Pooled Investment Portfolio Management Simulation Design Document - Phase 1

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>

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.

2. **Project Structure**:

<Insert Project UML>

3. **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. This definition is exactly what we adhered to when we designed every single one of our classes.

        ii. **Open Closed Principle**

We had made extensive efforts to ensure that our project designs adhere to the open-close 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.

        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.

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.

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 and Packaging Strategies

Our design strictly adheres to the clean architecture, which is made apparent and further reinforced by our choice of 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", "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.

c. Design Pattern Usage

i. Decorator

We implemented the decorator design pattern in how we designed our user classes to manage how we manage user authorities, which made each type of user authority more concrete by representing them as decorator classes, which also improved the modularity of our code, where to introduce new user authorities, we simply need to create new classes representing new authorities.

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 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. Iterator

The iterator pattern was used in our entity layer, particularly to determine the overall portfolio profitability from individual asset price changes, accommodating for all different types of assets.

v. Observer (Potential Implementation)

The observer design pattern can potentially be implemented to keep track of the status of a transaction that is in the process of being voted for

approval. This can potentially make the dependency relationships with respect to votes more clear.

d. Accessibility Considerations

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

Clean Architecture Adherence and Packaging Strategies(Langson)

Functional

- Package by feature rather than layer

e. More packages, less classes per package
f. We value the efficiency and ease of access of classes the most, so we decided that this package structure was the best
g. Each package has its own role rather than a jumble of loosely related classes

Refactoring (Raymond)

One major refactoring that we did from phase 0 was our command line. Before, we parsed the input string by cases and then executed commands based on whether a string matched a command. We now refactored that layer of logic to parsing by iteration instead. We now iterate through a list of command templates, each of which can have its own unique way of identification, so we no longer have to write a specific case for each possible command.

Other refactorings included general code smells like bloaters and too many parameters for classes. For example, some classes had parameters for information that that class itself does not use, such as storing transactions in the User and not the TransactionManager.

Summary (Langson)

In summary, we believe that our progress for Phase 1 has satisfied our initial goals. Due to a group of 8 people, our project has a lot of features, and we believe that we were able to complete almost all of them. All of the group members contributed greatly to the success of the project, and we often had many reconsiderations on the design of the program, which shows the depth of involvement on the project for each member. Although our Phase 1 was very successful, we are confident that there is still much more to improve. For example, we had little prior experience working with git, and especially in a group of 8. In the future, we wish to further study the features of git and to use them to their full potential. In phase 2, we plan to execute the following :

Phase 2 Plan: (Raymond)

- Use git better (issues, merges, pull requests, feature branches)
- Communicate more effectively
- Make plans before writing code
- Write tests and documentation proactively
- New user interfaces (ex. Discord bot)

h. Division of Labour

Name	Github Name	Primary Responsibilities	Secondary Contributions
Andrew Hanzhuo Zhang	a663E-36 z1120	- YahooFinanceInterface - DataAccessInterface - GraphicsPresenter Package - GraphicsUserInterface	- Design document - Test cases - Designed main dependency inversion logic for real price data access from yahoo finance. - SOLID principle and clean architecture adherence

			assurance.
Peng(Vim) Du	Vim0315	- User - ConcreteUser - UserAuthorities - BanAuthority - ControlVoteAuthority - Transaction - Portfolio	- Part of the test cases for entities. - Application of Design pattern.
Zixin(Charlie) Guo	charliegu o2021	- Vote - Asset - AssetManager - Cryptocurrency - Stock - USD	- Part of the test cases for entities - Applied various design pattern in the code, such as decorator, iterator, factory, and observer pattern while following the CRC and specification - Make entity and use case classes serializable
Jiamu(Jack) Sun	JackSunj m	- PortfolioManager - VoteManager - UserManager	- Part of the test cases for use cases
Yujin(Tammy) Liu	tammyliuu	- Transfer - Viewvote - CreateUser - Downvote - Upvote - Deposit money	- Test case for usecases - VoteManager - Checkprice - Help to modify the structure of the code to fit into the SOLID principle and clean architecture adherence assurance. -
Raymond Zhang	Raymond Zhang24	- Completed TranscationManger - Completed TransactionExecutor	- Test Cases - Help to modify the structure of the code to fit into the

		-	SOLID principle and clean architecture adherence assurance.
Langson Zhang	langsonzhang	- Identifiable class - Command interface - CommandManager - Help, Kick, Leave, CheckPrice, Upvote Commands - AssetType	- Test cases - Improved readability of code - Refactored code smells
Edward Li	Edward11235	- CommandParse class - Command Line interface - CreateUser, CreateAdmin, ListUser, Kick, Leave Commands	- Added tests - Improved readability of the code

-