Object Oriented Analysis and Design
Project 7 - Semester Project

Financial Tracker

A UI Tool to Manage and Display FInances
by David Chaparro and Julia Kent
Github Repositor: https://github.com/Chaps42/FinanceTrackerJava
Video link:
https://drive.google.com/file/d/18hG5BJmY6RLjHtQV4bEwTw_XCeYgscTp/view?usp=sharing

Final State of System Statement:

The application allows users to create accounts and transactions either via the terminal interface or by loading in a CSV file. The system then is able to calculate and apply interest rates, have transactions occur at a given frequency, have transactions automatically add or deduct from accounts, and calculate spending by category (amount and percentage). The application creates line plots of account information over time and pie plots of relative category spending. Every one of these commands is user prompted in the terminal. Some features originally planned were not implemented: Account goals and celebrations and allowing the user to custom create transaction categories. These features were considered "nice to haves" and put on the back burner until the core functionality was in place. We simply did not have enough time to come back around and implement them. We originally planned on having a JavaFX user interface, but since this was our first attempt at a GUI it proved too difficult within the allotted time frame, and we pivoted to a terminal user-interface. This seemed like a good idea because the core learning objective of this course is on design planning, object oriented patterns, and how classes interact with each other - so it was best not to eat up more time on the front end development. Other features that we would ideally have implemented in a marketable financial tracker include: comparing spending by month and loan pay off predictions, etc. Again, while we would like to have developed these, this development would not have changed the fundamental architecture of the system, and thus wouldnt change the patterns present and would not have been as intellectually invigorating. Furthemore, since the UI is essentially many different commands and UI displays, we could have spent significantly more time creating additional UI subclasses or Command subclasses that make our functionality fancy. Unfortunately, this would only be creating more subclasses instead of contributing to the overall design of the program, so we decided to pursue understanding the patterns first. In a fit-to-market application we would spend a lot more time looking for edge cases and making the system robust. There aren't protections in place to stop the user from behaving in unintended ways, such as the user listing an Account name for a Transaction that doesn't yet exist (or create a pop up prompting the user to create that Account with the new name). We would also like to see more automation in a marketable product. Ideally, a financial tracker application like this would automatically load existing files, apply interest or recurring transaction maths, and produce the plots in the GUI right away and again upon refresh. Still, we feel confident that this project is a strong demonstration of design process planning and patterns.

Final Class Diagram Comparison Statement:

There are many changes to our UML class diagram from our initial planning. During the planning phase, our understanding of the Builder Pattern was weak, so we had an abstract Builder class that our Transaction and Account Builders inherited from (though now it is clear that they share no methods or attributes, and only have their pattern in common).

We also simplified our objects a lot: there is no longer a `Category` class (this is handled by the Transaction's Enums), there are no longer separate `OneTimeTransaction` and `RecurringTransaction` classes (this is also handled by an Enum within Transaction), and there are no longer a `Goal` or `Celebration` classes (these were deemed a "nice to have" quality of the application that did not make it into our final project).

On the other hand, other classes were more complicated (or more classes) than anticipated. The `Math` class, for instance, is now a series of classes (`CategoryMath`, `DateMath`, InterestMath`, 'RecurringTransactionMath`, and `TransactionAccountMath`) because each one was large enough that it was worth making the code more navigate-able by separating them into their respective applications. Similarly the `Plotter` class became `LinePlotter` and `PiePlotter` because both classes use different libraries and imports. Both plotters have only one method call at this project iteration, but could easily be extended for more functionality (say comparing Account interest vs principal balance on a pie plot, or plotting category spending over time on a line plot). In both cases, the relationships shown on the UML are fairly unchanged.

Not listed, but every Enum is its own class in its own file. We tried to create helper files for Accounts and Transactions that contained all relevant Enums, but we couldn't get those Enums to be public outside of their package if not in their own file. So we apologize for the many small files, but hope that the package structure of our application helps with understanding.

The UI side of the class diagram changed dramatically. Original in JavaFX, the program would load an fxml file for the elements, and have an associated controller. The controller would abstract away the strategy and command patterns into the built in java library. But, since we could no longer do that, we had to create our own UI framework, which included an “await command” method in the abstract GeneralList class. This was the largest change to our class diagram, and it helped introduce more patterns and depth into the application.

