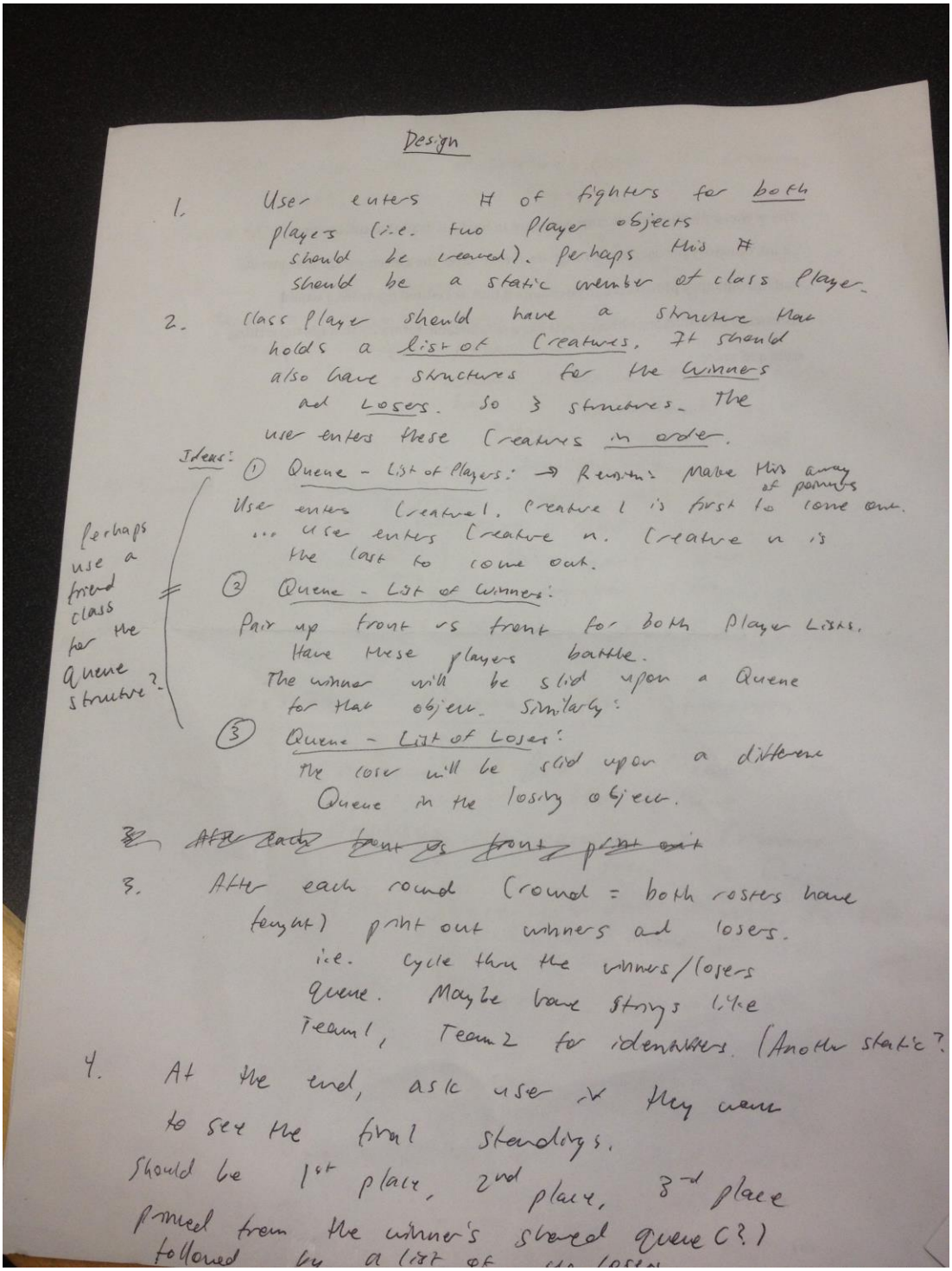
Alex Hoffer

Part 1) **Design:**



Design

1. User enters # of fighters for both players (i.e. two Player objects should be created). Perhaps this # should be a static member of class Player.
2. class Player should have a structure that holds a list of Creatures. It should also have structures for the winners and losers. So 3 structures. The user enters these Creatures in order.

