Shawn Hillyer

CS 162-400

10/29/2015

Assignment 3 – Analysis, Testing, and Reflections

Analysis

First, an analysis of what the classes might be and how to organize them. Obviously, a creature class will be superclass. Goblin, Barbarian, Reptile People, Blue Men, and The Shadow all inherit from creature.

Each of them has a certain configuration of strength points, armor, attack dice, and defense dice. Strength can be an integer. Armor can be an integer. Attack can be an array of Die from our earlier lab. Each class has 2 or 3 dice, so the array should be dynamically allocated during construction and deallocated during destruction. Likewise for defense, although each has 1 to 3 dice for defense. I'll want a private data member for each type of dice to indicate how many attack_dice and defense_dice a creature has; let's call them attack_dice_count and defense_dice_count.

The **Creature** will need to have an `attack_roll()` and `defense_roll()` method to get their rolls. Also we need to be able to `update_strength()` after each round of combat. We will need to `check_armor()` before applying damage. And we need to `check_strength()` to see if a creature is alive before proceeding after an attack is made.

To handle the combat, I will create a **Combat** class. The combat class will take two creature objects for its constructor and store them into an array of creatures. It will track the current combat `round`, then begin simulating combat rounds by calling `resolve_combat`(). `resolve_combat()` will in turn call a method, `resolve_round()` that executes a single turn. `resolve_combat`() will be responsible for tracking and incrementing the round, determining if a creature has died, and announcing the result of the combat.

Each `resolve_round()` call will iterate on our array of creatures, having each creature take a turn conducting an attack roll, with the defender conducting a defense roll. It will then subtract the defense roll from the attack roll (unmitigated_damage = creature->attack_roll() – creature->defense_roll()) to calculate the damage. The method will then call check_armor on the defender, and subtract it from the damage (mitigated_damage = unmitigated_damage – creature->check_armor();) It's likely or possibly I may factor out some of these smaller tasks as private methods depending upon complexity. Especially the attack itself.

Finally we will pass the mitigated_damage amount to update_strength(), which will subtract that value from the defending creatures strength. In any case, if the damage calculated at any point is 0 or less, we will be able to stop evaluating feature calls and just say that the attacker missed.

At the end of each attack (not every round, every *attack)*, it should be verified that both creatures are alive. So we can have a quick check_strength() call on each creature and end combat immediately. I'll refactor this out to a Combat method called check_alive(). We can then print a summary showing the winner using `print_summary()`.

Handling the Goblin, Barbarian, Reptile People, Blue Men, and The Shadow classes will basically have default constructors that pass the quantity and types (d6, d10, etc) of their attack and defense dice, their armor and

strength values, and a string storing their type_name all back to the creature constructor. This means we basically just define some variables inside each class to pass to the constructor, making most of the classes easy.

For Goblin, we'll need to figure out a way to handle the Achilles Tendon status. The simplest way is to just add bools, `special_active / torn_tendon` and check them with `has_special_active()` and `has_torn_tendon()`. Every creature attack_roll() method will need to simply check the has_torn_status value before returning the total of the dice roll, and return half if true. In order to apply this status, we will need to override the Goblin classes attack_roll() function to do the normal call from the base class, but THEN check if it rolled a 12. If it rolls a 12, then the goblin can update the has_special_active bool be true for himself.

The combat class should be responsible for checking every attacker's has_special_active during each resolve_round(); if an attacker has the status has_special_active right after his roll, we should call a method tear_tendon() on the attacked creature. So we need to add a mutator, tear_tendon() to the creature class, so that any creature can have a torn tendon status updated.