For the central handling of the application, the mediator was used as a way to pass messages between the UI and Data Facade. It was also responsible for directing the flow of the program. In the run() function, this is where the mediator would direct the UI to prompt the user for input. From there, the commands would pass messages through the mediator and to the Data facade. The mediator also acted as an observer to the subjects in the rest of the program. As the observer, we would notify the observer when there were changes in the system, so we could use it for debugging. Initially, we wanted to create a UI updating command, but that was scrapped for a simpler system. This changed slightly from the original class diagrams, as the mediator became an observer that would watch the rest of the system. Again, we originally had plans for when the mediator was notified of an update to the data, it would update the UI. Since we found a simpler solution, we went with the simpler version, but could implement the observer function in that way.

The Command patterns as also used as a way to create ways for the user to control the program. All Command subclasses had an execute() method in the subclasses. The commands would be on the buttons if we used a JavaFX UI, but since we did not we had to manually create our commands. We design the commands to be used polymorphically, and assigned dynamically, but since time was short we again decided to focus on solidifying the class structure rather than adding fancy edge cases.

OO Design Patterns that were used are:

- **Singleton**, for the `Database`, the `Mapper`, and the `FileManager`.
  All are lazy Singleton (waits till a thing is called and checks to see if it exists before returning itself). This pattern ensures that there is only one instance of these classes which ensures nice memory usage and less room for errors. This was particularly important for the `Database` (which stores all of the `Account` and `Transaction` data), and a convenience for the `Mapper` and `FileManager` (for which there is no need to have more than once since they store no attribute information).

- **Builder**, for `Accounts` and `Transactions`. The Builder Pattern is a good alternative to passing in lots of arguments to an object's constructor. Instead, we are able to sequentially call methods such as `addTransactionCategory()` in a way that passes in all of the same information to the final object, but makes it much more clear what each argument expects as input. This was very appropriate for the `Account` class which has attributes: name, value, refords, interest rate information (rate, frequency, type, date last calculated). Similarly the `Transaction` class has a lot of Enums referring to if it recurs, if so, how often, and what category it is (as well as the typical name, date, amount, and account it comes out of or into).

- **Data Mapper**, for interacting with the `Database`. Typical Data Mappers have functions for getting, searching, adding, and removing information from the Database. Our `Mapper` class has methods that map to these purposes for Transactions and for Accounts. There is search capability for Accounts or Transactions by name, by category, by type, and by one-time/recurring. No other object interacts with the `Database` object, not even the `FileManger` (which too goes through the `Mapper`).

- **Observer** 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. When other commands are called, we can also notify the observer for further troubleshooting.

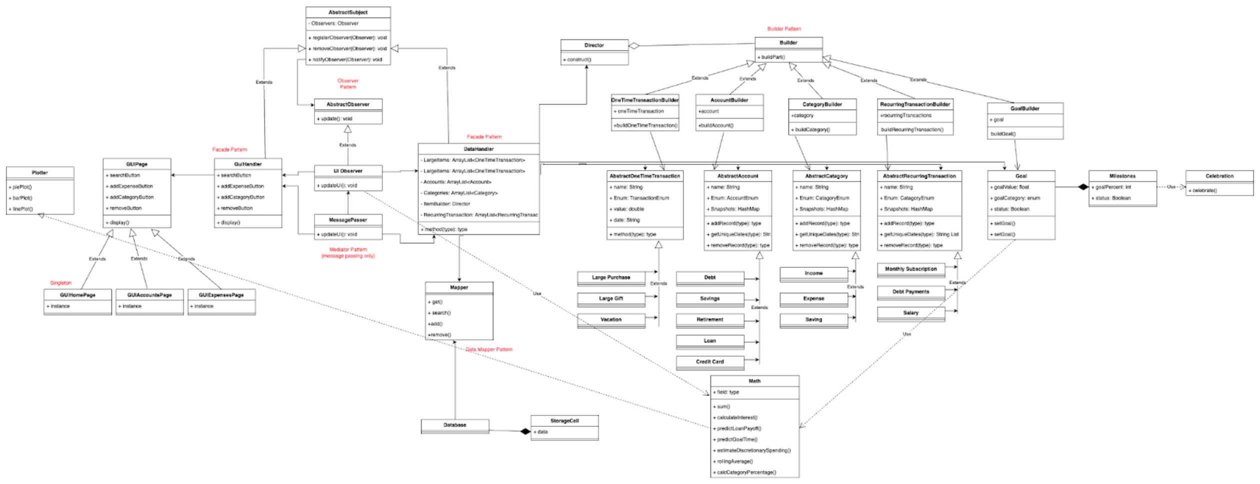
- **Facade**, for simplifying connections to the UI and the Database. The dataFacade is responsible for connecting with and returning various datatypes associated with accounts and financial information. For instance, the facade will take a request to return us all accounts that are considered assets, 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 has methods that will allow the UI to update the information that is displayed. The data facade is largely a way for the UI to have a reference to the database, because most commands will come from the UI itself.

