POINT OF SALES SYSTEM

Test Oriented Software Development – Final report

Saad Abdullah Gondal
Matriculation Number: 244527

Table of Contents

System Description and Implementation:

This is a general-purpose POS system that can be implemented in small stores. It manages stores (Organizations), users (Operators), items to sell, their stocks and records the sales (Transactions).

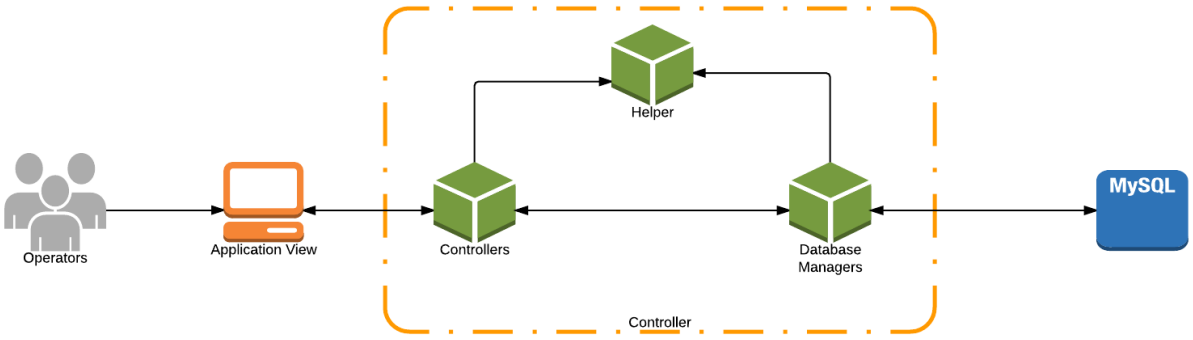
For this project, it has been assumed that the user View has been implemented already and the system receives user inputs. The database schema and business logic has been implemented. These business methods are invoked through test cases and they manage the communication with the database.

Data Definition:

- Organization:
  - A single independent shop that controls its items, stocks and sales
- Operator:
  - The person who is responsible to check out customers
  - Can add or remove stocks of items
  - Can give a discount
- Items:
  - The articles/products/units that are sold in the store.
- Transaction Line:
  - An item, its quantity and price that is sold
- Transaction:
  - A list of transaction lines.
  - Calculates the total price and generates bill.

System Architecture

It is a 3-tier architecture containing Application View, Controller and Database Model.



Application view:

A web or a desktop application that takes the user input and forwards it to the core business area, that is, the Controller. This part is assumed to be completed and properly running for the sake of this project.

Controller:

Receives the user inputs from the Application View and performs business rules. Controller is sub-divided into three types of classes ‘or packages’.

- Controllers: receives inputs from the Application View and implements business logic. They make use of Helpers in order to structurally send data to Database Managers.
- Helpers: Act as a helper object which makes it convenient for the Controllers and Database Managers to perform data communication.
- Database Managers: Receive data from Controllers by the help of Helpers and then communicate with the database

Model:

Database model which is implemented in MySQL.

Unit Requirements

RE-1: Add new Items

Description

The operator shall be able to add new products into the system. These are the items that system stores and use them as a reference for keeping track of Stocks.

Dependency

A valid Organization and Operator's credentials.

RE-2: Add a new Stock

Description

The operator shall be able to add a single stock of the item present in the system. The application need these stock values to perform validations and generate receipts for the sales.

Dependency

RE-1: Stocks shall be only added if corresponding items exist in the system.

RE-3: Add multiple stocks per item

Description

The operator shall be able to add multiple stocks of the same item. This functionality is provided to allow the users to renew the stocks of the items before they are completely sold out.

Dependency

RE-1: Stocks shall be only added if corresponding items exist in the system.

RE-4: Current Stocks calculation

Description

The system shall keep a sum of all stock quantities related to items. When a new stock is added, the quantities should be calculated. These are used to perform validations.

Dependency

RE-3: Stocks need to be added to quantify them.

RE-5: Stocks validation

Description

The system shall validate the stock quantities before they can be sold. These quantities should always be greater than 0. They can never be negative and quantities with 0 values cannot be sold. Moreover, they should be greater than or equal to the amount that is supposed to be checked out by the customer.

Dependency

RE-4: Calculated stocks are used to perform these validations.

RE-6: Check-out Items based on Stocks

Description

The system shall check out the items and properly remove the sold values from the stocks count. The system shall start subtracting values in the ascending order of stocks arrival, that is, oldest stocks should be sold first then the newer ones.

