Bilkent University

Department of Computer Engineering

CS 319 - Object-Oriented Software Engineering

CS319-2F-DE: Defender

Iteration 2 Design Report

Group Members

- Büşra Ünver
- Celal Bayraktar
- Javid Haji-zada
- Samir Süleymanlı
- Selen Uysal

Instructor: Eray Tüzün

Teaching Assistant: Alperen Çetin

Project Analysis Report

Dec 1, 2019

Table of Contents

1. Introduction

1.1 Purpose of the System

Defender is a desktop arcade game which proposes to defend the spaceship that went on a space mission and save astronauts from enemies in dangerous space conditions. The desktop version of the game will give players the opportunity to enjoy this nostalgic arcade game with fancier, and innovative features. We will offer users several distinguishable features one of which is letting players to challenge and compete with each other in two-players endless mode. Other new features include shop, where the player can purchase different visual versions of the spaceship and also, a background theme. Aim of the project is to implement Defender with a different perspective and features.

1.2 Design goals

1.2.1 Ease of Use

Our main goal during the implementation will focus on creating an “easy to understand and play” game. On this perspective, the player should be able to easily use the game interface and understand the main gameplay. To achieve this, we will make the low-level design in a way that the sizes and places of the screen objects and their interactions are intuitive. The requirements for these aims are as follows:

- The game will include “How To Play” screen that describes the gameplay, key-bindings, the modes, monsters and detailed information about how in-game scores and coins are gained.
- There will be a “Story Mode” that will simulate all of the features of the game thoroughly. A player will not be able to play other modes without completing this tutorial throughout “Story Mode”.

1.2.2 Efficiency

The game will be designed to be compatible with poor hardware specifications. We aim to minimize the time and space complexities of our algorithms to improve game performance to achieve higher FPS rate. The following requirements are planned to be met.

- FPS will not drop below 60.
- The executable file should not consume more than 500 KB of Storage.
- The program should not consume more than 512 MB of RAM.

1.2.3 Reliability

We plan to ensure that a player will not experience an unexpected occurrence like loss of data, sudden crashes or problems with gameplay. To achieve this goal we will add necessary boundary checks for user key inputs and I/O operations. By this, our game will be reliable enough to meet the following specifications:

- Previous scores (Highest Scores) data should not be lost.
- User's coins and purchased items will not be lost.
- Game will not crash on excessive button press.

1.2.4 Extendibility

For this project, to be able to improve the game afterwards, we aim to make our game extendible. To achieve this aim we will group related classes into different subsystems. This will also help us to develop a system that has high cohesion and low coupled classes. In order to achieve this division, we will use several design patterns which are explained in *Section 4.3*.There will be no conflict between components of game after we make changes with the user feedback and comments.

In addition, the extendibility requires the source code to be readable and understandable. For this we will add rich comments, and give the variables and the classes meaningful names which implies their functionalities.

2. High-level Software Architecture

2.1 Subsystem decomposition

In order to achieve our design goals, we need to have strong decomposition of our system. Each subsystem consists of strongly dependent classes. For example, we have user interface subsystem that includes Game Components and Screens as well. Each of these packages have their own common rendering methods corresponding to components and screens. Hence, our system has higher cohesion. As we use MVC pattern, our controllers will be our Managers. These managers will control screen navigation flow, game object rendering, and game data saving. This type of subsystem decomposition decision also helped us to easily distribute the implementation work between our group members.

