Name: Alexander Ray

Project Title: *Cinnamon: Money Management in a Spicier Package*

Project Description (similar to Pt 2): *Cinnamon* is a personal money management app–akin to Mint–to help track day-to-day spending from different user-defined accounts, including support for external report generation and multiple types of quick summaries. This app facilitates smarter, more conscientious spending habits from it's users and incubates a goal-oriented mindset to money management through its explicit, no-nonsense interface. Specifically, *Cinnamon* provides utilities for purchase tracking in both the retail and dining spaces. Factoring in information about the user including location and income, *Cinnamon* provides an avenue to easily track spending, provide up-to-date information about accounts, and generate CSV and JSON reports for external consumption. Finally, *Cinnamon* provides support for account "extras" like saving by rounding to the nearest dollar as well as live email updates on spending.

Features that were implemented:

Use Case ID	Use Case Title
UC-1	Sign Up
UC-2	Log In
UC-3	Log Out
UC-4	Add Account
UC-5	View Spending
UC-6	Log Spending
UC-7	Generate Report

Every use case listed in Part 2 of the project has been implemented here. The rubric asked for only ID and Title in this table–to see descriptions, see my submission for Part 2.

Features that were not implemented:

There were no features listed in Part 2 of the project that have not been implemented.

Final Class Diagram:

