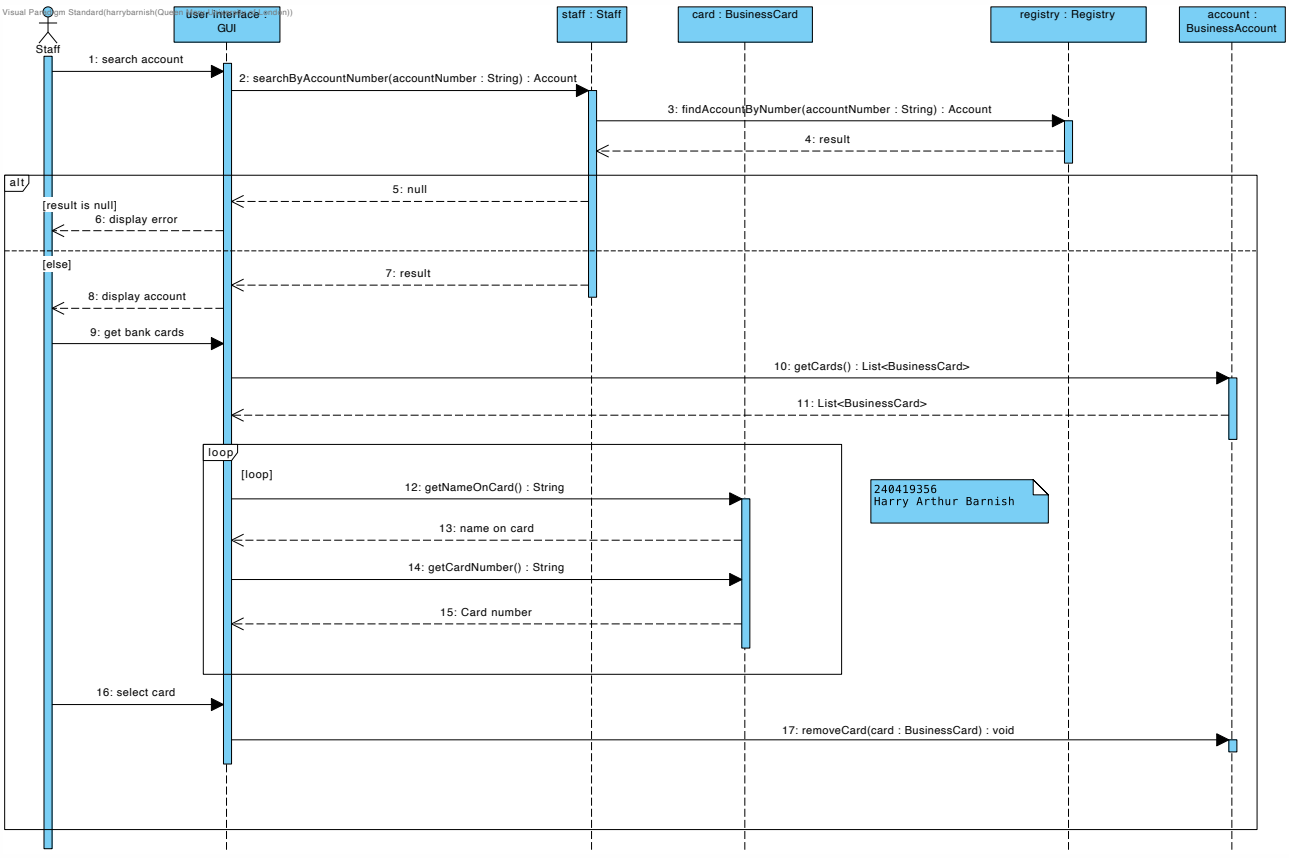
ECS505U Software Engineering Coursework II

