Pooled Investment Portfolio Management Simulation Design Document - Phase 2

Project Repository

<https://github.com/CSC207-UofT/course-project-codemonkeys>

Contributors

Langson Zhang <https://github.com/langsonzhang>
Tammy Yujin Liu <https://github.com/tammyliuu>
Jiamu Sun <https://github.com/JackSunjm>
Andrew Hanzhuo Zhang <https://github.com/a663E-36z1120>
Peng Du <https://github.com/Vim0315>
Raymond Zhang <https://github.com/RaymondZhang24>
Edward Li <https://github.com/Edward11235>
Zixin (Charlie) Guo <https://github.com/charlieguo2021>

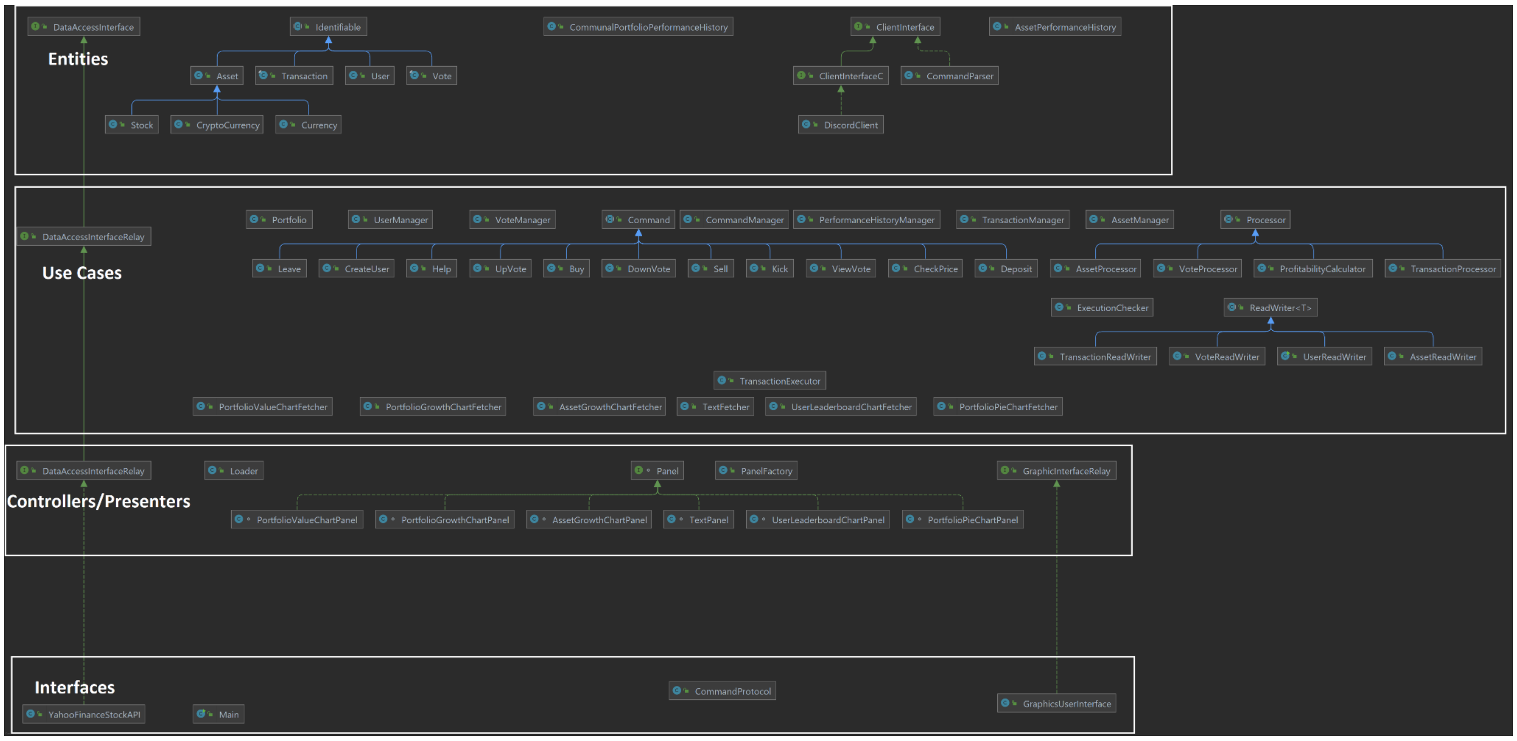
CRC Representation

< CRC Cards.pptx >

1. Updated Specification:

The purpose of the project is to develop a management tool for a simulated pooled investment portfolio, where multiple parties pool their funds together to form a single investment portfolio in the financial market. The program should enable all invested parties to democratically make investment decisions for the portfolio through calling and casting votes, given that the voting power of each party varies according to the profitability of their past votes. Users use the command line to create votes, downvotes, upvotes and check existing votes. A GUI featuring a suite of graphs will inform the users of the performance of the portfolio. There are also commands for Admins users to ban/promote regular users and override votes. Once a vote is approved, the corresponding transaction happens behind the scenes. Users interact with the system through a discord server, which acts as a multi-user command-line environment orchestrated by a discord bot.

2. Project Structure (classes organized by layer) :



3. Major Design Decisions

a. Facade Design Pattern Applied To Portfolio

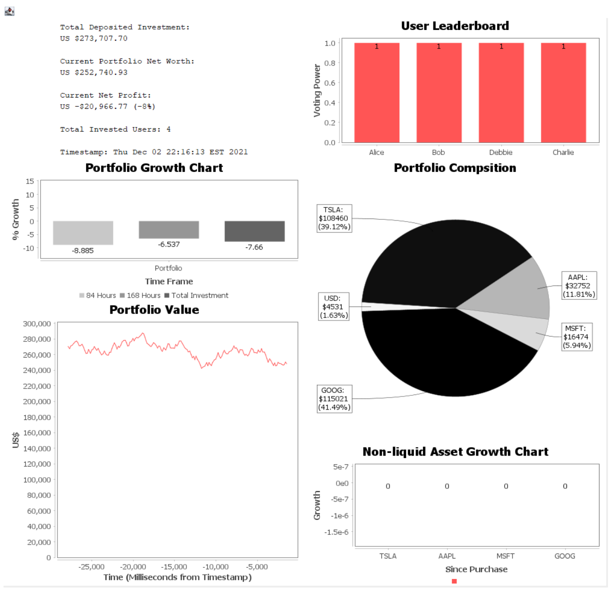
Because our portfolio interacts with too many classes, we decided to apply the facade design pattern to hide its complexity. Specifically, we have created different processors to segregate other classes from the portfolio class, as well as to instantiate them as the front-facing interfaces masking the complicated underlying code and logics.

b. Graphic Interface To Reflect Group Status

To provide users with a direct insight of the group status, we have implemented a graphic interface to reflect most of the group information, which include:

1. **Portfolio Composition**, using a pie chart to present the assets in the group portfolio.

2. **User Leaderboard**, which displays the users with the most voting power.
3. **Non-Liquid Asset Growth Chart**, which reflects the growth of Non-Liquid Asset invested by the group.
4. **Curve Chart of Portfolio Total Value**, which reflects the changes of the total value of the group portfolio over time.
5. **Bar Chart of Portfolio Growth**, which reflects the growth of the group portfolio in different time frames.



```
Total Deposited Investment:
US $273,707.70

Current Portfolio Net Worth:
US $252,740.93

Current Net Profit:
US -$20,966.77 (-8%)

Total Invested Users: 4

Timestamp: Thu Dec 02 22:16:13 EST 2021
```

c. Applied Serialization To Set Up The Local Server

Since we do not have an online database yet, we need a local server in order to keep the system running. To accomplish this, we have applied serialization to our system, so that we can store and read information at a local directory. In this way, we are able to run the system on a computer and save/retrieve data through a local database.

d. The Algorithm of VotingPower:

We have decided the VotingPower to be calculated as:

- When the transaction is **profitable**, if the user is the INITIATOR of the transaction, his/her voting power will increase by *40%*, if the user is a upVoter, his/her voting power will increase by *20%*, if the user is a downVoter, his/her voting power will decrease by *10%.*
- When the transaction is **not profitable**, if the user is the INITIATOR of the transaction, his/her voting power will decrease by *20%*, if the user is a upVoter, his/her voting power will decrease by *10%*, if the user is a downVoter, his/her voting power will increase by *20%*

e. Implementation of Abstract Identifiable Class

We have implemented an identifiable class in entity, which provides the ability for implemented instances to be uniquely identified and compared using UUID. Compared to random strings/indices, using UUID is a safer way to identify instances as it effectively avoids overlapping. This also provides a new way of comparing instances, since two instances are equal if and only if they occupy the same UUID.

