Battle Arena – A Java OOP Project

# Index

* 1. Introduction
* 2. Libraries and Frameworks
* 3. Bottom-Up Design Overview
* 4. Class Interactions (with Diagram Summary)
* 5. OOP Concepts in Practice
* 5.1 Encapsulation
* 5.2 Information Hiding
* 5.3 Polymorphism
* 5.4 Composition
* 5.5 Reuse
* 5.6 Abstraction
* 5.7 Inheritance
* 5.8 Subtyping
* 5.9 Exception Handling
* 5.10 Strategy Pattern
* 5.11 Factory Pattern
* 6. Additional Complexity Features
* 6.1 Multiple players
* 6.2 AI-Assisted Gameplay
* 6.3 Environmental Obstacles
* 6.4 Dynamic Bonuses
* 6.5 Level Progression System
* 6.6 Exception Handling and Game Logic Enforcement
* 7. Conclusion

# 1. Introduction

Battle Arena is a turn-based combat simulation game developed in Java. The game models a system where heroes automatically battle AI-driven enemies across multiple levels with varying challenges like traps, obstacles, and dynamic bonuses. The project showcases the complete application of object-oriented programming (OOP) principles through carefully structured, modular code.

# 2. Libraries and Frameworks

The game is developed in Java 17 using IntelliJ IDEA. Maven is used for dependency and project management. Only Java standard libraries are used, including java.util for collections and randomness. No third-party frameworks were required.

# 3. Bottom-Up Design Overview

The system starts with classes such as Hero, Enemy, Weapon, Bonus, and Obstacle. Interfaces define common behavior (IBonus, IEnemy), and abstract classes such as Enemy provide shared logic. These classes are assembled into a Level, and levels are managed by LevelManager. The game runs without player input—heroes and enemies interact based on coded logic.

# 4. Class Interactions

Entities interact through method calls, interface references, and modular factories. Each Hero has a Weapon (composition), and Level instances contain lists of Enemies, Bonuses, and Obstacles. Enemies are assigned behaviors using the Strategy Pattern (AttackStrategy), allowing flexible combat logic. BonusSelector dynamically chooses bonuses based on hero state, and ObstacleFactory adjusts trap and wall difficulty. All gameplay is managed sequentially by LevelManager.

*(Diagram: Hero → Weapon, Hero → IBonus, Enemy → AttackStrategy, Level → Enemy/Bonus/Obstacle, Enemy ← Zombie/Ogre, Enemy ← Strategy)*

* Flow Diagram

What it shows:

**Main** calls factories → builds all level content

**LevelManager** controls game flow

**Level** handles everything in order:  
Trap → Wall → Bonus → Enemy → Hero → End

A diagram of a diagram

AI-generated content may be incorrect.

* UML Class Diagram

What it shows:

Inheritance: Zombie and Ogre → Enemy

Strategy use: Enemy→AttackStrategy→Aggressive/AreaDamageStrategy

Bonuses: Hero uses IBonus → HealthBoost, DamageBoost

Composition: Level has all game components

LevelManager manages Level

A diagram of a diagram

AI-generated content may be incorrect.

# 5.1 Encapsulation

**Definition**: Encapsulation is the practice of bundling data with methods that operate on that data, while restricting direct access to internal variables.

**In Battle Arena**: All fields in core classes are declared private and are accessed/modified only via public getters and setters. This ensures data safety and separation of internal representation.

A screen shot of a computer program

AI-generated content may be incorrect.

# 5.2 Information Hiding

**Definition**: Information hiding prevents external classes from relying on the internal workings of a class.

**In Battle Arena**: Heroes apply bonuses without knowing their logic, and obstacles are triggered through interfaces.

A screen shot of a computer

AI-generated content may be incorrect.

# 5.3 Polymorphism

Polymorphism is shown through overriding methods like specialAttack(), and by processing objects through interfaces like List<IBonus> or List<IEnemy>. The correct version of apply() or attack() is resolved at runtime.

A screenshot of a computer program

AI-generated content may be incorrect.

A computer screen shot of a program code

AI-generated content may be incorrect.