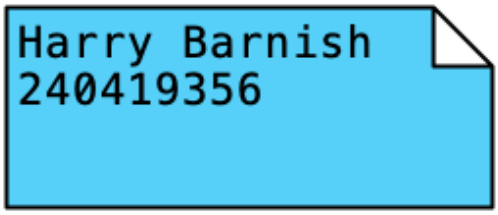
Harry Barnish 240419356

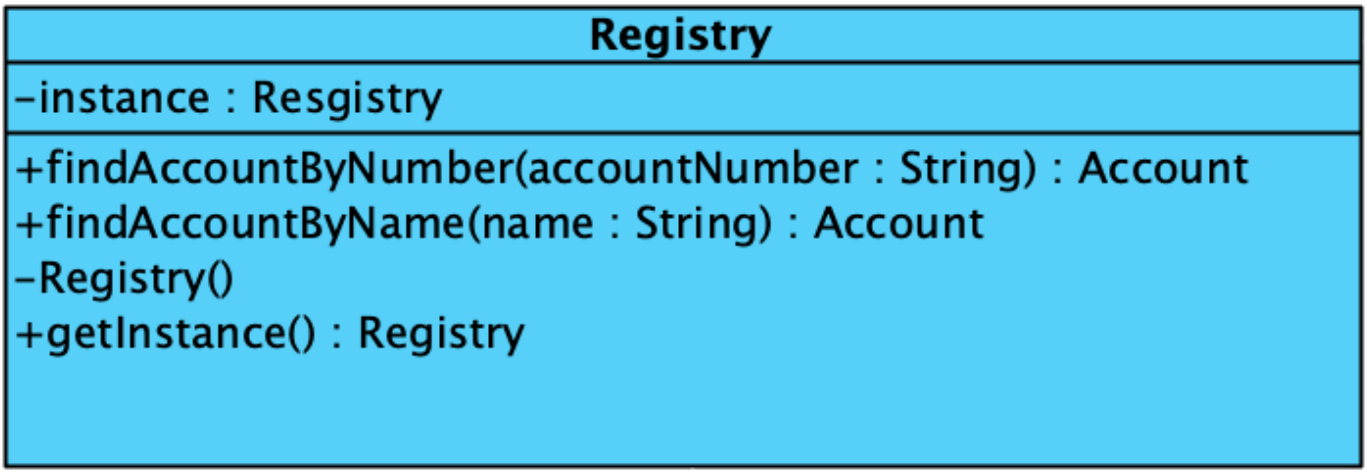
Question 1



Question 2

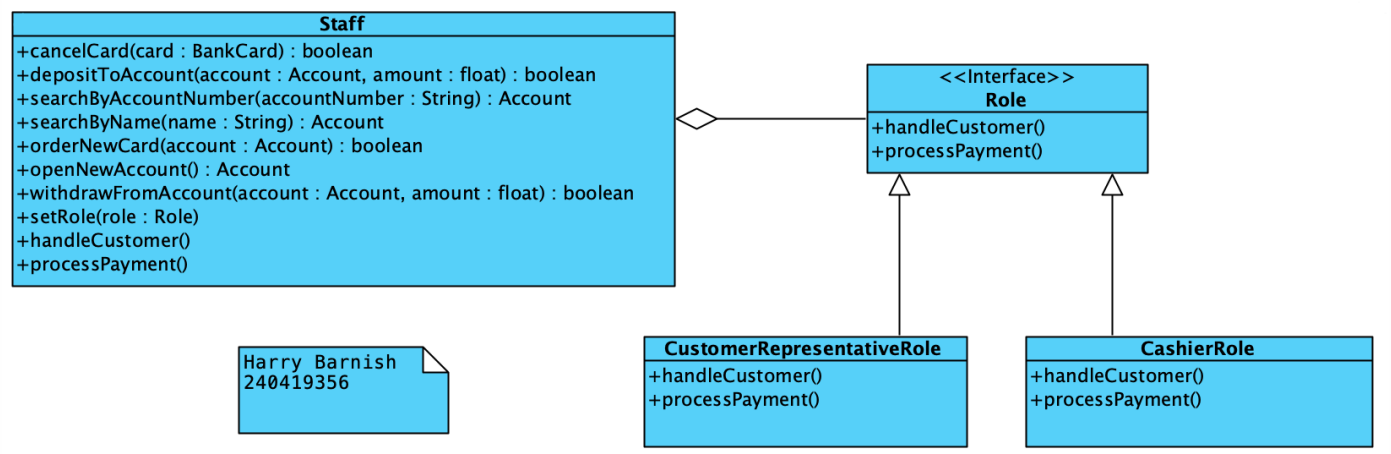
Part A (Only One Registry Object Must Exist)