The stocks that have emptied should be marked 'inactive'.

Retail price of each stock should be used to check out.

Dependency

RE-5: Stocks should be validated before making a transaction.

RE-7: Perform transaction and create a bill

Description

The system shall perform and maintain transaction records. The retail price of each item should be calculated along with the quantity and a transaction line must be created. These transaction lines should be added up to create a total bill.

Dependency

RE-6: Stocks shall be checked out to create a transaction and a total bill

RE-8: Perform transaction and create bill according to discount rules

Description

If the quantity of a single item is more that 15, a 2% discount is offered by the store. For more that 50 units of a single item, a discount of 5% and 10% for 100 units to 500 units. If the quantity goes beyond 500, the system gives a 20% discount.

Dependency

RE-6: Stocks shall be checked out to create a transaction and a total bill.
RE-7: Transaction must be performed to use discount rules.

Test Strategy

Risk Analysis

At the time of testing, only the controller and database tiers are developed. The controller needs inputs from the User View in a structured way. Converting the exact values that should be sent from GUI to a Unit Test is a bit risky. This is because, from GUI the operator can see the Items and click on them to send back to the system. The system takes as input, the Item IDs, therefore during testing these components, the tester needs to enter the Item IDs in the Collection for the functions to work properly.

Test design methods – Component Testing

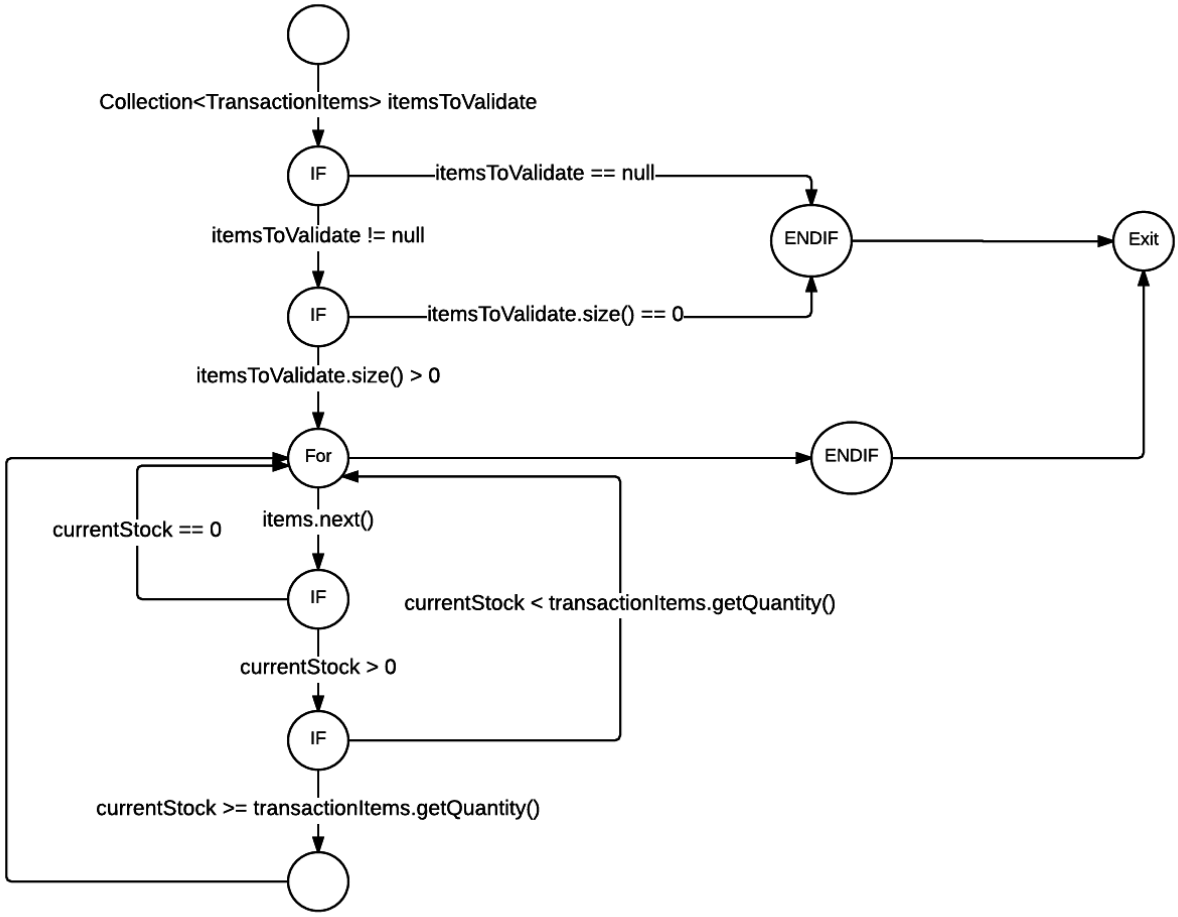
TransactionsManagement

This is the core application component that generates a transaction and checks out the customer for the selected items. This class has two functions validateStocksToSell and checkOutCustomer. They were tested individually according to the strategies mentioned forward.

Component: validateItemsToSell

COMPONENT	validateItemsToSell
TEST OBJECT	Function
DESCRIPTION	This function is called when the Operator clicks to generate bill. It receives the selected items and validates them for their stocks. For each item, it checks if there are any stocks available in the system or not. Moreover, it also checks if the quantity that is requested is available in the system or not.
