UNIVERSITY of GREENWICH
Alliance with FPT Education

BTEC

ASSIGNMENT 2 FRONT SHEET

Qualification	**BTEC Level 5 HND Diploma in Computing**		
Unit number and title	Unit 20: Advanced Programming		
Submission date	26/08/2023	**Date Received 1st submission**	
Re-submission Date		**Date Received 2nd submission**	
Student Name	TRUONG DUONG HONG PHUOC	**Student ID**	GCC210321
Class	GCC1001	**Assessor name**	Tran Thi Kim Khanh
Student declaration I certify that the assignment submission is entirely my own work and I fully understand the consequences of plagiarism. I understand that making a false declaration is a form of malpractice.			
		Student's signature	

Grading grid

P3	P4	M3	M4	D3	D4

❒ Summative Feedback:	❒ Resubmission Feedback:	
Grade:	**Assessor Signature:**	**Date:**
Lecturer Signature:		

ASSIGNMENT 2 BRIEF

Qualification	**BTEC Level 5 HND Diploma in Computing**		
Unit number and title	Unit 2: Advanced Programming		
Assignment title	Application development with class diagram and design patterns		
Academic Year	2018-2019		
Unit Tutor	Doan Trung Tung		
Issue date	30 April 2019	**Submission date**	11 May 2019

Submission Format:

Format: The submission is in the form of an individual written report. This should be written in a concise, formal business style using single spacing and font size 12. You are required to make use of headings, paragraphs and subsections as appropriate, and all work must be supported with research and referenced using the Harvard referencing system. Please also provide a bibliography using the Harvard referencing system.

Submission Students are compulsory to submit the assignment in due date and in a way requested by the Tutors. The form of submission will be a soft copy in PDF posted on corresponding course of http://cms.greenwich.edu.vn/ together with zipped project files.

Note: The Assignment *must* be your own work, and not copied by or from another student or from books etc. If you use ideas, quotes or data (such as diagrams) from books, journals or other sources, you must reference your sources, using the Harvard style. Make sure that you know how to reference properly, and that understand the guidelines on plagiarism. *If you do not, you definitely get fail*

Assignment Brief and Guidance:

Scenario: (continued from Assignment 1) Your team has shown the efficient of UML diagrams in OOAD and introduction of some Design Patterns in usages. The next tasks are giving a demonstration of using OOAD and DP in a small problem, as well as advanced discussion of range of design patterns.

Tasks:

Your team is now separated and perform similar tasks in parallel. You will choose one of the real scenarios that your team introduced about DP in previous phase, then implement that scenario based on the corresponding class diagram your team created. You may need to amend the diagram if it is needed for your implementation. In additional, you should discuss a range of DPs related / similar to your DP, evaluate them against your scenario and justify your choice.

In the end, you need to write a report with the following content:

- A final version of the class diagram based on chosen scenario which has potential of using DP.

- Result of a small program implemented based on the class diagram, explain how you translate from design diagram to code.
- Discussion of a range of DPs related / similar to your DP, evaluate them against your scenario and justify your choice (why your DP is the most appropriate in that case).

Learning Outcomes and Assessment Criteria		
Pass	**Merit**	**Distinction**
LO3 Implement code applying design patterns		
P3 Build an application derived from UML class diagrams.	**M3** Develop code that implements a design pattern for a given purpose.	**D3** Evaluate the use of design patterns for the given purpose specified in M3.
LO4 Investigate scenarios with respect to design patterns		
P4 Discuss a range of design patterns with relevant examples of creational, structural and behavioral pattern types.	**M4** Reconcile the most appropriate design pattern from a range with a series of given scenarios.	**D4** Critically evaluate a range of design patterns against the range of given scenarios with justification of your choices.

Contents

1 Introduction

- This document outlines a comprehensive analysis of scenario-driven design pattern implementation in software development. The focus is on effectively integrating design patterns within practical coding scenarios, accompanied by clear explanations and illustrative diagrams. This document aims to provide a succinct overview of the primary components covered and the approach adopted for analyzing and implementing design patterns within real-world software projects.
- In the broader context, my application involves managing three distinct lists comprising Characters, Monsters, and Weapons, which are integral elements within a game environment. At present, my primary task revolves around overseeing these lists and facilitating testing procedures to ensure seamless functionality

