|  |
| --- |
| Department of computer science & Engineering  University of Nebraska—Lincoln |
| One Basket Finance Project |
| Computer Science II Project |
|  |
| **Jared Ott and Ryan Wallace** |
| **3/3/2017**  **Version 0.3** |

|  |
| --- |
| This document contains up to date information on the application for One Basket Finance. The goal of the application is to develop an effective application by using Object-Oriented Programming that can meet all of One Basket Finance’s functional needs. |

# Revision History

|  |  |  |  |
| --- | --- | --- | --- |
| Version | Description of Change(s) | Author(s) | Date |
| 0.1 | Initial draft of this design document | Ryan Wallace/Jared Ott | 2017/02/01 |
| 0.2 | First revision  Added XMLable and JSONable interfaces  Added Portfolio Class and description  Included more definitions  Included more abbreviations  Were more descriptive | Jared Ott/Ryan Wallace | 2017/15/02 |
| 0.3 | Revised Class information  Removed XMLable and JSONable interfaces  Added robust database design | Ryan Wallace/Jared Ott | 2017/03/01 |
| 0.4 | Revised database information, added functionality of database and tables. | Jared Ott/Ryan Wallace | 2017/3/14 |
| 1.0 | Finalized product. Completely organized and polished everything for 100% greatness | Ryan Wallace/Jared Ott | 2017/4/20 |

Contents

[Revision History 1](#_Toc474954047)

[1. Introduction 4](#_Toc474954048)

[Purpose of this Document 4](#_Toc474954049)

[1.2 Scope of the Project 4](#_Toc474954050)

[1.3 Definitions, Acronyms, Abbreviations 4](#_Toc474954051)

[1.3.1 Definitions 4](#_Toc474954052)

[1.3.2 Abbreviations & Acronyms 5](#_Toc474954053)

[2. Overall Design Description 5](#_Toc474954054)

[2.1 Alternative Design Options 5](#_Toc474954055)

[3. Detailed Component Description 5](#_Toc474954056)

[3.1 Database Design 5](#_Toc474954057)

[3.1.1 Component Testing Strategy 7](#_Toc474954058)

[3.2 Class/Entity Model 7](#_Toc474954059)

[3.2.1 Component Testing Strategy 9](#_Toc474954060)

[3.3 Database Interface 9](#_Toc474954061)

[3.3.1 Component Testing Strategy 10](#_Toc474954062)

[3.4 Design & Integration of Data Structures 10](#_Toc474954063)

[3.4.1 Component Testing Strategy 11](#_Toc474954064)

[3.5 Changes & Refactoring 11](#_Toc474954065)

[4. Additional Material 11](#_Toc474954066)

[4.1 External Libraries 11](#_Toc474954067)

[5. Bibliography 11](#_Toc474954068)

# Introduction

One Basket Finance needs a better way to keep track of their assets, portfolios, and the people that own and manage them. It is the job of Task Force .JAR to create a program that makes this task easy and automated for our client. At this phase in the production, the program is centered around class construction and implementation, as well as outputting the objects in human readable formats. The end goal of the production is to have a database to easily and efficiently keep track of the assets.

## Purpose of this Document

The purpose of this document is to effectively and clearly communicate how the program works and is designed at a high level. This document provides basic information about the program and it’s working parts.

## Scope of the Project

The task is to replace One Basket Finance’s previous financial application. This involves reading in the data from the previous program and storing it in a database. Furthermore, the goal is to create methods and functions that aid in accessing the data, changing it, and performing other operations that return valuable information to the operator. Our plain is to automate the creation and storage of assets and portfolios and all that acts upon them.