TEST TYPE	Unit
TEST STRATEGY	White-box – Branch Testing and Coverage
TEST EXIT CRITERIA	90% Branch Coverage
REASONING	This is a critical function which acts as a source of valid items for the next function. The items that are validated through this function must have available stock quantity. Also, the requested quantity should be lesser than or equal to the stock quantity available in the system. Due to this criticality, this function is tested through white-box technique and hence at least 90% coverage is needed to pass this function.

Control Flow of function validateStocksToSell()



Total number of branches = 14

Component: checkOutCustomer

COMPONENT	checkOutCustomer
TEST OBJECT	Function
DESCRIPTION	Once items are validated, the collection is passed on to this function to calculate the final bill. The system allows some fixed discount percentages depending on the quantity bought. The total bill is calculated based on these discount prices. The discount rules are mentioned in the **Unit Requirement (RE-8).**
TEST TYPE	Functional Testing
TEST STRATEGY	Black-box – Equivalence Class Partitioning
TEST EXIT CRITERIA	85% Equivalence Class Coverage
REASONING	At least 85% coverage is needed as all the values from equivalence classes should act according to the business rules. This strategy is used as this function incorporates the core business logic implementation. The function expects different 'Quantity' values and decides what kind of **Functional Requirement** to implement.

Following are the Equivalence classes that will cover all the Functional requirements as mentioned in the **Unit Requirement (RE-8)**.

Valid Equivalence Classes

Parameter	Equivalence classes	Representative
Quantity	0 < x <= 15 (0% Discount)	9
	15 < x <= 50 (2% Discount)	25
	50 < x < 100 (5% Discount)	80
	100 <= x <= 500 (10% Discount)	360
	x > 500 (20% Discount)	900

Invalid Equivalence Classes

Parameter	Equivalence classes	Representative
Quantity	x = 0	0
	x < 0	-97
	NaN (Not a Number)	"hi"

Test design methods – Integration Testing - TransactionsManagement

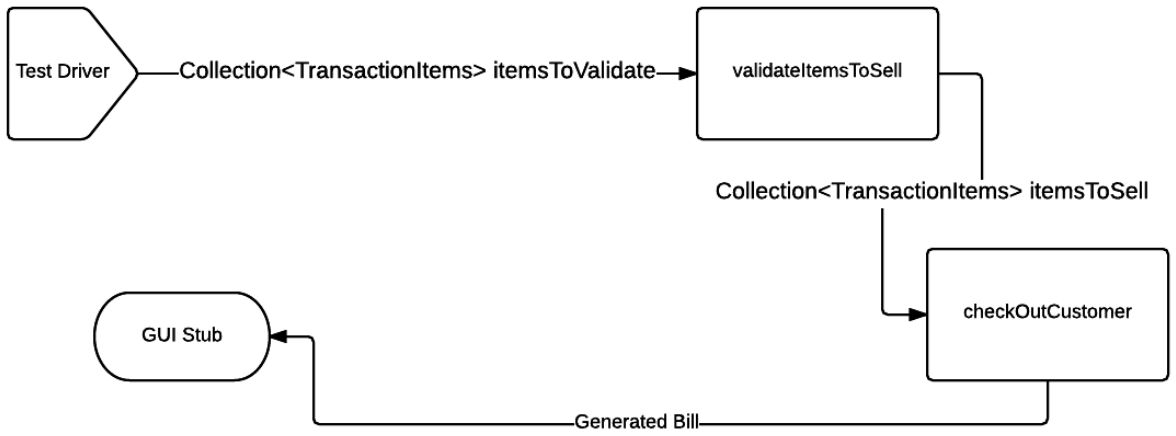
The components that were tested during the Component Testing were dependent on each other.

