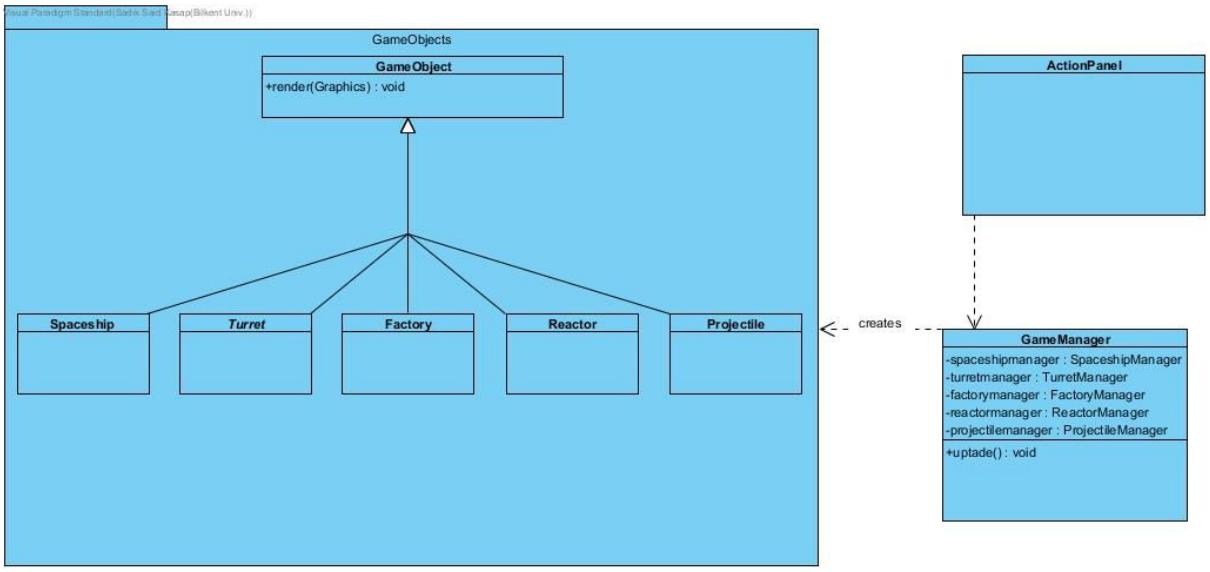
3.1.1 MVC Architectural Design

In MVC design application's concerns divided 3 parts which are Model, View and Controller. So we separated our design into these three component. These components are called GUI Subsystem as View, Logic Subsystem as Controller and Entity Subsystem as Model. 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

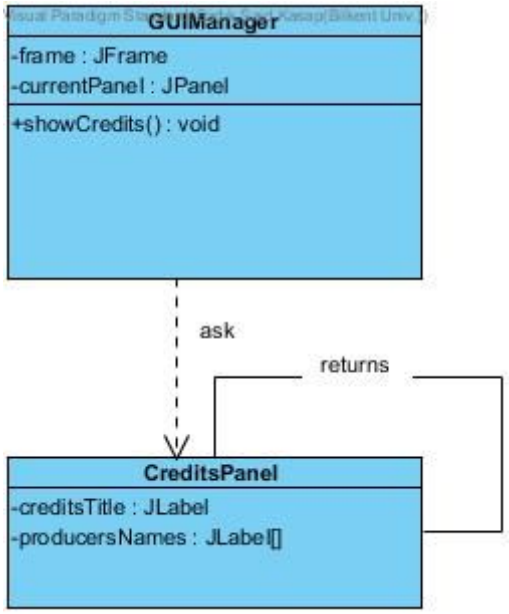
Façade Pattern contains a class that works as an interface for the user by providing suitable methods to manipulate all of the subsystem. This kind of approach hides complexities of the underlying design and makes the system more understandable and user-friendly. In order to make our code more organized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



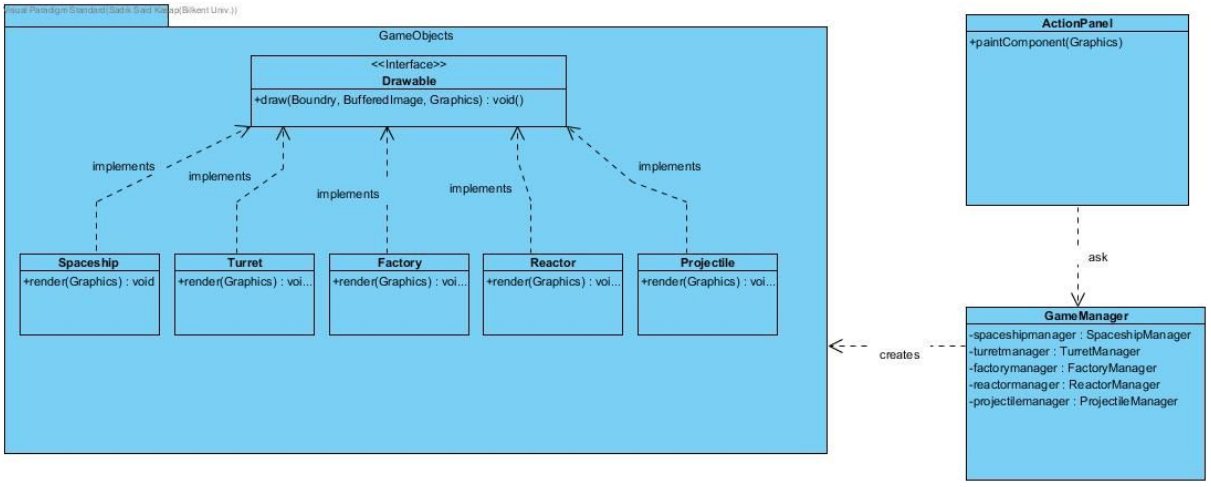
3.1.3 Singleton Pattern

Singleton pattern ensures that 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 subsystem to make our code more reliable. For the most of the classes in this subsystem, we do not need more than one instance running at a time. An opposite situation can give a birth to many possible mistakes.



