DESIGN & REFLECTION DOCUMENT

SANDRO AGUILAR
PROJECT 2 - SECTION 400

February 02, 2019

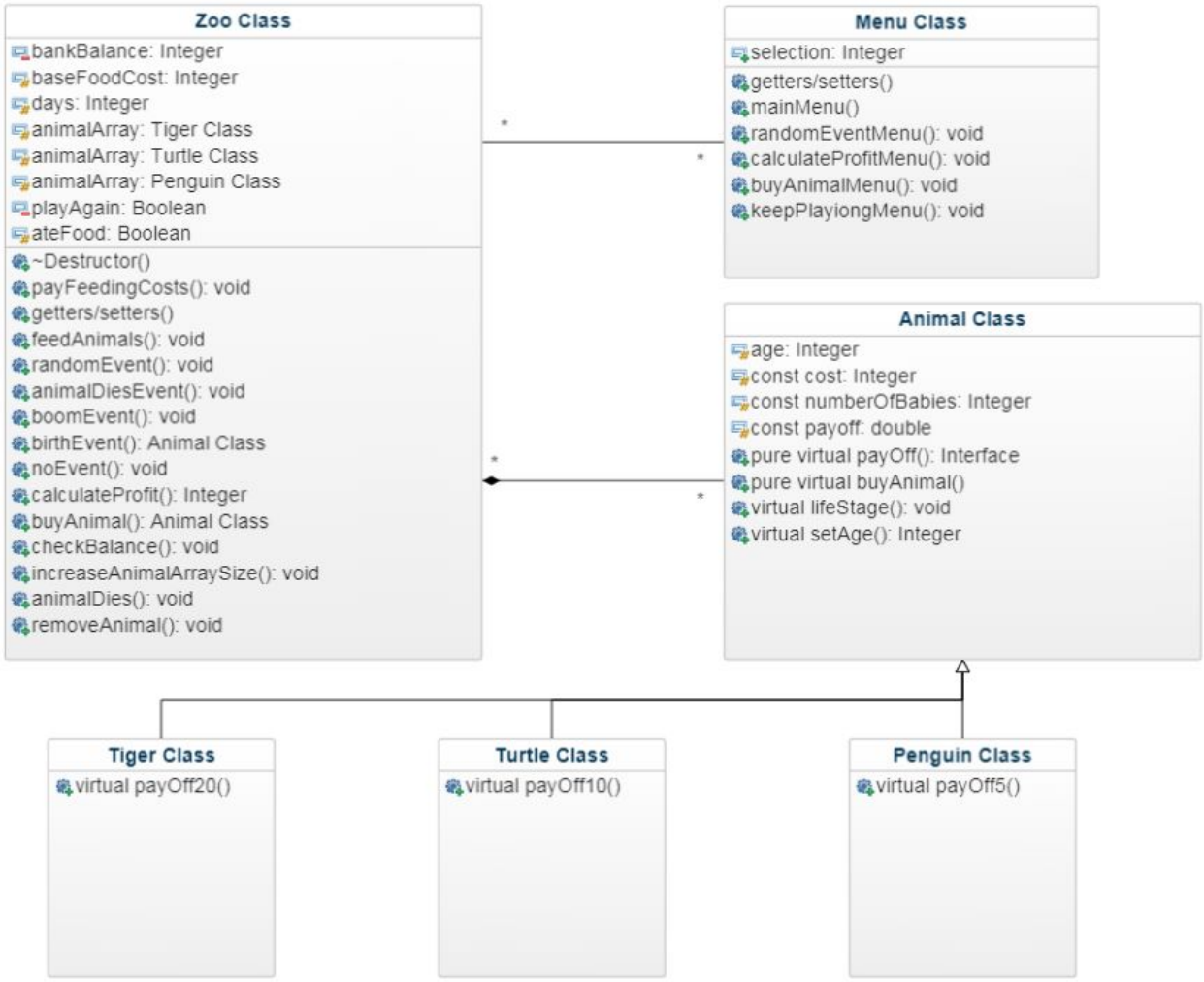
DESIGN

Today I am planning the design of project 2. I began with a top level design of the game architecture. For this game, we are required to have the classes Zoo, Animal, Tiger, Penguin, and Turtle. However, since I plan on reusing my validation menu, I plan on on creating a Menu class.

