CS-319 Term Project

Defender

Design Report

Team:

- Hanzallah Azim Burney
- Abdul Hamid Daboussi
- Perman Atayev
- Gledis Zeneli
- Endri Suknaj

Instructor: Ugur Dogrusoz

Teaching Assistant(s): Hasan Balci

Contents

Design Report

Defender

1 Introduction

1.1 Purpose of the system

Defender is designed to give users a modern experience of an old arcade game. It uses an OOP approach on the original game making the game software more robust, extendable and reliable. Defender puts a new spin on the original game making the game more creative in terms of the different features, making it more intuitive to play and giving users the chance to interact with a more user friendly UI. It is designed to entertain its users while allowing them to possibly reminisce about the era of arcade games.

1.2 Design goals

There are several non-functional requirements that need to be translated into specific design goals for the game which carry forward from the analysis report.

1.1.1 Usability

Defender will be optimized in terms of intuitiveness and effectiveness of the components so that the user achieves their objectives with great efficiency. This will start from the minimalistic game UI that will allow the user to focus on the game rather than worry about understanding how different components act and react. Having a simple UI directly translates into an intuitive UX. The simple menu and understandable game interface will lower the learning curve necessary for the game and familiar keyboard inputs such as spacebar and the arrow keys will also enable the user to adapt quickly to the game environment. All of this is expected to reduce the number of times the user has to go to the help menu to understand the game and is expected to lead to more user interaction with the game.

1.1.2 Extendibility

The game design creates room for easily changing and adapting features as well as functionalities in the future depending on things such as user feedback. This is achieved through high coherence inside components and low coupling between them allowing minimal disruption in case of change. Abstraction and division of classes into different layers and subsystems also enables this. For example, instead of the normal gameplay, we can have another gameplay consisting of various different challenges defined by the game that can be played by the user. Adding such a feature later on will be far less cumbersome if proper extendibility was implemented from the start.

1.1.3 Modifiability

The game is multi-layered with multiple subsystems that are loosely coupled which allows the system and code to be easily modifiable. By having loosely coupled components, change to one has minimal effect on the rest of the system, which allows quick and easy edits. This is especially true in the case that people other than the original developers attempt to modify parts of the project later on or in the case of modifying the project ourselves in the second iteration.

1.1.4 Reusability

The game design has a structure from which different components can be reused in other games of similar nature. These include aspects such as the game objects, the game engine responsible for controlling other components and most importantly the collision detector which can be used in any application where the collision of different components occur. The GUI components of the main menu and Scene can also be reused by others for graphics generation. Furthermore, classes will be separated in packages at the code level, which itself facilitates reusability at the package level.

1.1.5 Portability

The decision was made to develop the game using Java for numerous reasons. Mainly, however, it was due to Java's high portability. "Write once, run anywhere" is what Java does best and this allows our program to be easily portable to different platforms.Also, JavaFX will be used for the game. From Java 7 and on JavaFX is fully integrated with the Java Runtime Environment(JRE), and Java Development Kit(JDK). And since JDK is available on Windows, Mac OS X, Linux, our application is portable.

1.1.6 Robustness

There is not much room for invalid inputs in the game since users can only move, shoot and navigate in the menu bar. Test cases that will test the general features of the game will be created. Jubula might be used for automated testing purposes as well.

1.1.7 Performance

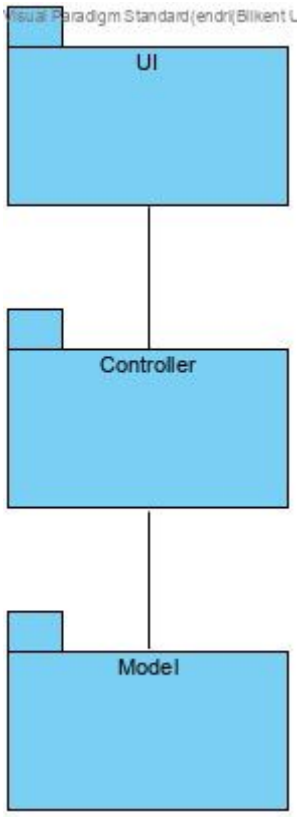
On the design level, the MVC design pattern is used. The design is optimized to make the best use of available resources, such as classes and their methods.

2 System Architecture

2.1 Why MVC?

There are many different architectural styles that could have been used for the project, however, most of them are not suitable for developing games. One of the good candidates for the style was the three-tier architecture. Since the game is going to be single-player and not synchronised across players, there was no need to connect the game with a database, which could also slow down the game. Therefore, we decided to use the MVC (Model View Controller) architecture. There are many models that are similar to MVC, such as HMVC, MVVM, MVP. However, the scale of the project is not big therefore there is almost no difference among these architecture styles. In our MVC, the Model is a GameObject component which is essentially every object that is a part of the game. The View consists of the menu and scene generators. And our Controller, is the Game State class.

