Shawn S Hillyer

CS 162-400

November 13, 2015

Assignment 4 – Containers / Fantasy Combat Tournament - Analysis, Design, and Reflections

Analysis and Design

First, after examining my client from Assignment 3, I decided to refactor my code out a bit. Specifically, I spent a little time developing a more robust/flexible Menu class. I did this because I figured failure to do so would result in a bunch of extra code and would be hard to make changes to dynamically/easily.

So I made a Menu class sometime earlier this week. It's an extension of my earlier menu designs. First, there are two ways to load it. I can pass a vector of strings into the menu, or a string consisting of a filename. Both methods load a vector of std::string into an items variable that allows me to encapsulate the number of items in the list easily. Then, I added methods to print the list with a corresponding number for each item and prompt the user with the menu and validate input is in the range allowed. (1 to vector.size()).

This will make any menu prompts much more flexible going forward. I did this ahead of knowing the details of this assignment. Now that I know them, I can use it to handle a variety of data collection issues.

Okay, on to the analysis. I'm going to use my Client class to call functions of the tournament class so I avoid overextending the Client class. So Client will just encapsulate the main interface, and Tournament will handle the specifics for running the tournament itself.

The tournament class is going to be responsible for several tasks. It will need to prompt for a number of creatures allowed in each player's lineup. It will need to prompt for a creature name and for a creature type. For creature name, I'll make a simple string function that accepts any name 3 characters or longer. For prompting for the creature type, I'll use my menu class and have the user enter an integer representing which type they want to use. Once that data is done, I'll create the relevant Creature and add it to a QueueList (from our earlier lab). This will require I extend Creature to have a name for each Creature; this will not be the same as the type_name, but the user supplied name.

I think a simple array of QueueList<Creature*> player_lineup[2] will let me use the index 0 for player 1, and index 1 for player 2, per my assignment 3 design already returning the index 0 or 1 for the creature that won in a lineup. The QueuList will be used because, of course, the players will enter the monsters in the order they will play, so they get pushed to the back and popped off the front with .remove().

In order to handle the recycling of monsters, the Tournament should have a .play_round() method that will first grab the first creature from each list, call the combat.resolve_combat() method to find the winner, and then handle the winner/loser. Processing the winner involves letting him revive some strength points, which I'll encapsulate as a method, revive_creature(). Then I will remove the creature and add it to the Queue again. The loser will be removed from its' players creatures and then popped onto the QueueStack of losers, let's just call it QueueStack<Creature*> losers. Finally, to track which

team the losers are on, I'll have a "parallel" stack QueuStack<int> loser_team that is called every time I access the loser stack and passed in the index of the loser's player (0 or 1).

Tournament should have a play_tournament() method that will just cycle through the various methods. First, get the # of creatures, then get the player 1 lineup (which gets the creature type and the name for each), add that to their queue, repeat for player 2, and then call a loop on play_round(). Play_round() will continue until one of the player_lineup Queues is empty. To check this, I'm going to add a .is_empty() method to my Queue and Stack's (for consistency). It will simply return true if the "head" (front for the Queue, top for the stack) points to Null.

During each play_round, I will have to update a simple array of int player_scores[2] with the score. To start with, I'll make a simple 1 point per battle scoring system then I can extend that method to calculate a more interesting score system after I've tested the simple method. At the time of this call, the winner of the round and the creature types that fought should be announced. I believe my combat class already does this type of reporting, although I will need to add a reference to the creatures names.