Today I was able to create a rough model of what my program will look like. I plan on making to use object composition where the zoo holds animal objects. The animal objects are classes themselves where the Animal class will be the base class and the Tiger, Penguin, and Turtle will be derived classes of Animal.

I have provided an initial sketch of the program architecture for the zoo game. I used object composition with the zoo class and animal class because zoos have animals in them. I think it goes without saying that animals are not zoos so there is no inheritance from the Zoo to the Animal. However, Tigers, Turtles, and Penguins are animals so therefore I made them inherit from Zoo. Assignment calls for dynamic arrays for each of time animal in the zoo so I have have included three data member variables in the Zoo class that are arrays of pointers that point to the animal class with each holding a specific type of animal pointer. I purposely added only a generic title to all of the getters and setters required to access and modify data member variables since naming them all adds little value to the diagram. Another item worth mentioning in the my planning phase is that the menu class will be associated with the Zoo class however there is no hierarchy between them and is only associated with the Zoo class. The Animal class is required to be the base class for the types of animal therefore many of the types of animals will

inherit from Animal. Animal will also have two pure virtual function and several other virtual functions meaning that Animal will be an abstract class. So far I have only identified two virtual functions in Animal.



Finally, since I will be dynamically allocating animals to various arrays, I have to make sure to deallocate them in the proper location which in this case, would be the Zoo class.

TEST TABLE

This test plan is provided to show how I tested the program. My goal was to create a test plan that is robust enough to provide assurance that the program behaves like it is meant to. The bulk of the test plan makes sure that the game parameters are properly set as this is what drives the rest of the program.

Test Case	**Input Values**	**Expected Output**	**Actual Output**
Main menu User can only purchase 1 or 2 animals of each time at game start	Input < 1 Input > 2 Input = any characters, spaces, and number combination I.e., 1 1 1, i.1 10000, -100000, 11.1, 11. 12	Any input that is not 1 or 2 should be rejected. Initialize animal quantities in internal data members of Zoo class (tigerQty, penguinQty, turtleQty) to the animals quantities entered.	Only characters 1 or 2 are accepted, all others rejected. The selected starting animals are properly initialized in the Zoo quantities data members. The actual output can be seen when the inventory report shows the animal quantities.
Select Feed User can only enter numbers 1, 2, 3.	Input < 1 Input > 3 char == any combination of numbers, characters, and spaces and decimals	Only values 1, 2, or 3 should be accepted. Should reject any values greater or less than this range. Characters and decimals are rejected. Spaces are rejected. Feed costs should be properly reflected	Only characters 1, 2 or 3 are accepted, all others rejected. Feed cost is properly reflected as shown in daily budget report.
Animal Diseases	Random Roll Let animal qty fall below 1	A random animal should die when the disease random event gets selected. If user has no animals of the type selected, no animals will die.	Animals are randomly chosen to die and are removed from animal count/inventory. If an animal is chosen to die and player has none, no animals die.
Spawn Animals	Random Event - spawn animal	Tigers should spawn 1 baby tiger, age 0 Penguins should	A tiger spawns 1 baby tiger when a tiger is selected to spawn

		spawn 5 baby penguins, age 0 Turtles should spawn 10 baby turtles, age 0	A penguin spawns 5 baby penguins when a penguin is selected to spawn A turtle spawns 10 baby turtles when a turtle is selected to spawn
Choosing whether to buy animals at the end of the turn	Input < 1 Input > 2 Input == any combination of numbers, characters, and spaces	A user is prompted to buy an animal with and the animal chosen should be purchased. Selecting 1 should continue with purchase, selecting no should go to the next menu.	Selecting 1 continues with next animal purchase screen. Selecting 2 skips the animal purchase
Choosing an animal to buy.	Input < 1 Input > 3 Input == any combination of numbers, characters, and spaces	The animal purchased by the user should be reflected accurately.	Correct animal count was observed. Zoo member function countAnimals() counts the animals.
Have all animals die and continue the next day and see the effect (i.e., are revenues and expenses still being generated?	tigerQty = 0 penguinQty = 0 turtleQty = 0; Have all animal quantities drop to zero.	No revenues or costs should be generated since no animals are owned. No change in bank balance	No revenues and costs were generated. Bank balance remains the same as prior day’s balance.
Bank Balance Accuracy	Daily budget report	The daily balance should be computed by first subtracting the starting animals costs from the bank balance of 100,000. Each day, revenues and expenses should be tallied, and any bonuses earned added to the bank balance. The bank balance should be accurate.	The daily balance menu display shows the correct prior day balance and current day revenues, expenses, and boom bonus. Bank balance is properly reflected to show net income/loss added each day.
Bankruptcy	Bank balance < 0	Program should inform user that they have gone bankrupt at the start of the day. A player can have a negative balance before the day ends	End of day shows that a negative balance exists from daily activities. On the next day, the user is informed that they have gone bankrupt and the program ends.