f. Implementation of ClientInterface

In phase 1, our Commands had no way of returning information to the client other than to hardcode it. Thus, we have implemented ClientInterface as an extra layer between our UI (discord bot) and actual business logic. In this way, Commands can complete tasks and return meaningful information without worrying about where to output the information.

g. Refactoring back to By-Layer Packaging Strategy

We decided to refactor our packaging strategy back to the by-layer strategy from the by-component strategy in phase 1. This decision was made to make spotting clean architecture violations from import statements easier. All clean architecture violations are thus subsequently resolved after this refactoring.

4. Design

a. SOLID Principle Adherence and Design Decisions

i. Single Responsibility Principle

Our project design strictly adheres to the single responsibility principle. We have debated as a group how broadly we should consider a collection of multiple tasks to be a ‘single responsibility’. We eventually decided that the optimal definition of ‘single responsibility’ was that every method of a class must perform a function only necessary to fulfill exactly what the name of the class describes, and that each class should only be responsible for one actor and one actor only. This definition is exactly what we adhered to when we designed every single one of our classes, where we encountered classes that don’t satisfy this definition from phase 1, we broke them up into facades.

ii. Open Closed Principle

We had made extensive efforts to ensure that our project designs adhere to the open-closed principle through making our designs as modular as possible. One example of this is how we designed our GUI through using the factory patterns, where more ChartPanels can be added in the future to our graph suites if needed by making simple modifications of the PanelFactory class and adding a new ChartPanel module to the GraphicsPresenter module as needed. Another example of a class that demonstrates the open-closed principle is our ClientInterface class. By simply requiring the implementation of the input() and output() methods, we can effectively abstract away any concrete implementation of our gateways. This way, future developers can implement their own custom clients without any compatibility issues.

iii. Liskov Substitution Principle

We had ensured that all of our inheritance hierarchies adhere to the Liskov substitution principle, where we made sure that every subclass is able to be used interchangeably with every other subclass of the same super class. Particular examples of this is how we managed the inheritance hierarchy of our different types of assets and different types of portfolio in our code. Furthermore, our many implementations of ClientInterface and DataAccessInterface serve as modular classes that can be switched in and out of use without affecting the operation of the program.

iv. Interface Segregation Principle

We made sure that every interface implementation satisfies the interface segregation principle, where we had bigger interfaces that resulted in redundant method implementation, we either modified the scope of the interface or split it up into multiple more specific smaller interfaces. Interface classes in our project include ClientInterface, DataAccessInterface and to a certain extent, Command and Identifiable as well. Furthermore, we also leveraged this principle extensively to resolve clean architecture violations by implementing segregated interfaces to invert dependencies and relay dependencies across layers.

v. Dependency Inversion Principle

The dependency inversion principle was applied to key components of our design by introducing abstract interfaces to reduce coupling, such that interface level modules do not depend on asset level modules and also vice versa. We have utilized the dependency inversion principle primarily to update asset prices, which is at the entity level of our code, using the DataAccessInterface to fetch real-time stock prices from the YahooFinance input interface, which exists at the interface level.

b. Clean Architecture Adherence

We designed our program to strictly follow the layers of clean architecture. For example, when we execute a Command and want to return something to the user, we do so by using ClientInterface as a dependency inversion for our communication gateway between the Discord Bot and our server. Elsewhere, we implemented our Classes to strictly follow the Single Responsibility Principle so that we can easily navigate and visualize the flow of our program from the client's computer, through the different layers of clean architecture and all the way to our Entities where actual business objects are mutated. Furthermore, our adherence to clean architecture can be demonstrated easily by the packaging strategy we implemented.

