CSCE 156 – Assignment 3

Fall 2013

Introduction

In the previous assignment you began initial work to support a portfolio management system for DCH Financial Group. In this iteration, you will continue to design classes to support the application as well as modifying your previously design classes to have add functionality rather than being mere data containers. Refer to the previous assignment for the project requirements and details.

Phase II

In this phase we will add functionality to our classes from Phase I and design new classes to complete the core functionality of the portfolio management system. We will integrate all of our classes to produce two reports. The first will be a portfolio summary report that will report overall figures and totals for all portfolios. The second will report the details of each portfolio.

Data Files

The data files for persons and assets will be the same as in Phase I. A new additional data file will be provided that contains all information relating to portfolios. As before, you may assume that all data is valid and properly formatted and all data files are named as specified.

Portfolio Data file

- Portfolio data is contained in `data/Portfolios.dat`
- The first line contains an integer indicating the number of records in the file. Each subsequent line contains the following data, separated by semicolons.
    - Portfolio code – An alpha-numeric identifier used by the old system that uniquely identifies the portfolio
    - Owner code – an alphanumeric code corresponding to the person who owns the account
    - Manager code – an alphanumeric code corresponding to the person who manages the account
    - Beneficiary code – an (optional) alphanumeric code corresponding to the person who is the designated beneficiary for the portfolio
    - Asset list – the remaining item is a comma-delimited list of assets belonging to the portfolio. Each asset is represented by an asset code along with a numeric value (delimited by a colon). Depending on the type of asset the asset code corresponds to, the numeric value should be interpreted differently. For deposit accounts it is the total balance, for stocks it is the number of shares owned, and for private investments, it is the percentage of the stake that the portfolio owns (on the scale [0, 100]).

- An example was provided in the same ZIP file in the previous phase (along with expected output).

Source Files

You will be required to design Java classes to model the system and functionality as specified and hold the appropriate data. Which classes you implement and design and what you name them and how you relate them 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.

Make your program executable by placing a main method in a class named `PortfolioReport`. Once invoked, your program should load the data from all the data files, construct (and relate) instances of your objects, and produce 1) a summary report, and 2) detailed portfolio report for each portfolio. Output will be directed to the standard output.

Formatting details are left up to you but your reports should be readable and communicate enough information to verify that your code is correct and that you've followed all requirements and specifications. An example has been provided (see the ZIP file and/or webgrader) but you are free to make it prettier and communicate more information if you wish.

The summary report should contain, at a minimum, the following:

- Portfolio code
- Portfolio owner
- Portfolio manager
- A total of all fees
- A total of all commissions
- The weighted omega measure
- A total of all expected annual returns
- A total of all asset values

The detail report should print details about each portfolio including line-item details on each asset (code, label, return rate, omega risk measure, annual return, and value).

Design Process

You have some flexibility in how you design and implement this phase of the project. However, you must use good OOP practices. You should non-trivially demonstrate the proper use of the four major principles: inheritance, abstraction, encapsulation, and polymorphism. When thinking of your design, keep the following in mind.

- What should the public and private interface of each of the classes be? Don't make them simple data containers—what methods (or services) should each class provide?

- Think about the state and methods that are common and/or dissimilar in each of your objects. What would be an appropriate use of inheritance and which methods/state are specific to subclasses? What, if anything should the subclasses define or override?
- What classes should own (via composition or some other method) instances of other classes? How are complex relationships made between objects?
- Think about how to utilize polymorphic behavior to simplify your code. You should not have to handle similar objects in a dissimilar manner if you have properly defined a common public interface.

Object-oriented design is fundamentally different from the top-down procedural style that you may be used to. Rather than breaking a problem down into sub-parts, we instead do a bottom-up design. We identify the entities and design classes that can be used as the building blocks to implement a larger application. *You are highly encouraged to completely understand the problem statement and have a good prototype design on paper before you write a single line of 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; you don't have access 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 (say, calculated by hand). Providing your test case early along with your design document will be worth extra 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 include all of your source code in a separate ZIP file 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 assume that they are in the same working directory (with names specified as above)
- To demonstrate that you tested your program, you should also hand in at least one non-trivial test case (data files with the format and naming specified above that you produced along with a correct expected output in a plain text file—name it `output.txt`). Making your test case available to us in advance is worth extra credit.
- Turn in a single runnable JAR (Java Archive) file named `PortfolioReport.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 PortfolioReport.jar`
- Turn in a Zip archive file containing all of your source code named `PortfolioReport.zip`
- In addition, you will be updating your 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).
- Classes (at least not all of them) should not be mere data containers: if there is some value of a class that depends on aggregating data based on its state, this should be done in some method, not done outside the class and a field set (violation of encapsulation—grouping of data with the methods that act on it). 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.