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. 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

The game is based upon a very simple concept and it must be easy to use to complement the simplicity. The game would not have any complex, cluttered screens. Will allow fast level loading and in-game processing, the mechanics will be kept simple and no complex concepts that cannot be grasped within the first 5 game runs will be added. The input will also be kept simple with a minimal number of keys to ease user access.

1.2.2 High Performance

Efficiency and high performance will be maintained due to the rudimentary nature of the game. C++ will be used and the mechanics, for instance, those for shooting and controlling will be written such that the most efficient data structure and algorithms are implemented. Furthermore, the least possible amount of memory will be used and memory leaks will be avoided.

1.2.3 Component Reuse

Components such as projectiles, ships (spaceship and aliens) etc. will be made such that they can be used for other cases, that may be similar or those that may arise in the future. The object-oriented nature of C++ will be key in accomplishing this since object classes can be used in more than one context with different properties.

1.2.4 Readability

Since multiple individuals are working on the software the written code must be readable. The programming paradigms will be followed for ease of readability. Comments for code sections and descriptions for other developed components will be maintained. This is also important for extendibility since new features will be built upon the already existing ones, readability must be maintained throughout development.

1.2.5 Rapid Development

Due to time constraints, it is important to have a rapid development of the game. The deadlines for the project need to meant hence each component must be developed and tested within the allotted time frame to prepare the deliverables on time.

1.3 Trade-Offs

1.3.1 Understandability VS Functionality

In designing our game, we have not included a lot of different functions, this has been done on purpose to reduce the complexity of the game. Aside from the arrow keys for movement and the fire button we have included the shoot missile button, shoot clear bomb screen button and the button to escape to the pause menu. This way our game has a minimal learning graph and everyone can play this game easily increasing understandability.

1.3.2 Development Time VS Portability

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 and we did not choose Python as it would require serious reworking for portability. Thus, we increased portability but also increased the development time.

1.3.3 Game Performance VS Efficiency

We will be using multiple threads to run our game which would increase the CPU load and reduce efficiency but will increase the game performance. For example, we have an enemy generator in the game which is meant to generate enemies in the game during runtime and does not depend on any other factors for its process so we will have a separate thread running the enemy generator making it independent from the game and increasing the game performance.