c. Packaging Strategies

We decided to utilize the "by-layer" packaging strategy where we decided to package our system according to each of the layers as specified by the clean architecture, namely, "Entities", "UseCases", "Controllers/Presenters", "Interfaces". This structure makes it clear modules from which layer each of our classes depend on explicitly in our import statements during our development process, such that we could ensure from the very start that low-level classes will never depend on high-level modules and dependency does not "cross layers" where they are not permitted by the clean architecture. This reduced coupling in our code to a minimum. The "by-layer" packaging strategy also made serializing our core business data very simple, as all the core business data is stored in the "Entities" package. Within each layer we also packaged classes that act together to perform a similar function together into sub-categories to ensure that our project structure is more readable. Overall, the packaging strategy of our group has saved us crucial development time.

d. Major Design Pattern Usage

i. Facade

The facade design pattern is implemented to hide the complexity of the portfolio class. Different Processor classes adhering to the single responsibility principle are responsible for carrying out each portfolio related logic behind the portfolio facade.

ii. Factory

The factory design pattern was extensively implemented in various modules in our system to abstract away the process of directly creating related classes. In the GraphicsPresenter package in our controller layer, we implemented a PanelFactory to generate JPanel's to be displayed on our JFrame-based GUI. The factory design pattern was similarly used in how all of our entity manager classes in our use case level manages our entities, such as how asset, transaction, and vote managers each correspondingly instantiate different asset, transaction, and vote classes.

iii. Command

The command design pattern was applied extensively in our project for relaying commands from our CLI to orchestrate behaviours in the entity and use case layers. Our group has spontaneously implemented this design pattern before even learning about it formally, where our commandParser class takes a CLI input as a string and returns a newly instantiated command object representing that CLI input, while the commandParser class acts as the invoker that calls the execute() method of each command when they need to be executed.

iv. Singleton

The singleton design pattern was used extensively in our system. Examples include classes in the PerformanceHistories package in the entity layer and the various classes in the Managers package in the use case layer. We implemented this pattern as such for classes that act as one single definitive source of information or logic in our system, where we want to ensure that the information that every part of the system receives and the behaviour of our system is synchronized across board.

v. Dependency Injection

We implemented dependency injection patterns wherever we performed dependency inversion. Since our system strictly adheres to the Liskov substitution principle, this greatly improved upon the modularity and flexibility of our system, as different unique subclasses can be instantiated and injected into the various components to induce varying the exact desirable behaviours depending on different contexts. A particular example of this is our DataAccessInterface, where new clients of the system can choose to get their real-time price data from some other source than YahooFinance and inject their own DataAccessInterface classes into the system.

e. Accessibility Considerations

We have designed our GUI with accessibility in mind to accommodate people with all types of color blindness by using a grayscale color scheme on our bar charts and pie charts where multiple bins/chart sectors need to be distinguished from one another.

5. Data persistence

We have implemented a readwriter interface and its 4 subclasses(ReadWriters for assets, transactions, users and votes) to maintain data persistence. Each of the ReadWriters have two methods: saveToFile and readFromFile; saveToFile is used to serialize data and store it at a local directory, while readFromFile is used to retrieve data from the local file. This ensures that our program data can be saved at a local server, which means restarting the program wouldn't result in the loss of data. In addition, our database does not depend on discord channels, which means switching channels also wouldn't lose any data.

6. Test Coverage:

Entity

Assets:

- In the Asset and Stock, all the methods are tested in general cases. Since the complexities of methods are pretty low, general cases should be sufficient to test the correctness.
- In CryptoCurrency, Currency and dataAccessInterface, there are no new methods created, so the tests are not needed.

User:

- User class is fully tested in both general and special cases. Specifically, there is an UserTest that tests the general methods stored in the user, and an AdminTest that tests the authority system of users.

Containers:

- Portfolio and Transaction are tested in general cases.
- No tests needed for Vote since it has no methods other than constructor.

Use Cases

Commands:

- The CommandManager is tested in general cases, while UpVote/DownVote are fully tested in both general and special cases (for instance, when the initiator is banned).
- The AssetReadWriter, UserReadWriter, TransactionReadWriter and VoteReadWriter are tested in general cases.

Managers:

- The VoteManager, UserManager, TransactionManger, AssetManager and TransactionExecutor are tested in general cases.

Interfaces

- The CommandParser is partially tested, which is to create multiple users simultaneously.

