SG Bank Assessment

Before starting this assessment be sure to watch the SG bank videos in the LMS and download the SGBank source code from the section files.

Throughout the course we have been adding additional layers of abstraction. The C# language starts with an abstraction over memory and processing with our basic variables, loops, arrays, and flow of control statements. Next we looked at classes and how we could create an object definition as a more complex abstraction that is used to store structured data and perform actions (methods). Interfaces and polymorphism are the next level of abstraction and are unfortunately the most difficult one to master. Instead of working with a **concrete type** like the FreeAccountDepositRule, we are coding against an **abstract type** like IDeposit.

Fortunately, in an entry level position you are rarely expected to come up with an architecture that supports this. You are more likely to be asked to use existing interfaces and extend the functionality of a pre-existing system. This assessment is intended to help you go through these processes and prepare you for the mastery project, which will use a similar structure coded from scratch. Implementing new concrete types and modifying existing code to use these types is a pattern, and like any pattern practice will build familiarity and comfort.

Rules

There are 3 types of accounts in our system:

|  |  |
| --- | --- |
| Account | Description |
| Free | * Cannot deposit or withdraw more than $100 at a time * Deposits must be a positive number * Withdrawals must be a negative number * Cannot overdraft the account |
| Basic | * Can deposit any positive amount * Withdrawals must be negative and cannot exceed $500 * Can overdraft up to $100 not including a $10 fee deducted from the balance |
| Premium | * Can deposit any positive amount * Withdrawals must be negative with no limit except the overdraft limit * Can overdraft up to $500 with no fee. |

Scenario

Your senior developer has gotten the SG Bank project started and completed the free account deposit rule, but has been called away to other tasks. You met with them yesterday, and they provided an overview of how they created the application up to this point and have left you task instructions on what needs to be done to finish the application.

Finishing Free Account Testing DONE

Our free account loading object returns the account no matter what the input, this prevents us from testing the account not found logic. Let’s refactor our LoadAccount() method to check the account number properly.

|  |  |
| --- | --- |
| Task | Description |
| Test the account not found logic in the UI | * Modify the FreeAccountTestRepository LoadAccount() method to check that the incoming account number is equal to the account number of the \_account field. Return null if it isn’t. * Run the application and test the lookup account functionality to verify that 12345 returns account information and a different entry gives an error message. * In the CanLoadFreeAccountTestData() method of the FreeAccountTests in the test project update the LookupAccount() method call to pass in the account 12345 as the parameter * Run your unit test and ensure it passes |

Create the IWithdraw Interface DONE

Since we have several withdraw behaviors we will want to create an IWithdraw interface to represent these behaviors so that we can substitute behaviors on demand. This is the same pattern as the IDeposit interface.

|  |  |
| --- | --- |
| Task | Description |
| Create the IWithdraw interface | * In the SGBank.Models.Interfaces folder, add a new public interface IWithdraw |
| Create the return object for the Withdraw() method we will create in the IWithdraw interface | * In the SGBank.Models.Responses folder, add a new public class AccountWithdrawResponse * Inherit from the Response type * Add public properties for the following: * Account * OldBalance * Amount |
| Add the Withdraw() method to the interface | * Add a method to the IWithdraw interface named Withdraw that returns an AccountWithdrawResponse and receives an Account and a decimal amount as parameters. |

Create the WithdrawRulesFactory DONE

We need a factory class to instantiate the correct IWithdraw implementation depending upon the account type that is loaded by the account manager.

|  |  |
| --- | --- |
| Task | Description |
| Create the WithdrawRulesFactory | * Add a folder to the SGBank.BLL project called “WithdrawRules” * Add a class called WithdrawRulesFactory to the folder |
| Create the factory method that will return an IWithdraw implementation | * In the WithdrawRulesFactory class add a public static method called Create() which returns an IWithdraw type and receives an AccountType as a parameter. * In the Create() method, throw a new NotImplementedException for now. |

Create the Withdraw() method on the AccountManager class DONE

The AccountManager class is in charge of communicating with the user interface. We need a method for the UI to call for withdrawals:

|  |  |
| --- | --- |
| Task | Description |
| Add the Withdraw() method | * Add a method to AccountManager named Withdraw() that returns an AccountWithdrawResponse and receives an accountNumber and amount as parameters. |
| Write the Withdraw() method implementation | * Instantiate an AcountWithdrawResponse object * Assign the account property of the response object to the result of the LoadAccount() method of the account repository field. * Check to see if the account property is null * If null, set the response object Success property to false and the Message should notify the caller that the account number is not valid. * Return the response object * Declare a local variable of type IWithdraw to hold the IWithdraw implementation. * Assign the IWithdraw variable to the WithdrawRulesFactory.Create() method * Assign the response object to the IWithdraw variables Withdraw() method. * If the response.Success property is true, save the account information using the account repository field. * Return the response object. |

