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 small, self-contained RPG battle simulator implemented in Python.
It models turn-based combat between characters (heroes and monsters), demonstrating core OOP concepts and providing a compact, testable game loop. The simulator is intended as an educational assignment: small scope, complete features, and easy to demo.

Main features:

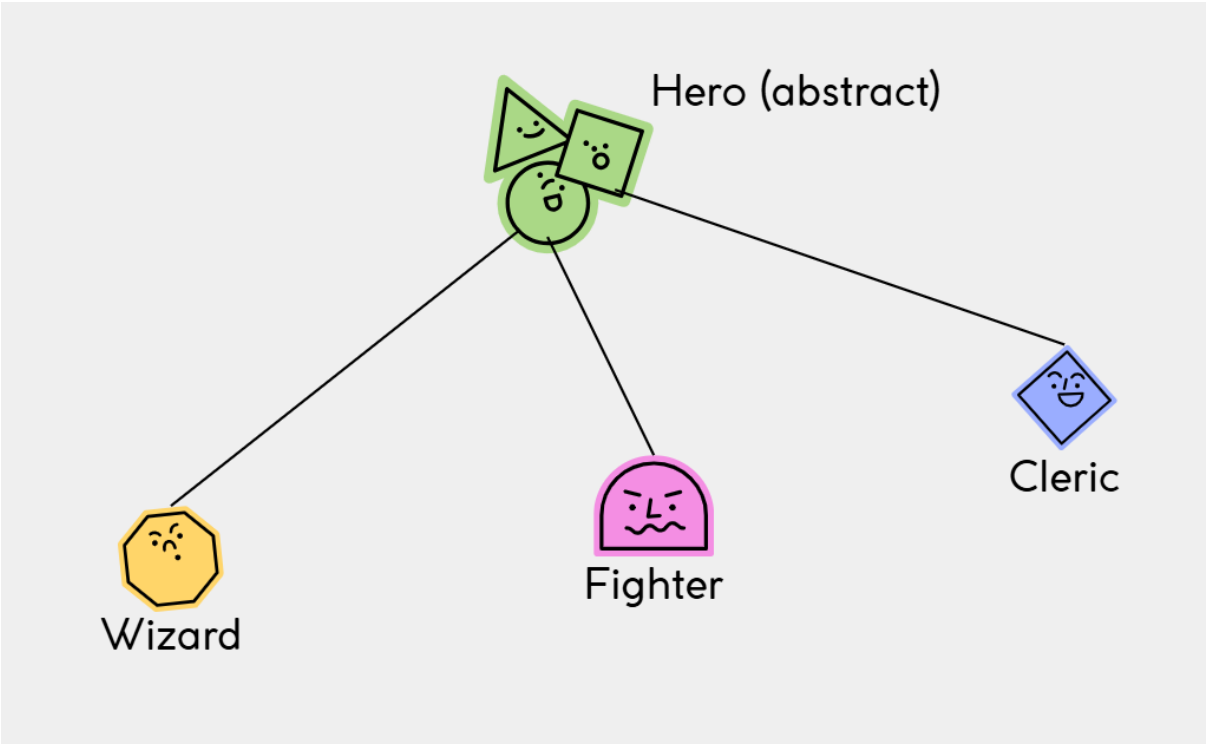
- Creation of heroes of different classes and races (race bonuses for health/damage).
- Turn-based battle loop with normal attacks and special abilities.
- Centralized health management (take_damage / heal).
- Simple Factory for creating hero instances.
- Unit tests (pytest) verifying core behaviors.

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:



List of main classes and responsibilities:

- **Hero (abstract)**
    - **Abstract base class defining the common interface and shared logic for all characters.**
    - **Provides fields and shared methods: name, hero_class, race, damage, health, max_health; methods: hit, take_damage, heal, set_damage, is_alive, and abstract use_special.**

- **Wizard, Fighter, Cleric, Barbarian, Druid, Warlock (concrete subclasses)**
    - **Each inherits from Hero and implements use_special differently:**
        - **Wizard: Fireball (damage)**
        - **Fighter: Power Strike (damage)**
        - **Cleric: Divine Blessing (self-heal)**
        - **Barbarian: Rage (high damage)**
        - **Druid: Transform/Restore (heal)**
        - **Warlock: Eldritch Blast (damage)**
- **HeroFactory**
    - **Simple Factory pattern that creates heroes by class name string (e.g., "Wizard", "Fighter").**
- **(Module-level) functions / data**
    - **race_bonus — mapping of race names to damage/health bonuses.**
    - **battle(hero1, hero2) — the turn-based combat loop that runs until one participant falls.**

Relationships and interactions:

- **Inheritance: Wizard, Fighter, Cleric, Barbarian, Druid, Warlock are children of Hero.**
- **Collaboration: battle uses Hero interface methods (hit, use_special, is_alive) to run combat; HeroFactory constructs concrete subclasses.**
- **Responsibility summary:**
    - **Hero: shared state and common behavior, enforces abstract contract use_special.**
    - **Subclasses: provide class-specific specials and initial stat values.**
    - **HeroFactory: encapsulates creation to centralize and simplify instance construction.**
    - **battle: controls game flow and turn logic.**

(UML-like summary)

- **Hero (abstract)**
    - **^ inherits to**
        - **Wizard**
        - **Fighter**
        - **Cleric**
        - **Barbarian**
        - **Druid**
        - **Warlock**

- **HeroFactory → creates instances of the concrete classes**
- **battle(hero1, hero2) → orchestrates interactions via Hero's public interface**

