Implementation of Inheritance and Encapsulation in OOP-based RPG Game Fighting System

Case Study: Character Class-Based Battle System

May 8, 2025

Jap Robertus K.S
NRP: 5803024004

Evan William
NRP: 5803024001

Contents

1 Introduction

In this report, we present a case study on the combat system in a character class-based RPG (Role-Playing Game). This type of game is quite popular because it allows players to choose characters with unique play styles, such as *Warrior*, *Mage*, *Archer*, and *Assassin*. Each of these characters has specific attributes and abilities, but they all still refer to one basic structure, which is the class `Karakter`.

The reason we chose this scenario is because the concepts of inheritance and encapsulation in object-oriented programming (OOP) are very suitable in the game world. For example, when creating a character system, we don't need to rewrite attributes such as `HP`, `Mana`, or `Attack` for each class. Simply create one base class, then other characters can inherit that structure and customize it as needed.

In addition to inheritance, the concept of encapsulation is also very important here. By hiding attributes (such as `HP` and `Mana`) and accessing them through getter and setter functions, we can control these values so that they remain valid throughout the game. So, there are no stories of characters suddenly having negative HP or being able to attack indefinitely.

Overall, we would like to point out that by utilizing PBO principles such as inheritance and encapsulation, the creation of a combat system becomes much neater, flexible, and ready for further development. This is an approach that is not only efficient, but also realistic for real game scenarios..

RPG games are also proven to be effective as a medium for learning OOP concepts. Research by Wong et al. showed that a game-based approach such as Odyssey of Phoenix can significantly improve students' understanding of OOP concepts. The game teaches principles such as encapsulation and inheritance through narrative flow and interesting challenges, while maintaining the player's continuous learning motivation [1].

To show how these principles are applied in real life, in the next section we will discuss a Python code snippet that implements a simple inheritance and encapsulation-based RPG character system.

2 Code and Output

The following Python code snippet shows the application of the concepts of inheritance and encapsulation.

Base Class: Karakter

```
class Karakter:
    def _init_(self, nama):
        self.__nama = nama
        self.__hp = 100
        self.__hp_max = 100
        self.__mana = 50
        self.__mana_max = 50
        self.__attack = 10
        self.__defense = 5

    def get_nama(self): return self.__nama
    def get_hp(self): return self.__hp
    def get_hp_max(self): return self.__hp_max
    def get_mana(self): return self.__mana
    def get_mana_max(self): return self.__mana_max
    def get_attack(self): return self.__attack
    def get_defense(self): return self.__defense

    def set_hp(self, nilai):
        self._hp = max(0, min(nilai, self._hp_max))

    def set_mana(self, nilai):
        self._mana = max(0, min(nilai, self._mana_max))

    def set_attack(self, nilai):
        self.__attack = max(0, nilai)

    def set_defense(self, nilai):
        self.__defense = max(0, nilai)

    def serang(self, target):
        damage = max(1, self.__attack - target.get_defense() // 2)
        target.set_hp(target.get_hp() - damage)
        return f"{self.__nama} menyerang {target.get_nama()} dan memberikan {damage} dam
```

Warrior Class

```
class Warrior(Karakter):
```

```
    def _init_(self, nama):
        super()._init_(nama)
        self.set_hp(self.get_hp_max() + 50)
        self.set_defense(self.get_defense() + 10)
        self.__rage = 0

    def get_rage(self): return self.__rage

    def set_rage(self, nilai):
        self.__rage = max(0, min(nilai, 100))

    def serang(self, target):
        self.set_rage(self.get_rage() + 10)
        damage = max(1, self.get_attack() - target.get_defense() // 2)
        target.set_hp(target.get_hp() - damage)
        return f"{self.get_nama()} menyerang {target.get_nama()} dengan pedang dan membe
```