3.1.4 Factory Design Pattern

In our design, we decided to use factory design pattern in order to create game objects without exposing them to outside. Concrete classes: Spaceship, Factory, Turret, Reactor and Projectile implements Drawable interface, that has “draw()” method with parameters Boundary, BufferedImage and Graphics. Those classes have their own draw methods and they define their own draw method differently according to their own specialties, such as various images for different types of spaceships. GameManager is the factory class that creates gameobjects. Action panel is the client object that has own render method. Action panel calls draw method of every object after asking from GameManager class. ActionPanel can access these objects only by using GameManager class.



3.2 GUI Subsystem

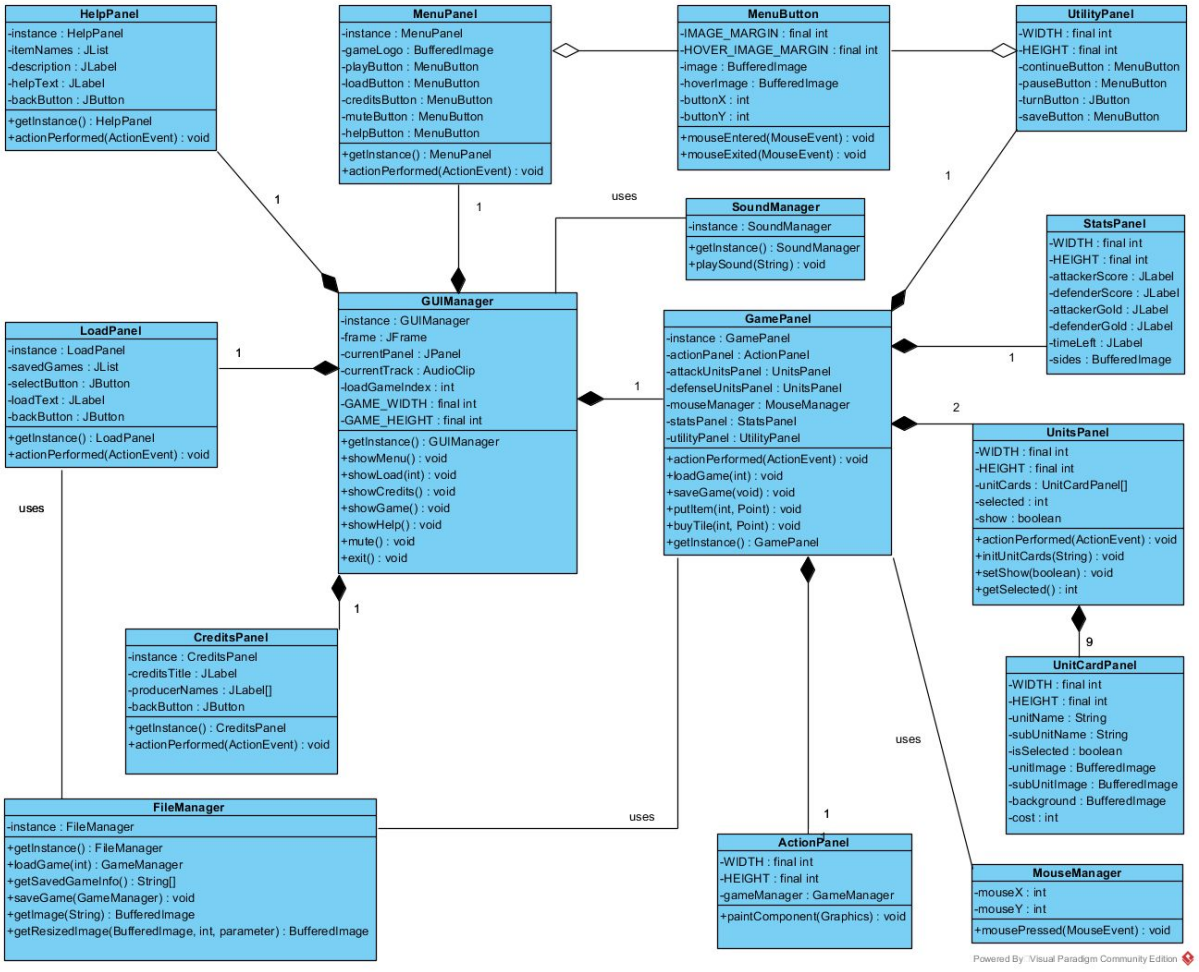


Figure 3.2.1 (Overall GUI Subsystem)

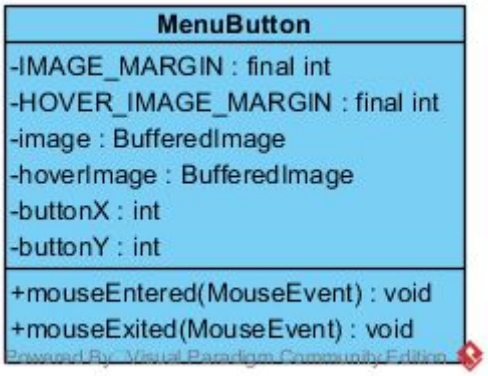
GUI Subsystem shows the current state of the game to user as well as allows user to interact with the program. It includes all the boundary objects.

GUIManager

GUIManager is a singleton class that is responsible for displaying the desired panel on the screen. For this purpose, the class has a frame which contains “currentPanel” - that is being used at that moment. It has a “currentTrack” object to play the music. Additionally, “loadGameIndex” is used to load a game. This attribute will not be used if the user does not use load game functionality.