Test Coverage Conclusion:

In terms of the test coverage, most of our tests for the backend only cover the base cases; this is sufficient since the correctness of our program relies on the modularity of each class rather than the coupling of multiple classes. Furthermore, we are confident in the testing done so far on the backend, and so we are devoting more attention to ensuring that the frontend runs smoothly for a perfect user experience. We have done dozens of tests and demos on the CommandLine and Discord Bot fixed many issues. Currently, our program works just as our specification described.

7. Progress Report **(Since Phase 1)**

- **Andrew Hanzhuo Zhang:**
  1. Work summary since Phase 1:
     - Designed and implemented the entirety of our system's GUI and data visualization layer.
     - Re-introduced the DataAccessInterface class into the system to gain access to real-time financial asset price data after several major refactorings since phase 1.
     - Performed code review and involved in major design decisions in later refactorings and redesigns to enforce clean architecture adherence.
     - Drafting and review of the design document.

  2. A significant pull request:
     - Link: https://github.com/CSC207-UofT/course-project-codemonkeys/pull/53
     - Description: Re-implemented DataAccessInterface and re-integrated its dependency inversion logic with the new, reconsolidated version of our system in phase 1.
     - Link: https://github.com/CSC207-UofT/course-project-codemonkeys/pull/82
     - Description: Implemented all components of the graphics interface using JFreeChart and JFrame, which included a suite of graphs for data visualization on the performance of the communal portfolio. Later refactored to resolve clean architecture violations.

- Peng(Vim) Du:
  1. Work summary since Phase 1:
     - Completed Accessibility Report and Design Document.
     - Constantly polished entity classes to improve program rationality.
     - Wrote tests for most of the entities.
     - Review pull requests to assure the quality of the main branch.
     - Wrote tests for ReadWriters.
     - Fixed clean architecture violations.
  2. A significant pull request:
     - Link: https://github.com/CSC207-UofT/course-project-codemonkeys/pull/38
     - Description:

       Our code in phase1 was messy and required lots of changes, and the branching was also very disordered. In this pull request, all the entities are completed and tested, and the program files are reorganized so that the autograding is passed. This is a great start of phase2 because it not only makes the upper layers able to refer to the entities, but also ensures the main branch to always be valid after that point.

- Zixin(Charlie) Guo:
    1. Work summary since Phase 1:
        a. Immediately following the phase one presentation, I engaged my teammates to discuss parts of the project that we can work on. I proposed and clarified the internal logic of managing the flow of assets, such as the transaction flow, so that the members who implemented the business rule layers could have a clearer understanding of the purpose of these layers.
        b. In addition, after receiving feedback from phase one, I quickly reached out to my teammates and initiated a facade pattern for the portfolio class. As a result, I successfully segregated vote, asset, and transaction classes into different modules, thus enabling the program to achieve a facade pattern while keeping SRP.
        c. To have interactions with our small database (.ser file), I also made all the necessary classes serializable, and implemented a read writer interface and various concrete read writers that are able to retrieve and write the required classes to execute the program.
        d. Moreover, not only did I perform tests for entity and manager classes, but I also created the ability for manager classes to save and load their own data.
        e. Lastly, I contributed to the Java doc and helped polish various other classes.
    2. A significant pull request:
        - Link: https://github.com/CSC207-UofT/course-project-codemonkeys/pull/38
        - https://github.com/CSC207-UofT/course-project-codemonkeys/pull/78
        - Description: Throughout project phase 2, among various implementations that I have completed, I would like to present two major pull requests. After the phase 1 presentation, I immediately discussed with my teammates, and redesigned, and took over the first draft of the majority of entity classes. Our final work wouldn't have been successful if entity classes weren't implemented correctly and followed the clean architecture. The second significant pull request occurred near the end of our development. At the refactoring stage, I proposed a facade design pattern on the Portfolio class, to segregate transaction, vote, and assets from the portfolio class. The newly constructed portfolio class becomes a facade that strictly follows the

SRP. Moreover, I have proposed and implemented serialization gateway classes that are able manipulate the data i.e. to read from and write into the database.

