**9.5 Finale Project**

The idea for this project came from a toy I had in elementary school called a Tamagotchi pet. A Tamagotchi is a little handheld egg with a small screen and three buttons. When you first got it, a little egg would appear on screen, after a while a pet would pop out for you to care for. You could feed, play with, discipline them, and clean up after them. If you over fed, did not play with, or did not clean up after your pet they could become sick and die. This was the basic idea for my project. Users can buy various types of pets. Pets each have a happiness level, bathroom level, hunger level and energy level. Users can interact with there animals to care for them. If the animal is not cared for in a certain area they will run away. My idea was these animals could be some virtual pets you have that you care for when you are on the computer. It is not meant to be challenging or something that takes a lot of time to maintain but something you can check on occasionally, and if you forget you lost them.

The three patterns I chose to implement were the Factory, Template and Observer patterns. I will be using the Shevts pdfs given in the course reading for each pattern for my sources. The factory pattern is “A creational design pattern that provides an interface for creating objects in a superclass but allows subclasses to alter the type of objects that will be created.” To relate this to my project I have and abstract animal class. This animal’s class declares all methods and fields that each concrete animal must have. I chose to make the animal class abstract instead of an interface because all animal types have some behaviors that are similar, this way all similar fields and methods can be stored in the animal class and then subclasses can make a call to the base class instead of duplicated code in each subclass. The IAnimalFactory is an interface that describes the methods all concreate factories will use to create and set up animals. IAnimalFactory has a create method with a return type of Animal. Each concrete factory returns the animal type that the user selected, because all concreate animals are of the animal class this works. The concreate factories also set up the animas traits that are unique to the animal types like happiness, hunger, energy, and bathroom level and gives the animal the correct toys.

The Template pattern is “A behavioral design pattern that defines the skeleton of an algorithm in the superclass and lets subclasses override specific steps of the algorithm without changing its structure. The abstract class is the animal class. The abstract animal class oversees setting up the animal’s timer as well as the logic behind what should happen when the timer elapsed. When the timer elapses, the abstract class checks the animals’ levels. If one of the levels is below where it should be the abstract class calls a method to fix it. For example, if the energy level is less than or equal to zero the animal class calls its sleep method. The concrete subclasses of the animal override what happens in the Play(), Eat(), UseBathroom() and Sleep() methods. Each animal type has a unique behavior on how they use these methods. This is very helpful because instead of having to set up a timer and a method to deal with it ending in each type of animal class, they can all share the same template and use it to their own needs.

The Observer pattern is what allows me to keep the main window updated with pet’s information. I also found this pattern challenging to implement, I am still a little unfamiliar with using action and delegates and how to use them effectively, but I managed to get it working and I believe I implemented it correctly. The Observer pattern is a “Behavioral design patten that lets you define a subscription mechanism to notify multiple objects about any events that happen to the object they’re observing.” The pattens has an ISubject interface, this interface has methods to register and remove observers of the IObserver interface as well as notifying observers of state changes. The IObserver just has an update method. In my project the class inheriting the ISubject was the abstract animal class. I wanted the main window to automatically reflect state changes from the animal. To do this I made the person class inherit the IObserver interface. When an animal is added to the person pets list the animal registers the person to them. Every time the animal changes its state it notifies its owner. The persons update action is attached to the main windows populate list box method so every time an animal notifies the user, the user calls its update action, and the main window populates its list boxes.

I will also be using the Shevts pdf on the SOLID principles for my sources. The single responsibility principle states “A class should have just one reason to change” I find that the single responsibility principle is very easy to understand while also being kind or difficult. To explain, I think the idea of a class just overseeing one aspect of the program makes a lot of sense, you can keep everything separate and not have a giant class, I believe these are called blobs. However, where it becomes difficult for me is decided when I have separated a class to much. I do not think I fell into that problem in this program because for the most part classes are only focused on changing themselves. The best example of the single responsibility principle in my program is the SaveAndLoadPets class. This classes only job is to save and load a list of pets, I originally had these methods in the main window but by separating them I was just focusing on button clicks in the main window and moved saving and loading to its own class. I think an area I could improve on this principle is in the animal class. The animal oversees setting its timer and the algorithm when the timer elapsed. I could make a separate timer class that is focused on doing this which would allow the animal to just focus on changing its state.

The Open/Closed principle states “Classes should be open for extension but closed for modification”. Two great examples of this principle in my project are the animal factories and the animal types. I created an IAnimalFactory class that describes the methods every factory should have. This class would be very easy to extend, say you wanted to create dinosaurs you could create a dinosaur factory to return animals of that type. If the dinosaur factory had a create method with a return type of animal, it would work. The animal types are another example of this principle. The animal abstract class hold all methods similar to all animals. To extend the animal class you just need a class that eats, uses the bathroom, sleeps, and plays if it does that the class can be extended.

The Liskov Substitution principle states “When extending a class, remember that you should be able to pass object of the subclass in place of object of the parent class without breaking the client code”. A good example of this principle in my project was the ToString override in the animal and sub animal classes. The abstract animals ToString override displays all the traits common to all sub animals like happiness, energy, hunger, and bathroom level. In each sub animal class, they also have a ToString override method. First the sub animal calls the base to string in the abstract animal class. The sub animal then takes this string and add the unique element specific to this sub animal. This way sub animals are still compatible with superclass and instead of just replacing what the superclass does it extends the base first.

The Interface Segrgation principle states “Clients shouldn’t be forced to depend on methods they do not use”. I followed this principle by keeping my interfaces lean. Each interface only has methods that each class that implements it would need. My IObserver method only has an update method because that is all the IObserver needs to know or do. The IAnimalFactory only has a create, set state, and give toys method because that is all the factory methods need to create animals.

The Dependency Inversion principle states “High level classes shouldn’t depend on low-level classes. Both should depend on abstraction. Abstractions should not depend on details. Details should depend on abstraction.” I find this principle the most difficult to comprehend, I think I understand the idea of it, but the wording trips me up. I believe I implemented this principle in the PurchasePetWindow class. In this class there is an animal factory field of type IAnimalFactory. Depending on which type of animal is selected to buy this animal factory field instantiated a concreate factory to create the animal. By doing this the creating method is dependent on an abstraction and the detail of the animals creating depend on the abstraction as well.

They struggles I had were setting up saving/loading and the observer pattern. As I stated early setting up the action to automatically update the main window was as bit confusing. After reading some documentation and going over the assignment in class I was able to get it working. My only question is if I implemented the pattern efficiently of if there was an easier way to implement it. My other struggle was with saving and loading. My problems came because as first I was declaring the whole class serializable, because I had some timers in the animal class it was not able to serialize that in the saving process. The only thing I would add is I know how to set that process up to load and save something, but I am not quite sure what is happening at each step, it’s not totally necessary right now I just needed it to work but some more reading in that area could probably help me get a better understanding.

If I had more time, I would add some mini games of some sort for users to manipulate their pets, this would make the process much more engaging for users. I would also set up some visual icons for the pets. I gave users the option to select their pet color in the hopes of being able to use some clip art of animals in the program. This way you could show a graphic of our specific-colored animal doing things like going for a walk or a laser pointer with a cat. I am not sure how to get images working in a WPF window and because I was sort on time that idea got scrapped. Overall, I had a lot of fun creating this project and with a few tweaks and improvements I could see it being a fun little mini game to keep running in the background on your desktop to check in on your pets when you are bored or need a break from work or studying.

(Also a faster way to see the process of animals running away or manipulating themselves is to increase the changes in the Animals Time Elapsed method form 1 to 10 or make the timer faster.)