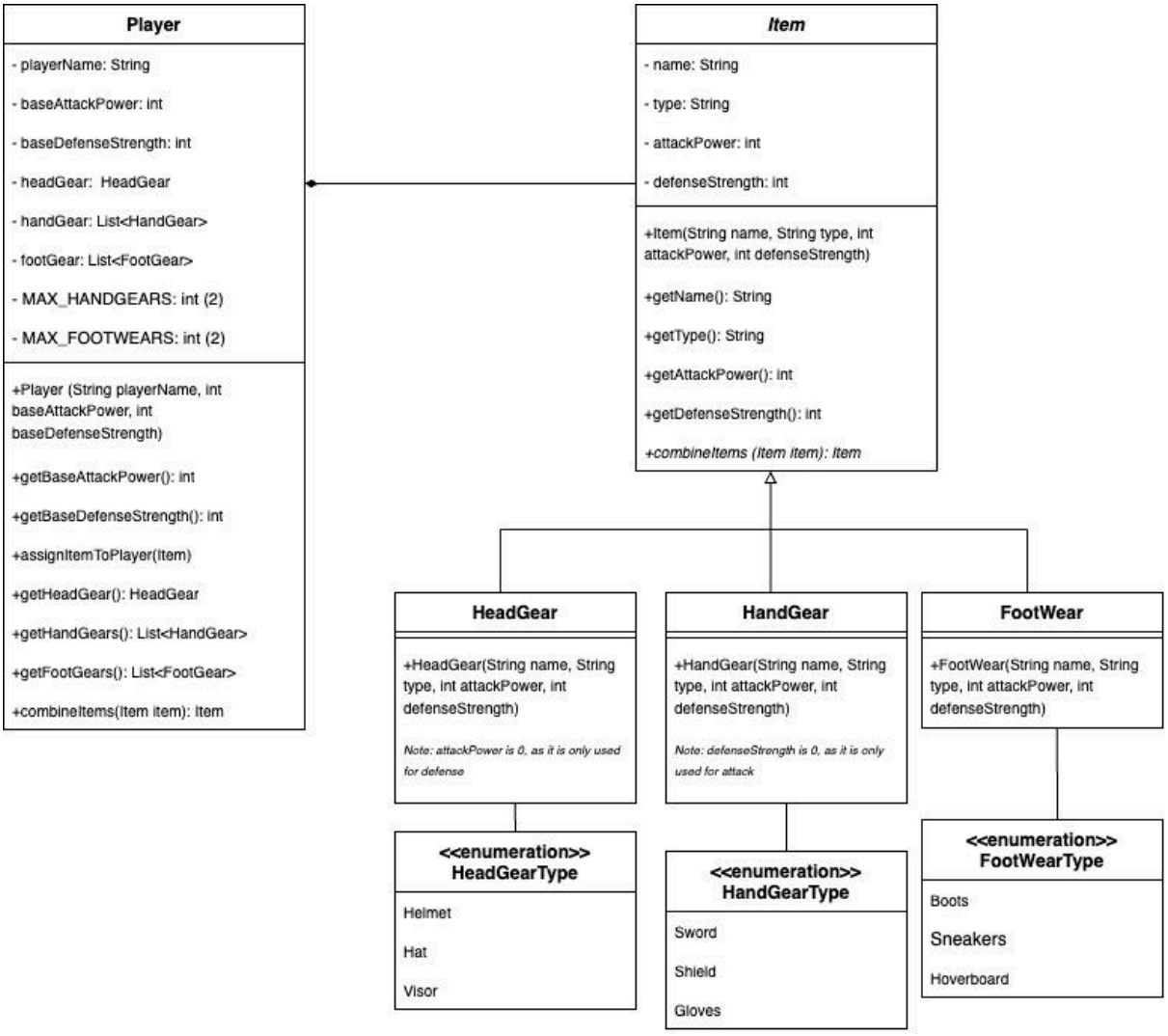
PDP Lab 2

UML


Player
- playerName: String
- baseAttackPower: int
- baseDefenseStrength: int
- headGear: HeadGear
- handGear: List<HandGear>
- footGear: List<FootGear>
- MAX_HANDGEARS: int (2)
- MAX_FOOTWEARS: int (2)
+Player (String playerName, int baseAttackPower, int baseDefenseStrength)
+getBaseAttackPower(): int
+getBaseDefenseStrength(): int
+assignItemToPlayer(Item)
+getHeadGear(): HeadGear
+getHandGears(): List<HandGear>
+getFootGears(): List<FootGear>
+combineItems(Item item): Item
Item
- name: String
- type: String
- attackPower: int
- defenseStrength: int
+Item(String name, String type, int attackPower, int defenseStrength)
+getName(): String
+getType(): String
+getAttackPower(): int
+getDefenseStrength(): int
+combineItems (Item item): Item
HeadGear
+HeadGear(String name, String type, int attackPower, int defenseStrength)
Note: attackPower is 0, as it is only used for defense
HandGear
+HandGear(String name, String type, int attackPower, int defenseStrength)
Note: defenseStrength is 0, as it is only used for attack
FootWear
+FootWear(String name, String type, int attackPower, int defenseStrength)
<<enumeration>>
HeadGearType
Helmet
Hat
Visor
<<enumeration>>
HandGearType
Sword
Shield
Gloves
<<enumeration>>
FootWearType
Boots
Sneakers
Hoverboard