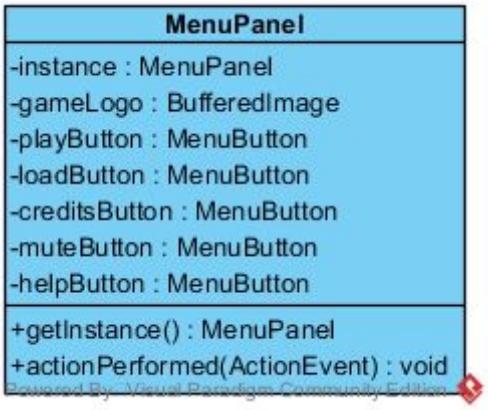
‘Show methods’ are used to change the panel being displayed at that moment. mute() method is used to turn off the sound. exit() method closes the game.

MenuButton



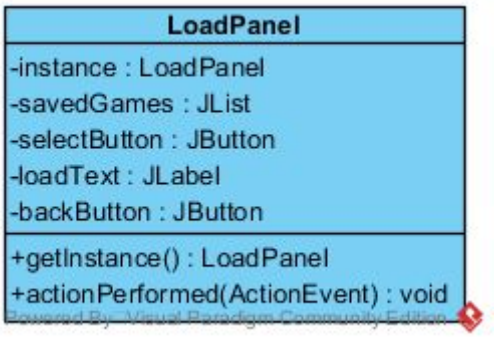
MenuButton class is a customized version of JButton. It has two different images. Normally, the image held by "image" attribute is shown on screen. If the mouse hovers over the button, the image held by "hoverImage" attribute is shown on screen.

MenuPanel



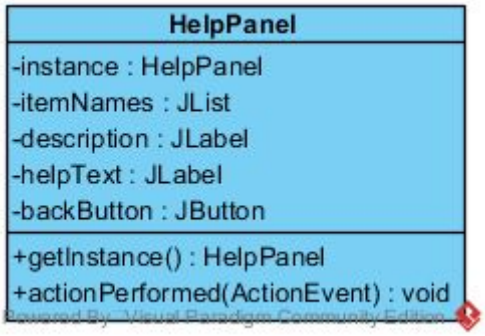
MenuPanel is a singleton class that is used to navigate between different functionalities of the game. By clicking buttons (loadButton, playButton etc.), the user can reach to LoadPanel, GamePanel, CreditsPanel and HelpPanel. Also, the user can turn off the music. To provide these functionalities, it uses 'show methods' of GUIManager.

LoadPanel



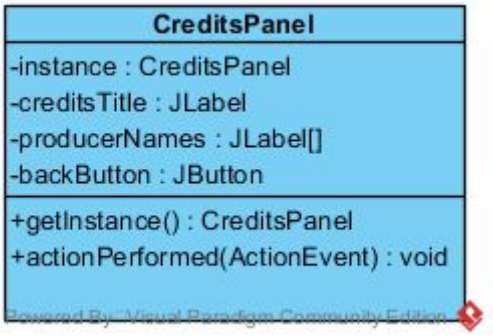
LoadPanel is singleton class that has “savedGames” attribute of type JList that contains the games that are saved. These list will be displayed on screen. The information of saved games will be obtained using getSavedGameInfo() method of FileManager class. If the user selects a saved game to be loaded, “loadGameIndex” attribute of GUIManager will be modified. And the game will be loaded accordingly.

HelpPanel



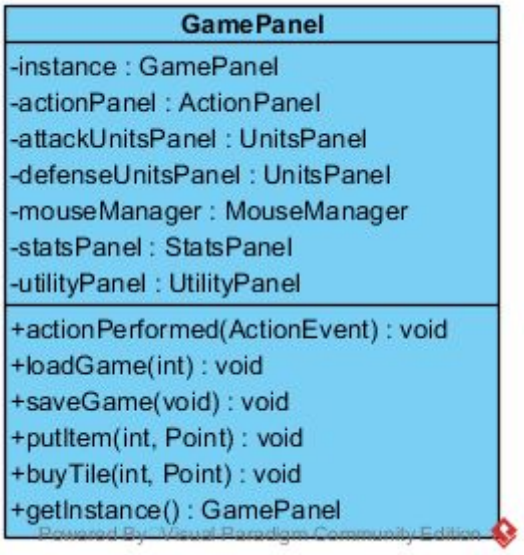
HelpPanel is a singleton class that shows the information about items in the game. “description” attribute will show the info about the selected item on the screen.

CreditsPanel



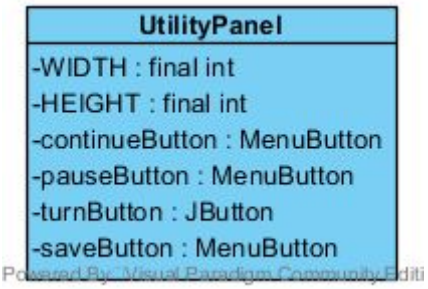
CreditsPanel is a singleton class that shows the names of producers of the game on the screen.

GamePanel



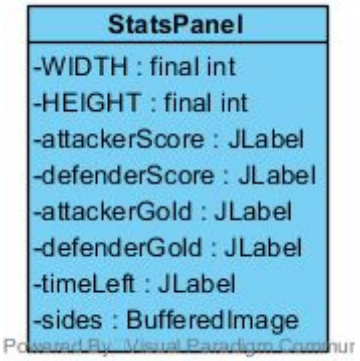
GamePanel is a singleton class that is the screen where the user play the game. It is divided into many parts - “actionPanel” in the middle, “attackUnitsPanel” on the left, “defenseUnitsPanel” on the right, “statsPanel” at the bottom and “utilityPanel” at the top . It also provides control utilities for user to place items, buy tiles, save the game, load a saved game (putItem(), buyTile(), saveGame(), loadGame() methods are used for these purposes).