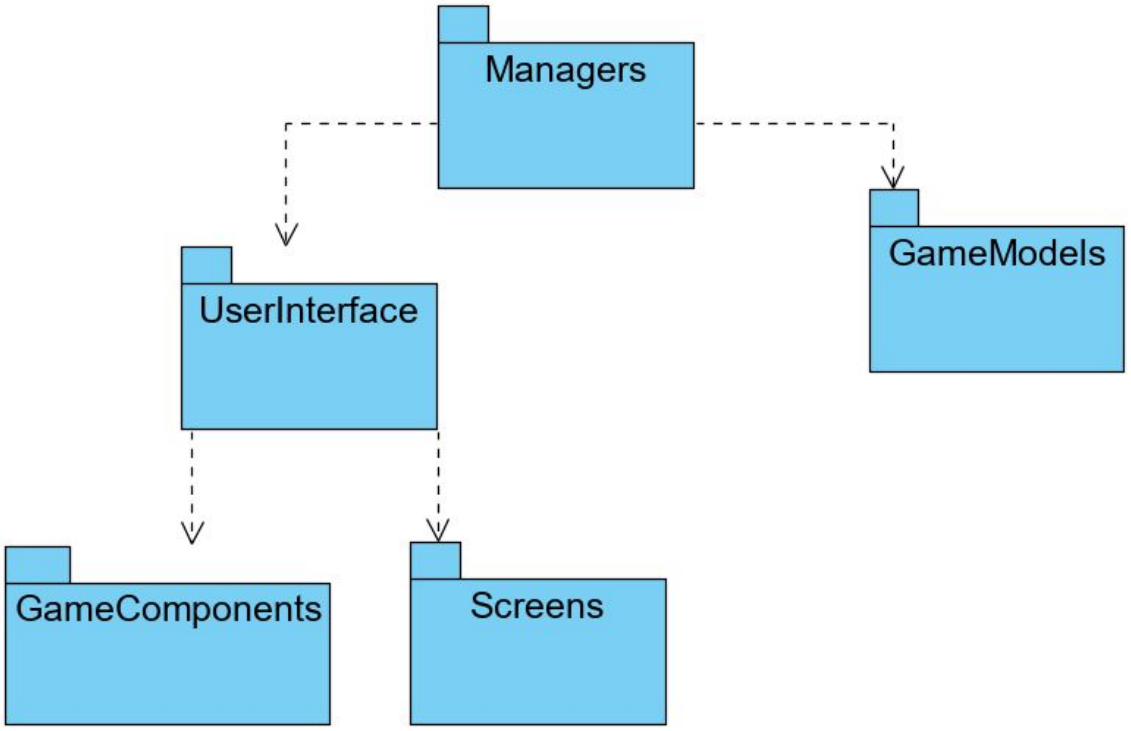


Figure 1: Subsystem dependencies

Our system will include 3 main subsystems. These subsystems and their dependencies are shown in *Figure 1.* Detailed explanation of each subsystem is given below.

2.1.1 Managers

Managers are the main subsystem to control game flow. Managers will deal with screen navigation and data flow, screen transitions, game logic, displaying components on the screen.

2.1.2 User Interface

User Interface will include *GameComponents* and *Screens*. The *Screens* subsystem encapsulates all of the screen classes. *GameComponents* encapsulates the objects to be displayed on the screens. These subsystems will include only objects related to game display, there will not be any logic about game play.

2.1.3 GameModels

This subsystem is planned to be independent from other subsystems. It will be used to store all of the static data related to game and its current state while a user plays it. We call this subsystem independent since it is the core model of the game logic.

2.2 Hardware/software mapping

Our system will not need additional hardware requirements. Software will be mapped into hardware through standard libraries. We will implement the game using Java. This implementation will ensure that our game will be platform-independent and Java provides strong OOP tools. The mappings on hardware devices will be as follows:

- Screen display; JavaFx will be used which will handle display rendering. The game graphics will be mapped to monitor by JavaFx's hardware interface.
- Sound output; will be provided by javax.sound. Thus, our music and sound effects will be mapped into audio device through this library.
- Keyboard inputs; will be listened by JavaFx's key listeners. Key events are mapped to keyboard through JavaFx' hardware capabilities.

2.3 Persistent data management

Highest Scores, items, coin, sound settings and complete, incomplete game modes related data of the game needs to be accessed whenever users wants to reach them. This data will be stored under the game folder under a folder named Defenders. There will be a separate folder for each data type. There will be sound settings and game-mode related data as well. The folder structure will be as below:

- Game mode data: ${HOME}\Documents\Defenders\game_mode.dat
- Sound Settings: ${HOME}\Documents\Defenders\sound_settings.dat
- High Scores: ${HOME}\Documents\Defenders\high_scores.dat
- Items data: ${HOME}\Documents\Defenders\items.dat
- Coin Data: ${HOME}\Documents\Defenders\coin.dat

2.4 Access control and security

The game will not have any online database related requirements. So, we will use only local storage, thus we will not implement any access control. Also, we will not need extra security concerns.

2.5 Boundary conditions

2.5.1 Application setup

Defenders, due to its low RAM and Storage requirements, there will not be any setup file or installation of any other third party extensions to install the game. As mentioned in *Section 2.3* any required in-game data will be fetched from Defenders folder. *SplashScreen* will firstly welcome the players and then *MainMenuScreen,* that provides main navigation flow of the game, will be displayed.

2.5.2 Terminating the Application

We are planning to place *Exit Game* button on *MainMenuScreen* which will let players to easily close the game. We will also have *"Exit to Desktop"* button on Pause Pop-up inside the game. This will ensure that a player can quit the game when he is playing any of three modes.

2.5.3 I/O exceptions

The only possible input in Defenders will be placed inside a Pop-up dialog to get player's name when he presses *Play* button. We will set maximum and minimum length to the input field in order to ensure that player enters a valid name and also prevent him/her from entering a very long name. Our system may have problems related to game data files. For example, if those files are deleted, game will be basically on "first-time installed" state. However, this will not create any crash in the game.

2.5.4 Critical Errors

Any critical error that leads to collapse of the application will result in score and coin loss if the crash occurs while the player plays any mode. In any other case, there will not be any data loss. After collapse, user can recover the game with regular procedure.

3. Subsystem Services

Our system's services and their relations and dependencies can be wrapped up as shown in the deployment diagram.

Hardwares are shown by notes in the UML notation.