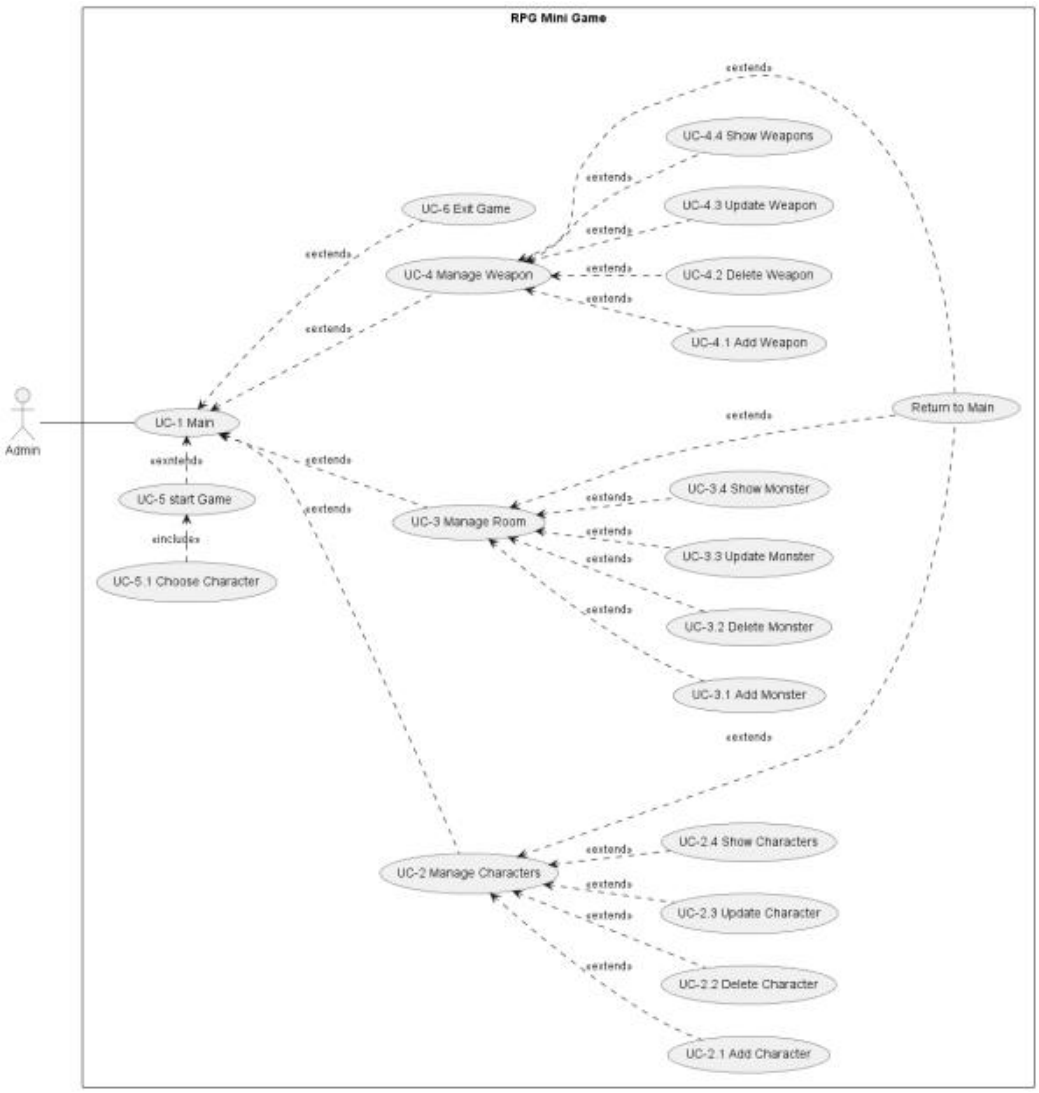
2 Scenario analysis

2.1. Scenario

I have been hired to work on a game development project called "Epic Quest." Players will control different characters and explore a fantasy world full of adventures. My specific focus is on designing character management and customization functionalities. The game features character classes like Knights, Archers, Assassins, and Mages, each with unique attributes. Players can encounter monsters in various rooms and engage in thrilling battles. Additionally, they can equip characters with weapons such as swords and bows. Throughout the development, I utilize essential Object-Oriented Programming principles to ensure a modular and efficient game architecture. The goal is to deliver an immersive and engaging gaming experience for players to enjoy

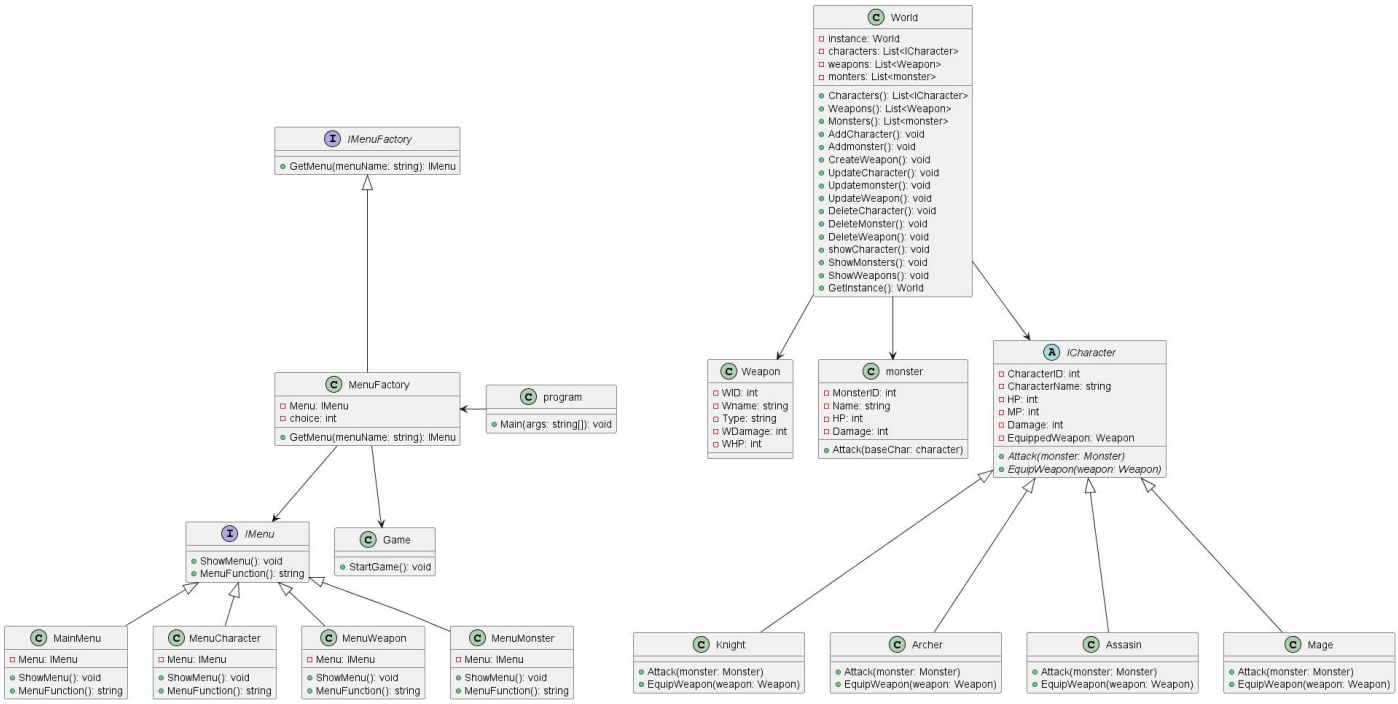
2.2. Diagram

2.2.1. Usecase Diagram



Explain:The 'Admin' actor initiates interactions by accessing the 'Main Menu,' which offers a spectrum of administrative functions including character, room, and weapon management, starting the game, and exiting. Each chosen action leads to specific use cases, enabling the Admin to perform tasks, select characters, progress to the game, and conclude by exiting the RPG Mini Game system

2.2.2. Class Diagram



This is Look not really similar to my before Class diagram I have change some of method make it move to other class and MenuFunction now return string type in order to Fully Develop of functions. Here the describe of Class Diagram:

The 'Admin' actor initiates interactions by accessing the 'Main Menu,' which offers a spectrum of administrative functions including character, room, and weapon management, starting the game, and exiting. Each chosen action leads to specific use cases, enabling the Admin to perform tasks, select characters, progress to the game, and conclude by exiting the RPG Mini Game system

3 Implementation

3.1. Code

Class-Funtions	Code	Explain

Class: Program
Function: Main
	public class program
	{
	public static void Main(string[] agrs)
	{
	
	MenuFactory menuFactory = new MenuFactory();
	IMenu menu = null;
	string MenuName = "main";
	do
	{
	menu = menuFactory.GetMenu(MenuName);
		- The 'program' class serves as the entry point for the application. - The 'Main' method is the starting point of execution. It creates instances of the MenuFactory and iterates through menu interactions,

	MenuName = menu.MenuFunction(); } while (MenuName.Equals("exit")); } }	creating menus and executing their functions based on user input.

Class-Function:
- Class: IMenuFactory
- Function: GetMenu
 	
Class: MenuFactory
- Function: GetMenu
	interface IMenuFactory
	{
	public IMenu GetMenu(string Menu);
	}
	public class MenuFactory : IMenuFactory
	{
	IMenu Menu;
	int choice = 0;
	public IMenu GetMenu(string menuName)
	{
	string choice = menuName.ToLower();
	IMenu menu = null;
	
	while (!choice.Equals("exit"))
	{
	switch (choice)
	{
	case "main":
	menu = new MainMenu();
	break;
	case "character":
	menu = new MenuCharacter();
	break;
	case "weapon":
	menu = new MenuWeapon();
	break;
	case "monster":
	menu = new MenuMonster();
	break;
	case "game":
	Game game = new Game();
	game.StartGame();
	break;
	case "exit":
	Console.WriteLine("Out Game");
	Environment.Exit(0);
	break;
	}
	
	if (menu != null)
	{
	choice = menu.MenuFunction(); // Get the next menu choice
	}
	}
		- The 'IMenuFactory' interface defines a contract for a factory responsible for generating instances of the 'IMenu' interface.
		
