CS 319

Object-Oriented Software Engineering

Project Design Report

Defender

GROUP 2-D

Taha Khurram - 21701824

Mian Usman Naeem Kakakhel - 21701015

Muhammad Saboor - 21701034

Sayed Abdullah Qutb - 21701024

Balaj Saleem - 21701041

1. Introduction

1.1 Purpose of the system

Defender is a two-dimensional space shooter game set on an alien planet [1]. The player controls a spaceship that can move in all four possible directions; however, only horizontal scrolling is permitted. During the game, the player is faced with waves of enemies. There are also humans (astronauts) located on the surface of the planet. These astronauts are to be protected by the spaceship. The enemies aim to abduct these humans and take them off into space (off-screen) and the player must prevent this. The player must also avoid enemy attacks since contact with an enemy or projectile lead to the player's death.

1.2 Design goals

1.2.1 Ease of Use

Efficiency and high performance will be maintained due to the simple nature of the game. We will be limiting our memory usage by placing a maximum limit of 50 objects in the game, the objects include bullets, the spaceship, enemies, asteroids, and humans. We will use C++ so that we can destroy objects not needed such as a bullet when it goes out of the camera view.

1.2.2 High Performance

Efficiency and high performance will be maintained due to the simple nature of the game. We will be limiting our memory usage by placing a maximum limit of 50 objects in the game, the objects include bullets, the spaceship, enemies, asteroids, and humans. We will use C++ so that we can destroy objects not needed such as a bullet when it goes out of the camera view.

1.2.3 Extendability

We will design the system to make future implementations easier by introducing minimal changes to our design. For example, we may choose to improve game mechanics such as better shooting or increase the variety of enemies for which we will have to just extend upon the already existing basic game design which reduces the amount of work needed.

1.2.4 Portability

The code for our game is being written using C++ which has made our game available for both Windows and Linux platforms and by using the JNI(Java Native Interface)[2] the game can be ported to Android devices further. To port the game to Windows, we will just have to change the architecture of the SDL library from Linux to windows. To port the game to Android, we will have to write configuration files (like AndroidManifest.xml) and wrap the code so that JNI can run the C++ code [3].

2. System Architecture

2.1 Subsystem Decomposition

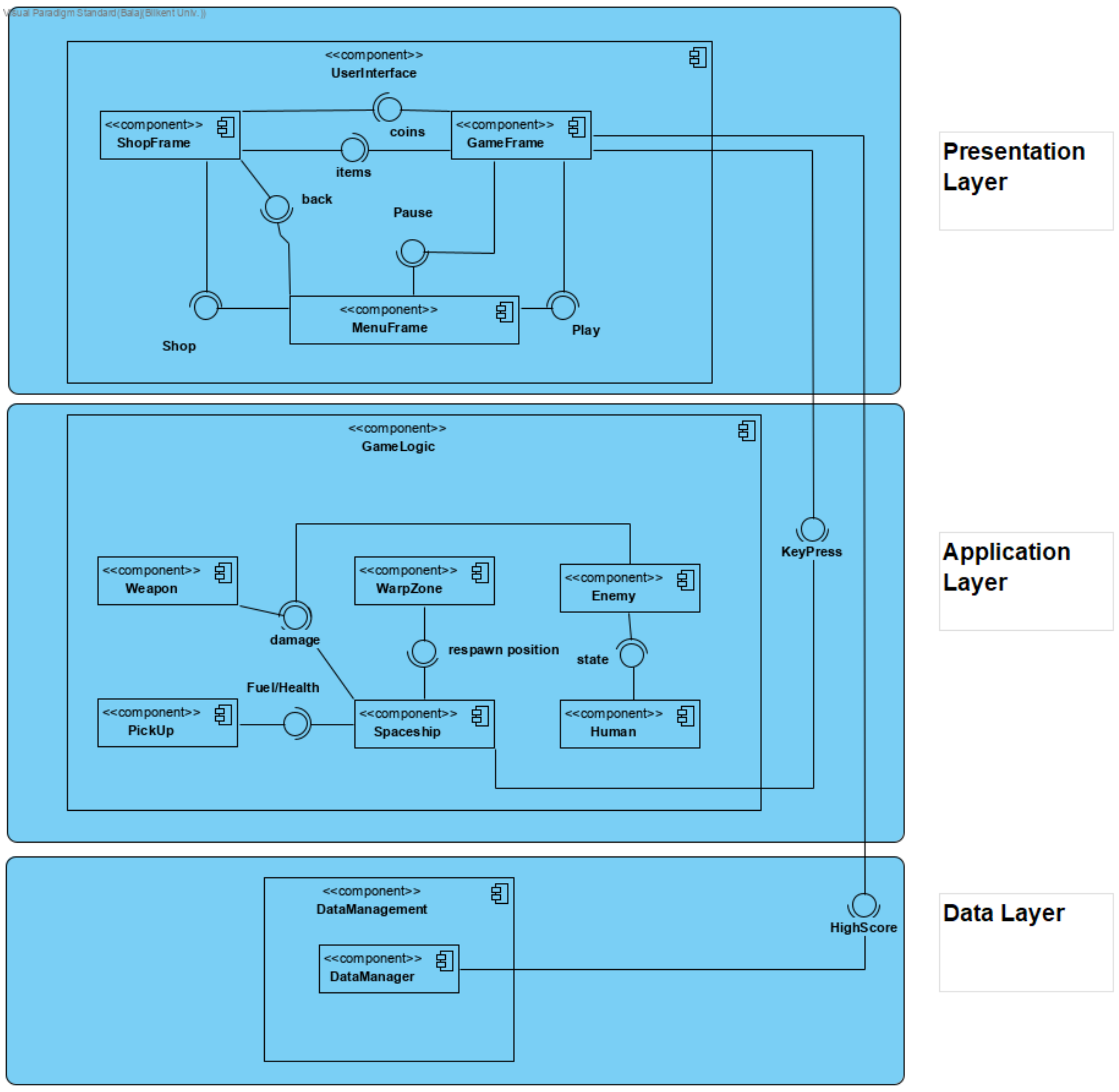

Figure 1 - Subsystem Decomposition

In this section, we will decompose our system into subsystems. Our aim with this decomposition was to reduce dependencies between the subsystems. By removing these dependencies, implementing modifications to the subsystems becomes easier.
During the decomposition of the system, we have decided to use a 3-Tier system design pattern for our game. We have divided our system following the rules of the 3-Tier system design pattern as can be seen above in figure 1.
The 3-Tier system design pattern suits us for the following reasons:

1. Our system has exactly three layers which match with the 3-Tier system design pattern:
    a) We have a **UserInterface** subsystem which is our **presentation layer**. This contains the user interface components for the player.
    b) We have the **GameLogic** subsystem which is our **application layer**. This contains the game logic components which are used to manage the game entity objects.
    c) We have the **DataManagement** subsystem which is our **data layer**. This contains the datamanager component which stores the data regarding the game such as the score.
2. The relationship between components can be easily represented by the 3-Tier system design pattern.
3. The 3-Tier system design pattern improves the efficiency of the program by splitting up the system into independent layers which can be worked upon individually making the entire process faster and easier to modify.

This will be further expanded upon later below in the document.

As we will be using the 3-Tier system design pattern all of our subsections will be following its strict communication rules. The presentation layer will be able to access elements from the application layer and the data layer but the application layer can only access elements of the data layer while the data layer being at the lowest cannot access any of the parent layer elements. This will help to improve the speed of development, scalability, and performance.

The layers are explained as follows:

- The **UserInterface** subsystem in the presentation layer communicates with the GameLogic system through the GameFrame component which detects the users key pressed (using key listeners) and this data the keypress is sent to the SpaceShip component where it will perform the desired action of the user. The UserInterface sends the HighScore through the game frame to the DataManagement component in the DataManagement subsystem where the player's high score is updated. This event occurs every time the player chooses to close the game upon which this data is first sent and stored after which the game closes.
- The **GameLogic** subsystem in the application layer receives the keypress event from the presentation layer and uses this data to calculate the move made by the player such as shoot or move the spaceship.
- The **DataManagement** subsystem in the data layer receives the HighScore from the presentation layer after the user selects to quit the game. This high score is then saved if it is greater than the previous high score. Furthermore, its function is to save data and hence does not correlate well to the GameLogic or the UserInterface layers. It also aligns with the 3-tier architecture we are using for the game since it does not need to use any method of the GameLogic or the UserInterface layers. Furthermore, it has room for expansion in the future when the whole game session could be potentially saved.
- The existing library that we will be using is **SDL 2** [4], the methods of this library can be used by any subsystem. This library will help us with input handling, image handling, video display, thread management and window management – such as toggle full screen or minimize, etc.

