# Design Document for **Dragon Ball Game**

Name: **Quoc Phi Long Pham**

Student ID: **104771041**

Demonstration video link: <https://youtu.be/mtv1ut0h_DE>

# Introduction

In this document, I will demonstrate my custom program for Distinction task and its functionalities. This program is a simple battle simulation game inspired from the Dragon Ball universe. The objective of the game is for players to recruit powerful characters, transform them into many different forms, and engage in strategic battles against challenging villains. The game implements object-oriented principles and has several key design patterns, making it an ideal demonstration of OOP concepts.

# Program’s Functionalities

# Game Menu

The Game Menu is where players start the game. It allows them to view their characters and stats, their amount of zeni (using GameMenu class) and choose between different actions (using InputHandler class):

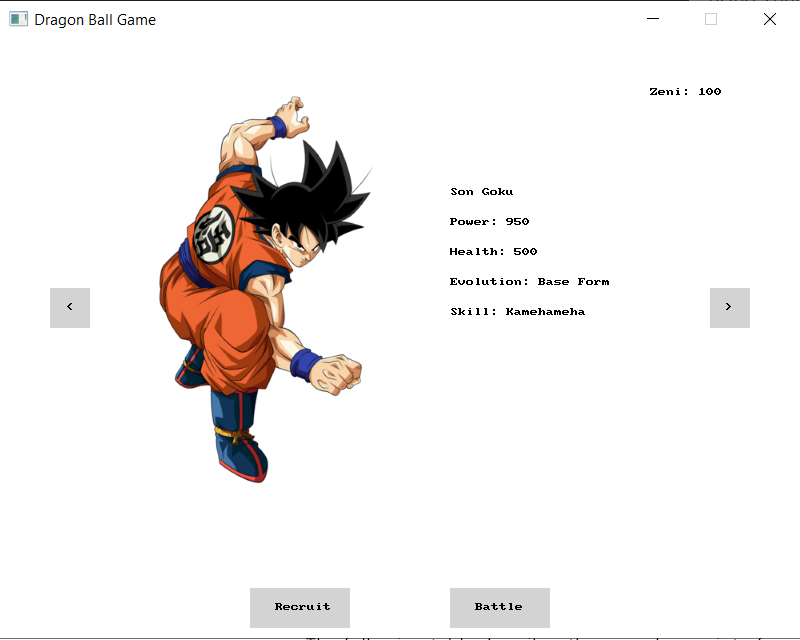
* Character Recruitment: Recruit new characters for the player’s list.
* Character Selection: Select which character to use in battle.
* Start a Battle: Begin a fight against a villain with the selected character.

Figure 1: Game Menu

# Character Recruitment

The player begins with only Son Goku but can recruit more characters by spending Zeni, which can be earned by winning battles. Each character has unique abilities and transformations but all derived from a base Character class. If a player recruits a duplicate character, the original one evolves to the next transformation level, making them stronger. This feature is handled by RecruitSystem class, which follows the Singleton pattern to ensure there is only one recruiter managing characters throughout the game.

# Figure 2, 3: Character recruited and evolved

# Difficulty Selection

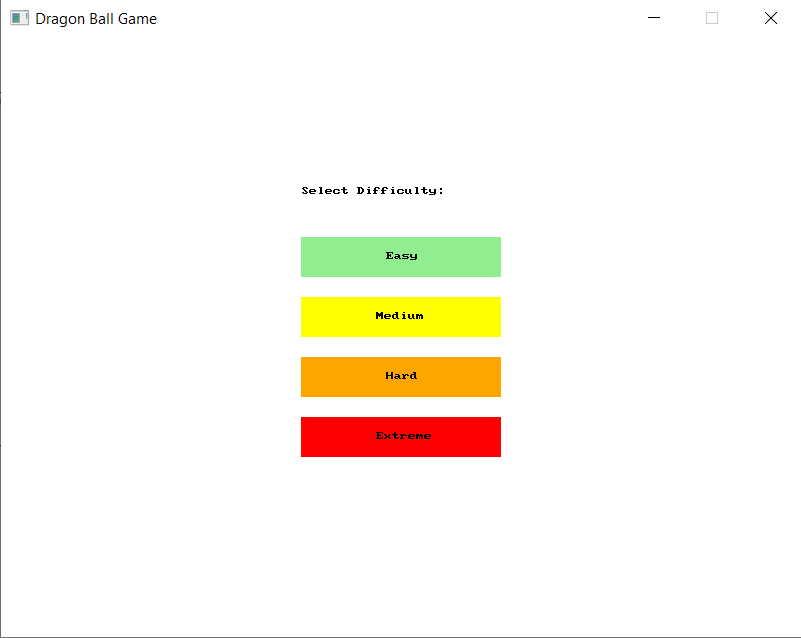
The difficulty selection menu appears when starting a battle. This feature is managed by the BattleManager class and allows player to choose Easy, Medium, Hard, and Extreme level by clicking the buttons on the screen. The BattleSystem uses the difficulty level chosen to create an appropriate villain through the CharacterFactory, which centralizes character creation.

