CSCE 156 – Assignment 2

Fall 2013

Introduction

Over the course of this semester, you will incrementally build a substantial database-backed application in Java. Each phase of the project will have you focus on a particular component which will have deliverables that you must hand in by a certain date. These deliverables will include self-executing JAR (Java Archive) files, ZIP archive files of your source code, well-written test cases, database schemas and a well-written technical design document. Each phase builds upon and may require updates and modifications to prior phases. It is important that you understand the entire scope of the project so you should read all of the assignments to get an understanding of where the project will be going.

The iterative nature of this project means that it is important that you do not fall behind. In each phase it is also important that you have a good, well-thought design to make subsequent phases easier to design and implement. Poor designs, bad implementations, bugs and broken code will mean subsequent phases of this project will suffer. Investing a little extra time and resources upfront will minimize your "technical debt" and mitigate the need to update or refactor your design later on. Remember the Golden Rule of Coding: only code that which you would not mind having to maintain.

Project Statement

The DCH Financial Group has hired you to design and implement a simple financial management system to replace the aging AS400 green-screen system that they currently use. It will be your responsibility to design a new system from scratch that is Object Oriented, written in Java, and supports DCH's business model by implementing their business rules and providing the functionality as stated below.

DCH manages portfolios of various assets for their customers. The application that you will build will need to support 3 different kinds of assets: Deposit Accounts, Stocks, and Private Investments. Each investment consists of various pieces of data that define their value, rate of annual return, and a measure of *risk*.

Each asset has an alpha-numeric *code* which uniquely identifies it (in the old AS400 system) and a *label* to describe what the asset is.

- Deposit Accounts are FDIC (Federal Deposit Insurance Corporation) insured accounts like Savings Accounts, CDs (Certificates of Deposit), and MMAs (Money Market Accounts). Each Deposit Account also has a *balance* (its value) and an *APR* (annual percentage rate). Interest is compounded, so a better estimate of the annual rate of return is obtained by computing the APY (annual percentage yield) using the formula $APY = e^{APR} - 1$ (where APR is on the scale [0, 1]). Since Deposit Accounts are FDIC insured, their risk measure is zero.

Stocks and Private Investments are investment accounts whose value is at some risk of loss. The financial industry has several measures of risk (alpha, beta, r-squared, etc.). DCH however, has developed a proprietary measure of risk called the "Omega" measure whose details are a tightly controlled secret. Each asset has been examined and assigned an omega measure which is simply a number on the interval [0, 1] with 0 being no risk and 1 being extremely risky. In addition, investment assets have a *base rate of return* and may pay *quarterly dividends* which contribute to the expected annual return beyond the base rate of return. Thus, the expected annual return of an investment account is its base rate of return multiplied by its value plus the value of its 4 quarterly dividends. An investment asset's rate of return is the expected annual return divided by its total value.

- Stocks are investment accounts that consist of a *number of shares* of a particular publicly traded stock. The total value is of course the number of shares times the *share price*. In addition, all stock assets have a *base* risk measure of 0.10 (thus, the total risk of a stock is its base risk plus its individual omega measure). The quarterly dividend is a dividend that is paid on a per-share basis.
- Private Investments are investment accounts that represent investments in private enterprises (rental properties, small to medium businesses, franchises, etc.). Private Investments are, in general more risky than stocks so they have a base risk measure of 0.25 (plus its individual omega measure). Instead of shares, a private investment consists of *amount* invested (its value) and a *stake* in the investment (a percentage). The quarterly dividend for this type of investment is a *total dividend* paid to all stake holders in proportion to their percentage stake.

Each portfolio consists of any number of assets that are managed by an SEC (Securities and Exchange Commission) certified broker. For tax purposes, each portfolio is exclusively owned by one person. The owner may also *optionally* designate a beneficiary of the portfolio who will receive ownership of the portfolio upon the owner's death. In addition, each portfolio also has per-asset annual fees and commissions associated with it.

There are two types of brokers: Expert brokers and Junior brokers. Portfolios managed by a Junior broker have a $50.00 per-asset annual fee and are assessed a commission equal to 2% of the total annual expected return of the portfolio. Portfolios managed by an expert broker have a $10 per-asset annual fee and a 5% commission rate. Each broker also has a unique alphanumeric SEC identification code.

A portfolio has an omega risk measure which is a weighted according each asset's omega rating weighted according to its value with respect to the total portfolio value. That is, if there are n assets then for each asset a_i, let o_i be its omega measure, v_i be its value and let V be the total value of the entire portfolio. Then the portfolios weighted omega measure is

$$\omega = \sum_{i=1}^{n} o_i \frac{v_i}{V}$$

Finally, each person in the system, regardless of role (owner, broker, beneficiary) is represented with the same basic data. Every person has an alphanumeric code that uniquely identifies them (in the old system), a last name, first name, and an address (street, city, state, zip, country). In addition, each person can be associated with any number of email addresses.

