OOP Java project
Budmate:
Personal budget manager

alessandro.stefani10@studio.unibo.it
giulio.salotti@studio.unibo.it
paolo.pietrelli@studio.unibo.it
zhaohui.song@studio.unibo.it

August 17, 2022

Contents

Chapter 1

Analisi

1.1 Requisiti

In the year 2022 due to the federal reserve's decision, a lot of liquidity floods into the market, many people are lured in to make some bucks. But the market is like an ocean, teeming with volatilities. Though various platforms have emerged, we lacked a tool to manage every income and expenditure, our idea is to build a cloud-based platform that connects to various markets, provides analysis tools to clients to minimize the risk, and get the big picture of actual assets. . It's a tiny project of about 80 hours amount of work, hence it will only be a prototype.

Elementi positivi

- The software should be able to manage various assets of a client.
- management of one or more profiles including registration and switching between accounts.
- account management, piggy banks, investment management, and expense management.
- vision of the stock and crypto markets.

Elementi negativi

- This software depends on if a platform such as *Binance* [1] gives an open API, then your action on the budmate will actually be executed on the Binance platform.

[1] `https://www.binance.com/en/binance-api`

Esempio

This software was supposed to provide visual tools to analyze market conditions, but it will be implemented once all basic functionalities are satisfied. Such as using deep learning to analyze future accounts' conditions based on the historic data. (Backtracking).

Requisiti funzionali

- There should be a login and registration screen upon software's activation, authentication of profile via database or google / Facebook authenticator.
- A profile page contains everything about the user, including total value, number of accounts, activities, subscription plan, whether the client needs to pay fees or not, and even accessing friends' pages.
- Investment page: an over*view* of all assets owned on various platforms, a chart showing the trend of the total value of the investments. The ability to purchase and sell assets, such as BITCOIN, and APPLE. In the future, there will also be NFT markets, government bonds, real estate, and maybe even gaming assets(metaverse).
- Accounts page: Bank accounts, investment accounts, Budmate accounts, with various information containing IBAN, and swift code. The ability to create or connect to an existing bank account, whether is Uni credit or Goldman Sachs.
- Expenditure page: shows various kinds of expenses done in the shopping mall, grocery store, book shop, restaurants, cafe bar, monthly subscriptions, student loans, and charts with visual future trends.

Requisiti non funzionali

- For the workflow, we use git and develop the software on different machines and OS, including windows, ubuntu 22.04, 20.04, and macOS.
- The architecture of software should be highly independent, If one member couldn't work full time, that shouldn't bother the others' work.

1.2 Analisi e modello del dominio

Our app starts from the profile class, in the profile, there can be many types of accounts: such as expenses, bank accounts, investment accounts, holding accounts, and even Budmate accounts. Those accounts have similar functionalities, in below, you can find specific implementations.

Elementi positivi

- easy structure, simple responsibility.
- independent implementation without depending on the other's realization, the use of interface.

Elementi negativi

- Its functionality is dependent on access to the internet, If a user is offline, it's hard to do any trading operation.

Esempio

You can see the architecture below.

