Phase 2

Step 1: Refactor your code in any way that improves SOLID adherence, include design patterns where useful, and encapsulates as much of your code as possible. (We will try to get Phase 1 feedback to you as soon as possible.) See Week 9 Lab for tips.

Step 2: Add the following functionality:

+ All Debt accounts should have a maximum amount of debt you can incur.

+ Add one more type of account (of your choice) that has at least one property that is different from existing accounts. Feel free to look at bank websites for ideas.

+ Add one more type of employee of the bank who has less access than the manager, but can also be a user with their own accounts.

+ Make sure you can demo time-sensitive features, like adding interest to savings accounts each month.

+ Improve the manager "undo" functionality. There is more than one way to interpret this statement. For example, you could have unlimited undo capability for the manager, or the ability to specify a number n so that the system undoes n transactions on an account, or you could have undo functionality for a user's previous n transactions (as opposed to an account's previous transactions). You can interpret this statement in any reasonable way that is as complex as the above suggestions.

+ Refactor so that you can have join accounts where two distinct users can both withdraw, deposit, and check the balance on the same account.

Step 3: Personalize your program. Add any features you want so that your finished program is twice as complex (or more) than your Phase 1 submission. Below are some directions in which you could go. If you can think of another direction, feel free to pursue it.

+ You could create a GUI with Java's swing package (JFrames, JPanels, etc.), with XML (there are Java XML tutorials online), make a more robust and feature-rich command line interface, or other sort of UI.

+ You could find ways to incorporate other currencies, including a way to exchange one currency for another, a primary currency for each account, a primary currency for each user, unless they request an account in a second currency, updated transfer functionality, associated bank fees, etc.

+ You could build an elaborate bank fee system including products like mortgages (or other products) which generate fees in different ways. This could include services provided for the bank that you can design. For example: Guaranteed Interest Certificates (GICs) that automatically return money to the account of origin at the end of the investment term, etc.

+ You could design the ATM of the future, so that it handles things that are not money, as well. For example, your program could also track and update on-line shopping history **OR** social credit scores for each user. Possible examples: collect the urls of each website and money spent there with input from the

user or a file that simulates input from their browser, including unfinished purchases, returns, and other things that your program can remind the user about; allow people access create/update/send a professional profile to apply for jobs and track which job postings are still open and which have closed; an account that tracks an individual's social credit score such as the one China has implemented, with updates from the government; etc).

Your design is good if it is relatively easy for you to implement these new features. Before implementing anything, ask yourself **"Is there any way to change our design so that it is easier to implement this new feature?"** If the answer is "yes", refactor the code (change the design) before implementation. This will get you a higher mark than forcing many impressive features into a not-so-impressive design.