		- The 'GetMenu' function within this interface is expected to take a 'Menu' parameter and return an instance that adheres to the 'IMenu' interface.
		
		- The 'MenuFactory' class implements the 'IMenuFactory' interface, providing functionality to create instances of 'IMenu' based on user inputs.
		
		- The 'GetMenu' function takes a 'menuName' parameter, converts it to lowercase, and iterates through a menu loop until the user decides to exit.
		
		- Within the loop, a switch-case structure determines which type of menu to create based on the 'menuName'. For example, if the 'menuName' is "main," a 'MainMenu'

	return menu; } }	instance is created. - The 'choice' variable manages user input and loop continuation. If the user selects a valid menu, the 'MenuFunction' of that menu is invoked to obtain the next menu choice. - The loop continues until the user chooses to exit, at which point the application is gracefully terminated. - The 'MenuFactory' class demonstrates the factory pattern, where it encapsulates the process of creating different types of menus based on user input.

Class: IMenu
- Function: ShowMenu, MenuFunction
 	
- Class: MainMenu
- Function: MenuFunction, ShowMenu
	public abstract class IMenu // base menu
	{
	public abstract void ShowMenu(); // show Menu
	public abstract string MenuFunction(); // Menu Function Handle
	}
	
	//Main menu inheriate from Imenu Class
	public class MainMenu : IMenu
	{
	IMenu Menu;
	public override string MenuFunction()
	{
	string choice = "";
	ShowMenu();
	int choose = int.Parse(Console.ReadLine());
	switch (choose)
	{
	case 1:
		- The 'IMenu' class serves as a base abstract class for creating menu types.
		
		- The abstract method 'ShowMenu' is declared to define how menus should be displayed.
		
		- The abstract method 'MenuFunction' is defined to handle menu functionality and returns a string representing the

	choice = "character";
	break;
	case 2:
	choice = "weapon";
	break;
	case 3:
	choice = "monster";
	break;
	case 4:
	choice = "game";
	break;
	case 0:
	choice = "exit";
	break;
	
	}return choice;
	}
	public override void ShowMenu()
	{
	Console.WriteLine("1.To CharacterMenu\n2.To Weapon Menu\n3.To Room Menu\n4.Start Game");
	Console.WriteLine("input your choice");
	}
	}
	}
		user's choice.
		
		- The 'MainMenu' class inherits from the 'IMenu' class and defines specific menu behavior. - The 'MenuFunction' function processes user input and returns a choice based on the selected option. - Inside 'MenuFunction,' the 'ShowMenu' method is called to display available options to the user. - The user's choice is obtained using 'Console.ReadLine' and is used in a switch-case structure to determine the next menu or action. - The 'ShowMenu' function presents the available menu options to the user, prompting for input. - The 'MainMenu' class encapsulates the main menu's behavior, including displaying options and determining user selections.

- Class: MenuCharacter - Function: MenuFunction	public class MenuCharacter : IMenu
	{
	private World World;
	public MenuCharacter(World world)
	{
	this.World = world;
	}
	public MenuCharacter()
	{
	
	}
		- The 'MenuCharacter' class represents a menu for managing characters and inherits from the 'IMenu' class. - The class includes constructors to either initialize the 'World' instance or

	public override string MenuFunction()
	{
	
	int choice = 99;
	string ExittoMain = "";
	while (choice !=0)
	{
	ShowMenu();
	choice = int.Parse(Console.ReadLine());
	switch (choice)
	{
	case 1:
	World.Instance.AddCharacter();
	break;
	case 2:
	World.Instance.DeleteCharacter();
	break;
	case 3:
	World.Instance.UpdateCharacter();
	break;
	case 4:
	World.Instance.showCharacter();
	break;
	case 0:
	ExittoMain = "main";
	break;
	default:
	Console.WriteLine("Invalid choice. Please select a valid option.");
	break;
	}
	}
	return ExittoMain;
	}
		create an empty constructor. - The 'MenuFunction' method handles user interactions within the character menu. - Inside a while loop, the user is repeatedly presented with the character menu options. - The 'ShowMenu' method displays the available options to the user. - The user's choice is processed through a switch-case structure, where each case corresponds to an action related to character management. - Options include adding, deleting, updating, and showing characters in the 'World' instance. - If the user chooses to return to the main menu (choice 0), 'ExittoMain' is set to "main." - Any invalid choice results in an error message.

- Class: MenuWeapon - Function: MenuFunction, ShowMenu	public class MenuWeapon : IMenu
	{
	public override string MenuFunction()
	{
	
	int choice;
	string exit = "";
	
	ShowMenu();
	choice = int.Parse(Console.ReadLine());
	switch (choice)
	{
	case 1:
		- The 'MenuWeapon' class represents a menu for managing weapons and inherits from the 'IMenu' class. - The 'MenuFunction' method handles user interactions within the weapon menu. - Inside the method, the user is prompted

	World.Instance.CreateWeapon();
	break;
	case 2:
	World.Instance.DeleteWeapon();
	break;
	case 3:
	World.Instance.UpdateWeapon();
	break;
	case 4:
	World.Instance.ShowWeapons();
	break;
	case 0:
	exit = "main";
	break;
	default:
	Console.WriteLine("Invalid choice. Please select a valid option.");
	break;
	
	}return exit;
	}
	
	public override void ShowMenu()
	{
	Console.WriteLine("Select an option:");
	Console.WriteLine("1.To Add Weapon");
	Console.WriteLine("2.to Remove Weapon");
	Console.WriteLine("3.to Update Weapon");
	Console.WriteLine("4.to Show Weapon");
	Console.WriteLine("0. Exit to Main menu");
	}
	}
		to input a choice. - A switch-case structure processes the user's choice and executes corresponding actions related to weapon management. - Options include adding, deleting, updating, and showing weapons in the 'World' instance. - If the user chooses to return to the main menu (choice 0), the 'exit' variable is set to "main." - Any invalid choice results in an error message. - The 'ShowMenu' method displays the available options to the user. - The 'MenuWeapon' class encapsulates weapon menu behavior, allowing users to interact with weapon-related actions in the game.

- Class: MenuMonster
- Function: MenuFunction, ShowMenu
	public class MenuMonster : IMenu
	{
	public override string MenuFunction()
	{
	string choise = "";
	int choice;
	
	ShowMenu();
	choice = int.Parse(Console.ReadLine());
	switch (choice)
	{
	case 1:
	World.Instance.Addmonster();
	choise = "monster";
	break;
	case 2:
	World.Instance.DeleteMonster();
		- The 'MenuMonster' class represents a menu for managing monsters and inherits from the 'IMenu' class.
		
		- The 'MenuFunction' method handles user interactions within the monster menu.
		
