Object Oriented Programming

CPS 2004

Video:
https://drive.google.com/file/d/13MNxvLmP1-_3HeRn6i6aU2ghmZEZOJvB/view?usp=sharing

Owen Attard
0202503L
Bachelor of Science in Information Technology (Honours) (Artificial Intelligence)

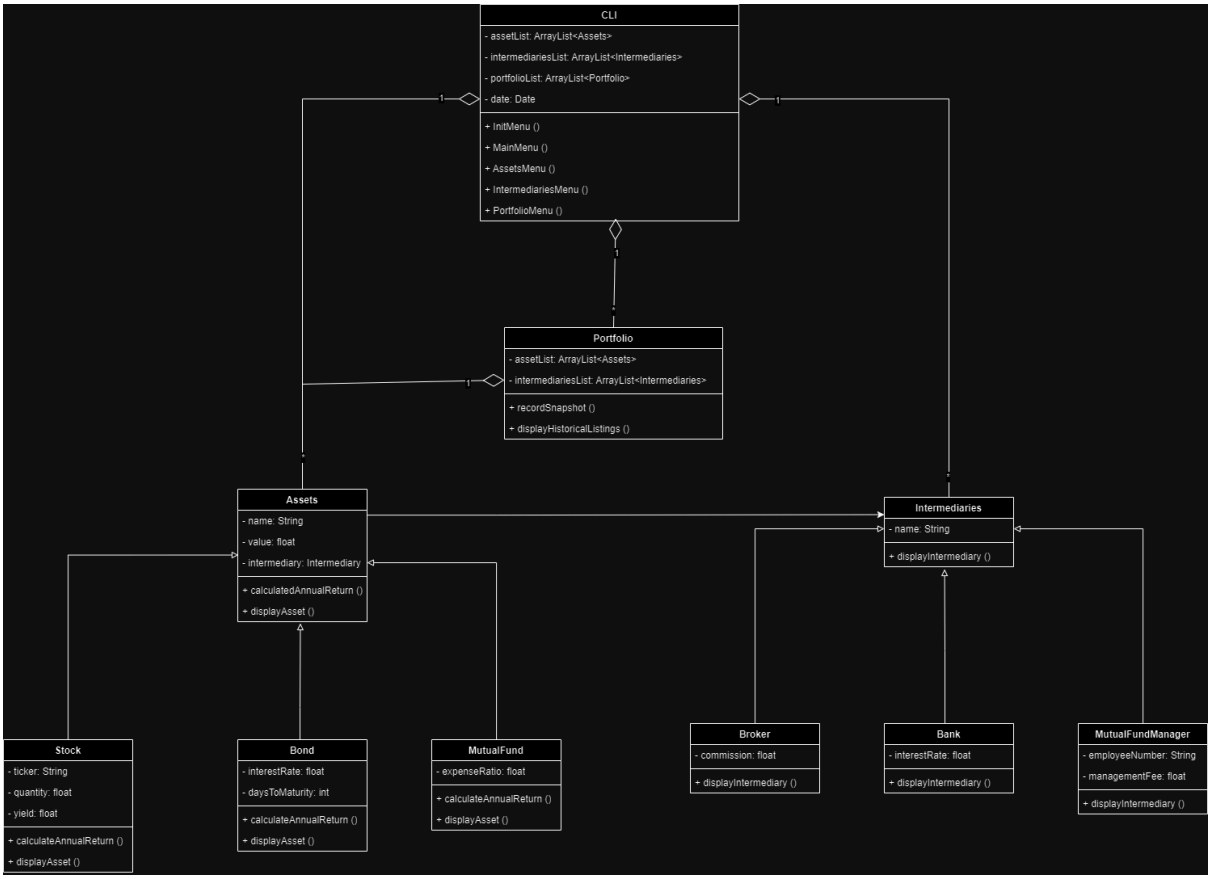
Table of Contents

Statement of Completion

Task	Status
Task 1A - UML Diagram	Completed
Task 1B - Java Implementation	Completed
Task 1C - C++ Implementation	Completed
Task 2A - Historical Snapshots	Completed
Task 2B - Builder Pattern	Completed
Task 2C - Task 2 Justification	Completed
Task 3A - Proto File	Completed
Task 3B - Facade	Completed
Task 3C - Asset Viewer	Psuedo Code Available
Task 4A - Remote Service Proto File	Completed
Task 4B - Updated Java Main	Completed
Task 4C - Updated Main (Other Language)	Not Attempted

Task 1

UML Diagram



Java Implementation

The Java implementation uses a Maven multi-module project meaning having a parent pom.xml file which is depended by its project modules, each module has its own pom.xml file. A module is created for each Java related task.