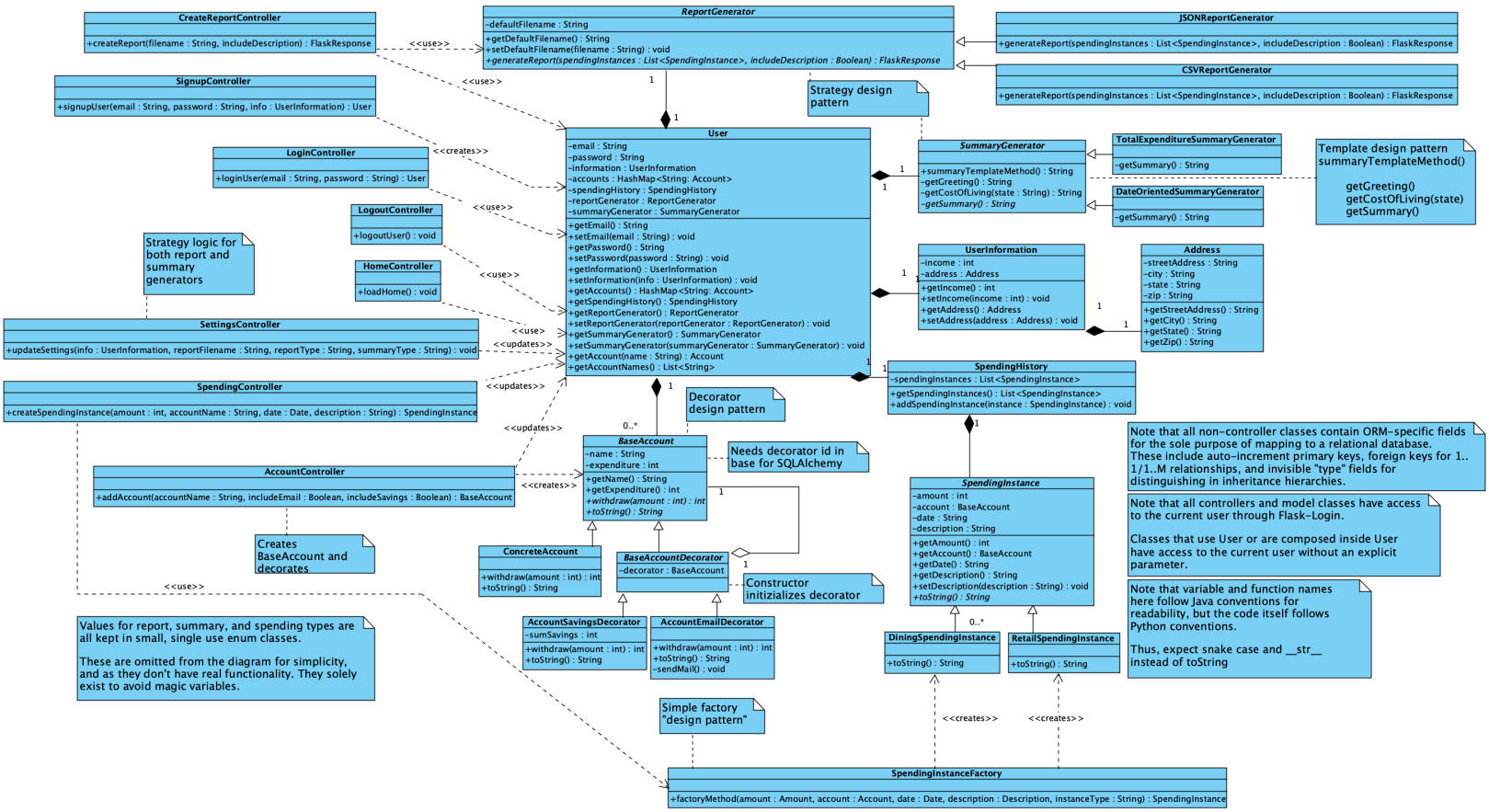

Figure 1: *Full class diagram*

Overall, there was not that much change between the original UML class diagram from Part 2 and this one. The following are notable alterations between the two versions:

1. Decorator was added to include additional functionality with Accounts. This pattern wasn't considered for the previous diagram.

2. Strategy and Template Method were included for summary generation. This functionality wasn't considered for the last diagram.

3. Spending instance creation was amended to use a more rigid version of the Simple Factory Method design pattern that was discussed in class.

4. Controllers were split up to be one method per class, mostly due to constraints with the Flask framework.

5. `ReportGenerator` now generates `FlaskResponse` objects as that affords the ability to serve static files (reports) from a Flask site. This detail wasn't considered during Part 2.

6. A few smaller changes such as removing unnecessary getters were made as the implementation made clear some shortcomings of the original diagram.

   Overall, the diagram from Part 2 was a very good diagram and the parts that were planned at that stage "aged" remarkably well as the project went on. Going back through to update the diagram, I realized that I was truly able to write code while referring directly to the diagram and that shows in the final code; it made code creation much quicker and also allows for explicit iteration on design concepts (as opposed to just quick mental thought).

Design Patterns:

1. Strategy, used for report generation.

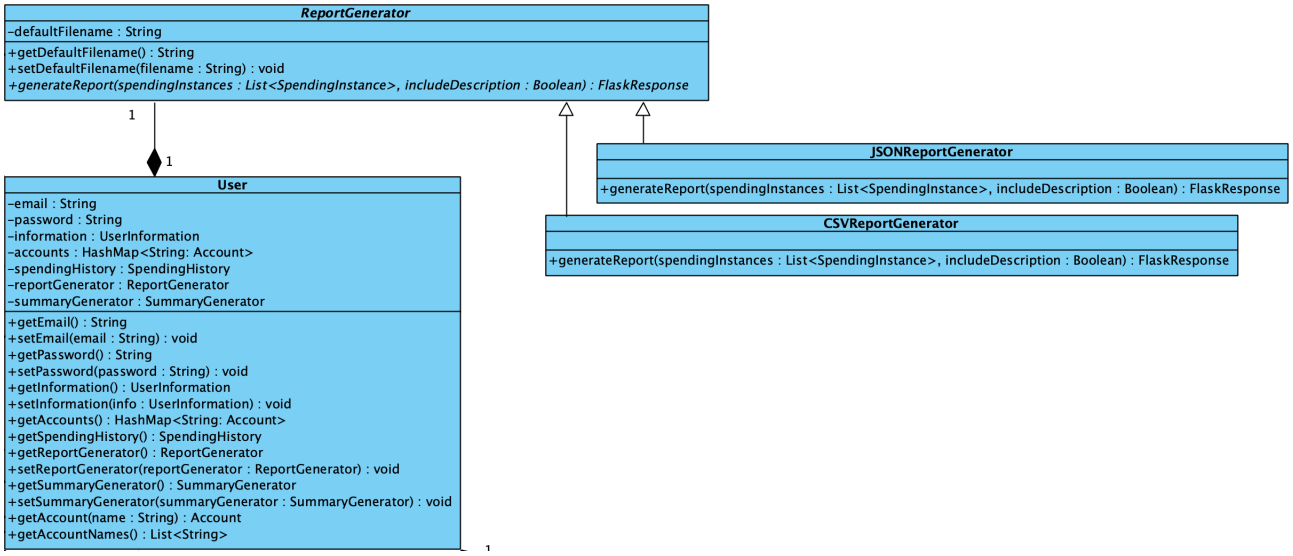

Figure 2: *Strategy design pattern used for report generation.*

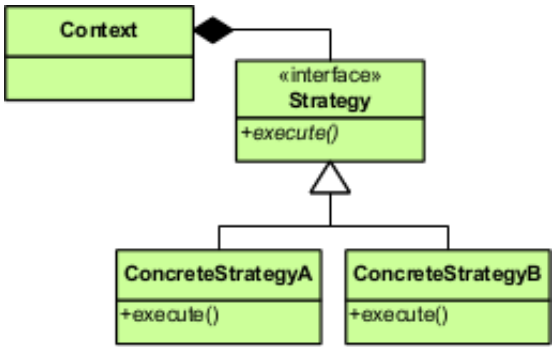

Figure 3: *Example strategy diagram, from https://en.wikipedia.org/wiki/Strategy_pattern.*

I implemented this design pattern with an abstract class `ReportGenerator`, with the two concrete classes for CSV and JSON generation inheriting from this class and implementing the abstract generator method. The `User` object contains a `reportGenerator` variable. As mentioned in the full class diagram, the `SettingsController` contains the logic to switch strategies at runtime, based on user input from the settings page.

I chose this design pattern for the ability to choose a report generation algorithm at runtime. The abstract report generation class affords the ability to maintain convenient

data like a default filename, etc, but the user is still able to configure the report generation algorithm through the settings page.

2. Template method and strategy, used for summary generation.

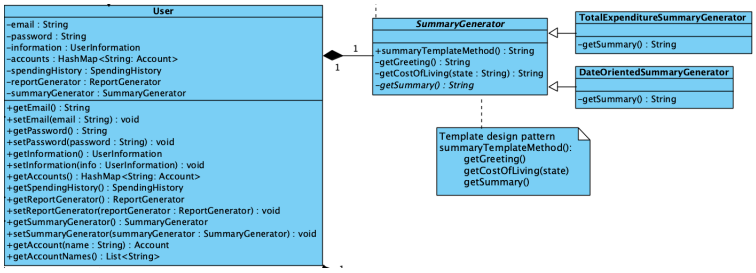

Figure 4: *Strategy and template method design pattern used for summary generation.*

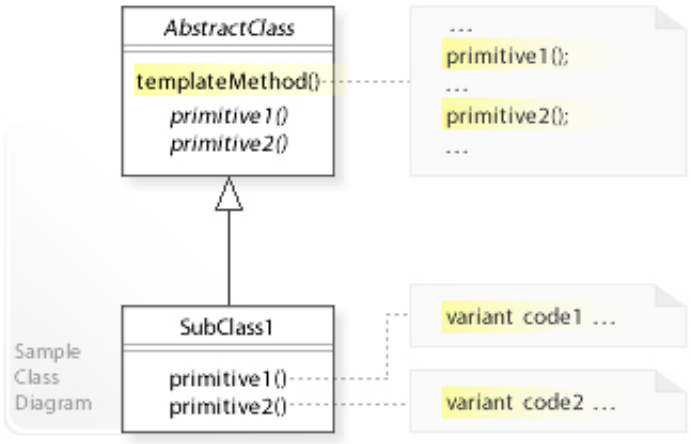

Figure 5: *Example template diagram, from https://en.wikipedia.org/wiki/Template_method_pattern.* ***The strategy pattern diagram can be found in the previous bullet.***

I implemented these design patterns with an abstract class *`SummaryGenerator`*, with the two concrete classes for different summaries inheriting from this class and implementing the abstract generator method. The `User` object contains a `summaryGenerator` variable. As mentioned in the full class diagram, the `SettingsController` contains the logic to switch strategies at runtime, based on user input from the settings page. Note that this setup is, as described, exactly the same as report generation. As with report generation, I wanted to use strategy to afford runtime switching of algorithms.