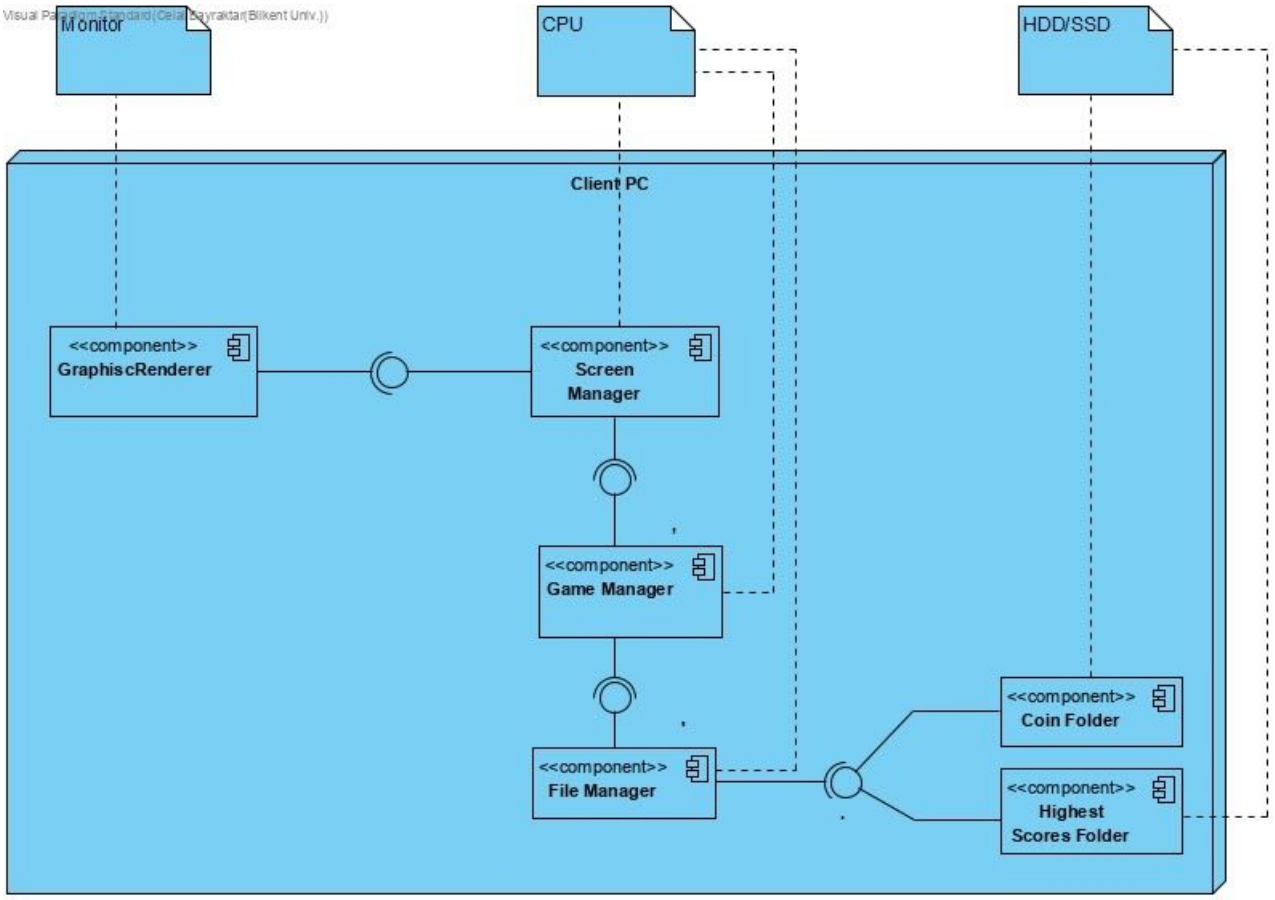


Figure 2: Deployment diagram of the system

3.1 User Interfaces Subsystem

We decided to separate our GUI Layer into two parts: Basic and Advanced GUI. We thought, this might help us to differentiate between library-provided components and custom components.

3.1.1 Basic GUI Layer

Basic GUI Layer will contain components that JavaFX library provides to users such as buttons, geometrical shapes, scrollable lists. Our implementation will be easier and more reliable due to already stabilized components offered by JavaFX.

