Homework 6: Adding Players, Monsters and Items

Description:

In the previous assignment, you used an object-oriented approach to implement rooms and a dungeon that a player could explore. Now it's time to complete this game application by adding players, monsters, and items. Players should be able to battle monsters, collect treasures and win the game!

Required Concepts:

Object-Oriented Design
UML Class Diagrams
UML Class Relationship Diagrams
Inheritance (is-a)
Polymorphism
Abstract Classes

Task:

This assignment will build upon your code from the previous assignment. You should already have a program that models the dungeon exploration aspect of the game. However, the game still requires a combat system, inventory management system, and a win condition. To accomplish this you must use an inheritance hierarchy to add players, monsters, and items into the game. First, you should read all of these instructions, then UML diagram the classes and their relationships, and finally implement the project.

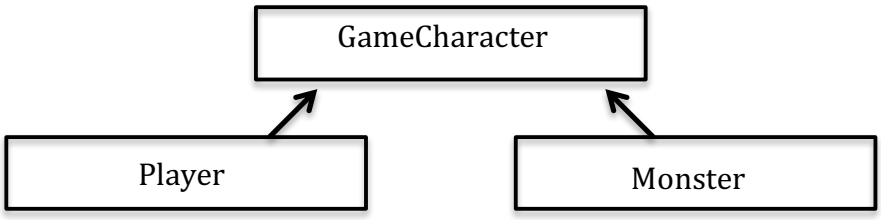
Modeling Players and Monsters using an Inheritance hierarchy:

The first step to adding `Player` and `Monster` classes to your game is to model players and monsters as objects. The first question you need to answer is: What information do I need to store to model a `Player` and a `Monster`? For a `Player`, the information includes: `name`, `health`, `attackPower`, and `mana`. For a `Monster`: `name`, `health`, `attackPower`, `xP` (the amount of experience the `Player` gets for defeating that `Monster`). This information will be the instance variables for the classes.

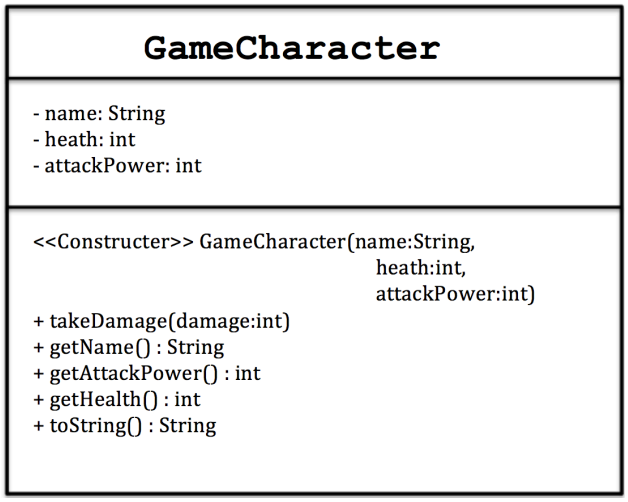
The next question is: what should a `Player` and a `Monsters` be able to do? From the software design standpoint, we are looking to decide what methods should be in the classes. Since these classes need to be designed to be able to take part in combat, we will need methods to facilitate that. For a `Player`, we'll need an `attack` method that will allow a `Player` to `attack` a `Monster`, a `castSpell` method to allow a `Player` to cast a spell at a `Monster`, a `chargeMana` method to allow the player to increment their `mana`, a `takeTurn` method to allow the user to choose what to do and then do it, and a `takeDamage` method to allow a `Player` to take damage when attacked. You may find that as writing the class you will need additional methods such as helper methods, getter methods, or setter methods, but this list above is a good starting point. For a `Monster`, the list will be similar. The `Monster` will have all the same methods except for the

`castSpell` and `chargeMana` methods. For `takeTurn`, the `Monster` can simply always choose to `attack`.

We can see right away that the `Player` and `Monster` classes are very similar. When we have cases like this, we can use Inheritance to our advantage. Rather than having duplicate code in multiple classes, we pull the commonalities between the classes out into a Superclass. For out purposes, we will create a superclass `GameCharacter` and have the `Player` and `Monster` classes inherit from it. Below is illustrated the inheritance hierarchy and UML diagrams for these classes.



GameCharacter

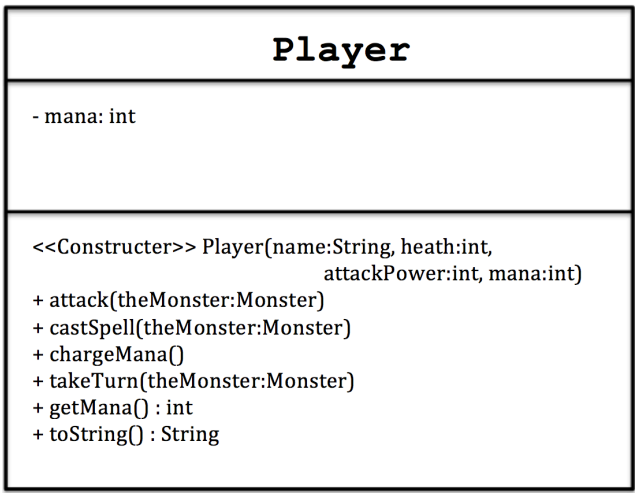


UML class diagram for GameCharacter

GameCharacter Class

The `GameCharacter` class is the superclass of the `Player` and `Monster` hierarchy. It has the common variables and functionality of its subclasses. It has the ability to `takeDamage`, get methods to get the values of it's private instance variables, and a `toString` method to return a String that states the `name`, `health`, and `attackPower` of this `GameCharacter`.