## Definitions, Acronyms, Abbreviations

### Definitions

* Abstract Class – a way to group classes into an object and assign abstract functions to be implemented in its subclasses.
* Base Rate of Return – The gain or loss on an investment over a period of time.
* Beta Measure – The measure of relative risk for highly traded commodities I.e. stocks. It measures the return of a certain stock against the return of the overall market (usually an index like the S&P 500) using the Covariance (asset, market) divided by the variance of (market).
* Class – a way in Java to group information of a real-world object into a computable Object.
* Encapsulation – the bundling of data with the methods operating on that data.
* Instance – a single implementation of an object.
* Interface – a purely abstract class used to group classes into broad groups and still assign abstract functions to be implemented.
* Object – a grouping of data in programming that can be implemented with multiple instances. This is the basis of Object-Oriented Programming.
* Object Oriented Programming – a programming paradigm that predominantly uses Objects.
* Omega Measure – Assesses the risk of private investments. For the use of this project we will represent final risk as a 0 or 1 with 0 being low risk and 1 being risky.
* Polymorphism – the ability to classify a subclass as its superclass.
* Quarterly Dividend – The money dealt out to shareholders of a company every fiscal quarter.
* SEC Identifier Number – Takes the form of sec###. This number is used to identify and verify brokers.
* Subclass – a class that inherits fields and methods of the superclass, or parent class.

### Abbreviations & Acronyms Alphabetically

* APR – Annual Percentage Rate
* char – a single character
* CRUD – Create, Retrieve, Update, and Destroy
* FK – Foreign Key
* JDBC – Java Database Connectivity
* OOP – Object Oriented Programming
* PK – Primary Key
* SQL – Structured Query Language
* varchar – an array of chars

## Overall Design Description

The project includes the implementation of multiple classes that represent real world objects, and it is the goal to provide an interface to interact with the objects and provide methods that help the user access the information about these objects. As previously stated, the goal of the project is to (eventually) store information into an optimized database for easy automated information retrieval. At this point in production, we have implemented class-based organization of elements provided using Object Oriented Programming. The design is simple yet efficient: take in and process a \*.dat file, convert the information into classes, and output those classes into a human readable formatted \*.txt file.

## Alternative Design Options

Our solution to One Basket Finance’s needs is centered around Object Oriented Programming. The best way to represent real life entities through programming is by using objects. The Asset class was chosen to be defined as an abstract class to prevent anyone from trying to instantiate an “asset,” for there is no such thing. We could have made Asset an interface, however this would have taken away some of its functionality as an abstract class, such as defining a base constructor. Only the subclasses are actual real world entities, while they are all under the broad category of “asset.” Furthermore, some variables that may be represented as integers were chosen to be represented with Strings because there may be varying input that include non-integers. For example, the ZIP code *could* have been represented with an integer, but it was chosen to represent zip code as a String because it could be the ZIP code +4 which usually includes a hyphen. This should create no problem because no integer operations should be performed on the ZIP code, and it will mainly be used to print out an address.

# Detailed Component Description

Section 3 details the structure design of the program. This includes databases, classes, and their methods. This section’s goal is to explicitly outline the program and allow easy identification of how the program’s components fit together.

## Database Design

Table 1 – Person : Holds person information

* personID integer [PK]
* personCode varchar(20)
* lastName varchar(255)
* firstName varchar(255)
* addressID integer [FK] (Table 3)

Table 1.1 – BrokerStatus : Holds info if Person is

a Broker

* brokerId integer [PK]
* brokerType char
* secId varchar(10)
* personId integer [FK] (Table 1)

Table 2 – Asset : Holds info of 3 different types

of assets.

* assetID integer [PK]
* assetType char
* assetCode varchar(20)
* apr float
* label varchar(255)
* quarterlyDividend double
* rateOfReturn float
* risk float
* symbol varchar(5)
* value double

Table 3 – Address : Basic address information

* addressID integer [PK]
* streetAddress varchar(255)
* zipCode varchar(10)
* city varchar(255)
* stateID integer [FK] (Table 3.1)

Table 3.1 – State : State/province within a

country

* stateID integer [PK]
* name varchar(255)
* countryID integer [FK] (Table 3.2)

Table 3.2 – Country : A country

* countryID integer [PK]
* name varchar(75)

Table 4 – Portfolio : Basic portfolio information

* portfolioID integer [PK]
* ownerID integer [FK] (Table 1)
* brokerID integer [FK] (Table 1)
* beneficiaryID integer [FK] (Table 1)
* title varchar(100)

Table 5 – Email : Email address

* emailID integer [PK]
* address varchar(255)

