CS3307 Group 7 Code Inspection

Structural correspondence between Design and Code:
Are all the classes and interrelationships programmed in the application explicitly represented in the class diagram of the system?

Analysis: The private and public classes of their header files correspond with those represented in the diagram
The UML diagram accounts for generalizations of the *Person* class and enumerations of *Accounts*
Functions of each class are also present in the UML Diagram

Findings: Inspecting the UML diagram and code provided by group 7 shows an exact correspondence between the diagram and the code.

Functionality:
Do all the programmed classes perform their intended operations as per the requirements?

Analysis: Running the program through test scenarios and edge cases gave us insight on how functionally complete the code was. All the required banking functions worked as intended, including warnings and invalid operations (such as transferring from a non-existent or overdrawing). The Execution Trace is the segment of code missing just *"time of system access"* and the ability to write to file before a program unexpectedly exits. The program does seem to write upon selecting 'Exit' in the program menu, but the file was not explicitly written to close.

After acceptance testing we also inspected the code to see where functionality could have been overlooked. We can see the main under the *Banking* class handles all the menus and submenus, while the *Account*, and *Person* class has their respective constructors and getters/settters. We should point out that both BankManager and Maintenance has the redundant attribute 'id_' which is already present in the *Person* class they extend.

Looking at the code we can see the execution trace never tracks system time, and additionally only closes the *ofstream* file when the execution trace is turned off, or program is Exited through the menu. Since the file is not flushed after each write the traces won't be saved if the program were to crash, a time when the execution trace is most useful.

Findings: The classes perform their intended operations as per the requirements. Only missing feature is centralized around the execution trace.

Cohesion:
Do the methods encapsulated in each programmed class, together perform a single, well defined, task of the class? (High-Cohesion: the functionalities embedded in a class, accessed through its methods, have much in common, e.g., access common data)

Analysis: The *Person* class contains functions for opening and closing accounts and their justification is sound, as a person is the one who 'owns' an account and opens and closes them.

The *Maintenance* class contains all the necessary functions and one unnecessary function that is never called, setID. Their menu system bypasses setting up an ID for a Maintenance user, so it is never used.

The *Manager* class contains functions that should belong to a manager, including the ability to open and close client accounts, views their details, and search for customers. As with the *Maintenance* class *Manager* also contains a setID function, that is hardcoded to always be 91.

Both the *Maintenance* and *Manager* subclass though extends the *Person* class, meaning they also inherit all the functions that only pertain to a *Customer Person.* Since the project specification was vague on whether a BankManager/Maintenance worker were also customers of the bank it seems fair to say that they too should have the ability to have a bank account.

Taking into account their design decision we can say their code has high cohesion, it should be noted that the class also contains a finite *Person* class array to manage all the customers. This makes sense, as a manager *has* users, but it is also a case of high-coupling; a database class for customers would be useful here.

Findings: Their code has good cohesion for the design they've chosen. Their use of a BankManager as a gatekeeper of all the clients may affect their extensibility in the future and couple classes have a redundant function.

Coupling:
Do the programmed classes have excessive inter-dependency? (High Coupling: In this case a class shares a common variable with another, or relies on, or controls the execution of, another class.)

Analysis: As mentioned in the *Cohesion* portion, the program would benefit by having a database class to manage their customers rather than being shoehorned into the *BankManager* class.

More importantly there should be a *Customer* class that extends *Person.* This enables future *Person's* to not inherit functions that may only apply to a customer. As it stands both *BankManager* and *Maintenance* worker inherits all the methods of *Person* even if they aren't a customer. By decoupling a *Customer* from a *Person* the program becomes more extendible and modular. Currently, every single *Person* is automatically a *Customer* which may not always be the case.

Findings: Their code could have lower coupling when it comes to the *Person* class. Any subclass that extends a *Person* class must inherit all the functions of a customer.

Separation of concerns:
Is the scoped problem decomposed into separate concerns where each concern is encapsulated in a construct such as a class with well-defined interface and cohesive functions with minimal of connections with other concerns?