		- Inside the method, the user is prompted to input a choice.

	choise = "monster";
	break;
	case 3:
	World.Instance.Updatemonster();
	choise = "monster";
	break;
	case 4:
	World.Instance.ShowMonsters();
	choise = "monster";
	break;
	case 0:
	choise = "main";
	break;
	default:
	Console.WriteLine("Invalid choice. Please select a valid option.");
	break;
	
	}return choise;
	}
	
	public override void ShowMenu()
	{
	Console.WriteLine("Select an option:");
	Console.WriteLine("1.To Add Monster");
	Console.WriteLine("2.to Remove Monster");
	Console.WriteLine("3.to Update Monster");
	Console.WriteLine("4.to Show Monster");
	Console.WriteLine("0. Exit");
	}
	}
		- A switch-case structure processes the user's choice and executes corresponding actions related to monster management.
		
		- Options include adding, deleting, updating, and showing monsters in the 'World' instance.
		
		- If the user chooses to return to the main menu (choice 0), the 'choise' variable is set to "main."
		
		- Any invalid choice results in an error message.
		
		- The 'ShowMenu' method displays the available options to the user.
		
		- The 'MenuMonster' class encapsulates monster menu behavior, allowing users to interact with monster-related actions in the game.

- Class: Game
- Function: StartGame, ChooseHero, RandomizeMonster, MonsterisDefeated
	public class Game
	{
	private World instance;
	List<ICharacter> listCharacter = World.Instance.Characters();
	List<Weapon> listWeapon = World.Instance.Weapons();
	List<monster> listMonster = World.Instance.Monsters();
	
	public string StartGame() // Main Game
		- The 'Game' class represents the main game logic and interactions.
		
		- The 'StartGame' method initiates the game and handles the main gameplay loop.

	{
	//add to test
	ICharacter Enma = new Knight("Enma", 2000, 20);
	listCharacter.Add(Enma);
	Weapon Excalibruh = new Weapon(2,
	"Excalibruh", "Sword", 20, 10);
	listWeapon.Add(Excalibruh);
	monster Jose = new monster("Jose", 300, 500);
	listMonster.Add(Jose);
	//store HP of Character To reset later
	int BaseHP;
	int BaseAttack;
	Console.WriteLine("----------Welcome to Epic
	Quest----------");
	Console.WriteLine("Enter to Continue");
	Console.ReadLine();
	//CHOOSE CHARACTER AND STORE IN
	VALUE PLAYER
	ICharacter Player = ChooseHero();
	BaseHP = Player.HP; // store HP of Character
	BaseAttack = Player.Damage;
	List<monster> monstersInRoom =
	RandomizeMonster();
	
	while (monstersInRoom.Count > 0 || Player.HP
	<=0)
	{
	//print 2 opponent
	Console.WriteLine(Player);
	Console.WriteLine(monstersInRoom[0]);
	// battleling
	
	Player.Attack(monstersInRoom[0]);
	Console.WriteLine("monster HP: " +
	monstersInRoom[0].HP);
	if (monstersInRoom[0].HP < 0) // if monter is
	defeated
	{
	MonsterisDefeated(monstersInRoom, Player);
	}else if(monstersInRoom[0].HP > 0)
	{
	monstersInRoom[0].Attack(Player);
	Console.WriteLine("Player HP :" +
	Player.HP);
	Console.ReadLine();
	if (Player.HP <= 0)
	{
	Console.WriteLine("Hero is down");
	Console.WriteLine("You have been
	defeated");
	Console.WriteLine("Bad End\n return to
		- Inside the loop, the player chooses a hero character using the 'ChooseHero' method.
		
		- Randomized monsters for the room are obtained using the 'RandomizeMonster' method.
		
		- The game loop manages the battle process between the player's character and monsters.
		
		- When a monster is defeated, the 'MonsterisDefeated' method handles the outcome, such as removing the monster from the room, dropping a weapon, and equipping the weapon to the player.
		
		- If the player's character is defeated or all monsters are defeated, the game loop ends.
		
		- After the loop, the player's character is reset to its initial state, and a return value indicating whether the player wants to return to the main menu is provided.
		
		- The 'Game' class encapsulates the core gameplay

	Main menu");
	return "main";
	}
	}
	if(monstersInRoom.Count == 0)
	{
	Console.WriteLine("your quest is complete would you like to try again");
	}
	
	}
	Player.HP = BaseHP;
	Player.EquippedWeapon = null;
	Player.Damage = BaseAttack;
	return "main";
	}
	
	public ICharacter ChooseHero()
	{
	World.Instance.showCharacter();
	Console.WriteLine("choose your Hero by Name");
	string name = Console.ReadLine().ToLower();
	ICharacter character = null;
	for (int i = 0; i < listCharacter.Count; i++)
	{
	if (listCharacter[i].CharacterName.ToLower().Equals(name))
	{
	character = listCharacter[i];
	Console.WriteLine($"your hero is {listCharacter[i].CharacterName}");
	}
	}
	return character;
	}
	public List<monster> RandomizeMonster()
	{
	List<monster> list = new List<monster>();
	Random random = new Random();
	for (int i = 0; i < listMonster.Count; i++)
	{
	int randomIndex = random.Next(0, listMonster.Count);
	list.Add(listMonster[randomIndex]);
	}
	return list;
	
	}
	
	public void MonsterisDefeated(List<monster> monstersInRoom, ICharacter Player)
		logic, character selection, battling, and outcome handling.

	{
	Random random = new Random();
	Console.WriteLine("Monster is Down");
	monstersInRoom.RemoveAt(0);
	Console.WriteLine("it Drop something");
	int ramdomindex = random.Next(0, listWeapon.Count);
	Console.WriteLine($"it a {listWeapon[ramdomindex]}");
	Player.EquipWeapon(listWeapon[ramdomindex]);
	}
	}	

Abstract Class: character
Functions: Attack, Equip weapon
Class : Knight, Archer, Assasin, Mage. Functions: Attack,Equip Weapon
	public abstract class ICharacter
	{
	//properties
	private int _id;
	private string _name;
	private int _hp;
	private int _damage;
	Weapon _weapon = null;
	public static int temp = 0;
	//getter and setter
	public int CharacterID
	{
	get { return _id; }
	private set { _id = value; }
	}
	
	public string CharacterName
	{
	get { return _name; }
	set { _name = value; }
	}
	
	public int HP
	{
	get { return _hp; }
	set { _hp = value; }
	}
	
	public int Damage
	{
	get { return _damage; }
	set { _damage = value; }
	}
	
	public Weapon EquippedWeapon
	{
	get { return _weapon; }
	set { _weapon = value; }
	}
		The ICharacter class is an abstract base class representing a character in the game. It contains properties for character attributes, an equipped weapon, and methods for attacking and equipping weapons.
		
		Each derived class (Knight, Archer, Assassin, and Mage) represents a specific type of character and inherits from ICharacter. These classes implement the abstract methods for attacking and equipping weapons.
		
		The Attack method is implemented in each derived class and represents the character's attack action against a monster.
		