2.2 Hardware/Software Mapping

Although the original defender game was designed to work with a joystick and arcade controls, this version would run on a PC and use the keyboard arrow keys for movement of the spaceship along with keyboard keys for firing bullets and missiles. Each of these will be scripted during development and the player will not be able to modify these.

The SDL library is being used for development and this contains and InputManager class that would assist in key mapping and handling key inputs, however the control functionality that each keypress will be developed.

Currently, the game is compatible with any Linux distro and can be played in that. With slight modifications to the preprocessing directives (headers) in the source code files, e.g. deleting the SDL2 keyword from the preprocessor, it can run on Microsoft Windows too without any problems. The next step for the game would be making a mobile version of it so that it can be played in Android Platform smartphones too.

2.3 Persistent Data Management

The save file (used to store previous high score) will be stored locally on the player's machine (Linux or Windows) inside the user's system directory. The high score for the player will be stored as a text (.pref) file however the data would be obfuscated. Other game-specific data, such as sprites, sounds, animations, and game-specific information will be kept in the install directory of the game.

The libraries (SDL specifically) and other dependencies will also be added during installation.

2.4 Access Control and Security

The game will only be accessible locally on the player's machine. The game data (high scores) will be stored in an obfuscated format so that the player will not be able to understand the data inside it and what would change by modifying the data. If multiple users are using a machine the data will be stored specifically for their user account.

Furthermore, there will be no network connectivity hence only the users of the machine can access the stored data of the game. The game will use this obfuscated data at runtime to determine the high scores again. There can be only one player active per user account for the game.

2.5 Boundary Conditions

2.5.1 Installation

This is the package installation for the game, this would add all the required data such as sprites, sounds, animations, classes and libraries to the user's system. This is a one-time operation and the user will be able to play the game after this. This will also set up necessary directories. The necessary files such as that for high-score will be instantiated.

2.5.2 Initialization

This is the case when the game is run. This will also check if all necessary assets for the game are available. Furthermore, it will read from the data files of the game that contain a high score. The start menu will be displayed in this case.

2.5.3 Termination

This is the case when the user exits. The user may choose to end a session and quit from the start menu in which case the game will check if the player has exceeded the prior high score and if that is true the high score file is updated. This can also be done using the close button from the window frame in which case these checks will not be applied and the game will close after a warning is displayed that data will not be saved. If the process is killed by the user, these checks will not be applied and the game will close without any warning.

2.5.4 Failure

There are a couple of failures listed below which will occur most likely and how these failures are handled, or in some cases not handled due to different reasons.

2.5.4.1 Library Failure

In this scenario, the SDL2 library used in the game will fail to initialize a component or render a sprite or an object during game run-time. This results in the game not starting at all and prints a message pointing to the error related to it.

2.5.4.2 IO Failure

When there are some unexpected problems in the Windows/Linux file system while loading some resources during run-time and the game is running, those objects/resources will not be rendered for that cycle.

2.5.4.3 Process Termination

Process termination occurs whenever the game is quit manually by the user or when an unprecedented error in the OS occurs, e.g. manually closing the game from Task Manager, or graphics driver stops working. These scenarios are out of hand and cannot be handled.

2.5.4.4 Memory Not Available

Memory shortage is a failure that is caused by the environment the game is being played on. It might be caused by the OS or memory hardware failures. Therefore it cannot be handled.

2.5.4.5 Renderer Not Loaded

If there occurs a problem with the renderer, the sprites and objects on the screen cannot be rendered or updated, hence the game shows an error message and closes instantly.

3. Subsystem Services

3.1 UserInterface Subsystem

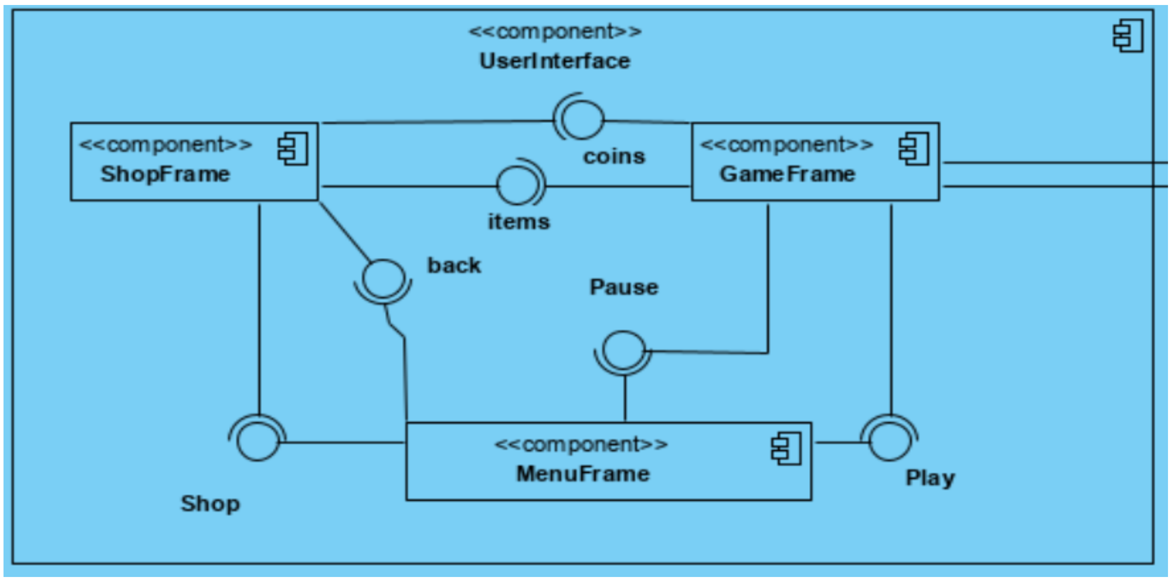

Figure 2 - UserInterface Subsystem

The UserInterface Subsystem provides the UI for the Defender: The components that the user interacts with. It is made up of three main components:
Note: In the descriptions below, provided interfaces are also referred to as “data”, this is only to ensure a better understanding of component dependencies for the reader.

3.1.1 GameFrame component

The GameFrame component communicates with the ShopFrame component by providing it the data regarding the number of coins which can be used in the ShopFrame by the user to purchase items, the GameFrame also communicates with the MenuFrame by providing the Pause data which, when selected by the user, pauses the game and displays the pause menu. In terms of receiving data, the GameFrame receives the Bought Items from the ShopFrame which the GameFrame implements in the game for the user and receives the Play data from the MenuFrame to continue or start a new game. We have decided to use the Singleton design pattern as well as the façade design pattern for the GameFrame. A single instance of the GameFrame will be active at any given time. We chose the façade design pattern to allow central control of other components (used in layers below), the related methods of other components can be called inside the GameFrame upon actions taken by the user (e.g. instead of delving into how each keyPress will be handled, a handleControls method is used to deal with user input and control spaceship - more details in design decisions section). Furthermore, the two outgoing lines represent highscore (provided to the DataManager component of the Data Layer) and

KeyPress(provided to the Spaceship component in ApplicationLayer). Since the higher layer can communicate with the lower layers, this follows convention.

3.1.2 ShopFrame component

The ShopFrame component is responsible for sending the Bought Items data to the GameFrame so the item's effects can be implemented in the gameplay and it receives the number of coins from the GameFrame which the user can spend while in the ShopFrame to purchase items. The ShopFrame also receives the Shop data from the MenuFrame which means that the user decided to open the ShopFrame. The ShopFrame can return to the MenuFrame by using back (backButton method). A single instance of the ShopFrame component is active at any given time

3.1.3 MenuFrame component

The MenuFrame is where the user will find themselves after pausing the game, returning from the ShopMenu or when they first initialize the game. Here the user can select to start the game or go to the shop for which the MenuFrame will send the Play data to the GameFrame to start the game or the Shop data to the ShopFrame to send the user to the shop. Similar to ShopFrame MenuFrame will also have one active instance at any time.

