Pooled Investment Portfolio Management Simulation Progress Report

Project Repository

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

CRC Representation

CRC Cards.pptx

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>

Project Summary:

1. **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,s and check existing votes. There are also commands for Admins to ban/promote regular users. Once a vote is approved, the corresponding transaction happens behind the scenes.

2. **CRC Model**:

The design of our CRC models followed the Clean Architecture. From the innermost Enterprise Business Rule layer, we have entity classes, including User, Admin(Child class of User), Transaction, Vote, Portfolio, and Asset. In the Application Business Rule, we designed use cases that manipulate the entities. Specifically, we have UserManager to delete, add, get and set users; VotingManager to add, delete and store votes; TransactionManager to delete, add, get, set, and store transactions, and PortfolioManager to update global portfolio, determine the net worth and store performance history. In addition, we have the DataAccessInterface to load data from API, and the Command interface to force the implementation of execute() in command classes. The command classes that implement Command include Help, Kick, Leave, CheckPrice, Upvote, Downvote, CreateUser, ViewVote, buy and sell. Moving one layer further, CommandParser, CommandExecuter, and GraphicPresenter are presented as controllers classes. CommandParser parses the input string and outputs a command that was defined in Use cases, and the CommandExecuter takes the command and executes it. GraphicsPresenter is used to visualize the current status of the portfolio. In the outermost layer interface, CLI is implemented to handle the inputs on the command line, GUI displays the graph generated by GraphicsPresenter and YahooFinance inputs real-time financial information into the system.

3. **Scenario Walk-through:**

   A typical scenario of our program would be: A new simulation starts with an admin user having already signed into the program. A newly registered user, Charlie, enters the simulation through the arrive command on CLI. Charlie is interested in purchasing Intel stocks. He uses a command to acquire a real-time share price of Intel through the CLI, which shows Intel is currently trading at $50/share. Charlie has put $1000 into the liquidity pool, but he would like to purchase 100 shares of Intel's stock at the current price. Charlie then decides to buy the asset with funds in the liquidity pool, which has $10,000. His decision to purchase initiates a vote. Because his current share of the fund ($1000) is less than the amount required for the purchase order ($5000) and that he has no prior performance record, his vote does not carry enough voting power to allow the admin, who also has voting power, to immediately approve the transaction. Instead, the transaction is sent to a vote for 24 hours to allow other users to have their say. Different users will carry different voting power based on the profitability of their prior transactions. After the voting concludes, the transaction is approved. The admin makes the purchase and the asset enters the portfolio. After a month, the admin realizes Charlie's decision to purchase Intel is not a good move and decides to use admin privilege to override the user group consensus and sell all of the shares. The asset is sold and money is added to the liquidity pool.

4. **Skeleton Program:**

Through a dedicated effort, our group was able to implement the majority of the design in accordance with Clean Architecture and the code is able to successfully compile and run. The code consists of four layers: entities, use cases, controllers, and interfaces, and the classes of each layer are grouped into the same folder. In addition to most of the functionalities we implemented in user manager and vote manager classes, we also focused on the CommandLine and its parser. Using the command line, all the actions in the scenario walkthrough can be accomplished. Beyond the scope of phase 0, we have solidified most of the classes and the program is already partially functional. However, some of the features are not yet implemented and future modifications are still required to follow the design patterns/principles and improve the program structure. The skeleton program includes three test classes, with each of them having one test case. More tests will certainly be added in the future, and ideally, every part of our program should be tested.

5. **Group Member Responsibilities**

Task	Member
CRC Entities	Langson
CRC Use Cases	Raymond
CRC Controllers	Raymond
CRC Interface	Edward
Project Specification	Andrew
Scenario Walk-Through	Andrew, Edward
Entities (skeleton)	Langson
Controllers (skeleton)	Tammy, Jack
Use Cases (skeleton)	Tammy, Jack, Charlie, Vim
Interface (skeleton)	Edward
Progress Report	Vim, Charlie

6. **Future Responsibilities**

Task	Member
Log-in System(Authentication)	Pending

Polishing the code (SRP and so on)	Pending
Implement GUI/Visualization	Pending
Plan for the front end	Pending
Auto-Backup feature	Pending(Andrew)
Discord input interface	Andrew
Implement more test coverages	Pending

7. **Things that have worked well with our design**

   1. It is easy to implement since the clean architecture makes the program structure clear and manageable.

   2. It is not necessary to modify our design structure after implementing the skeleton program because our design fits the specification and does not create any bugs in the walk-through scenario.

   3. It is possible to add new features to the existing code without modifying the current classes (OCP)

8. **Struggles:**

It was challenging to think of the use cases all at once.

Every group member has different points of view, which makes it hard to decide how some of the classes are implemented (e.g. whether Asset should be an abstract class).

It is hard to apply all principles learned in class immediately to the project. Time-consuming to compare, discuss and fix potential mistakes in the skeleton design.

9. **Questions and Concerns**

   1. Is it eligible to have the *performTransfer* method in the Vote class? If not, where should we put this method?

2. We have downscaled a lot from our original plan to reduce the program complexity. We are worried if even more downscales are needed in the future.

Things that have worked well:

1. Our design follows the principles of clean architecture well.
2. The deadlines are effective to put everyone into work.
3. The tasks are well-split so that everyone shares a portion of the work.
4. All group members are actively involved and communicate frequently with each other.