		The EquipWeapon method is also implemented in each derived class. It allows a character

	//constructor
	
	public ICharacter(string characterName, int hP, int damage)
	{
	CharacterID = ++temp;
	CharacterName = characterName;
	HP = hP;
	Damage = damage;
	EquippedWeapon = null;
	}
	
	//abtract method
	public abstract void Attack(monster monster);
	public abstract void EquipWeapon(Weapon weapon);
	//override
	public override string ToString()
	{
	return String.Format("CharacterID: {0}, CharacterName: {1}, HP: {2}, Damage: {3}, Equipped Weapon: {4}",
	this.CharacterID, this.CharacterName, this.HP, this.Damage, this.EquippedWeapon);
	}
	
	}
	
	public class Knight : ICharacter
	{
	// properties
	const double increaseHP = 1.3;
	//constructor
	public Knight(string name) : this(name, 1000, 200) { }
	public Knight(string characterName, int hP, int damage) : base(characterName, hP, damage)
	{
	
	}
	//abtract method
	public override void Attack(monster monster)
	{
	monster.HP -= this.Damage;
	Console.WriteLine($"{this.CharacterName} attack {monster.Name} Damage: {this.Damage}");
	}
	
	public override void EquipWeapon(Weapon weapon)
	{
	if (checkWeapon(weapon, this))
	{
		to equip a weapon if it matches the character's class.
		
		The ToString method is overridden in the ICharacter class to provide a formatted string representation of a character's attributes.
		
		Each derived class (Knight, Archer, Assassin, Mage) has similar structures, with properties, constructors, Attack methods, and EquipWeapon methods specific to their respective character types.

```
            this.EquippedWeapon = weapon;
        }
        //increase status
        this.Damage += weapon.WDamage;
        this.HP += weapon.WHP;
    }
    public static bool checkWeapon(Weapon weapon,
ICharacter s)
    {
        if (weapon.Type.Equals("Sword"))
        {
            Console.WriteLine($"{s.CharacterName} is
using {weapon}");
            return true;
        }
        else
        {
            Console.WriteLine("your class is not able to use
this weapon");
            return false;
        }
    }
}
}
// class Archer
public class Archer : ICharacter
{
    // properties
    const double increaseHP = 1.3;

    // constructor

    public Archer(string characterName, int hP, int
damage) : base(characterName, hP, damage)
    {
    }

    // override the abstract method Attack
    public override void Attack(monster monster)
    {
        monster.HP -= this.Damage;
        Console.WriteLine($"{this.CharacterName} attacks
{monster.Name} for {this.Damage} damage.");
    }
    // override the abstract method EquipWeapon
    public override void EquipWeapon(Weapon weapon)
    {
        if (weapon.Type == "Bow")
        {
            this.EquippedWeapon = weapon;
            Console.WriteLine($"{this.CharacterName}
```

```
equipped {weapon.Wname}.");
        }
        else
        {
            Console.WriteLine($"{this.CharacterName}
cannot equip {weapon.Wname}. Archer can only equip
bows.");
        }
    }
    public static bool checkWeapon(Weapon weapon,
ICharacter s)
    {
        if (weapon.Type.Equals("Sword"))
        {
            Console.WriteLine($"{s.CharacterName} is using
{weapon}");
            return true;
        }
        else
        {
            Console.WriteLine("your class is not able to use
this weapon");
            return false;
        }
    }
}

// class Assasin
public class Assasin : ICharacter
{
    // properties
    const double increaseHP = 1.3;

    // constructor
    public Assasin(string characterName, int hP, int
damage) : base(characterName, hP, damage)
    {
    }
    // override the abstract method Attack
    public override void Attack(monster monster)
    {
        monster.HP -= this.Damage;
        Console.WriteLine($"{this.CharacterName} attacks
{monster.Name} for {this.Damage} damage.");
    }
    // override the abstract method EquipWeapon
    public override void EquipWeapon(Weapon weapon)
    {
        if (weapon.Type == "Dagger")
        {
            this.EquippedWeapon = weapon;
```

```
            Console.WriteLine($"{this.CharacterName}
equipped {weapon.Wname}.");
        }
        else
        {
            Console.WriteLine($"{this.CharacterName}
cannot equip {weapon.Wname}. Assasin can only equip
daggers.");
        }
    }
    public static bool checkWeapon(Weapon weapon,
ICharacter s)
    {
        if (weapon.Type.Equals("Sword"))
        {
            Console.WriteLine($"{s.CharacterName} is using
{weapon}");
            return true;
        }
        else
        {
            Console.WriteLine("your class is not able to use
this weapon");
            return false;
        }
    }
}
// class Mage
public class Mage : ICharacter
{
    // properties
    const double increaseHP = 1.3;
    // constructor
    public Mage(string characterName, int hP, int damage) :
base(characterName, hP, damage)
    {
    }

    // override the abstract method Attack
    public override void Attack(monster monster)
    {
        monster.HP -= this.Damage;
        Console.WriteLine($"{this.CharacterName} attacks
{monster.Name} for {this.Damage} damage.");
    }
    // override the abstract method EquipWeapon
    public override void EquipWeapon(Weapon weapon)
    {
        if (weapon.Type == "Staff")
        {
            this.EquippedWeapon = weapon;
```

	Console.WriteLine($"{this.CharacterName} equipped {weapon.Wname}.");
	}
	else
	{
	Console.WriteLine($"{this.CharacterName} cannot equip {weapon.Wname}. Mage can only equip staffs.");
	}
	}	

- Class: monster
- Function: Attack
	public class monster
	{
	private int _id;
	private string _name;
	private int _hp;
	private int _damage;
	public static int temp = 0;
	// Constructor
	public monster(string name, int hp, int damage)
	{
	_id = temp++;
	_name = name;
	_hp = hp;
	_damage = damage;
	}
	
	// Getters and Setters
	public int ID
	{
	get { return _id; }
	private set { _id = value; }
	}
	
	public string Name
	{
	get { return _name; }
	set { _name = value; }
	}
	
	public int HP
	{
	get { return _hp; }
	set { _hp = value; }
	}
	
	public int Damage
	{
	get { return _damage; }
	set { _damage = value; }
	}
	public void Attack(ICharacter character)
		The monster class represents a monster in the game. It has properties for the monster's ID, name, HP (hit points), and damage. The temp variable is used to generate unique IDs for each monster created.
		
		The constructor initializes the properties of the monster when an instance is created.
		
		Getter and setter methods are provided for the properties.
		
		The Attack method allows the monster to attack a character by reducing the character's HP based on the monster's damage. A message is displayed indicating the attack.
		
		The ToString method is overridden to provide a formatted string representation of the monster's attributes.

	{
	character.HP -= this.Damage;
	Console.WriteLine($"{this.Name} attack {character.CharacterName} Damage:{this.Damage}");
	}
	public override string ToString()
	{
	return String.Format($"Monster:{this.Name} |HP: {this.HP} |Damage: {this.Damage} |");
	}	

- Class: Weapon	public class Weapon
	{
	//properties
	private int _wid;
	private string _wname;
	private string _type;
	private int _damage;
	private int _hp;
	//get set
	public int WID
	{
	get { return _wid; }
	private set { _wid = value; }
	}
	
	public string Wname
	{
	get { return _wname; }
	set { _wname = value; }
	}
	
	public string Type
	{
	get { return _type; }
	set { _type = value; }
	}
	
	public int WDamage
	{
	get { return _damage; }
	set { _damage = value; }
	}
	public int WHP
	{
	get { return _hp; }
	set { _hp = value; }
	}
	//constructors
	public Weapon(int wID, string wname, string type, int damage, int hP)
	{
		The Weapon class represents a weapon in the game. It has properties for the weapon's ID, name, type, damage, and HP (hit points).
		