2. System Architecture

2.1 Subsystem Decomposition

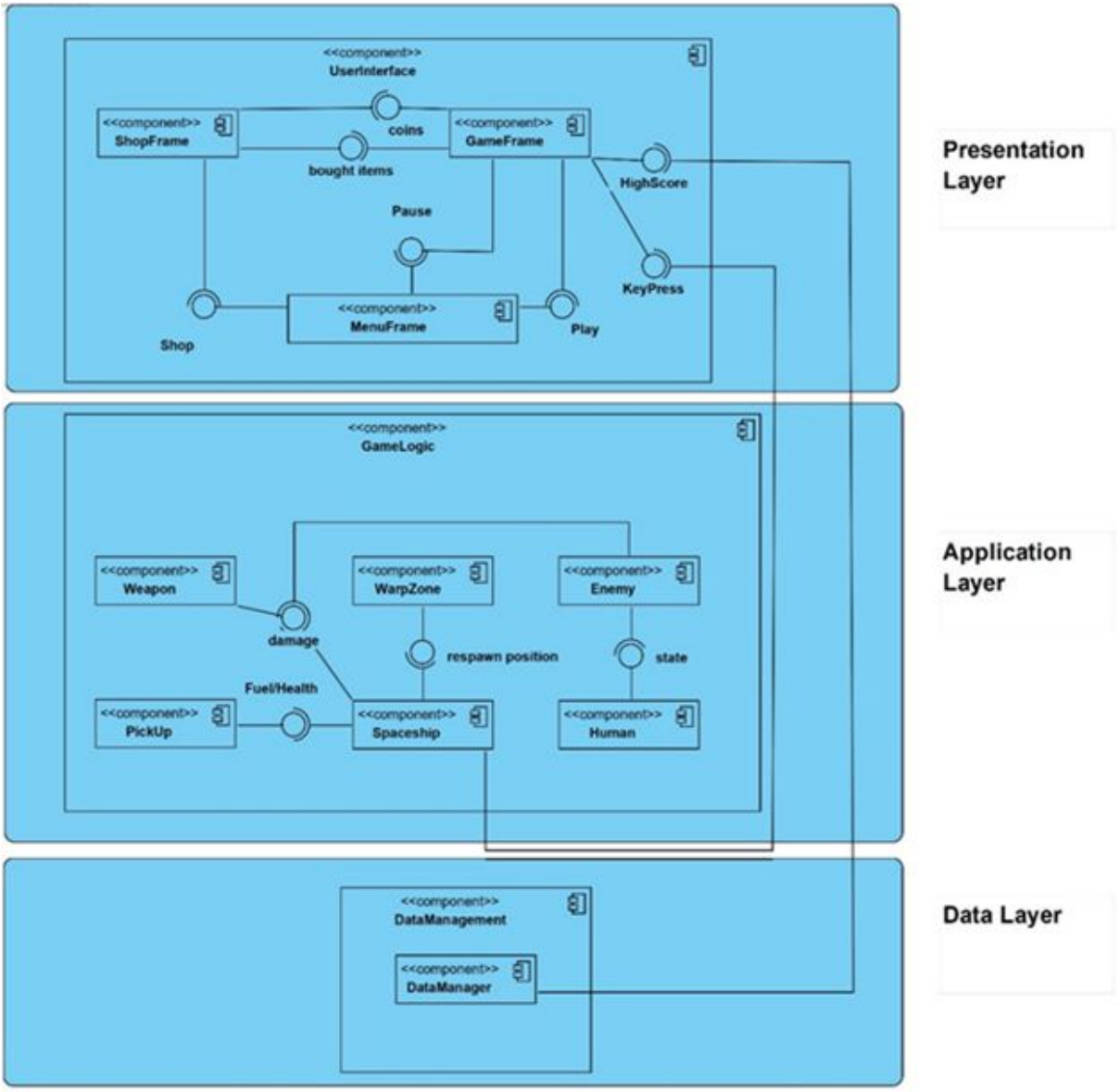

Figure 1 - Subsystem Decomposition

In this section, we will decompose our system into subsystems. Our aim with this decomposition was to remove 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.
- The existing library that we will be using is **SDL 2**, 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 some specified keys (Z for shooting bullets, X for shooting missiles and C for the clear screen bomb). Each of these will be scripted within 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.

For selections in main/pause menu and shop, mouse input, left-click, will be used to make selections and ClickListener will be used to handle these inputs.

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

This occurs when either one or more files are missing or corrupt or the game runs into some other unprecedented issue during run-time, this will result in a crash and will be handled by the OS. No files will be updated in this case.