3.1.2 Advanced GUI Layer

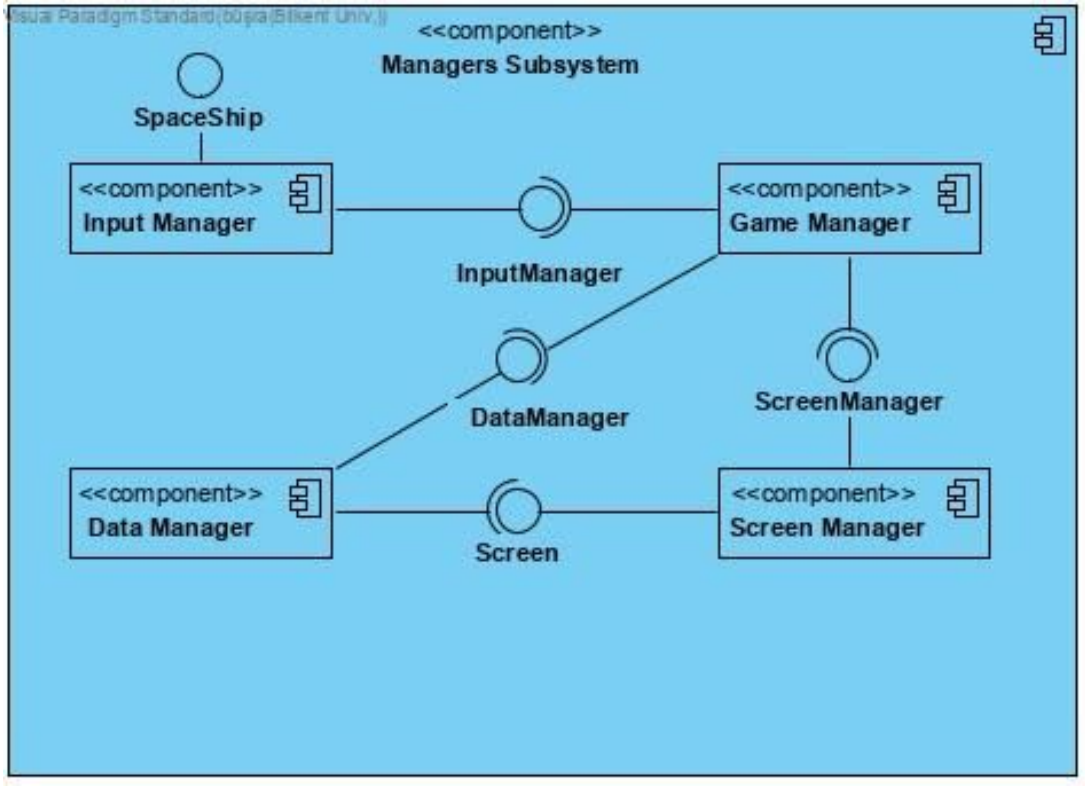
Advanced GUI layer will contain game components that JavaFX does not provide us directly such as spaceships, aliens, etc. We will complete this rendering process through a GraphicsContext object of a Canvas component. We will implement these components ourselves because basic GUI components that is provided from JavaFX does not match with our Defender design.

3.1.3 Keyboard Input Layer

Keyboard Input Layer will be embedded within GUI layers that we mentioned above because keyboard key strokes are directed to the GUI components. These keystrokes will be handled by the key listeners of JavaFx library. For basic GUI components we use default key listeners of JavaFx however, for advanced GUI components we implement our own listeners to handle the actions of the keystrokes.

3.2 Managers Subsystem

This subsystem is responsible for all the components that related to functionality of the game. It starts a new game, ends a game, creates menu and game loops, brings game data and controls game components. Currently system design for management is planned to have 4 sub components.



3.2.1 Data Manager

This manager handles game data through managing files like highest scores and gained coines, has the getter and setter classes for those attributes. Also handles the graphic and sound files, some boolean information like completing tutorials through the game.

3.2.2 Screen Manager

This manager is responsible for showing popups and setting screens. It consists of game screens, various components use this to manage popups.

3.2.3 Game Manager

This manager is responsible for the game play. Actions of the components like alien spaceships, player's spaceship, score count, losing the game are managed by this component. It works together with other managers.

3.2.4 Input Manager

This manager listens all of the keyboard input actions that user provides, sends necessary information to other components that functions through those inputs. It has a SpaceShip object that it controls.

3.3 Game Models Subsystem

This subsystem manages our game objects. It has game objects manager component under its structure. It's controlled by input and game managers. There are 3 general components that it manages:

- Enemy components: These components can be split into different subsystems by their attack types but in common, they're the components that could damage the player's ship and cause the player to lose the game. There is a passive enemy type, asteroids, that will not be able to aim and shoot at the player but can cause it to lose health. There are active enemies, alien spaceships, that will aim and shoot to damage the player's spaceship through shootings.
- Player's Spaceship: This is the spaceship that will represent the player, it could move and shoot other objects to defend itself.
- Astronauts: Those are allies that player could help through the game by picking them up from the planet below.

4. Low-Level Design

This section will give the UML class design of our program and explain each object also some of the low level design decisions of the system.

4.1 Object Design Trade-Offs

4.1.1 Efficiency vs Portability:

Since defenders isn't a game that requires lengthy operations that could affect time and space complexities, we decided to focus more on portability by using Java. Another programming language could've provided more efficiency (by appropriate usage of pointers and such) but considering the game's size and functions the cost isn't very

impactful. We decided to use Java virtual machine to make our game portable regardless of the operating system and make our game as efficient as possible in it's respectful conditions.