2.2 Subsystem Decomposition



Our system is divided into three subsystems respectively the UI, Controller and the Model subsystem. The UI subsystem will handle the display of the game and will have two components, one of which will display the main menu screen and the other will generate and update the display during the gameplay. The Controller subsystem will read data from the Model and use that info to update the UI. The Model will contain all the data of our game such as time, location, health, collisions etc.

2.3 Hardware / Software Mapping

Our game operates in a single machine and thus does not require complex hardware-software coordination. The game is played with the keyboard and the menus can be navigated with both mouse and keyboards. The coordination of this hardware input is made possible through Java classes and does not require us to do any additional work.

2.4 Persistent Data Management

Defender does not require a significant number of complex data to be stored either in a local cache or in a database. It will only store the images in a folder which are needed for the UI representation of the game objects. It will also require the storage of high scores, which will be stored with the help of Java's Highscore class which protects the data from being modified by the user while maintaining low complexity of design.

2.5 Access, Control and Security

Defender in its current form will not be accessing any database or any internet/network connection for that matter which minimizes security risks and data leaks. The game allows

any user to click on it and immediately start playing without the need to register first or provide any identity so access control from the user end in unnecessary. The high scores section will be unique to each computer rather than keeping scores from among all possible devices where this game has been played. Although this limits the games' features, it first allows the system to be simple to design and implement, and also limits the possibility of security leaks.

2.6 Boundary Conditions

The game design takes into account boundary conditions that are needed to ensure the reliability of the game.

2.6.1 Initialization

We decided to implement the game using Java for multiple reasons. The main reason being the portability that Java inherently provides thanks to the JVM. The game comes in a JAR file that can be clicked and run on any PC where Java is installed, no need to worry about the process of installing the game. Clicking the JAR initializes the game and creates the main menu. Clicking play on the main menu then initializes the game engine, generates the GUI and runs the game logic.