Additionally, polymorphism is achieved using a Strategy Pattern that allows enemies to determine their behavior dynamically. Each Enemy has a reference to an AttackStrategy interface, which is implemented by classes such as AggressiveStrategy, DefensiveStrategy, and AreaDamageStrategy. These enable polymorphic execution of different attack() logics without modifying the enemy class itself.

A close up of a black background

AI-generated content may be incorrect.

A black background with white text

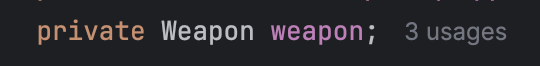
AI-generated content may be incorrect.

A screen shot of a computer

AI-generated content may be incorrect.

# 5.4 Composition

Hero has a Weapon (object in a field), and Level has lists of Enemies, Bonuses, and Obstacles. These demonstrate has-a relationships central to composition.

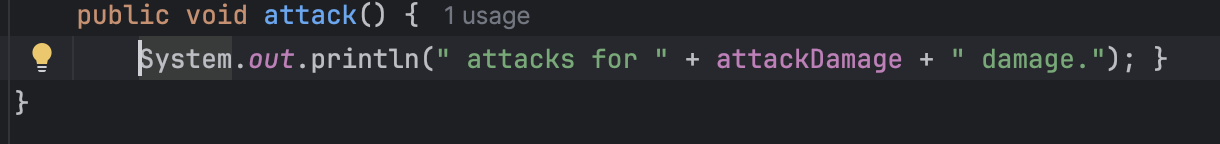


A screen shot of a computer

AI-generated content may be incorrect.

# 5.5 Reuse

Shared behaviors such as attack() and setHealthPointsRemaining() are defined in Enemy and reused in Zombie and Ogre. Similarly, bonus logic is reused through IBonus interface implementations.



# 5.6 Abstraction

**Definition**: Abstraction hides the implementation details while exposing a clean API.

**In Battle Arena**:

* Interfaces like IBonus, IObstacle and abstract class Enemy define contracts for implementation.

A screen shot of a computer

AI-generated content may be incorrect.



# 5.7 Inheritance

**Definition**: Inheritance allows one class to acquire properties and methods from another.

**In Battle Arena**:

* Enemy is inherited by Zombie and Ogre
* specialAttack() is overridden in each subtype

A black background with white text

AI-generated content may be incorrect.

# 5.8 Subtyping

Code uses interfaces as types:  
List<IBonus> bonuses = List.of(new HealthBoost(), new DamageBoost());  
Subtypes are interchangeable through interface references.

A screenshot of a computer program

AI-generated content may be incorrect.

# 5.9 Exception Handling

**Definition**: Exception handling allows programs to catch and deal with errors gracefully. CharacterDeadException is thrown when a hero's HP drops to 0. This exception is caught in LevelManager to stop the game and display an appropriate message.

A screen shot of a computer

AI-generated content may be incorrect.

# 5.10 Strategy Pattern

The Strategy Pattern is used to enable flexible and interchangeable enemy behavior. Each enemy can be assigned a strategy implementing AttackStrategy, such as:

* AggressiveStrategy: targets the strongest hero
* DefensiveStrategy: targets the weakest
* AreaDamageStrategy: targets all heroes

A screen shot of a computer code

AI-generated content may be incorrect.

A screen shot of a computer

AI-generated content may be incorrect.

A screen shot of a computer program

AI-generated content may be incorrect.

# 5.11 Factory Pattern

The Factory Pattern is used extensively throughout the project to encapsulate the creation logic of enemies, heroes, bonuses, and obstacles. This design improves modularity, supports the Open/Closed Principle, and isolates complexity from the Main class.

Multiple factory classes were implemented, including:

* EnemyFactory: generates enemies based on difficulty, assigning strategies dynamically.
* BonusFactory: selects bonus types and even includes randomized penalties in HARD mode.
* ObstacleFactory: creates traps and walls with scalable impact.
* TitanHeroFactory: provides the correct team composition (Titan or dual heroes) according to game difficulty.

A screen shot of a computer program

AI-generated content may be incorrect.

A screen shot of a computer program

AI-generated content may be incorrect.

