Bilkent University

Department of Computer Engineering

CS 319 - Object-Oriented Software Engineering

Term Project - Design Report

Project Name : Slay The Spire

Group No : 1A

Group Name : 1A - SS

Group Members : Gamze Burcu Ayhan
Ekin Doğaç Öztürk
Farid Mirzayev
Okan Alp Ünver
Nihat Bartu Serttaş

Table Of Contents

1. Introduction

1.1. Purpose Of The System

Slay the Spire is a single player desktop game for defeating enemies using selected cards in an most optimal way. The game aims to entertain players while also training risk management skills of players. The Desktop version of a game will be implemented in a way that will provide an easy to play, simple and attractive user interface for users. The distinctive feature of this version of Slay the Spire will be additional critical damage feature. This feature will make the game more enjoyable and unpredictable because players will get a chance to critically hit an enemy by luck. Also, at the beginning the luck of the player to get critical hits will be low, but as the game progresses the chance of getting critical hits will be increased and by this way the progression of the game will speed up. In substance, the main purpose while implementing the Slay the Spire will be making an easy to use, fast, and entertaining game.

1.2. Design Goals

The primary design aim is to provide classes that fulfill requirement demands correctly and use object-oriented design concepts in such a way that they comply with the original design of the game. Maintainability is one of the primary objectives of design to keep implementation design smooth, easy to understand and enduring to alter even as implementation complexity increases.Extensibility is another design goal for Slay the Spire to provide a simple and synchronized addition of new features to the game. For these goals in mind, this implementation will aim to include a design that is sustainable, extendable and reusable.

2. High-Level Software Architecture

2.1. Subsystem decomposition

We wanted to get our device decomposed into 4 bits. Explanation is provided in subsections below for each subsystem.

2.1.1. Managers

It is the central game-state and resource control subsystem. Control subsystem struggle with the controlling and execution of data flows, file operations, screen transitions, and the algorithms that construct the main game logic.

2.1.2. Screens

Display objects are the items of the highest level for showing the user's game status and menu interfaces. The Screens subsystem encapsulates all the objects on the monitors and their rationale. Screens are often responsible for handling image-based inputs (click on a certain item by mouse).However, there is no particular reasoning about the game or any design process about the components of the game in the Screens. But it relies on subsystems from GameViews and GameModels.

2.1.3. GameViews

This subsystem is used to accurately render the elements of a game to the computer. The current state of the game is displayed to the player via this subsystem's components. This subsystem is controlled by Screens subsystem.

2.1.4. GameModels

This subsystem is intended to store the game data and its current status. In an abstract manner it stores data related to challenges. This subsystem is not relying on any other subsystem as it is the heart of the game logic.

2.2. Hardware/Software Mapping

Our game will be entirely implemented on the software, and some standard libraries will link it to hardware. For the following purposes, we should use Java to accelerate our implementation: it guarantees platform-independent applications and offers important object-oriented tools such as inheritance and polymorphism (see Section 4.3 for patterns of applied design).

● We'll use a JavaFX stage for screen display, which manages both GPU rendering and platform-independent frame formation.

● The javax.sound package provides one with a proper interface for sound outputs.

● We'll use the mouse listeners in JavaFX for mouse input.

2.3. Persistent Data Management

Our game will create a hidden folder for storing the game and player based data in json files. Data which is going to be saved will be splitted into parts based on their categories. As a result of categorization, everything will be organized, so it will be easier to save and retrieve data. Planned structure of the files is listed below.

2.4. Access Control And Security

We will not use any external databases as stated in Section 2.3, we will only use the internal storage and we will not implement access control. Our device does also not support network connection that is why malicious software will not be able to expose it.

2.5. Boundary Conditions

2.5.1. Application Setup

There will not be any additional installation while setting up the Slay the Spire game. All of the required files that contain game and user data will be read from json files which are stored in a hidden folder.

2.5.2. Terminating The Application

Slay the Spire can be finished by pressing the exit button, which is placed on the game window. By using this feature,the user can terminate the game whenever he wants.

2.5.3. Input/Output Exceptions

Several cases may cause potential game failures. If a file reading problem occurs during initialization, which can be triggered by an exception or an invalid task, then the game can have trouble loading Start-Combat Screen and Continue-Combat Screen properly. The second case can be solved by detecting and deleting the challenging data.