Like report generation, I wanted the user to have the ability to switch algorithms at runtime; thus, I used the strategy design patternt to compose the algorithm inside

the user. However, unlike the report generation, summary generation has significant shared code and shared steps within the overall algorithm. So I switched around the internals of the abstract class to define an explicit template method and concrete implementations of two-thirds of the algorithm (a greeting and a statement about cost of living). Thus, the subclasses of `summaryGenerator` implement the single abstract method. The `sendMail` method in the email account decorator and the `loadHome` method in the home page controller class both call `summaryTemplateMethod` from the composed object in the current user to generate an up-to-date summary.

The first part of the template algorithm is a simple generic greeting. The second makes use of the user's location to give coarse-grained information about the user's cost of living in their area; this is intended to remind the user about the area they live in and how it affects their financials. The overall choice for summary also makes use of the user's income when presenting their expenditure; conversely, the date-oriented summary allows the user to see their spending over multiple time ranges.

3. Decorator, used for dynamically adding functionality to user-created accounts.

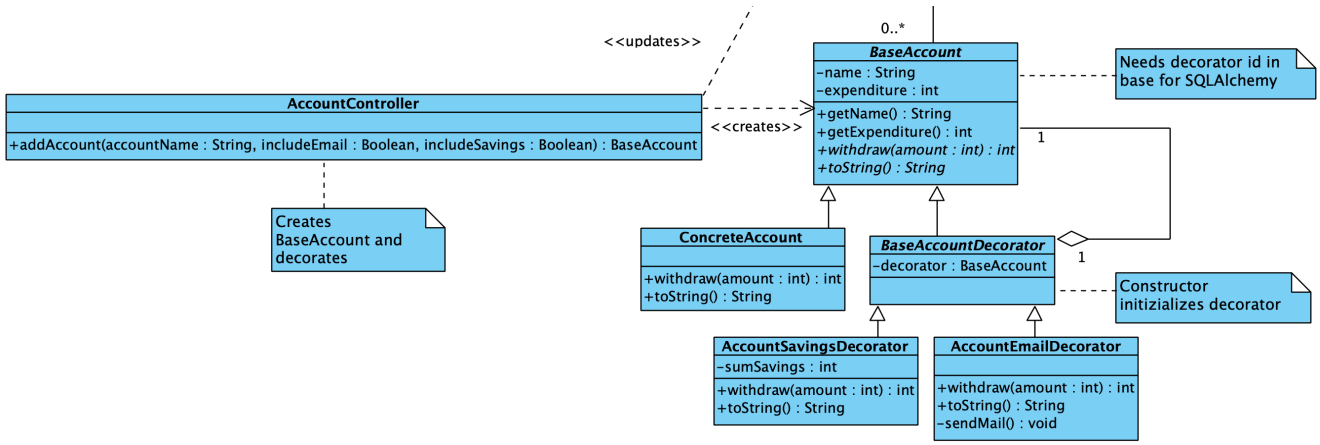

Figure 6: *Decorator design pattern used for accounts.*

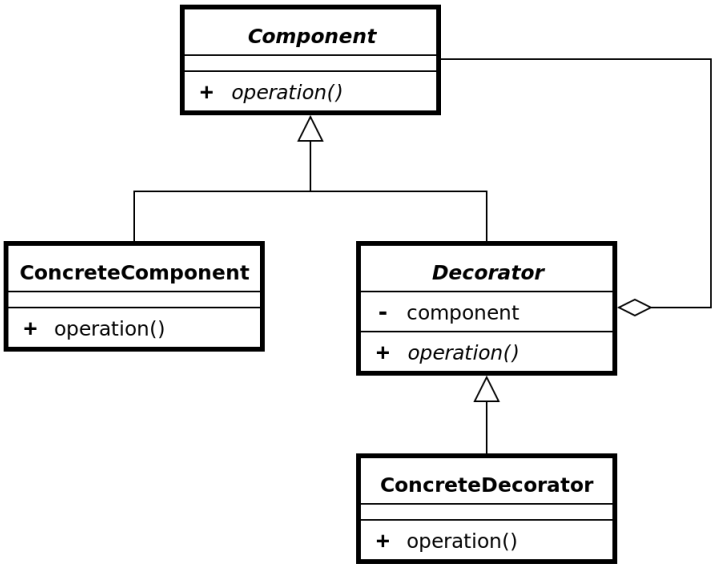

Figure 7: *Example decorator class diagram, from https://en.wikipedia.org/wiki/Decorator_pattern.*

I implemented this design pattern with an abstract class *`BaseAccount`* as the main component, with a concrete subclass `ConcreteAccount` and an abstract decorator subclass *`BaseAccountDecorator`* which contains a reference to the *`BaseAccount`* it will decorate. *`BaseAccountDecorator`* has two concrete decorator implementations, corresponding to functionality that saves money to the nearest dollar on withdrawals and functionality that emails updates to the user on withdrawal. Savings are reflected in the string representation of an account as well.