		Getter and setter methods are provided for the properties.
		
		The constructor initializes the properties of the weapon when an instance is created.
		
		The ToString method is overridden to provide a formatted string representation of the weapon's name.

	WID = wID;
	Wname = wname;
	Type = type;
	WDamage = damage;
	WHP = hP;
	}
	public override string ToString()
	{
	return String.Format("{0}", this.Wname);
	}
	}	

- Class: World
-Function:
AddCharacter,
AddMonster,
CreateWeapon,
UpdateCharacter,
Updatemonster,
UpdateWeapon,
DeleteCharacter,
DeleteMonster,
DeleteWeapon,
showCharacter,
ShowMonsters,
ShowWeapons
	public sealed class World
	{
	private static World instance;
	private List<ICharacter> characters;
	private List<Weapon> weapons;
	private List<monster> monters;
	
	// RETURN LIST
	
	private World() // create List
	{
	characters = new List<ICharacter>();
	weapons = new List<Weapon>();
	monters = new List<monster>();
	}
	
	public List<ICharacter> Characters()
	{
	return characters;
	}
	public List<Weapon> Weapons()
	{
	return weapons;
	}
	public List<monster> Monsters()
	{
	return monters;
	}
	
	public static World Instance //INstance
	{
	get
	{
	if (instance == null)
	{
	instance = new World();
	}
	return instance;
	}
	}
		The World class is designed as a singleton using the Singleton design pattern, ensuring there's only one instance of the class throughout the program's execution.
		
		It contains private lists (characters, weapons, and monsters) to store instances of characters, weapons, and monsters in the game.
		
		The constructor is private to prevent direct instantiation of the class. Instead, you can only access an instance using the Instance property.
		
		Getter methods (Characters(), Weapons(), and Monsters()) allow external access to the private lists.
		
		The Instance property provides a way to access the

```
        // ADD CHARACTER
        public void AddCharacter()
        {
            Console.WriteLine("input Character Name");
            string characterName = Console.ReadLine();
            Console.WriteLine("input Character HP");
            int HP = int.Parse(Console.ReadLine());
            Console.WriteLine("input Character Damage");
            int Damage = int.Parse(Console.ReadLine());

            Console.WriteLine("Input Character Class ");
            Console.WriteLine("Class Knight || Class  Mage ||
Class Archer || Assasin");
            //choose Class
            string clas = Console.ReadLine().ToLower();
            if (clas.Equals("knight"))
            {
                ICharacter s = new Knight(characterName, HP,
Damage);
                characters.Add(s);
            }
            else if (clas.Equals("archer"))
            {
                ICharacter s = new Archer(characterName, HP,
Damage);
                characters.Add(s);
            }
            else if (clas.Equals("assasin"))
            {
                ICharacter s = new Assasin(characterName, HP,
Damage);
                characters.Add(s);
            }
            else if (clas.Equals("mage"))
            {
                ICharacter s = new Mage(characterName, HP,
Damage);
                characters.Add(s);
            }
            else
            {
                Console.WriteLine("Your character must in one
of 4 class |Knight|Mage|Archer|Assasin|");
            }
        }

        //CREATE MONSTER
        public void Addmonster()
        {
```

singleton instance of the World class. If an instance doesn't exist, it's created; otherwise, the existing instance is returned.

Other methods (like AddCharacter, AddMonster, CreateWeapon, UpdateCharacter, Updatemonster, UpdateWeapon, DeleteCharacter, DeleteMonster, DeleteWeapon, showCharacter, ShowMonsters, ShowWeapons) perform various operations related to characters, monsters, and weapons within the game.

```
            Console.WriteLine("Input monster Name:");
            string name = Console.ReadLine();
            Console.WriteLine("Input monster HP:");
            int hp = int.Parse(Console.ReadLine());
            Console.WriteLine("Input monster Damage:");
            int damage = int.Parse(Console.ReadLine());

            // Create a new monster with the provided details
            monster newmonster = new monster(name, hp,
damage);

            // Add the new monster to the queue
            monters.Add(newmonster);

            Console.WriteLine("monster added successfully.");
        }

        // CREATE WEAPON
        public void CreateWeapon()
        {
            Console.WriteLine("Input weapon Name");
            string Name = Console.ReadLine();
            Console.WriteLine("Input weapon Type");
            string Type = Console.ReadLine();
            Console.WriteLine("Input weapon HP");
            int HP = int.Parse(Console.ReadLine());
            Console.WriteLine("Input weapon Damage");
            int Damage = int.Parse(Console.ReadLine());
            int newWeaponID =
World.Instance.weapons.Count + 1;

            Weapon newWeapon = new
Weapon(newWeaponID, Name, Type, Damage, HP);
            World.Instance.weapons.Add(newWeapon);
        }

        //UPDATE MENTHODS

        //update HEORES
        public void UpdateCharacter()
        {
            Console.WriteLine("Input ID of character to
Update");
            int ID = int.Parse(Console.ReadLine());
            foreach (ICharacter c in instance.characters)
            {
                if (c.CharacterID == ID)
                {
                    Console.WriteLine("input character name:");
                    c.CharacterName = Console.ReadLine();
                    Console.WriteLine("input character HP");
```

```
                c.HP = int.Parse(Console.ReadLine());
                Console.WriteLine("input character
Damage");
                c.Damage = int.Parse(Console.ReadLine());
            }
        }
    }

    //UPDATE MONSTER
    public void Updatemonster()
    {
        Console.WriteLine("Input ID of the monster to
update:");
        int idToUpdate = int.Parse(Console.ReadLine());

        // Find the monster with the specified ID in the
queue
        monster monsterToUpdate = null;
        foreach (monster monster in instance.monters)
        {
            if (monster.ID == idToUpdate)
            {
                monsterToUpdate = monster;
                break;
            }
        }
        if (monsterToUpdate == null)
        {
            Console.WriteLine("monster with the provided
ID not found!");
            return;
        }

        Console.WriteLine("Input monster Name:");
        monsterToUpdate.Name = Console.ReadLine();
        Console.WriteLine("Input monster HP:");
        monsterToUpdate.HP =
int.Parse(Console.ReadLine());
        Console.WriteLine("Input monster Damage:");
        monsterToUpdate.Damage =
int.Parse(Console.ReadLine());
        Console.WriteLine("monster updated
successfully.");
    }

    //UPDATE WEAPONS
    public void UpdateWeapon()
    {
        Console.WriteLine("Input ID of weapon to
update");
        int ID = int.Parse(Console.ReadLine());
```

```
            Weapon weaponToUpdate =
instance.weapons.Find(w => w.WID == ID);
            if (weaponToUpdate == null)
            {
                Console.WriteLine("Weapon with ID " + ID + "
not found!");
                return;
            }
            Console.WriteLine("Input weapon Name:");
            weaponToUpdate.Wname = Console.ReadLine();
            Console.WriteLine("Input weapon Type:");
            weaponToUpdate.Type = Console.ReadLine();
            Console.WriteLine("Input weapon HP:");
            weaponToUpdate.WHP =
int.Parse(Console.ReadLine());
            Console.WriteLine("Input weapon Damage:");
            weaponToUpdate.WDamage =
int.Parse(Console.ReadLine());
        }

        //DELETE METHODS

        //DELETE character with ID
        public void DeleteCharacter()
        {
            Console.WriteLine("Input ID of character to
Delete");
            int ID = int.Parse(Console.ReadLine());

            // Use RemoveAll method to remove all characters
with matching CharacterID
            instance.characters.RemoveAll(c => c.CharacterID
== ID);

            Console.WriteLine("Character(s) with ID " + ID +
" deleted successfully.");
        }

        //DELETE MONSTER WITH ID
        public void DeleteMonster()
        {
            Console.WriteLine("Input ID of weapon to
delete");
            int ID = int.Parse(Console.ReadLine());

            monster monster = instance.monters.Find(m =>
m.ID == ID);
            if (monster == null)
            {
                Console.WriteLine("Weapon with ID " + ID + "
```

```
not found!");
                return;
            }

            instance.monters.Remove(monster);
            Console.WriteLine("monster with ID " + ID + "
deleted successfully.");
        }

        // DELETE WEAPON
        public void DeleteWeapon()
        {
            Console.WriteLine("Input ID of weapon to
delete");
            int ID = int.Parse(Console.ReadLine());

            Weapon weaponToDelete =
instance.weapons.Find(w => w.WID == ID);
            if (weaponToDelete == null)
            {
                Console.WriteLine("Weapon with ID " + ID + "
not found!");
                return;
            }

            instance.weapons.Remove(weaponToDelete);
            Console.WriteLine("Weapon with ID " + ID + "
deleted successfully.");
        }

        // SHOW METHODS

        //SHOW HERO
        public void showCharacter()
        {
            foreach (ICharacter c in instance.characters)
            {
                Console.WriteLine(c);
            }
        }

        //SHOW MONSTERS
        public void ShowMonsters()
        {
            Console.WriteLine("---- Monsters List ----");
            foreach (monster monster in instance.monters)
            {
                Console.WriteLine($"Monster ID:
{monster.ID}, Name: {monster.Name}, HP:
{monster.HP}, Damage: {monster.Damage}");
            }
```