Mage Class

```
class Mage(Karakter):
    def _init_(self, nama):
        super()._init_(nama)
        self.set_mana(self.get_mana_max() + 70)
        self.set_attack(self.get_attack() + 5)
        self.__elemen = "Api"
        self.__spells = {
            "Api": {"damage": 1.8, "mana_cost": 30, "effect": "membakar"},
            "Air": {"damage": 1.5, "mana_cost": 25, "effect": "membekukan"},
            "Tanah": {"damage": 1.6, "mana_cost": 28, "effect": "menghancurkan"},
            "Angin": {"damage": 1.4, "mana_cost": 20, "effect": "menyayat"}
        }

    def gunakan_skill(self, target):
        spell = self._spells[self._elemen]
        if self.get_mana() >= spell["mana_cost"]:
            self.set_mana(self.get_mana() - spell["mana_cost"])
            damage = int(self.get_attack() * spell["damage"])
            target.set_hp(target.get_hp() - damage)
            return f"{self.get_nama()} meluncurkan {self.__elemen} BALL dan {spell['effe
        else:
            return f"{self.get_nama()} tidak memiliki cukup Mana!"
```

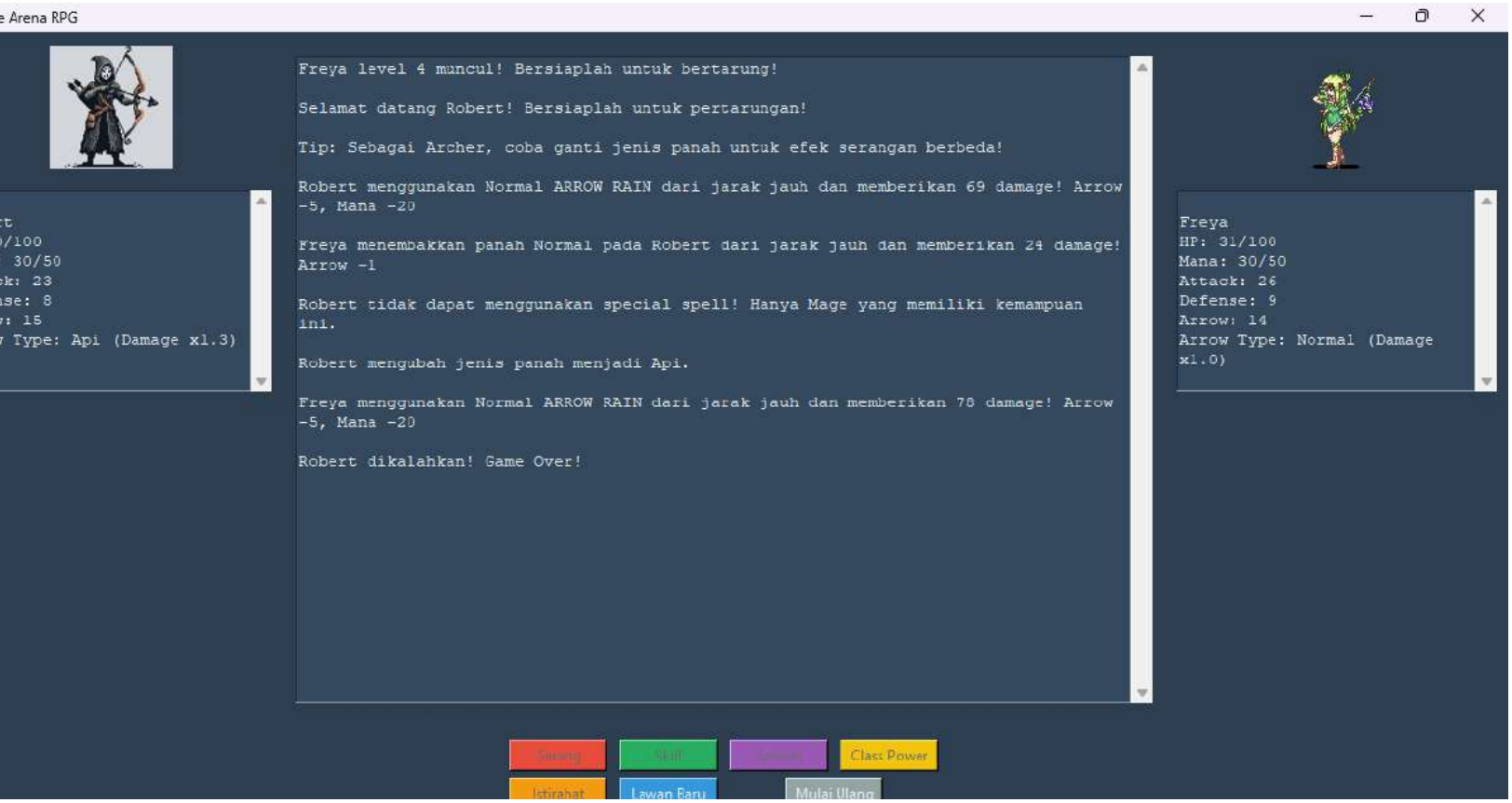

Figure 1: Interaction Output between Characters in RPG Games

3 Diagram

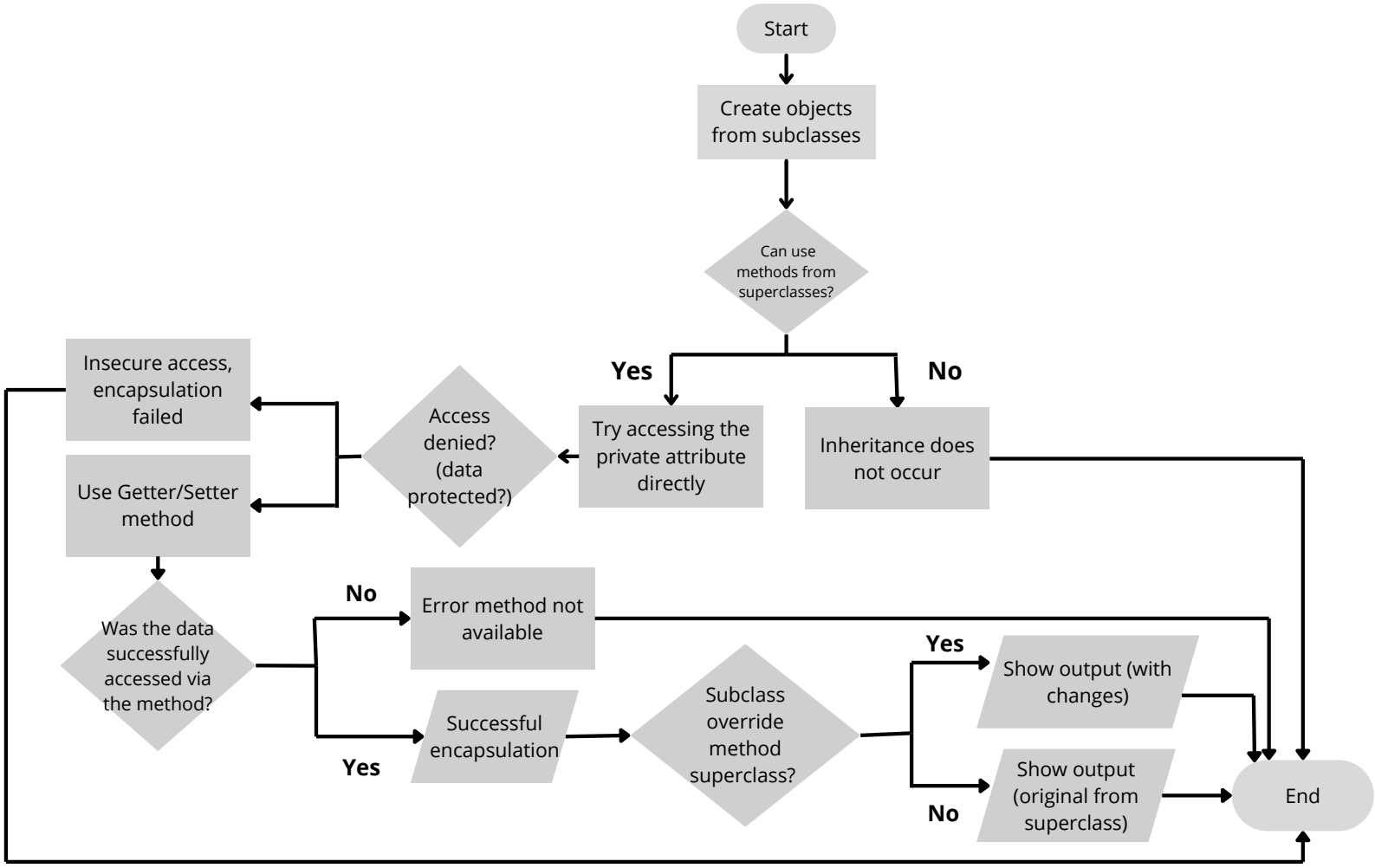

Figure 2: Inheritance and Encapsulation Diagram

4 Analysis

Inheritance

In the program we created, classes such as `Warrior`, `Mage`, `Archer`, and `Assassin` is an instance of the class `Karakter`. This means that all classes automatically have basic attributes and functions such as `get_hp()`, `serang()`, and others. This makes the code shorter and more consistent, as we only need