The Java implementation for task 1 is found in Java/oopresit/task1/, where in this directory there is the pom.xml file of task1, additionally this directory is further split into three subdirectories, *Saves* which stores the previously saved application state, *src* which withholds all java files and *target*.

To run the java project use Maven: go to the directory Java/oopresit/task1 and write mvn exec:java, you might need to previously run mvn clean install

This implementation is split into 5 Java files:

- Intermediaries
- Assets
- Portfolio
- CLI
- Main

Intermediaries

Intermediaries.java contains 4 classes, Intermediaries is an abstract base class to 3 sub-classes, and Intermediaries is made up of 1 private string variable *name* and 1 polymorphic method *displayIntermediaries()*.

There are 3 subclasses, Broker, Bank and MutualFundManager, all three sub-classes inherit the *name* variable and the method *displayIntermediaries()* from Intermediaries. In addition to the inherited attributes each sub-class has its own additional variables:

- Broker: float commission
- Bank: float interestRate
- MutualFundManager: string employeeNumber, float managementFee

Lastly, each sub-class overrides the inherited *displayIntermediaries()* function using getters for *name* and the class's own variables to formulate a more specific message for its own class.

Assets

Assets.java contains 4 classes, Assets is an abstract base class to 3 sub-classes, and Assets is made up of 3 private variables and 2 polymorphism methods. The private variables are string *name*, float *value* and an associated intermediary from the Intermediaries class. As for the polymorphic methods they are *calculateAnnualReturn()* and *displayAsset()*.

There are 3 subclasses, Stock, Bond and MutualFund, all three subclasses inherit the attributes of Assets. In addition to the inherited attributes each sub-class has its own variables:

- Stock: string *ticker*, float *quantity*, float *yield*
- Bond: float *interestRate*, int *daysToMaturity*
- MutualFund: float *expenseRatio*

Each subclass overrides the inherited methods from Assets, for *displayAssets()* each subclass uses getters for the inherited variables of Asset and its own variables to formulate a more specific display message for its own class. Similarly, each subclass overrides the *calculateAnnualReturn()* using the getters mentioned to equate a predicted annual return for the current financial year of the specific asset. These equations were built to avoid negative returns as we are aiming for large profits.

Stock calculateAnnualReturn()

```
return 365*(get_yield()+broker.get_commission()) * (get_value()*get_quantity());
```

Bond calculateAnnualReturn()

```
if (get_daysToMaturity() < 365) {
    Bank bank = (Bank) intermediary;
    return get_daysToMaturity() * (get_interestRate() + bank.get_interestRate()) * get_value();
}else{
    return 0;
}
```

MutualFund calculateAnnualReturn()

```
return 365 * (get_value() * (get_expenseRatio() + mutualFundManager.get_managementFee()));
```

Portfolio

The Portfolio class is used to manage a collection of assets and historical snapshots of these assets. The class takes an ArrayList of type Assets as a parameter for its construction and uses an ArrayList to store the list of currently owned assets and a Map that maps a date to strings, these strings will be formulated when recording snapshots which will use the polymorphic methods *displayAsset()* and *displayIntermediaries()* functions which were mentioned earlier.

Portfolio has two main methods *recordSnapshot*(Date) and *displayHistoricalListings(int sort)*

Record Snapshot

This method takes a date as its parameter which will be used as a key to insert a new entry into the Map. The method simply iterates over all assets in the assets list, for each asset it uses the getter in Assets to retrieve the associated intermediary, having both the asset and its associated intermediary the function simply concatenates the *displayAsset()* and *displayIntermediaries()* functions together as a string and adds the concatenated message in an ArrayList of strings. Finally, the method passes the date parameter alongside the ArrayList of concatenated string messages to the Map.

Display Historical Listings

Takes an integer as a parameter which signifies if the desired order of listings should be ascending or descending as desired by the user's input. Then the function will iterate over all dates of the map, for each date it will list all snapshot details of that current date's map.

CLI

The CLI class is used to control the user interface by controlling several methods to display a variety of CLI menus as selected by the user. The CLI class uses a few private variables, namely ArrayLists for the collection of currently held Assets, Intermediaries and Snapshots, also, a date variable which is used for creating snapshots and an integer *selection* variable which is used for validation in the cli menus.

The *date* variable was added to create a better experience for the portfolio snapshots as instead of always taking the current timestamp for testing, the *date* variable is used as a global variable and is incremented for every snapshot recorded.