INTEGRATION STRATEGY	Ad hoc integration
COMPONENTS	validateItemsToSell + checkOutCustomer
DESCRIPTION	This module is responsible for validating the items to sell and then applying discount rules to generate bill.
TEST OBJECT	Class - TransactionsManagement
TEST STRATEGY	Black-box – Equivalence Class Partitioning
TEST EXIT CRITERIA	83% Equivalence Class Coverage
REASONING	This module requires a set of input values and an expected output can be tested. Therefore, a set of valid and invalid Equivalence Classes are needed to check the outputs. Ad-hoc integration is used since this is the core of the application and was finished first.

For ad-hoc integration, **Stubs** and **Test Drivers** are required.

Test Driver: It is a unit test that would generate values according to the equivalence portioning.

Stub: It is the graphical user interface where generated bill will be displayed.



Following are the Equivalence classes that will cover all the Functional requirements of this complete module.

Valid Equivalence Classes

Parameter	Equivalence classes	Representative
Item ID	x | x exists in database	5
Quantity	0 < x <= 15 (0% Discount) 15 < x <= 50 (2% Discount) 50 < x < 100 (5% Discount) 100 <= x <= 500 (10% Discount) x > 500 (20% Discount)	9 25 80 360 900

Invalid Equivalence Classes

Parameter	Equivalence classes	Representative
Item ID	x < 0 x = 0 x | x does not exist in database NaN (Not a Number)	-20 0 400 “to”
Quantity	x = 0 x < 0 NaN (Not a Number)	0 -97 ‘hi’

