DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
UNIVERSITY OF NEBRASKA—LINCOLN

One Basket Financial Management System

Computer Science II Project

[Zeyuan Kong & Vinay Singh]
[4/18/17]
[Document Version 4]

Revision History

Version	Description of Change(s)	Author(s)	Date
1.0	Initial Design of Classes	Vinay Singh & Zeyuan Kong	2017/02/01
2.0	Added Functionality to Classes	Vinay Singh & Zeyuan Kong	2017/02/16
3.0	Added Database Documentation, Completed Design Description, Added to Definitions & Abbreviations, Updated bibliography	Vinay Singh & Zeyuan Kong	2017/04/02
4.0	Completed Database Documentation, Revised Component Description, Revised Design & Integration of Data Structures	Vinay Singh	2017/04/18

Contents

1. Introduction

This financial management system is implemented to replace the aging AS400 green-screen system that is currently being used by One Basket Finance. This updated software is written in Java, object-oriented, database-backed and conforms to the business model necessities of the aforementioned organization.

To conform to the object oriented paradigm and business model, every aspect of the business is modeled using objects. For example, assets and clients.

1.1 Purpose of this Document

The purpose of this document is to explain the whole project in depth. This includes the design process, functionality and applications.

1.2 Scope of the Project

This management system will be used to organize data efficiently for the client by performing various needs according to the specifications of the client. First, as this project is an upgrade to the old system, the user would be able to import data from the old system which is extremely important. In addition to that, the user would be able to create entries of users and assign specific assets and various other information. As this is an object oriented design, the assets too would be specific objects which have different pieces of data (APY, value, etc). Therefore, the user would be able to modify information about those assets (if needed) through this application as well.

1.3 Definitions, Acronyms, Abbreviations

1.3.1 Definitions

methods-functions that specific blocks of code perform

get() methods- code that extracts specific information from an object

Class - A piece of code that represents a real world entity (eg- assets, clients)

Object Oriented Programming-A programming paradigm where problems are modeled around objects as opposed to procedural or functional.

XStream-Library used to generate files in the XML format.

Gson-Library used to generate JSON files.

amountOwned-Variable used with asset classes to quantify the value of that particular asset within a portfolio.

result set-A collection that contains the output from a query to a database.

prepared statement-A processed SQL statement used to maintain security and integrity of database.

1.3.2 Abbreviations & Acronyms

OOP– Object-oriented Programming

API– Application Programming Interface

APY-Annual Percentage Yield

JSON-Javascript Object Notation

XML-Extensible Markup Language

SQL-Structured Query Language

ADT-Abstract Data Type

2. Overall Design Description

As stated in the introduction, this is an object-oriented application. Therefore, in the design process, we adhere to the four pillars of OOP, that is Encapsulation, Inheritance, Abstraction and Polymorphism. First, we implement encapsulation by modelling assets and clients as separate objects. Besides that, we adhere to abstraction by hiding specific calculations(eg-risk measures) within the asset objects and only provide get() methods to extract those calculated values. In addition to that, we also adhere to the remaining two pillars of OOP and will be discussed further down.

We modeled the design of this application using various classes to represent various entities such as Person, Asset, Address, and various other classes that provide functionality to the application.

To connect to the MySQL database, we use an API, specifically JDBC to cater to our needs. This is implemented through our Java class called DatabaseAbstraction. The tables within the database are related to each other by making use of foreign keys.

Errors encountered within the application are logged using the built in Java Logger class which suppresses unnecessary outputs.

2.1 Alternative Design Options

The initial plan was for each asset only to have specific information about its general state. For example, the asset objects were only supposed to contain general information such as name, type, value, base risk, and annual return rate while allowing the portfolio objects to handle how those assets behave within specific portfolios (differing stake, aggregate risk, total annual return, and etc). This would have made the design negligibly simpler but in the process, break encapsulation. Therefore, we decided to

include copy constructors in each asset so that they may be copied with additional specific information that would pertain to specific portfolios. For example, portfolio A and B both have a stake of ownership of private investment C with the amounts of 40% and 60% respectively. In this case, we make 2 copies of private investment C with the amountOwned variables set to 0.4 and 0.6 respectively to indicate its value.

3. Detailed Component Description

To accurately model the financial management system, different classes are used. For example, for the human component of the operation, a Person class exists that extends to a subclass called Broker to model the existence of some people who are brokers and some who are not. To model the business aspect of the system, the application has a Portfolio class to represent portfolios, and an abstract class, Asset that extends to the respective assets the company deals with-Stocks, PrivateInvestment, and DepositAccount. In addition to that, we have a plethora of classes that contribute to an application that adheres to the OOP paradigm which will be discussed below.