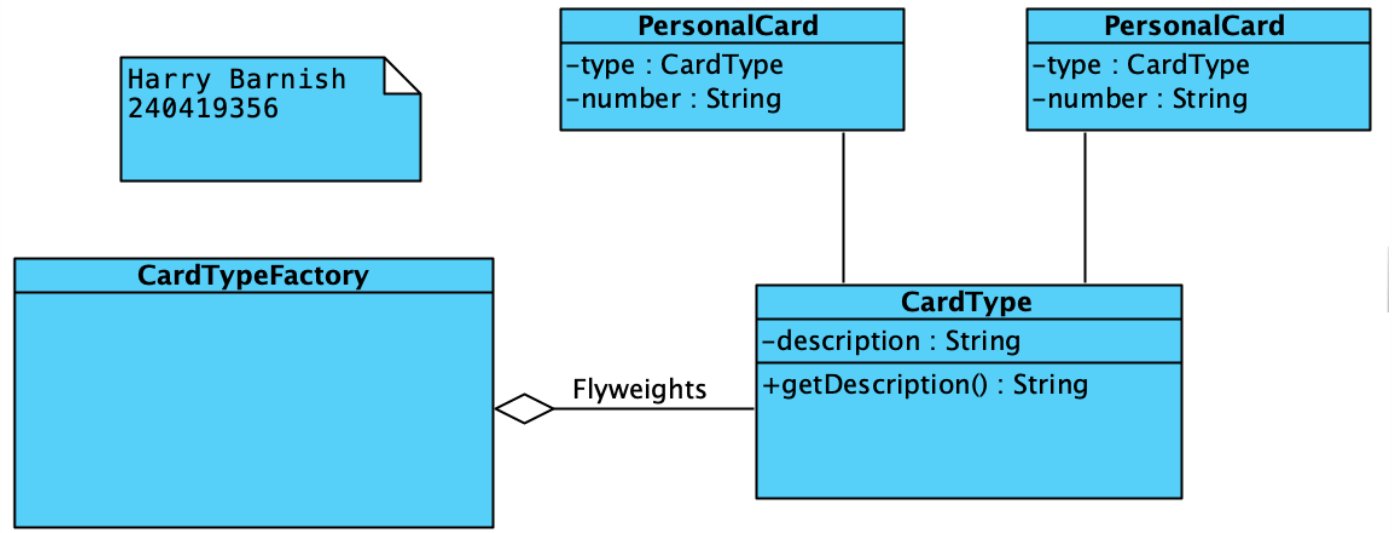
The system requires that only one registry exists to store and manage all accounts and bank cards. If we were to have more than one, we would have potential data duplication and consistency issues. The most obvious way to solve this is by using a Singleton design. This will work well as it ensures that there is only one instance of the Registry class and thus controlling all of the bank card and account information. By centralising the management of accounts and cards into one Registry class, we ensure there is no conflicting data and it allows for lookups, e.g., `findAccountByNumber()` to be performed more reliably since we know that all the accounts are in that specific Registry class and not distributed amongst many instances.

Part B (Staff Acts as Cashier or Customer Representative, Depending on Context)



Staff members in a branch need to switch between the roles of customer representative roles and cashier roles frequently throughout the day. What role they take depends on their workstation. To allow for this without hassle, I have implemented a Strategy pattern. Instead of having different staff roles, which will result in duplicate logic, I have an interface called `Role` which `CustomerRepresentativeRole` and `CashierRole` both implement. The Staff class maintains a reference to a Role strategy and delegates operations such as `handleCustomer()` and `processPayment()` directly to the active strategy. The active strategy can be changed by calling the `setRole()` function in the Staff object. This allows for staff members to be able to switch roles easily at runtime.

Part C (Card Type Description is Shared Across All Cards; Avoid Memory Waste)



Each bank card type, including personal and business cards, has a consistent description across all cards of that type. Storing this information separately in each card object leads to unnecessary duplication and increased memory usage. Especially as card numbers increase. To solve this, I have used the Flyweight design pattern. The shared intrinsic data, such as the card type description, is put into a separate `CardType` flyweight object and a `CardTypeFactory` object manages these objects to ensure only one instance exists for each card type. Then, individual BankCard instances reference their relevant `CardType` rather than storing the description themselves, thus reducing the dubplication and memory usage while still allowing for each card to keep its own attrubites, E.g., card number, owner name etc. The Flyweight pattern offers an efficient and clean solution that meets the requirement to minimise memory usage when handling a large number of bank cards.