Phase I – Data Representation & EDI

Objects represent and model real-world entities. The *state* of an object consists of the pieces of data that conceptually define what that object is. Often it is necessary to be able to transmit objects between different systems which may use completely different languages and technologies. The transfer of such data is known as Electronic Data Interchange (EDI) and is achieved by translating objects into a platform-independent data format such as XML (Extensible Markup Language) or JSON (JavaScript Object Notation) representations. Once transferred, the data on the other end can then be translated into an object in the second system. Different languages and frameworks have their own terminology for this process (marshaling/unmarshaling, serializing/deserializing, etc.).

The first phase of the project will focus on the design and implementation of several Java classes to model the various assets and persons in the system. For the first phase, your classes may be simple data containers (full behavior and methods will be designed and implemented in Phase II). That is, your classes will define, conceptually, what each of the entities are (their data and types and accessor/mutator methods) as well as provide means for creating and building instances of those entities.

You will also write a parser to process "flat" data files containing data on individuals and assets and build instances of each object. These files were dumped from the old system and are in a non-standard semi-colon and comma separated value format.

Finally, you will also implement functionality to serialize your objects into a data interchange format. You may choose to serialize them into either XML or JSON (or both for extra credit).

Data Files

The data dumped from the old system is separated into several files. A full example of well-formatted input and acceptable output has been provided. However, you will also be required to develop your own non-trivial test case. In general, you may assume that all data is valid and properly formatted and all data files are named as specified. You should assume that all data files are located in a directory called `data` and any output files should be saved to the same directory.

Persons Data File

- Data pertaining to each person on the system is stored in a data file in `data/Persons.dat`
- The format is as follows: the first line will contain the total number of records (an integer). Each subsequent line contains semicolon delimited data fields:
    - Person Code – the unique alpha-numeric designation from the old system

- Broker data – if the user is not a broker, this field will be empty. Otherwise it will contain two pieces of data separated by a comma. The first piece is either E or J (indicating an Expert or Junior broker) and the second is the person's SEC identifier
- Name – the person's name in "last-name, first-name" format
- Address – the mailing address of the customer. The format is as follows: "STREET,CITY,STATE,ZIP,COUNTRY"
- Email Address(es) – an (optional) list of email addresses; if there are multiple email addresses, they will be delimited by a comma.

Asset Data file

- Data pertaining to each asset on the system is stored in a data file in `data/Assets.dat`
- The format is as follows: the first line will contain the total number of records (an integer). Each subsequent line contains semicolon delimited data fields depending on the type of asset.
  - Deposit Accounts have the following format:
    code;D;label;apr
  - Stocks have the following format:
    code;S;label;quarterly dividend;base rate of return;omega measure;stock symbol;share price
  - Private Investments have the following format:
    code;P;label;quarterly dividend;base rate of return;omega measure;total value
- All percentage rates are on the range [0, 100] so you may need to adjust appropriately.

Source Files

You will be required to design Java classes to model the problem above and hold the appropriate data. The classes you design and implement, their names, and how they relate to each other are design decisions that you must make. However, you should follow good object oriented programming principles. In particular, you should make sure that your classes are designed following good practices.

However, to make sure that your program is executable you will export your project as a runnable JAR file. Place your main method in a class named `DataConverter`. Upon execution your program should load the data from all the data files, construct (and relate) instances of your objects, and produce two output files, one containing converted Person instances and one containing converted Asset instances.

You have a choice of which output format to follow. You may either produce XML (in which case name your output files `Persons.xml` and `Assets.xml` respectively) or JSON (in which case name your output files `Persons.json` and `Assets.json` respectively). Your output files should be placed in the same `data` directory as the input files.

There is no need to define a rigorous schema in either case. However, your XML or JSON should be well formatted and valid (in particular, you *may* need to escape certain characters). You should follow the general structures in the examples provided, though some flexibility is allowed for tag and element names. However, the output should *conceptually* match the expected output. It must also pass any and

all validation tests (using the validators listed below). In addition, you *may* (in fact are encouraged to) use a library to convert your Java classes to XML or JSON if you wish. Some common libraries and more information on each of the formats can be found with the following resources:

- W3C's XML Tutorial: http://www.w3schools.com/xml/default.asp
- An XML Validator: http://www.w3schools.com/xml/xml_validator.asp
- JAXB, a standard Java-XML binding framework: http://www.oracle.com/technetwork/articles/javase/index-140168.html
- XStream, a lighter-weight XML binding framework: http://xstream.codehaus.org/
- JSON Introduction: http://json.org/
- A JSON Validator: http://jsonformatter.curiousconcept.com/
- Google-gson library to convert between Java objects and JSON: http://code.google.com/p/google-gson/

Process

For this initial phase, your objects may be simple data containers since the only thing you are doing with them is parsing data files, creating object instances, and exporting them in a different format. However, much of the code you write in this phase will be useful in subsequent phases, so ensure that you have well-designed, robust, bug-free and reusable code.

Testing

You are expected to thoroughly test and debug your implementation using your own test case(s). To ensure that you do, you are required to design and turn in at least one non-trivial test case to demonstrate your program was tested locally to some degree. Ultimately your application will be tested on multiple test cases through webgrader. However, webgrader is *not* a sufficient testing tool for you to use. Webgrader is a "black-box" tester: you don't have access to its internals, to the test cases, nor can you debug with respect to it.