3. Subsystem Services

3.1 UserInterface Subsystem

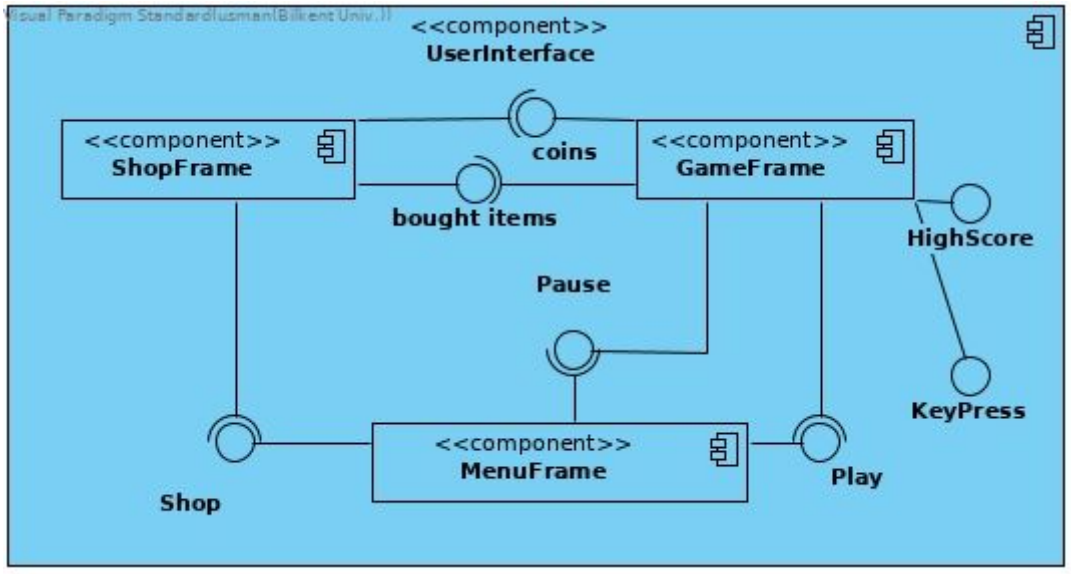

Figure 2 - UserInterface Subsystem

The UserInterface Subsystem provides the UI for the Defender. It is made up of three main components:

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. The Singleton design pattern was chosen to ensure that we only have a single instance of the GameFrame component to avoid performance issues and we chose the façade design pattern to allow hidden methods to be called inside the GameFrame upon actions taken by the user (e.g. A Collision will call the detectCollision method).

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. We have implemented the singleton design pattern for the shop frame to avoid performance issues as we do not want multiple instances of the ShopFrame to be open at the same time.

3.1.3 MenuFrame component

The MenuFrame is where the user will find themselves after pausing the game 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. We have chosen to use the Singleton design pattern for the MenuFrame to avoid having multiple instances of the MenuFrame which will help with performance and ensure smooth gameplay.

3.2 GameLogic Subsystem

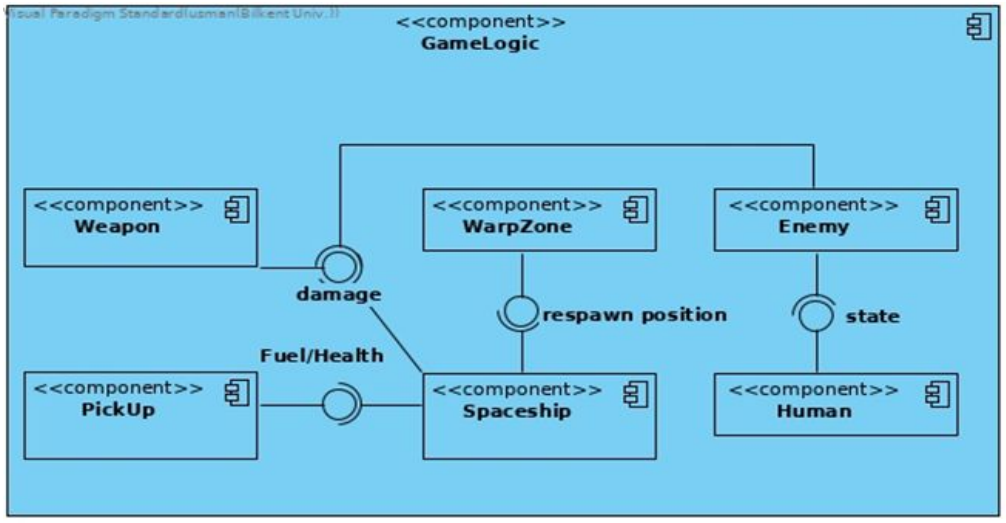

Figure 3 - GameLogic Subsystem

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 implementing the Strategy Pattern (also known as the Policy Pattern) for all our components in the GameLogic Subsystem, this is done because we wish to use several algorithms during run time for example: when the player shoots the enemy carrying a human we would want to call the dropHuman method and the destroyEnemy method. With the Strategy, design pattern calling codes during runtime will become more flexible and reusable making our gameplay smooth. The main components of the GameLogic subsystem are:

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 enemy units 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.

3.3.3 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.

3.3.4 Human

The Human component is responsible for sending the state date to the enemy component as the humans in the game will only have 2 states that are, they will either be on the ground or mid abduction by the enemy.

