COMP10062: Assignment 6

The Assignment

This assignment is about effectively using inheritance, association, overriding, and constructors. You're going to work alone or with a partner to translate a description of the inhabitants of a game world into a set of classes that could serve as part of a **model** for a game or gaming utility, draw the UML class diagram for these classes, and then implement them in Java.

Read the descriptions below carefully and try to take every detail into account in your design. There might be multiple ways to implement each attribute and ability. Just because your idea is different, doesn't mean it's wrong.

The Inhabitants of Foon

In town of Hogsface in the magical land of Foon there are three kinds of Monsters: Orcs, Goblins, and Manticores.

Attributes

All Monsters have a clan affiliation that cannot change. They all have numeric ratings for three basic attributes that describe their abilities: Ferocity, Defense, and Magic. These attributes can be raised or lowered by one point at a time, within the range 0 to 20. The clan affiliation and basic attributes of a Monster are obvious and can be easily retrieved by anyone.

All Monsters have 0 or more treasure. They all have a numeric health rating. If they have health > 0 they are alive, otherwise they are dead.

The clan affiliation of a Monster cannot change, unless that monster is a Manticore – they're fickle and change clans all the time.

Every Goblin has exactly one other Goblin that is its sworn enemy. This enemy is fixed at birth and never changes, even if it dies.

An Orc can be Infantry or Warlord. Warlords have an integer leadership rating from 1 to 5. This rating can be raised or lowered by 1 point at a time. Every warlord has exactly 5 Infantry in his or her command, and every Infantry member has exactly one Warlord that commands it. The assignments of Infantry to Warlord are frequently changed.

Abilities

Attack and Defense

When a Monster attacks, it produces a battle score equal to the average of its Ferocity, Defense, and Magic scores. If its opponent's attack score is higher, it loses health points equal to the difference in attack scores. (e.g. if a Manticore with an attack score of 18.5 fights a Goblin with an attack score of 10.7, the Goblin will lose 7.8 health.)

When a Manticore or a Warlord attacks, its battle score is greater than a normal Monster's score by a factor of 1.5.

Dead Monsters cannot attack or defend themselves.

Healing

When a Warlord sounds its battle cry, it produces a health boost score equal to its leadership rating multiplied by 5 and raises the health score of each of the Infantry in its command by that amount.

Dead Warlords cannot sound their battle cry and dead Infantry cannot be healed.

Treasure

A monster can gain or lose treasure as the result of a fight. When a warlord gains treasure, it gets a +1 leadership boost for every 10 points of treasure it receives.

A Dead monster cannot gain treasure.

The Design

Draw a UML class diagram for a set of classes to represent the Monsters of `Foon`. Make sure to represent all instance variables, parameters, return values, and relationships between classes. I recommend you use draw.io for UML diagrams. **Hand drawn UML diagrams will receive a penalty unless they are of excellent quality**. You should be using draw.io, other drawing tools tend to be a lot of work, and are not recommended.

Use inheritance to eliminate code duplication and overriding to avoid if statements (e.g. don't have an if statement to check if the monster is a Manticore before calculating its attack score).

Use encapsulation to restrict access to the instance variables as much as possible.

Include the minimum set of methods to implement the world as described above.

Include two constructors for each class: one that sets all instance variables, and one that sets just the clan affiliation and fills in default or random values for everything else.

Include a `toString()` method for every Monster type that prints its class name, its clan affiliation, whether it is alive or dead, and its ratings (ferocity, magic, etc.). Make sure the output is easy to read.

The Implementation

Implement the classes you designed in Java.

It would be a good idea to test each class by creating at least two instances (one for each constructor), calling each method, and printing the results of each method to standard output. But this is not required. For this assignment, you are developing the **model** but not the **view**.

Use automatic code generation as much as you can (use "Insert Code" to create the starter code for constructors, getters, setters and `toString()` methods).

Use the `super` and `this` keywords to eliminate code duplication wherever possible.

Check for errors in all methods (e.g. parameters out of range, null parameters, actions can't be taken when the Monster is dead, etc.). If an error happens, print a meaningful error message to standard output and return immediately from the method. Make sure you test the error cases.

Optional Extras

This is purely a design exercise, but maybe it could be turned into some sort of a game. For example, can you simulate turn-based combat between an Orc and a Goblin? Can you make it graphical? Is there anything else you can do to make the town of Hogsface to life?

Handing In

See the due date in Canvas. Hand in by attaching a zipped version of your **.java** (not .class) files and your class diagram to the Canvas Assignment.

Make sure you follow the **Documentation Standards** for the course.

Special Instructions for Pairs

If you work with a partner, follow these special instructions:

1. Both partners should work together on the design and the implementation. Use pair programming when writing the code.
2. Both partners must hand in a copy of the design and the implementation for this assignment. Late days for each partner will be tracked separately.
3. Each partner must also fill out the "statement of effort" and include that in the hand in.

Warning: If you do not include a statement of effort, it will be assumed that you worked alone, which means that your solution better not match anyone else's!

Evaluation

Your assignment will be evaluated for design (40%), implementation (40%), and documentation (20%) using the rubric in the drop box.