3. Application of OOP Principles

The project applies all four main principles of OOP:

Encapsulation

- Health and damage are managed through methods rather than many direct writes. The methods take_damage(amount) and heal(amount) centralize state changes and enforce invariants (no negative HP, heal capped by max_health).
- set_damage is the controlled way to change damage.
- Benefit: prevents inconsistent states and reduces duplication.

```
def take_damage(self, amount: int):
    self.health = max(0, self.health - amount)

def heal(self, amount: int):
    self.health = min(self.health + amount, self.max_health)

def hit(self, target: 'Hero'):
    target.take_damage(self.damage)
    print(...)
```

Inheritance

- Concrete hero classes inherit shared logic from the Hero base class (constructor logic, hit/heal/take_damage and common fields).
- Reuse: subclasses reuse the hit and take_damage logic and only implement the unique use_special.

Polymorphism

- The use_special abstract method is implemented differently by each subclass. The battle loop calls attacker.use_special(defender) without needing to know which subclass is running — behavior varies at runtime.
- This allows the same battle function to support all hero types.

```
if random.random() < 0.7:
    attacker.hit(defender)
else:
    attacker.use_special(defender)
```

Abstraction

- Hero is an abstract base class (ABC) with an @abstractmethod use_special. It defines the contract for "what a character can do" (hit/heal/use_special/is_alive) while hiding implementation details of each special.
- This enforces that no generic Hero can be instantiated without providing a use_special implementation

```
class Hero(ABC):
    ...
    @abstractmethod
    def use_special(self, target: 'Hero'):
        pass
```

4. Use of Design Patterns

Factory Pattern

- **HeroFactory centralizes creation of hero objects from string identifiers.**
- **Justification: the factory decouples client code (main, tests, battle setup) from concrete constructors. Adding a new hero requires adding a subclass and a single change to the factory (or a later refactor to a registry), which keeps creation logic separate from usage.**

```
class HeroFactory:
    @staticmethod
    def create_hero(name, hero_class, race):
        if hero_class == "Wizard":
            return Wizard(name, race)
        ...
```

Notes / future pattern ideas:

- **Strategy pattern could be used to swap attack behavior dynamically (melee vs spell).**
- **Observer/Event Logger could decouple printing/logging from game logic for better testability and UIs.**

5. Explanation of SOLID Principles

Single Responsibility Principle (SRP)

- **Classes have focused responsibilities: Hero handles character state and shared behaviors; each subclass implements its special action only. HeroFactory only constructs heroes; battle only orchestrates turns.**
- **This separation keeps each unit easy to reason about and test.**

Open/Closed Principle (OCP)

- **The system is open for extension (you can add new hero subclasses and specials) but closed for modification in many areas — battle does not need to change to support new hero types.**
- **One small caveat: the factory currently uses an if/elif chain; converting it to a registry would increase OCP compliance by avoiding modifications to factory code for new classes.**

(Other SOLID notes)

- **Dependency Injection could be used to inject RNG or loggers for easier testing (improvement opportunity).**
- **SRP trade-offs include keeping print statements inside domain methods (acceptable for demo, but should be decoupled for production).**

6. Testing Strategy

Framework:

- pytest is used for unit testing.

What was tested:

- Damage and heal behavior: take_damage, heal, bounding behavior (no negative HP, capped heal).
- Special abilities: ensure use_special methods apply the expected effect (damage or heal).
- Victory condition and deterministic battles: monkeypatching random.random to force deterministic branch and verify that one participant ends as alive and the other dead after the battle.

Example test cases:

- test_take_damage_and_heal: verify health after damage and after heal, and that health clamps to zero on overkill.
- test_victory_condition_deterministic: patch randomness to predictable value and confirm a battle outcome.
- test_specials_effects: confirm special damage/heal values and behavior.

Test benefits:

- Ensures core mechanics behave correctly across changes.
- Provides reproducible checks for grading/demo.

7. Key Design Decisions and Challenges

Key challenges:

- **Balancing simplicity vs extensibility: the project is intentionally small for coursework, so some production-level decoupling (e.g., logging, registries) was simplified.**
- **Randomness in battle: randomness introduces non-determinism that makes testing harder; we used monkeypatching in tests to make runs deterministic when required.**

Design trade-offs:

- **Kept prints inside methods for straightforward demo output; this simplifies the demo but mixes concerns (logic vs presentation). For a larger project, an EventLogger (Observer) or using Python logging would be preferable.**
- **Factory implemented as if/elif — simple and clear. A registry-based factory would scale better but adds a bit more code.**
- **Centralized take_damage/heal enforces invariants, at the cost of extra method indirection compared to direct attribute access — the trade-off favors reliability.**

8. Conclusion

This project successfully implements a compact RPG battle simulator demonstrating the four OOP pillars and a design pattern, and includes automated tests to verify core behavior. Strengths:

- Clear separation of concerns: shared logic in Hero, class-specific specials in subclasses.
- Test coverage for essential mechanics and deterministic checks for the battle outcome.
- Easy-to-run demo and tests for quick verification.

Improvements given more time:

- Replace prints with an Event Logger or logging module (Observer pattern).
- Introduce Strategy pattern for attack behaviors to enable runtime behavior swapping and finer control.
- Convert factory to a registry pattern to follow OCP more strictly.
- Add more unit tests (edge cases, error handling) and integrate a simple CI pipeline.

I learned to apply abstract base classes, centralize state changes for safety, and write small, focused tests that make development and demonstration reliable.