The CLI class has a few main functions for CLI menus and additional helper functions to process a selected action:

Init Menu
The initialisation menu allows for three options either start from scratch as a new save, load a new save or change your mind and exit the application.

Main Menu
The main menu allows for four options either to enter the Portfolio menu, the Assets menu, the Intermediaries menu or to Exit and Save the program.

Portfolio Menu
Allows 4 options, either Record Snapshot, Display Historical Listings, Calculate Annual Return (of currently owned assets) or go back to Main Menu.

Assets Menu
The Assets Menu allows for 5 options Create Assets, Read Assets, Update Assets, Delete Assets or go back to Main Menu.

Intermediaries Menu
The Intermediaries Menu allows for 5 options Create Intermediaries, Read Intermediaries, Update Intermediaries, Delete Intermediaries or go back to Main Menu

Create Assets Menu
The Create Assets Menu allows for 4 options Stock, Bond, Mutual Fund or go back to Assets Menu

Create Intermediaries Menu
The Create Intermediaries Menu allows for 4 options Broker, Bank, Mutual Fund Manager or go back to Intermediaries Menu

As mentioned earlier there are many other helper functions which take care of processing any selected action from the CLI Menus above:

- Getters and Setters for *Date* variable
- Increment_date: adds 86400000 ms to the current *Date* variable
- createStock: checks if an associated intermediary is available and if so, takes care of user input for creating the Stock Asset
- createBond: checks if an associated intermediary is available and if so, takes care of user input for creating the Stock Bond
- createMutualFund: checks if an associated intermediary is available and if so, takes care of user input for creating the Stock MutualFund
- createBroker: takes care of user input for creating Intermediaries Broker
- createBank: takes care of user input for creating Intermediaries Bank
- createMutualFundManager: takes care of user input for creating Intermediaries MutualFundManager
- readAssets: lists all assets in ArrayList<Assets>
- updateAssets: list all assets in ArrayList<Assets> as a CLI Menu, passing the selected asset to either updateStock(Stock stock), updateBond (Bond bond) or updateMutualFund (MutualFund mutualfund) respectively.
- updateStock: lists each set variable of Asset Stock and updates each variable per user input
- updateBond: lists each set variable of Asset Bond and updates each variable per user input
- updateMutualFund: lists each set variable of Asset MutualFund and updates each variable per user input
- deleteAsset: lists all sets in ArrayList<Assets> as a CLI Menu, deleting the selected asset accordingly after asking for confirmation of deletion
- readIntermediaries: lists all Intermediaries in ArrayList<Intermediaries>
- updateIntermediaries: lists all Intermediaries in ArrayList<Intermediaries> as a CLI Menu, passing the selected Intermediaries to either updateBrokerDetails(Broker broker), updateBankDetails(Bank bank) or updateMutualFundManagerDetails(MutualFundManager mutualfundmanager) respectively
- updateBrokerDetails: lists each set variable of Intermediaries Broker and updates each variable per user input
- updateBankDetails: lists each set variable of Intermediaries Bank and updates each variable per user input
- updateMutualFundManagerDetails: lists each set variable of Intermediaries MutualFundManager and updates each variable per user input

- deleteIntermediaries: lists all IntermediariesIntermediaries in ArrayList<Intermediaries> as a CLI Menu, deleting the selected Intermediaries accordingly after asking for confirmation of the deletion
- loadState: lists all previously saved states from the Saves/ directory as a CLI Menu, and after selection of save by user sets the ArrayLists of CLI.java to the ones deserialised in loadState
- saveState: saves the current state action of the program under the filename of the current date timestamp in the Saves/ directory, writing the three ArrayLists used by CLI.java, assetsList, intermediariesList and portfolioList.
- displayHistoricalListings: asks the user to select a sorted order for the historical listings, either ascending or descending and calls the displayHistoricalListings from Portfolio.java
- annualReturn: iterates over all currently owned assets, for each asset we get the its annual return using the polymorphic method of Assets *calculateAnnualReturns* and the function will sum up all returns and display it to the user.

Main

The main function simply calls the *InitMenu()* function of the CLI class to start the program.

C++ Implementation (Differences from Java)

The C++ implementation uses Cmake to create a main project with multiple executables for the varying subtasks of C++. The C++ implementation for task 1 is found in /Task-1/C++ Version.

To run task 1 navigate the path to Task-1\C++ Version\out\build\MyPreset1 and run the following command: .\Task-1-C++.exe

In this section, we will discuss the differences between the Java version of task 1to the C++ version.