I used this design pattern to allow functionality to be dynamically added to accounts, without needing every combination of subclass (e.g. `Account`, `AccountWithEmail`, `AccountWithSavings`, and `AccountWithEmailAndSavings`). Furthermore, I wanted to make accounts more open to extension, as there are many other pieces of functionality that accounts could have. `AccountController` creates an account and decorates it appropriately based on user input.

The major difficulty with this pattern was that mapping this to a database with an ORM requires both the base class and any decorators to be distinct rows in one (or many, depending on implementation) table. Thus, I needed to make `withdraw` return the current value for the `expenditure` variable to allow all classes associated with the overall account to update their `expenditure` values. Furthermore, while the

`decorator` variable was able to be properly placed in the abstract decorator, the foreign key `decorator_id` for this self-referential link needed to be placed in *`BaseAccount`*. Beyond those notes, the implementation of this design pattern closely followed the in-class discussion as well as the implemention in Homework 2.

4. ***Note that this is not an actual GoF design pattern, I just wanted to include the explanation of my thought process. My three required design patterns are listed above.*** Simple factory (or factory, depending on who you ask), used for spending instance creation.

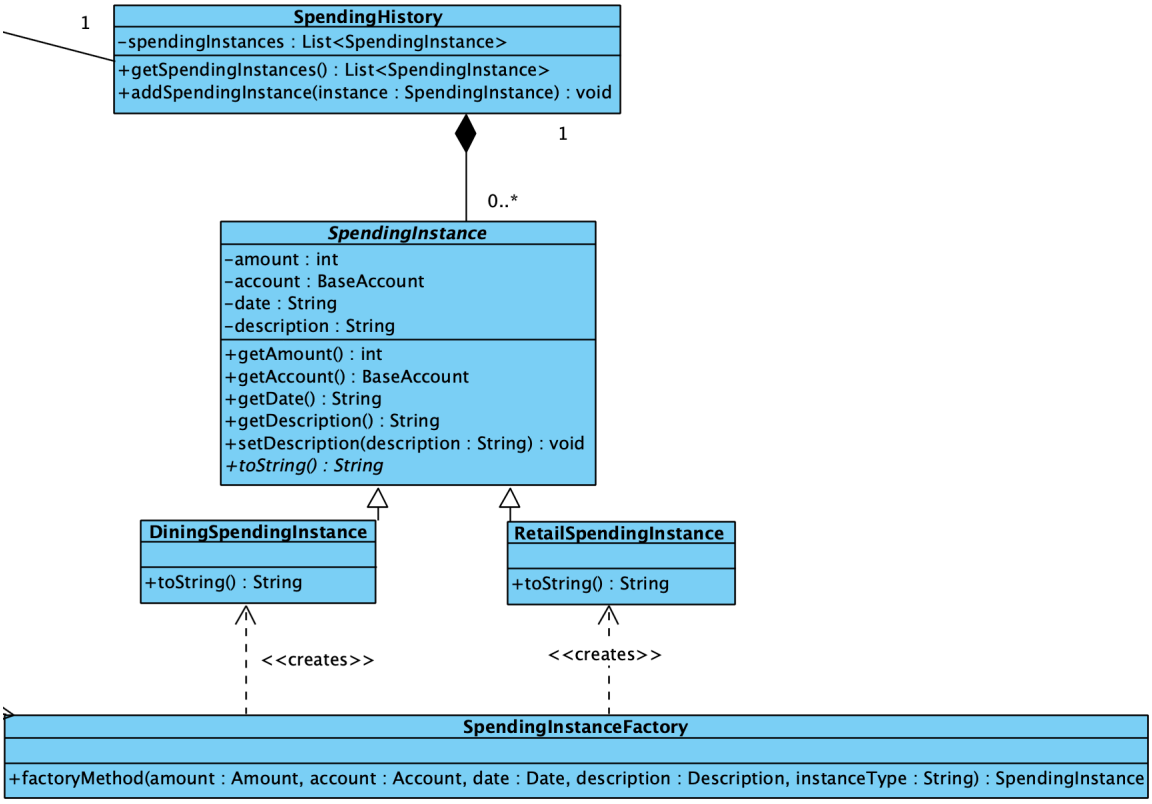

Figure 8: *Simple factory design pattern used for spending instance creation.*

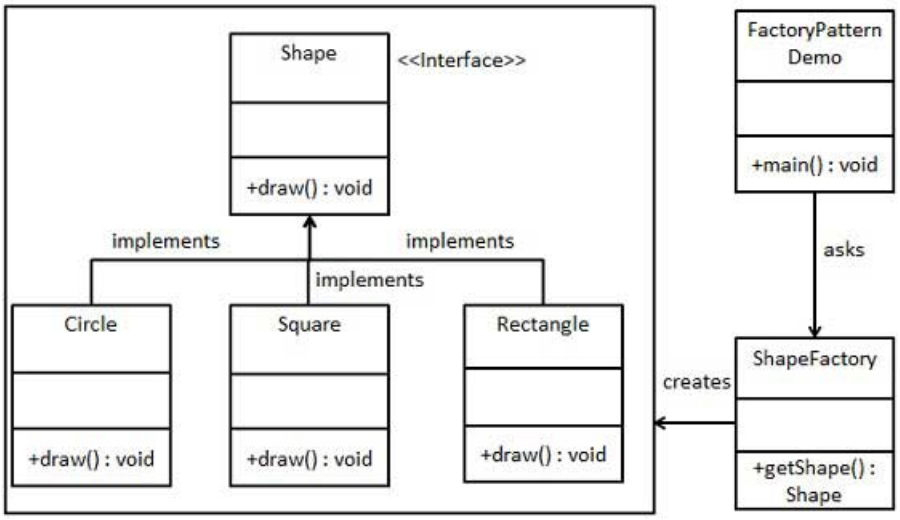

Figure 9: *Example factory design pattern, from https://www.tutorialspoint.com/design_pattern/factory_pattern.htm.*

I implemented this design pattern with an abstract class `SpendingInstance`, with the two concrete classes for dining and retail inheriting from this class and implementing their own `toString` methods. The `SpendingHistory` object contains an arbitrary number of spending instances.

As mentioned above, this is an implementation of the simple factory method design pattern discussed in class. I do not count this as a real design pattern, but I do have the three required design patterns listed above. This variation of factory forgoes an abstract creator in favor of a single concrete creator class with a static factory method. In my case, the factory method contains the logic to instantiate either of the two spending instance types. In this situation, a full factory method or abstract factory was unnecessary due to the relative simplicity of the task; at the same time, the factory moves logic from the `SpendingController` class and greatly simplifies creation structure.