UtilityPanel



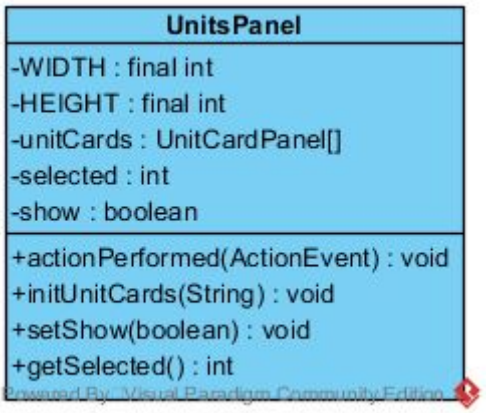
UtilityPanel is a panel that appears at the bottom of the screen for the user to reach different functionalities (save, pause, continue, switch turns) during the gameplay.

StatsPanel



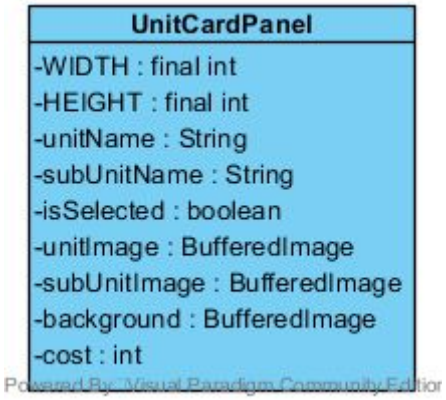
StatsPanel is a panel that appears at the top of the screen for the user to see scores, golds for both players and the time left during the gameplay.

UnitsPanel



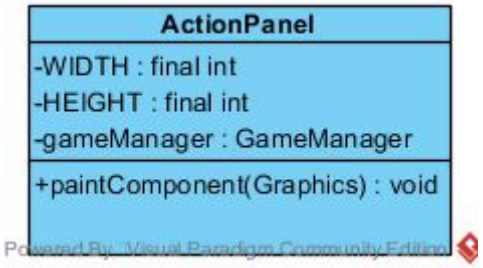
UnitsPanel is a panel that appears on the right and on the left of the screen for the user to selects items that he/she wants to place on the map during the gameplay. It holds unit cards for either attack units or defense units. initUnitCards() method takes a string parameter that specifies which type of units will be contained. Fetching information from “factory classes”, that will be explained later , the method initializes these unit cards.

UnitCardPanel



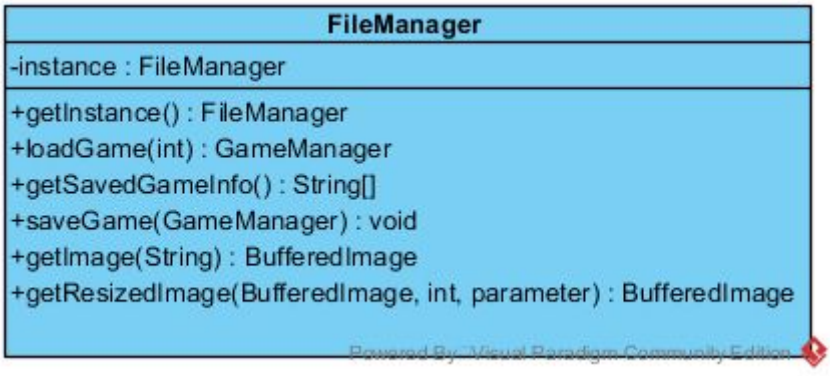
A UnitCardPanel is a panel that shows a single item and its sub item. For example a factory and the spaceship it produces or a turret and the projectile it fires. Item names for both unit and sub unit are also shown.

ActionPanel



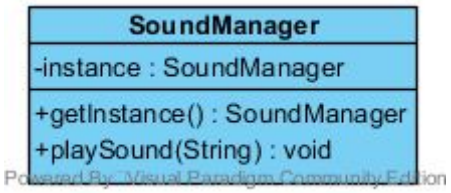
ActionPanel is the panel in the middle of the gameplay screen. It holds a "gameManager" attribute and displays all game entities that is included inside that attribute. Therefore, the flow of game entities will be seen inside this panel.

FileManager



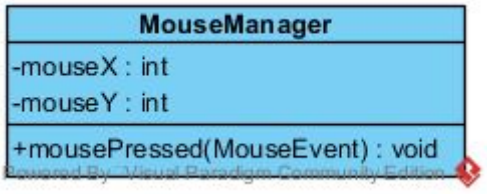
FileManager is a utility singleton class that is included in the GUI subsystem. It provides save/load functionalities by interacting with the filesystem. When other clases needs to save/resize images or save/load games, this class is used.