4.1.2 Security vs Usability:

In this project we made compromises from security by not creating accounts with passwords so users won't have to take that extra step, yet their game experience will be more open to unwanted changes.

4.2 Final Object Design

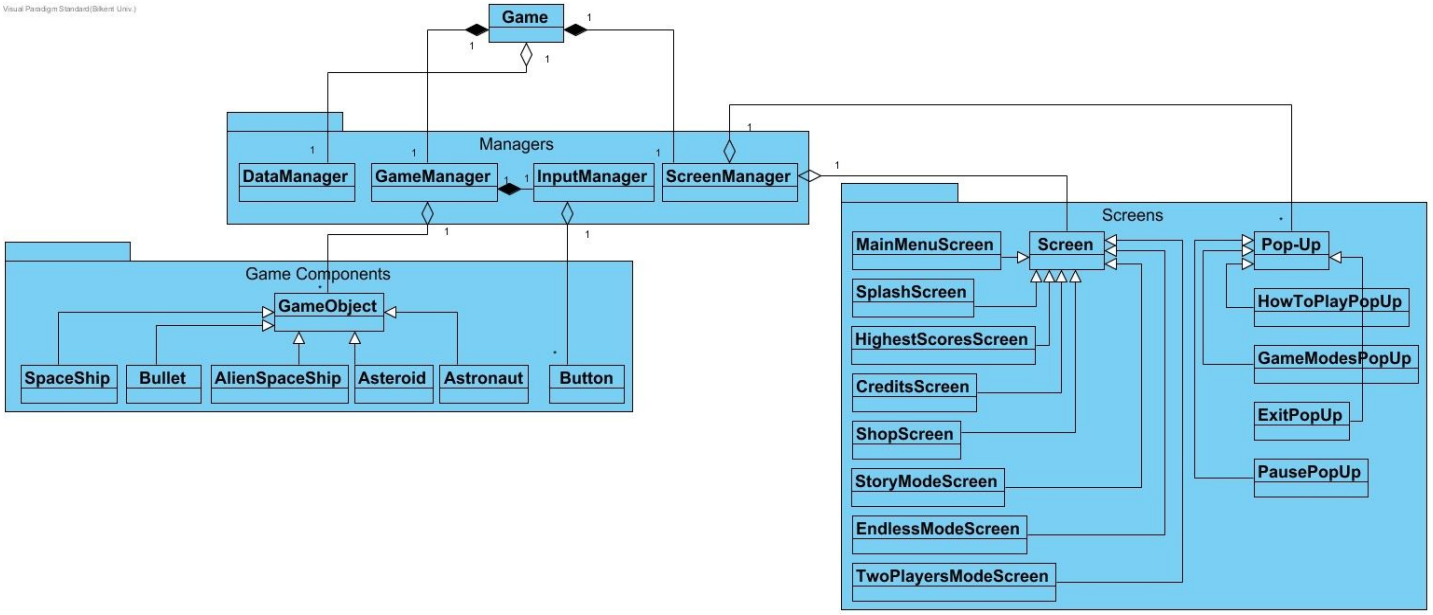


Figure 3: The general class diagram

In *Figure 4,* the general class design of our subsystems, their structures and interactions inside a package is shown. Different packages are planned to be prepared in a related manner with their corresponding subsystem definitions.

4.2.1 Game Class

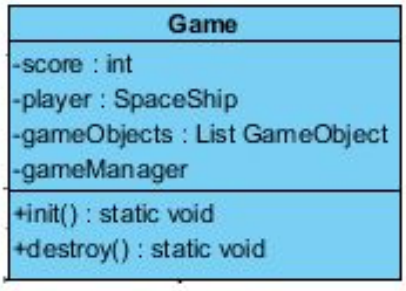


Figure 4: Game Class

Game class contains static *init()* and *destroy()* functions. We designed it according to Singleton Design Pattern. This will make sure that a time we will only have single Game initialized (and destroyed as well).