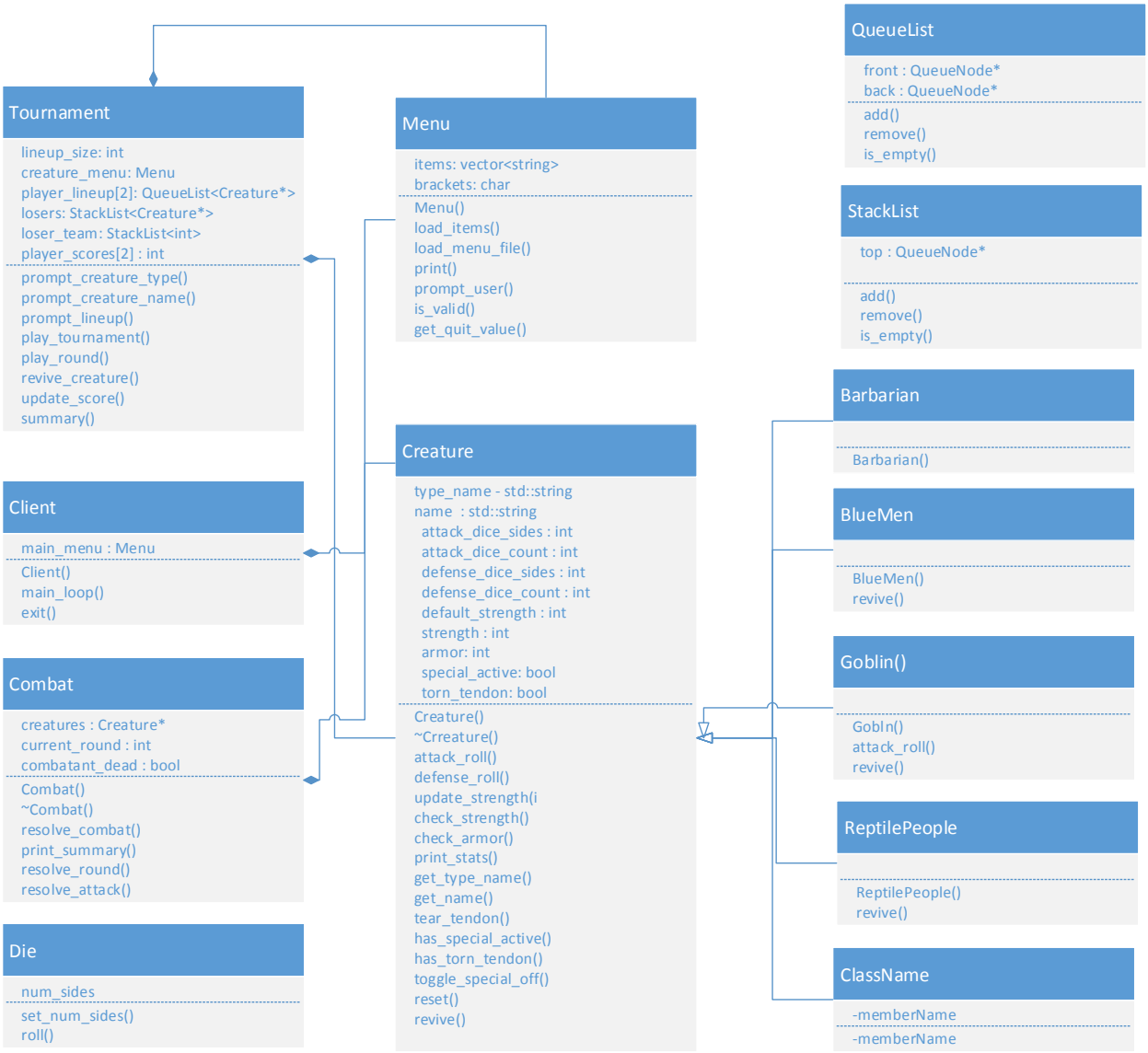
The scoring system itself will use a hierarchy. If a creature beats another in it's same tier or lower, then it's worth 1 point. If they beat somebody 1 tier higher, they get 3 points. And if they beat somebody in 2 tiers higher, they get 5. I'll make goblins the lowest tier, barbarians and shadow tier 2, and reptile and bluemen tier 3. This is based on their winrates in my assignment 3 combat logs. I might tweak this later, but this should be relatively interesting and might be more balanced than a simple point per win scheme. This should be broken out into its own function; I'll call it update_score() for now. I'll have to pass in the winner's index and the creature type (or a pointer to the creature) so I can figure out which tier each winner/loser was in to calculate.

Finally, for the final reporting, I'll break that into its own method. Once all tournament rounds are resolved, I will call tournament.summary() and manipulate the stacks and queues to report the relevant data. Specifically, it will need to print out the top three creatures. First place will be the creature that survived the most battles or just the creature(s) that are alive still. Tracking the score would mean adding a score member to the Creature class and incrementing that during each round. The other method would just call .remove() on the winning player's queue until I've removed 3 values or until the queue is empty. I'll remove them from the queue and then push them to a new local ranking queue. So the first one in line will be 1st, 2nd will be 2nd, third 3rd, and their teams. During this summary, we also need to report the player_scores and say which side scored more points. There will be an option for the user to see a list of the final order of all fighters, so after printing the top 3, there should be a prompt to "See all fighter rankings" and this will simply remove() the loser pile in order and pop it onto a queue again, but I'll probably pop on the top 3 as well for reference. Then just print it all.

To handle in-between round regeneration, I'm going to add a revive_winner () method to be called at the same time I add them to the end of their players Queue. This should be overridden in a few of the subclasses to demonstrate polymorphism. I'll come up with some unique ideas and add that to the subclasses. The default regen will be to restore 50% of their missing health. So if they have 50%, they will heal for 25%. If they have 25%, they will heal for ~37.5%, etc. This means I'll have to extend Creature to have a .revive() method. I might not need to factor out the revive_winner() method, but it might help to do so in case I change implementation later.

That should cover most of the high-level design. Below is the UML diagram I came up with before beginning any programming.