SoundManager



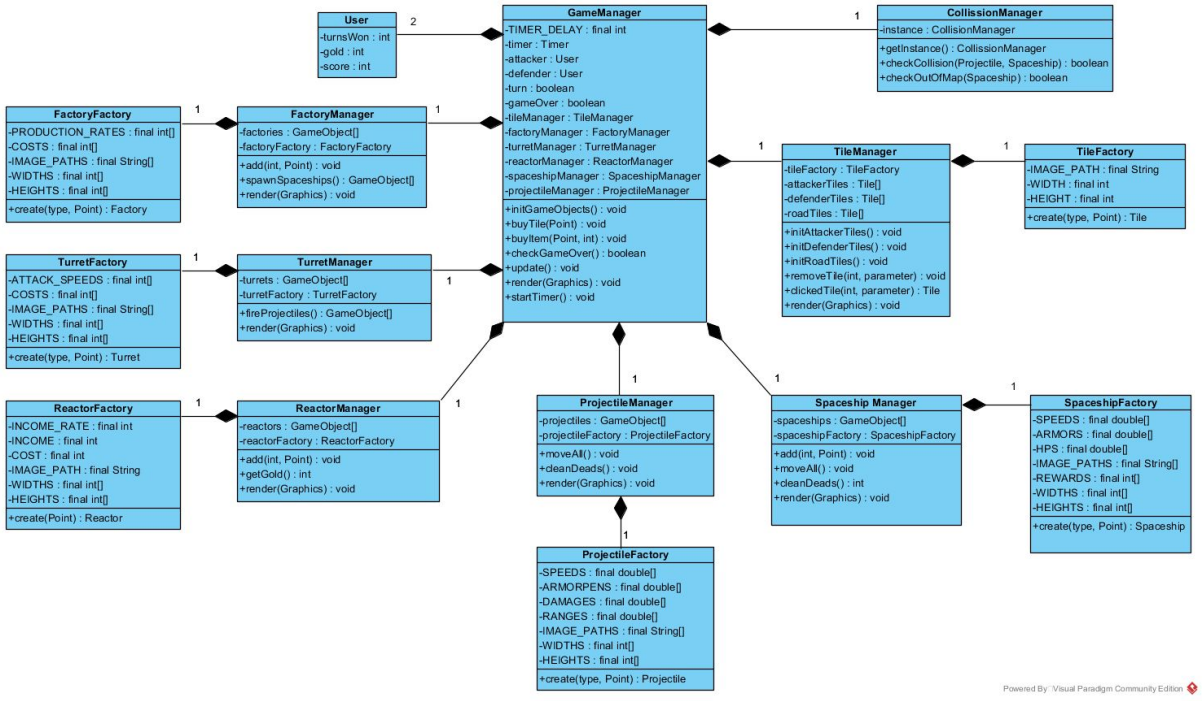
SoundManager is a singleton class that provides only one functionality: playing the audio file at the given file path.

MouseManager



This class a control class that will be used during the gameplay. mouseX and mouseY specifies the current location of the mouse and mousePressed() method is an event handler that works when the user clicks somewhere on the screen.

3.3 Logic Subsystem



Logic Subsystem keeps the current game state. It is also a intermediate layer between GUI Subsystem and Entity Subsystem. All mechanics in the game are implemented in this layer.

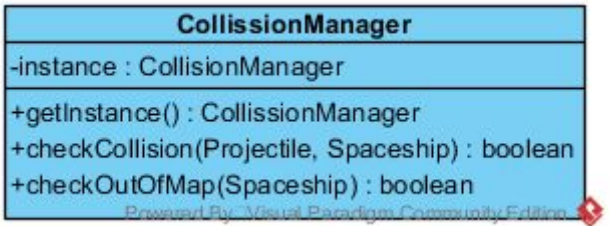
GameManager

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. initGameObjects() method will be used to initially create the required items on the predefined game setup. buyTile() method will find the Tile object corresponding to the given point and buy it if the user has enough money and the Tile 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 "spaceshipManager", fireProjectiles() method of "turretManager" etc. render() method calls render() methods of all manager objects to display game entities. checkGameOver() will update "gameOver" attribute if all turns are completed.

User

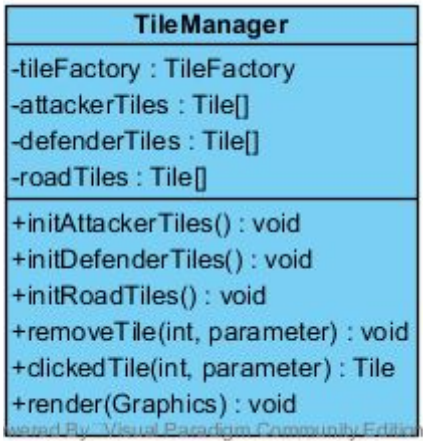
User class holds simple information about the users such as their scores, how many turns they won so far and how much gold they have to put more items on the map.

CollisionManager



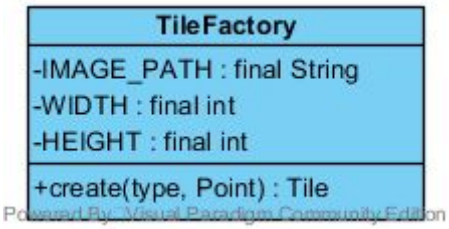
CollisionManager is used to detect collisions in the game. First type of collision occurs between a projectile and a spaceship. checkCollision() method is used to detect these situations. Second type of collision occurs when a spaceship goes out of the map - checkOutOfMap() method will be used to detect these situations.

TileManager



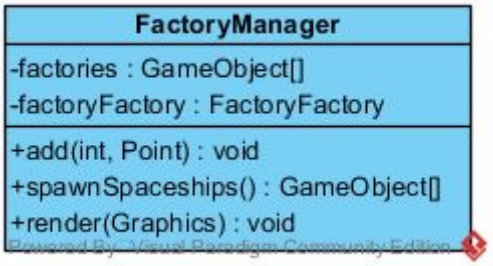
TileManager class holds all Tile objects that are present in the game. initAttackerTiles() method is used to initialize tiles on the attacker side (left). initAttackerTiles() method is used to initialize tiles on the attacker side (right). initRoadTiles() method is used to initialize tiles of the path that the spaceships are flying over. removeTile() method removes the tile at the specified location. clickedTile() method returns the clicked tile when the mouse location is given. render() methods draws all tiles on the screen.