Figure 4: Difficulty Selection Menu

# Battle Mechanics

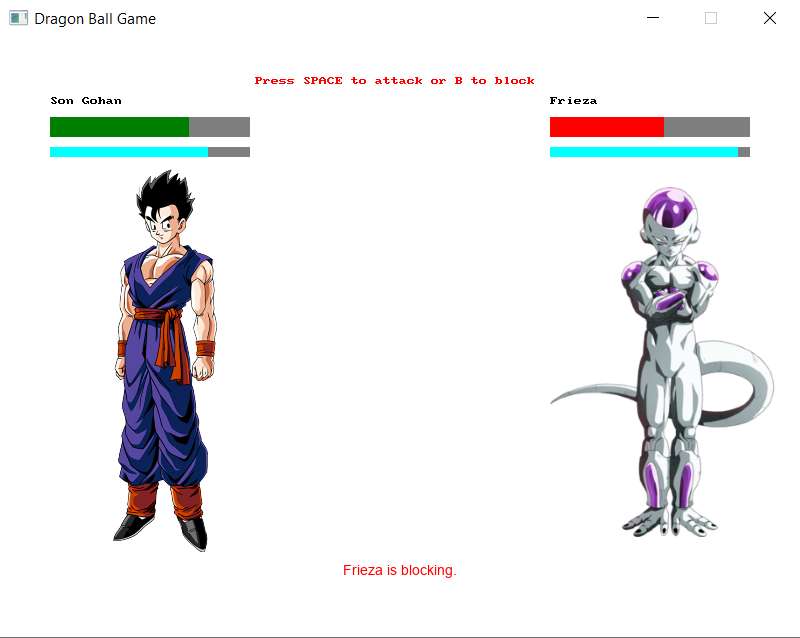
I implemented a turn-based battle mechanics for this game in BattleSystem class. During a battle, players can attack or block. Both actions charge the character's energy bar, and when it reaches 100, the character can perform a special attack with increased damage. The blocking mechanism reduces incoming damage by a half, adding a layer of strategy for players to decide whether to attack or block. The player and villain take turns, and the battle continues until one side's health reaches zero. The BattleUI class handles drawing character health, energy bars, and messages to the screen after each action taken.

Figure 5: Battle Interface

# Villain’s AI

The villain’s AI is implemented using the VillainAI class with the Strategy Pattern to dynamically adjust the villain’s behavior. The AI uses different strategies (aggressive, defensive, default) based on the villain's health during the battle and power compared to the player. The VillainAI chooses the appropriate strategy (AggressiveVillainStrategy, DefensiveVillainStrategy, DefaultVillainStrategy) at each turn. This AI logic ensures that villains adapt to the player's actions: for example, if a villain’s health is low, they are more likely to block. This feature makes it more challenging for players as they need to be prepared for both offensive and defensive style from the bot.

# Design Patterns Used

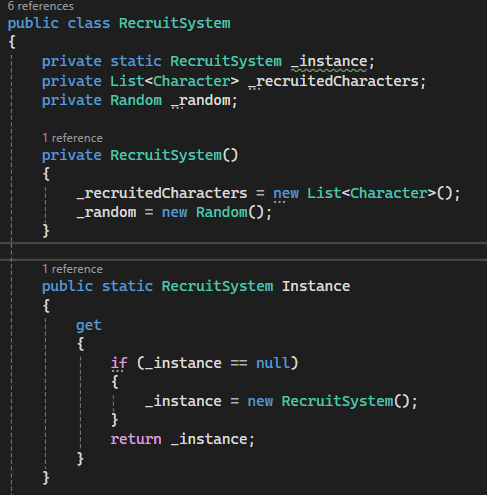
**Singleton Pattern**: The RecruitSystem class uses the Singleton pattern, ensuring there's only one instance managing the recruitment process. This approach helps maintain consistency when recruiting or evolving characters.