3.1 Class/Entity Model

We use objects to model real world entities. When it comes to objects in this project, it refers to assets and clients. In the case of assets, classifying all of them into one class would not be a good idea as there are various different types of assets. Therefore, we extend the asset class to model specific sub-classes such as Deposit Accounts, Stocks and Private Investments to account for data that is exclusive while still inheriting general data that comes with all assets (eg-broker and risk). This fulfills the inheritance pillar of OOP. In addition to that, the Person class has a constructor to create person objects with relevant information of a person such as id, first name, last name, address, and email. To model brokers, we extend the person class to a subclass called broker which requires 2 additional inputs-broker type and Identifier.

Also, we provide risk measure methods for each of these three different kinds of assets. The beta measure is use for the stocks; the omega measure is for measuring the private investments; the aggregate risk measure for individual portfolio. (Note that the aggregate risk measure is the weighted sum of all risk rate to their assets)

Assets have various return rates that depend on their type, therefore we designed different functions in the class. The deposit accounts is based on different period of time the asset saved.(annually, monthly, and daily); Stocks and Private investments have a fundamental return rate that depend on how much the asset is expected to increase on an annual period.

Apart from the aforementioned classes, we also designed additional classes that perform specific functions which are summarized below:

PortfolioReport

The class that contains the main method which loads the portfolios from the database using the DatabaseRetriever class.

DataConverter

In phase 1 of the project, this was the superclass that did all the work of pulling in data from flat files, creating lists of assets and people, and output the results to JSON and XML format files. For subsequent phases, it was tweaked to pull data from a SQL database.

DataConverterObject

To reverse the poor design of phase 1, this class creates an object based on DataConverter that contains lists of addresses, people and assets to be used in other parts of the application. It also contains a constructor that takes in 3 lists to form the object.

FileIn

Handles all parts of the application related to reading in of flat files. It has an input method that takes a filename as a string argument and outputs a list of content from the flat file. Each entry of the list corresponds to a line of data.

FileOut

Handles application output. The write method takes in a list of objects, primarily portfolios and writes the relevant data in a well-formatted manner to the standard output.

DatabaseInfo

Contains information pertaining to database connection.

DatabaseAbstraction

Factory method that handles connecting and disconnecting from a database. The connect method connects to the database using credentials provided in DatabaseInfo and returns a connection.
The disconnect method takes in a connection, result set and prepared statement and in sequence, terminates them.

DatabaseRetriever

Class that handles interactions with the database to return portfolios in an arraylist. To do this, the class contains its required helper methods that interact with the database to return specific portfolio

components. Examples include getAddress, getAssets, getPerson and getEmail which are all called by its getPortfolio method.

Address

Contains specific components of an address-street,city,state,zipcode,country to maintain formatting of data.

Name Comparator

Implements the comparator interface to specifically compare two portfolios using their owners' last name/first name and return a value corresponding to the appropriate ordering.

ManagerComparator

Implements the comparator interface to specifically compare two portfolios using their manager type (Expert or Junior) followed by the owners' first name and last name. The comparator then returns a value corresponding to the appropriate ordering.

ValueComparator

Implements the comparator interface to specifically compare two portfolios using their total value. The comparator then returns a value corresponding to the appropriate ordering.

PortfolioList

This is an implementation of our Sorted ADT. Instead of sorting itself at every insertion, this data structure is designed to maintain ordering. It is a parameterized implementation of the Iterable interface, meaning that it can hold any type of objects.

PortfolioData

This is the class that is designed to provide an interface to the application and user. It is composed of various methods such as getPortfolio(), getAsset(), addPortfolio, addAsset, removePeson, removeAllPersons, removePortfolio, remoeAllPortfolios, etc which form the basis of interaction with the appication.

In a nutshell, the design of our application is accurately modeled by the Unified Modeling Language (UML) diagram in figure 1.

