CSC 171 PROJECT 2

DUE APRIL 12TH AT 11:59PM

Purpose

A smart programmer recognizes the fact that the vast majority of problems have already been solved, and rather than trying to invent a new solution, it is best to leverage the optimal solutions that have been used in the past. Many of these solutions have been documented as Design Patterns. A design pattern is a generally agreed upon "best practice" that is documented and shared with other software developers. A design pattern is not code, but rather a general purpose design that may be adapted to a specific application.

One of the most important books written on the subject is *Design Patterns: Elements of Reusable Object-Oriented Software*, which was written by the "Gang of Four" (Gamma, Helm, Johnson, and Vlissides). In the years since its first publication, many other books on the subject have been written to capture countless other design patterns.

The Model View Controller (MVC) pattern was first documented in *The Journal of Object Technology* in 1988, but was first implemented as early as the 1970s. The MVC pattern is used when designing user interfaces. As its name suggests, the MVC is separated into three parts:

1. The *Model* contains the application data and logic. The model provides methods that can be used to retrieve or change the state of the application, and generates notifications when changes are made.
2. The *View* displays the relevant parts of the model to the user, and responds when the model is changed.
3. The *Controller* accepts input from the user and uses it to interact with the model.

It is common to provide many different user interfaces into the same application. An application may be accessible via a desktop app, on a mobile device, or through a browser (for example). Novice programmers often include large parts of the application in the user interface, so replacing the UI requires much of the application to be rewritten. Not only is this tiresome and inefficient, it is dangerous as well. Keeping the different versions of the application in sync is problematic. The primary goal of the MVC pattern is to separate the majority of the application (the model) from the user interface (the view) so that different views may be used with the same application without requiring large parts of the application to be rewritten.

Problem Domain: Banking

Many Banks provide a variety of interfaces for customers to use when interacting with their accounts: ATMs, mobile apps, web banking, and so on. In this project you will implement a basic banking application that provides at least two user interfaces. Your application will be expected to divide your application into at least 4 parts, each of which should be implemented in a separate package (e.g. **`csc171.project02.model`**):

a. The model that contains all of the classes required to represent the Bank, Customers, Accounts, etc. according to the detailed requirements below.
b. A controller that accepts input/commands from a view and manipulates the elements of the model.
c. A console-based user interface that prompts the customer to enter text-based commands to interact with their account(s). The UI should translate user commands into calls on the controller. The UI should also update the information displayed to the user when the state of the model is changed.
d. A Graphical User Interface (GUI) that uses Java components from the Swing and AWT toolkits in the standard Java class libraries to allow the customer to interact with their account(s). The GUI should translate the user's interaction with the GUI components into calls on the controller, and should update the information that is displayed to the user when the model changes.

For full credit, your application must provide at least the following functionality. All functionality must be available in all views.

1. A new customer must be able to create a new customer account by providing at least the following information: first name, last name, birthdate, username, and password. A unique 10-digit account number should be generated and assigned to each customer account. If an account with the same username already exists, the customer should be asked to choose a different username.

2. An existing customer must be able to log into their account by providing the correct username and password. They should also be able to log out of their account. Only customers that are logged in should be able to access the features below.

3. A customer must be able to open a new Savings account. A savings account has a balance, an interest rate, and a unique 10-digit account number that is

automatically assigned. Savings accounts should keep track of at least the last 10 transactions (deposits and withdrawals).

4. A customer must be able to open a new Checking Account. A checking account has a balance, and a unique 10-digit account number that is automatically assigned. A checking account should keep track of at least the last 10 transactions (deposits and withdrawals). Checking account withdrawals come in three distinct varieties: ATM/teller withdrawals, debit-card deductions that include the name of the payee, and cashed checks which include a check number.

5. A customer should be able to see a list of their current accounts, including the account number and the current balance.