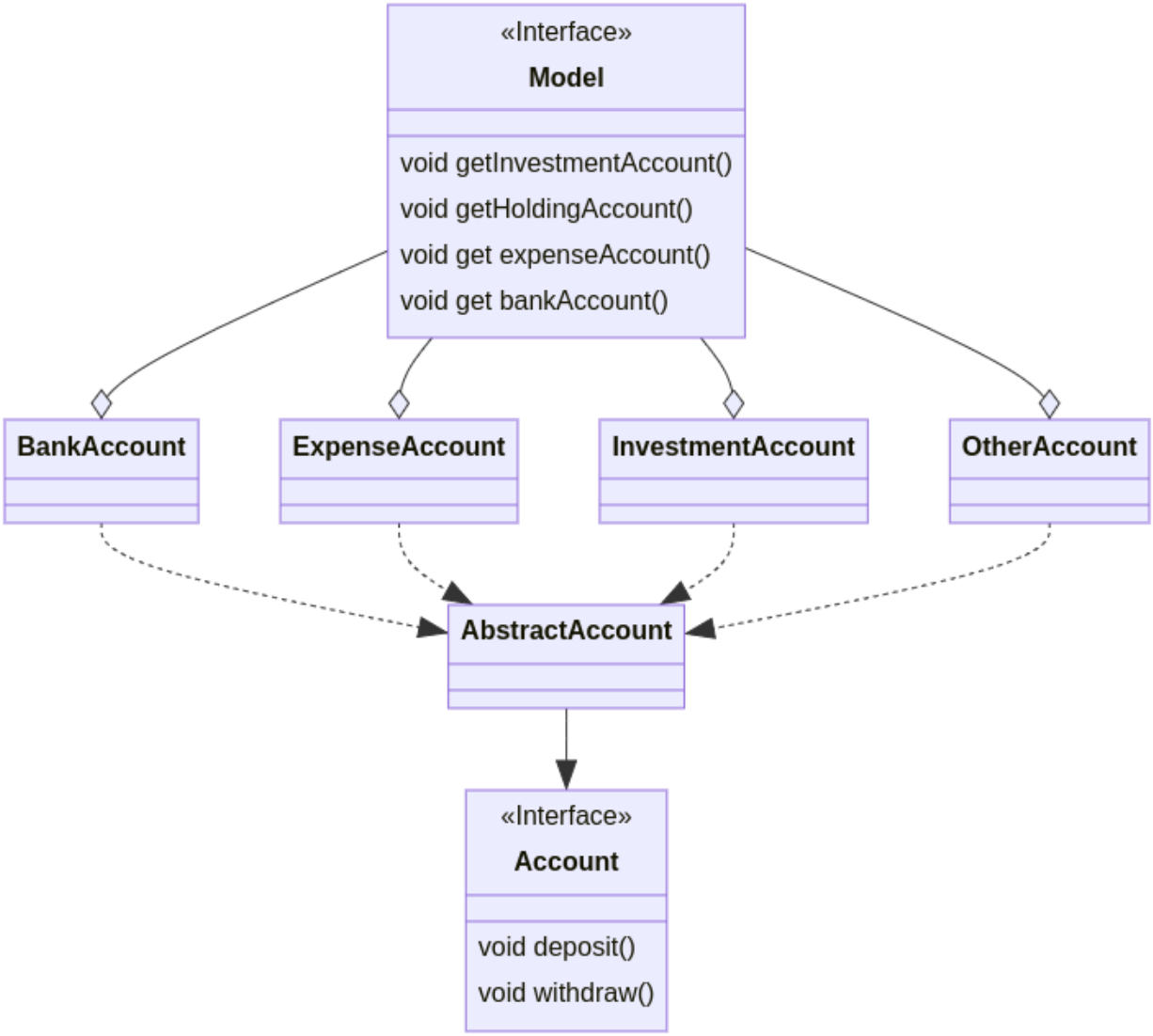

Figure 1.1: Our structure of model

Chapter 2

Design

Our design fulfills tremendously *5 design principles*

principles

- DRY: Don't repeat yourself
- KISS: Keep it simple, stupid
- SRP: Single Responsibility Principle
- OCP: Open-clodes principle
- DIP: Dependency-inversion principle

2.1 Architettura

We use design pattern MVC(*view*, control, model) for our whole structure, we use JavaFX for our *view* and other *view*s for logging. A controller that gets notified by JavaFX event, access model for computation then updates the *view*.

Elementi positivi

- By using MVC and Interface, we can switch between JavaFX and other graphic libraries without changing other parts.
- We use other threads to compute tasks that uses a lot of time.

Elementi negativi

- We asked many people and in the forum, but it seems we couldn't create a reference of the controller without being static.

Esempio

Even though using the private static volatile controller is a bad solution, it works correctly with synchronization between the thread from the controller and the components on the JavaFX thread. I have been working on that for more than 15 hours, but still couldn't find a solution. In this case, we have multiple *view*s, each of them is attached to the one controller, it wasn't something that we didn't intend to do. Instead on the he part of controller, all *view*s are accessible from a thread pool created by Executor services.