Having multiple different types of spending allows me to have different `toString` behavior, and also gives the opportunity for additional functionality. For example, dining instances automatically add a tip to the amount.

What have you learned about the process of analysis and design:

This project has taught me a number of things about analysis and design in the object oriented space:

1. The value of deeply considering design before writing code is very underrated. As we've learned in the class, there are a great number of problems that have been solved before; thus, taking a step back and considering how existing solutions can be fit into your problem is extremely useful. This utility is twofold, as it allows you to think more deeply about your problem, affording more opportunity for refinement. It also allows you to consider the problems at hand from a different lens–that of fitting solutions to a problem instead of finding a solution to a problem.

2. Design patterns are not just raw recipes to be followed blindly. Just as many of the design patterns discussed in class have multiple correct ways to implement them, every design pattern can be adapted and morphed to fit into different spaces. I did this twice in this project–first, by combining template and strategy to minimize code reuse while enforcing an execution order and allow for runtime switching and second by fitting decorator into an ORM. Both of these situations still include the basic problem that these patterns exist to solve, but the patterns themselves are adaptable beyond exactly what is taught. This can also be observed by implementing design patterns in dynamic languages like Python (instead of Java), where adaptation is necessary to use mechanics like multiple inheritance in the absence of interfaces.

3. Finally, I've learned that design is a very iterative activity. You can't expect to always sit down and get it right the first time, there's definitely a period of experimentation necessary as well. What's more, analysis and design is practicable, and the more it's practiced the quicker the iteration will become.