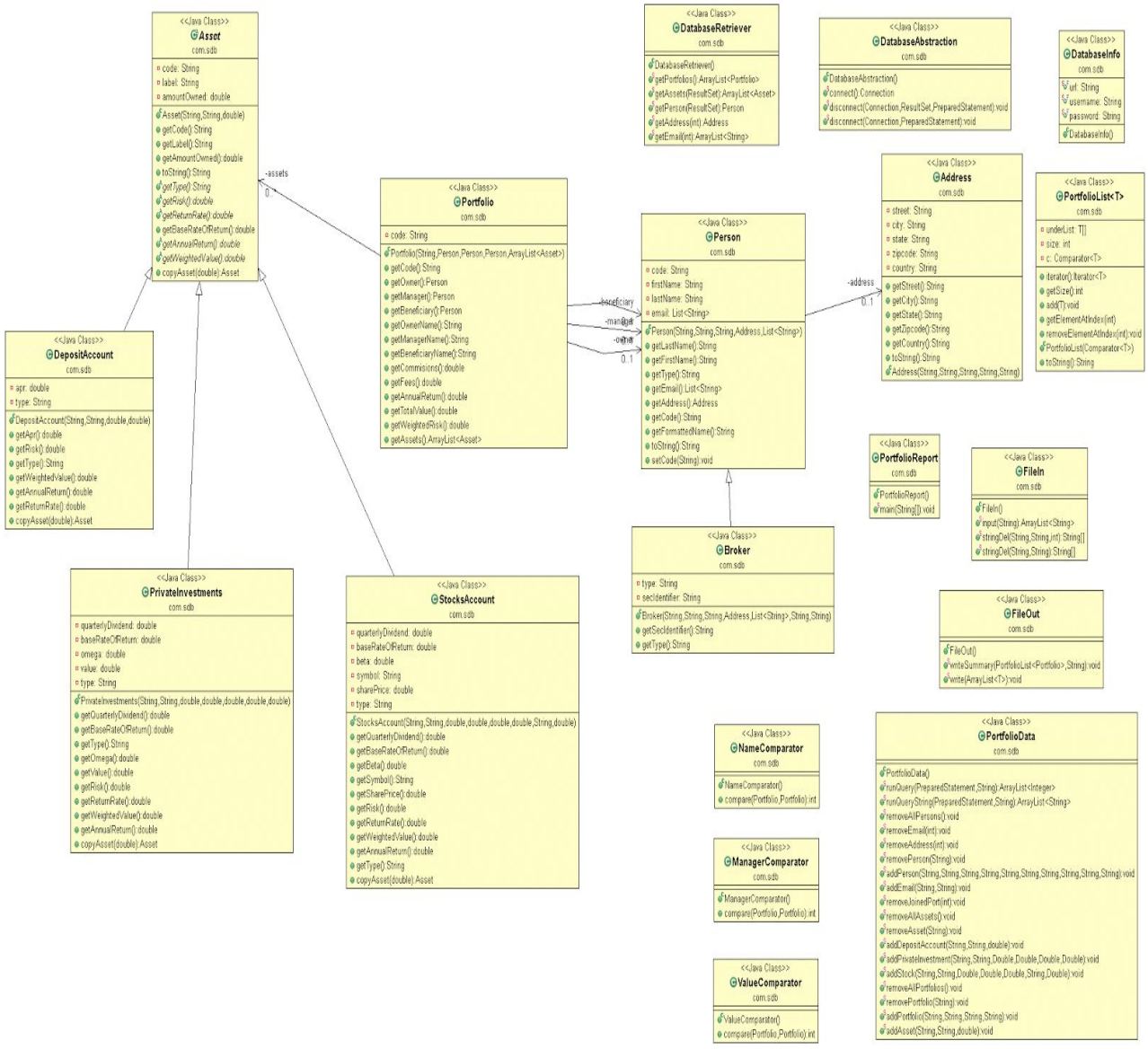

Figure 1: UML of the One Basket Financial Management System

3.1.1 Component Testing Strategy

Predefined test cases were created with a plethora of possible boundary cases and combinations to assure a robust application. To test the construction of appropriate asset and person objects, the application was initially designed to output the relevant objects to JSON and XML formatted flat files.

To test the accurate construction of portfolios, The application output summaries of each portfolio to

the standard output with comprehensive information such as total annual return, total risk, owner, beneficiary, manager, fees, and commissions. This brought to light many bugs in the manner we handled annual return and risk within the private investments and stocks which were inaccurate. This method of testing allowed us to identify every flaw in our system and apply the appropriate fix.

3.2 Database Design

The database was designed based on Relational database. Eight tables was created for this project according to the requirement (see figure 2). To be specific, We used a MySQL database.

Person, Email, and Address

Person contains personId, lastName, firstName, personCode, secIden (second Identification), and the brokerType. By using the foreign key personId, both Email and Address tables contain information for any specific person. The table, Email, contains emailId, address, and PersonId; Address table contains values such as addressId, street, city, stateId, zipcode, and contryId. Notice that the countryId and StateId are foreign key links to Country and State.