The C++ implementation is made up of the following .cpp and .h files:

- Assets.h
- Assets.cpp
- Intermediaries.h
- Intermediaries.cpp
- Portfolio.h
- Portfolio.cpp
- CLI.h
- CLI.cpp
- SaveHelper.h
- SaveHelper.cpp
- main.cpp

Apart from the obvious header files, the other main differences are:

- SaveHelper files
- Assets and Intermediaries now use an enum for type since Java could use instanceof
- Instead of ArrayLists which was a main part of the Java implementation C++ uses vectors

<u>Save Helper</u>
The save helper class is used in conjunction with the CLI class to serialise and deserialise the program state and write/read to and from files, in java de/serialisation was implemented using Serializable.

Save Helper has two functions *writeString* and *readString* which read and write a string to a binary file. The CLI class uses these two functions to save and load the program's state.

Task 2

Historical Snapshots

The Java implementation for task 2 is found in Java/oopresit/task2/, where in this directory there is the pom.xml file of task2 which has dependencies on the parent and task1 pom.xml files, additionally, this directory is further split into two subdirectories, src and target.

To run the Java project use Maven: go to the directory Java/oopresit/task2 and write mvn exec:java, you might need to previously run mvn clean install

Task 2 uses Assets.java and Intermediaries.java from task1, with the following new task-specific Java files Main2.java, CLI2.java and the new **HistoricalSnapshots2.java which replaces Portfolio.java**.

Main2

Main2 is the same as Main just instead of calling CLI.java it calls CLI2.java's InitMenu() function to start the program.

Historical Snapshots

Create a dedicated class hierarchy of immutable classes to represent the historical snapshots of the financial assets.

HistoricalSnapshots2.java contains 4 classes, HistoricalSnapshots2 is an abstract base class to 3 sub-classes and is made up of several private variables: string *assetname*, float *assetvalue*, string *intermediaryname*, Map from a date to a list of strings *historicalSnapshots* and Date *snapshotDate*. Additionally, it also has three polymorphic methods *displaySnapshot()*, *generateSnapshotDetails()* and *createNewInstance(Map)*.

HistoricalSnapshots2 uses deepCopyto create deep copies of the Map, which helps maintain immutability as a Map variable is mutable.

There are 3 sub-classes: StockSnapshot, BondSnapshot and MutualFundSnapshot each inheriting the attributes of HistoricalSnapshot2 and additionally having their own class-specific attributes:

- StockSnapshot: string *ticker*, float *quantity*, float *yield*, float *commission*
- BondSnapshot: float *interestRate*, int *daysToMaturity*, float *Intermediary_interestRate*
- MutualFundSnapshot: float *expenseRatio*, string *EmployeeNumber*, float *managementFee*

Lastly, each sub-class overrides the inherited polymorphic methods *displaySnapshot()*, *generateSnapshotDetails()* and *createNewInstance(Map)* to make them specific to their respective sub-class. So for example *generateSnapshotDetails()* each sub-class lists the inherited variables and the class's variables.

Builder Pattern

Building on the information mentioned above on HistoricalSnapshots2, this section will focus on the build pattern class implemented in each sub-class of HistoricalSnapshots2.

Each builder class has the same variables as their parent sub-class. So for example: StockSnapshot inherits 5 variables from HistoricalSnapshots2 and has an additional 4 variables itself, this means that the build class of StockSnapshot has all 9 variables initialised.

Each builder class has a setter for each of its available variables, so building on the previous example the builder class of StockSnapshot has 9 setter methods. Lastly, after all the setter methods each builder class has a build method which ensures that all required fields are set and handles the creation of a fully initialised instance of its snapshot sub-class.

Justification

In this section, we will focus on the benefits that accompany the use of immutability and builder pattern design in software development with regard to the software's maintainability and robustness.

A program's maintainability refers to how easily it may be changed to amend issues and make enhancements or adaptations [1]. A program's robustness refers to its dependability concerning external errors [2].

Immutability

Since immutable objects in our case the historical snapshots cannot be modified, there is no need to implement any additional code for safety/protection which results in a smaller codebase that is easier to modify and maintain. As for the robustness of using immutable objects like historical snapshots, there is a clear and safe origin, this is very beneficial to the financial industry for example when processing data of transactions, the data is generated at the moment of the transaction and it is unable to be changed for security and safety reasons i.e. no tampering from any external environment.

Builder Pattern

Implementing a builder pattern design allows for more readable code, instead of having complex constructors the builder pattern allows us to make use of simple method calls to build our desired object. They also allow for a more manageable complex interface with the possibility of having optional or required fields constructed in a simple step-by-step approach. Basically, they create an organised approach to object creation allowing a developer the ability to control and make amendments to the object's creationin a cleaner and simpler way.