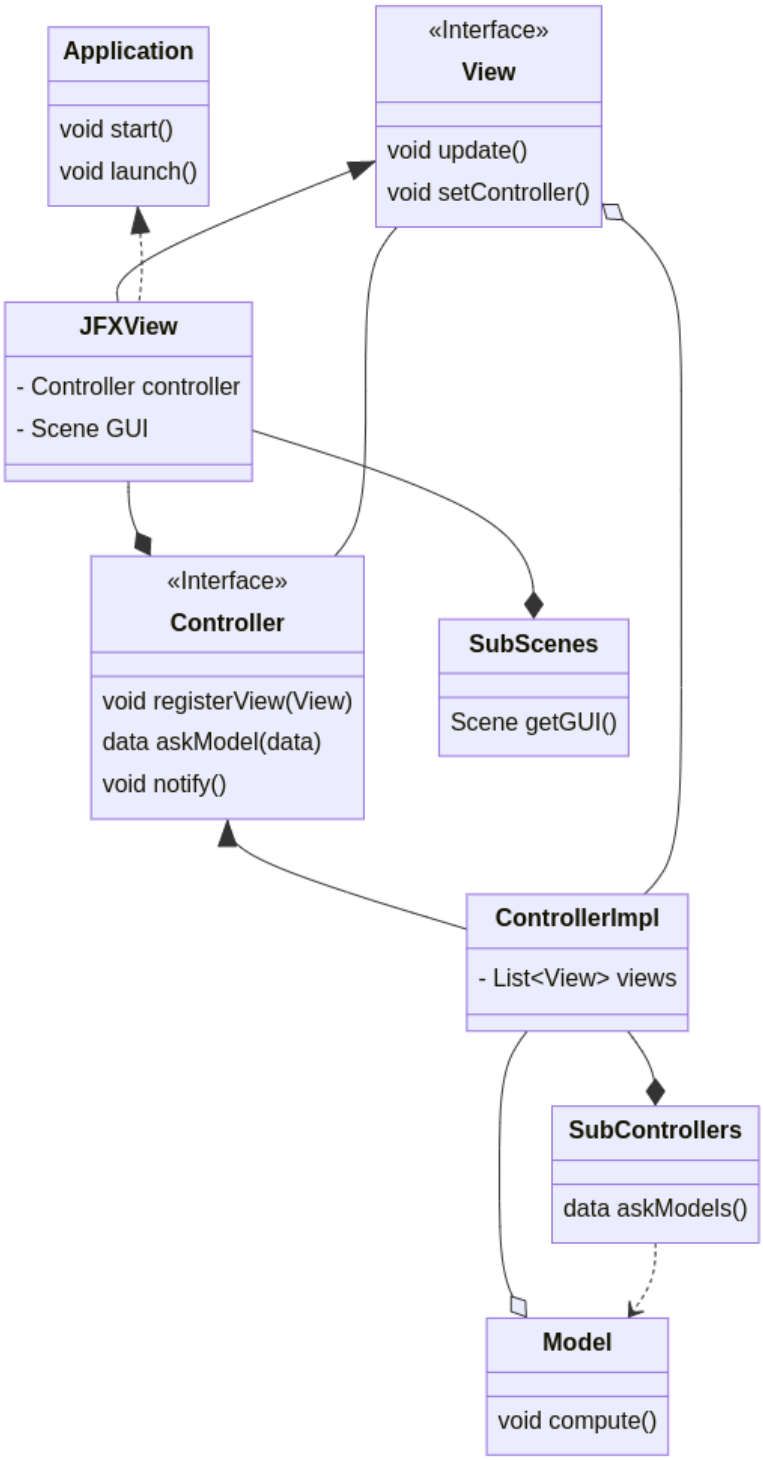

Figure 2.1: Our structure of MVC implementation

2.2 Design dettagliato

2.2.1 Zhaohui Song

Since I started the project earlier than the others, I feel that I don't want the others to rewrite the same piece of code multiple times, so every design should at least satisfy DRY OCP, either for the Model or the GUI.

Some patterns are perfectly fit for my solution including the *Factory method, Builder, Template method, Iterator, Strategy, Observer, and Decorato*r, I avoided *singleton* as much as possible.

Some functions are left open because as mentioned above, we may need to add the third-party API to fulfill the order done by the client. Like if a user Bob wants to buy a share of Amazon stock since our app is a kind of portfolio tracker, he can hit the button Buy in our app, then our app will send the order to the actual broker app, the advantage is you can analyze the data using our app, then executing the order automatically to various platform, in the case one platform bankrupts, you won't lose everything.

