DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
UNIVERSITY OF NEBRASKA—LINCOLN

TBF Financial Management System

Computer Science II Project

Aime Nishimwe

3/5/20

Document Version 1.2

This project is a replacement to an aging AS400 green-screen system that Too Big to Fail Financial currently uses. .

Revision History

The table below documents significant changes that take place throughout developing the system.

Version	Description of Change(s)	Author(s)	Date
1.0	The initial draft of this design document	Aime Nishimwe & Kalen Walin	2020/02/05
1.1	Applied feedback for the initial draft	Aime Nishimwe	2020/02/19
1.2	Added database design and revised OOP design	Aime Nishimwe	2020/03/05

Table of Contents

1. Introduction

This project is a replacement to an aging AS400 green-screen system that Too Big to Fail Financial currently uses. This financial management system is an object-oriented program written in Java. This financial management system supports TBF's business model by implementing its business rules and providing the functionality

1.1 Purpose of this Document

The purpose of this document is to outline the design of this financial management system. This document intends to serve as a guide to reproduce the financial management system with identical functionality without having access to the code base.

1.2 Scope of the Project

The TBF financial management system stores three types of assets. These assets include Deposit Accounts, Stocks, and Private Investments. Each consists of various pieces of data that define their value, annual rate of return, and a measure of risk. Each Asset also has a unique alpha-numeric code that identifies it (in the old AS400 system) and a label to describe what the Asset is. The new financial management system replaces TBF's old AS400 green-screen system.

1.3 Definitions, Acronyms, Abbreviations

1.3.1 Definitions

Definitions to be added in subsequent phases.

1.3.2 Abbreviations & Acronyms

APR: Annual Percentage Rate

CSV: Comma-Separated Values

JSON : JavaScript Object Notation

XML: Extensible Markup Language

SQL: Structured Query Language

TBF: Too Big to Fail

2. Overall Design Description

This financial management system is built upon object-oriented programming. The objects model real-world entities, which are the building blocks of the system. Data from a flat CSV file will be converted into objects through a java program. These objects are exported to JSON and XML files. A summary report will be created that combines the pieces of data stored in these objects.

This financial management system connects to a MySQL database to handle data storage.

More details to be added in subsequent phases.

2.1 Alternative Design Options

The version 1.0 of the program has a main class (DataConverter, driver class) with a massive amount of code to parse data from flat files. The parser functions in the driver class were not reusable, and it introduced the yo-yo antipattern problem where one has to loop up and down within a driver class to fix bugs and make some modifications. Therefore, a general parser class is created, and it has type-specific subclasses to parse assets and person files. Besides, the parser class extends Program Output class that gives it the ability to dump various class instances to a JSON or XML file.

3. Detailed Component Description

This java program uses multiple classes and subclasses to store information about different elements of the financial management system. There is a main driver class called DataConverter that implements ProgramIO or program input/output entity to parse raw asset and person data, create instances of appropriate classes, and export data to platform-independent data formats. A more detailed explanation of the interactions and relationships between different entities is explained in section 3.2.

3.1 Database Design

Details will be added in a subsequent phase.

3.1.1 Component Testing Strategy

Details about database testing will be added in a subsequent phase

3.2 Class/Entity Model

The system design follows object-oriented programming principles, and it is designed to be a new multi-tier system to support TBF's business model by implementing overall business rules and functionality. Different classes are designed to model different entities present in the system. Since TBF manages portfolios of various assets for their clients, Asset and People/Person are the main entities of the system.

The the overall interaction between both of these main classes as well as other supporting and system driver class. ProgramIO is a general and stand-alone entity in the system that implements interfaces and classes to read raw Asset and Person data from a file and to dump the output or instances of Assets and Person classes to files in either JSON or XML, platform-independent data format as part of Electronic Data Interchange (EDI). The driver class, DataConverter, is a stand-alone class that provides input for the ProgramIO entity to create instances accordingly, and save the output in the right format.

In Figure 1.0, a more detailed relationship between entities and their states is shown. The ProgramIO entity consists of type-specific parser classes that extend the main and more general parser class. The main parser extends ProgramOutput to allow the system to write class instances to JSON and XML files, and it also implements the Loader interface. The loader interface is designed to make sure that a transition to reading data from other sources will not be complicated and difficult.

Both Person and Asset classes are designed to be stand-alone entities with encapsulation and abstraction principles in mind. More general functionality for each subclass is defined in superclasses, and sub-class specific functionality is defined in sub-classes. For instance, the getValue() method is an abstract method in Asset superclass, and the actual implementation differs per each type of Asset. However, getAssetCode() is defined and implemented in the superclass as it is more general to each type of Asset. Furthermore, a Person can be a broker or just an individual in the system, and a broker can be a junior or an expert. An interface called PortManagement or Portfolio Management is created to be implemented by each of the types of a broker since the way they manage Assets differs.

Find Figure 2.0 on the next page.