Having this organised object creation approach allows the program to avoid any external faults by preventing invalid objects due to the objects being created consistently and safely.

Task 3

The Java implementation for task 3 is found in Java/oopresit/task3/, where in this directory there is the pom.xml file of task3 which has dependencies on all previous Java tasks' pom.xml files, additionally, this directory is further split into two subdirectories, src and target.

To run the Java project use Maven: go to the directory Java/oopresit/task3 and write mvn exec:java, you might need to previously run mvn clean install

Proto File

In the src subdirectory, there is another directory called main which then splits into java and proto. The Test.proto file is found in the proto directory i.e. under Java\oopresit\task3\src\main\proto\.

This task doesn't use the latest Protobuf version 4.28.0 but instead uses Protobuf version 4.27.0 as 4.28.0 still has some issues due to its recent release.

Test.proto

Test.proto uses syntax proto3 and consists of 16 messages split across 3 sections for Assets, Intermediaries and HistoricalSnapshots.

Intermediaries

To depict intermediaries in proto, 5 messages are used.

- Intermediaries_Proto which acts as an array holding multiple Intermediary_Proto messages.

- Intermediary_Proto which holds string *intermediary_name* and uses oneof to incorporate one of three messages: Broker_Proto, Bank_Proto or MutualFundManager_Proto.

- Broker_Proto: holds a float *broker_commission*

- Bank_Proto: holds a float *bank_interestRate*

- MutualFundManager_Proto: holds a string *mutualfundmanager_employeenumber* and a float *mututalfundmanager_managementfee*.

Assets

To depict assets in proto, 5 messages are used.

- Assets_Proto which acts as an array holding multiple Asset_Proto messages.

- Asset_Proto which holds string *asset_name*, float *asset_value* and uses oneof to incorporate one of three messages: Stock_Proto, Bond_Proto or MutualFund_Proto.

- Stock_Proto: holds string *stock_ticker*, float *stock_quantity* and float *stock_yield*.

- Bond_Proto: holds float *bond_interestrate* and int32 *bond_daystomaturity*

- MutualFund_Proto: holds float mutualfund_expenseratio

Historical Snapshots

To depict historical snapshots in proto, 6 messages are used.

- HistoricalSnapshots_Proto which acts as an array holding multiple HistoricalSnapshot_Proto messages.

- HistoricalSnapshot_Proto: holds google.protobuf.Timestamp *date* and multiple (repeated) AssetSnapshot_Proto
  HistoricalSnapshot_Proto takes a Map format.

- AssetSnapshot_Proto uses oneof to incorporate one of three messages: StockSnapshot_Proto, BondSnapshot_Proto or MutualFundSnapshot_Proto.

- StockSnapshot_Proto holds:
  - string *StockSnapshot_asset_name*
  - float *StockSnapshot_asset_value*
  - string *StockSnapshot_ticker*
  - float *StockSnapshot_quantity*
  - float *StockSnapshot_yield*
  - string *StockSnapshot_intermediary_name*
  - float *StockSnapshot_intermediary_commission*

- BondSnapshot_Proto
  - string *BondSnapshot_asset_name*
  - float *BondSnapshot_asset_value*
  - float *BondSnapshot_interest_rate*
  - int32 *BondSnapshot_days_to_maturity*
  - string *BondSnapshot_intermediary_name*
  - float *BondSnapshot_intermediary_interest_rate*

- MutualFundSnapshot_Proto
  - string *MutualFundSnapshot_asset_name*
  - float *MutualFundSnapshot_asset_value*
  - float *MutualFundSnapshot_expense_ratio*
  - string *MutualFundSnapshot_intermediary_name*
  - string *MutualFundSnapshot_intermediary_employee_number*
  - float *MutualFundSnapshot_intermediary_management_fee*

Facade

Facade.java is used to save and load the program's action state using protocol buffers.
The class has 3 private variables which are all ArrayLists that store intermediaries, assets and snapshots respectively. Apart from this the class only has 2 publicly accessible functions which are *save* and *load*. Finally, the class also has a global Builder variable for each list type initialised, these are of type Assets_Proto, Intermediaries_Proto and HistoricalSnapshots_Proto.

Facade has several helper functions which help *save* and *load* do their job.

Save

Save takes 3 parameters: ArrayList of Intermediaries, ArrayList of Assets and ArrayList of Historical Snapshots.