to write the basic logic once in the class. `Karakter`. After that, each class can add specific features-for example, `Mage` can use elemental spells, while `Assassin` Besides inheritance, another important principle in OOP is encapsulation. After understanding how the classes in this game inherit attributes and methods from each other, we will now discuss how these attributes are secured and managed properly using encapsulation techniques.

Encapsulation

Important attributes such as `HP`, `Mana`, and `Defense` we make as private variables (using two underscores). To access or change those values, the character must use a function like `get_hp()` or `set_mana()`. By doing this, we can ensure that values stay within their intended boundaries—for example, making sure HP doesn't exceed its maximum limit and Mana never drops below zero. This safeguards the integrity of the system and helps prevent strange bugs. Beyond just controlling access, encapsulation also plays a vital role in maintaining long-term development flexibility. As Snyder points out, good encapsulation allows internal implementations to be changed without affecting derived classes, as long as the external interface remains the same.

However, many programming languages weaken this principle by allowing direct inheritance of private attributes. Encapsulation not only ensures runtime stability, but also significantly impacts software maintenance. Joyce notes that when systems are designed with encapsulated modules based on clear separation principles, they become more resistant to errors during updates or adaptive maintenance. While initial experiments showed little difference during early maintenance phases, systems built with strong encapsulation guidelines tend to be less prone to "corruption" when modified later on.