- **Mediator** acts 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 aren't many explicit methods in the mediator, as adding a method for every single message would get complicated. 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.

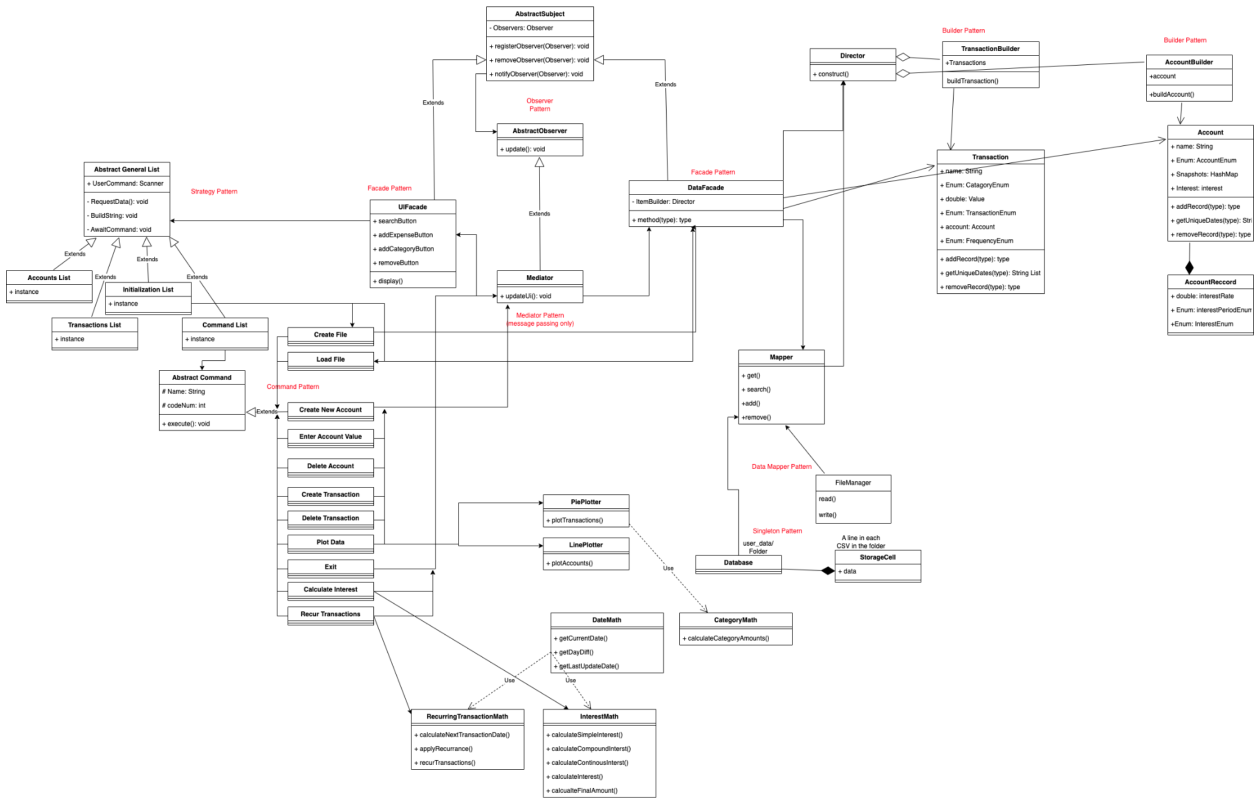
- **Command** acts as a way for the UI to interact with the database. The abstract command object acts as a template for all other commands, and each subclassed command contains the same execute function. The abstract command contains a reference to the Mediator, which in turn gives each subclassed command a reference to the data facade and the UI Facade, allowing for commands to control the program. The commands do not take the role of the mediator's job, and they frequently call getData() so the mediator can return the facade.