TileFactory



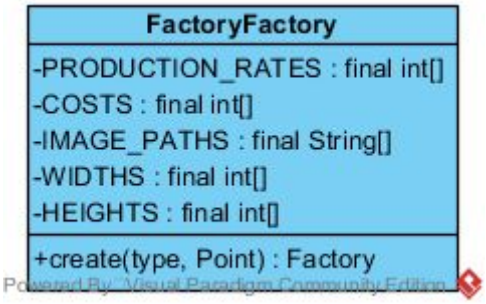
TileFactory class holds all the information about tiles - “IMAGE_PATH”, “WIDTH”, “HEIGHT”. create() method is used for creating a tile at the specified location.

FactoryManager



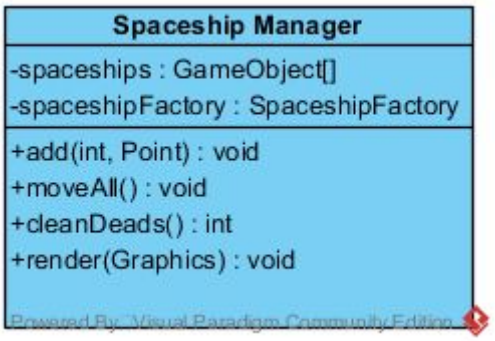
FactoryManager class will handle all the factories that are present on the map. It holds all Factory objects that are present in the game. add() method adds a factory of given type to specified location. spawnSpaceships() method will create new Spaceship instances for all factories depending on their timing. render method is used to draw all factories on the screen.

FactoryFactory



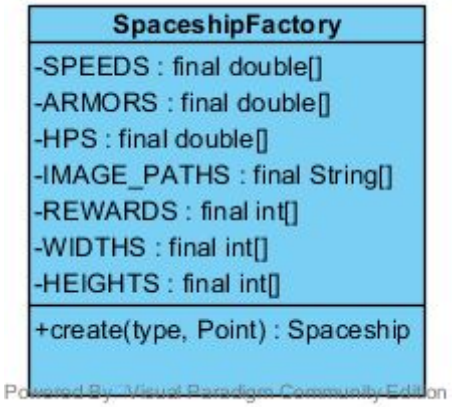
FactoryFactory class holds all the information about factories - “PRODUCTION_RATES”, “WIDTHS”, “HEIGHTS”, “COSTS”, “IMAGE_PATHS”. create() method is used for creating a specific type of factory at the specified location.

SpaceshipManager



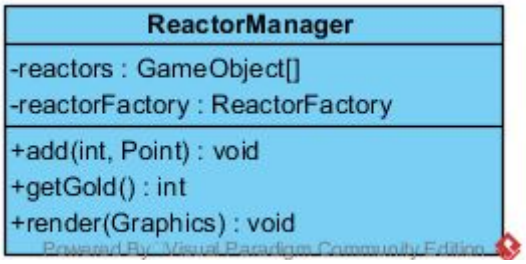
SpaceshipManager class holds all Spaceship objects on the map. moveAll() method moves all spaceships on the map. cleanDeads() method removes spaceships which are killed or exceeded map boundaries. render() method is used to draw all spaceships on the map.

SpaceshipFactory



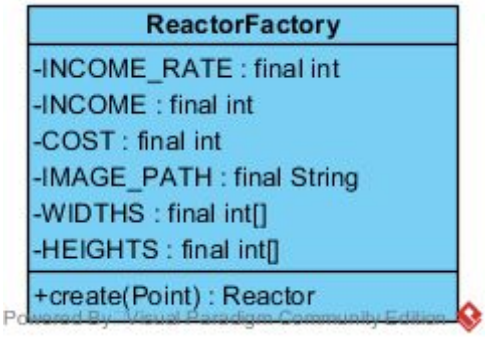
SpaceshipFactory class holds all the information about spaceships - “SPEEDS”, “ARMORS”, “HPS”, “REWARDS” , “WIDTHS”, “HEIGHTS”, “IMAGE_PATHS”. create() method is used for creating a specific type of spaceship at the specified location.

ReactorManager



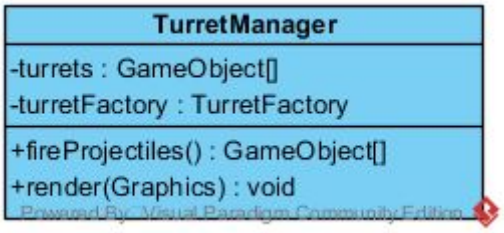
ReactorManager class holds all Reactor objects on the map. add() method creates a new Reactor on the specified location. getGold() methods harvests all reactors and returns total income that will later be added to the attacker user.. render() method is used to draw all reactors on the screen.

ReactorFactory



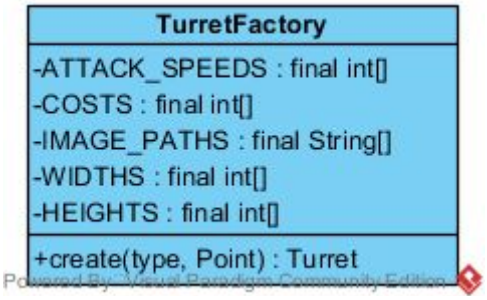
ReactorFactory class holds all the information about reactors - “INCOME_RATE”, “INCOME”, “COST”, “IMAGE_PATH”, “WIDTHS”, “HEIGHTS”. create() method is used for creating a reactor at the specified location.

TurretManager



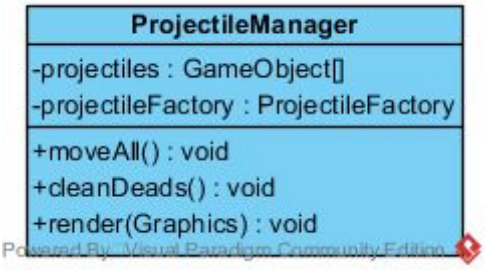
TurretManager class handles all operations related to turrets. It holds all Turret objects that are present in the game. add() method creates a new Turret of the given type on the specified location. fireProjectiles() method iterates all turrets and fires projectiles depending on corresponding turrets’ timing. render() method is used to draw all turrets on the screen.