Understand what a test case is: it is an independent input-output that is *known* to be good using a method independent of your program (calculated or verified by hand). Providing your test case early along with your design document will be worth bonus points. We will use these test cases when grading other assignments, so be adversarial: design test cases to probe and break "bad" code, but stay within the requirements specified above.

Artifacts – What you need to hand in

You will turn in all of your code as a runnable JAR file using the CSE webhandin. You will also include a ZIP archive of all your source code so that we may evaluate your code. Specifically, make sure you adhere to the following:

- For grading purposes, place all of your code into a package named `unl.cse`
- You may hardcode the data file names and the `data/` directory as specified above
- To demonstrate that you tested your program, you should also hand in at least one non-trivial test case (2 input data files with the format and naming specified above that you produced

along with correct expected output files). Making your test case available to us in advance is worth extra credit.

- Turn in a single runnable JAR (Java Archive) file named `DataConverter.jar`
- We will grade your assignment from the command line as follows (you should make sure that it executes on CSE using the webgrader program):
  ```
  ~>java -jar DataConverter.jar
  ```
- To make sure your code is gradeable, generate a ZIP archive file of your source code and turn it in via webhandin
- See Appendix A for details on creating runnable JAR files and ZIP files of your source code in Eclipse.
- In addition, you will be writing a design document. The first draft of this document is due 1 week prior to this assignment.

Bonus Items

There are several opportunities for bonus points. Sometimes these opportunities are functional; sometimes they involve elements in your Design Document. All bonus opportunities will be enumerated in the rubric. If you wish for us to consider any of these bonus items, clearly indicate in your rubric that we should evaluate them. Failure to indicate this means that bonus items will not be considered.

Common Errors

Some common errors that students have made on this and similar assignments in the past that you should avoid:

- Design should come first—be sure to have thought out a design for your objects and how they relate and interact with each other before coding.
  - OOP requires more of a bottom-up design: your objects are your building blocks that you combine to create a program (contrast with a Procedural Style which is top-down)
  - Worry about the design of objects before you worry about how they are created.
  - A good litmus test: if you delete your driver class, are your other objects still usable? Is it possible to port them over to some other uses or another application and have them still work without knowledge of your driver program? If yes, then it is probably a good design; if no, then you may need to reconsider what you're doing.
- Objects should be created via a constructor (or some other pattern); an object should not be responsible for parsing data files or connecting to a database to build itself (a Factory pattern is much more appropriate for this kind of functionality).
- Encapsulation should be respected. Appropriate data fields and appropriate types should be defined for each class. Visibility should be restricted with access done through accessor/mutator methods. Any methods or functionality that acts on a class's data should be *encapsulated in the class* (unless usage of an external library makes it inappropriate). If a value is based on an object's state and that state is mutable, then the value should be recomputed based on the state it was in at the time that the value was asked for.

- Classes should be designed as stand-alone, reusable objects. Design them so that they could be used if the application was changed to read from a database (rather than a file) or used in a larger Graphical User Interface program, or some other completely different framework.S

Appendix A: Creating Runnable JARs and Archive Files

These instructions are for creating runnable JAR file in Eclipse (Indigo); instructions may differ for other versions or other IDEs.

Creating a Runnable JAR file:

1. Run your program at least once, this creates a “Launch Configuration” in Eclipse
2. Right click on your project, select “Export”
3. Under the Java folder, select “Runnable JAR file”, click Next
4. Select the “Launch Configuration” corresponding to your main method
5. Under “Export destination:” click Browse and select a director location and file name (ending in .jar) where you want the JAR file to be exported to
6. Under “Library handling:” be sure that “Package required libraries into generated JAR” is selected
7. Click Finish
8. The JAR file should now be in the directory you indicated

Creating a ZIP archive of your source code:

1. Right click your project, select “Export”
2. Under the “General” folder select “Archive File”
3. Click Next
4. Click Browse and select a directory/file name to export to
5. Under “Options” make sure that “Save in zip format” is selected
6. Click Finish; the ZIP file should now be in the directory/file you indicated

Appendix B: Adding External JAR Libraries to an Eclipse Project

These instructions are for creating runnable JAR file in Eclipse (Indigo); instructions may differ for other versions or other IDEs.

There are many ways to import external JAR libraries (examples: gson-1.7.1.jar, joda-time-2.0.jar). The following instructions will be most compatible with how we expect you to build your runnable JAR file.

1. Create a folder in your project: Right click project > New > Folder
2. Name your folder “lib” (short for library)
3. Drag and drop your JAR file to this folder, be sure to select “copy files”
4. Right click the new JAR file in your lib folder and select Build Path > Add to Build Path

Appendix C: Partner Policy

If you choose, you may work in pairs (no groups larger than two though) for each phase of the project. If you choose to work in pairs, you must adhere to the following:

1. All work *must* be the result of an equal collaborative effort by both partners
2. Turn in only one copy of the design document with both your names on it.
3. You must turn in only one copy of the JAR file under the first author's login
4. You must follow any additional policies regarding late passes or other items as described in the syllabus