You may not see all implementations of these functionalities, but I think when it comes to design an app, I ought to make it equipped with **scalability, interoperability**(coherent with other existing apps, because I don't want to write something that already exists), **standardized, feature-complete, and easy for the rest of team members to develop**. And by thinking that, I used most of my 80 hours to make it easier for the others use.

Generalise Account as much as possible and create my part InvestmentAccount

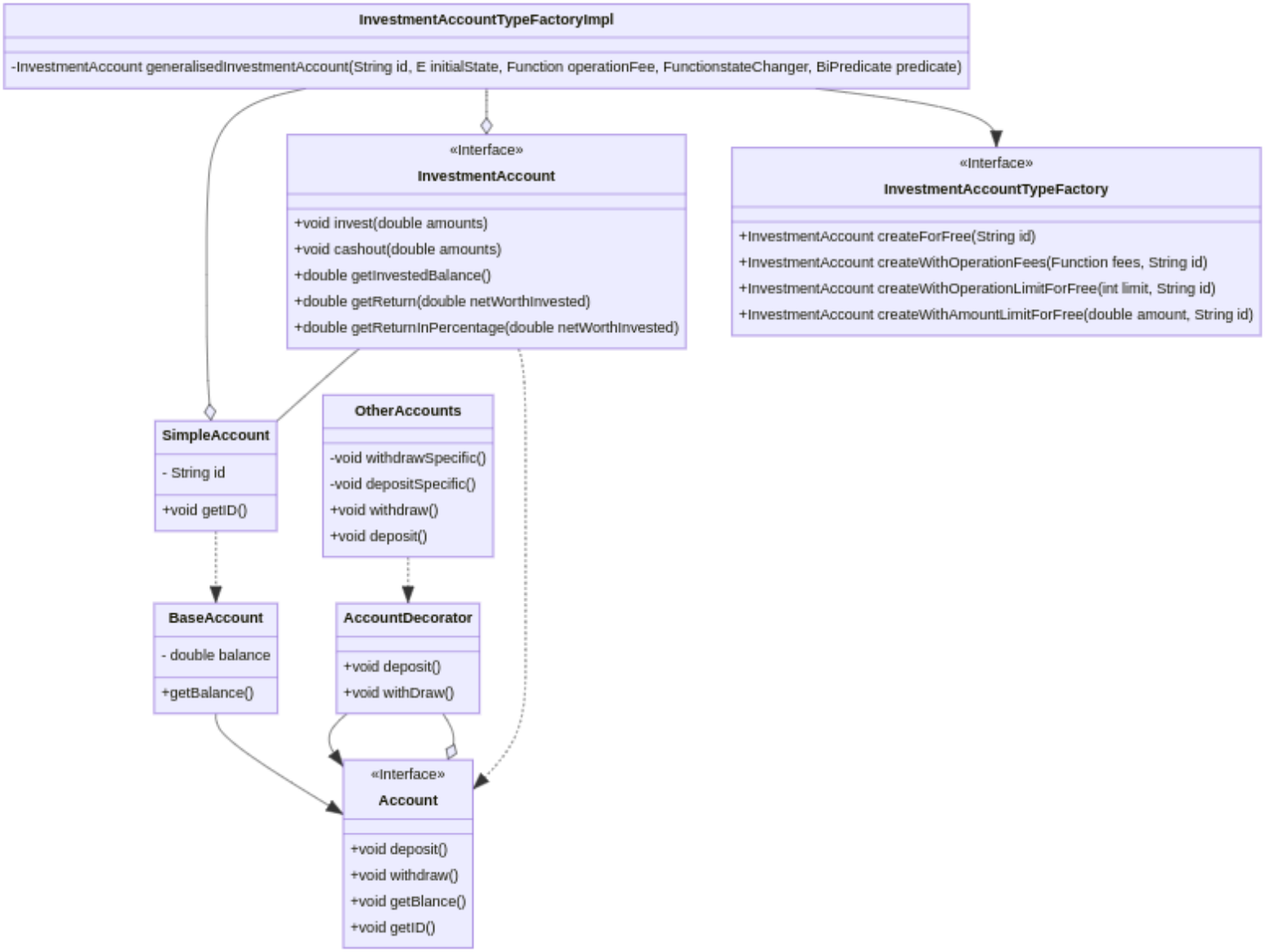

Figure 2.2: generalised Account

Problem: Make an account interface that's common to everybody

Solution: An interface that satisfies basic operations such as deposit and withdrawal. In my opinion, all accounts are basically simple accounts, So

you can either extend it to make a more detailed realization or implement ***decorators*** to add single responsibilities to the easy account, or even use it as a component as I did in my investment account factory.
Note: The decorator here is not implemented, because it was designed for open changes. As an example, if someone wants to add a layered fee based on the accounts' features or traits, one can simply create an account using a new AccountWithFeePlanPlatinium(new Account) without the necessity of using the factory method.