TurretFactory



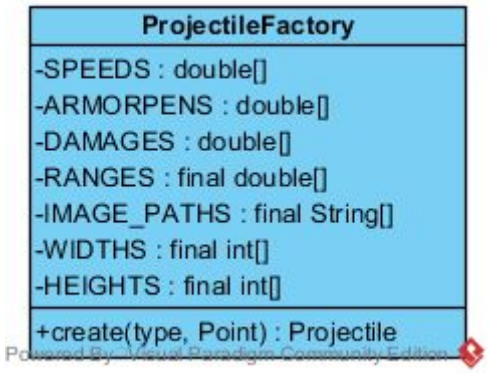
TurretFactory class holds all the information about reactors - “ATTACK_SPEEDS”, “COSTS”, “IMAGE_PATHS”, “WIDTHS”, “HEIGHTS”. create() method is used for creating a turret at the specified location.

ProjectileManager



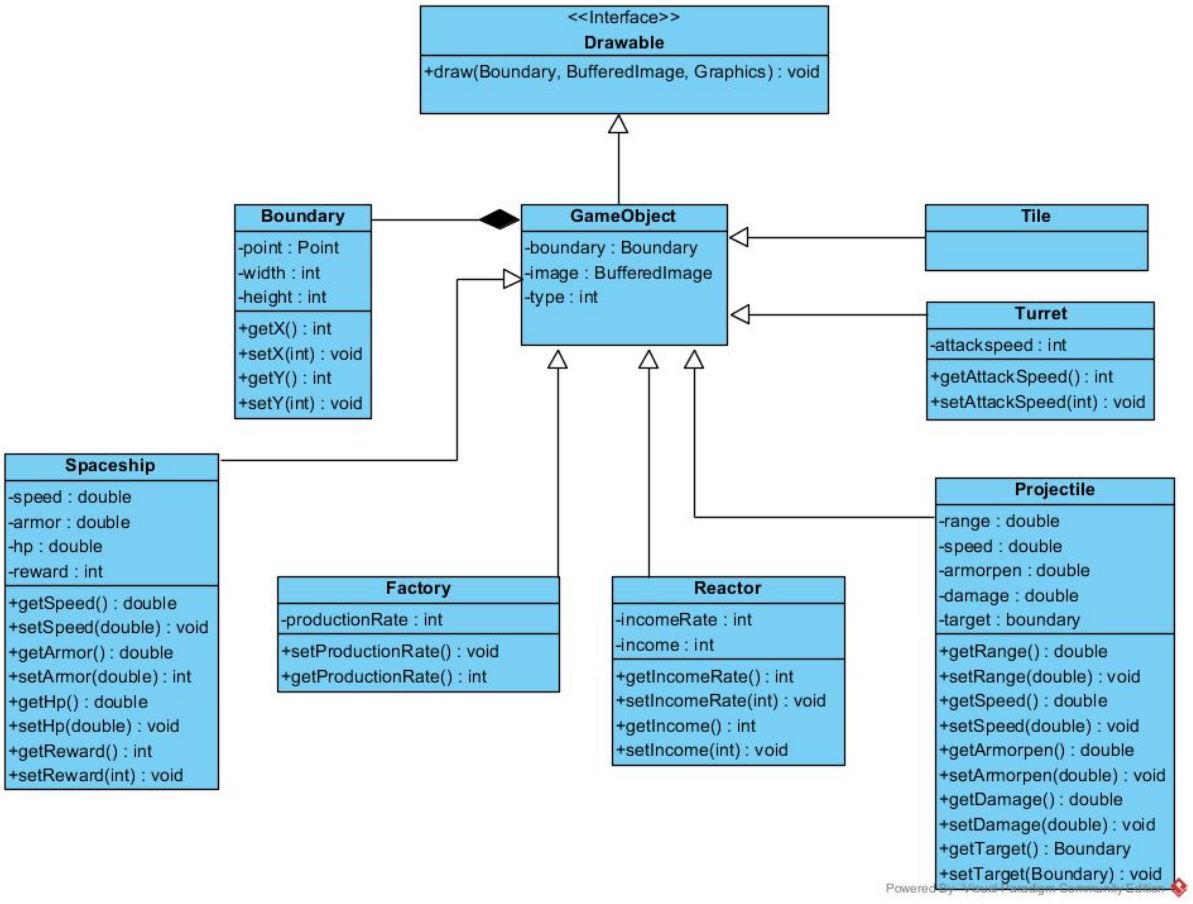
ProjectileManager class holds all projectiles that are present in the game. moveAll() method moves all projectiles on the map towards their targets. cleanDeads() method removes projectiles that are out of range or hit to target. render() method is used to draw all projectiles on the screen.

ProjectileFactory

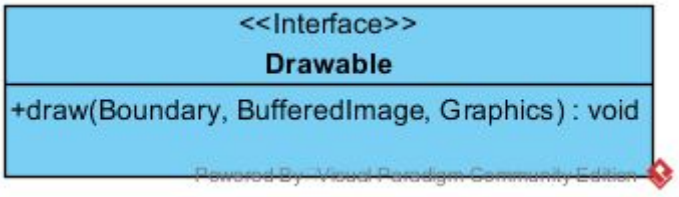


ProjectileFactory class holds all the information about reactors - “SPEEDS”, “ARMORPENS”, “DAMAGES”, “RANGES”, “IMAGE_PATHS”, “WIDTHS”, “HEIGHTS”. create() method is used for creating a projectile at the specified location.

