Project Part 6: Final report & Code Completion

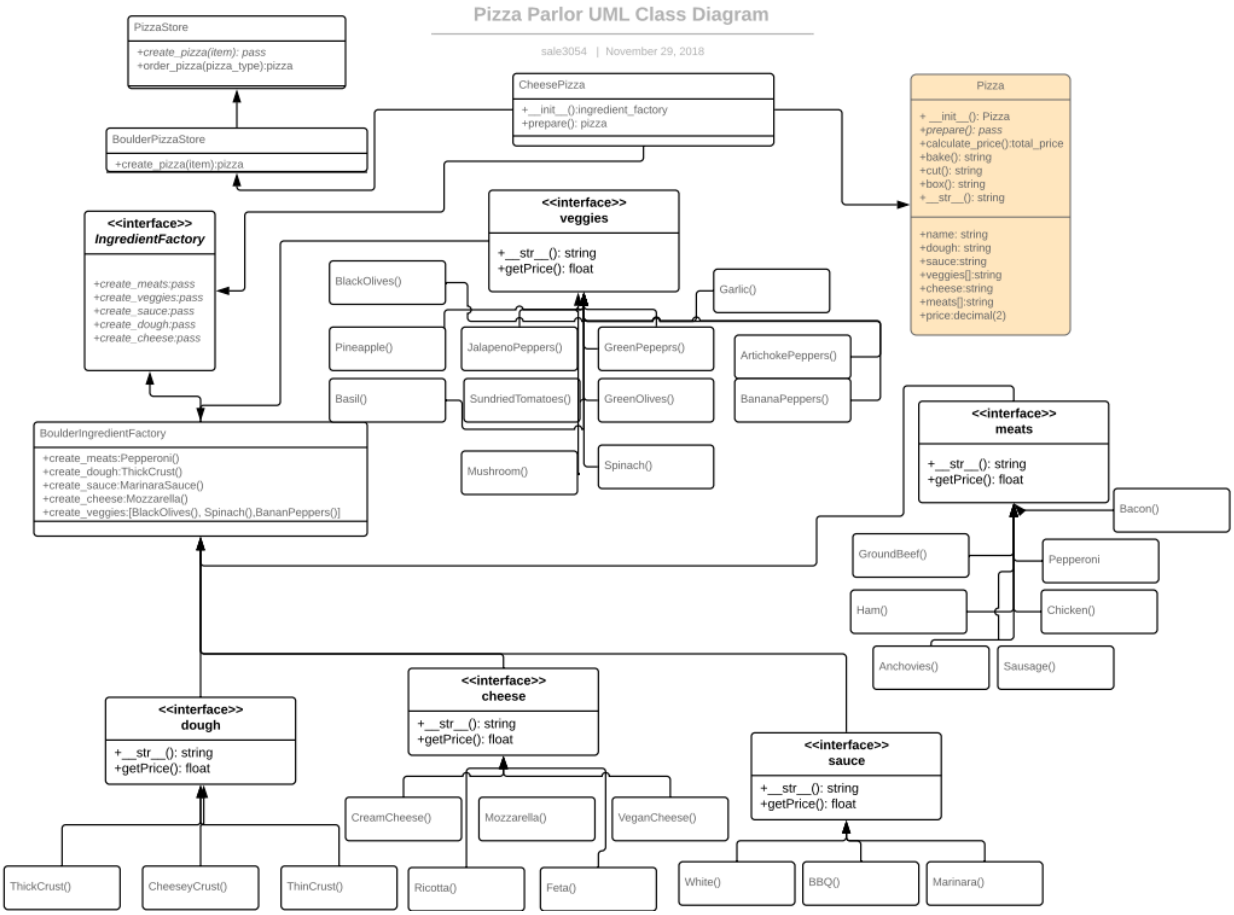
1. **Name:** Samuel Leon
2. **Project Description**: Lightweight Pizza Parlor web application that utilizes Flask and the SQLite3 database framework for Python to allow for users to order a pizza that they desire. This system utilizes the Factory Design Pattern due to the similar nature of Pizzas.
3. **Implemented features:** (These were not in the github, just with me locally as I had started back over, and was unsure of the protocol for this.)

