Final Project:
Combat Game

By: Álvaro Naval, Javier Suárez-Bárcena, Guillermo Sabaté, Fernando Alonso & Alejando Montero

1. Introduction

We have designed a turn-based terminal combat game in Java and structured the project using several design patterns to keep the code clean, organized and easy to expand. These patterns helped us separate different functions in the game.

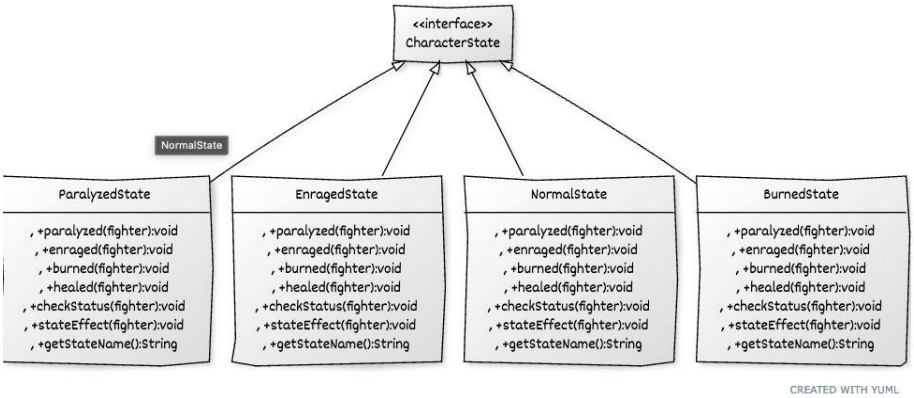
For example, we used the State pattern to manage status effects like being paralyzed. Instead of checking for flags around the code, each state has its own class that defines how the character behaves depending on the state it has.

We also used the Decorator Pattern to handle attacks. Rather than stuffing all the attack logic into the weapon classes, we wrapped weapons in attack objects like Heavy swing or fireball.

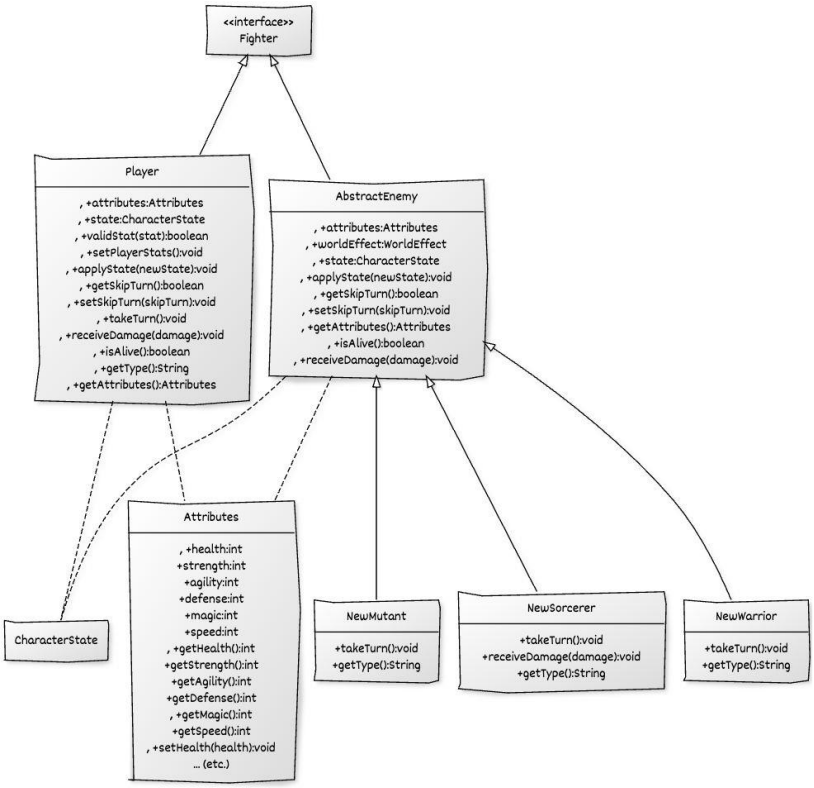
Using these patterns, it helped us develop the code in a more organized way so that it was easier to update along the way and gave us a solid structure to build on as the game grows.