Create the FreeAccountWithdrawRule DONE

We are ready to create our first rule for the factory to create.

|  |  |
| --- | --- |
| Task | Description |
| Add a new class | * Add a class named FreeAccountWithdrawRule to the BLL\WithdrawRules folder. * Make the class public and implement the IWithdraw interface (you will need a using directive for SGBank.Models. |
| Write the Withdraw() method implementation | * Instantiate an AcountWithdrawResponse object * If the account type is not Free * Set the Success bool to false * Set the Message to “Error: a non-free account hit the Free Withdraw Rule. Contact IT” * return the response * If the amount is greater than or equal to zero * Set the Success bool to false * Set the Message to “Withdrawal amounts must be negative!” * return the response * If the amount is less than negative one hundred * Set the Success bool to false * Set the Message to “Free accounts cannot withdraw more than $100!” * return the response * If the current account balance plus the amount is less than zero * Set the Success bool to false * Set the Message to “Free accounts cannot overdraft!” * return the response * If checks 2-4 were successful we can debit the account and fill out the response object: * Set the Success bool to true * Set the Account property to the account parameter * Set the Amount equal to the amount parameter * Set the OldBalance equal to the account balance * Set the Account Balance equal to itself plus the amount * Return the response object |

Update the WithdrawRulesFactory DONE

Now that we have a rule, we can start updating our factory class to return concrete instances of rules.

|  |  |
| --- | --- |
| Task | Description |
| Replace the throw statement with a switch statement | * In the Create() method of the WithdrawRulesFactory * Delete the throw statement * Add a switch statement for the type parameter * Add a case for the Free AccountType * return a new FreeAccountWithdrawRule instance in the case * Throw a new exception with the message “Account type is not supported!” outside the case statement |

Unit Test the Rules DONE

We are not finished with our business logic unless we test it! To test our rules and all of their cases (amounts, balances, etc.) we will need to create test cases that pass in all the data needed for an account object, the amount we wish to test, and the expected test result.

|  |  |
| --- | --- |
| Task | Description |
| Create the method for the free account deposit rule tests | * In the FreeAccountTests class add a new void method named FreeAccountDepositRuleTest with the following parameters: * string accountNumber * string name * decimal balance * int accountType * decimal amount * bool expectedResult |
| Create the test cases | * Add four [TestCase] attributes to the FreeAccountDepositRuleTest() method: * Case 1 (fail, too much deposited) * “12345” * “Free Account” * 100 * 1 * 250 * false * Case 2 (fail, negative number deposited) * “12345” * “Free Account” * 100 * 1 * -100 * false * Case 3 (fail, not a free account type) * “12345” * “Free Account” * 100 * 2 * 50 * false * Case 4 (success) * “12345” * “Free Account” * 100 * 1 * 50 * true |
| Write the test implementation | * Create a variable of type IDeposit and assign it to a new FreeDepositRule instance * Create a variable of type Account and assign it a new Account instance * Set the properties of the account object to the appropriate method parameters * Create a variable of type AccountDepositResponse and assign it to the result IDeposit variable’s Deposit() method. * Assert that the expectedResult is equal to the response’s Success property |
| Run your tests | * The tests should all pass. |
| Repeat for the withdraw rule | * Create a void method FreeAccountWithdrawRuleTest with the same parameters. Since the process for testing withdrawals is similar to deposits, use the previous implementation as a template. * Add [TestCase] attributes to cover all of the Withdraw scenarios per the requirement specification. * Positive withdrawal amount (fail) * Negative withdrawal over limit (fail) * Wrong account type (fail) * Overdraft (fail) * Successful withdrawal (succeed) * Write the test implementation. Use the FreeAccountDepositRule test as a template, swapping out deposit objects for withdraw objects. * All of the tests should pass before moving on. |

Update the UI to Support Withdrawals DONE

Now that our rules are working, we need to add the functionality to the user interface.

|  |  |
| --- | --- |
| Task | Description |
| Create the workflow class | * In the Workflows folder of the UI project add a new public class called WithdrawWorkflow |
| Create and implement the Execute() method | * Add a public void method called Execute() to the workflow class. You may use the Execute() method of the DepositWorkflow as a template. * Clear the Console * Create a local variable of type AcccountManager and assign it to the result of the Create() method of the AccountManagerFactory * Prompt the user for an account number, store it in a variable. * Prompt the user for a withdrawal amount, store it in a variable * Create a local variable of type AccountWithdrawResponse and assign it to the result of the account manager Withdraw() method. * If the response Success property is true output a success message and receipt to the console similar to the DepositWorkflow * If the Success property is false print the error message * Prompt the user to press any key to continue and use the ReadKey() console method to pause until the user is ready to return to the menu. |
| Add the withdraw workflow to the menu | * In case “3” of the Start() method of the Menu class: * Create a variable of type WithdrawWorkflow and instantiate it. * Call the Execute() method of the instance |