3.3.5 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.

3.3.6 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

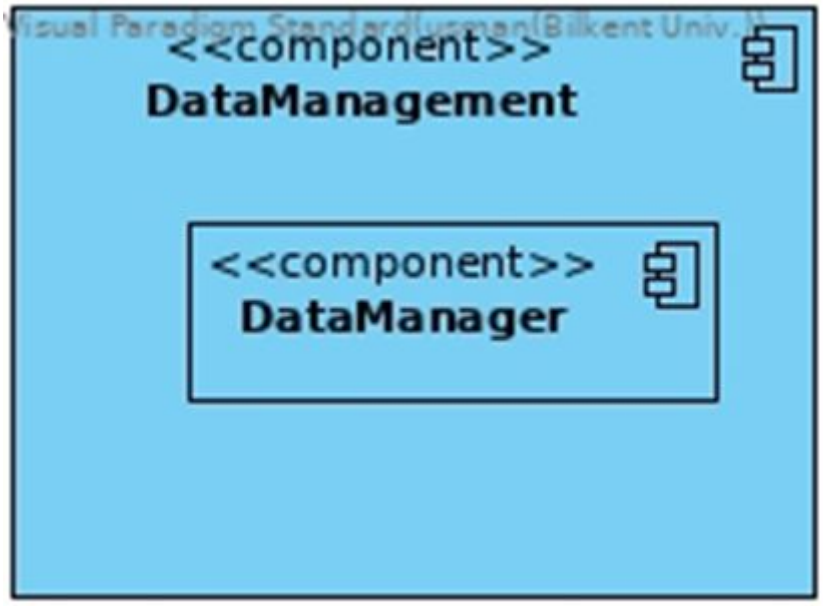

Figure 4 - DataManagement Subsystem

The DataManagement subsystem is consisting 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.