Player

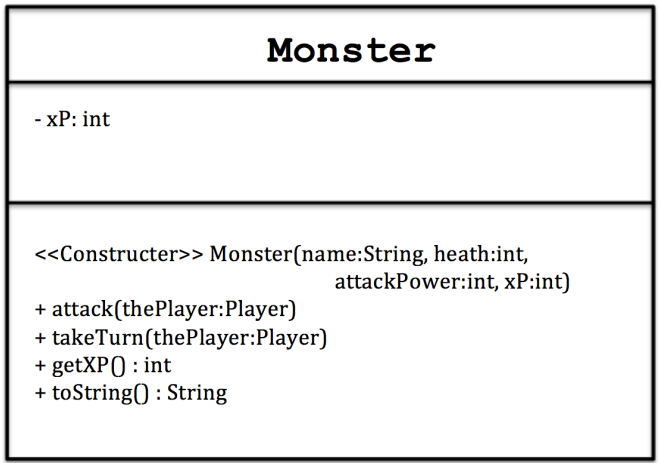


UML class diagram for Player

Player Class

The `Player` class represents a human player. It has the ability to `attack` and to `castSpell` on a `Monster`. It also has the ability to `chargeMana` and a `takeTurn` method that allows the user to choose what to do and then takes that action. Lastly, it has a get method to get the value of the instance variable mana and a `toString` method that returns a string with the `name`, `health`, `attackPower`, and `mana` of this `Player`.

Monster



UML class diagram for Monster

Monster Class

The Monster class represents a monster. It has the ability to `attack` Players. It also has a `takeTurn` method that for a simple `Monster` will just call the `attack` method. Lastly, it has a get method to get the value of the instance variable xP and a `toString` method that returns a string with the `name`, `health`, `attackPower`, and `xP` of this `Monster`.

Items:

You will also need to add Items to your game. Rooms, Players, and Monsters can have Items. When a `Player` enters a `Room` with one or more `Item`(s) in it, then the `Player` can equip the Items, that is, remove them from the `Room` and add them to the `Player`. A `Monster` can also have Items. When a `Player` defeats a `Monster`, if the `Monster` has any Items, the `Player` can take the `Item`(s) from the `Monster` and add them to the `Player`'s inventory. You should have as a minimum 2 types of Items, `Weapon` and `Potion`. A `Weapon` modifies how much damage a `Player` does when attacking. Potions can be used to increase health or mana. The `Player` should be able to use potions during combat.

Game Application

As in the previous assignment, the `Player` should be able to explore the `Dungeon`. But now, your `Room`(s) can have items. They should also have a random chance of spawning a `Monster` that the player needs to defeat. If a `Monster` is spawned, the `Player` and `Monster` enter a combat loop until one of them is defeated or the `Player` runs away. The `Monster` should have a random chance of holding 0 or more items for loot. These items can be picked from a random list of items that you instantiate before the game starts, or you can randomly generate items if you want (perhaps have a `RandomItemGenerator` class).

As the `Player` explores the `Dungeon`, the `Player` can accumulate and use items found in any `Room`(s) or from `Monster`'s loot and can grow in level as they defeat Monsters. For the Win Game Condition, you can have the user defeat "all the Monsters" in the dungeons (that is, a set number of monsters that you decide), defeat a "Boss Monster" after growing in level, or make it "through the dungeon" or to a particular `Room` in the `Dungeon` (Make sure you give instructions when the game starts for how to defeat the game.). In the last case, in order to pass through a particular room, the player must defeat any monsters in it. Otherwise, when they run away, the `Player` must go back to the `Room` they had just come from.

Bonus:

Implement unique attack powers for the different types of monsters. To accomplish this, you may want to model `Monster` as an abstract class. Each subclass that extends the `Monster` class (i.e. Goblin, Dragon, Vampire) can then define its intended, specific behavior for the abstract method: `takeTurn(…)`.

Possible 'Monster Attack' behaviors:

So far, we have only defined monster attacks as something that reduces the player's health. However, monsters might have powers that affect the hero in other ways. Consider that a hero has several different properties that a monster can target: Health, Magic, Attack Power, and Level. Think of interesting ways that you can hinder the player's stats using monster attacks. Below are just a few examples that you might consider:

- Hit player for random amount of damage with uniform distribution
- Hit player for random amount of damage with normal/gaussian distribution
- Hit player for static amount of damage
- Hit player and reduce their mana points
- Hit player and reduce their level/attack power
- Hit player for damage and monster heals itself for that same amount
- Hit player and swap life totals
- Hit player with an attack that may exponentially increase each round

- Hit player with an attack and steal their weapon
- Hit player with an attack and stun them, such that they lose their next attack action
- Hit player and grapple them so that they cannot run next round
- Hit player and cause them to expend mana and cast a spell on themselves

Submission instructions

Create the Class UML Diagrams and the Class Relationship UML Diagrams and include them along with your project source code. Save the UML diagrams as pdfs. Submit your source and UML diagrams through git in a hw6 directory. **The UML diagrams will count 25% toward this homework grade.**