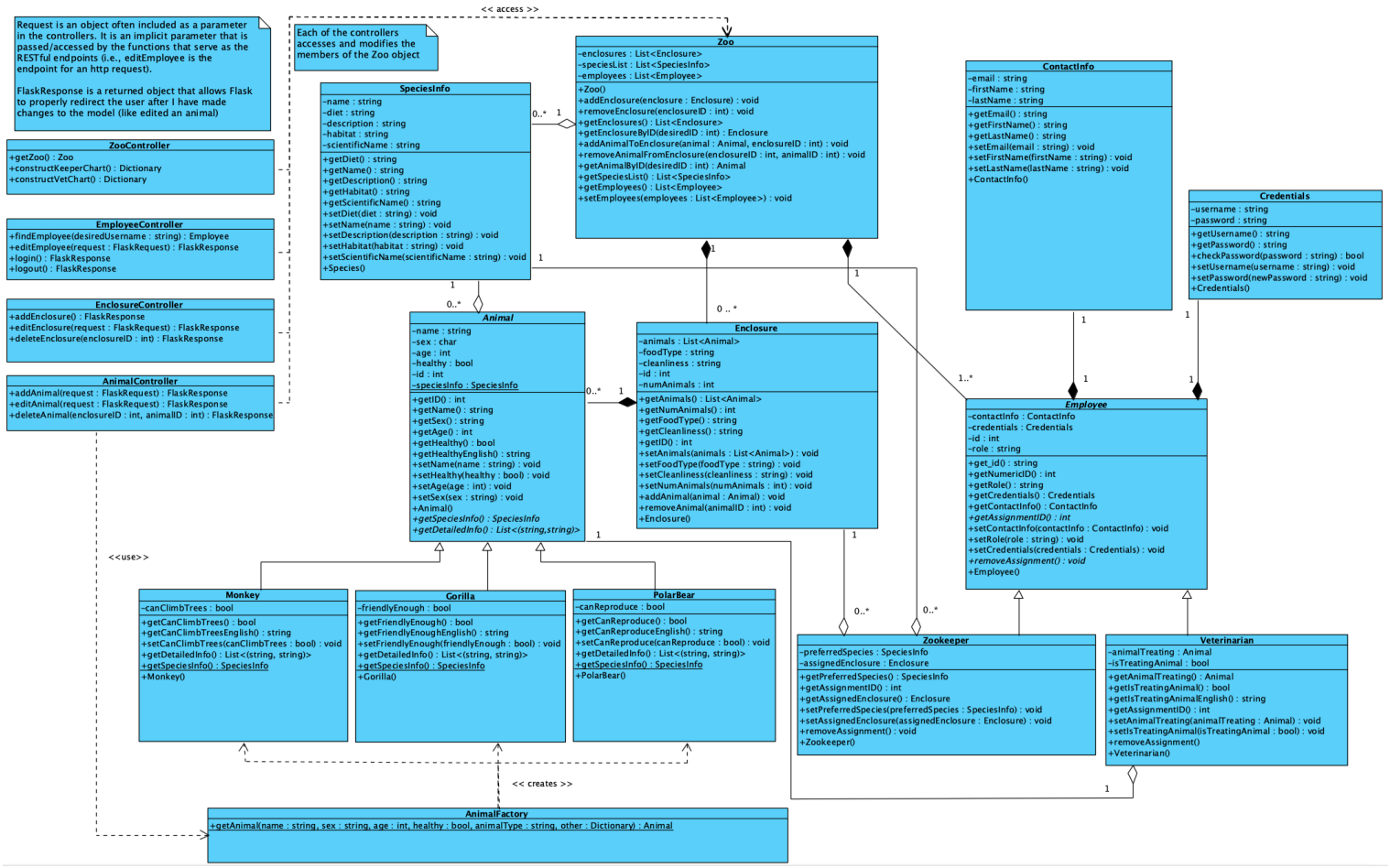
ZooManagement Part 6: Report

1. **Name**: Landon Baxter
2. **Project Description**: A Python webapp created with Flask that allows zookeepers and other zoo employees to easily manage their zoo. Employees can view and edit information about the zoo, including animal, enclosure, and species information. Depending on the position of the logged in user, the application will display different functionality.
3. **Implemented features:** I implemented all features outlined in Part 2. See below:

ID	User story
US-01	As an employee, I want to log in so I can view relevant information to my position at the Zoo.
US-02	As a zookeeper, I want to view information about the enclosures, including feeding info, cleanliness, etc, so that I can better understand the status of the zoo.
US-03	As a zookeeper, I want to view a summary of each species so that I can better understand the animals.
US-04	As a zookeeper, I want to be assigned to an enclosure so that I know where I need to work.
US-05	As a part of the zoo planning staff (zookeepers), I want to view a summary of the animals so that I can make decisions about the enclosures.
US-06	As a part of the zoo planning staff (zookeepers), I want to add enclosures so that guests can visit and see more animals.
US-07	As a part of the zoo planning staff (zookeepers), I want to remove enclosures so that we can make room for others.
US-08	As a part of the zoo planning staff (zookeepers), I want to add animals to enclosures so that they have enough animals.
US-09	As a part of the zoo planning staff (zookeepers), I want to remove animals from enclosures so that enclosures don't get too crowded.
US-10	As a veterinarian, I want to see a list of animals that need assistance so I can treat animals accordingly.
US-11	As a veterinarian, I want to be assigned an animal for caretaking purposes so that I can keep track of what animal I'm taking care of.
US-12	As an employee, I want to log out so that someone else can log in while I'm away.
US-13	As an employee, I want to change my details, including contact information, so I can keep my information up to date.
US-14	As an employee, I want to change my password so that my account is secure.