4.2.2 Manager Classes

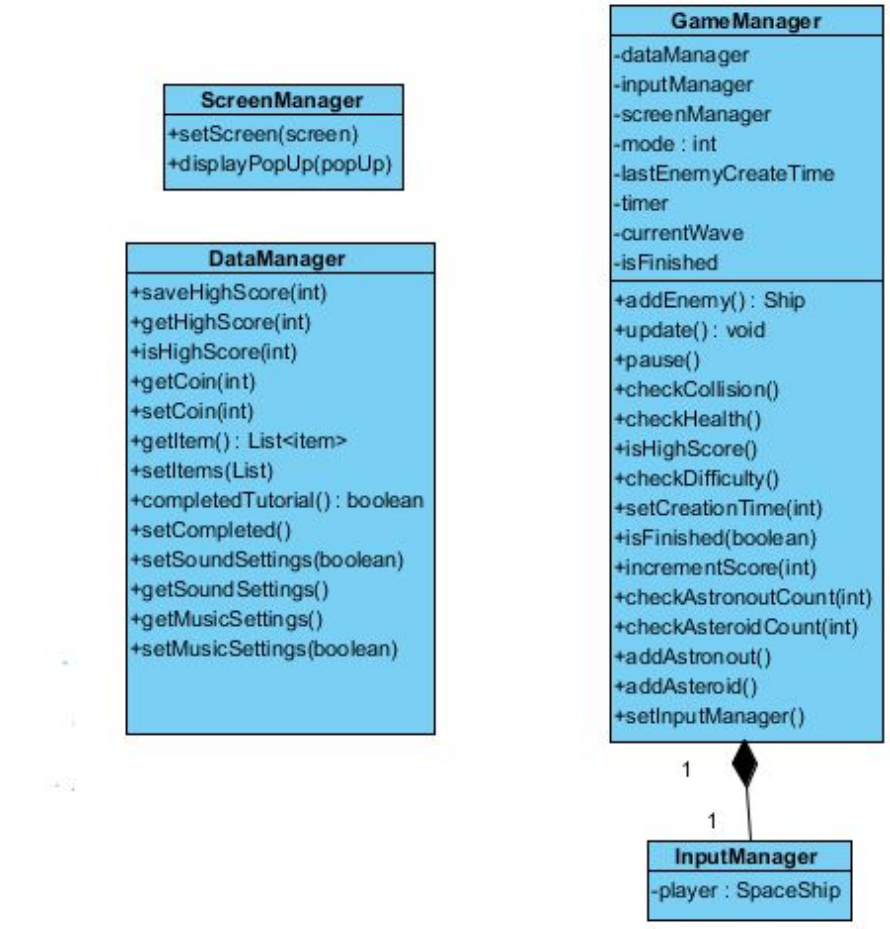


Figure 5: Manager Classes

4.2.2.1 Screen Manager

This manager sets the screen of the game according to the screen object parameter it takes. It controls the game's screen. ScreenManager will also display pop-up dialogs that appears on the screen.

4.2.2.2 Game Manager

This class holds the Game instance. It initializes which Game instance to play, adds star, asteroid and alien spaceships to that Game instance as well as updating the content, checking the collision, resuming or quitting the game. Therefore, this class manages all the features of the Game.

4.2.2.3 Input Manager

This class reacts when the user clicks a key. The player will be able to move up, down, right, left according to these clicked buttons. This manager is only in-game manager which means it will be only used while user plays the game (the user is on any GameModeScreen)

4.2.2.4 Data Manager

This class manages the data of the game such as the highest scores. With this class, we can get and set the highest scores displayed in the HighestScoreScreen.