Test the UI for Free Account Withdrawals DONE

|  |  |
| --- | --- |
| Task | Description |
| Test withdrawals | * In the UI test all cases of withdrawals * Account number that does not exist * Positive withdrawal amount * Negative withdrawal amount over the limit * Successful withdrawal * Overdraft withdrawal |

Implementing the Basic Account Functionality

Now that we have completed and tested the free account logic, we are ready to implement the basic account rules and logic. During this part of the assessment, pay close attention to the idea that by declaring interfaces and coding against them in our AccountManager class that now adding new rules and account types is a matter of simply adding a few new classes and updating the factory methods. We do not have to modify the UI or the AccountManager classes at all since the interfaces have made our rules and behaviors swappable.

Create a BasicAccountTestRepository DONE

|  |  |
| --- | --- |
| Task | Description |
| Create class & implement interface | * In the Data project add a new public class called BasicAccountTestRepository. * Implement the IAccountRepository interface * Using the FreeAccountTestRepository as a template: * Create a private static field of type Account instantiated with sample data * Name = “Basic Account” * Balance = 100M * AccountNumber = “33333” * Type = AccountType.Basic * Use the same Load and Save implementation from the FreeAccountTestRepository |

Add a NoLimitDepositRule DONE

The basic and premium accounts have no limit on the deposit amounts and can share this implementation. The still require checking the amount values and account types however.

|  |  |
| --- | --- |
| Task | Description |
| Add a new class | * Add a class named NoLimitDepositRule to the BLL\DepositRules folder. * Make the class public and implement the IDeposit interface (you will need a using directive for SGBank.Models. |
| Write the Deposit() method implementation | * Instantiate an AcountDepositResponse object * If the account type is not Basic and the account type is not Premium * Set the Success bool to false * Set the Message to “Error: Only basic and premium accounts can deposit with no limit. Contact IT” * return the response * If the amount is less than or equal to zero * Set the Success bool to false * Set the Message to “Withdrawal amounts must be negative!” * return the response * If checks 2-4 were successful we can debit the account and fill out the response object: * Set the Success bool to true * Set the Account property to the account parameter * Set the Amount equal to the amount parameter * Set the OldBalance equal to the account balance * Set the Account Balance equal to itself plus the amount * Return the response object |

Update the DepositRulesFactory DONE

Now that we have a new rule, the factory class that instantiates the rules must be updated.

|  |  |
| --- | --- |
| Task | Description |
| Update Create() method | * In the Create() method of the DepositRulesFactory add an additional cases for Basic account types that returns a new NoLimitDeposit object instance. |

Unit Test the NoLimitDepositRule DONE

The unit test for the NoLimitDepositRule will be very similar to the FreeAccountDepositRule excepting that there is no test case for an over the limit deposit.

|  |  |
| --- | --- |
| Task | Description |
| Create the test class | * Add a new public class named BasicAccountTests to the SGBank.Tests project * Place the [TestFixture] attribute on the class |
| Create the method for the free account deposit rule tests | * In the BasicAccountTests class add a new void method named BasicAccountDepositRuleTest with the following parameters: * string accountNumber * string name * decimal balance * int accountType * decimal amount * bool expectedResult |
| Create the test cases | * Add three [TestCase] attributes to the FreeAccountDepositRuleTest() method: * Case 1 (fail, wrong account type) * “33333” * “Basic Account” * 100 * 1 * 250 * false * Case 2 (fail, negative number deposited) * “33333” * “Basic Account” * 100 * 2 * -100 * false * Case 3 (success) * “33333” * “Basic Account” * 100 * 2 * 250 * true |
| Write the test implementation | * Create a variable of type IDeposit and assign it to a new NoLimitRule instance * Create a variable of type Account and assign it a new Account instance * Set the properties of the account object to the appropriate method parameters * Create a variable of type AccountDepositResponse and assign it to the result IDeposit variable’s Deposit() method. * Assert that the expectedResult is equal to the response’s Success property |
| Run your tests | * The tests should all pass. |

Add a BasicAccountWithdrawRule DONE

We continue following the pattern for interface applications by creating new classes that implement the existing interfaces.