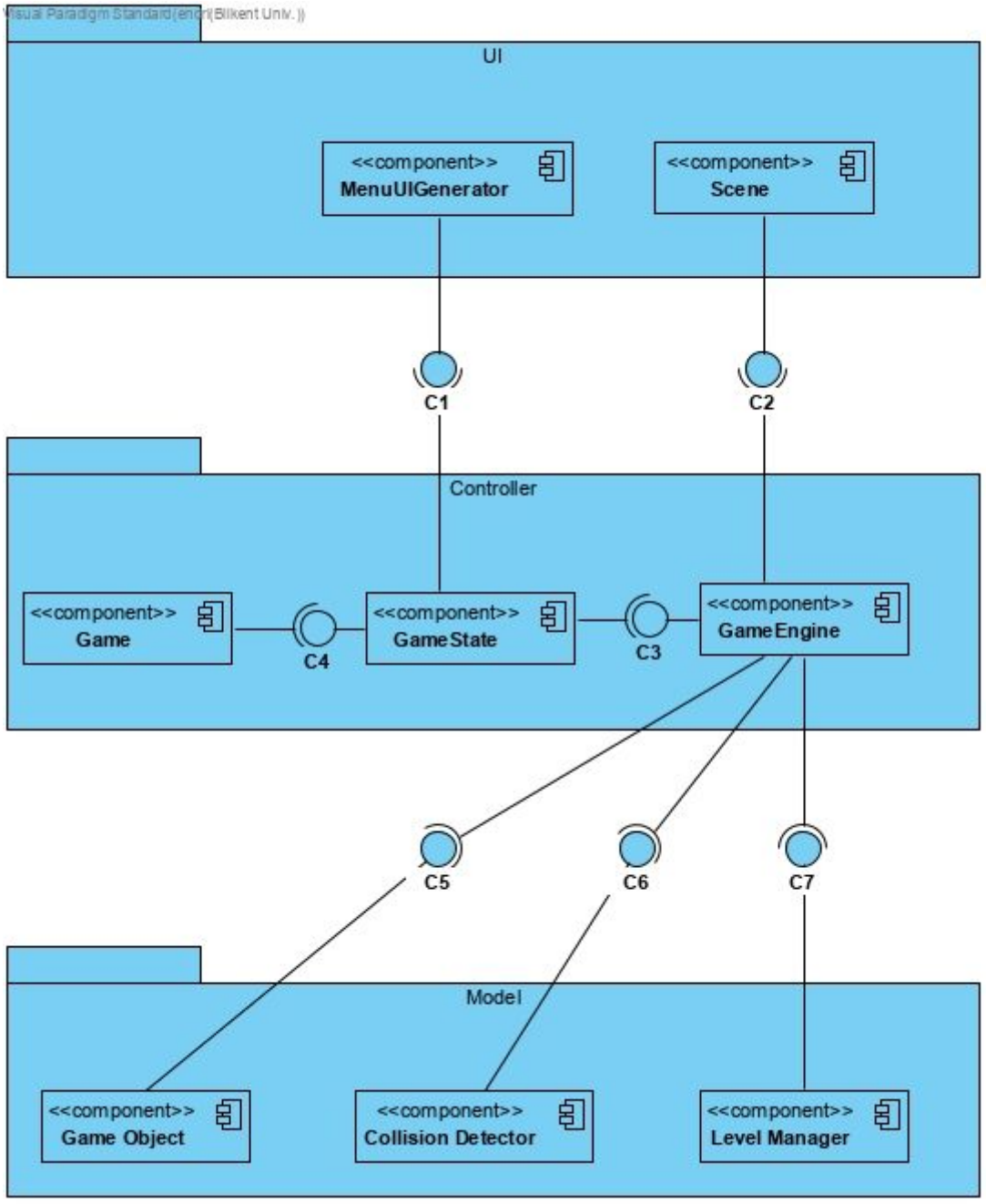
2.6.2 Termination

The game is terminated either by clicking the ESC button on the game window or by clicking the exit button on the main menu. The user can also temporarily pause the game in which case they will return to the main menu and exit from there or once again resume the game.

2.6.3 Failure

If the game collapses due to an unexpected performance issues in the game design, then the score as well as the current game state will not be retrievable since the game ended abruptly. If there are difficulties in loading the object images due to corrupt files then this would be detected early on and the user will not be allowed to proceed with the game until the file in question is fixed.

3 Subsystem Services



.

3.1 UI Subsystem

The UI subsystem will handle the display of the game. This subsystem contains MenuUIGenrator and Scene components. The MenuUIGenerator component will create the menu screens (specifically the main menu which appears at the start of the game, and the pause menu which appears when the user pauses) and the subsequent screens which will be accessible from the menus (the High Score screen, the About screen, and the Help screen). Scene component will be responsible for handling the gameplay UI. Scene updates the

display 60 times a second by changing the location of Mothership, Aliens, Humans, Projectiles and Mines and checking collisions accordingly.

3.2 Controller Subsystem

The Controller subsystem contains the Game, Game State and Game Engine components. The Controller serves as a mediator by reading data from the model subsystem and then updating the display/UI using that information. The Game component will launch the game by initiating the Game State component. The Game State component will mediate the changes from gameplay to menu by coordinating the Game Engine component and the MenuUIGenerator component. It will notify the Game Engine component to stop when the user pauses and will tell the MenuUIGenerator to display the pause menu. On the other hand, it will initiate the Game Engine when the user hits play on the main menu, or when they resume from the pause menu. The Game Engine component controls the gameplay by receiving the logic from the Model components and passing this information appropriately to the Scene component so it can update the gameplay UI accordingly. It also has the ability to pause the functionality of the Scene component when the user has paused the game.

3.3 Model Subsystem

The Model Subsystem contains the Game Objects, Collision Detector and Level Manager components. This subsystem holds the information about the objects in our game such location, health etc. and does all the computations such collision detection and change of levels. The Game Objects component consists of a unified arrangement of data for all the moving objects in our game, specifically the mothership, aliens, humans, and projectiles. This component will be utilized by the Game Engine component to create, store, and use the data of all our game objects. Collision Detector component will receive information from the Game Engine component and will calculate for collisions and handle their consequences. The Lever Manager component will calculate scores and keep track of time according to the information fed to it from the Game Engine component and it will change the levels of game.