Figure 6: RecruitSystem public instance

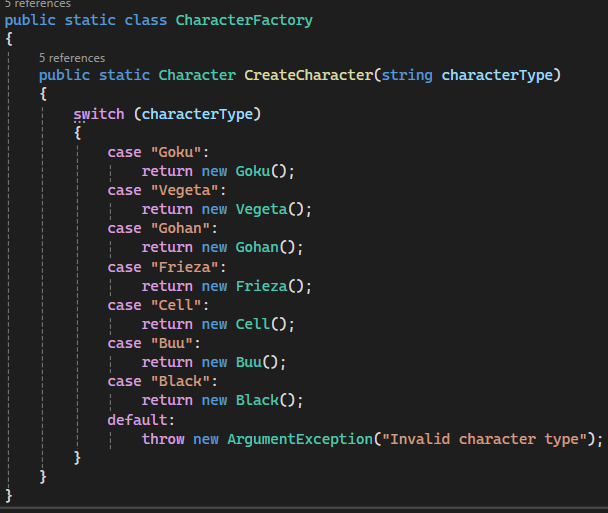
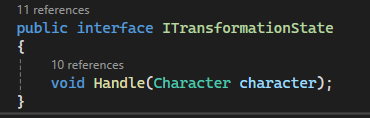
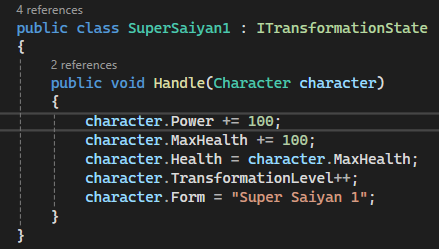
**Factory Pattern**: The CharacterFactory class employs the Factory pattern to simplify character creation like Goku, Vegeta, or villains based on the current needs of the game. This makes adding new characters straightforward.

Figure 7: CharacterFactory class

**State Pattern**: State pattern was implemented for handling character transformations. The ITransformationState interface and classes like SuperSaiyan1, 2, 3… manages different transformations. When characters evolve, they transition between these states, allowing them to increase stats.

Figure 8, 9: Interface and class for transformation state

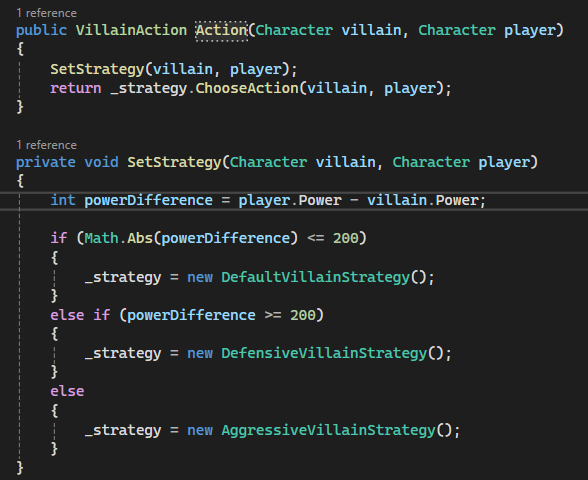
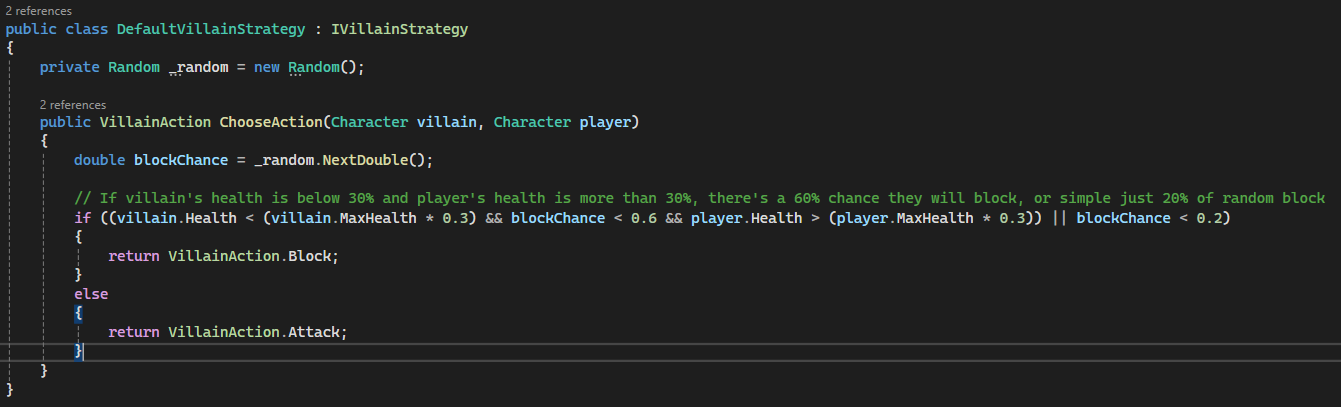
**Strategy Pattern**: The villain’s behavior uses the Strategy pattern. The VillainAI class dynamically selects a strategy for each villain's action based on their health and the difference in power compared to the player. Different strategy classes encapsulate specific behavior, enhancing modularity and making it easy to adjust villain tactics.