- The function starts by calling *setLists* with the ArrayList parameters it received.
- Calls *processIntermediaries*() to serialise intermediariesList
- Calls *processAssets*() to serialise assetsList
- Calls *processHistoircalSnapshots*() to serialise snapshotsList
- Calls *saveState*() to write to files

Load

To return multiple arrayLists, the load function is of type ArrayList<Object>. The load function simply:

- Calls *resetLists*()
- Retrieves all previously saved files using *getFiles*()
- Calls *loadCLI* with the retrieves files as a parameter
- Populates the ArrayList of Objects and returns it

Other Functions

- setLists:
  Parameters: 3 ArrayLists of type Intermediaries, Assets and HistoricalSnapshots
  Sets Facade.java ArrayLists to the parameters' values

- resetLists: resets global ArrayLists

- BrokerHelper:
  Parameter: Broker
  Builds an Intermediary_Proto of type Broker_Proto using the builder pattern of protobuf and returns it

- BankHelper:
  Parameter: Bank
  Builds an Intermediary_Proto of type Bank_Proto using the builder pattern of protobuf and returns it

- MutualFundManagerHelper:
  Parameter: MutualFundManager
  Builds an Intermediary_Proto of type MutualFundManagerr_Proto using the builder pattern of protobuf and returns it

- processBroker:
  Parameter: Broker
  Adds the built Intermediary_Proto from calling *BrokerHelper* with the received parameter to the global builder for Intermediaries_Proto

- processBank:
  Parameter: Bank
  Adds the built Intermediary_Proto from calling *BankHelper* with the received parameter to the global builder for Intermediaries_Proto

- processMutualFundManager:
  Parameter: MutualFundManager
  Adds the built Intermediary_Proto from calling *MutualFundManagerHelper* with the received parameter to the global builder for Intermediaries_Proto

- StockHelper:
  Parameter: Stock
  Builds an Asset_Proto of type Stock_Proto using the builder pattern of protobuf and returns it

- BondHelper:
  Parameter: Bond
  Builds an Asset_Proto of type Bond_Proto using the builder pattern of protobuf and returns it

- MutualFundHelper:
  Parameter: MutualFund
  Builds an Asset_Proto of type MutualFund_Proto using the builder pattern of protobuf and returns it

- processStock:
  Parameter: Stock
  Adds the built Asset_Proto from calling *StockHelper* with the received parameter to the global builder for Assets_Proto

- processBond:
  Parameter: Bond
  Adds the built Asset_Proto from calling *BondHelper* with the received parameter to the global builder for Assets_Proto

- processMutualFund:
  Parameter: MutualFund
  Adds the built Asset_Proto from calling *MutualFundHelper* with the received parameter to the global builder for Assets_Proto

- processIntermediaries:
  Iterates over all intermediaries in intermediariesList, for each intermediary pass it to its respective process function i.e. either *processBroker*, *processBank* or *processMutualFundManager*

- processAssets:
  Iterates over all assets in assetsList, for each asset, pass it to its respective processing function i,.e. either *processStock*, *processBond* or *processMutualFund*.

- saveState
  - Gets the current timestamp
  - Builds all three global builders of Assets_Proto, Intermediaries_Proto and HistoricalSnapshots_Proto
  - Writes the 3 files of serialised date, 1 for each proto type
    - Each file name is in the following format type_currenttimestamp.bin.
    - For each file it writes the respective built Protos message to it

- getFiles:
  Parameter: String
  Returns all available files that start with the received string prefix and end with .bin
  Note: This is usually used with the intermediaries prefix to have a unique list of all different saves

- loadCLI:
  Controls a command-line interface for selecting and loading a saved intermediary file, along with its corresponding assets and snapshots files. Finally, it calls the *deserialise* function using the files as parameters.

- Deserialise:
  Parameters: 3 Files depicting saved intermediaries, assets and snapshots respectively
  Reads the Intermediaries_Proto from its file and populates the global intermediariesList by calling *deserialiseIntermediaires* using Intermediaries_Proto as a parameter
  Reads the Assets_Proto from its file and populates the global assetsList by calling *deserialiseAssets* using Assets_Proto as a parameter
  Reads the HistoricalSnapshots_Proto from its file and populates the global snapshotsList by calling *deserialiseHistoricalSnapshots* using HistoricalSnapshots_Proto as a parameter
  Finally, prints a successful message

- deserialiseBroker:
  Parameter: Intermediary_Proto of type Broker_Proto
  Converts a Broker_Proto into a Broker object and returns it

- deserialiseBank:
  Parameter: Intermediary_Proto of type Bank_Proto
  Converts a Bank_Protointo a Broker object and returns it

- deserialiseMutualFundManager:
  Parameter: Intermediary_Proto of type MutualFundManager_Proto
  Converts a MutualFundManager_Proto into a Broker object and returns it