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 Final object design

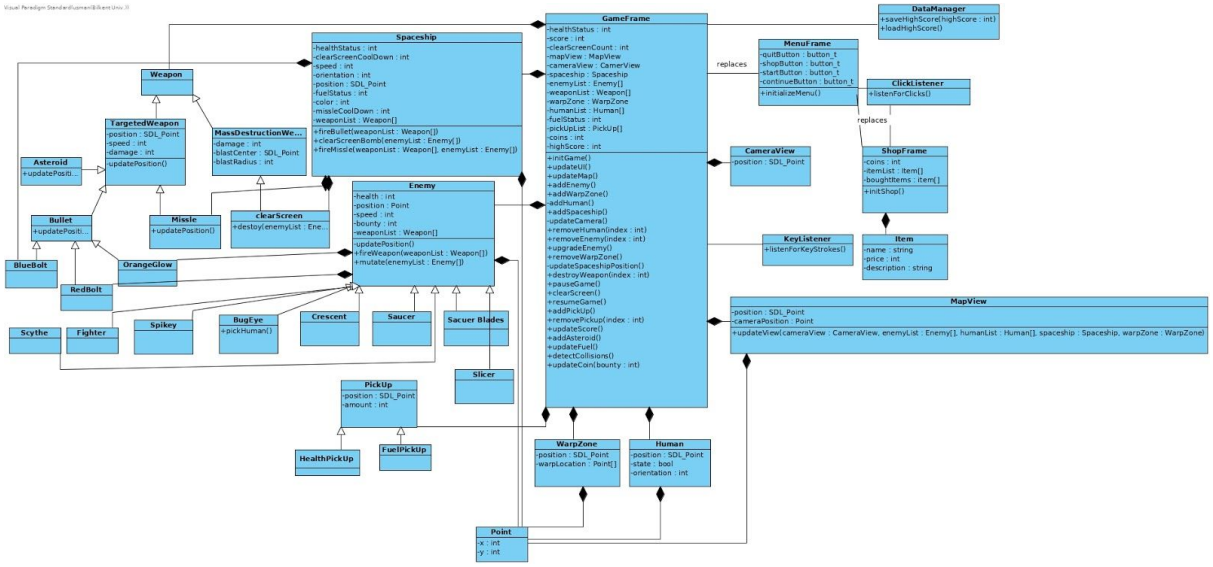

Figure 5 - Class Diagram

4.2 Design Decisions-Design Patterns

4.2.1 Façade Design Pattern

To decrease the complexity of the system, we have used the Façade design pattern. GameFrame will act as a Façade to encapsulate the multiple complex operations behind a single method. For example, the detectCollision() method in the GameFrame class will call all the necessary methods that should be the result of a specific collision. In layman's term, a collision between weapon and enemy will result in checking whether the health of the enemy has gone below the threshold of 0 or not. If it is below 0, the enemy is removed from the frame and coins and scores are updated for the player using updateCoin() and updateScore() methods. Since all of these operations are hidden behind a single detectCollision() method call, the GameFrame class acts as a Façade. One more such example is the collision between pick-up and the player's spaceship. If a collision is detected between these two, then the pick-up is removed from the frame using the removePickUp() method and then spaceship's health or fuel status is updated by calling its setter methods. In both of these cases, GameFrame communicates with the relevant classes within the system to change their attributes or perform their operations.

Using this design pattern helps us reduce the complexity of the system while increasing the maintainability of the system by decreasing the coupling between the classes.

4.2.2 Singleton Design Pattern

We used the singleton design pattern to increase the reliability of the program. Some classes should be instantiated only once throughout the game. For example, GameFrame, ShopFrame, MenuFrame, MapView, and CameraView. There should be only one instance of these classes present at any time in the game. If there is more than one instance of these frames, it can cause performance issues and can cause unexpected behaviors due to different objects communicating with different instances of these classes. By applying a single design pattern to these classes, we can overcome these issues

4.2.3 Strategy Design Pattern

Since in our class diagram, there are some classes whose functions do the same task but with different algorithms. For example, Boss enemies (Scythe and Fighter) will target the spaceship with RedBolt bullets if the health threshold of the spaceship is greater than 20. Otherwise, they will target the spaceship with OrangeGlow bullets which have more damage. Since this decision is to be made at runtime using the healthStatus of the spaceship, the bullet firing mechanism must be changed for the boss enemies. Using the strategy design pattern allows us to do exactly that. It encapsulates the two algorithms in the fireWeapon() method of the Scythe and Fighter classes. This increases the maintainability of the code because altering one algorithm will not affect the other algorithm.

4.3 Class Interfaces

4.3.1 GameFrame Class

This can be presumed to be the center point of our game as all of the classes have relations with this class and they communicate with each other through this class. This class is a part of our UI layer and thus will control what is displayed on the screen and will handle all of the commands given by the user during the game.

Attributes:

- **private int healthStatus:** This is the health of the spaceship at any given time in the game and its initial value is 100. It will be deducted according to the damage received to the plane. The health is declared here to show it on the screen.
- **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. The score is declared here to show it on the screen.
- **private int clearScreenCount:** This is the count of the deadly screen bombs that the player can explode. They will be initialized to 3 at the start of the game and will reduce only if the player fires them. They are declared here to show the remaining value of the bombs on the 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 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 int fuelStatus:** This is the fuel status of the spaceship and will show the value of the fuel remaining at any given time. The initial value of the fuel is 100 and reduces by a value of 0.5 after every second that the spaceship is not destroyed.
- **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 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 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:

- **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 constructor. This method calls all the methods necessary for starting the game that are: addSpaceShip(); addEnemy(); addWarpZone(); addHuman(); addPickup(); addAsteroid(); The workings of these methods are further explained under their own heading.
- **public void updateUI():** This method is called from the game loop and it has the job of calling the updateCamera() method and rendering all of the game objects on the window.
- **public void updateMap():** This method is called from the game loop and it controls what will be rendered on the map at the top of the screen. The map does not show pickups and such as it only shows the player, enemies, and humans.
- **public void addEnemy():** This method is called from the initGame() and a new thread is created which basically controls the random spawning of enemies in the game and then adding them into the enemyList. This method is not called again as the method will be running independently of the main thread.
- **public void addWarpZone():** This method is called from the initGame() and a new thread is created which controls the random spawning of warp zones in the game. This method is also not called again after the initial call as it will run on its thread.
- **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.