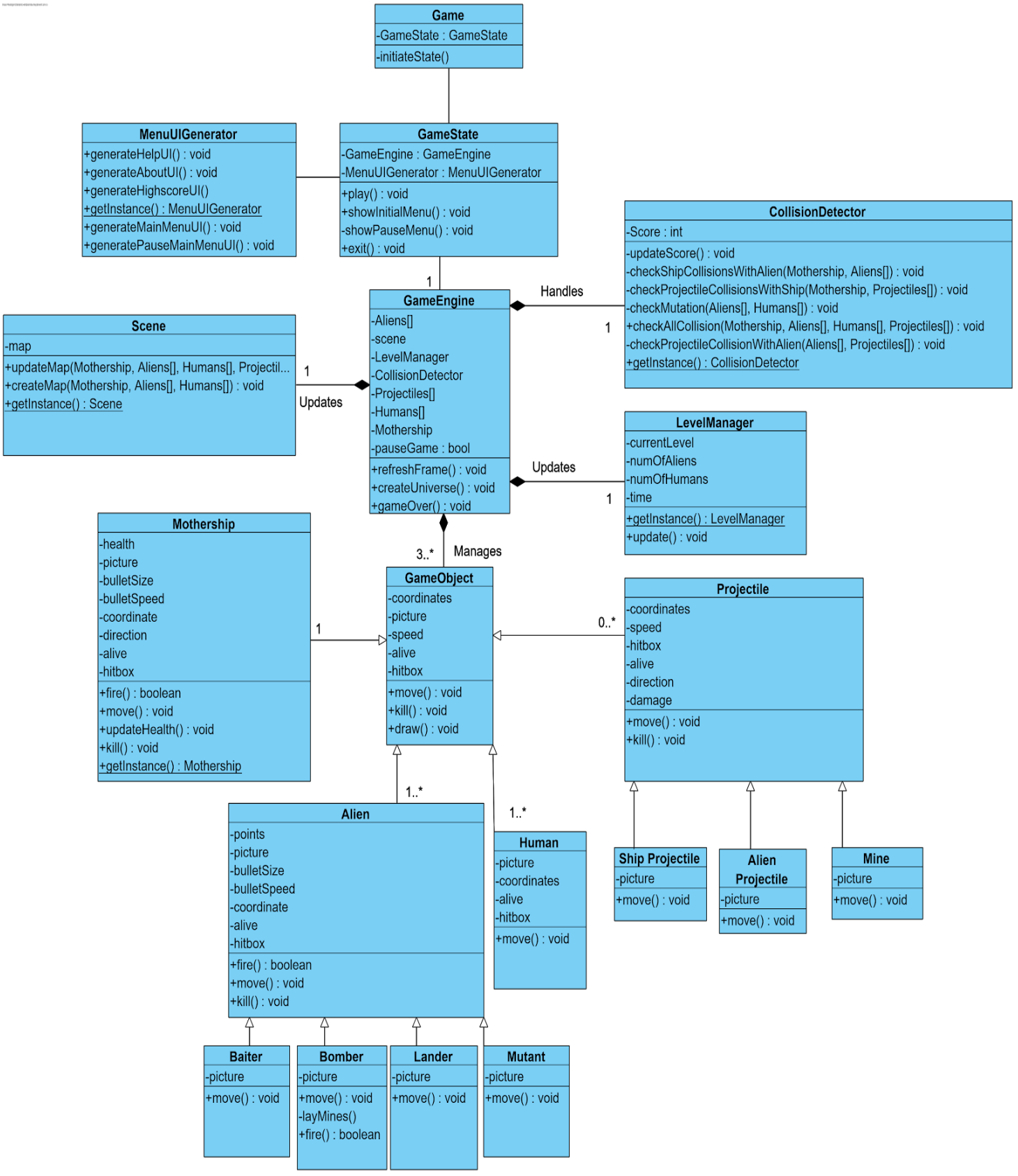
4 Low-Level Design

4.1 Object Design Trade-offs

The project, adhering to best software engineering practices, incorporates multiple design patterns. The decision to choose one pattern over the other was debated by weighing their pros and cons. As such, trade-offs naturally occurred.

Our most recurring pattern was abstraction-occurrence, it was chosen over the square variant because it most closely followed OOP's concept of abstraction. Another pattern which is used throughout our system is the singleton design pattern. This is extremely important in the case of classes such as GameEngine, Scene, CollisionDetector, MenuUIGenerator, LevelManager and Mothership because such classes would create problems if instantiated multiple times. Delegation was also utilized, in the case of the CollisionDetector for example. The GameEngine delegates collision detection to CollisionDetector because it does not make sense to make use of inheritance in here. Finally, we made use of the ideas behind the Façade pattern. Although we do not explicitly have a Façade class in our design, the idea behind a Façade class and what it provides caught our attention. We designed our GameEngine class with Façade characteristics in mind. The GameEngine class is the only linking point between all the model classes (game objects) and the controllers and UI. Logic and methods in the GameEngine class communicate directly with the game objects, gather data and perform changes, then communicate that to the UI and the rest of the system. By doing so, changes to the UI for example will not affect the game objects interface, and changes to the game objects, such as addition of new classes would only require the GameEngine to be modified.

4.2 Final Object Design



4.3 Packages

The project's code will be divided into three packages as follows: UI, Model and Controller. Such division builds upon OOP design principles of having separate highly decoupled components which can be reused. Furthermore, packages from JavaFX will be included such as `javafx.scene*` and `javafx.util.*` .

4.4 Class Interfaces

4.4.1 KeyListener

This interface will be called upon whenever a keyboard key is pressed or released. The game engine will implement this to allow the user to move and shoot.

4.4.2 MouseListener

This will be invoked whenever a mouse action is detected. It will primarily be used in the main menu to browse through the menu options.