ID	Description	Priority
UR-01	User can create Pizza	Critical
UR-02	User can view what's available	Critical
UR-03	User can interact with order form	High
BR-01	Store user info in Database	Critical
BR-02	Must use an OO framework (Flask)	Critical
BR-03	Be able to add new pages easily	Critical

4. **Not implemented (but planned) features:**

ID	Description	Priority
UR-04	User can "Make your own" Pizza	High
BR-04	Allow for foods other than Pizza (i.e. Breadsticks, garlic knots, etc.)	High
BR-05	Allow for users to submit custom info w/ each order	Medium
BR-06	Implement relational database rules for SQLite	Medium
BR-07	Add style sheets for web app	low
BR-08	Put app up on live server	low
BR-09	Implement multiple Pizzas (Other than Cheese Pizza)	High

5.**Show your final class diagram.**



Unfortunately, the whole diagram changed (about three times). At first, I was unsure what design pattern I would be implementing. However, as I was slowly sold on the factory method, I had to design around that pattern. I realized that the implementation would need to be fairly complex to allow for as many ingredients as would be necessary in a Pizza parlor. In a static context, concrete classes are okay, but in a fast paced business like a Pizza parlor, that is not so.

6. As a result, I opted for less and less concrete classes so that I could create a multitude of abstract factories. These factories are then handled through their interfaces. I selected the factory design pattern as it seemed to be the obvious choice for creating items of similar-but-different blueprints. A pizza is basically the same across the board, but with very minute differences. As a result, there's no good reason to recreate the wheel at every step for each new pizza item we create. Further, if we were to own a pizza parlor, having so many ingredients could get expensive *fast*. As a result, it would make sense that it should be easy to remove mentions of an ingredient and replace it with another without having to rewrite the whole codebase. As things stand in

this implementation, due to the abstract factory design pattern, it is the case that almost all of the code could be deprecated and replaced with minimal overhead. That is *incredible!* This code was a pain in the $!@ to set up, however, now that it is at a certain point, maintenance is way easier than its inception.

So, how was it done? The Pizza class stores all of the important attributes of our Pizza. When we get down to a base case, we'll have something like CheesePizza(), which is a Pizza object, that was created by a Pizza Store. The Pizza Store creates the Pizza by using the functions received from the Ingredient Factory. And then, this all goes further up their inheritance trees to their base cases. Through the use of a lot of abstract methods, polymorphism, and interfaces, the Abstract Factory method was formed for this Pizza parlor.

7. **What have I learned?**

First, I really hate UML diagrams. I *hate* them! They are a pain to make.

Second, I love UML diagrams. I *love* them! They are a pain to make. Once you have them, they make coding much easier, as you're just following the path you created for yourself. Unfortunately, however, planning ahead to such a large degree often makes me feel like an orangutan. This brings me to my first point; it's a good idea to plan ahead, however, **implementation specific ideas should not be placed anywhere in the overarching conceptual planning**. Why? Because it often ends up being the case when we are writing a program, that it is the first time we are writing that *thing*. As a result, it is very likely that I will not plan for a certain pitfall or corner case that outside of what was originally anticipated. My subset of anticipated problems was incomplete. As a result, it is important to keep each job specific to its role. The conceptual overlook is for just that. Don't skimp on planning.

Thirdly, **don't code to what you know, code to what is right for the job**. I tried to force this project in eight different directions at first, as I was being stubborn and didn't want to learn a new framework. As a result, trying to take the shortest path, I took the longest path on a few occasions; eventually having to rewrite the code anyway, after my hacks couldn't take me any further. On this note, it is also important to frequently reassess how accurate your original diagram and model is with the current system. If it no longer fits/makes sense with the reality of the present product, it's time to make some changes.

Fourthly, the **guidelines for the GoF Design Patterns are just that, *guidelines***. It is impossible to follow the guidelines of the Factory Design Method to the Tee, as one of its specifications is that you don't use *any concrete classes*. Now, that's all fine and dandy until you have a giant glob of abstract classes with nothing to do. Then you start back at round one and go, "Wait, so what are we doing here?"

Finally, **I learned how to use the Factory Design Pattern, and where it is appropriate**, **and what many design patterns *shouldn't* do.** Being new at something is really good for sucking at something, and when it's a mystery, you're bound to use whatever it is in unusual ways. This happened on a myriad of occasions throughout the progression of my project. So, what's important? The Factory Design Pattern is used in the case that there are many similar but just *barely* different objects wandering around in a database or store. For example, an industrial car line. All the cars are the same model, but some of them are *blue* and some of them are *green*. Just because that is the case does not mean I am going to rewrite the whole program for blue and green cars (*I hope*). When we implement the Factory method, not only can we account for minute changes like that, but we can totally reconstruct an item and make substitutions based on what is currently available. In a concrete class, this wouldn't be the case without some hardcore overhauls of the code.

Learning how to properly utilize dependency inversion has reformed my perspective on coding- and I think I'm better for it, both organizationally and in my understanding of *why* it's important.