Problem: When this app is ready, I need to think about how to make a profit, depends on the current user's description plan, we may charge a fee to his trading operation, add a limit to the number of times that a user can trade, and add the limit amount for withdrawal, etc.

Solution: As I will have multiple genres of investment Accounts, I create a factory that builds different investment accounts for me. Here I used **strategy**(function and bipredicate) via lambda expression for how to charge a fee form client; An initial state with a generic type to define based on what factor a limit can be added. The pattern used here consists of the **Factory method, template method**.

Need a place to trade stocks with the real-time price change

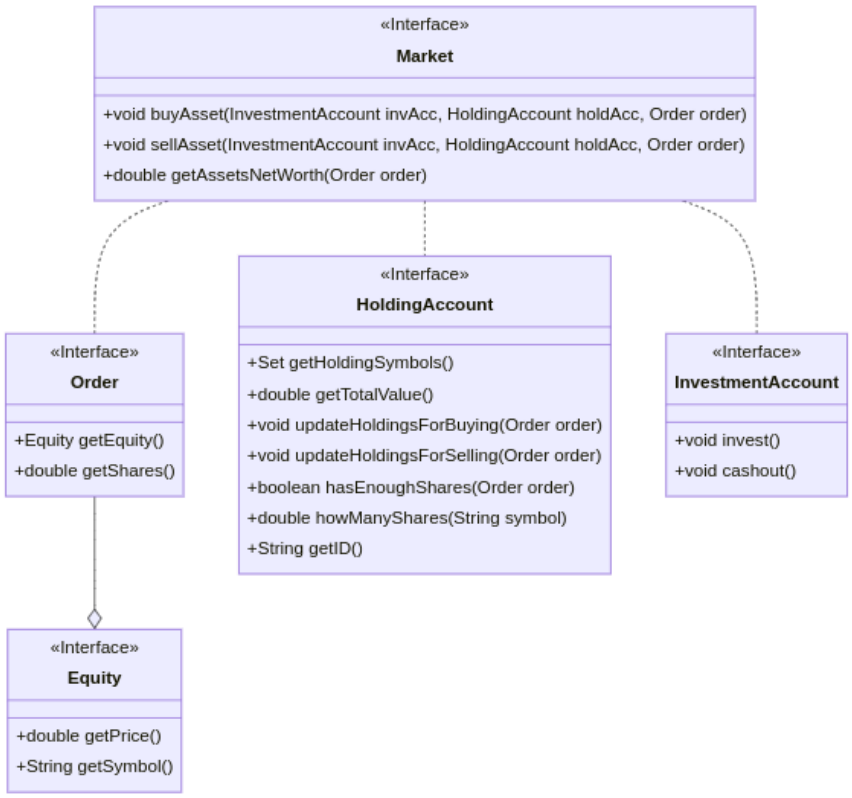

Figure 2.3: market

Problem: How to trade generalised assets with real-time prices

Solution: Here I generalize every kind of asset into class Equity, which can be either stock, cryptos, NFTs, commodities, ETFs, etc. All we need is a symbol and its price. As you see in the figure above, when a user trade with a symbol, assets will be added to the holding account, and money will be decreased in the corresponding investment account.