	}
	
	//SHOW WEAPONS
	public void ShowWeapons()
	{
	Console.WriteLine("----Weapons List----");
	foreach (Weapon weapon in instance.weapons)
	{
	Console.WriteLine($"Weapon ID: {weapon.WID}, Name: {weapon.Wname}, Type: {weapon.Type}, HP: {weapon.WHP}, Damage: {weapon.WDamage}");
	}
	}	

3.2. Program screenshots

```
1.To CharacterMenu
2.To Weapon Menu
3.To Room Menu
4.Start Game
input your choice
1
```

This is main menu I will go to Character menu

```
Select an option:
1.To Add Character
2.to Remove Character
3.to Update Character
4.to Show Character
0. Exit to Main menu
1
```

Choose 1 to add character

```
input Character Name
Enma
input Character HP
2000
input Character Damage
20
Input Character Class
Class Knight || Class  Mage || Class Archer || Assasin
Knight
```

Input character information

```
Select an option:
1.To Add Character
2.to Remove Character
3.to Update Character
4.to Show Character
0. Exit to Main menu
4
CharacterID: 1, CharacterName: Enma, HP: 2000, Damage: 20, Equipped Weapon:
```

Show the character which is just input

```
Select an option:
1.To Add Character
2.to Remove Character
3.to Update Character
4.to Show Character
0. Exit to Main menu
3
Input ID of character to Update
1
input character name:
Jose
input character HP
8000
input character Damage
99
Select an option:
1.To Add Character
2.to Remove Character
3.to Update Character
4.to Show Character
0. Exit to Main menu
4
CharacterID: 1, CharacterName: Jose, HP: 8000, Damage: 99, Equipped Weapon:
```

Update Character with ID 1

```
Select an option:
1.To Add Character
2.to Remove Character
3.to Update Character
4.to Show Character
0. Exit to Main menu
2
Input ID of character to Delete
1
Character(s) with ID 1 deleted successfully.
Select an option:
1.To Add Character
2.to Remove Character
3.to Update Character
4.to Show Character
0. Exit to Main menu
4
Select an option:
```

Remove character with ID 1

```
1.To CharacterMenu
2.To Weapon Menu
3.To Room Menu
4.Start Game
input your choice
2
Select an option:
1.To Add Weapon
2.to Remove Weapon
3.to Update Weapon
4.to Show Weapon
0. Exit to Main menu
1
Input weapon Name
Excalibur
Input weapon Type
Sword
Input weapon HP
500
Input weapon Damage
20
```

Add a new Weapon

```
1.To CharacterMenu
2.To Weapon Menu
3.To Room Menu
4.Start Game
input your choice
3
Select an option:
1.To Add Monster
2.to Remove Monster
3.to Update Monster
4.to Show Monster
0. Exit
1
Input monster Name:
Tuna
Input monster HP:
800
Input monster Damage:
45
monster added successfully.
```

Add a monster

```
1.To CharacterMenu
2.To Weapon Menu
3.To Room Menu
4.Start Game
input your choice
4
---------------Welcome to Epic Quest---------------
Enter to Continue
```

In main menu choose 4 to Start the game

```
---------------Welcome to Epic Quest---------------
Enter to Continue

CharacterID: 2, CharacterName: Enma, HP: 2000, Damage: 50, Equipped Weapon:
choose your Hero by Name
Enma
your hero is Enma
CharacterID: 2, CharacterName: Enma, HP: 2000, Damage: 50, Equipped Weapon:
Monster:Jose |HP: 300 |Damage: 30 |
```

Choose hero Enma

```
nma attack Jose Damage: 50
onster HP: 250
ose attack Enma Damage:30
layer HP :1970

haracterID: 2, CharacterName: Enma, HP: 1970, Damage: 50, Equipped Weapon:
onster:Jose |HP: 250 |Damage: 30 |
nma attack Jose Damage: 50
onster HP: 200
ose attack Enma Damage:30
layer HP :1940

haracterID: 2, CharacterName: Enma, HP: 1940, Damage: 50, Equipped Weapon:
onster:Jose |HP: 200 |Damage: 30 |
nma attack Jose Damage: 50
onster HP: 150
ose attack Enma Damage:30
layer HP :1910

haracterID: 2, CharacterName: Enma, HP: 1910, Damage: 50, Equipped Weapon:
onster:Jose |HP: 150 |Damage: 30 |
nma attack Jose Damage: 50
onster HP: 100
ose attack Enma Damage:30
layer HP :1880
```

Monster and Player take turn to Attack

```
Monster is Down
it Drop something
it a Excalibruh
Enma is using Excalibruh
your quest is complete would you like to try again
1.To CharacterMenu
2.To Weapon Menu
3.To Room Menu
4.Start Game
input your choice
```

Monster HP is drop to 0 and there is no Monster in the list Monster drop Item Enma is suit for the item Equip the item the Quest is compete due to no more monster in the list.