Analysis: Each method deals with a specific concern and are well separated from code unrelated to the function. All the classes have good separation of concerns. The exception to this is in the *Banking* class where the entire menu system is squeezed into main.

While it seems to make sense since the entire class deals with the interface there are over 400+ lines of code in one method. It would have been reasonable to separate the menus into their own methods that are called upon depending on the user that is logged in. This would separate submenus from their parent menus, making development and debugging easier.

Findings: While all their methods have good separation of concerns the same could not be said about their main method, which should have a divided menu system.

Do the classes contain proper access specifications (e.g.: public and private methods)?

Analysis: Looking at their UML Diagram we can see that private attributes that are only modified by functions within the class are all protected and properly accessed using getters and setters.

Findings: All classes contain proper access specifications.

Reusability:
Are the programmed classes reusable in other applications or situations?

Analysis: Excluding the mutator methods most the code is not reusable. There are dozens of specific messages that are printed to console using ‘cout’ rather than a *Printer* method and magic numbers are constantly used.

Some code for the chequing and savings account could have been reused with the aid of a boolean flag, but instead the code was repeated with hardcoded messages.

Lastly the use of gotos in the main method makes it a very rigid program.

Findings: The code is highly specific to the task they were set to accomplish. The use of magic numbers, gotos and hardcoded messages makes it difficult to repurpose the code.

Simplicity:
Are the functionalities carried out by the classes easily identifiable and understandable?

Analysis: All the methods within their classes are logically named and related to the code. No methods are overly long, with the exception of the *Banking* class, which handles the menu system.

Methods could be improved by defining magic numbers, and more detailed variables names should be used rather than 1 letter names. There are also several cases where a function could be simplified by the use of a loop and a dynamic message rather than if-else statement.

Findings: The methods are aptly named, and carry out their functions in a succinct manner. A little more work could be done to improve the simplicity of the code, especially when it comes to repeated code.

Do the complicated portions of the code have /*comments*/ for ease of understanding?
Analysis: All classes do not contain any comments, only their header files do. The complicated nested menu system, has no comments and with the liberal use of ‘gotos’ some line numbers would be appreciated.

Findings: Both simple and complicated code are not accompanied by comments.

Maintainability:
Does the application provide scope for easy enhancement or updates? (i.e., enhancement in the code does not require too many changes in the original code (see, for example, requirements of the “enhancement” project))

Analysis: This program would be very difficult to maintain, and enhance. There is a severe lack of commenting, poorly named (not descriptive) variables, magic numbers, hardcoded errors messages printed to console, incorrect/non-existent indentation and spaghetti code.

Since C++ is a case-sensitive language the group has decided to take advantage of that and name multiple variables ‘amount’ with different casing (amount, aMount, amounT) to represent entirely different entities. This not only makes development and debugging difficult any enhancements of the application will require a thorough look through the code to see what each notation represents.

The *Person* class coupled as a customer also makes future additions to the code difficult. If the bank wanted to add a clerk, or a security worker they would have to create a new *Person* type that would not have an account with the bank.

Findings: The code is extremely difficult to maintain due to some missteps in development.

Efficiency:
Does the design introduce inefficiency in code (e.g., causes too many nested loops or delays in concurrent processing)?

Analysis: There aren't any nested loops, or delays in the code. The least efficient segment of code may be the 'find' method. It seems to iterate through the entire client array (of fixed size 1000) even though clients are numerically organized by their IDs.

Findings: Nothing overly inefficient.

Depth of inheritance:
Do the inheritance relationships between the ancestor/descendent classes go too deep in the hierarchy? (The deeper a class in the hierarchy, the greater the number of methods it will probably inherit from its ancestors, making it harder to predict its behaviour).

Analysis: Both the *BankManager* and *Maintenance* class inherit from *Person* but never use the functions within *Person*. The *BankManager* and *Maintenance* class also never use the 'cheq' or 'sav' account.

Findings: Two of their classes inherited unnecessary attributes and methods.

Children:
Does a parent class have too many children classes? (This could possible suggest an abstraction problem.)

Analysis: The *Person* class has two children, and the *Accounts* class has none.

Findings: They do not have too many children.