Also, the export phase of the Challenge can fail due to issues of access permission. This can be solved by modifying the access permission settings for the folder by giving the appropriate instructions to the user.

2.5.4. Critical Errors

In case of any application crash, the game will recover in the same way as a regular start but in this case, user progress data will not be saved and will be lost.

3. Subsystem Services

3.1. User Interface Layer

3.1.1. Basic GUI Layer

This layer will include basic components such as buttons, text fields and scrollable lists supported by the JavaFX library. These can make implementation easier and also improve software reliability as many other JavaFX applications use standard library components as stable and reliable tools.

3.1.2. Advanced GUI Layer

This layer will contain the components such as , player character, enemy, cards, relics and potions. They will be rendered by a java object in ImageView. We decided to implement some GUI components ourselves, similar to the game's, because we thought we would get more sufficient and good-looking game elements this way.

3.1.3. Mouse Input Layer

This layer is embedded in the other layers of the GUI as the mouse clicks are directed to the components of the GUI (both basic and advanced). Clicks to the basic GUI layer will be done via JavaFX library's mouse listeners. To progress clicks on the Interface layer, we must add our own handlers, on top of the click listener of canvas component which we will implement.

3.2. Application Layer

3.2.1. Representing the Challenge Data

Representing the challenge screen objects in memory was one of the most challenging tasks of our design. Since the combat includes different objects (cards, enemies and relics) with similar features, we did not want to store duplicate data in memory in order to use them. Although the cards, enemies and relics have similar features, their functionalities, representations made the work harder.

3.2.2. Working with the Challenge Data

The data of the combat screen must be changed during various actions (Start combat, Abandon combat, End turn, End combat). In order to implement object oriented design, we decided to make a Controller class which can be accessible from other subclasses(methods) and so the necessary subclass can call and change the controller accordingly. By this way the performance of a game while changing data will not be affected.

3.3. Storage Layer

Our game will require access to the storage as described above. This layer must communicate with the combat layer's task data part. The details of a game will be stored using a .json file type.

4. Low-level design

4.1. Object design trade-offs

4.1.1. Functionality vs Usability

Our game offers a very basic drag-and- drop control system. We have chosen usability over functionality as the only interaction device we have made is mouse. The game then becomes easy to understand, to play and to control. Since our game is going to be easy to interact, therefore you can easily adapt too.

4.1.2. Efficiency vs Portability

Slay the Spire will be implemented in Java environment and it will use Java virtual machine which reduces efficiency but is independent of our program OS. So, we chose portability over efficiency. We expect that not even the inefficiently implemented game algorithms will

take significant resources such as time and memory because the game itself is a simple one.