3.4 Entity Subsystem

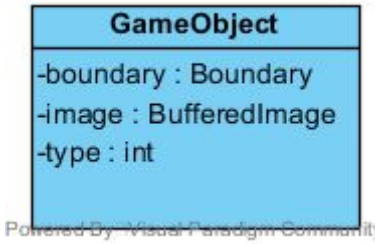


Drawable Interface



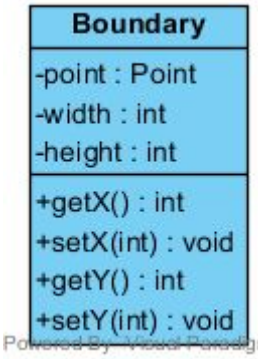
Drawable interface has “draw()” abstract method that all gameobjects will be used for drawing.

GameObject



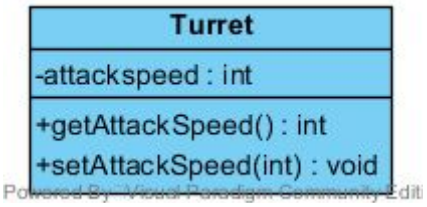
"Gameobject" will be instantiated after users decide the locating towers and factories. Also, "GameObject" will be instantiated in game. GameObject class have three attributes. "boundary" object will use the determining size of the object and position the object. "image" object will use for storing object 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.

Boundary



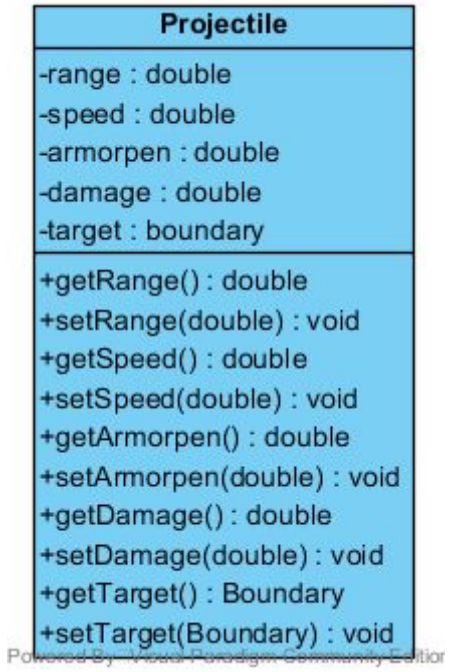
"Boundary" class has three attributes. "Point" stores the information of the x and y coordinates of the object. Width and height attributes store the dimensions of the object.

Turret



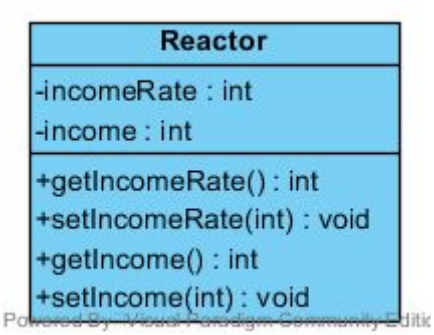
"TurretClass" has one addition to "GameObject" class attributes. In "attackspeed", the system stores the information of turret fire in how many seconds it projectiles.

Projectile



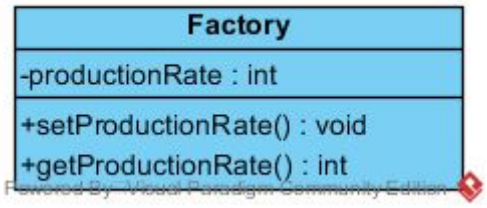
"Range" will store the maximum distance that projectile can advance. "speed" stores the speed of the project tile. "armorpen" will be used in the calculation of damage that it creates, when projectile hits the spaceship. "damage" stores the damage amount when projectile hits the spaceship. "target" will store position of the target spaceship.

Reactor



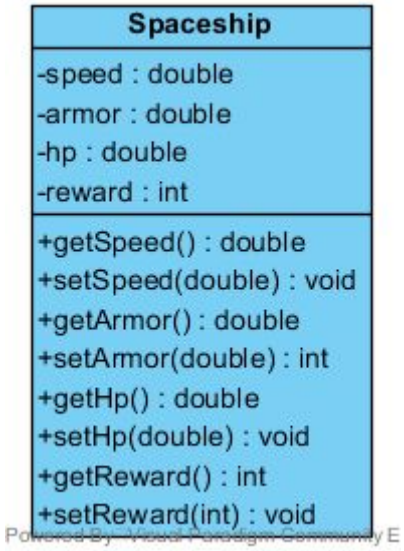
"Reactor" class has two additional attributes to "GameObject" class attributes. "income" stores an int value for how much reactor object create gold. "incomeRate" stores the int value that we will use determining time that "getIncome()" called. "getIncome()" method increases the gold amount.

Factory



"productionRate" stores the int value that we will use to determine the time when spaceship spawn.

Spaceship



"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. "move()" method moves the spaceship with respect to speed.

Tile

4. Improvement Summary

- In first iteration we were using three layer architectural style but in the second iteration we changed to MVC upon revision of our instructor.
- Load and save features are added in this iteration.
- HelpPanel is added in the GUI subsystem.
- For singleton and façade patterns we added explanatory diagrams. In addition, factory design pattern is used in Logic subsystem. Factory classes will be useful to create new objects.
- Timer that is used for iterating the game state is moved into GameManager class from ActionPanel class
- CollisionManager and SoundManager classes are added
- Methods added to FileManager class
- Drawable interface is added. It will be convenient while drawing different types of objects onto screen.
- MenuButton class is added to GUI subsystem. It's a customized JButton with smooth and responsive appearance.
- GamePanel and UnitsPanel classes are divided so UnitCardPanel, UtilityPanel and StatsPanel classes are added.
- Most of the utility is shifted to GamePanel class from ActionPanel class.