A group of databases to retrieve real world information

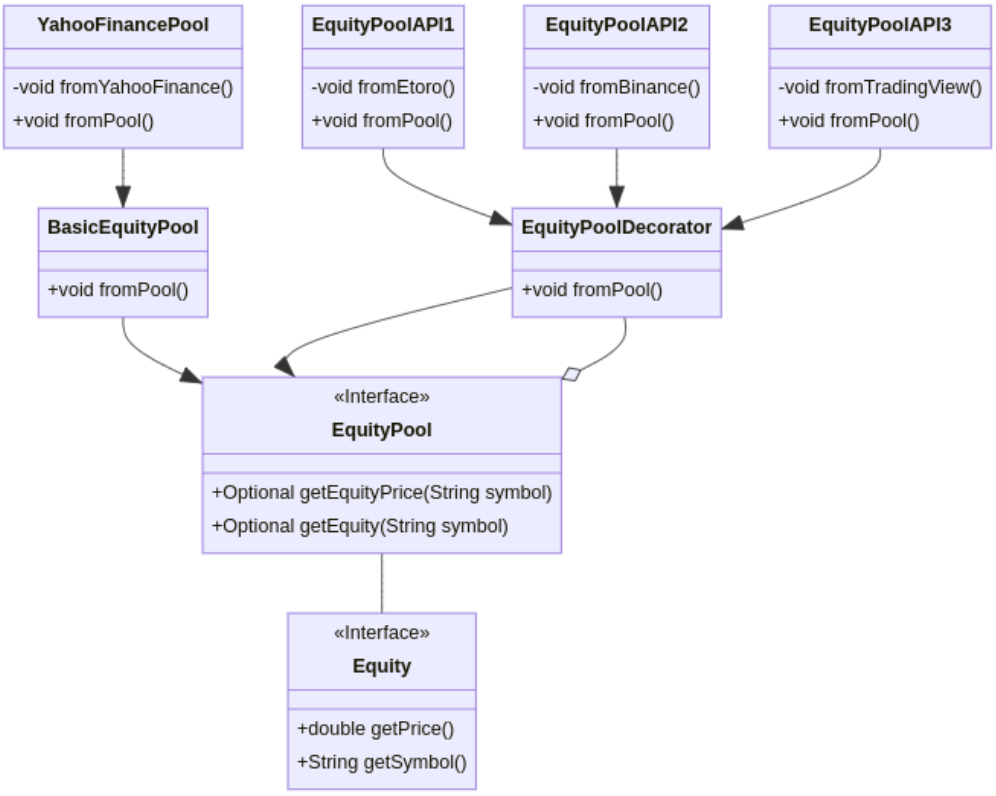

Figure 2.4: EquityPool

Problem: How to query prices from more platforms

Solution: Using the design pattern **decorator** perfectly resolved this problem, firstly we search the price on our default platform Yahoo Finance. In the case we know that an asset may not be found there, we can add more layers for searching from other platforms. It's like a cache hit(how the CPU uses cache to find data in the memory). If the price can be found in the cache of level 1, then that's it, otherwise, it will searches from the cache level 2...n till the central memory. As an example, if a user were searching for a 10-year government bond, it's likely that it won't appear in the market. Then we can do Equity eq = new EquityPoolApiBond(new YahooFinancePool()); So we cover the searching as much as possible.