- **public void addSpaceship():** This method is called from initGame() and instantiates a SpaceShip in the spaceShip attribute. This spaceship is then controlled by the user.
- **private void updateCamera():** This method is called by the updateUI() method and 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.
- **public void removeHuman(int index):** This method is called by the detectCollisions() method when the enemy is colliding with the human and the top of the screen or if the human suddenly falls from the sky and collides with the land. In any case, the index of the human fallen is given and it is removed from the humanList.
- **public void removeEnemy(int index):** This method is called by the detectCollisions() method when the player or a bullet collides with the enemy and its health less than or equal to 0.
- **public void upgradeEnemy():** This method is called by the detectCollisions() method only when the last human is removed from the humanList. This method takes all of the enemies in the enemyList and calls their mutate() method.
- **public void removeWarpZone():** This method is called by the detectCollisions() method when the spaceShip collides with the warpZone and the spaceShip is teleported to its new position. This method then removes the warpZone from the game until addWarpZone() adds it again.
- **public void updateSpaceShipPosition(int direction):** This method is called by the game loop when the keyListener detects some input from the user. This method then calls the setPosition() method of the spaceShip and updates its current position by the speed of the spaceShip.
- **public void destroyWeapon(int index):** This method is called by the detectCollisions() method and it is only called when a weapon collides with anything. The weapon at the given index is removed from the weaponList.
- **public void pauseGame():** This method is called from the game loop and is only called when the user presses the pause button. All of the individually running threads are paused.
- **public void resumeGame():** This method is called from the game loop and is only called when the user presses the resume button. All of the individually running threads are resumed.
- **public void clearScreen():** This method is called from the game loop and is called when the game is paused so that white screen could be rendered over the currently rendered objects.
- **public void addPickUp():** This method is called from the initGame() and a new thread is created which controls the random spawning of pickups in the game near the player. The pickups are added to the pickUpList. This method is also not called again after the initial call as it will run on its thread.
- **public void removePickUp():** This method is called by the detectCollisions() method and it is only called when a pickup collides with the spaceship. The pickup at the given index is removed from the pickUpList.
- **public void updateCoins(int bounty):** This method is called from the detectCollisions() method and it is called if the collision ends up removing an enemy. The amount of the update in the coins will be equal to the bounty of the removed enemy.
- **public void updateScore():** This method is called from the game loop and if the health of the spaceship is greater than 0, 1 score will be added to the current score.
- **public 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.

- **public void updateFuel():** This method is called from the game loop and if the health of the spaceship is greater than 0, 1 score will be subtracted from the current fuelStatus.
- **public void detectCollisions():** This method is called from the game loop 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 method is called according to the situation. All of the methods that can be called from the detectCollision are explained above and in each of them, their calling condition is also mentioned.

4.3.2 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 loadHighScore() method of the same class.
- **public int loadHighScore():** 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.

4.3.3 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 keyListener detects the ESC 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 continue 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.

4.3.4 ClickListener Class

This class is the base class for getting the button clicks from the user's mouse.

Attributes:

We do not have any attributes in this class.

Constructors:

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

Methods:

- **listenForClicks():** This method listens to the clicks which are done by the user, e.g. any of the menu buttons are clicked with the mouse.

4.3.5 KeyListener Class

Base class for listening to the keystrokes pressed by the user on the keyboard.

Attributes:

We do not have any attributes in this class.

Constructors:

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

Methods:

- **listenForKeyStrokes():** This method listens to the keystrokes/key presses done by the user and acts accordingly, e.g. User presses ESC key and the menu is opened for the user.

4.3.6 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 SDL_Point position:** This attribute holds the physical position of the cameraView on the computer screen.

Constructors:

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

Methods:

This class does not have any methods except for a single setter and a single getter for the position.

4.3.7 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 item[] itemList:** Array of all items available in the shop to be bought by the user from the shop.
- **private item[] boughtItems:** Array of the items which are already bought by the user from the shop.