Join Table 1 – AssetPortfolio : Joins Asset to

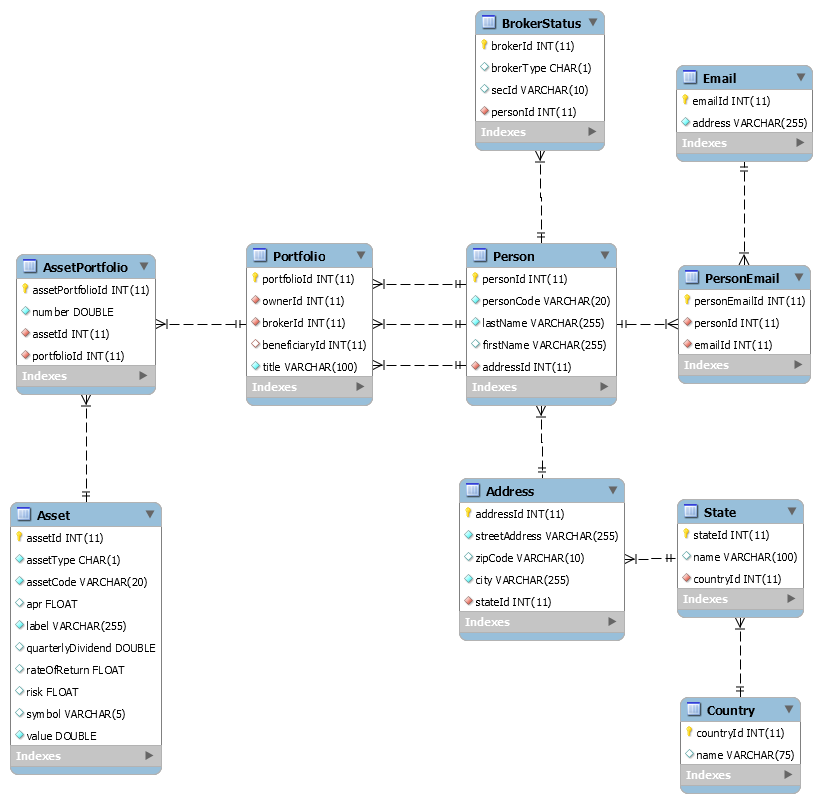
Portfolio with a many to many relationship

* assetPortfolioID integer [PK]
* number float
* assetID integer [FK] (Table 2)
* portfolioID integer [FK] (Table 4)

Join Table 2 – PersonEmail : Joins Person to

Email with many to many relationship

* personEmailID integer [PK]
* personID integer [FK] (Table 1)
* emailID integer [FK] (Table 5)



**Figure 1: Above is a diagram of the SQL database table structure. A connection with one prong represents an n-to-1 relationship, while a connection with multiple prongs represents n-to-many relationships.**

Many ideas lead to the design choices made to make this database work as well as possible for its intended purpose. For instance, join tables were chosen above foreign keys to facilitate a many to many relationship between Portfolio and Asset as well as between Emails and Person. For AssetPortfolio, the decision mainly allowed for different portfolios to have different number of shares in the same stock, to keep the uniqueness of the Asset table. Similarly, PersonEmail allows multiple people to share the same email in the system and retain uniqueness.

Some fields also are not allowed to be null, such as person/asset/portfolioCode or lastName in Person. This is represented by a filled diamond next to the field in figure 1. In the case of names stored in the many of the tables, it is reasonable to assume every person or country (etc.) has a name, which allows for easier data retrieval and simply makes sense. A Portfolio, on the other hand, has ownerId and managerId not nullable, but beneficiaryId nullable. This is because a Portfolio absolutely must have an owner and manager linked to it, but doesn’t particularly need a beneficiary listed with it.

### Component Testing Strategy

This component can be easily tested by running queries using MySQLWorkbench to troubleshoot the tables and the info contained therein. This consists of creating, retrieving, updating, and destroying data to ensure everything works in the tables as intended.

## Class/Entity Model

Class 0 – Main Class

* Contains main method. Uses all other classes in some way to get the job done.