Abstracting javafx from the *view*

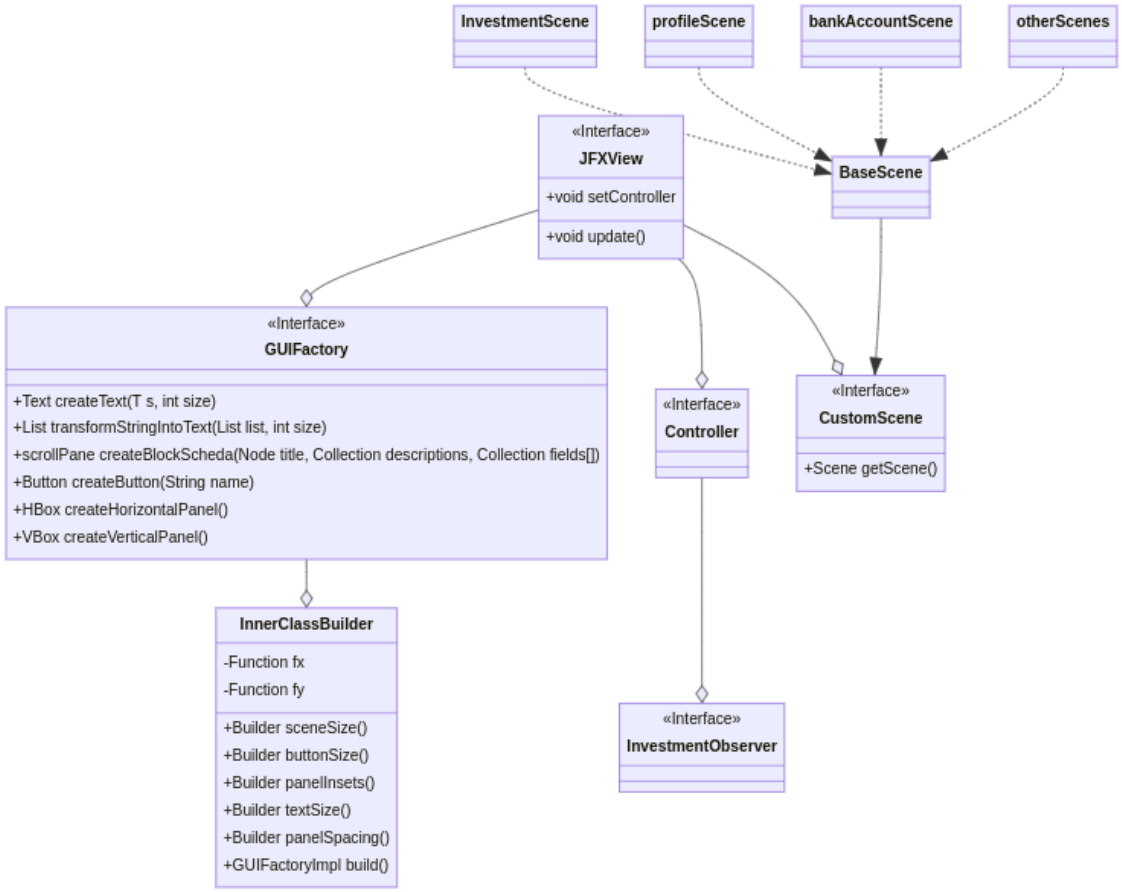

Figure 2.5: Observer for *view*s with multithreading

Problem: I spent time on learning JavaFX, how to avoid it for the others

Solution: Gathering all JavaFX creations inside of one class *GUIFactory* is extremely easy for everyone, whoever needs a component, can be created using a factory class without tedious settings. I used an inner class Builder to set configurations. In this design, every element's size of the GUI is in the percentage of the current screen. So no matter whether you open the app on a 1080p laptop or 8k curved monitor, it's should be always legitimate to read.

Problem: Everyone needs to create their own scene, how to avoid everyone working on the same files.

Solution: I created a common class Custom Scene, its attached controller, and main stage, so each member can write its counterpart scene in their class without losing the flexibility of MVC.

Problem: Some tasks such as reading data from databases can take a lot of time

Solution: By introducing Task in the controller, some tasks can be done in other background thread, GUI changes on the JavaFX thread, computational works on background threads, and main app on the main thread.

Problem: Every time a new task emerges, a new thread is being created, how to avoid thread throttling?

Solution: By using ExecutorService we create 10 threads at first, then every task will only be executed on these specified threads, so to avoid throttling

Problem: How to convert all kinds of data into Text with the defined format for display.

Solution: In the class of *GUIFactory*, there is a function/algorithm, that accepts arbitrary data type and converts it into String, by using the technique of Reflection in java.

Problem: How to update the widgets with infinite arguments with each of them a different type

Solution: Every time when controller updates a widget on the *view*, instead of passing tons of arguments, a *Queue(List(?)) packet* can be used. The merit is you can get an iterator from that, and get infinite of arguments with the arbitrary data type.

Elementi positivi

- By using design pattern, it easily solved most of the problems
- Every piece of code is reusable
- A lot of use of Interface that satisfies Dependency-inversion principle

Elementi negativi

- In order to be flexible and reusable, each piece of code might be a bit difficult for the others to understand, especially for those who didn't understand OOP concepts.
- I've done every class beforehand, but without immediate feedback, I couldn't understand other team member's real needs.
- Since I keep improving my code, there were lots of changes every time I push the code to git, but I still tried to do my best.

Chapter 3

Sviluppo

3.1 Testing automatizzato

For automated testing, we use the Junit 5 to test them, thanks to Gradle, we have a simple command to test all tests on the terminal, without the dependency on any editor.

3.1.1 Zhaohui Song

I didn't spend a lot of time testing every possibility, however a minimum of tests that are deployed are:

- TestAccount.java
- TestInvestmentAccount.java
- TestMarket.java
- TestGuiFactory.java

3.2 Metodologia di lavoro

3.2.1 Zhaohui Song

We have decided together that we are going to use JavaFX, from that point I have been charged to design the whole app architecture(Because I am most available on the project).