A screen shot of a computer program

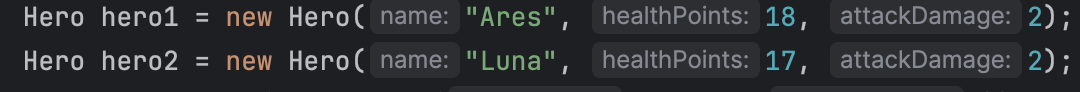
AI-generated content may be incorrect.

A screen shot of a computer program

AI-generated content may be incorrect.

# 6.1 Multiple Players

The system supports multiple heroes, each with independent state and behavior. They participate in a shared battle loop and are managed using List<Hero>. While there is no manual input, this simulates local multiplayer logic where all heroes act in turn.





A computer screen shot of a program code

AI-generated content may be incorrect.

A computer screen shot of white text

AI-generated content may be incorrect.

# 6.2 AI-Assisted Gameplay

All combat actions are automatically performed through predefined AI logic:

* Heroes auto-attack enemies
* BonusSelector uses the hero’s condition to determine bonus type
* Critical hits and special attacks are triggered probabilistically
* Enemies use assigned combat strategies through the Strategy Pattern:
* AggressiveStrategy: attacks the strongest hero
* DefensiveStrategy: attacks the weakest hero
* AreaDamageStrategy: attacks all living heroes simultaneously

A screen shot of a computer code

AI-generated content may be incorrect.

# 6.3 Environmental Obstacles

Each level includes environmental elements that influence gameplay difficulty through damage or penalty. Obstacles are dynamically generated using ObstacleFactory based on the current difficulty level.

There are two main types of obstacles:

* Trap

A screen shot of a computer code

AI-generated content may be incorrect.

* Wall

A computer screen with text

AI-generated content may be incorrect.

The strength of each obstacle is determined by the difficulty setting:

* In **EASY**: mild effects ( -1 HP, block 1 ATK)
* In **HARD**: severe effects ( -4 HP, block 3 ATK)

A screenshot of a computer

AI-generated content may be incorrect.

# 6.4 Dynamic Bonuses

Heroes receive both:

* Predefined bonuses (+1 HP, +1 ATK)
* Smart bonuses, chosen dynamically based on their current health
* BonusSelector.java:

A computer screen shot of a program

AI-generated content may be incorrect.

* Level.java:

A computer screen shot of a program

AI-generated content may be incorrect.

# 6.5 Level Progression System

Levels are modularly created using the Level class and managed by LevelManager. Each level may have its own configuration of:

* Enemies
* Obstacles
* Bonuses
* Difficulty level (EASY, MEDIUM, HARD)

The hero team composition dynamically changes based on difficulty:

* In EASY and MEDIUM: two heroes (Ares and Luna)
* In HARD: a single powerful hero (Titan)
* Main.java



A screen shot of a computer program

AI-generated content may be incorrect.

* LevelManager.java

A screen shot of a computer program

AI-generated content may be incorrect.

# 6.6 Exception Handling and Game Logic Enforcement

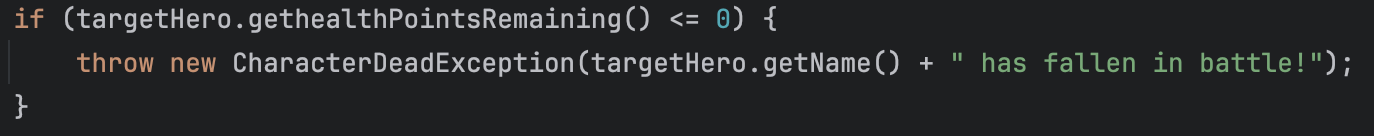
The game defines a custom CharacterDeadException, which is thrown when a hero dies. This ensures:

* Immediate termination of the level
* Clean and safe game-over state
* Clear separation of gameplay and control logic
* CharacterDeadException.java:

A screen shot of a computer

AI-generated content may be incorrect.

* Thrown in Level.java:



* Caught in LevelManager.java:

A screen shot of a computer program

AI-generated content may be incorrect.

# 7. Conclusion

Battle Arena is a complete implementation of object-oriented programming principles. It demonstrates the power of encapsulation, abstraction, and polymorphism in building flexible, extensible systems.Difficulty scaling and dynamic team formation via TitanHeroFactory support gameplay variety and clean code separation.