Java Finance Tracker
David Chaparro and Julia Kent

Work Done: Most of the work done has been on the backend, as this area had the most pattern usage that would be relevant to the class. Our work has been on creating facades for the UI and the DataFacade to interface with each other. Since a majority of the commands will originate from the UI, the mediator which passes messages between the two sides of the application will not have control of the messages sent. However, the mediator is also acting as an observer, notifying when any changes to the data are made. When a change is made, it will send a command to the UI interface to update the current values. In this business logic, we can see mediator, facade, and observer, which handle most of the program's interactions. One improvement could be to separate the observer and mediator pattern, but that would be only to increase cohesion as the two responsibilities don't depend on each other very much. David was responsible for the UI, Observer, Facades, . Julia was responsible for the Builder Pattern and database management (CSV files and the Data Mapper pattern).

Challenges: Some challenges we have had have been focused on the UI. JavaFX has an excellent tool called javaFX Scene Builder which automatically creates your user interfaces in a graphical approach. After that you add a controller in order to connect the business logic with the UI. While the design of the UI is near completion, connecting the UI logic with the backend logic has proven to be a challenge, as there are many bugs when connecting the two. In fact, for one group member we spent an entire day working on only the UI and not any backend. Because of this, for our checkpoint we decided to focus on the backend as it has a larger focus on the patterns in use. So far we have not had many major changes to our project, except minor edits from our original proposal.

Pattern Description:

Observer - The observer pattern is implemented by the mediator in order to keep a log of the requests given to the Data handler facade and requests given to the UI. This way we have a method of troubleshooting. Furthermore, if there is ever an update in the data, the observer will notice a change and tell the UI to update the current displayed values. Both the data facade and the UI facade will act as subjects for the observer, and the mediator/observer will record events from both objects.

Facade - The facade pattern is used to simplify connections to the UI and the Database. The dataFacade is responsible for connecting with and returning various datatypes associated with accounts and financial information. For instance, the facade will take a request to return us all accounts that are considered assets, or debts. This allows us to access all the data without going into our data structure to look for these objects ourselves. We will also have methods for getting unique dates for display, and getting unique categories for use in the UI. The UI Facade, while unfinished, will have methods that will allow the UI to update the information that is displayed. The data facade will largely be a way for the UI to have a reference to the database, because most commands will come from the UI itself.

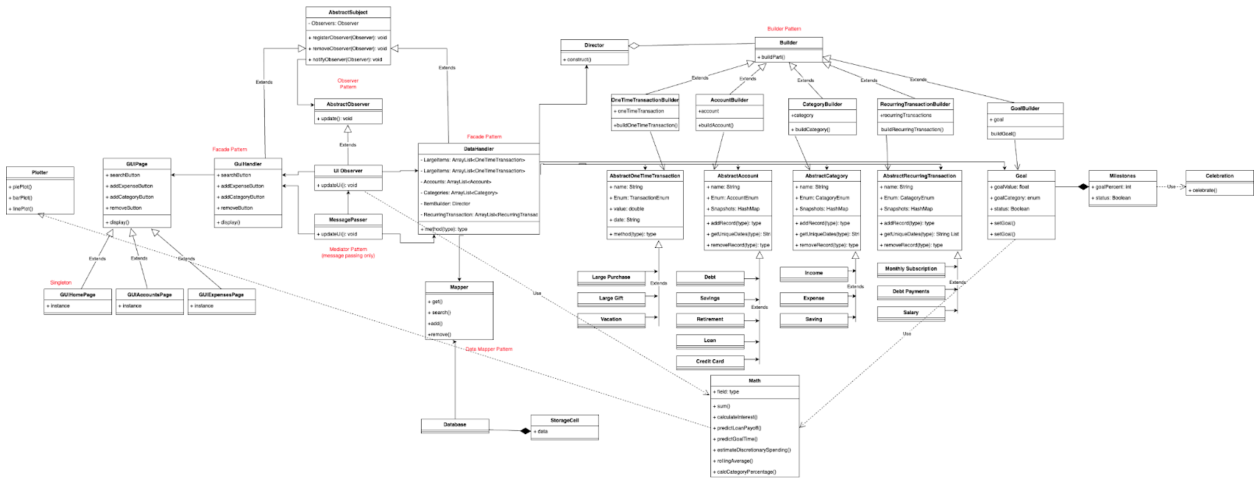
Mediator - We also have a mediator to act as a bridge between the two facades. The mediator is responsible for recording all of the messages passed between the two sides of the application, and proves both facades with references to the other facade so they can send messages across objects. There arent many explicit methods in the mediator, as adding a method for every single message would get complicated. By having the references to the two facades allows us to connect any part of the application to the mediator so they can have access to the data or UI.