This highlights how a well-encapsulated design supports long-term code sustainability. Now that we've explored how encapsulation ensures consistency and flexibility, let's look at the real-world benefits of applying these principles in the development of an RPG game.

In-Game Benefits

- Avoid bugs such as characters having minus HP or unlimited attacks because attributes are controlled with validated setters.

- When you want to add a new class in the future (for example: *Paladin* or *Necromancer*), we just create a new class that inherits from `Karakter` without having to start from scratch.
- Code is easier to read, modify, and manage because each section has its own responsibilities.

5 Reference

1. Y. S. Wong, M. H. M. Yatim, and T. W. Hoe, "Learning Object-Oriented Programming Paradigm via Game-Based Learning Game – Pilot Study," *International Journal of Multimedia & Its Applications (IJMA)*, vol. 10, no. 6, pp. 181–195, Dec. 2018. DOI: 10.5121/ijma.2018.10615.
2. A. Snyder, "Encapsulation and Inheritance in Object-Oriented Programming Languages," in *Proceedings of the OOPSLA '86*, pp. 38–45, Sep. 1986. DOI: 10.1145/28697.28702.
3. D. Joyce, "An identification and investigation of software design guidelines for using encapsulation units," *Journal of Software and Systems*, vol. 7, no. 4, pp. 315–325, 1987. DOI: 10.1016/0164-1212(87)90028-8.