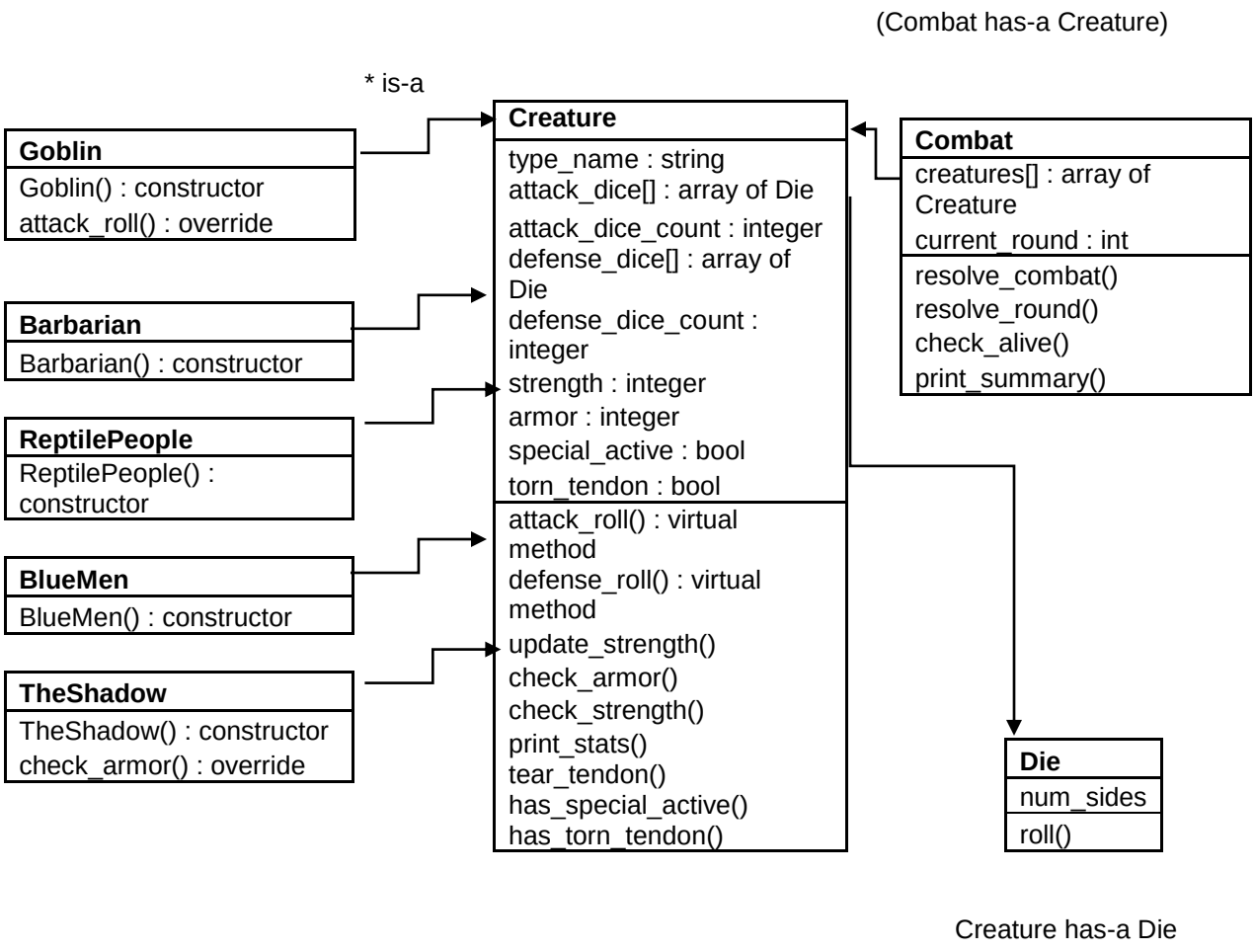
The reason I plan to encapsulate the has_special_active into the Creature base class is so that I can use polymorphism and avoid the slicing problem (I want to use a pointer-to-Creature for everything in Combat, so it won't have access to specific methods.)

Another solution if that doesn't work or is clunky would be to check the name of the attacker and, if it's a goblin, then cast the pointer and do some other stuff, but I think that would ultimately be more convoluted. Also, this might be extensible if we wanted to add new status effects in the future and more specials.

For The Shadow, we need to handle their special case. Based on my design already, the simplest thing to do would be to override the check_armor() function for this class. When the check_armor() call is made on the shadow, it should first randomly generate a value of 0 or 1. If a 0, we can just return a 0 armor status. Otherwise, we should return some arbitrarily high armor value (sufficient to make sure no attack roll can get through, but also so high that maybe new, more powerful creatures wouldn't break through. Maybe 1000 just to be absurdly overcareful.). What this means is that no class needs to know if the effect is active until this method is called, which is perfect and lightweight.

Diagram on next page.

So sketching out these methods and classes:



Testing Plan

For testing purposes. I will conduct incremental testing. I'm going to construct a Simulator object that will start a main_loop(). The simulator will prompt for input and then conduct various tests based on some pre-programmed values to avoid having to manually enter data. I'll dump the results of the combat rounds into a file for analysis. For any given simulation, I can run multiple iterations to get a higher statistical sample.

I think the best way to run a ton of simulations is to set up an array of attackers and an array of defenders. Each array will have pointers to each one of the attacker and defender types. I'd use a nested loop to iterate through all combinations. Alternatively, if I didn't want duplicate battles (Goblin vs. Barbarian should be the same results as Barbarian vs. Goblin), I could use the following pattern to develop my calls:

	Goblin	Barbarian	Reptile	BlueMen	Shadow
Goblin	0,0	0,1	0,2	0,3	0,4
Barbarian		1,1	1,2	1,3	1,4
Reptile			2,2	2,3	2,4
BlueMen				3,3	3,4
Shadow					4,4

After each Combat::resolve_combat() call, I'd need to re-initialize each creature to reset its status effects and health (strength). I think I'll do that by reallocating the memory or something. I'll have to play with it more to know for sure.