Fully Independent work:

- src/model/account: *Account, BaseAccount, InvestmentAccount, InvestmentAccountTypeFactory, InvestmentAccountTypeFactoryImpl, NotEnoughFundsException, NotEnoughSharesException, SimpleAccount*
- src/model/market: *Equity, EquityImpl, EquityPool, EquityPoolStock, EquityStock, HoldingAccount, HoldingAccountImpl, Market, MarketImpl, Order, OrderImpl.*
- src/view: *BaseScene, CustomScene*
- src/view/investment: *InvestmentScene*
- src/control/investment: *InvestmentViewObserver, InvestmentViewObserverimpl*

Cooperated work:

- src/view: *JavaFxView, GUIFactory and GUIFactoryImpl(I've done 95%)*
- src/control: *Controller, ControllerImpl*

3.3 Note di sviluppo

3.3.1 Zhaohui Song

I configured the whole project using Gradle, built dependency, including creating branches for DVCS, mainly we worked on our feature branches, I was the manager of our repository, and other members sent me code via pull request, I checked them, wrote some comments if they had programmed correctly using OOP concepts. Our main code was on the branch develop, I think it will be merged into the branch master before handing in this project. Sometimes one of them commit something on the branch develop, then I had to rebase the divergence, and maintain the repository. Or maybe checkout an an file in an old hash. It was an essential tool for this project to work with more people.

- The interested feature used during development:
  - Designed functions with generics, as example, the use of generics bounded (? extends Numbers) (? extends Nodes) etc..

- Intensive use of lambda expressions (for interface Function and predicate)
- Use of `Stream`, of `Optional`
- Use of reflection
- Use of volatile variable for synchronization
- Use of third-party api: YahooFInance, ExecutorService, JavaFx, com.google.guava.

Used code from the Internet

I made a special package for the code of other engineers src/main/util:

- AutoCompleteTextField from `https://gist.github.com/floralvikings/10290131` author Caleb Brinkman
- Pair from our class @author professori di OOP

Chapter 4

Commenti finali

4.1 Autovalutazione e lavori futuri

Zhaohui Song

In this project, I have been working around 5 8 hours per day for a month, I started the project on 15 July, as today it's 16 Agost. I worked in an agile way, such as writing code, then finding it redundant, then designing again, rewrite the code. It took me about 40 % of the time on design, 40% on coding + improving the quality, and 20% on debugging(especially the part of threading). I tried to generalize my code as much as possible just to make this project feature-complete and reused for the future. There are too many features that can be added, such as adding a comparator to the menu, so a user can order its stocks and crypto in a preferred way, and everything you can read on readme or I mentioned above.

4.2 Difficoltà incontrate e commenti per i docenti

Zhaohui Song

Thanks to prof Viroli for extending our deadline to carry out the project. This course was amazing, more than I expected from what could be learning programming in java. A big thank you to every professor of this course, your lecture was easy to understand, and the lessons in the lab were so important for me to grasp these codes, along with those exercises on github.

Appendix A

Guida utente

fattello voi un po'... L'ho quasi finito la relazione tutto da solo di nuovo. :(sigh...

Elementi positivi

- Si istruisce in modo semplice l'utente sull'uso dell'applicazione, eventualmente facendo uso di schermate e descrizioni.

Elementi negativi

- Si descrivono in modo eccessivamente minuzioso tutte le caratteristiche, anche minori, del software in oggetto.
- Manca una descrizione che consenta ad un utente qualunque di utilizzare almeno le funzionalità primarie dell'applicativo.

Appendix B

Esercitazioni di laboratorio

Esempio

B.0.1 Zhaohui Song

- Laboratorio 06: https://virtuale.unibo.it/mod/forum/discuss.php?d=87880
- Laboratorio 07: https://virtuale.unibo.it/mod/forum/discuss.php?d=88829
- Laboratorio 08: https://virtuale.unibo.it/mod/forum/discuss.php?d=89272
- Laboratorio 09: https://virtuale.unibo.it/mod/forum/discuss.php?d=90125
- Laboratorio 10: https://virtuale.unibo.it/mod/forum/discuss.php?d=91128