|  |  |
| --- | --- |
| Task | Description |
| Add a new class | * Add a class named BasicAccountWithdrawRule to the BLL\WithdrawRules folder. * Make the class public and implement the IWithdraw interface (you will need a using directive for SGBank.Models. |
| Write the Withdraw() method implementation | * Instantiate an AcountWithdrawResponse object * If the account type is not Basic * Set the Success bool to false * Set the Message to “Error: a non-basic account hit the Basic Withdraw Rule. Contact IT” * return the response * If the amount is greater than or equal to zero * Set the Success bool to false * Set the Message to “Withdrawal amounts must be negative!” * return the response * If the amount is less than negative five hundred * Set the Success bool to false * Set the Message to “Basic accounts cannot withdraw more than $500!” * return the response * If the current account balance plus the amount is less than -100 * Set the Success bool to false * Set the Message to “This amount will overdraft more than your $100 limit!” * return the response * If checks 2-4 were successful we can debit the account and fill out the response object: * Set the Success bool to true * Set the Account property to the account parameter * Set the Amount property equal to the amount parameter * Set the OldBalance equal to the account balance * Set the account balance equal to itself plus the amount * If the account balance is negative deduct an additional $10 from the balance as the overdraft fee * Return the response object |

Update the WithdrawRulesFactory DONE

The pattern requires we update our factory to enable the new rules.

|  |  |
| --- | --- |
| Task | Description |
| Update Create() method | * In the Create() method of the WithRulesFactory add an additional case for Basic account types that returns a new BasicAccountWithdrawalRule object instance. |

Unit Test the BasicAccountWithdrawRule DONE

Because we have introduced an overdraft fee, we will need an extra parameter in our test case for the new balance to check that the fee was applied. We only need to check this if the response is successful.

|  |  |
| --- | --- |
| Task | Description |
| Create the method for the free account deposit rule tests | * In the BasicAccountTests class add a new void method named BasicAccountWithdrawRuleTest with the following parameters: * string accountNumber * string name * decimal balance * int accountType * decimal amount * decimal newBalance * bool expectedResult |
| Create the test cases | * Add five [TestCase] attributes to the BasicAccountWithdrawRuleTest () method: * Case 1 (fail, too much withdrawn) * “33333” * “Basic Account” * 1500 * 2 * -1000 * 1500 * false * Case 2 (fail, not a basic account type) * “33333” * “Basic Account” * 100 * 1 * -100 * 100 * false * Case 3 (fail, positive number withdrawn) * “33333” * “Basic Account” * 100 * 2 * 100 * 100 * false * Case 4 (success) * “33333” * “Basic Account” * 150 * 2 * -50 * 100 * true * Case 4 (success, overdraft fee) * “33333” * “Basic Account” * 100 * 2 * -150 * -60 (100 – 150 – 10) * true |
| Write the test implementation | * Create a variable of type IWithdraw and assign it to a new BasicAccountWithdrawRule instance * Create a variable of type Account and assign it a new Account instance * Set the properties of the account object to the appropriate method parameters * Create a variable of type AccountWithdrawResponse and assign it to the result of the IWithdraw variable’s Withdraw() method * Assert that the expectedResult is equal to the response’s Success property * If the Success property is true, also assert that the newBalance parameter is equal to the response’s account balance |
| Run your tests | * The tests should all pass. |

Test the UI for Basic Account Deposits and Withdrawals DONE

|  |  |
| --- | --- |
| Task | Description |
| Update AccountManagerFactory | * In the AccountManagerFactory class, add a case for “BasicTest” that returns a new account manager whose constructor takes a new BasicAccountTestRepository object. |
| Update app.config | * In the app.config file of the SGBank.UI project change the “Mode” key value to “BasicTest” |
| Test withdrawals | * In the UI test all cases of withdrawals * Account number that does not exist * Positive withdrawal amount * Negative withdrawal amount over the limit * Successful withdrawal * Overdraft withdrawal |

Now You Try!

To complete the assessment, please implement the remaining logic to work with the premium account type:

* Add a premium account test repository with a sample account. DONE
* Update the DepositRulesFactory to use the NoLimitDepositRule for premium account types DONE
* Add the premium withdrawal rule to the BLL and implement the logic per the specification DONE
* Update the WithdrawRulesFactory to return your premium rule for premium accounts DONE
* Unit test your premium withdrawal rule DONE
* Test the UI for premium accounts DONE

Making the Application Live

Create a FileAccountRepository that implements IAccountRepository to read and write account data from a file named Accounts.txt. See the previous section System.IO samples on how to read and write data from files.

Your file data should be structured like this:

AccountNumber,Name,Balance,Type

11111,Free Customer,100,F

22222,Basic Customer,500,B

33333,Premium Customer,1000,P

The type should be F for free, B for basic, and P for premium. You will have to write logic to translate these codes to the correct AccountType enum.

Don’t forget to update your app.config mode to switch to using the file repository!

Bonus Challenge (Optional)

Instead of using factory classes, some development teams leverage IOC containers such as Ninject and Unity. Once your application is working, create a copy of it and in that copy attempt to remove the factory classes and replace them with Ninject using the Kernel object and Modules that are loaded based on the config file Mode value.