4.2. Final Object Design

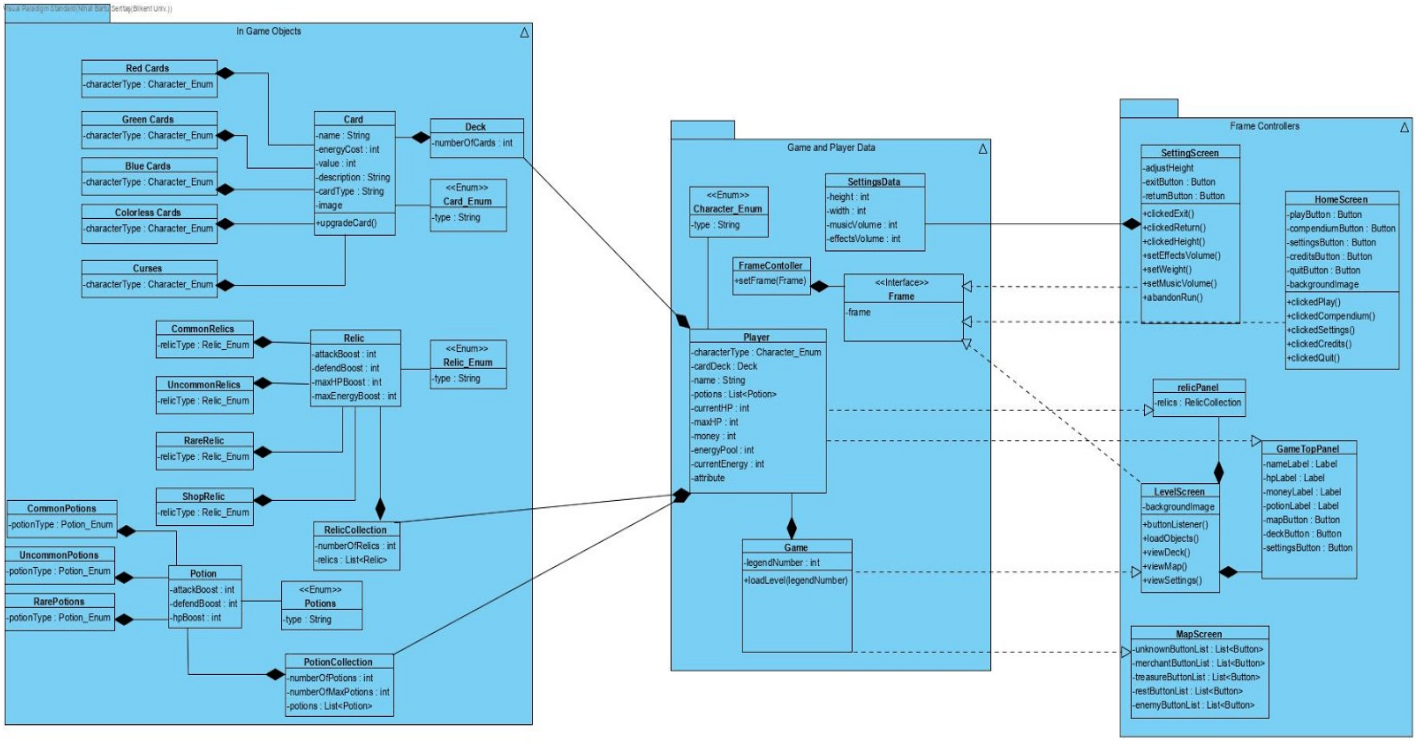

Figure 4.2.1 General Design Diagram

Discuss the classes and their architecture in this section. We modeled the low-level structure of our subsystems and their interactions in a packaged fashion, as shown in Figure 1. Class packaging is performed in accordance with the definitions of the unit (See Section 4.4 for further detail on packaging). The following subsections provide further descriptions of the grades.

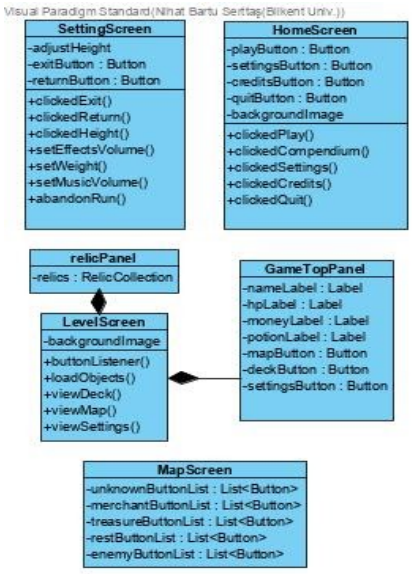

Figure 4.2.2 Screen Controllers Diagram

These are the controllers for the fxml files of each different screen throughgout the game. They control the transitions between the screen, what button does what action and so on. They are also responsible for saving necessary data (for example player health and money, chosen options in the setting screen and so on) on to the created .json files. After the project progresses

HomeScreen:

The home screen is the first screen the player sees after running the game. From this screen player can use the **settingsButton** and go to the setting screen. Also the player can choose to view the credits by the **creditsButton**, or exit the game using the **exitButton**. Finally, there is the **playButton** which initializes the game after being clicked. For now, the play button takes the player directly to the map screen with the default character and a certain beginner card deck.

MapScreen:

When the player starts the game, they will be introduced to the **mapScreen.** On this screen there will be randomly placed level buttons, which will be locked except the starting points. The player can click on the unlocked starting buttons which will send the player to the level screen, and lock the other starting points. When the player returns from the **levelScreen** to the **mapScreen,** the map will be

updated and the buttons connected to the previously clicked button will now be unlocked.