3.2 GameLogic Subsystem

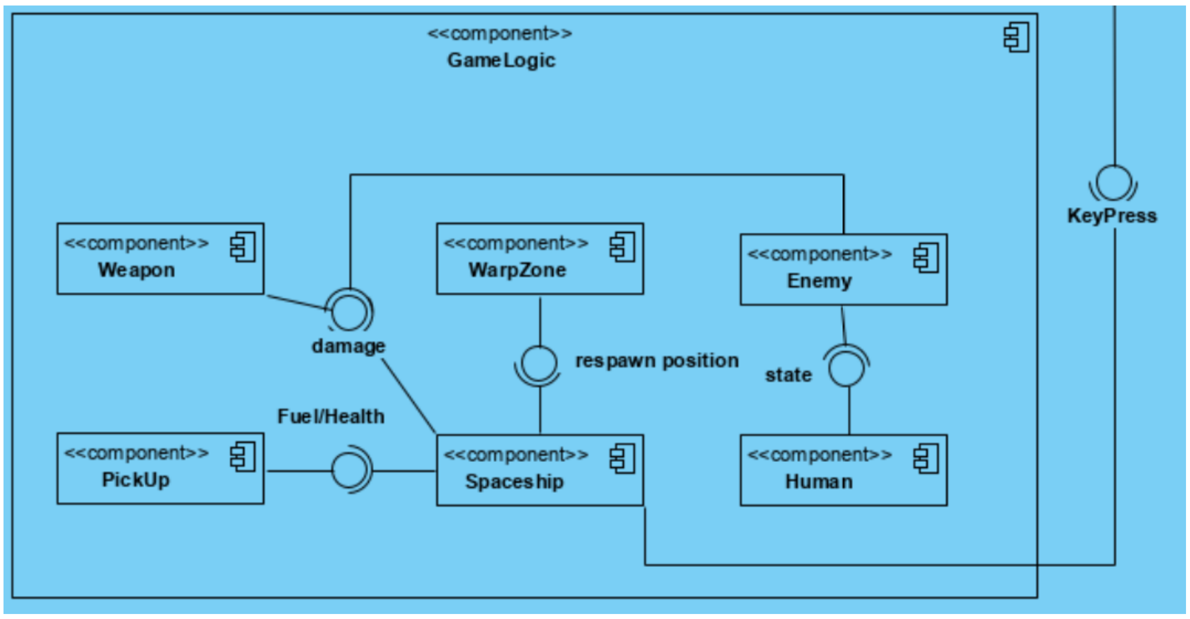

Figure 3 - GameLogic Subsystem

The GameLogic subsystem is responsible for game mechanics. It detects the actions taken by the user and accordingly calls the method corresponding to that action during gameplay. We will be applying the Strategy Pattern in this layer specifically for the Enemy Component since a lot of subclasses of enemies have related behavior (i.e. shooting) and centralizing such behavior. Using interfaces for it seems to be a good choice here, for it to reduce duplication, favor extendability and allow it to dynamically change the behavior of the component. The details of this choice are provided in the design decisions section (4.2).

The components and their relationships in this layer are as follows:

3.2.1 Weapon

Both the player and the enemy have access to weapons and they shoot at each other to deal with the damage. The Weapon component will provide the data Damage to both the SpaceShip and the Enemy components who will accordingly deduct health from their respective units.

3.2.2 Enemy

The Enemy component receives the Damage data from the Weapon component which it uses to reduce the health of a player in the game. The component receives the State data from the Human Component which informs the enemy if it has picked up or dropped the human, i.e. the current state of the human.

3.2.3 WarpZone

The warpZone component sends the respawn position to the Spaceship, the Spaceship after colliding with a warpzone in the game will be then sent to the new position by receiving this data.

3.2.4 Human

The Human component is responsible for sending the state date to the enemy component as the humans in the game will only have 4 states that are: they will either be on the ground, mid abduction by the enemy, saved by spaceship or falling.

3.2.5 Spaceship

The Spaceship component receives the Damage data from the Weapon Component, this damaging data is used to reduce the health of the spaceship when it gets hit by enemy fire. The Spaceship component also receives the Respawn position from the WarpZone component which it uses to place/warp the Spaceship to another location on the map and finally the Spaceship component receives the Fuel/Health data from the PickUp component which is used to restore the values for the health and fuel percentages of the spaceship. The GameFrame(from the Presentation layer) also provides the spaceship with KeyPress to determine whether to shoot or to move.

3.2.6 PickUp

The PickUp component sends the Health/Fuel data to the Spaceship component. This data is sent when the player collides with the fuel or health pickups in the game as they are meant to restore the spaceship’s stats.

3.3 DataManagement Subsystem

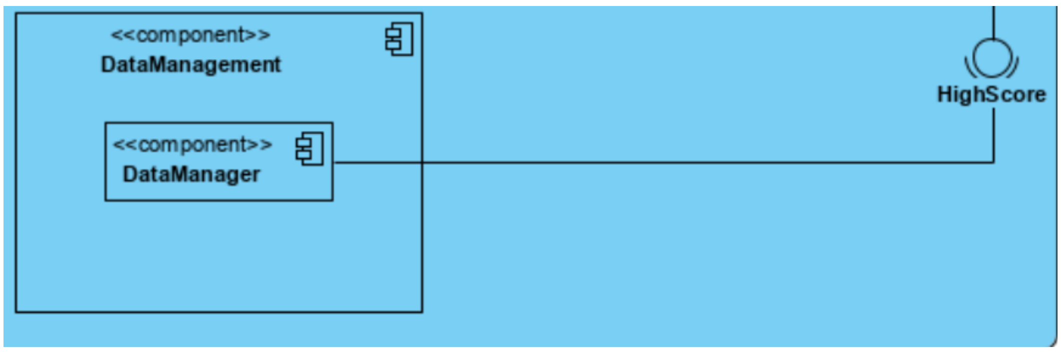

Figure 4 - DataManagement Subsystem

The DataManagement subsystem consists of only one component called the DataManager component, the purpose of the DataManager component is to store the current score of the player if it exceeds the preceding high score.
The reasons there is a separate layer/subsystem for DataManagement are the following:

- It's (current) primary function is to save and load data which does not correlate well to the GameLogic or the UserInterface layers. It also aligns with the 3-tier architecture we are using for the game since it does not need to use any method of the GameLogic or the UserInterface layers.
- This layer may be expanded to include session management, and in-game data recording, such as logs, hence a separate layer would be important for that.
- Separating it into a different layer allows for better coherence, since this way, each component in a specific layer is more interconnected and accomplishes similar functions, whereas this is isolated.

The datamanager, is provided with the high score data (coming from the presentation layer's GameFrame) to save it by writing to a file.

We will be encrypting the high score using the XOR cipher to provide a basic layer of security. The high score will be saved in the home directory of the game and if a file is already present it will rewrite the data onto that high score file, this file is later accessed to compare the current score with the high score.

4. Low-level Design

4.1 Trade Offs

4.1.1 Understandability vs Functionality

In designing our game, we have not included a lot of different features, this has been done on purpose to reduce the complexity of the game as the user may feel overwhelmed. Aside from the arrow keys for movement and the fire button we have included the shoot missile button, drop nuclear bomb button and the button to escape to the pause menu. This way our game has 8 buttons the user may interact with (z, x, c, the esc key, and the directional arrow keys) increasing the understandability of

the game and the user will be able to understand how to play the game over the course of just 1 game session.
This way we increase the understandability of the game but decrease the functionality.

4.1.2 Portability vs Development Time

Our game will be coded in the C++ language. We have chosen C++ to provide support for our game on both Windows and Linux operating systems hence increasing the portability of the game. Our game will also be easy to port onto android using the JNI (Java Native Interface). However, C++ has memory leaks so it will take more time for us to write the code without memory leaks. We did not choose Python as it would require serious reworking for portability.
Thus, we increased portability but also increased the development time by choosing C++.

4.1.3 Game Performance vs Readability

We will be using two threads to run our game which would increase the game performance by using one thread to render the UI and the second thread for dealing with the backend processes, we chose this approach as every CPU has its processing speed which affects the rendering time meaning that the UI and Backend will not be synchronized in one thread as our backend has processes such as generating enemies which are dependant on time which limit its speed. Therefore, by having two threads we can have the backend synchronized with the UI that will be rendered separately on another thread. However, we are using C++ which does not have an automatic garbage collector so we will have to deal will garbage collection ourselves resulting in lower readability of the code for the future developers.
Hence, we increase the game performance but decrease readability.