Class 1 – Person

* Person Code a String
  + Unique designation (from the old system)
* First name as String
* Last name as String
* Address as Address
  + (see class 3)
* (optional) Email address or multiple email addresses, stored in an ArrayList of Strings.

Subclass 1.1 – Broker

* + Includes Broker status as BrokerType
    - Enumerated type which indicates Junior or Expert
  + Broker’s SEC identifier as String.

Class 2 – Asset (Abstract Class)

* Label as String
* Code as String

Subclass 2.1 – Deposit Account

* + APR as double

Subclass 2.2 – Stock

* + Quarterly Dividend as double
  + Base Rate of Return as double
  + Beta Measure as double
  + Stock Symbol as String
  + Share Price as double

Subclass 2.3 – Private Investments

* + Quarterly Dividend as double
  + Base Rate of Return as double
  + Omega Measure as double
  + Total value as double

Class 3 – Address

* Street address as String
* City as String
* State as String

(Class 3 – Address)

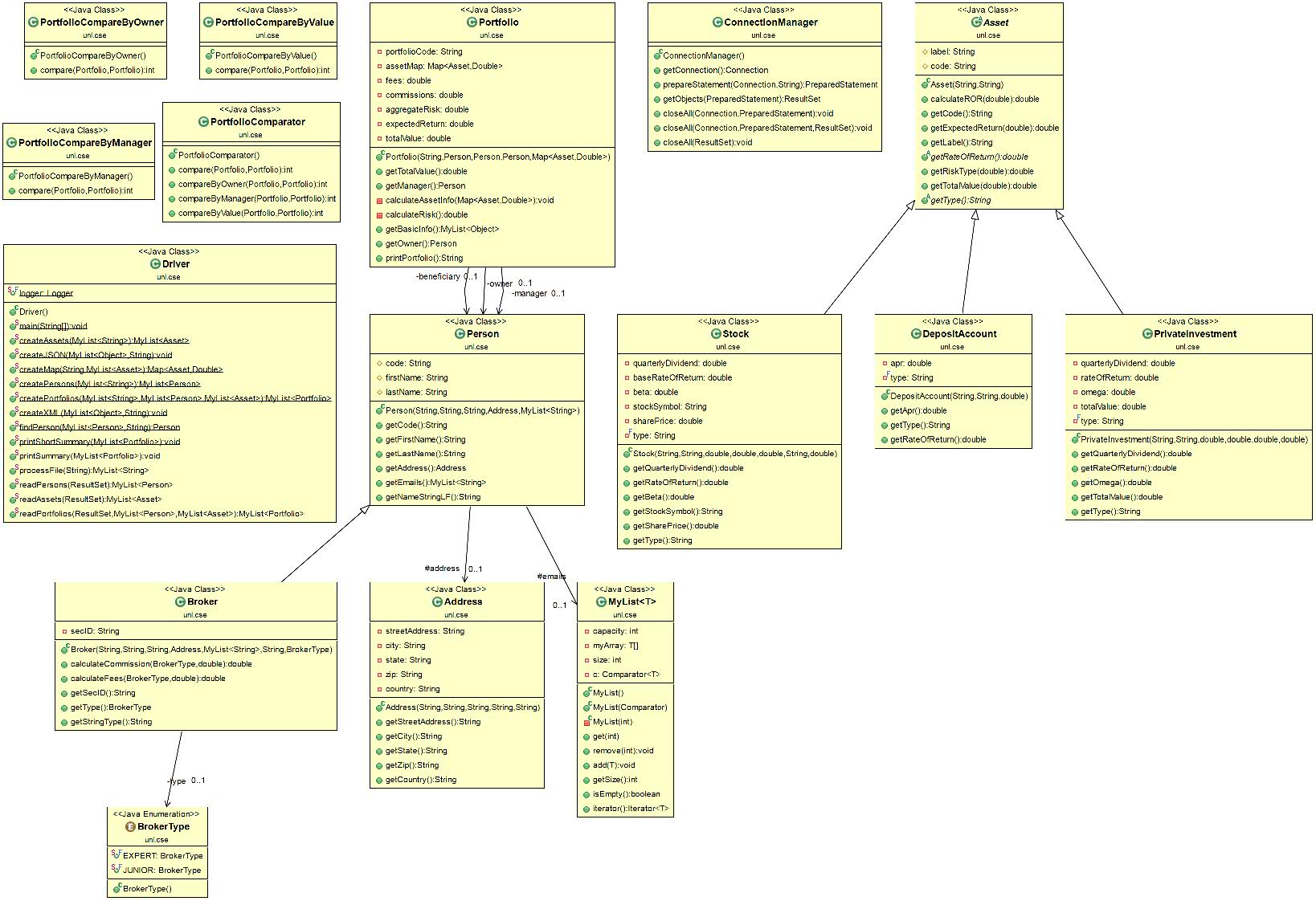
* ZIP as String
* Country as String

Class 4 – Portfolio

* Portfolio Code as String
  + Uniquely identifies portfolio.
* Owner as a Person
  + Corresponds to the Person who owns the account.
* Manager as a Person
  + Corresponds to the Person who manages account.
* Beneficiary as a Person (Optional)
  + Corresponds to the designated beneficiary for the portfolio.
* Asset map as a Map mapping Assets to Doubles
  + Maps the assets to the special provided numbers.

Class 5 – Portfolio Comparator

* Allows that comparison of two portfolios for ordering the output



**Figure 2: Above is a Class diagram of the class structure. An arrow with a white head represents inheritance, while an arrow with a black head represents composition.**

### Component Testing Strategy

For this component, the best testing strategy is to provide the program \*.dat files as input and make sure the program outputs a well-formatted \*.txt file. The idea is for all the portfolios, persons, and assets to print as they are intended. If it doesn’t output correctly or if something went wrong, some re-implementation of the methods is necessary until it does as is expected and as it should. As we have come further, the option of connecting to the database to use the fields as inputs also opens to double check the class functionality.