There are 5 different types of level buttons which are the same with the original game and they decide the content of the level. The number of each distinct button will be pre defined and they will be placed and connected to each other randomly. When the player clicks on the button they will be sent to the **levelScreen.**

LevelScreen:

The levelScreen is where the contents of the levelButtons are created and presented to the player. For example if the player clicked on a merchantButton on the map, when they come to the screen the merchant character will be placed on the screen and it's functionalities will be activated. However if the player clicked on a combatButton, enemie(s) will be placed on the screen and combat functionalities will be activated and the endTurnButton will be placed as well.

SettingScreen:

Setting screen has only three functionalities at this moment because it would slow down the progress of the game and a diversion from the actual game. There is an adjust height button which will be used for adjusting the resolution of the game later on. And at this moment there of course return and exit buttons. Return button simply closes the setting screen and returns to the screen which was open prior to the setting screen, and exit button saves the necessary data on to the .json file and terminates the program.

GameTopPanel:

GameTopPanel will be a JavaFX object which can be used in multiple screens. This panel will give an option to players to see the game data easily. The panel will contain the player's name, health points, current money and potions. In addition to these, buttons which routes to the map, deck and settings screens will also be placed in this panel.

RelicPanel:

This is a part of the game levelScreen which is used by the player to view their currently equipped relics and their effects.

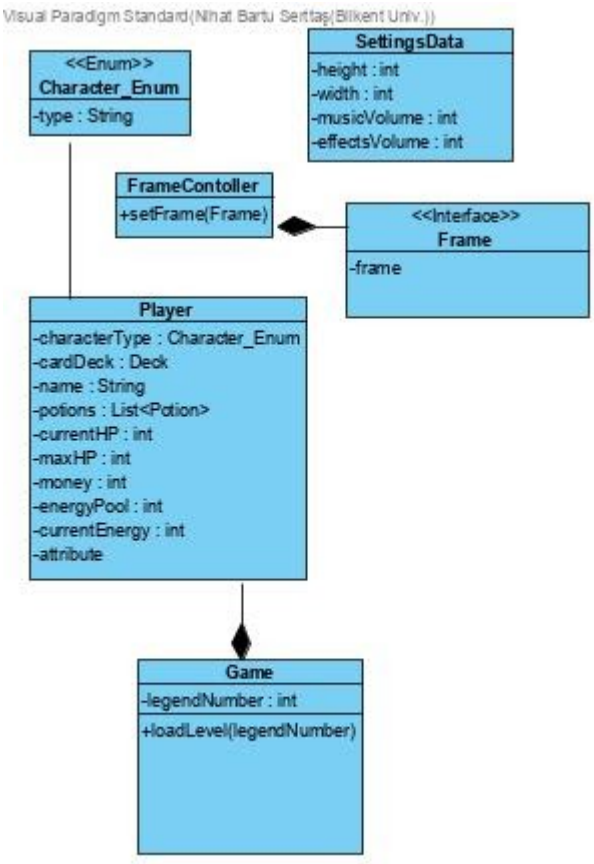

Figure 4.2.3 Player and Game Data Diagram

Player:

Player class stands for holding and managing all of the properties of the player object. This object will represent a player as a set of player attributes which of them could be interpreted as a with components of player object.

In an effort to introduce these properties, charachterType stands for holding character types which are Iron Clad, Silent, Defect. CardDeck stands for a deck which is currently owned by the player. Name stands for name of the Player. Potions list will be a list of potions that the player is currently owned. currentHp will hold the health point of our character. maxHP is the maximum health point that player can

ever get which means maximum value that player can ever reach that can change as the player proceeding in game and gaining enhancement on maximum health of his. Money stands for the player's current money and so on.

This class will be the main data provider for the main game. And player objects will have several connections with other classes such as Frame, Combat and Map. Player should see his hp, name and money in the mainframe, to give an example for a combat, player should see his/her deck in combat and so on.

Game:

Game class is a identifier for loading a level for a player which are merchant, treasure, rest and enemy.

Frame:

This is an interface for the FrameController

FrameController:

FrameController will be used for getting the data from .json files to provide other controllers with the required data. As an example, if the user clicks the map button, this controller will get the player's progress data from player_progress.json file. So, the map controller will draw the clickable and locked buttons based on the information FrameController gives.

Charachter_Enum:

Charachter_Enum will identify the type of the character from a finite set which are IronClad, Silent, Defect. By using enumeration we avoid type check errors and will provide readable code for us.

SettingsData:

SettingsData holds the settings which contain height, width, music volume and effect volume. The settings can be adjusted by the user.