		however the player must be informed that they have gone bankrupt at the beginning of the next day and program must terminate.	

REFLECTION

During the implementation of my Zoo project, I had to quickly redo the architecture. I decided that it was best to include the Menu class within the Game class which would make it possible to communicate with the menu using the instance of the game class instead of doing it in main. Good thing I quickly realized this design change otherwise it would have been a waste of time trying to do it later down the road. Now the Menu class is inside the Game class (composition).

I was able to ask on Piazza how the ages were determined for each animal. Our assignment states that all newly bought animals at the start of the game are 1 day old. However, there is no mention of this is the age on the start of the day or if the age should be 2 on the start of the day. I decided to keep things simple and instantiate the starting animals with an age so that they will have an age of 1 on the first day of the game.

I found a similar question like this for when a player purchases animals at the end of the day. Since the assignment states that all purchased animals are 3 days old on the day they are purchased, then this means that the purchased animals will be age 4 on the next day. I identified a similar issue with babies born however that was more clear in the assignment. The assignment states that babies are born with an age of 0 during the turn so their age will be 1 the following day.

I was able to sketch out my initial design document showing the structure of my program as shown above in the design phase.

I was able to make good headway into starting my project. I first created the structure skeleton which consists of the classes and setting up the class hierarchy and class composition. So far it looks like this is the right structure to have. The project calls for a dynamic array for each type of animal with a starting capacity of 10 animals for each type of dynamic array.

I ran into difficulties during this part. I had first made my Zoo class to have private variables that were arrays of pointers to animal objects of size 10. However, I soon realized that this would not work because I would need to later change the array size whenever I need additional room to increase the size of the array and having a pointer of array such as array[10] in my .hpp file would not work since I would not be able to resize it later. Therefore, I instead removed the [10] brackets and just left an animal pointer for each type of animal. I had to research a bit on how to dynamically allocate my array using the class constructor and I found the solution I was looking for. This will now allow me to re-size the array later down the road. I used a pointer to a pointer where I could create a 2D array of Animals type. The 2D array contains 3 arrays (the rows), one for each animal type (since we must have a dynamic array for each type of animal) and the animal quantities are represented by the columns.

Since we are working with an array, we cannot simply increase the array capacity during runtime. So instead, after much research, I found that we need to create a new array, copy of the contents of the old array into the new array, delete the memory location being pointed to by the “old” array, and point the “old” array to the location being pointed at by the new array. Making sure to delete the old array location is important to avoid memory leaks. I tested the new array out to see if it was properly created by trying to access objects within the array bounds and outside the array bounds and I was able to confirm that it was properly working. I also tested out my project for memory leaks and none were identified. A big challenge I had in this section was discovering a bug that yields undefined behavior and still works. This leads you to believe that your code is working properly yet when you test it, you find that there are unexpected results. I discovered that you can

declare a pointer to a class object and set it to null and yet still be able to call methods on the null pointer. Here is the code snippet:

```
class Object {
public:
        void sayHi() { cout << “Hi”; } // method that displays hi
};

int main() {
        Object *objPtr = nullptr;       // pointer pointing nowhere
        objPtr->sayHi();                // returns “Hi”
        return 0;
}
```

The code above runs when compiled. I was getting this same behavior when using an array of pointers to objects as well. From my understanding, the code above should not run because the pointer has been initialized to null yet if you call a method on that pointer, you will be able to access the method of the class. After a lot of research, I found that this is the result of “undefined” behavior that works depending on your compiler. I found this bug when testing to see whether I was creating the right amount of Animal objects in my array. I was getting frustrated because my code was telling me that it was able to call the methods of non-existent objects! https://stackoverflow.com/questions/2474018/when-does-invoking-a-member-function-on-a-null-instance-result-in-undefined-beha.

