Name: Daniel Ems
Date: 03/11/2017
Current Module: Object Oriented Programming
Project Name: Bank of Nerds

Project Goals:

The project goal is to create a bank. The bank must be able to provide multiple products to multiple customers. The products the bank must be able to provide are; checking, savings, and 401k(retirement) accounts. Each customer must be able to withdraw and deposit money, as long as the account does not have stipulations preventing such. The customer must be able to list their accounts, and the user of the program must be able access any account the bank holds.

Considerations:

- What will be class candidates?
- What will be sub-class candidates?
- What attributes and methods should each class possess?
- What will be the best structure for a menu?
- What forms of error handling need to be implemented?
- What will be the best architecture?

Initial Design:

The initial approach to this project was to break out the different elements into classes and subclasses. This required what each element of the project “is”, and “has”. For example, a bank has customers, a customer has accounts, and accounts have balances? Furthermore, there was additional analysis required to determine what methods each class should possess. For example, originally, it made sense that the bank possessed the method to deposit and withdraw. However, further thought led to the decision that the accounts should instead possess those methods. The analogy that led to this conclusion was the relationship between cards and decks. In this scenario, the accounts would be the deck, and their balances would be the decks cards (more specifically money).

Once classes were determined, the next design was that of the menu, and program architecture. This quickly became a difficult question to answer. The dilemma resides in the identifying the role of a menu within a bank. The conclusion derived was the menu should be presented in the form of a bank teller. This decision helped to tie together the user interfacing with the bank. In the real world, a customer interfaces with a bank through the tellers. The tellers have access to certain bank methods, and information. Likewise, the bank has access to the tellers methods. Unfortunately, this did not offer an absolute conclusion. Admittedly, upon completion of the project, there are still elements of the teller that are arguable. However, a strong argument can be made for their final placement within the teller.

The final design aspect that needed to be addressed was that of error handling. The variety, volume, and spread of user input throughout the program required special attention to properly insert error handling in the programs architecture. The most difficult aspect was not knowing where error handling would be needed until the input was built. This led to further questions of “does that error check belong in that class”? However again, there is are arguments both for their final location, and other potential locations. Ultimately, It was decided that the teller should do most of the error handling

much like a teller would at a bank. For example, if you wished to withdraw negative funds, it would be the teller's job to notify the customer this is not allowed.

Data Flow:

1. The tellers welcome method which welcomes the customer and presents an initial menu.
2. The initial menu offers the options of new customers, existing customers and exit. (user selects new Customer)
3. The tellers new customer method asks the user for their name and age.
4. The tellers new customer method passes the name and age to the banks new client method
5. The tellers select product method prompts the user to select the type of account they wish to open
6. The teller passes their choice to the banks new account method
7. The teller presents the user with the account actions menu (deposit, withdraw, create account, list accounts, main menu). (deposit, or withdraw)
8. It calls the tellers list accounts method, a wrapper function for the customers list accounts function.
9. The customer is prompted to select an account and then calls the tellers deposit or withdraw method
10. The deposit or withdraw method is a wrapper for the accounts deposit/withdraw methods.
11. Finally the user is prompted for an amount, the action is completed, and the tellers account actions is called.(7.)
12. Account actions (list accounts)
13. The tellers list accounts (8.) is called, and the account actions I called(7.)
14. Account Actions (create accounts)
15. The teller calls the select product method
16. The user is prompted to select a product and the product is passed to the banks create account method and calls the tellers account actions method.
17. account actions (main menu)
18. The teller calls its welcome method.
19. welcome method(existing customer)
20. The teller calls the list customers function that wraps the banks list clients method.
21. The user is prompted to select a customer.
22. Repeat from step 7.

Potential Pitfalls:

The most easily identifiable pitfall is that of the architecture and the role of the teller. The difficulty in clearly determining the role of the menu within the program, led to the creation of the teller. There are arguments for and against the role of the teller. This is a potential pitfall because the teller is a cornerstone of the programs architecture. This means the validity of the tellers role is tied hand in hand with the quality of the architecture.

While the role of the teller can be argued as being a poor design choice, it shall be noted the teller was not developed without being given a great deal of thought. As a result, a case can be made for the necessity of the tellers role in the programs architecture, and flow. Due to the highly subjective and abstract concept of a menu within this program, it should be further noted that a teller, helps to remove abstraction. In doing so, this removes the subjective nature of a menu and gives it a proper role within the object oriented frame work.

Test Plan:

User Test:

1. Implement a single requirement
2. Identify potential implementation flaws (accessing menu options, inserting negative values)
3. Identify the result of user Input on class attributes, and within methods (trying to add strings and numbers)
4. Identify whether the user input requires a specific type. (floats for deposits and withdraws, nits for age)
5. Address error checking of steps 2-4
6. Identify if implementation and requirements are appropriately placed within architecture.
7. Make changes to ensure the programs architecture and user interface are without flaw
8. repeat steps 1-7
9. Try to break program and repeat steps 1-7 as necessary.

Test Cases:

1. Try to input invalid values for menus (if offered option a,b,c, what happens if I select d)
2. Try to input invalid types (if 1,2,3 are options, what happens if I select a)
3. Try irrational values. (what happens if I try to withdraw or deposit negative amounts)
4. Try to get stuck in a loop (are there any potential cases where you can not exit)
5. Try to handle control c and d at any point throughout the program.
6. Check and see if certain inputs require case specific input
7. Be sure that special cases handle appropriately (not going negative in withdraws)

Conclusion:

The project reinforced countless concepts we have covered throughout our Object Oriented Programming course. The most notable of which is identifying objects. There was a great deal of confidence in determining what is and is not a object. This served to build a strong foundation for the architecture of the program.

While there was a great deal of clarity as to what constitutes an object, there are still instances where this is not as clear. One instance of such in this project are the different form of accounts. There is a strong case for why they could be subclasses. However, there is a strong argument that they are all the same, just accounts. The question is who imposes the restrictions that come with certain accounts? Are they built into the account or is it imposed by the bank. While the accuracy of the decision may not correct, the argument can be held that the logic supporting the decision is still valid. For example, an account only holds money. An account can deposit or withdraw into itself. However, restrictions such as age limits. Are these truly attributes of the account? Or , are they methods of a bank, or whoever creates the account. This can be further examined by viewing the penalties that come with drawing early. The ability to draw before the standard age is still present, however it comes with a penalty. To say the account possess the attribute to implement requirements for withdraw, would also mean the account possess the method to collect penalties. This is where the decision to make restrictions on withdrawing early a method outside of the account.

Finally, object oriented programming helps to give direction in developing the architecture of your program. While, this applies to many elements of a project, there are still many instances where

there really is no cookie cutter object oriented model. For instance the menu/teller. Perhaps this is do to a developing understanding of the concepts. However, continuous class discussion about the subjective nature of certain elements leads one to believe that there are certainly aspects of object oriented programming that do not fit the standard object approach used for the rest of the architecture.