Portfolio

As you might noticed in the previous section, everyone who is an asset owner in the data record has a Portfolio for them. In the Portfolio table, there are values portfolioId, portfolioCode, ownerId, managerId, and beneficiaryId. OwnerId, managerId and beneficiaryId are foreign keys that link to personId in the Person table.

JoinedPort

The database might be slightly different from the previous design partly due to the structural difference. In the old system, every single line of the Portfolio.dat were comprised of several assets. In order to facilitate management in a relational database, we create the JoinedPort. The JoinedPort connects the Portfolio to the Assets that related to the owner. For example, the asset owner Andy has three assets, three JoinedPorts will be created for the Portfolio. Each JoinedPort holds the JointedPortId as always, assetId (Foreign Key links to asset) ,portfolioId (Foreign Key links to the Portfolio), and the amountOwned.

Asset

We created an Asset table with 10 values for three different types of asset. They are type, name, assetCode,baseRteOfReturn, quarterlyDividend, sharePrice, beta, omega, value, and apr. These fields

may be null to facilitate the different types of assets that hold different combinations of the aforementioned fields. Through linking JoinedPort by assetId, we can access every asset by knowing what the JoinedPort is.

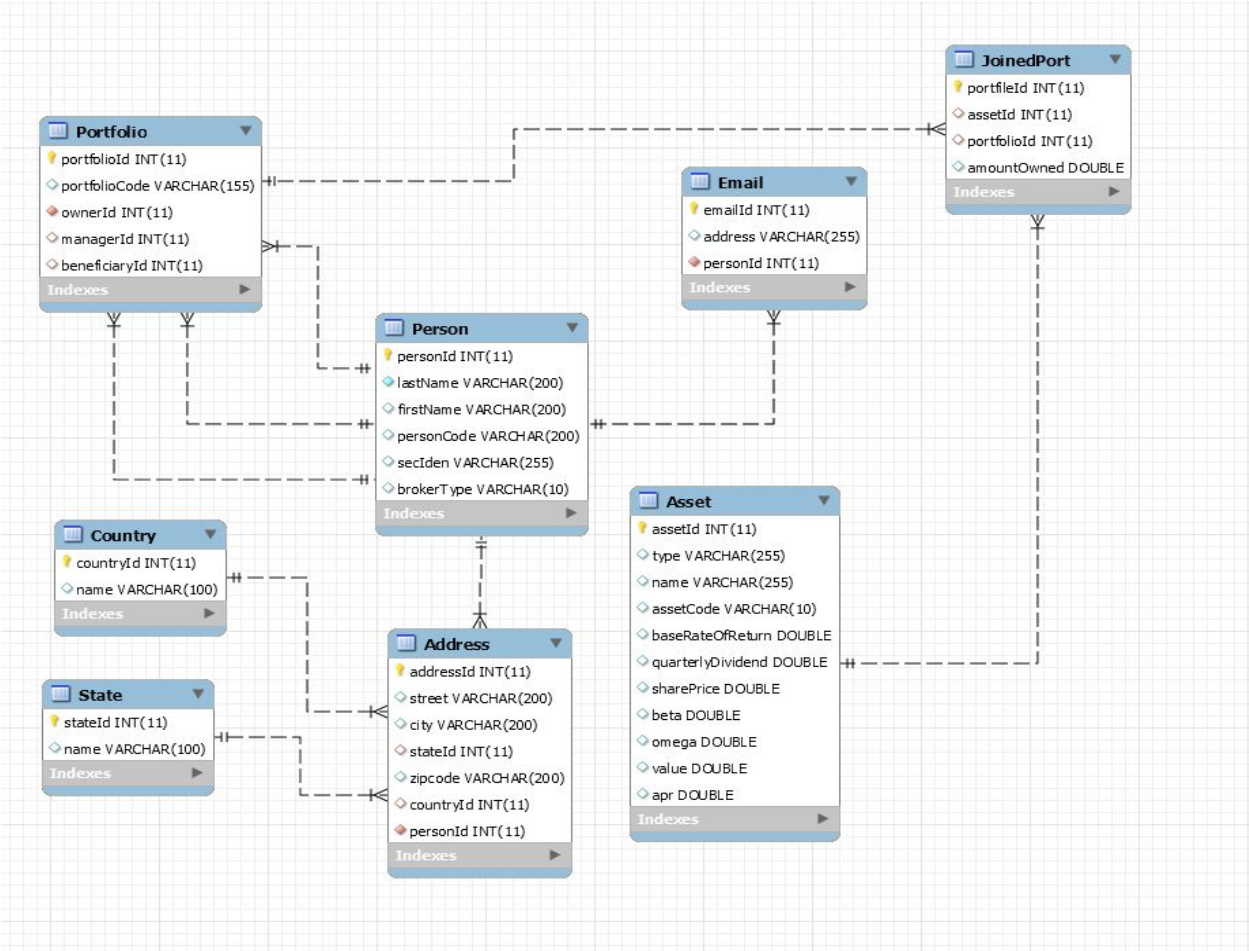

Figure 2: Blueprint for the Database of One Basket Financial Management System

3.2.1 Component Testing Strategy

Similarly, we use the predefined data that contain different conditions in this phase of testing. Two test used query files was created. One was created to store the data in database, the other was several lines of queries that manipulates the data. Again, the printing of the data to the standard output was ensured to enable the developer to identify discrepancies with predefined test cases.

3.3 Database Interface

Java provides rich APIs for developers, and in this application, JDBC is used for the Database Interface. The class DatabaseInfo contains the connection information. By creating an object of DatabaseInfo, the class DatabaseAbstraction makes a connection to the database. an object of DatabaseAbstraction at this point has defined by Dataconverter. Finally, in the DataConverter, several functions were called to manipulate the data. Addidtionally, the PortfolioData class was created that contains many additional methods that allow seamless interaction with the database. Essential methods such as adding and removing assets, persons and portfolios are all implemented through specific methods in this class. It was designed to be an interface between the application and users or developers for further development and use, such as building a front-end user interface.

3.3.1 Component Testing Strategy

Several queries are used in the DataConverter to test complete functionality. Apart from that, an online webgrader was set up to test a variety of instructions on our database. These include but are not limited to removing all objects from our database and reloading new data, all through the implemented database interface. Accurate summaries of corresponding portfolios were the indicator of accuracy.