2. Design Patterns & UML Diagrams

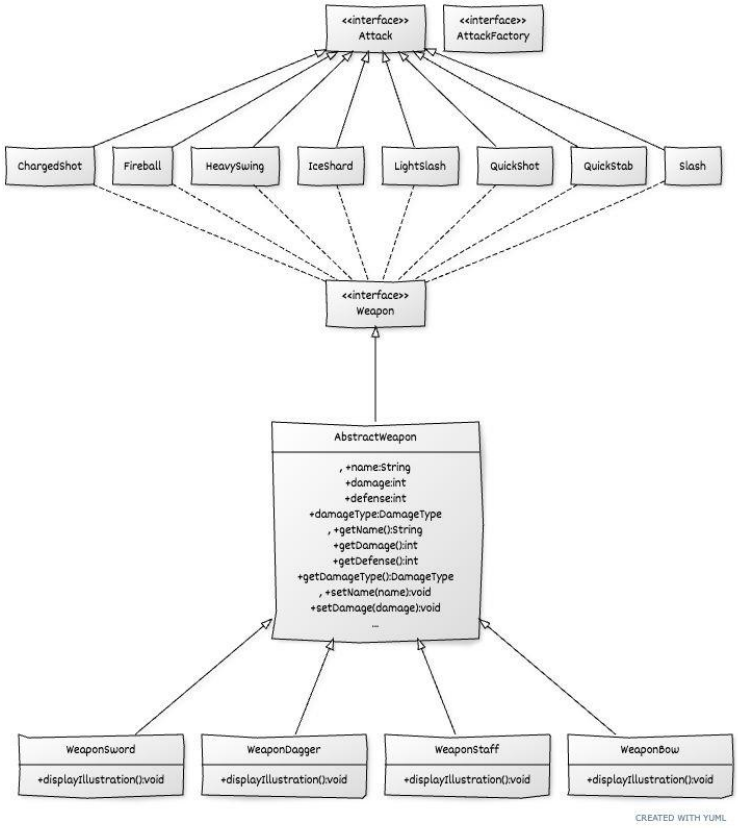
2.1 Character State Interface UML Diagram



2.2 Characters UML Diagram



2.3 Weapon & Attack type UML Diagram



3. Design Decisions

3.1 Objects Design Patterns (Decorator, Strategy & Factory / Abstract Factory Patterns)

- Decorator Pattern: Used to create special attacks for the existing weapons without changing the weapon itself. (Example: Heavyswing)
  The Decorator pattern wraps the weapon class inside another class to change its behavior

- Strategy Pattern: Used through the attack interface and its implementations, allowing weapons to perform several attacks by using different attack strategies. The attack interface has a common function for all attack types “void executeattack();”, Each attack class implements this interface designing its own attack.

- Factory Pattern: Used to generate different attack behaviors for each weapon. Rather than having to instantiate attacks like new(Heavyswing)

3.2 Characters Design Patterns (State & Factory Patterns)

- State Pattern: Used to manage the states of characters (Paralyzed, enraged, normal...) and to define how they behave depending on their current state.
  We have a CharacterState interface which defines the common methods that states need to implement, we then have the concrete classes like “Normalstate, paralyzedstate, etc” which individually define different behavior.

- Factory Pattern: Used for the Player, NewMutant, NewSorcerer and NewWarrior method and constructors.
  These characters need a couple of attacks based on the weapon they are holding, instead of hardcoding the attack in every class we used the factory pattern that knows which attacks to return depending on the weapon used.

3.3 Score Design Patterns (Singleton & Observer Patterns)

- Singleton Pattern: Used to ensure that only one instance of the score class exists throughout the game.
  Required to track the global score across players, enemies and game events.

- Observer Pattern: Listens for events from the game, specifically when an enemy is defeated. When this happens, the score object reacts and increases the code.