We should expect to see a 50% win rate over a large enough sample for any same-type battle (Goblin v Goblin, Barbarian v Barbarian, etc).

I will also set up a way to manually select a creature a and creature b so that I can simulate individual fights.

To check my results, I'm going to be visually inspecting various interactions as they go. I need to make sure that the specials work as expected, so I'll need to verify that the status effect is applied and stays on for the rest of that creatures combat. I'll have to inspect the attack values and check the math to see that it occurs.

Likewise, I'll need to verify the shadow properly avoids attacks. This should occur about 50% of the time.

Another thing I need to do is make sure that the attacker is random each round, otherwise the creature that attacks first each time in equal matches is more likely to win.

The next page shows some of the things I'll inspect. Note that I've already extensively tested the Die class so I'm taking it for granted (mostly) that this works.

The green cells are the cells that passed the test (in this case, all). I did it this way to save space.

If you run the main auto simulation or check the file dumps it created you'll see how I validated winrates, which intuitively looked right as well. You can test the goblin and shadow specials by checking the combat log using the custom match simulation (Option 2, I think).

	Goblins	Barbarians	Reptile	BlueMen	Shadow
Attk Rolls	2	2	3	2	2
Range	1-6	1-6	1-6	1-10	1-10
Attk Total	2-12	2-12	3-18	2-20	2-20
Def Rolls	1	2	1	3	1
Range	1-6	1-6	1-6	1-6	1-6
Def Total	1-6	2-12	1-6	3-18	1-6
Armor effect	-3 dmg taken	0	7	3	0
Special	If goblin special activiates, enemy should get half-value attacks until dead	NA	NA	NA	Should avoid completely 50% of attacks during armor check
Special	Roll of 12 applies torn_tendons to target	NA	NA	NA	NA
~50% win vs same kind	50%	50%	50%	50%	50%

Other

Str only goes down

Combat stops when a creature dies

Dead creature never makes an attack

Creatures randomly get to attack first

Correct winner announced

Reflections / Modifications

First, I had to add a method to toggle_special_off() so that the attacking Goblin didn't get to keep their special on indefinitely and I used it for the shadow, as well, so they could avoid an attack. In reflecting on this now, it wouldn't have really mattered if the goblin's special stayed on, because the tendon cut lasts for the rest of combat. However, for the shadow, this was helpful. I could have made his armor or defense roll just return a super absurd value instead of making the has_special_active variable. At this point I'm going to keep my design as is.

As I wrote code for handling the shadow's ability, I realized I could use the same pattern – toggle off the special, check if it activates, then handle it in the combat class if it did. This is what I ultimately ended up doing and I kept it this way.

I added a member to the Combat class, combatant_dead, to toggle on if a combatant died and short circuit future rounds. Basically this was my flag to each of the layers of loops to say, okay, fight's over. It worked very well.

Added a report_winner_index() to Combat to track winners over multiple battles. I did this to verify statistics much faster than I could myself. I ended up extending this idea to a combat_log to write logs to file showing the win-rates.

I decided to scrap storing an array of Die – it's overcomplicated. I changed to just storing the number of sides on each type of die as member variables. This greatly simplified several things and allowed the base class to just check how many dice and sides the dice has for whichever subclass was calling the .attack or .defend methods. This just meant I created a local Die during those calls which, even over tens of thousands of function calls, had no discernible overhead.

All of my subclasses remained basically unchanged from designs. The base class, Creature, needed a few getters I didn't realize I'd want (simply for labeling things for screen/file output, really). So I added the get_name() and check_strength() methods so I could print names and allow the combat class to know if a creature was dead or not. I also added a reset() method to allow me to reset the creatures back to a default status, which I implemented to greatly simplify testing the classes. It seemed useful if I would ever need to, for instance, reset a dead player because they have multiple lives, etc. so I kept it in.

The combat class I just added the resolve_attack method, which was really just refactoring out a chunk of code to make things more readable in resolve_round.

I went a bit beyond what I really needed to do in creating my combat_log, but I got the assignment done so quickly and I wanted to see what I could do. I ended up making a kind of bloated-looking Client class, but it works well. The various prompt calls and a separate report_statistics and log_combat method just helped refactor out some complexity to abstract away the details as I worked on larger problems.