4.2.3 Screen Classes

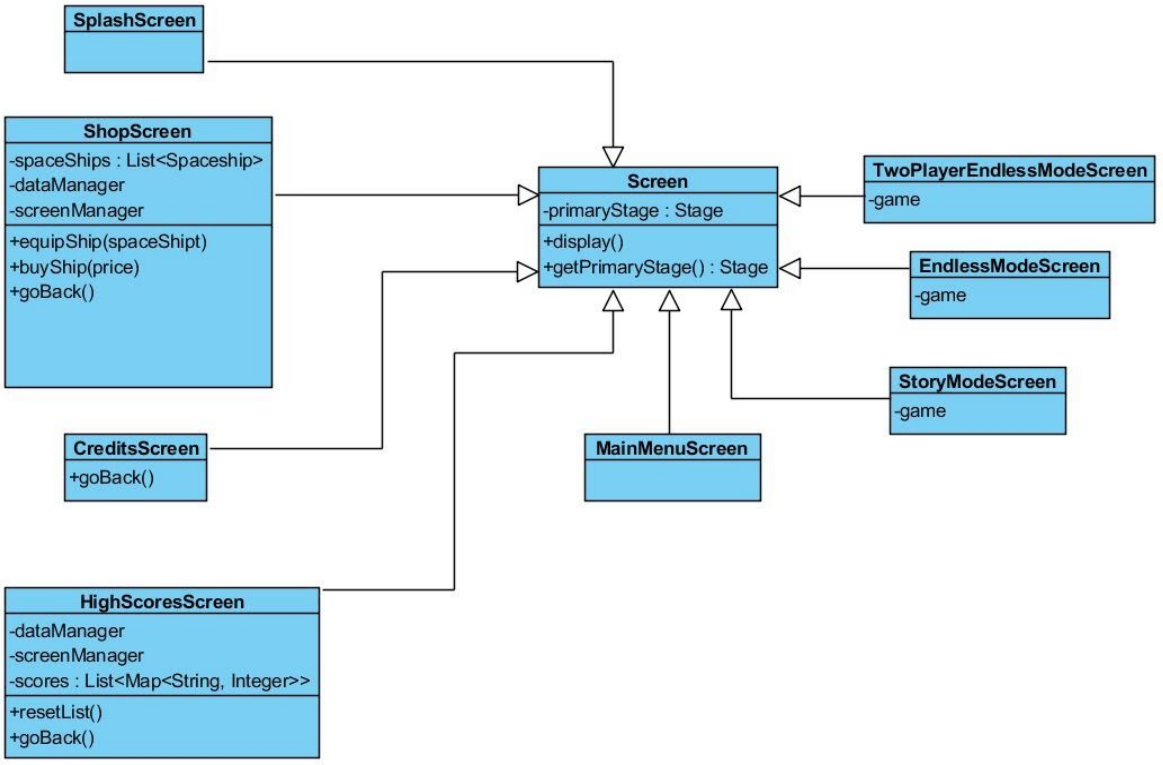


Figure 6: Screen Classes

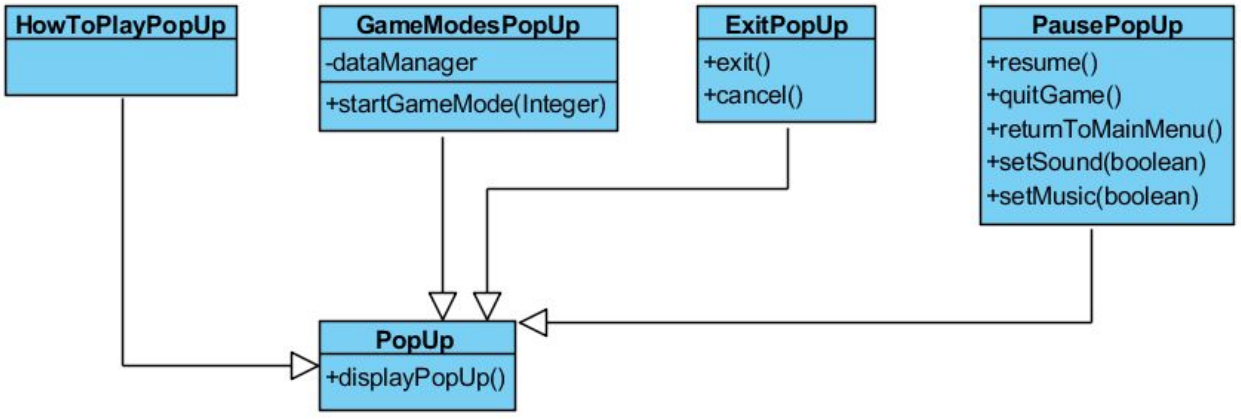


Figure 7: Pop-Up Classes

The screens package consists of all of the Screens and Pop-up dialogs to be shown in the game. Screens will also have managers to interact with players.

4.2.3.1 Screen

This class is a generic structure for all screens that will be explained in this section.

4.2.3.2 StoryModeScreen

GUI components specifically related to Story Mode will be shown inside this screen. (Story related texts)

4.2.3.3 EndlessModeScreen

GUI components specifically related to Single Player Endless Mode will be shown inside this screen (Endless Mode simply extends GameScreen).

4.2.3.4 TwoPlayerModeScreen

GUI components specifically related to Two Player Endless Mode will be shown inside this screen (There will be two GameScreens inside this Screen 1 for each player).

4.2.3.5 GameScreen

This class will structure and contain all the base Defender game related GUI components. Previous three screen classes will extend and have extra components on it.

4.2.3.6 HighestScoresScreen

In this screen, list of most 10 highest scores will be shown. There will be a backButton to return to MainMenu and resetButton to reset the scores list.

4.2.3.7 CreditsScreen

In this screen, GUI components including credit information and a backButton to return MainMenuScreen.

4.2.3.8 ShopScreen

This screen will contain a list of different space ship models. There will also be a buyButton and equipButton under each model depending on whether the ship is already bought. A backButton will be used to return to MainMenuScreen.

4.2.3.9 SplashScreen

This screen will contain GUI components to welcome users when they start the game.

4.2.3.10 MainMenuScreen

This screen will contain GUI components that lets users to access other screens. User will also be able to see sound settings related buttons on this screen.

4.2.3.11 PopUp

This class will be a general container for different Pop-Up dialogs which will be centered on the screen.

4.2.3.12 PausePopUp

This screen will contain GUI components that provides user to pause the game and mute/unmute music and effects while doing so. It also contains buttons to resume, restart or quit the game.

4.2.3.13 ExitPopUp

This pop up screen will include GUI components that enables the user to exit the game or continue playing the game.

4.2.3.14 HowToPlayPopUp

This pop up screen will contain GUI components that list the instructions on how to play the game. There will be a button to go back to the game.

4.2.3.15 GameModesPopUp

This pop up screen will contain GUI components that shows the player the three modes (Single Player Story Mode, Single Player Endless Mode, Two Players Endless Mode) of the game. If the Single Player Story Mode is not finished yet, other modes will be displayed with a lock symbol on them.