## Database Interface

The program allows for the use of the previously implemented classes to interact with the SQL database’s tables by using JDBC. This entails a use of JDBC’s PreparedStatement class used with Connection and ResultSet for optimal CRUD. A ConnectionManager class was also created for use of JDBC’s classes in a more streamlined and efficient manner. To facilitate better error practices, a Logger method was also used, which details what happened when something goes wrong.

### Component Testing Strategy

Testing this is simply a matter of running it with queries that should or shouldn’t work. If they return the proper values, it works correctly; however, if it outputs to the logger, it is easy to find out what went wrong, then fix and work from there.

## Design & Integration of Data Structures

In addition to using Java’s HashMap to keep track of the assets, an ADT class, “MyList<T>”, was created to organize and sort the information with which the program is working. Three Comparators were also created for the ordering of Portfolios by Owner, Manager, and total Value. These Comparators are used to keep the “list” ordered at all times, thus no sort method is needed.

### Component Testing Strategy

Simply printing out the data and making sure it is ordered properly was enough to test these data structures.

## Changes & Refactoring

Originally, the intended purpose was to work with \*.dat files and output XML and JSON formatted class strings. We continued by organizing these into aesthetically pleasing portfolio summaries. We then moved on to allow for database connectivity, greatly increasing the efficiency of creating, updating, retrieving, and deleting these records. Finally, we implemented JDBC in order to connect the two parts in harmony.

# Additional Material

## External Libraries

We imported a few external libraries to make our XML and JSON output more automated. These libraries are:

* Jackson-annotations-2.8.0.jar
* Jackson-core-2.8.6.jar
* Jackson-databind-2.8.6.jar
* Jackson-dataformat-xml-2.8.6.jar
* Jackson-module-jaxb-annotations-2.8.6.jar
* Stax2-api-3.0.1.jar
* Woodstox-core-asl-4.2.0.jar
* DataConverter.jar
* log4j-1.2.17.jar
* mysql-connector-java-5.1.41-bin.jar

The Jackson, Stax, and Woodstock jar files allow for easy exportation of our objects into machine readable formats i.e. XML and JSON. The log4j jar was used to log errors, and the mysql jar was integral to connecting an persisting our database.

# Bibliography

[1] Bourke, C. (2017). *Computer Science II,* <http://cse.unl.edu/~cbourke/CSCE156/>