4.2 Final object design

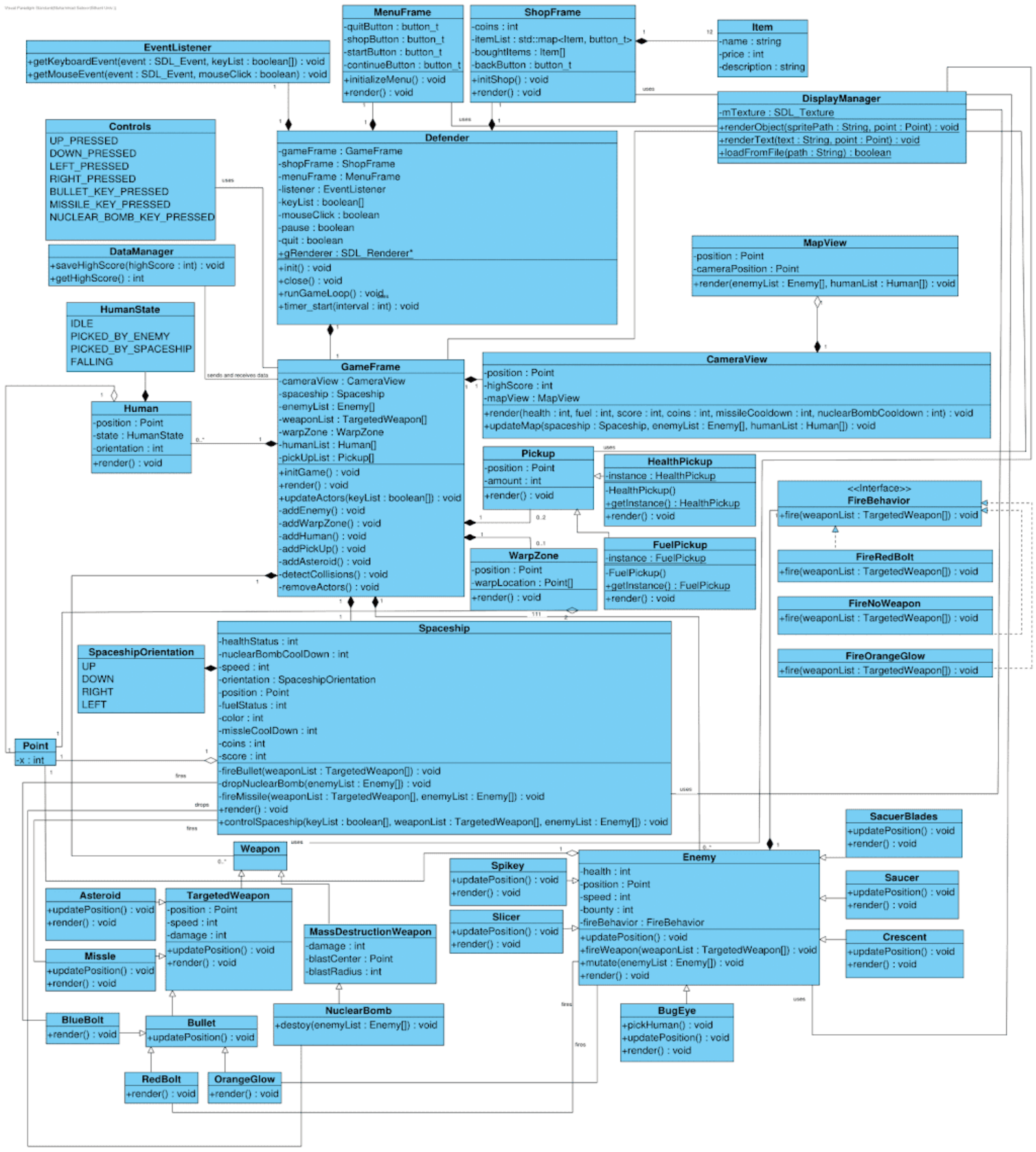

Figure 5 - Class Diagram

4.3 Design Decisions-Design Patterns

4.3.1 Strategy Design Pattern

Since in our class diagram, there are some classes whose functions do the same task but with different algorithms. For example, different enemies have different strategies for firing weapons. For example, BugEye and Spikey do not fire at all, Crescent will fire OrangeGlow bullets, and Saucer, SaucerBlades, and Slicer will fire RedBolt bullets.
We applied the strategy pattern to the firing behavior, which is common in all the enemies, to the Enemy class. We achieved this by creating FireBehavior interface which is implemented by FireRedBolt, FireOrangeGlow, and FireNoBullet. fireWeapon() method of the Enemy class uses the fire behavior according to its fireBehavior attribute.
Using the strategy pattern helps us to prevent duplication of code since RedBolt bullet is fired by 3 of the enemies. It also increases the extendability of the design since we can add different firing behavior later on. In the future, it may also help the developers to dynamically change the firing behavior of an enemy just by changing the fireBehavior attribute.
This also increases the maintainability of the code because altering one algorithm will not affect the other algorithm.

4.3.2 Façade Design Pattern

The basic controls of spaceship include movement controls (using arrows keys) and firing 3 kinds of weapons. This means that 7 keys are used to change the state of the spaceship. Handling these keys inside the Gameframe increases the complexity of the code. To decrease the complexity and encapsulate the control of the spaceship, we applied Façade design pattern to the GameFrame class which only needs to call controlSpacehip() of Spaceship class. This controlSpaceship() method will hide the complexity of the code handling the controls of the spaceship according to the key pressed.

4.3.3 Singleton Design Pattern

We used the singleton design pattern to increase the reliability of the program. During the game, we must have at most 1 HealthPickup and FuelPickup at a time. Therefore, applying singleton design to these classes will ensure that the second instance of HealthPickup or FuelPickup is not created if there is an instance of these classes already present in the GameFrame.

4.4 Class Interfaces

4.4.1 Defender Class

This class will contain the main game loop and will handle input from the user and provide those events to the class which requires those inputs. It also handles the timer of the game which will be used to perform different system activities in the game like the generation of pickups.

Attributes:

- **private GameFrame gameFrame:** This represents the game frame in which the main game and all of its components like spaceship, enemies, etc will be displayed.
- **private MenuFrame menuFrame:** This represents the menu frame that will be displayed when the game is started or when the user presses ESC during the game to pause it.
- **private ShopFrame shopFrame:** This represents the shop frame which will be accessible from the menu frame and will contain items that user can buy during the game.
- **private EventListener listener:** This is the event listener which listens for the user keyboard and mouse inputs.
- **private boolean[] keyList:** This is the list of boolean values used to represent which keys are pressed and which keys are not pressed.
- **private boolean mouseClick:** This boolean value is used to represent whether the mouse is clicked or not.
- **private boolean pause:** The boolean value is used to represent whether the user has paused the game or not.
- **private boolean quit:** The boolean value is used to represent whether the user has quit the game or not.
- **public static SDL_Renderer* gRenderer:** This object contains the rendering state of the game and will be used by renderable objects to render themselves on the screen.

Methods:

*all trivial setter and getter methods are assumed to be already there.

- **public void init():** The method will be called when the game will start and will be used to call the main game loop and start the event listener. It will also set up necessary frames (menu frame) and show it to the user.
- **public void close():** This method will be used to free any memory used by the game and then close the game.
- **private void runGameLoop():** This method is called from the updateActors() and it randomly spawns a warpZone on the screen if 15 seconds have passed from the last time the warpZone was used.
- **private void startTimer(int interval):** This method runs on a separate thread to start a timer to count “interval” seconds from the time of starting.

4.4.2 GameFrame Class

This class is one of the main classes which controls the working of the game and managing all of the actors being used in the game.

Attributes:

- **private CameraView cameraView:** This is the camera view object which is declared to update its position with the spaceShip and tell the renderer what to render on the screen.
- **private SpaceShip spaceShip:** This is the ship that will be rendered on the screen and the player will be able to control it.
- **private Enemy[] enemyList:** This is the list of the enemies that have been spawned in the game and are currently alive.

- **private Weapon[] weaponList:** This is the list of weapons that have been spawned on the screen whether by the spaceship or by the enemies.
- **private Human[] humanList:** The list of humans that are yet alive. The initial number of these humans will be 10.
- **private PickUp[] pickUpList:** The list of pickups that have been spawned on the map. Initially, this list is empty and will be filled eventually as the game proceeds.
- **private WarpZone warpZone:** This is the warpZone which will be able to transfer the spaceship to a random location if the spaceship enters it.