- Jiamu(Jack) Sun:
  1. Work summary since Phase 1:
     - Completed and improved UserManager, VoteManager, AssetManager
     - Wrote test for most of the Managers and the UpDownVoteTest, the UpDownVoteTest has a detailed scenario walk through about what will happen if you do an upvote or downvote, which ensures the main functionality of our program runs without error.
     - Participated in testing our program on discord and fixed bugs in our program.
  2. A significant pull request:
     - Link: https://github.com/CSC207-UofT/course-project-codemonkeys/pull/54
     - Description: Wrote the manager classes of our program based on the changes on Entity classes after phase1, applied **Singleton design pattern** among the manager classes, and java docs are added to let other team members understand the methods when they are trying to find the right method to use. Tests are added to make sure the code runs without error

- Yujin(Tammy) Liu:
  1. Work summary since Phase 1:
     - Complete Design Document
     - Finish half of the Command parts, mainly focus on the classes need to call VoteManager and UserManager
     - Complete the Execution_Checker
  2. A significant pull request:
     - Link: https://github.com/CSC207-UofT/course-project-codemonkeys/pull/63
     - Description: Finish Execution_Checker, which includes the voting power algorithm and some conditions to check whether a transaction is executable. This is a critical part of our project, we give them the same voting power at first and then increase/decrease different percentages when a transaction exists. Details are described in the **Major Design Decision** part.

- Raymond Zhang:
  1. Work summary since Phase 1:
     a. propose the
     b. Discussed and reflected the problem we had on Phase 1, and came up with our own standard proper procedure trying to solve those issues, such as make sure there is no direct modification to the main branch and peer review in our pull requisition.
     c. Completed TransactionManager and TransactionExecutor and tests as well
     d. Refactoring the code smells. Ensure that our code's documentation and naming convention strictly follow java standard.
     e. Prepared the presentation material
  2. A significant pull request:
     - Link:
       https://github.com/CSC207-UofT/course-project-codemonkeys/pull/56
       https://github.com/CSC207-UofT/course-project-codemonkeys/pull/64
     - Description:

       The TransactionManger class is responsible to store and access a list of pending transactions and TransactionExecutor will be used when a transaction is performed. The executor will take the two assets, by using assetmanager and usermanager, the assets will be modified and placed into the correct spots. This will help us enforce the single responsibility principle and make sure the code is in a clean architecture structure. In addition to those pull requests, I also refactored the program to match our package strategy to eliminate the potential code smell that might be contained in our project. Moreover, I was using refactoring to make sure the standard java documentations applied to our project.

- Langson Zhang:
  1. Work summary since Phase 1:
     - Contributed to design decisions.
     - Developed Identifiable, Command, CommandManager, and CommandProtocol classes along with a handful of Command subclasses
     - Implemented Dependency Inversion for ClientInterface class and Command Design pattern for Command class
     - Reviewed and edited the Design Document
     - Led an overhaul of our Design Specification
     - Reviewed and merged pull requests
  2. A significant pull request:

- Link: https://github.com/CSC207-UofT/course-project-codemonkeys/pull/74/files
- Description:

  Most of my work was done throughout many pull requests, but this one in particular had many significant changes to the structure of our Command system. First, I provided more methods in the CommandManger class for finding and generating commands. Then, I refactored the signature of the Command constructor to make the ordering of the parameters more natural. Finally, I developed the CommandProtocol class, which can be modified or extended in the future to accommodate new Command types. On a side note, I decided to use reflection in CommandManager because I believe the performance cost of using it is negligible for the scope and complexity of our project, and the convenience of using it is more attractive.

- Edward Li:
  1. Work summary since Phase 1:

     a. Learned how to create a Discord bot. Register CodeMonkey's monkey bot in Discord Developer Portal.

     b. Added DiscordClient class to communicate with the discord bot.

     c. Designed and wrote CommandParser class, which parses command coming from Discord

     d. Developed GraphicInterfaceRelay and fixed the issue that Commandparser in Controller folder implements GraphicUserInterface in Interface folder.

     e. Wrote a readme file.

  2. A significant pull request:
     - Link:

       https://github.com/CSC207-UofT/course-project-codemonkeys/pull/86/files

     - Description:

       I added CommandParser class, DiscordClient class, and a file to store our secret key of the discord bot. DiscordClient is the backend to the discord channel. When it receives a message from Discord channel, it automatically creates an event and will pass the event to the onGuildMessageReceived method in CommandParser object. Then CommandParser parses the command and executes the command.