- deserialiseIntermediaries:
  Parameter: Intermediaries_Proto
  - Creates an empty local indermediariesList
  - Iterates over all Intermediary_Proto in Intermediaries_Proto, for each Intermediary_Proto
    - Check Intermediary_Proto type and add to the local intermediariesList, the returned object from the respective deserialise function

```
if (intermediaryProto.hasBroker()) {
    intermediariesList.add(deserialiseBroker(intermediaryProto));
} else if (intermediaryProto.hasBank()) {
    intermediariesList.add(deserialiseBank(intermediaryProto));
} else if (intermediaryProto.hasMutualFundManager()) {
    intermediariesList.add(deserialiseMutualFundManager(intermediaryProto));
}
```

- deserialiseStock:
  Parameter: Asset_Proto of type Stock_Proto
  Converts the Asset_Proto to a Stock Asset and returns it

- deserialiseBond:
  Parameter: Asset_Proto of type Bond_Proto
  Converts the Asset_Proto to a BondAsset and returns it

- deserialiseMutualFund:
  Parameter: Asset_Proto of type MutualFund_Proto
  Converts the Asset_Proto to a MutualFudn Asset and returns it

- deserialiseAssets:
  Parameter: Assets_Proto
  - Creates an empty local assetsList
  - Iterates over all Asset_Proto in Assets_Proto, for each Asset_Proto
    - Check Asset_Proto type and add to the local assetsList, the returned object from the respective deserialise function

```
for (Asset_Proto assetProto : assetsProto.getAssetsList()) {
    if (assetProto.hasStock()) {
        assetsList.add(deserialiseStock(assetProto));
    } else if (assetProto.hasBond()) {
        assetsList.add(deserialiseBond(assetProto));
    } else if (assetProto.hasMutualFund()) {
        assetsList.add(deserialiseMutualFund(assetProto));
    }
}
```

- processHistoricalSnapshots:
  Iterates over all historical snapshots in snapshotsList. For each snapshot, updates it with the current date and converts it into a Protobuf HistoricalSnapshot_Proto object, which is then added to the global builder for HistoricalSnapshots_Proto.

- processAssetSnapshot:
  Parameter: Assets asset
  Checks the type of the received asset (Stock, Bond, or MutualFund) and calls the respective snapshot processing function either *processStockSnapshot*, *processBondSnapshot*, or *processMutualFundSnapshot* passing on the received parameter onto the called function.

- processStockSnapshot:
  Parameter: Stock
  Builds an AssetSnapshot_Proto of type StockSnapshot_Proto using the builder pattern of protobuf and returns it

- processBondSnapshot:
  Parameter: Bond
  Builds an AssetSnapshot_Proto of type BondSnapshot_Proto using the builder pattern of protobuf and returns it

- processMutualFundSnapshot:
  Parameter: MutualFund
  Builds an AssetSnapshot_Proto of type MutualFundSnapshot_Proto using the builder pattern of protobuf and returns it

- deserialiseHistoricalSnapshots:
  Parameter: HistoricalSnapshots_Proto
  - Creates an empty local snapshotsList
  - Iterates over all HistoricalSnapshot_Proto in HistoricalSnapshots_Proto, for each HistoricalSnapshot_Proto
    - Converts it to its respective historical snapshot type (Stock, Bond, or MutualFund) using *createSnapshotFromAssetProto*, and adds the result to the local snapshotsList. Finally, returns the populated snapshotsList.

- createSnapshotFromAssetProto:
  Parameters: AssetSnapshot_Proto and Date
  Checks the type of the asset snapshot (Stock, Bond, or MutualFund) from the received AssetSnapshot_Proto and calls the respective snapshot creation function either *createStockSnapshot*, *createBondSnapshot*, or *createMutualFundSnapshot*, passing the receiving parameters to the respective function.
  Finally, returns the built HistoricalSnapshots object.

- createStockSnapshot:
  Parameters: StockSnapshot_Proto and Date
  Converts the received StockSnapshot_Proto to a StockSnapshot and returns it.
  Conversion is done using the builder pattern developed in task 2 found in HistoricalSnapshots.

- createBondSnapshot:
  Parameters: BondSnapshot_Proto and Date
  Converts the received BondSnapshot_Proto to a BondSnapshot and returns it.
  Conversion is done using the builder pattern developed in task 2 found in HistoricalSnapshots

- createMutualFundSnapshot:
  Parameters: MutualFundSnapshot_Proto and Date
  Converts the received MutualFundSnapshot_Proto to a MutualFundSnapshot and returns it.
  Conversion is done using the builder pattern developed in task 2 found in HistoricalSnapshots

Asset Viewer

Since I was unsuccessful in setting up Protobuf for Cmake, there is a PsuedoCode.txt file in the Task-3/AssetViewer directory.