Constructors:

- **GameFrame():** This is the constructor of this class and is called when the class is instantiated. The initGame() method is called in this constructor which initializes the game and initializes all of the attributes to the initial values that have been specified above.

Methods:

*all trivial setter and getter methods are assumed to be already there.

- **public void initGame():** The initGame() method is called by the game loop when the game is supposed to start. This method calls all the methods necessary for starting the game that are: addEnemy(); addHuman(); The workings of these methods are further explained under their heading. Also, it instantiates a SpaceShip in the spaceShip attribute.
- **public void render():** This method is called from the game loop and it is tasked with setting the coordinates of the cameraView in such a way that the spaceShip is focused and the correct part of the game is rendered onto the screen and rendering all of the game objects on the window.
- **public void updateActors(boolean[] keyList):** This method is called from the game loop every iteration of the loop. Every time this method is called, all the objects in the game frame such as enemies, weapons are updated. This update includes updating their respective states and positions according to their functionality. This also calls the controlSpaceShip() method to handle any inputs from the player.
- **private void addEnemy():** This method is called from the updateActors(), initGame() and it spawns random enemies on the gameFrame and adds them to the enemyList. The number of enemies that are generated is dynamically decided to depend on the size of the resolution of the screen. On a normal 1080p resolution screen, this number would be calculated to 15. If an enemy is killed then to keep the number of enemies constant, another enemy is spawned to take its place somewhere on the map that is not the current screen.
- **private void addWarpZone():** This method is called from the updateActors() and it randomly spawns a warpZone on the screen if 15 seconds have passed from the last time the warpZone was used.
- **private void addHuman():** This method is called from initGame() and is used to spawn the humans in the game and then adding them into the humanList. The number of humans that are generated is dynamically decided by depending on the size of the resolution of the screen. On a normal 1080p resolution screen, this number would be calculated to 10.
- **private void addPickUp():** This method is called from the updateActors() and it randomly spawns a pickup on the screen depending on the situation of the fuel or health of the spaceship.
- **private void AddAsteroid():** This method is called from the initGame() and a new thread is created which basically controls the random spawning of asteroids in the game and then adding them

into the weaponList. This method is not called again as the method will be running independently of the main thread.

- **private void detectCollisions():** This method is called from the updateActors() and it checks the collision between any two objects present in the attributes of this class. That means that the weaponList, enemyList, spaceShip and warpZone, their positions are all checked and if for any of the objects another object is in its zone a collision between those objects occurs and an appropriate action is taken according to the situation, for example, the health of the spaceship is decreased and the bullet is destroyed if the collision is between an OrangeGlow and spaceship.
- **private void removeActors():** This method is called from the updateActors() and it checks all the objects present in the gameFrame. Any object that is not supposed to be present in the game, its destructor is called such as a bullet that goes out of the current screen.

4.4.3 DataManager Class

We have a DataManager class that is required to save the current score of the player if it is greater than the previous highScore to save the new highScore.

Attributes:

We do not have any Attributes in this class.

Constructors:

- **DataManager():** We do not perform any task in the constructor.

Methods:

- **public void saveHighScore(int highScore):** This method will be used to save a high score into a file in the user's home directory if a current high score is greater than what is already stored in the saved file. If the file is not present, the current high score will be saved in the file after creating it. The high score will be encrypted with XOR cipher and the result will be saved into the file. The high score from the file will be read using the getHighScore() method of the same class.
- **public int getHighScore():** This method will be used to load the high score from the user's home directory. If the save file is not present, it will return 0. Otherwise, it will read the save file and decrypt the data inside it with XOR cipher and the result will be returned.

4.4.4 MenuFrame Class

This class is related to the Menu of our game, which comes up whenever the game is opened initially and also when the EventListener detects the escape key because the user has pressed it. This includes the option for the user to start the game, quit the game, continue the game, or go to the shop.

Attributes:

- **private button_t quitButton:** This button is on the menu and when the user wants to quit the game can use it to stop playing the close the game.
- **private button_t shopButton:** This button is on the menu and when the user wants to open the shop and buy upgrades from the shop.
- **private button_t startButton:** This button is on the menu and when the user wants to start the game from scratch. Coins and his score is reset to zero when he starts the game.

- **private button_t continueButton:** This button is on the menu and when the user wants to resume his ongoing game.

Constructors:
- **MenuFrame():** Attributes, buttons, will be firstly initialized in the constructor.

Methods:

*all trivial setter and getter methods are assumed to be already there.
- **public void initializeMenu():** This method will initialize the looks and contents of the menu, which include the buttons, Start Button, Quit Button, Shop Button, Continue Button. It is called every time the user presses ESC key and the user will be redirected from GameFrame to MenuFrame.
- **public void render():** This method is used to render the MenuFrame on the screen.

4.4.5 EventListener Class

This class is the base class for getting any input from the user.

Attributes:

We do not have any attributes in this class.

Constructors:

- **EventListener():** Constructor will not perform any tasks/actions.

Methods:
- **public void getKeyBoardEvent(SDL_Event event, boolean[] keyList):** This method listens to any keyboard key the user pressed and then changes the state of the key pressed in this keyList.
- **public void getMouseEvent(SDL_Event event, boolean clicked):** This method listens to the clicks which are done by the user, e.g. any of the menu buttons are clicked with the mouse.

4.4.6 DisplayManager Class

Base class for rendering any object onto the screen. This is a static class with public static methods so any class in this project can access the render methods in this class without initializing its object.

Attributes:

- **private static SDL_Texture mTexture:** This is the texture required by SDL to render an image on the screen. When the loadFromFile() method is called, it creates a texture and sets this attribute to the texture created and then this mTexture is used in rendering onto the screen.

Constructors:

- **DisplayManager():** Sets mTexture to NULL.

Methods:

- **public static renderObject(string spritePath, Point point):** This method gets the path of the sprite that has to be rendered, and the point on the screen where it has to be rendered. It calls the loadFromFile() method and loads the image and then turns it into a texture finally rendering it onto the screen.
- **public static renderText(string text, Point point):** This method gets the text that has to be rendered, and the point where it has to be rendered. It turns the text into a texture and then renders it onto the screen.
- **public static loadFromFile(string spritePath, Point point):** This method gets the path of the sprite that has to be loaded and then the texture of that image is created and mTexture is set to that texture.

4.4.7 CameraView Class

This class holds the main screen of the game while the user is playing. It is updated as frequently as possible to maintain a smooth gameplay. This screen focuses on the spaceship, humans, and enemies which are in front of the spaceship.

Attributes:

- **private Point position:** This attribute holds the physical position of the cameraView on the computer screen.
- **private MapView mapView:** This is the whole map of the playing area and will be shown at the top of the screen as the mini-map. It will show all objects even if they are not in the camera focus.
- **private int highScore:** This is the score that is the highest that has been achieved on this game from the first time the game has been played.

Constructors:

- **CameraView():** The constructor initializes the position of the CameraView and focuses on the spaceship and its surroundings.

Methods:

*all trivial setter and getter methods are assumed to be already there.

- **public render(int health, int fuel, int coins, int score, int missileCountDown, int nuclearBombCoolDown):** This method gets the data that has to be displayed on the HUD of the screen and then displays it there with the help of the render methods from the display manager.
- **public updateMap(Spaceship spaceship, Enemy[] enemyList, Human[] humanList):** This method gets the humans, enemies, and spaceship and updates their locations on the mapView and then renders it.

4.4.8 ShopFrame Class

ShopFrame is responsible for the shop functionality of our game. Users can interact with the shop and buy upgrades.

Attributes:

- **private int coins:** Holds the number of coins the user has gained throughout the game and will be used to buy upgrades in the shop.

- **private map<Item, button_t> itemList:** All the items will be mapped to a button that can be pressed to buy that item. This will help us keep track of which item the player has requested to buy.
- **private item[] boughtItems:** Array of the items which are already bought by the user from the shop.
- **private button_t backButton:** This represents the back button which the user can press to go back to the pause menu.

Constructors:

- **ShopFrame()**: Constructor will initialize the attributes of the class, coins, itemList with all the items, and boughtItems to zero.

Methods:

*all trivial setter and getter methods are assumed to be already there.
- **public void initShop():** This method is called whenever the user presses the Shop button on the menu. It will show the shop items, already bought items, and the number of coins the user has.
- **public void render():** This method is called to update the shop frame on screen.