Figure 10, 11: Villain’s strategy code

# Required Classes / Interfaces

Table 1: Core classes and interfaces

|  |  |  |
| --- | --- | --- |
| Class/Interface | Type Details | Description |
| Character | Abstract class | Base class for all characters, containing core attributes like health, power, transformation level, and methods for evolving, attacking, and blocking. |
| Goku, Gohan, Vegeta | Class (inherit from Character) | Derived classes from Character, each representing a specific character with unique attributes, transformations, and special abilities. |
| Frieza, Cell, Buu, Black | Class (inherit from Character) | Derived classes from Character, representing villains with unique attributes and abilities. |
| ITransformationState | Interface | Represents different transformation stages for characters, allowing for polymorphic behavior when handling state transitions. |
| SuperSaiyan1, 2, 3… | Interface (implement ITransformationState | Represents different transformation states for characters, allowing them to evolve into more powerful forms. |
| CharacterFactory | Class (Factory) | Implements the Factory pattern to create instances of different characters, allowing centralized character creation. |
| Player | Class | Represents the player, managing Zeni, recruited characters, and interactions with game systems. |
| RecruitSystem | Class (Singleton) | Responsible for recruiting new characters and evolving existing ones when duplicates are recruited. Implements Singleton pattern to ensure a single instance exists. |
| InputHandler | Class | Handles user input, such as mouse clicks, for navigating the character selection and battle menus. |
| ImageManager | Class | Manages character images and loads the correct image based on the character's transformation state. |
| MessageManager | Class | Manages messages displayed to the player during the game, ensuring that they are clear and visible for a limited duration. |
| BattleManager | Class | Manages the initiation of battles, including difficulty selection and interaction with the BattleSystem to carry out battles. |
| BattleSystem | Class | Manages the turn-based combat logic, including player and villain actions, energy accumulation, special attacks, and invoking the villain AI strategy. |
| BattleUI | Class | Handles graphical representation and user interaction, such as drawing health and energy bars, and refreshing the screen during battle. |
| IVillainStrategy | Interface | Defines a method for villain action selection. Implemented by various strategy classes (e.g., AggressiveVillainStrategy, DefensiveVillainStrategy). |
| Default/Aggressive/  DefensiveVillainStrategy | Class (implements  IVillainStrategy) | Implements specific villain strategies for deciding actions during battles. |
| VillainAI | Class | Implements different strategies (aggressive, defensive, default) to decide how villains react based on health and power difference during battles. |
| GameMenu | Class | Manages the main game menu, allowing players to navigate options such as character recruitment and starting battles. |

Table 2: Enumerations details

|  |  |
| --- | --- |
| Enumeration | Description |
| **VillainAction** | Defines possible actions for villains (Attack, Block). |
| **RecruitStatus** | Defines the possible outcomes when recruiting characters (NotEnoughZeni, NoAvailableCharacters, NewCharacterRecruited, CharacterEvolved). |

# UML Class Diagram

Figure 12: UML Class Diagram for Dragon Ball Game