3.4 Design & Integration of Data Structures

The primary data structures that were used in this application are arrayLists and regular arrays.

The arrayLists were integrated to specific objects which resulted in the application containing lists that contained assets, persons, addresses, portfolios or regular strings. The biggest benefit to using arrayLists is the fact that arrayLists are adaptable to the number of inputs. As the number of data entries could vary greatly, arrayLists proved to be the ideal solution.

Regular arrays containing strings were initially used when dealing with data from a flat file. The primary purpose was so that we could split the text inputs by specific delimiters while still keeping track of the number of inputs. For example, a person's data from a flat file may or may not contain email addresses. To account for that, we created an array with the number of spaces that corresponds to the expected number of inputs. Therefore, if the specific index where emails are usually stored was empty, we can assume that the person in question does not have an email address.

In addition to the previously mentioned data structures, the PortfolioList class was also created to impose ordering on portfolios within the application. The class is made generic so that it may hold any types of objects. In addition to that, it implements the Comparator interface which allows the data structure to be used in an enhanced for-loop. In order for the list to impose a specific ordering, it has to be provided with a Comparator at instantiation. Within this application, it is instantiated with 3 different comparators, namely the NameComparator which orders based on last name/first name of the owner, ValueComparator which maintains order based on total value of the portfolios and lastly the

ManagerComparator which orders by manager type and owner name.

3.4.1 Component Testing Strategy

Similar to the testing in 3.1, predefined test cases were created and the application was designed to output the person and asset objects to JSON and XML files where we could manually trace accuracy. In addition to that, the summary of portfolios as mentioned in 3.1 was also an indicator of accuracy.

3.5 Changes & Refactoring

From the first phase of the project, the DataConverter superclass was redesigned by extracting the input and output classes (FileIn,FileOut) to separate independent classes to ensure that the application adheres to the OOP paradigm.

Initially, the application was pulling in all the person, asset and portfolio entries, creating appropriate lists for them and subsequently creating the portfolios as needed from those lists. The problem with that design was the fact that it was defeating the purpose of using a database. Therefore, the application was updated to only create objects (persons, assets, etc) that are needed to construct specific portfolio objects.

4. Bibliography

[1] Eckel, B. (2006). *Thinking in Java* (4th ed.). Prentice Hall.

[2]About GSON.(2014, January 25). Retrieved January 30, 2017, from GSON: https://code.google.com/p/google-gson/

[3] About XStream. (2014, January 25). Retrieved January 30, 2017, fromXStream: http://www.xstream.codehaus.org/