4.4.9 MapView Class

This class represents a small map which is located on top of the screen while gameplay. Since cameraView is focusing only on the spaceship and current enemies, mapView will show a bigger picture of the whole screen.

Attributes:
- **private Point position:** Position is a point that holds the position where the map should show in MapView.
- **private Point cameraPosition:** CameraPosition is responsible for storing the position of the MapView in the screen.

Constructors:

- **MapView()**: Constructor initializes the position and cameraPosition attributes accordingly.

Methods:

*all trivial setter and getter methods are assumed to be already there.
- **public void render(Enemy[] enemyList, Human[] humanList, Spaceship spaceship):** This method updates the view of the MapView every time there is a change in the screen, which is why it needs the info of enemies, humans and spaceship as arguments. render is called frequently because changes happen frequently in the game, e.g. spaceship moves, enemy moves are updated, etc.

4.4.10 Human Class

Human class is responsible for the humans in the game, and as one of the key features of the game, the user needs to protect these humans from being attacked and moved by the enemies.

Attributes:

- **private Point position:** Position is an SDL type point that holds the position of the human on the screen.
- **private bool state:** Humans in the game can have 2 states, either they are standing on the ground and are waiting to be picked up by the enemy, which the user needs to prevent, or the human is already picked up by the enemy, which the user needs to fire at the enemy and capture back the human and return it to safety.
- **private int orientation:** Orientation of a human shows whether the human is going up, being picked up and moved upwards by the enemy, or it is falling after the spaceship has shot down the enemy which had picked up the human.

Constructors:

- **Human():** Constructor initializes the position, state, and orientation of the human accordingly. All humans will start initially from the ground and their position and state will be set according to that.

Methods:

*all trivial setter and getter methods are assumed to be already there.

- **public void render():** This method is called to update the sprite of the human on screen.

4.4.11 WarpZone Class

Warp zones are one of the exciting features of the game in which the spaceship can enter the warp zone and exit from a different position on the screen, which helps the player escape difficult situations if they are stuck mid-game.

Attributes:

- **private Point position:** The position of a single warp zone on the screen is saved in this attribute.
- **private Point[] warpLocation:** This attribute includes the array/set of all warpzones which are included in the game.

Constructors:

- **WarpZone():** The constructor initializes the position of each of the warpzones and adds the total number of warpzones that are going to be in the game inside the warpzones[] array.

Methods:

*all trivial setter and getter methods are assumed to be already there.

- **public void render():** This method is called to update the sprite of the WarpZone on screen.

4.4.12 Point Class

Point class represents the position of any game feature, e.g. human, enemy, spaceship, warpzone, etc. This is an important class and has the coordinates of all the mentioned items on the screen.

Attributes:

- **private int x:** The x-coordinate of an item.
- **private int y:** The y-coordinate of an item.

Constructors:

- **Point():** The constructor initializes the x and y coordinates of the thing which it is used for.

Methods:

*all trivial setter and getter methods are assumed to be already there.

This class does not have any specific methods, except for the setter and getter methods.

4.4.13 SpaceShip Class

SpaceShip class represents the main spaceship the user is interacting with throughout the game. It is a base feature of the game and the user can control it with the keyboard, e.g. moving up, down, left and right, firing bullets, firing missiles, catching the falling humans, etc.

Attributes:

- **private int healthStatus:** The health of the spaceship is really important because after the spaceship takes damage from the enemy bullets and the health runs out, spaceship dies and the game ends. It is saved as an integer and starts from 100 and goes all the way to zero if the spaceship dies.
- **private int fuelStatus:** The spaceship has a fuel status feature which indicates how much longer the spaceship can stay alive and not go down. This feature is saved in this attribute as an integer, starting from 100 and going all the way to zero. The game ends if the spaceship does not pick up any fuel or buys fuel from the shop.
- **private int score:** This is the score of the player at any time. The score will be 0 at the start of the game and will increase quantitatively after every second only if the player is not dead.
- **private int coins:** This is the count of coins that the player has. These coins are initialized to 0 and are increased when enemies are killed.
- **private int missileCoolDown:** The spaceship can fire missiles in addition to bullets, but the missile cannot be shot at every instant, so there is a gap between each missile shot. This gap is saved as an integer in this attribute.
- **private int nuclearBombCoolDown:** This attribute relates to the dropNuclearBomb feature of the game, in which the user can remove all the enemies from the screen. This attribute will save time until the cool-down of the dropNuclearBomb feature, meaning the user can press the nuclearBomb button again.
- **private int speed:** The speed of the spaceship indicates how fast the spaceship is traveling and saved in this attribute.
- **private int orientation:** The orientation of the spaceship shows whether the spaceship is moving up or down the screen.
- **private Point position:** The position of the spaceship on the screen is saved in this attribute and is of type Point.
- **private int color:** Spaceships design is based on its color and the spaceship can have a bunch of different colors which the user can choose from and play the game with.

Constructors:

- **SpaceShip():** The constructor initializes all of the attributes of this class to the initial states, e.g. health and fuel is set to 100, color is set as the color chosen by the user, etc.

Methods:

*all trivial setter and getter methods are assumed to be already there.

- **private void fireBullet(TargetedWeapon[] weaponList)**: This method initiates the firing bullet situation in which the user presses the fire bullet keystroke and a bullet is fired at the enemy, may it hit the enemy or not.
- **private void clearScreen(Enemy[] enemyList):** This method will clear the whole screen from enemies and no enemy will be left on the screen. The parameter indicates the list of enemies that are currently in the game.
- **private void fireMissile(TargetedWeapon[] weaponList, Enemy[] enemyList):** As mentioned before, the spaceship has a missile option to fire with too, and this method will initiate the scenario in which the missile is being shot by the user.
- **public void controlSpaceship(boolean[] keyList, TargetedWeapon[] weaponList, Enemy[] enemyList):** This method controls the actions that the spaceship has to do such as moving and firing. The fire methods of the spaceship will be called from here. Also the score, coins and fuel will be updated from here.
- **public void render():** This method is used to update the sprite of the spaceship on the screen.

4.4.14 Weapon Class

Weapon class is the parent class for the different weapon types, it does not have any attributes for itself but its related child classes, weapon types' classes, will have the required attributes and methods.

Attributes:

This class does not have any attributes.

Constructors:

- **Weapon():** The constructor does not have any functionality as this class has no attributes.

Methods:

This class does not possess any methods.

4.4.15 MassDestructionWeapon Class

Mass Destruction Weapon is connected with the nuclear bomb functionality of the game, which eliminates all the enemies from the screen, not from the whole map though.

Attributes:

● **private int damage:** As the enemy triggers the Mass Destruction Weapon, the damage level of this weapon will depend on each enemy which is on the screen. The damage level will be different based on the enemies' health status, but either way, the health level of the enemies on the screen will be below zero, dead.
● **private Point blastCenter:** This attribute is the location of the center of the blast where the missile will be firing at.
● **private int blastRadius:** The radius of the blast of this missile is saved in this attribute to show how far away the missile affects enemies.

Constructors:

● **MassDestructionWeapon():** The constructor initializes the damage, blastCenter, and blastRadius based on the enemies which are present on the screen.

Methods:

Apart from the setter and getter methods, this class does not have any other methods.

4.4.16 TargetedWeapon Class

TargetedWeapon class represents the other types of missiles and bullets the spaceship or the enemy can fire at enemies. It is a parent class for Asteroid, Bullet, and Missile classes.

Attributes:
● **private Point position:** The position of the targeted weapon will be saved in this attribute.
● **private int speed:** Each targeted weapons bullet will have a speed that is saved in this attribute.
● **private int damage:** The damage that causes the enemies will be saved as an integer.

Constructors:

● **TargetedWeapon():** The constructor will initialize the position, speed, and damage of the targeted weapon accordingly.

Methods:

*all trivial setter and getter methods are assumed to be already there.
● **private void updatePosition():** This private method for this class will be called whenever the weapon shoots a bullet or a missile and the position of the bullet or missile will be updated accordingly.
● **public void render():** This method is used to update the sprite of the targetedWeapon on screen

4.4.17 Asteroid Class

Asteroids are random attacks not related to either the spaceship or the enemy and are spawned randomly by the system and are aimed towards the spaceship.

Attributes:

This class does not have any attributes.

Constructors:

- **Asteroid():** The constructor does not have any special functionality as the class does not have any attributes.

