Final OOP Project Design Report

Student: Fuad Ismayilbayli
Project Title: RPG Battle Simulator(actually I still smth from BG3)

Date : 20.11.2025

https://github.com/fudik94/battle_simulator.git

1. Project Introduction and Overview

This project is a **simple console RPG battle simulator** inspired by *Baldur's Gate 3*.
It allows the creation of heroes with different **classes** (Wizard, Fighter, Cleric, Barbarian, Druid, Warlock) and **races** (Elf, Human, Half-Elf, Tiefling, Githyanki).

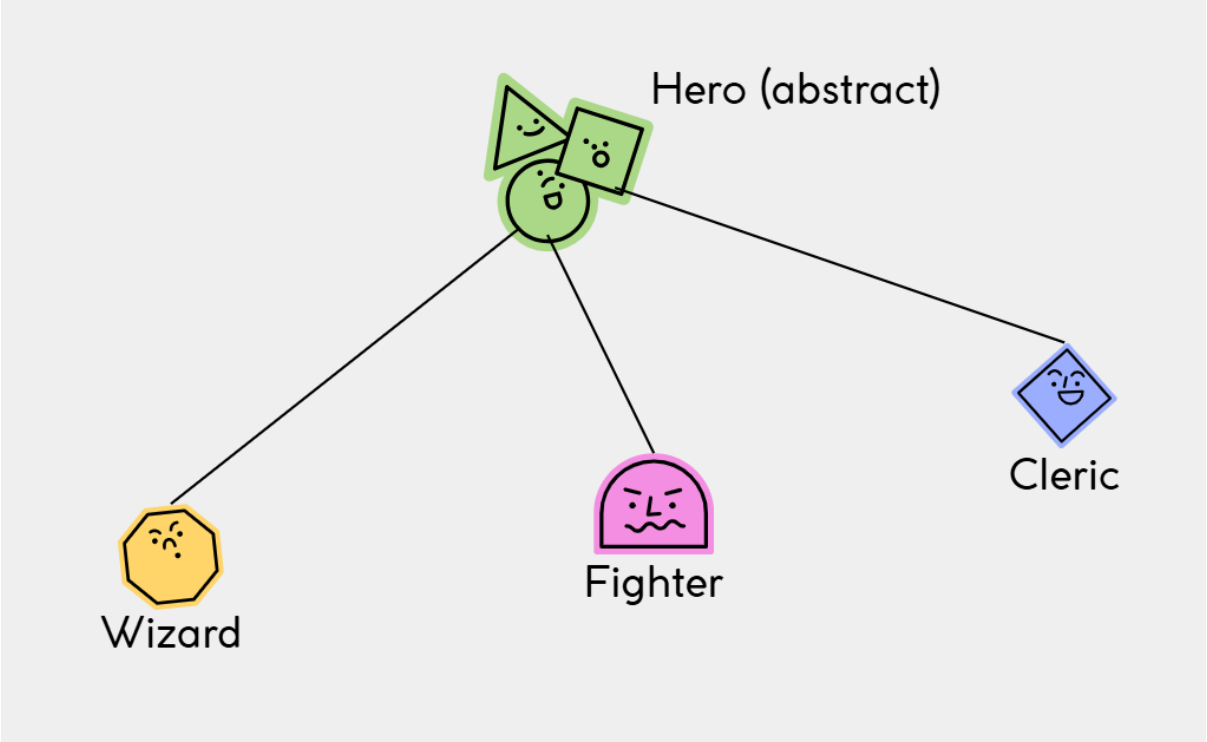
Each hero has **health, damage, and a special ability**. The program simulates a **turn-based fight** between two heroes, showing attacks, healing, and special moves.

The goal of the project is educational: to **demonstrate Object-Oriented Programming (OOP)** in Python, using classes, inheritance, polymorphism, abstraction, and a design pattern.
This project can be used by students to understand OOP concepts or by developers to experiment with small game mechanics.

The project is implemented in **Python 3** using only the standard library (abc for abstract classes, random for turn simulation, and unittest for testing).

2. Class Design and Relationships

The project uses several classes to represent game entities.
The main structure is shown below:



Class	Description	Attributes	Methods
Hero	Abstract base class. Defines shared attributes and actions for all heroes.	name, hero_class, race, health, max_health, damage	hit(), heal(), set_damage(), use_special() (abstract)
Wizard, Fighter, Cleric, Barbarian, Druid, Warlock	Subclasses of Hero. Each class has its own special attack and starting stats.	Inherits from Hero	Implements use_special() with unique behavior
HeroFactory	Creates hero objects without exposing class creation logic.	–	create_hero(hero_type, name, race)

Design decisions:

- Hero is abstract to **enforce implementation** of use_special() in subclasses.
- Each subclass has **specific behavior** for special attacks (polymorphism).
- Race bonuses are stored in a **dictionary**, separated from class logic.
- Only six hero classes were implemented to **keep the project manageable**.

3. Application of OOP Principles

The project applies all four main principles of OOP:

1. Encapsulation

All hero data (health, damage) is stored inside the class.
It can only be changed using specific methods such as hit() or heal().
This protects the internal state of objects.

```
def hit(self, target):
    target.health -= self.damage
```

2. Inheritance

All character types (Wizard, Fighter, Cleric) inherit from the base class Hero.
This allows them to reuse common attributes and methods.

```
class Fighter(Hero):
    ...
```

3. Polymorphism

Different hero classes share the same method name use_special(), but each one behaves differently.
This allows the same interface for different actions.

```
hero.use_special(enemy)
```

4. Abstraction

The class Hero is declared as **abstract** using the ABC module.
It defines the interface that all heroes must follow.

```
from abc import ABC, abstractmethod
class Hero(ABC):
    @abstractmethod
    def use_special(self, target):
        pass
```

This makes the code more structured and easier to extend with new hero types.

4. Use of Design Patterns

Factory Pattern is used to create hero instances without exposing the creation logic.

```
gale = HeroFactory.create_hero("Gale", "Wizard", "Elf")
```

This makes it easy to **add new hero types** in the future without changing the main program logic.

5. Explanation of SOLID Principles

*** Single Responsibility Principle (SRP):**
Each class has one responsibility: Hero manages hero stats, subclasses manage special abilities, HeroFactory handles creation.

*** Open/Closed Principle (OCP):**
New hero classes can be added by creating a new subclass of Hero without modifying existing classes.

*** Liskov Substitution Principle (LSP):**
Subclasses of Hero can be used wherever Hero is expected. For example, use_special() can be called on any hero.

6. Testing Strategy

The project uses **unittest** for testing:

*Test normal attacks (hit() reduces health correctly)

*Test healing (heal() does not exceed max_health)

*Test special abilities (use_special() has expected effects)

Example test:

```
def test_heal():
  hero = Wizard("Gale", "Elf")
  hero.health = 50
  hero.heal(30)
  assert hero.health == 80  # max_health
```

7. Key Design Decisions and Challenges

***Limited classes**: Only six hero types to keep the project small and focused.

***Race bonuses**: Implemented with a dictionary to separate data and logic.

***Console interface**: Chosen for simplicity; graphical version would be larger and harder to test.

***Random attacks**: Each turn may use normal or special attack for variability.

8. Conclusion

This project demonstrates a **working example of OOP in Python** with multiple classes, inheritance, polymorphism, and abstraction.
It also shows practical use of a **design pattern (Factory)** and **SOLID principles**.

Future improvements could include:

* Adding more hero classes and races

* Implementing items or weapons

* Adding a turn-based interface for multiplayer battles

Overall, the project helped understand **how to structure code clearly** and **organize complex interactions between objects**.