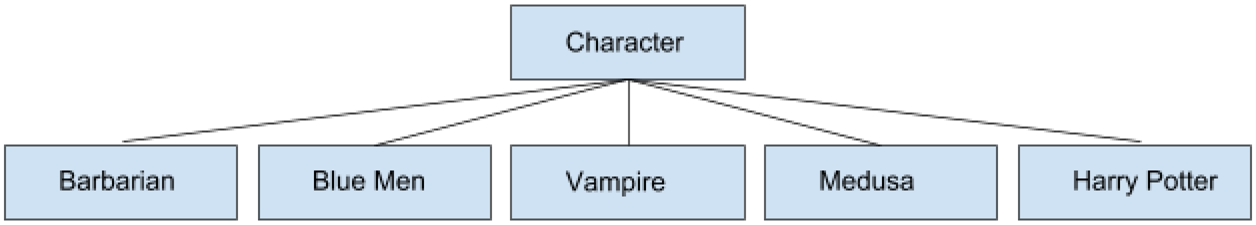
Kenneth Hall
CS 162 400 W2018
3-04-2018

Design + Reflection



Character abstract class provides the framework for all of its child classes, including variables for the number of faces and quantity of attack and defense dice, armor value, and strength points (hereafter SP). getSP() returns the SP of the Character, and is used to communicate with whatever function is handling the game if a Character is dead and the game should end. attack() and defend() have unique implementations in each child class, and are thus made virtual in Character. recover() is used to restore up to ⅓ of the Character's SP, or up to the Character's max SP, whichever is less.

All child classes make their own seed for random and do their own random number generation

***All child classes make a check in their defend() methods to see if the incoming attackVal is an instant kill (I call it a critical hit in implementation, and is indicated by the value 100, a value that is impossible for any character to roll naturally). I put a lot of thought into the requirement that "subclass should not have dependencies on which type of character the attacker is." I believe this implementation is both compliant with the requirement and allows Medusa to have a special attack that otherwise goes outside of the normal rules of combat; the instant kill value need not be specific to Medusa. The defender doesn't care if it was Medusa or another hypothetical Character that we implemented later with a similar ability that sends an attackVal of 100. The defender only cares that it received a "you're dead" command and then sets their SP to 0

Barbarian class:

is child of: Character

Barbarian has 2d6 on attack, 2d6 on defense, 0 armor, and 12 SP

The design of Barbarian's attack() and defense() methods are the foundation from which the corresponding methods in the child classes are derived:

attack():

rolls 2d6, reports the individual rolls, and then returns the sum of the rolls

defend(int attackVal):

rolls 2d6, checks to see if damage occurs, adjusts SP if attackVal is greater than the sum of defense rolls and armor, reports the individual rolls as well as attackVal. Checks if SP is less than or equal to 0, and reports that the Character died if that is true.

BlueMen class:

is child of: Character

BlueMen has 2d10 on attack, (starts with) 3d6 on defense, 3 armor, and 12SP

attack():

implementation is identical to Barbarian's attack() (save for type of die used)

defend():

for every 4 damage BlueMen takes, they lose one of their defense dice, and subsequent "rolls" for any missing dice are reported to the game as 0. Otherwise, implementation is identical to Barbarian's defense()

Vampire class:

is child of:

Character

Vampire has 1d12 on attack, 1d6 on defense, 1 armor, and 18 SP

attack():

implementation is identical to Barbarian's attack() (save for type and number of dice used)

defend():

special ability (Charm): a d20 is rolled, and if the result of the roll is greater than 10, Vampire completely ignores the incoming attack

if Charm does not activate, defend() otherwise behaves like Barbarian's defend() (save for number and type of dice used)

Medusa class:

is child of: Character

Medusa has 2d6 on attack, 1d6 on defense, 3 armor, and 8 SP

attack():

special ability (Glare): if Medusa rolls a 12, returns 100 (the value 100 triggers a "suffered a critical hit" condition in the defending Character

if Glare does not activate, attack() otherwise behaves like Barbarian's attack() (save for number and type of dice used)

defend():

Identical to Barbarian's defend() (save for number and type of dice used)

HarryPotter class:

is child of Character

HarryPotter has a member variable that none of the other classes do: extraLife. The first time HarryPotter has his SP fall to 0 or below, his extraLife is used and his SP is set to 20. HarryPotter has 2d6 on attack, 2d6 on defense, 0 armor, and 10 SP

attack():

identical to Barbarian's attack()

defend():

identical to Barbarian’s defend(), except in that the first time HarryPotter would die, he instead has his SP set to 20.

Queue class:

implements a struct QueueNode. QueueNode has members QueueNode* next, prev; Character* fighter; and string name. name represents the name of the fighter that the user provides.

The Queue class’s function is to hold QueueNodes. Queue is manipulated through its member functions addBack and removeFront. Information about the Queue is accessed through member functions isEmpty, getFront, getFrontName, and printQueue.

addBack(): adds a QueueNode to the back of the Queue
removeFront(): removes the QueueNode at the front of the Queue

isEmpty(): returns true if the Queue has no QueueNodes, otherwise returns false
getFront(): returns the Character* from the QueueNode at the front of the Queue
getFrontName(): returns the name string from the QueueNode at the front of the Queue
printQueue(): prints the contents of the Queue (prints each QueueNode’s name)

Main():

```
    three Queue*, one for losers, one for each of two lineups
    two ints to keep track of score
    three Character* to handle Character creation for each lineup and allow fighting to occur between two Characters
    two strings to allow users to name their Characters
    two ints to keep track of attacks to be passed between Characters
    Menu for Character selection
    Menu for “see losers” option
    Menu for “play/quit”
    int for menu choice to be stored

    While the user has indicated they do not want to quit
        set the number of fighters in each roster
        build the roster for each player
        make the rosters fight
            the top of each roster fights the other
            the loser is sent to the loser Queue, the winner is sent to the back of the roster Queue
            When one of the rosters is empty, display the results of the match
            Give the user the option to see the order of the defeated Characters in the loser Queue
        reset the roster and gameplay data
        allow the user to indicate whether they would like to play again or quit
```

Problems I encountered

I initially had issues with clearing out the space when the user indicated that they wanted to play again. I changed my initial design from building Queue objects at the start to implementing pointer to Queue, which could then invoke constructors/destructors as necessary on the object to which they are pointing

TEST TABLE

Condition Observed	**Expected Observation**	**Actual Observation**
Character loses	Character is moved to Loser Queue	Character is moved to Loser Queue
Character wins	Character is moved to back of own Queue	Character is moved to back of own Queue
Last Character for one or both teams dies	Score is displayed and winner announced; option to see Queue of dead Characters displayed	Score is displayed and winner announced; option to see Queue of dead Characters displayed
User plays again at end of a game	Team Queues and Loser Queue cleared to make room for new Characters	Team Queues and Loser Queue cleared to make room for new Characters