Builder - The Builder Pattern was used for creating Transactions and Accounts. The more we worked with the Builder Pattern the more confident we were that this was the correct pattern for our finance application. The Builder pattern allows us to neatly have keyword arguments for the creation of Transactions and Accounts. So we have set attributes that every Account or Transaction need to have, and then optional attributes that can be set to not-null values if need be. Understanding this about the Builder pattern allowed us to greatly simplify our UML diagram. Before, we had two Builders extending an Abstract Builder class, this was not necessary. We also had several subclasses of an Abstract account class, when their differences can easily be handled by an Enum. Similarly, we originally had Transactions and OneTimeTransactions (different classes for whether the transaction is repeated monthly, annually etc) with the only difference being if there is a transaction frequency. The Builder pattern handled this nicely. This led us to see that Goals, however, were not a candidate that makes the most of the Builder pattern.

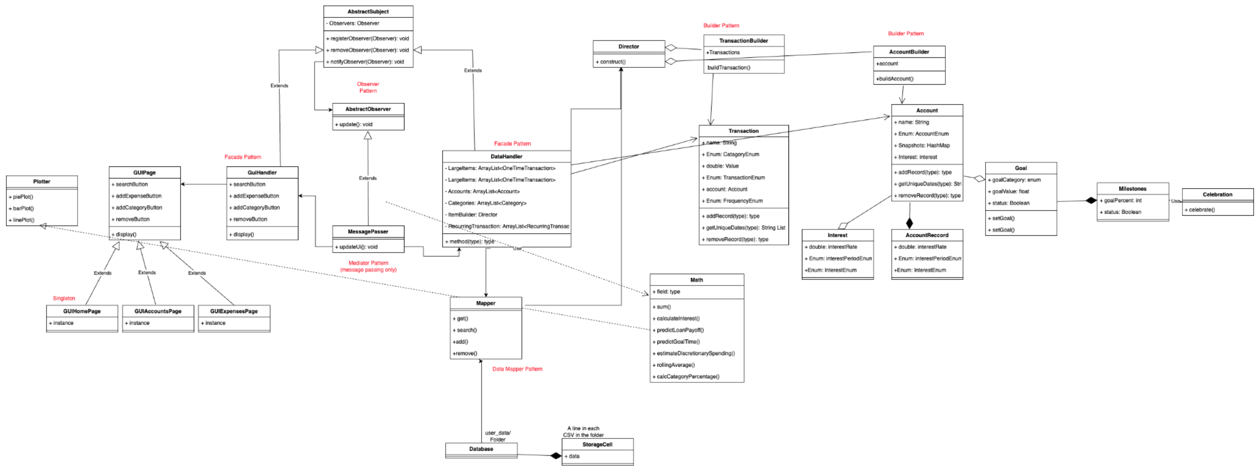
One challenge with the Builder Pattern is in the Accounts, we have optional attributes of interest rate, interest period, and interest type (to differentiate APR, compounded daily, etc). Currently all 3 of these attributes are optional, but we would like to make the code more robust by making these attributes a set - so either they are all null or none of them are. One option to do this is to add an Interest Class that must have these attributes, and then the Account can either have an interest attribute or not. This idea has not been fully implemented yet.

Data Mapper - The Data Mapper pattern is used to have a separate class that knows how to open, read/write, search, add/delete information from the database. This separates the responsibility of objects so that Accounts don't have to both know their information, and how to grab their information at start up from the CSV file, for instance. In our case, our database is a collection of CSV files. The Mapper class has all the methods for interacting with CSVs and the enforced formats of the transactions vs an account file. The greatest challenge with this pattern is the sheer number of options for a database that we had to pick from, before deciding that CSVs were the simplest viable solution for our application - and then having to pick from a plethora of CSV java packages. We don't have a sense for which packages in Java are the best, yet, so it is possible that we might yet again switch.

Original UML:



Updated UML:



Work to Do:

- Work on the MathUtil class
- Work out intricacies of the UI talking to the classes
- *nice to have* Account Goals
- *nice to have* user-input custom Transaction or Account category Enums