This pseudo-code shows a simple framework to the cpp script that would've been used to retrieve the serialised Assets from Task 3b and to process deserialising them to printing them for viewing. The structure follows the logic of Facade.java in Task 3b.

Task 4

The Java implementation for task 4 is found in Java/oopresit/task4/, where in this directory there is the pom.xml file of task4 which has dependencies on all previous Java tasks' pom.xml files and grcp dependencies as needed, additionally, this directory is further split into two subdirectories, src and target.

To run the Java project use Maven: go to the directory Java/oopresit/task4 and write mvn exec:java, you might need to previously run mvn clean install

Proto File

In the src subdirectory, there is another directory called main which then splits into java and proto. The stockService.proto file is found in the proto directory i.e. under Java\oopresit\task4\src\main\proto\.

This task doesn't use the latest Protobuf version 4.28.0 but instead uses Protobuf version 4.27.0 as 4.28.0 still has some issues due to its recent release. Also, uses Grcp version 1.66.0.

stockService.proto

stockService.proto consists of 2 messages and one service, the first message is StockService_Proto which is built using all of the variables normally found in a Stock Asset and additionally the commission that would normally be found with the Intermediary Broker of the Stock.

I.e. StockService_Proto consists of:

- string stockservice_name
- flaot stockservice_value
- float stockservice_quantity
- float stockservice_yield
- float stockservice_intermediary_commission

The second message of stockService.proto is StockService_Return_Proto which stores a variable of type double depicting the annual return of the Stock Asset.

Finally the service, StockService which defines a Remote Procedure Call (RCP) method called CalculateStockReturn that takes StockService_Proto as its parameter and returns StockSerice_Return_Proto

Remote Service

The remote service is split into 3 parts, StockServer.java, StockSvc.java and CLI4.java. Since in task 2, Portfolio.java was removed, the implementation of calculating the annual return is now in CLI4.java.

StockServer.java

StockServer takes care of implementing the grcp server with methods *start*, *stop*, *main* and *blockuntilShutdown*. Additionally an inner child class StockServiceImp, takes care of implementing the grcp service of calculating the stock return which was defined in stockService.proto with a public function *calculateStockReturn* which takes StockService_Proto as a parameter and returns StockService_Return_Proto which depicts the stockReturn of a particular stock..

StockSvc.java

StockSvc acts as an intermediary between StockServer and the main program, manages the connection to the grcp server which should always berunning on port 7000 as it is fixed. StockSvc provides 1 public function *calculateStockReturn* which takes all the parameters necessary to build StockService_Proto which was define din stockService.proto, these parameters are:

- string name
- float value, quantity, yield and commission

After taking these parameters the function will use the builder pattern generated by Protobuf to build StockService_Proto using the parameters received. Once the StockService_Proto is built it is used as a parameter for the function in StockServer.java *calcualteStockReturn* which was explained earlier. Finally, *calculateStockReturn* returns computed annual stock return of a particular stock.

CLI4.java

CLI4.java is an updated version of CLI3.java which was used in task3 but with the following changes:

- CLI4 now initialises a private instance of StockServer
- When the program starts in *InitMenu*() the new function *startServer*() is called to start the grcp server
- When the program is exiting after saving action state the new function *stopServer*() is called to stop the grcp server
- *startServer*(): calls the function *start*() of StockServer that handles turning on the grcp server. This is done using a try and catch, if starting the server fails an error message is printed, and if successful a success message is printed.
- *stopServer*(): checks if the global StockServer instance is not null, if it isn't, calls the *stop*() function of StockServer and then prints a message stating that the grcp server has been stopped,
- *annualReturn*() has been updated to use the rcp method for Assets of type Stock when calculating the annual return:
    - The updated *annualReturn* iterates over all assets in assetsList, for each asset, it checks if it is an instance of Stock, if it is, it uses the rcp method to calculate the annual return. Otherwise, it will use the polymorphic method found in Assets.java

Citations

[1] L. Ardito, R. Coppola, L. Barbato, and D. Verga, "A tool-based perspective on software code maintainability metrics: A systematic literature review," *Scientific Programming*, vol. 2020, Art. no. 8840389, 2020.

[2] J. Petke, D. Clark, and W. B. Langdon, "Software robustness: a survey, a theory, and prospects," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, Aug. 2021, pp. 1475–1478.