6. A customer should be able to choose one of the accounts and see the details of at least the last 10 transactions.

7. A customer should be able to deposit some amount of money into the account of their choice. The balance in the account should be updated to reflect the deposit. This should be recorded in the transaction history for the account.

8. A customer should be able to withdraw some amount of money from the account of their choice. If the withdrawal is from a checking account, they should be asked to provide the type of withdrawal (debit, check, or ATM/teller).

9. A customer should be able to transfer money from one account to another. This should be indicated in the transaction histories for *both* accounts (in one as a withdrawal, and in the other as a deposit).

10. A customer should be able to close an account. The account will remain associated with the customer, but the customer will no longer be able to interact with the account (deposit or withdraw money). The closure should show up as two transactions: a withdrawal for the remaining balance in the account, and the account closure.

Additional Enhancements (Extra Credit Opportunities)

Get creative. You may earn up to 20% in extra credit depending on how clever your enhancements are.

Grading Criteria

You will be graded according to the following criteria.

Class Definitions (20%)

At this point, you are expected to practice relatively sophisticated class design including use of inheritance and polymorphism where appropriate. Classes should be separated into four packages as described above (model, controller, console ui, and gui), and you should make appropriate use of access modifiers (public, private, protected, package protected) accordingly. Your design should reflect a clear separation between the model and the user interface. Static methods and variables should only be used when appropriate.

Bank UI Implementations (40%)

You will be graded based on your implementation of the requirements in the problem domain section, including the 10 specific requirements as well as two fully functioning user interfaces. You will also be graded on your implementation of the MVC design pattern. There should be a clear separation between your model and your views, maximizing the code reuse between the two different user interfaces that you will be implementing and minimizing the amount of application logic coded into your user interfaces.

Testing (20%)

You should write tests for each of your classes. It is strongly encouraged that the unit testing capability integrated into Eclipse is used to implement your tests (jUnit). You can find lots of documentation describing jUnit in Eclipse online. You may also simply write a separate test for each of your classes that includes a main method to test the class. For example, if you implement a “BankAccount.java” class you would create a “BankAccountTest.java” class that uses a main method to create and test a “BankAccount.” You will **lose credit** if your test methods are implemented in your classes (as opposed to a separate class).

Code Style (10%)

- Modularity of design - Objects that make sense. High cohesion, low coupling.
- Comments - Where appropriate!
- Names - Variables, Methods, Classes, and Parameters
- Indentation and White Space

Documentation (10%)

In addition to submitting the Java code (.java files), you are required to submit a README text document which specifies the following:

1. A one paragraph description of your class designs.
2. A one paragraph description of how you separated your application from your user interfaces.

3. A one paragraph description of your testing strategy.
4. Detailed instructions on how to compile and run your application and your tests. This should include command line arguments (if any), appropriate responses to any prompts, and instructions for using your GUI.
5. Full documentation for any enhancements that you wish to be considered for extra credit. YOU must explain what your enhancements are, and how the TA should enable them or otherwise detect their presence.

Additionally, provide a file called “SampleOutput” that includes the output of an example interaction with your console UI.

HAND IN

Projects will be submitted as ZIP archives via Blackboard. You will be given at least 2.5 weeks to complete the project, and there will not be a lab due the same week that the project is due. This is more than enough time to complete the project. You will also have unlimited attempts to submit your project (only the last submission will be graded), which means that you will be able to upload your solution as soon as you have something working, and try for improvements or extra credit afterwards. Because of this, late submissions **will not** be accepted.

You will be expected to submit:

- Java source files and compiled Java class files. You must also submit any additional files required to run the programs.
- Documentation described in the section above. If you submitted any additional files, make sure to explain their function in your documentation.

Grading

Each TA will compile and run your code using his or her own test cases to determine correctness based on the above criteria. If your program produces only partial functionality, providing your own test cases will give you partial credit for the parts that operate correctly. Each TA will also conduct a code review to evaluate the correctness of your code and the code design.