- **Strategy** acts as a way to specify the algorithms used when displaying information for the user. The abstract object GeneralList allows the creation of subclasses which treat the abstract commands differently. Each subclass such as AccountList and Transaction List fill out their own buildString method which allows them to create their UI. Each buildString method is different, but each UIClass is accessed by the UI facade to be controlled. In the UI Facades “updateUI()” method, the UI elements are iterated over polymorphically, thus the algorithm is changed just like the strategy pattern.

Original UML:



Final UML:



Third Party Code vs Original Code Statement:

We used 3rd party libraries for a few methods within the application. Line plots are done with XChart, Pie plots are done with JFreeChart, and reading and writing to CSV files uses OpenCSV. Some snippets of code are made while referring to various tutorials or package documentation. No code is 1-to-1 copied from third parties. Here are our citations (also noted in code comments throughout):

- Knowm "XChart Example Code" (https://knowm.org/open-source/xchart/xchart-example-code/).

- JFreeChart from Tutorials Point "JFreeChart Pie Chart" (https://www.tutorialspoint.com/jfreechart/jfreechart_pie_chart.htm).
- Baeldung "Open CSV" (https://www.baeldung.com/opencsv).
- Java T Point "Get Current Date and Time in Java" (https://www.javatpoint.com/java-get-current-date).
- StackAbuse "How to Get the Number of Days Between Dates in Java" (https://stackabuse.com/how-to-get-the-number-of-days-between-dates-in-java/).
- How to get the current working directory in Java, Mkyong (https://mkyong.com/java/how-to-get-the-current-working-directory-in-java/#:~:text=In%20Java%2C%20we%20can%20use,where%20your%20program%20was%20launched.)
- https://docs.oracle.com/javase/1.5.0/docs/api/java/util/Formatter.html#syntax
- https://www.geeksforgeeks.org/why-is-scanner-skipping-nextline-after-use-of-other-next-functions/

Statement on the OOAD Process:

List three key design process elements or issues (positive or negative) that your team experienced in your analysis and design of the OO semester project

One of the most important design process elements was the requirements analysis. Without the requirements analysis, we were unable to start writing any meaningful code. By specifying the classes beforehand and by specifying how each object interacts and with what pattern, you are easily able to code a workable program without much design work to do while coding the actual program. This vastly simplified our development process. However, one of the challenges was setting the design requirements. Translating design requirements into software methods, classes, and objects took a lot of work, and only when we started developing did we run into the problems with our first design. Creating good requirements is a must, especially when working on a team. The more specific you can make your requirements the better. Thankfully, as we came up with the requirements, we were able to be fairly flexible, perhaps in industry it may be more difficult.

The use of the Class diagram was instrumental in developing the OO program. By creating a class diagram, it was easy for our group members to communicate ideas with each other, and minimize the amount of words spent when developing. This hastened the development process and made everything more clear. Additionally, whenever we got stuck in the weeds of development, the class diagram became a roadmap, to remind ourselves of what we were building and how it played into the whole of the program. This roadmap was especially useful when changing the objects in the software, as it allowed us to easily see the changes graphically instead of abstractly. Again, the class diagram is a necessity and life saver when designing software. While our UML diagram changed a lot as our understanding of the system deepended, making the UML diagram was still a vital first step.

The architecture diagram, however, was not very useful for our understanding or planning of this specific application since it was so straightforward.

We tried to make sure we coded to an interface, not to an implementation. By using abstract classes for the command or for the UI elements, we tried to abstract away as many of the variations as possible. By only interacting with an object polymorphically, we were able to code to that interface as much as possible, reducing the need for coding to specific implementations of classes.