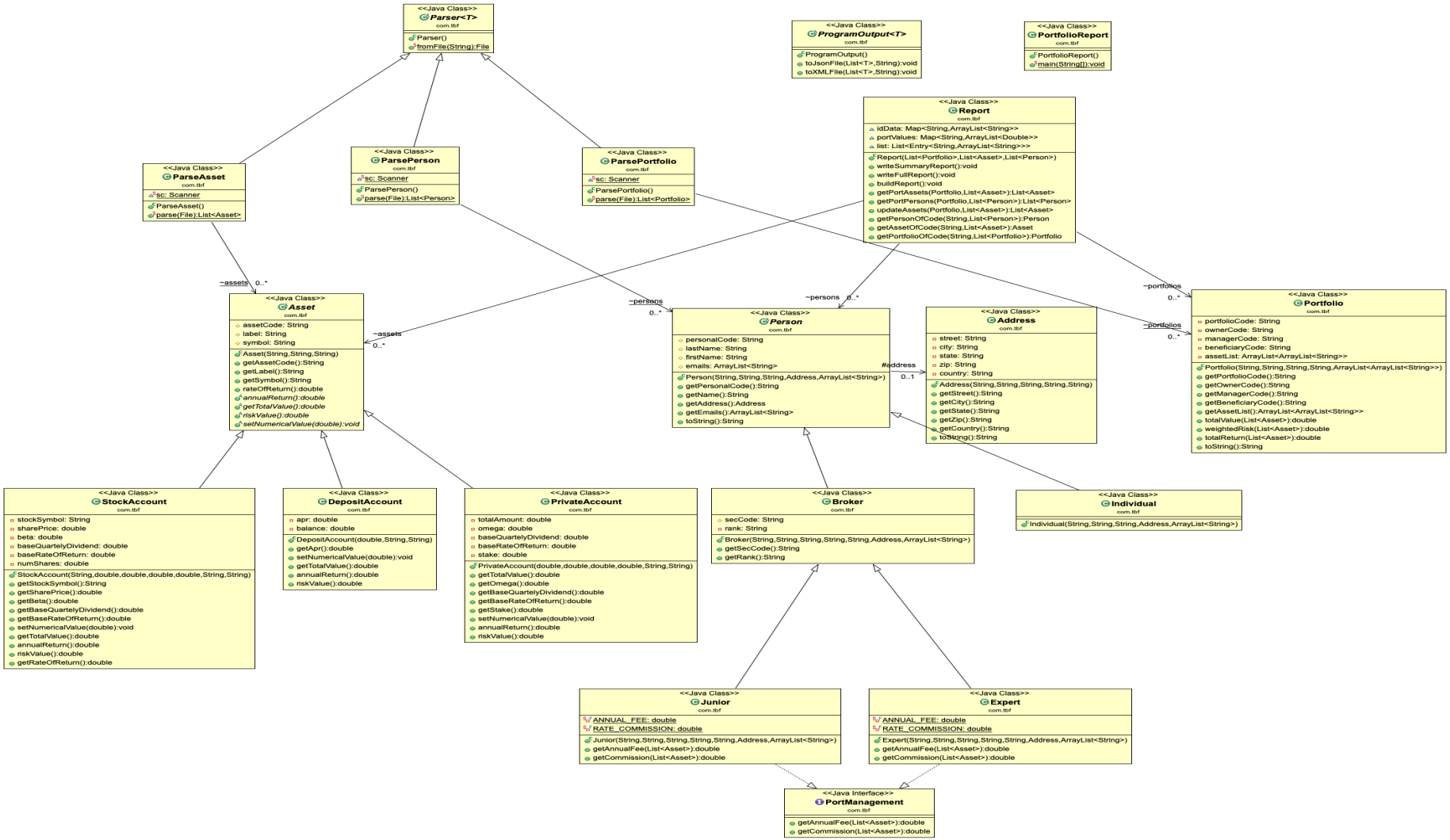

Figure 1 UML figure illustrates the relationship between each class

3.2.1 Component Testing Strategy

A test case was developed to test the DataConverter class's ability to convert data from a flat CSV file to the PersonPerson or asset class. These classes are outputted to JSON and XML files, where they are checked for accuracy compared to the expected file outputs.

The data in the test case was generated on the website: www.generatedata.com/

3.3 Database Interface

Database is added to reflect the OOP design above. Assets, persons, and portfolios are the main tables. They are connected to each other through foreign keys and joint tables. The asset table is connect to deposits, stocks, and private assets tables through one-to-many relationships. The portfolio asset table is a joint table that connects portfolio and assets that are available on a particular portfolio. As show in the figure 2.0, person table is connect to portfolio through one-to-many relation because from the person table, the owners, managers, and beneficiary of a particular portfolio are derived. In addition, the database design allows the user to store more than one email of a person. The emails are stored in a table that is connected to person table through one-to-many relationship from person to email.