Class Hierarchy



TESTING PLAN

I know the creatures work, but I need to test the following:

1) Creatures
    a. Make sure I can set and retrieve the creature's names
    b. Make sure revive works as intended (add a console message when they are revived showing str before and after, can manually check the math works that way)
2) Test every method of my tournament class

a. Play tournament should call the relevant subroutines
    i. It should play rounds until one list or the other is empty (player_lineup)
    ii. It should call the summary() method to show the final results
b. Play_round() should remove one creature from each player_lineup, run a Combat.resolve_combat() between them, determine the winner and loser, add the loser to the loser pile, and revive_creature on the winner.
    i. Winner should also go back in line to fight again
c. Update_score should be tested
    i. Test that tier 0 vs tier 0, tier 1 vs tier 1, and tier 2 vs tier 2 get 1 pt each
    ii. tier 1 vs tier 0 gets 1 pt
    iii. tier 2 vs tier 0 gets 1 pt
    iv. tier 2 vs tier 1 gets 1 pt
    v. tier 0 vs tier 1 gets 3 pts
    vi. tier 0 vs tier 2 gets 5 pts
    vii. tier 1 vs tier 2 gets 3 pts
d. Verify winner correctly reported at end
e. Prompt_creature_type should print menu and only accept integers matching the creature types
f. Prompt_creature_name should accept strings from 3 to 20 characters
g. Prompt_lineup() should ask a player for a creature type and a creature name by calling the relevant prompts, then create a creature of the correct type.

Testing Results:

After mapping out all of my intended functions and implementing them as blank functions to make sure I had all of my includes and no syntax areas, I began implementing one function at a time, starting with the smallest pieces and working up to the larger chunks.

Because of this, I used the above test outline to incrementally test various components. I used a separate table to test the scoring system since it was kind of complex / had a lot of variables. I also had to visually inspect each creature's regeneration to see that it matched my design.

One tricky thing is that a Goblin normally can't beat a reptile or bluemen in my simulations… I gave them a crazy revive() method that gives them double the health if they survive a battle. I figured if I pit like 10 goblins against 9 goblins then a reptile I could see the goblin beat the reptile or bluemen types. I tried this several times (which is kind of labor intensive) and couldn't even get them to beat a shadow. The cells without green are rows in which I couldn't get that creature in that column to beat the creature in that row.

See the various tables that follow:

Expected Scores

	Winner				
Opponent	Goblin	Barbarian	TheShadow	ReptilePeople	BlueMen
Goblin	1	1	1	1	1
Barbarian	3	1	1	1	1
TheShadow	3	1	1	1	1
ReptilePeople	5	3	3	1	1
BlueMen	5	3	3	1	1

Class	**Method**	**Purpose**	**Result**	**Notes**
Creature	Creature(std::string)	Verify constructor accepts string and sets creatures name property	PASSED	I used a default value so that old code could still call constructor without breaking the function calls
Creature	get_name()	Verify accurately returns the name (and now the type of the creature)	PASSED	
Tournament	Tournament()	Verify constructor loads the creature select menu and that the scores are set to 0	Expect scores to be zero; first trial this was not the case	I had forgotten to initlaize the scores array and saw random values in the scores at first
Tournament	prompt_creature_type()	Verify prints menu for user to select a creature type and returns the integer representing the correct type	PASSED	Not much code here; leveraged my menu class to handle the major logic
Tournament	prompt_creature_name()	Verify user can enter a name from 3 to 20 characters only for aname and that it returns that string	PASSED	Simple

Tournament	prompt_lineup()	Veriy allows a player to pick creatures until they have filled up their lineup based on the lineup size specified by the user.	PASSED	Appeared to be working fine (and indeed was after more testing). Testing of the QueueList class for prior lab made this trivial (also, I had made mine a template)
Tournament	revive_creature()	Verify polymorphism working for the revive() creature method and that feedback printed	Passed	I used a driver to just pass in a "beat up" creature in a short program which I've discarded now
Tournament	prompt_lineup_size()	Validate that user can only input values greater than one. Also put a "reasonable" cap on the size. Can verify using prompt_lineup that only the correct # of creatures can be added	PASSED	No notes
Tournament	play_round()	Verify function goes through the logic of popping creatures off of the lists, evaluating the winner using Combat class, and pushing loser into appropriate place. Also should call the relevant subroutines (mostly all tested so far)	Passed	Summary() was not implemented or tested when I made this yet
Tournament	update_score()	More complex script, see separate table	PASSED	Many variables to test here
Tournament	summary()	Should print accurate score for player based on who won and lost each round, and accurately report the top 3 creatures and give option to print the rest of the creatures	PASSED	No major hurdles; the function is a little large but not so big I felt need to refactor it (though I would if it grew larger)

Tournament	play_tournament()	Once all piecs working independently, piece them together for total logic flow and veryify can go through the steps outlined in the assignment	PASSED	none