Constructors:

- **ShopFrame()**: Constructor will initialize the attributes of the class, coins, itemList, 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.

4.3.8 Item Class

Item class is necessary for the shop because it includes all the shop items.

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.

4.3.9 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 SDL_Point position:** Position is an SDL type 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 updateView(CameraView cameraView, Enemy[] enemyList, Human[] humanList, Spaceship spaceship, WarpZone warpZone):** 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 cameraView, enemies, humans, spaceship, and warpzone as arguments. UpdateView is called frequently because changes happen frequently in the game, e.g. spaceship moves, enemies move, warpzones are updated, etc.

4.3.10 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 SDL_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.
Human class does not have any specific methods, except for the setter and getter methods.

4.3.11 WarpZone Class

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

Attributes:

- **private SDL_Point position:** The position of a single warp zone on the screen is saved in this attribute.
- **private Poinit[] warpzones:** 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.

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

4.3.12 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.

4.3.13 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 clearScreenCoolDown:** This attribute relates to the clearScreen 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 clearScreen feature, meaning the user can press the clearScreen 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 SLD_Point position:** The position of the spaceship on the screen is saved in this attribute and is of type SDL_Point.
- **private int fuel:** 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 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.
- **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 Weapon[] weaponList:** The spaceship has a variety of different weapons that have different shooting abilities. This is saved in this array of weapons.

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.

- **public void fireBullet(Weapon[] 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. The parameter indicates the number of bullets the player has shot with the specified weapon.
- **public 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 on the screen and need to be eliminated.
- **public void fireMissile(Weapon[] 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. It includes the enemyList because the missile can damage not one but multiple enemies at the same time.

4.3.14 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.

4.3.15 MassDestructionWeapon Class

Mass Destruction Weapon is connected with the clearScreen 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 SDL_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.

4.3.16 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 SDL_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.

4.3.17 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.

4.3.18 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.

4.3.19 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:
There are no methods associated with this class.

4.3.20 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:
There are no methods associated with this class.

4.3.21 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:
There are no methods associated with this class.

4.3.22 Missile Class

The missile is a child class of TargetedWeapon class and is fired whenever the user triggers it too.

Attributes:
This method 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.

4.3.23 ClearScreen Class

Clear Screen 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 method does not have any attributes.

Constructors:

- **ClearScreen():** The constructor initializes the clearScreen 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.

4.3.24 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 Weapon[] weaponList:** This attribute indicates the different types of weapons 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(Weapon[] 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.

4.3.25 Scythe Class

It is a child class of Enemy class and is one of the two boss-type enemies. It appears occasionally and will fire bullets at the spaceship.

Attributes:
This class does not have any attributes.

Constructors:

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

Methods:
This class does not have any specific methods.

4.3.26 Fighter Class

It is a child class of Enemy class and is one of the two boss-type enemies. It appears occasionally and will fire bullets with more damage at the spaceship. It can maneuver while firing at the spaceship.

Attributes:
This class does not have any attributes.

Constructors:

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

Methods:
This class does not have any specific methods.

4.3.27 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:
This class does not have any specific methods.

4.3.28 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.

4.3.29 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.

Attributes:
This class does not have any attributes.

Constructors:

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

Methods:
This class does not have any specific methods.

4.3.30 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:
This class does not have any specific methods.

4.3.31 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 Saucer Blades.

Methods:
This class does not have any specific methods.

4.3.32 Slicer Class

It is a child class of Enemy class and this is a low-level enemy that only attacks the player.

Attributes:
This class does not have any attributes.

Constructors:

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

Methods:
This class does not have any specific methods.

4.3.33 PickUp Class

This class is responsible for picking up feature of 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 SDL_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:

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

4.3.34 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:
This class does not have any attributes.

Constructors:

- **HealthPickUp():** The constructor initializes an instance of Health refill item.

Methods:
This class does not have any specific methods.

4.3.35 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:
This class does not have any attributes.

Constructors:

- **FuelPickUp():** The constructor initializes an instance of Fuel refill item.

Methods:
This class does not have any specific methods.