Methods:
- **public void updatePosition():** This public method for the asteroid class will update the position of the asteroid every time the asteroid is spawned by the system towards the spaceship.
- **public void render():** This public method for the asteroid class will be used to update the sprite of the asteroid on screen.

4.4.18 Bullet Class

Bullet is a child class of targetedWeapon and is used whenever the user wants to fire a normal bullet towards the enemy or when the enemy wants to fire a bullet towards the spaceship. The bullet class has subclasses which include the types of bullets the spaceship and enemies can shoot.

Attributes:
This class does not have any attributes.

Constructors:

- **Bullet():** The constructor does not have any special functionality as the class does not have any attributes.

Methods:
- **public void updatePosition():** This public method for the asteroid class will update the position of the bullet every time the bullet is fired by the user through the spaceship.

4.4.19 BlueBolt Class

This is a subclass of Bullet class and is fired by the spaceship.

Attributes:
This class does not have any attributes.

Constructors:

- **BlueBolt():** The constructor does not have any special functionality as the class does not have any attributes.

Methods:
- **public void render():** This method of the BlueBolt class will be used to update the sprite of the BlueBolt on screen.

4.4.20 RedBolt Class

It is a subclass of Bullet class and is fired by the enemy towards the spaceship and deals some damage to the spaceship.

Attributes:
This class does not have any attributes.

Constructors:

- **RedBolt():** The constructor does not have any special functionality as the class does not have any attributes.

Methods:
- **public void render():** This method of the RedBolt class will be used to update the sprite of the RedBolt on screen.

4.4.21 OrangeGlow Class

It is a subclass of Bullet class and is fired by the enemy towards the spaceship and deals some damage to the spaceship.

Attributes:
This class does not have any attributes.

Constructors:

- **OrangeGlow():** The constructor does not have any special functionality as the class does not have any attributes.

Methods:
- **public void render():** This method of the OrangeGlow class will be used to update the sprites of the OrangeGlow on screen.

4.4.22 Missile Class

The missile is a child class of TargetedWeapon class and is fired whenever the user triggers it too. It targets the closest enemy to the Spaceship.

Attributes:
This class does not have any attributes.

Constructors:

- **Missile():** The constructor initializes the missile itself.

Methods:

- **public void updatePosition():** This method updates the position of the missile after it is fired by the spaceship.
- **public void render():** This method of the Missile class is used to update the sprite of the missile fired on-screen.

4.4.23 NuclearBomb Class

NuclearBomb is one of the key features of the game and eliminates all the enemies from the screen. It is targeted at the enemies and is triggered when the spaceship fires a Mass Destruction Weapon.

Attributes:
This class does not have any attributes.

Constructors:

- **NuclearBomb():** The constructor initializes the NuclearBomb functionality of the game.

Methods:

- **public void destroys (Enemy[] enemyList):** This method destroys all the enemies which are visible on the screen, and the list of those enemies is given by the parameter enemyList.

4.4.24 Enemy Class

The enemy is the main opposition of the spaceship and is firing different types of ammo at the spaceship and at the same time is trying to abduct the humans from the ground and take them to space. Enemies can mutate and get fiercer and more powerful.

Attributes:

- **private int health:** All enemies have a health level which decreases if they are hit by the spaceship missiles or bullets. Health is an integer ranging from 0 to 100.
- **private int speed:** This attribute saves the speed at which the enemy can move.
- **private int bounty:** Whenever the spaceship manages to kill an enemy, the number of coins that will be awarded to the player will be saved in the bounty.
- **private Point position:** This attribute represents the coordinates of the enemy.
- **private FireBehavior fireBehavior:** This attribute indicates the type of weapon the enemy has.

Constructors:

- **Enemy():** The constructor initializes the given attributes of the class to the required measures.

Methods:

*all trivial setter and getter methods are assumed to be already there.

- **private void updatePosition():** This method updates the enemy’s position on the screen.

● **public void fireWeapon(TargetedWeapon[] weaponList):** This method triggers the firing ability of the enemies, and can fire different types of ammo depending on the weapons the enemy has in weaponList.
● **public void mutates (Enemy[] enemyList):** This method is called when the enemy reaches a point where it can mutate and become a more powerful enemy. The different enemies it can mutate to depends on the parameter enemyList.
● **public void render():** This method is used to update the sprite of the enemy object on the screen.

4.4.25 Spikey Class

It is one of the enemy types and this enemy moves in groups and will try to attack the player by getting close and exploding.

Attributes:
This class does not have any attributes.

Constructors:

● **Spikey():** The constructor initializes an instance of Spikey.

Methods:
● **public void updatePosition():** This method updates the position of the Spikey enemy after it is spawned in the game.
● **public void render():** This method of the Spikey class is used to update the sprite of the Spikey enemy on screen..

4.4.26 BugEye Class

It is a child class of Enemy class and is the enemy responsible for abducting and picking up humans from the ground.

Attributes:
This class does not have any attributes.

Constructors:

● **BugEye():** The constructor initializes an instance of BugEye.

Methods:
● **public void pickHuman():** This method is called when the BugEye picks up a human from the ground.
● **public void updatePosition():** This method updates the position of the BugEye enemy after it is spawned in the game.
● **public void render():** This method of the BugEye class is used to update the sprite of the BugEye enemy on screen.

4.4.27 Crescent Class

It is a child class of Enemy class and this enemy results after the mutation of weaker enemies and will only attack the player, this is the strongest type of enemy and shoots the OrangeGlow bullets.

Attributes:
This class does not have any attributes.

Constructors:

- **Crescent():** The constructor initializes an instance of Crescent.

Methods:
- **public void updatePosition():** This method updates the position of the Crescent enemy after it is spawned in the game.
- **public void render():** This method of the Crescent class is used to update the sprite of the Crescent enemy on screen.

4.4.28 Saucer Class

It is a child class of Enemy class and this is a low-level enemy that shoots red bullets at the player.

Attributes:
This class does not have any attributes.

Constructors:

- **Saucer():** The constructor initializes an instance of Saucer.

Methods:
- **public void updatePosition():** This method updates the position of the Saucer enemy after it is spawned in the game.
- **public void render():** This method of the Saucer class is used to update the sprite of the Saucer enemy on screen.

4.4.29 SaucerBlades Class

It is a child class of Enemy class and these are stronger versions of Saucers in terms of speed.

Attributes:
This class does not have any attributes.

Constructors:

- **SaucerBlades():** The constructor initializes an instance of SaucerBlades.

Methods:

- **public void updatePosition():** This method updates the position of the SaucerBlades enemy after it is spawned in the game.
- **public void render():** This method of the SaucerBlades class is used to update the sprites of the SaucerBlades enemy on screen.

4.4.30 Slicer Class

It is a child class of Enemy class and this is a low-level enemy that only attacks the player. It deals slightly more(+5) damage than the Saucer enemy.

Attributes:
This class does not have any attributes.

Constructors:

- **Slicer():** The constructor initializes an instance of Slicer.

Methods:

- **public void updatePosition():** This method updates the position of the Slicer enemy after it is spawned in the game.
- **public void render():** This method of the Slicer class is used to update the sprite of the Slicer enemy on screen.

4.4.31 Pickup Class

This class is responsible for the pickup feature implemented in the game, in which random health or fuel items are being dropped and later picked up by the spaceship. It is later inherited by two subclasses called HealthPickup and FuelPickup.

Attributes:

- **private Point position:** The position of each of the Pickup items is saved in this attribute.
- **private int amount:** This attribute saves the number of Pickups to be dropped in the game.

Constructors:

- **Pickup():** The constructor initializes the attributes of this class according to the game.

Methods:

Note: It has the trivial getter and setter methods

- **public void render():** This method is used to update the sprite of the Pickup object on screen.

4.4.32 HealthPickup Class

It is a child class for the Pickup class and is responsible for dropping the health refill item. This item is dropped randomly by the system.

Attributes:
- **private HealthPickup instance:** An instance of the HealthPickup class.

Constructors:

- **private HealthPickup():** The constructor initializes an instance of HealthPickup object.

Methods:
- **public HealthPickup getInstance():** This method is used to call the constructor of the HealthPickup class to return one instance of the HealthPickup class. If the object is Null it will return a new object otherwise it will return the already created object so that we do not have more than one instance.
- **public void render():** This method is called to update the sprite of the HealthPickup object on the screen.