Encapsulation of Different Item Types

a. Base Class: Item

- The Item class serves as a parent class for all item types (HeadGear, HandGear, and FootWear).
- It encapsulates common attributes like name, type, attackPower, and defenseStrength.
- This design allows for polymorphism, ensuring a uniform structure for handling different items while allowing them to have distinct behaviors.

b. Derived Classes: HeadGear, HandGear, and FootWear

- HeadGear (e.g., helmets, hats) is strictly for defense (attackPower = 0).
- HandGear (e.g., swords, shields) is strictly for attack (defenseStrength = 0).
- FootWear (e.g., boots, sneakers) can contribute to both attack and defense.
- This ensures strong encapsulation by restricting how each item type interacts with the player's attributes.

c. Player Class & Inventory Management

- One slot for HeadGear
- Two slots for HandGear
- Two slots for FootWear

If the player picks up an item of a type where the limit is already reached, the system automatically combines it with an existing item.

Representation of Item Attributes

Each item has the following attributes:

a. Name (String): Items follow an adjective-noun format, which plays a role when combining items.
b. Type (String): Defines whether the item is HeadGear, HandGear, or FootWear.
c. Attack Power (int): Represents the offensive capability of the item.
d. Defense Strength (int): Represents the defensive capability of the item.

Test Plan

1. **Assigning Items to Players**

a. Test: assignPlayerHeadGearTest

- Assigns HeadGear to two players and verifies the correct assignment.
- Ensures the Player class can correctly assign HeadGear and retrieve it.

b. Test: assignPlayerHandGearsTest

- Assigns two HandGear items to a player and verifies the assignment.
- Ensures that players can hold multiple HandGear items within the allowed limit.

c. Test: assignPlayerFootWearsTest

- Assigns two FootWear items to a player and verifies the assignment.
- Ensures that FootWear assignment functions correctly and respects inventory limits.

2. Item Combination and Restriction Tests

a. Test: testCombineDifferentItemTypes

- Attempts to combine items of different types (e.g., HeadGear with HandGear).
- Ensures the game prevents invalid combinations, throwing an IllegalArgumentException.

b. Test: testEquipHeadGear

- Assigns a HeadGear to a player, then tries to assign another one.
- The second assignment should fail with an IllegalStateException.
- Ensures that a player cannot equip more than one HeadGear at a time.

c. Test: testEquipHandGearBeyondLimit

- Assigns two HandGear items (valid limit), then attempts a third.
- The first and third items should combine instead of a third being added.
- Ensures correct handling of inventory limits for HandGear.

d. Test: testEquipFootWearBeyondLimit

- Assigns two FootWear items (valid limit), then attempts a third.
- The first and third items should combine.
- Ensures correct handling of FootWear inventory limit.

3. Battle Initialization and Setup

a. Test: testBattleInitialization

- Verifies that the Battle object is correctly initialized with two players and a list of gears.
- Ensures the Battle class properly initializes the game environment.

b. Test: testBattleInitializationWithNullPlayers

- Attempts to create a battle with a null player.
- Should throw an IllegalArgumentException.
- Ensures validation checks prevent invalid game states.

c. Test: testBattleInitializationWithNullGearList

- Attempts to create a battle with a null gear list.
- Should throw an IllegalArgumentException.
- Prevents an invalid battle setup.

d. Test: testAddGear

- Adds a new gear to the battle and verifies it exists.
- Ensures that new gears can be added dynamically.

e. Test: testAddDuplicateGear

- Attempts to add a duplicate gear to the battle.
- Should throw an IllegalArgumentException.
- Prevents duplicate gear assignments.

4. Dressing Players with the Best Gear

a. Test: testDressPlayers

- Assigns the best available gear to both players.
- Ensures the auto-equipping feature works correctly.

b. Test: testFindBestGearForPlayer

- Finds and assigns the best available gear to players.
- Ensures the logic for selecting optimal gear is functioning correctly.

5. Simulating Battle Outcomes

a. Test: testBattleOutcome_Player1Wins

- Equips Player 1 with better gear and runs the battle.
- Player 1 should win.
- Ensures the game correctly determines a winner based on gear stats.

b. Test: testBattleOutcome_Player2Wins

- Gives Player 2 stronger gear and runs the battle.
- Player 2 should win.
- Validates that stronger gear leads to victory.

c. Test: testBattleOutcome_Draw

- Equips both players with equal stats and runs the battle.
- The game should result in a tie.
- Ensures the battle logic handles edge cases where both players have equal stats.

6. Character State Verification

a. Test: testPrintCharacterState

- Prints player states after equipping gear.
- Ensures that equipped items are correctly assigned and displayed.