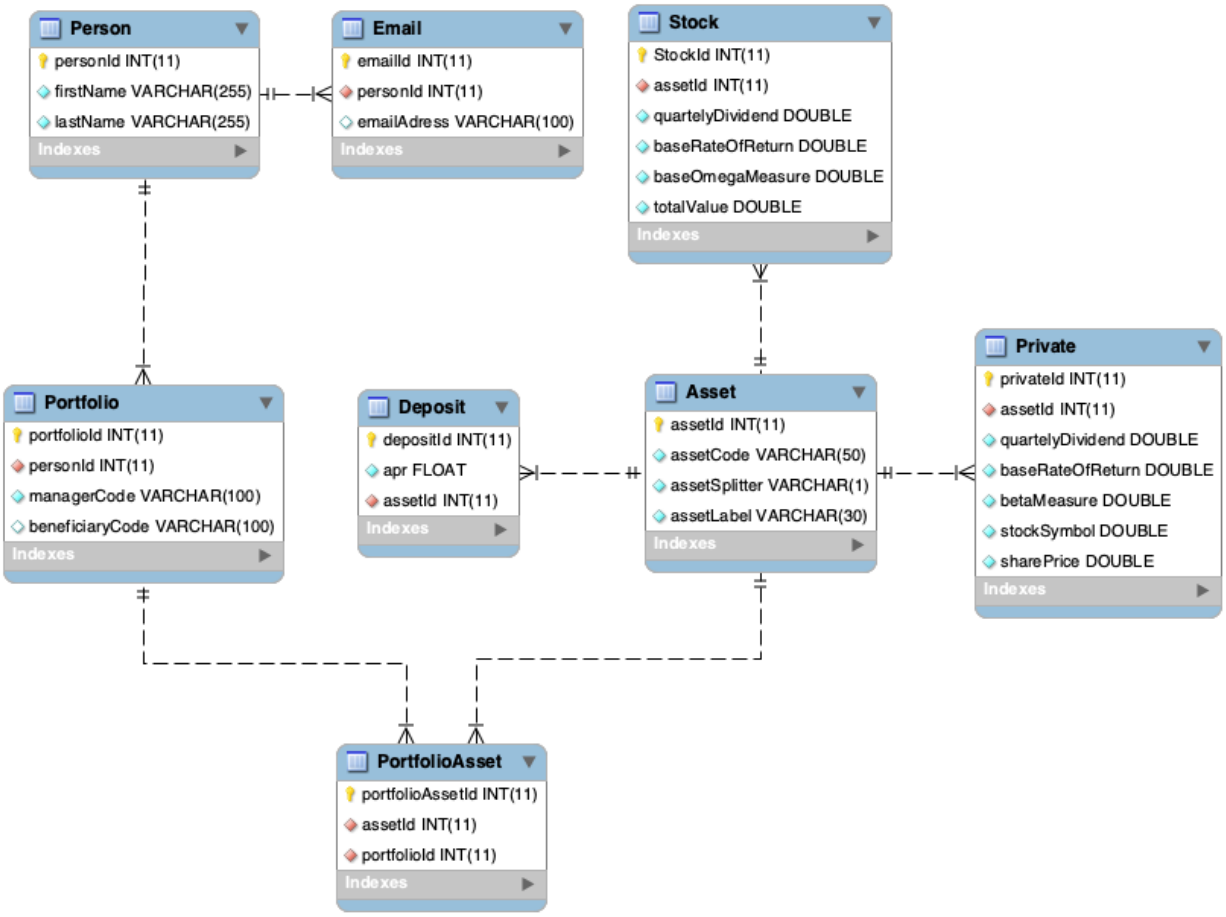

Figure 2 EER diagram to show entity relationship of database design

3.3.1 Component Testing Strategy

[Test cases will be added in a subsequent phase.

3.4 Design & Integration of Data Structures

Data structures will be added in a subsequent phase.

3.4.1 Component Testing Strategy

Test cases will be added in a subsequent phase.

3.5 Changes & Refactoring

Below is the outline of significant changes implemented in version 1.1:

- ProgramIO or program Input/Output entity is created to support the overall system in loading data from different sources and saving the output in different platform-independent formats like JSON and XML. This entity includes four classes (Program Output, Parser, ParseAsset, and ParsePerson) and one interface called loader
- A loader interface is created to ensure that future changes in the way of reading and export data will not affect any other part of the system.
- The class that models a person is subdivided into two more classes, broker and individual. Broker class was further subdivided into two more classes, junior and expert broker.
- A PortManagement or portfolio management interface is created to be implemented by both junior and expert brokers in different ways.

Below is the outline of significant changes implemented in version 1.2:

- The program input/output section of the design was overengineered, and it was revised to reinforce principles of abstraction and encapsulation.
- A class for portfolio and its parser was created to support the functionality of the system
- A database design was created, and the EER diagram was produced to summarize entity relationship in the database.

4. Additional Material

Additional Material may be added in a subsequent phase.

5. Bibliography

References may be added in a subsequent phase.