I worked on finding a way to grow the array as needed. For this, I need a way to be able to detect when the size of the animal array is insufficient. I knew from my initial design that resizing the array was necessary however I was not exactly sure of the implementation details. After thinking about it, test coding for a bit, and a lot of research, I created int variables to track the number of each type of animal as well as a variable to track the total capacity of each of the arrays. That way, if during the game we need to add additional animals, I can test to see if the number of animals exceeds the total capacity of the animal array and if it does, then dynamically create a new array with double the starting capacity. The specs state that the array should be “double the starting capacity” so I take this to mean to increase the size of the prior array by a factor of two.

However after looking at my original game design, I realized that I might have made a big mistake in the type of array that needs to hold my animals. The

assignment calls for us to be able to dynamically allocate an array of each type of animal and my design shows three different arrays with animals. This means that I should probably instead use a pointer to hold arrays of animals. That is, what I need to do instead is create a pointer to a pointer so that I can create a 2D array. That way, each array will hold a different type of animal and I can keep it all together all in one place. This also makes more sense because I can set any pointers in the array to null when no animal objects exists and only allocate space for them when a new animal needs to be added or deleted.

I was able to create a test function for a tiger to check to see if I was properly adding animals into my array. I have not seen any issues yet.

Another item I did not include in my plan was making sure that I include data members for the Zoo class to track the max capacity for the animal arrays. Our array capacity has to increase by 10 each time we run out of room in the arrays. Having these data members will allow for quick tracking of this data.

I am glad that I chose to include my menu class inside of the game (has-a relationship) because I am able to call my menu within my game object fairly easily. I can say the same about the animal class as well. This has allowed me to drastically reduce the code in my main function. Now all I have to do is call one function in my Zoo object which starts a cascade of other Zoo class function calls which basically run the game.

The gameflow of the game had me stumped for a while. I knew that I needed some sort of loop however I was not sure what type of loop. Eventually, I determined that I needed a loop within a loop in order to get the behavior I needed. However, I separated both loops into two separate functions which made my code more readable.

I also added a data member to help keep track of the daily profit/loss for the day. I found it useful to do this because it is easy to reset to 0 each day and because all of the Zoo class functions will have access to it in computing and reporting profits

It appears that there are no virtual functions in our Animal class since there are no virtual functions in our assignment requirements. It is possible to add them later if I want to assign unique capabilities to each type of animal such as making them say a sound.

I worked on obtaining the animals ages and created a function in Zoo to determine the age. The Age is a data member variable in the Animal class and the tigers, penguins, and turtles inherit this from the base object. It was fairly straightforward to create this function. Creating the associate report in the menu class was more tedious since you need to format it for display purposes.

I was able to work out the game flow logic. This was a bit complicated since we are to have a turn that counts as a day where activities take place. I did not include a function in my design to begin the game, so I added one in the Zoo class which starts initializing the game. Here I added a main loop where a player keeps playing unless he quits or goes bankrupt.

I was able to add a display menu that shows a user how much they spent on their initial animal purchases at the start of the game as well as giving them an updated bank balance. This was not a feature I had originally planned for so I added it during implementation since it is a good idea to provide these updates since a player would want to know and because it also serves to check if the game logic is good.

As I got closer to the end of creating my project, it started to look that there would be no virtual functions in my animal class since no functions required polymorphism so my initial design plan shows virtual functions that were not used. Therefore, none were incorporated into the implementation phase.

My favorite part about this project was learning how to handle a greater variety of interaction between the classes. Keeping track of them all was a big challenge and many times I had to reread what I had previously coded just

to refresh how my algorithms worked. I especially learned more and how to properly track objects in dynamically allocated arrays.