3.3. Test Plan

No	TEST CASE	FUNCTION	INPUT	EXPECT	ACTUAL	RESULT

			DATA	ED OUTPUT	OUTPUT	
1	Verify that program adds character when user inputs valid information	Add Character	Input: Enma, 2000, 20	Expected Output: Add success	Actual Output: Add success	Pass
2	Verify that program removes character when user inputs valid ID	Remove Character	Input: Character ID	Expected Output: Remove success	Actual Output: Remove success	Pass
3	Verify that program updates character when user inputs valid ID and new information	Update Character	Input: 1, Joe , 1500, 50	Expected Output: Update success	Actual Output: Update success	Pass
4	Verify that program displays the list of characters	Show Characters	N/A	Expected Output: List of characters displayed	Actual Output: List of characters displayed	Pass
5	Verify that a weapon is added successfully	Add Weapon Method	Input: 1, Excalibur, Sword, 50, 100	Expected Output: Add success	Actual Output: Add success	Pass
6	Verify that a weapon is removed successfully by ID	Remove Weapon Method	Input: 1,	Expected Output: Remove success	Actual Output: Remove success	Pass
7	Verify that a weapon is updated successfully by ID	Update Weapon Method	Input: 1, Carrot sword, Sword, 60, 120	Expected Output: Update success	Actual Output: Update success	Pass
8	Verify that a weapon is removed successfully by ID	Remove Weapon Method	Input: 1	Expected Output: Remove success	Actual Output: Remove success	Pass
9	Add Monster	AddMonster	"Jose", 300, 500	Add success	Add success	Pass
10	Remove Monster	RemoveMonster	0	Monster with ID 0 deleted successfully.	Monster with ID 0 deleted successfully.	Pass
11	Update Monster	UpdateMonst	0, "JHin",	Monster	Monster	Pass

		er	400, 600	updated successfully.	updated successfully.	
12	Display List of Monsters	ShowMonsters	N/A	List of monsters displayed	List of monsters displayed	Pass
13	Verify that the Main menu will return to the main menu when the user inputs an invalid choice	MainMenu	Input: 99	"Invalid choice. Please select a valid option."	"Invalid choice. Please select a valid option."	Pass
14	Verify that the main menu will move to the Character menu when the user inputs '1'	MainMenu	Input: 1	Expected Output: Move to Character Menu	Actual Output: Character Menu displayed	Pass
15	Verify that the main menu will move to the Weapon menu when the user inputs '2'	MainMenu	Input: 2	Expected Output: Move to Weapon Menu	Actual Output: Weapon Menu displayed	Pass
16	Verify that the main menu will move to the Monster menu when the user inputs '3'	MainMenu	Input: 3	Expected Output: Move to Monster Menu	Actual Output: Monster Menu displayed	Pass
17	Verify that the main menu will start the game when the user inputs '4'	MainMenu	Input: 4	Expected Output: Game starts	Actual Output: Game started	Pass
18	Verify that the user can choose a hero when inputting a valid name of a character	ChooseHero	Input: Enma	Expected Output: Enma	Actual Output: Enma	Pass
19	Verify that the game will end when the user's HP drops to 0	StartGame	N/A	Expected Output: Display "You have been defeated"	Actual Output: Displayed "You have been defeated"	Pass
20	Verify that the game will end when the user defeats all	StartGame	User defeats all monsters	Expected Output: Display "Your	Actual Output: Displayed "Your	Pass

	monsters			quest is complete. Would you like to try again?"	quest is complete. Would you like to try again?"	

4. Discussion

4.1. Range of similar patterns

Singleton pattern:

The Singleton pattern ensures that a class has just one instance and provides a global way to access that instance. It involves creating a private constructor to prevent external instantiation and a static method or property that controls the instance creation process. This method or property checks if an instance exists; if not, it creates one, and if it does, it returns the existing instance. Implementing thread-safe mechanisms guarantees proper functionality in multi-threaded environments. Singleton simplifies resource management by allowing a single point of control for shared resources, enhancing efficiency and organization in the codebase. (papers, n.d.)

Factory pattern:

The Factory pattern is a design pattern that enables the creation of objects without specifying their exact class. It involves creating a separate class, the factory, responsible for object creation based on certain conditions or parameters. This pattern centralizes the object creation process, enhancing flexibility and reducing coupling between the client code and the object classes. Clients request objects from the factory without needing to know their concrete implementations. This promotes code reusability, maintenance, and scalability, as modifications to object creation logic can be confined to the factory class, without impacting the client code. Overall, the Factory pattern streamlines object creation, making code more adaptable and manageable. (Erich Gamma, 1994)

Builder pattern:

The Builder pattern separates the construction of complex objects from their representation. It uses an abstract builder interface and concrete builder classes to create objects step by step, allowing for easy configuration of optional properties. A director class coordinates the building process. This pattern is ideal for objects with many options and promotes readable and maintainable code by avoiding large constructors. It simplifies object creation by providing a structured way to build objects with different features (Erich Gamma, 1994)

Façade Pattern:

The Facade pattern is a structural pattern that provides a unified interface to a set of interfaces in a subsystem. It simplifies the interaction with complex systems by offering a high-level interface that encapsulates the details and complexities of the underlying components. The Facade pattern is often used to

hide the intricacies of a system and provide a more user-friendly way to access its functionalities. (Erich Gamma, 1994)

Iterator Pattern:

The Iterator pattern is a behavioral pattern that provides a way to access the elements of a collection sequentially without exposing its underlying representation. It abstracts the process of traversal, allowing clients to iterate over the elements of a collection without needing to know its internal structure. This pattern separates the concerns of iteration from the underlying collection, promoting better code organization and reusability. (Erich Gamma, 1994)

Adapter pattern:

The Adapter pattern is a structural design pattern that allows objects with incompatible interfaces to work together. It involves creating an adapter class that acts as an intermediary, converting the interface of one class into an interface expected by the client. This enables objects with different interfaces to collaborate seamlessly. The Adapter pattern is particularly useful when integrating legacy code or third-party libraries into your application. It helps ensure that the client code doesn't need to change when dealing with different interfaces, promoting code reusability and flexibility. (Erich Gamma, 1994)

1. References

Erich Gamma, R. H. R. J. J. V., 1994. Design Patterns: Elements of Reusable Object-Oriented Software. In: *Design Patterns: Elements of Reusable Object-Oriented Software.* s.l.:s.n., p. Approximately 395 pages.

flm, n.d. [Online]
Available at: https://flm.greenwich.edu.vn/gui/role/student/SyllabusDetails?sylID=2543

papers, n.d. [Online]
Available at: https://papers.ssrn.com/sol3/papers.cfm?abstract_id=3439925