4. **Not implemented:** None
5. **Final class diagram (see next page, or follow the link for a full size image: https://i.imgur.com/ZzSJ4ly.jpg)**



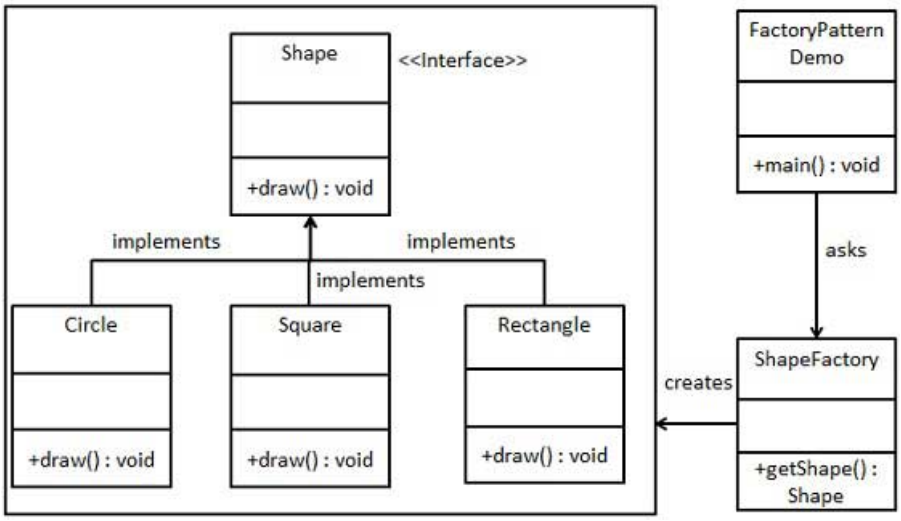
In short, the class diagram consists of animals, which belong to enclosures. The employees are partially composed of an animal or enclosure based on their assignments. For example, if a veterinarian is assigned to an animal, the animal object would be stored in that vet's object, while also existing in its enclosure. The set of enclosures, employees, and species are held in a zoo object that contains all information regarding the zoo. This zoo object is then accessed and modified by a set of controllers. Once the controllers edit the animals/enclosures in the zoo object, they redirect the user accordingly. Last but not least, the controllers make use of an Animal Factory, which streamlines the instantiation of new animals of different species.

a. **What has changed since part 2:**
   i. I originally did not have subclasses of Animal. This limited my ability to extend the functionality of the Animal class; I wanted to add more data and functions to different species, so I decided to subclass Animal.
   ii. I originally planned on making different enclosure types, but decided against it. It would have made the project needlessly complex for little gain.