Ideas:

① Queue - List of Players: → Reason: Make this away of passing

User enters Creature1. Creature 1 is first to come out. ... User enters Creature n. Creature n is the last to come out.

② Queue - List of Winners:

Pair up front vs front for both Player Lists. Have these players battle. The winner will be slid upon a Queue for that object. Similarly:

③ Queue - List of Losers:

The loser will be slid upon a different Queue in the losing object.

perhaps use a friend class for the Queue structure?

~~3. After each front vs front print out~~

3. After each round (round = both rosters have fought) print out winners and losers. i.e. cycle thru the winners/losers Queue. Maybe have strings like Team1, Team2 for identifiers. (Another static?

4. At the end, ask user if they want to see the final standings. Should be 1st place, 2nd place, 3rd place printed from the winner's shared queue (?) followed by a list of the losers

1. cont'd: enter type of creature, name of creature, maybe name of player?

3. cont'd: At the end of each round, assess damage of the winner.
Add new member function to parent class to do so.
Make it virtual, add a roll that re-generates some (Strength points
i.e. each creature type should have a specialized roll interval. Between 0 and something else.

3/4 cont'd: the winning team should have the highest kill/death ratio (i.e. $\frac{\text{kill}}{\text{death}}$) This means there should be a member that tracks this.

Possible Queue class Functions:

1) enqueue → Add a creature to the queue.
2) dequeue → remove a creature from the Queue.
3) delete Queue → for memory de-allocation, erase all info.
4) check empty → return true if first is NULL.

The class must have a Node class defined within it for the purpose of on-the-fly memory creation. Perhaps it should also have Player as a friend class so it can use Player functions if necessary.

Part II) **Reflection:**

The program is messier than I would like it to be, and I wish I would have put more energy into developing my design beforehand. The design completely neglects the need for a dynamic stack class (which I didn't know I needed until the implementation phase). The use of stack and queue objects in this program is a bit tenuous- there's simply too many instantiated objects of both classes and they are used inconsistently. If I were to start this program from scratch, I would put deeper thought into which structures would have worked best with the information I needed to be stored, maintained, and/or manipulated.

The general framework for the program is as follows. The italicized text are my comments about these steps.

1) Assemble a roster for a Player. *Very tedious. The user has to type in a lot of information and the program doesn't even work unless there's a good amount of competitors.*
2) Dequeue both Player's rosters, use the Creature attack function to make them battle. If the Creature wins, put it in that Player's winner's list. If it loses, put that Creature in that Player's list and push it onto the full, organized list of rankings. *Seems alright.*
3) Send both Player's winners queue to a winner's bracket. Dequeue each Creature from both and have them battle. If the Creature wins, enqueue it onto a queue called top three. If it loses, push it on the full list. *This won't work if there isn't enough Players.*
4) Sort the top three list for the Creatures with the highest kill counts. Push the remaining Creatures onto the full list. Return the sorted list. *I like this but I wish I didn't have to instantiate so many Stack, Queue objects.*
5) Print out top three, print out the remaining players, print out the team results.

Part III) **Testing:**

INPUT	**EXPECTED**	**OUTPUT**
Enter in a Creature that doesn't exist.	Re-calls function.	Re-calls function.
Putting in 5 Creatures for both Players will lead to the top three victors being printed out first, then the remaining Creatures, then the winning team.	All results should correspond to the reality of the computation.	Results consistent with expectations.
Enter 5 Blue Men for one team and 5 Hydra for another.	The Blue Men team dominates the Hydra team.	Results consistent.
A Creature gets all kills and doesn't die once.	Ratio system is broken, will print inf.	Handled this error. The ratio simply becomes the Creature's number of kills.
A battle ends and the results of the battle are printed.	The winner is pointed out and the loser is pointed out.	Results consistent.
A Creature who wins a round regenerates some health.	The Creature rolls a die with the same number of sides as their attack function, the result is tacked onto their strength points.	Results consistent with expectations.