4.4.33 FuelPickup Class

It is a child class for the Pickup class and is responsible for dropping the fuel refill item. This item is dropped randomly by the system.

Attributes:
- **private FuelPickup instance:** An instance of the FuelPickup class.

Constructors:

- **private FuelPickup():** The constructor initializes an instance of FuelPickup object.

Methods:
- **public FuelPickup getInstance():** This method is used to call the constructor of the FuelPickup class to return one instance of the FuelPickup class. If the object is Null it will return a new object otherwise it will return the already created object so that we do not have more than one instance.
- **public void render():** This method is called to update the sprite of the FuelPickup object on screen.

4.4.34 FireBehavior Interface:

This interface provides an unimplemented method to show different firing behaviors so that it can be used to extend the game in the future and the code does not have to be duplicated while adding different enemies.

Methods:
- **public void fire(TargetedWeapon[] weaponList):** This unimplemented method in the interface must be implemented by all of its children to show how the firing behavior of an object will work.

4.4.35 FireRedBolt Class

This class implements FireBehavior interface by implementing its fire method to fire RedBolt bullets.

Attributes:
This class does not have any attributes.

Constructors:

- **Pickup():** The constructor just creates an object of FireRedBolt class.

Methods:
- **public void fire(TargetedWeapon[] weaponList):** This method implements the fire method of its parent class to demonstrate the firing of RedBolt bullets.

4.4.36 FireOrangeGlow Class

This class implements FireBehavior interface by implementing its fire method to fire OrangeGlow bullets.

Attributes:
This class does not have any attributes.

Constructors:

- **Pickup():** The constructor just creates an object of FireOrangeGlow class.

Methods:
- **public void fire(TargetedWeapon[] weaponList):** This method implements the fire method of its parent class to demonstrate the firing of OrangeGlow bullets.

4.4.37 Item Class

Item class represents all the items that a player can buy from the shop.

Attributes:
- **private string name:** Items are distinguished by their names and the user can buy any of the items if he has enough coins.
- **private int price:** Each item has a price allocated to it. Users can buy that item if he has coins enough.
- **private string description:** A brief description of the item will be saved in this attribute to give details about the item.

Constructors:

- **Item():** Constructor initializes the attributes of the items of the shop. It will set the name, price, and description of each shop item.

Methods:

*all trivial setter and getter methods are assumed to be already there.

This class has no methods other than the getter and setter ones.

4.4.38 FireNoWeapon Class

This class implements FireBehavior interface by implementing its fire method so that it does not fire any weapon.

Attributes:
This class does not have any attributes.

Constructors:

- **Pickup():** The constructor just creates an object of FireNoWeapon class.

Methods:
- **public void fire(TargetedWeapon[] weaponList):** This method implements the fire method of its parent class to demonstrate no firing behavior of an object. This method will be empty.

4.4.39 HumanState Enumeration

This enum represents different states of a human in the game.

Attributes:
- **IDLE:** The state where a human is standing idly on the ground.
- **PICKED_BY_ENEMY:** The state where a human is picked up by a BugEye enemy and is going upwards.
- **PICKED_BY_SPACESHIP:** The state where a falling human is caught by the spaceship.
- **FALLING:** The state where a human is falling because the enemy who picked the human up is dead.

4.4.40 SpaceshipOrientation Enumeration

This enum represents different orientations of the spaceship in the game.

Attributes:
- **UP:** The orientation where the spaceship is moving upwards.
- **DOWN:** The orientation where the spaceship is moving downwards.
- **LEFT:** The orientation where the spaceship is moving towards the left.

● **RIGHT:** The orientation where the spaceship is moving towards the right.

4.4.41 Controls Enumeration

This enum represents different controls of the spaceship in the game.

Attributes:
● **UP_PRESSED:** The control represents that up key was pressed to move the spaceship upwards.
● **DOWN_PRESSED:** The control represents that up key was pressed to move the spaceship downwards.
● **LEFT_PRESSED:** The control represents that up key was pressed to move the spaceship towards the left.
● **RIGHT_PRESSED:** The control represents that up key was pressed to move the spaceship towards the right.
● **BULLET_KEY_PRESSED:** The control represents that the bullet key was pressed to fire the BlueBolt bullet from the spaceship.
● **MISSILE_KEY_PRESSED:** The control represents that the missile key was pressed to fire the Missile from the spaceship.
● **NUCLEAR_BOMB_KEY_PRESSED:** The control represents that the drop nuclear bomb key was pressed to drop a nuclear bomb from the spaceship.

5. Improvement Summary

The first iteration design report had mistakes regarding the use of vague and complex words that did not give precisely describe our aims and the implementation we hoped to have, in certain cases. These mistakes were fixed in the current iteration. More concise and to-the-point phrases and words are used to clarify the features of the game. Inconsistency in the formatting of the report was resolved. IEEE standard formatting for specification reports[5, p. 104] was followed in our design report to maintain consistency. The design patterns had mistakes previously and therefore all of the design patterns were revised and improved. In first iteration, we had a god class problem in our GameFrame class and we solved it by distributing its method to responsible classes in this iteration. Some of the particular design decisions which we had made previously were not explained well as to why this decision was taken, therefore appropriate explanation and future references for those design decisions were written in this iteration. Separate Enum classes were added for better code readability. System decomposition was further revised to include a more in-depth breakdown of layer elements and their interaction. The diagrams were revised and redrawn in Visual Paradigm and a better, more readable format was added since the first iteration had readability issues (details in the relevant sections available). Technical and game-related terms were used throughout the report, therefore a glossary at the end of the report was added and each term was explained briefly. In the end, the report was reviewed by each member of the group and spelling problems and minor inconsistencies were resolved and fixed.

6. Glossary

BugEye: These enemies are responsible for picking up humans. After abduction, they move to the top to mutate into Crescent.

Bullet: The spaceship will fire blue bullets which will damage the enemies. It has no cooldown when using it.

Crescent: This enemy results after the mutation of weaker enemies and will only attack the player.

Fuel pickup: The player will be able to utilize a fuel pickup which will restore the space ship’s fuel.

Fuel Upgrades: The player will be able to increase the fuel capacity by buying fuel tanks from the shop.

Health pickup: The player will be able to utilize a health pickup which will restore the space ship’s health.

Health Upgrades: The player will be able to increase the health capacity by buying health tanks from the shop.

Human: The humans on the ground will be abducted by enemies so that the enemy can mutate.

Missiles: The missile is a powerful weapon the player can use that targets the closest enemy and deals more damage than a bullet, it has a cooldown of 5 seconds.

Nuclear Bomb: This weapon will kill all enemies on screen for the player, it has a cooldown of 15 seconds.

Saucer Blades: These are stronger versions of Saucers in terms of speed.

Saucer: This is a low-level enemy that shoots bullets at the player.

Shop: The game will include a shop feature where the player can purchase upgrades.

Skin Mutation: The player will be able to change the skin colors of their spaceship in exchange for coins.

Slicer: This is a low-level enemy that only attacks the player.

Spikey: These enemies will try to attack the player by getting close to the spaceship and exploding.

Warpzone: Random warp zones generated on the map that will transport the spaceship(player) from one location to another.

References

[1] “CS319 - Group 2D - Project Analysis Report,” Google Docs. [Online]. Available: https://docs.google.com/document/d/1ho6j-uEJV7cxFchB4kYFe-pw97rxO_Uo1KD4Rhj6yz0/edit?ts=5dab252c#heading=h.waz2zrxivm6e. [Accessed: 03-Dec-2019].

[2] “Java Native Interface,” JNI APIs and Developer Guides. [Online]. Available: https://docs.oracle.com/javase/8/docs/technotes/guides/jni/. [Accessed: 03-Dec-2019].

[3] J. D, Banefane, Nate, Wesley, J. D, D. Bernard, Wolfos, Dan, L. Zimmerman, Kohlrak, Gavin, Jesse, Daniel, Juliano, Anta, John, A. Wittmann, Stu, SparkyNZ, E. Xavier, Krzyś, and Mkkui, “HowTo: SDL on Android part 1 - Setup,” DinoMage Games, 09-Jan-2015. [Online]. Available: http://www.dinomage.com/2013/01/howto-sdl-on-android/. [Accessed: 03-Dec-2019].

[4] “About SDL,” Simple DirectMedia Layer - Homepage. [Online]. Available: https://www.libsdl.org/index.php. [Accessed: 03-Dec-2019].