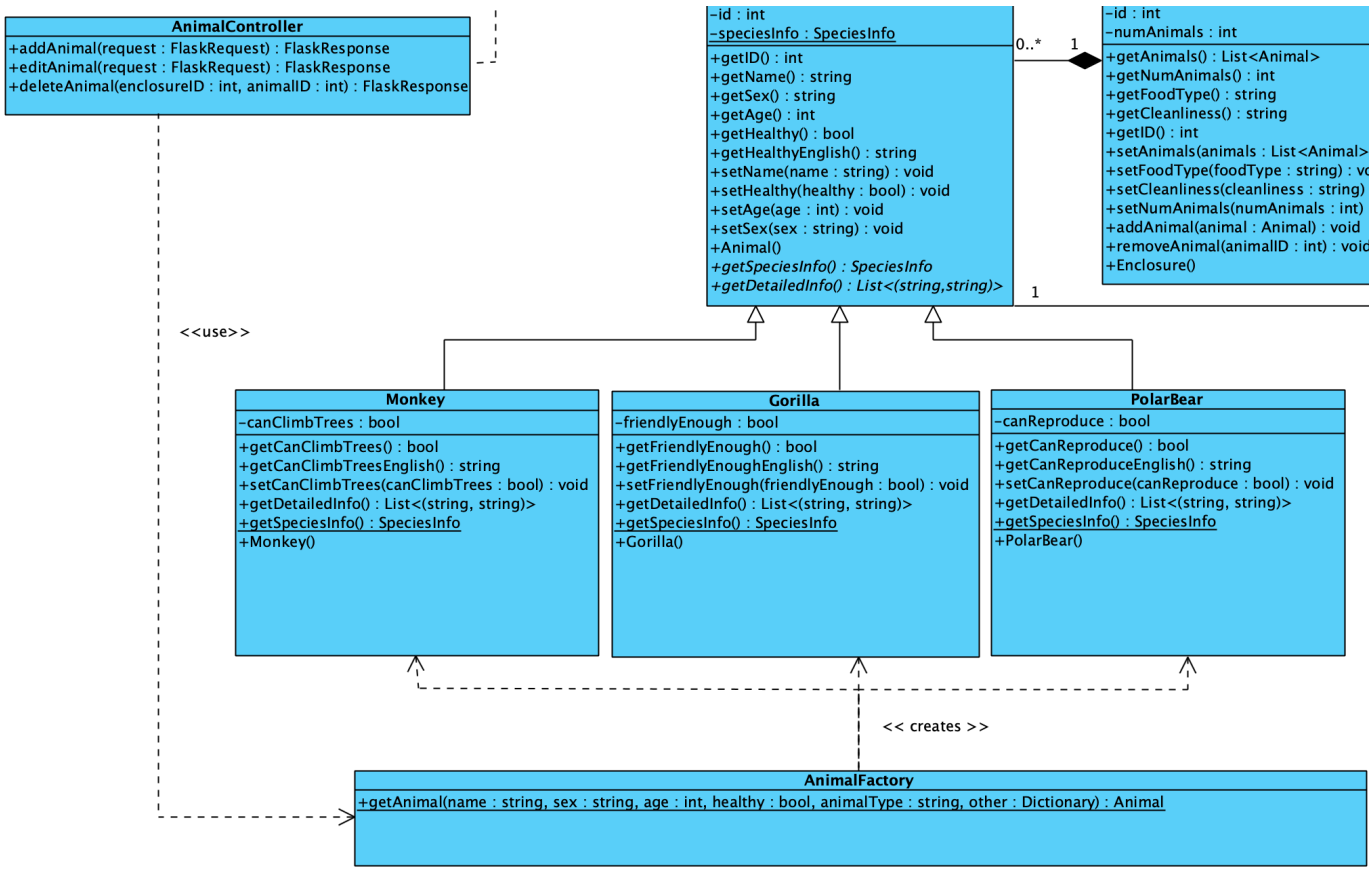
iii. I merged planning staff with zookeeper. These two roles seemed too similar after some thought.
iv. I added the controllers that were needed to achieve my desired functionality. Each controller has a set of methods that edit or update the models (animal, enclosure, zoo, employee).
v. I simplified ContactInfo a bit by removing a few variables; the application has no need for a user's address, for example.

6. **Design pattern: Factory**
   a. **Why and how**: In my project, I wanted to subclass the "Animal" class because each species can have different data and methods (e.g., a gorilla must be labeled as friendly enough to be in an enclosure with other animals, while a monkey has no need for this). Instead of cluttering my controllers with if-else blocks, I wanted to use the Factory design pattern to create instances of animals. This makes the controller code much more readable and allows for easy extension for more species because we are encapsulating what varies (the if-else block for instantiating Animal objects). Moreover, it abstracts away the implementation details of the instantiation of different objects, allowing the controllers to code to an interface rather than an implementation. The controllers do not need to have knowledge of the process by which objects are instantiated; rather, they simply need to know what parameters to pass to the factory. These benefits make the system much more robust, easy to understand, and easy to extend in the future when more species are inevitably introduced to the zoo.

      I implemented Factory with an AnimalFactory class. This has a static method getAnimal that takes a few parameters, including the animalType (species name, like "Monkey"), and creates a corresponding animal of that species.
   b. Factory Design Pattern (source: https://www.tutorialspoint.com/design_pattern/factory_pattern.htm)



   c. Factory in my class diagram (next page)



When a user submits a request to make a new animal, the application calls a method from the AnimalController, namely addAnimal(). The AnimalController then calls the static getAnimal function from the AnimalFactory class, passing in some necessary values and the name of the species (animalType). The AnimalFactory will then instantiate the correct animal and return it to the AnimalController, which updates the zoo accordingly. Compared to the Factory design pattern seen in part b, my AnimalFactory is like the ShapeFactory, the AnimalController is like the FactoryPatternDemo, and the animals are comparable to the shapes.

7. **What I learned**: During this process, I learned how important it is to do your research and planning ahead of time. I often would forego the planning stage during personal projects, but the class diagram and definition of user requirements made the implementation a lot easier. Instead of making things up on the fly, I could follow a clear structure and make changes where necessary. I have also learned how convenient it is to separate the concerns of different functionality and data. For example, I can now expand upon ContactInfo to include more data if I would like, all without having to modify the user class much at all. Additionally, I have learned about the importance of design patterns. I ultimately chose Factory, but when I was considering different designs of my project, I considered many other patterns, such as Template. Understanding and being able to apply these patterns clearly makes implementing a complex system a lot easier; additionally, the system becomes much easier to expand upon. Lastly, I have learned a lot of useful strategies for eliminating needless if statements. The inclusion of

different abstract methods and classes throughout my system, such as getDetailedInfo() for animals, made it very easy to design the views and controllers. Instead of having to check the subclass of each object, I learned how to call methods that were included in the base class. I can see myself using this type of polymorphism in the future to make my code cleaner and easier to extend.