4.2.4 GameComponent Classes

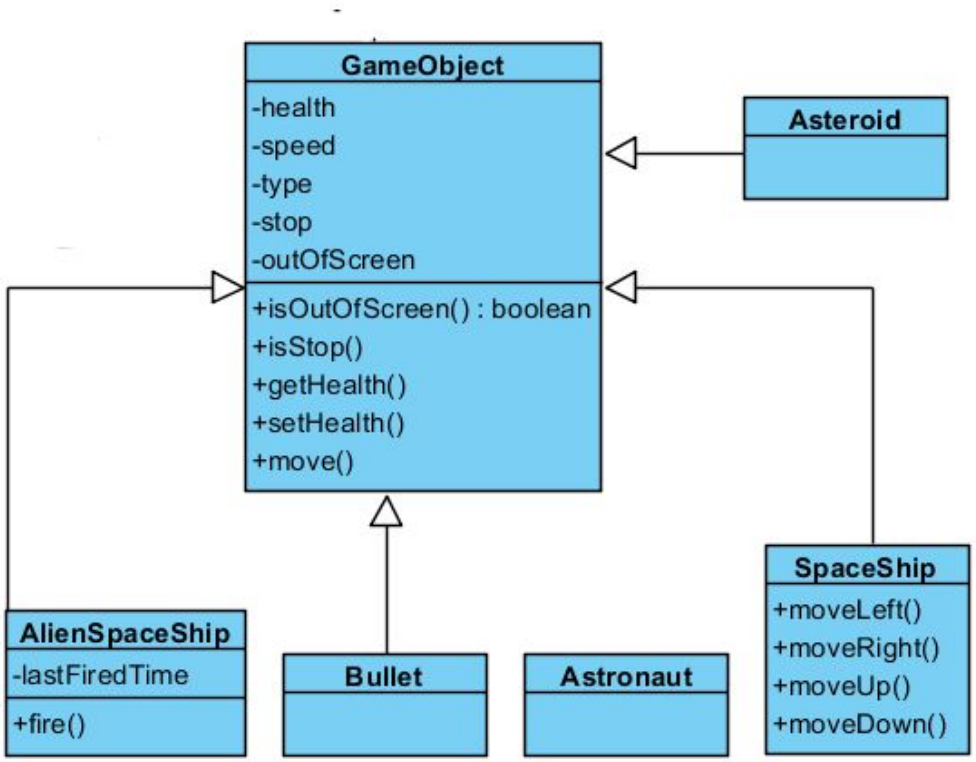


Figure 7: GameComponent Classes

4.2.4.1 GameObject

It gives a generic class for the game objects. It has position and speed attributes and a move method. SpaceShip, Bullet, AlienSpaceShip, Asteroid and Star classes are part of this class.

4.2.4.2 SpaceShip

This defines the generic functionality of the spaceship. It will have health and color attributes and methods for firing and setting the color of the spaceship.

4.2.4.3 Bullet

This class will contain the bullet's color as an attribute.

4.2.4.4 AlienSpaceShip

This class will have health attribute of the spaceship and will have a method to enable firing.

4.2.4.5 Asteroid

Asteroid class will have health and size attributes.

4.2.4.6 Button

This class has the dimension, title and function of the buttons in the game. It also has methods to get the function of the game and whether it is clicked or not.

4.3 Applied Design Patterns

4.3.1 Model-View-Controller Design Pattern

To understand each component better we decided that best design approach for are project is MVC. Our object has functional and graphical components that need to work with each other coordinately and MVC pattern offers a clear understanding on how different logics should work together therefore helps program to have easier implementation.

4.3.2 Singleton Design Pattern

To avoid conflicts regarding to main game managers that maintains the game flow we will use the singleton design pattern to ensure there is only one instance of the objects that are needed to be only one at the time.

4.3.3 Immutable Design Pattern

To control communication between objects as we need we decided to limit the access permissions of objects from different classes. To achieve that we need to use immutable objects with Java's appropriate keywords (final etc...). The objects that will be implemented as immutable objects are asteroid, bullet. Additionally, Credits Screen, pop-ups will be implemented as immutable objects.

4.4 Packages

4.4.1 Internal Subsystem Packages

As mentioned in Section *4.2,* our classes will be split into four packages, managers, screens, gameModels, gameComponents, according to the subsystem they belong to. This will help us to ensure that private accesses into packages will be more efficient, secure and organized. This pattern will also make our implementation faster.

4.4.2 Other Internal Packages

In order to manipulate data flow in and out of the game, create readable code and better UI graphics we will include packages like FileUtils, DrawUtils, Rectangle2D any other packages that might be needed during implementation.

4.4.3 External Packages

For hardware/software mapping as mentioned in *Section 2.2* javajx.sound and java.util will be used. We will also consider new packages during implementation stage.

5. Improvement Summary

- We went through the feedback that TA gave us, starting from some grammar, styling and formatting mistakes.
- Design goals are reviewed. They now clearly list our goals and show how we will address these goals. Our report now provides measurable requirements.
- The parts that makes design ambiguous are extracted from the class diagram.
- A simple diagram is added to visualize Management Subsystem.
- Separation of basic and advanced GUI layers are clearly justified.