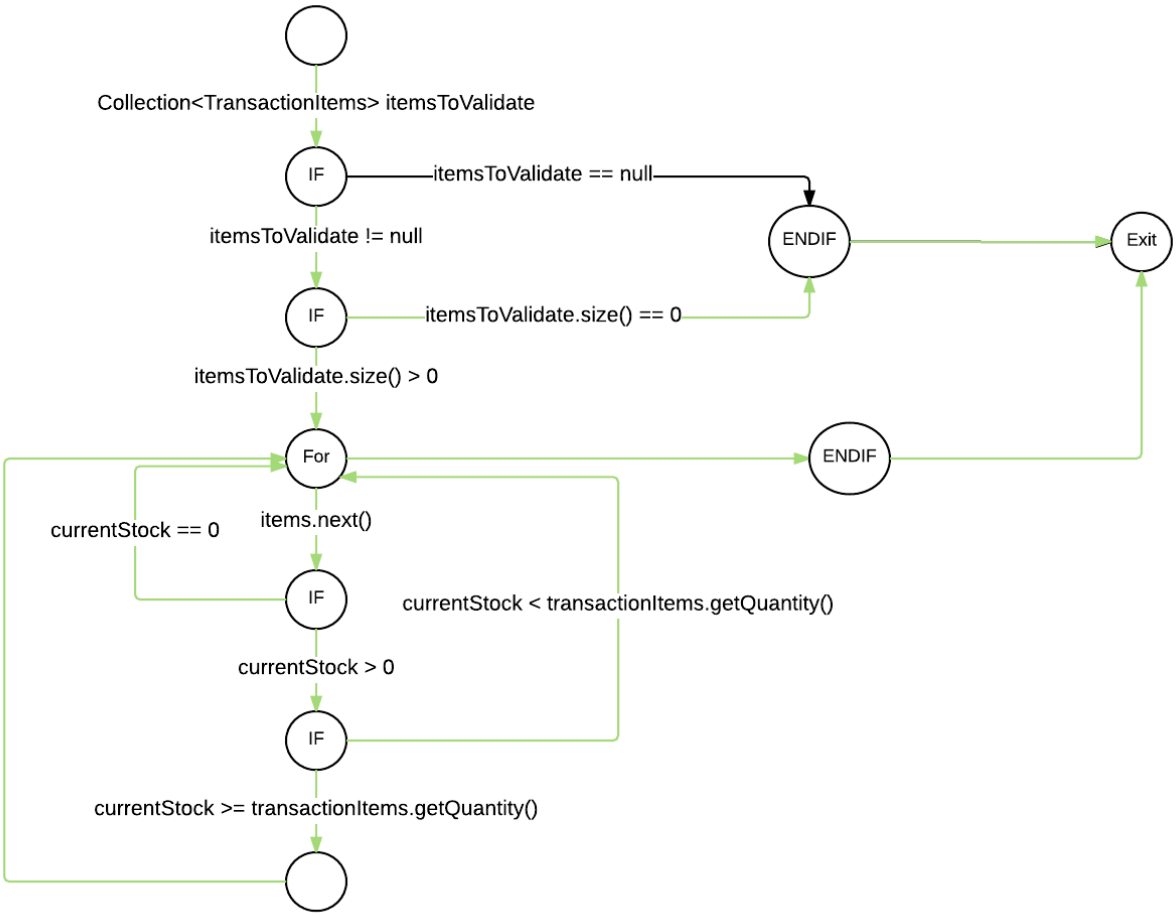
Dynamic Test Cases

Component: validateItemsToSell

This function was tested using **Branch Coverage**. Following are the test cases that show steps to perform a maximum code coverage in this function.

Test Case Code	Title	Description	Precondition	Test Steps	Expected Result
					Branch Coverage
TC-01	Validate 5 items with no stock entries	Insert a collection of items who have no stock values.	All items must exist in the system. The collection shall be populated and passed as argument to the function.	1. Add 5 different items into the system. 2. Go to items check out screen and select all 5 of them from the menu. 3. Click check out button from the screen.	Validation cycle runs and ignore all items and throw a notification to the operator that no valid stocks exist. Expected Result = 0
					(7 / 14) * 100% = 50%
TC-02	Validate 5 items with 2 invalid items	Insert a collection of 5 items in which 2 of them will not have any valid stocks. Hence the validated list shall contain only 3 items.	All items must exist in the system. The collection shall be populated and passed as an argument to the function.	1. Add 5 different items into the system. 2. Add stocks with quantities and prices for 3 of them. Leave 2 of them. 3. Go to items check out screen and select all 5 of them from the menu. 4. Click check out button from the screen.	Validation cycle runs and creates a list of 3 items for which the valid stocks exist. Notifies the operator about the 2 items for which there are no valid stocks. Expected Result = 3
					(10 / 14) * 100% = 71.43%
TC-03	Validate 5 items	Insert a collection of 5 items with valid stocks for all 5 of them.	All items must exist in the system. The collection shall be populated and passed as an argument to the function.	1. Add 5 items into the system. 2. Add stocks with quantities and prices for all 5 of them. 3. Go to the items check out screen and select all 5 of them from the menu. 4. Click check out button from the screen.	Validation cycle runs and creates a list of all 5 items. Expected Result = 5
					(9 / 14) * 100% = 64.3%
TC-04	Empty collection	Insert a collection which		1. Go to items check out screen.	Generates the notification that no

		has no items selected		2. Click check out button.	items are selected from the checkout menu. Expected Result = 0
					(4 / 14) * 100% = 29%
TC-5	Validate 1 item where quantity to sell is more that the stock available	Insert a collection with 1 item. This item should have a stock quantity which is less than the quantity which is to be sold.	The item must exist in the system and its stock should be added.	1. Add an item into the system. 2. Add its stock with the quantity of 50. 3. Go to checkout screen and select the item. 4. Give the quantity of 100. 5. Click checkout button from the screen.	The system generates the notification that the item quantity is more that the current available stock. Expected Result = -97
					(8 / 14) * 100% = 57.14%



Figure#4: Control Flow Diagram of validateItemsToSell function. Green lines showing the branches covered by all mentioned test cases.

So according to the formula:
Branch Coverage = (number of executed branches / total number of branches) x 100%
Branch Coverage = (13 / 14) * 100% = 99%
The branch coverage should be at least 90% for this function. In this way, the branches with most impact would be covered and validated items would be forwarded for further processing to the next function. The branch that is not covered is because the compiler gives a warning while using an object without initializing its instance.

Component: checkOutCustomer

	Parameter					
Test Case	**Item Price p**	**Quantity x**	**Total Price T = (p * x)**	**Discount% d**	**Expected Result r = T – T*d**	**Status**
1	1500	9	13,500	0%	13,500	Pass
2	1500	25	37,500	2%	36,750	Pass
3	1500	80	120,000	5%	114,000	Pass
4	1500	360	540,000	10%	486,000	Pass
5	1500	900	1,350,000	20%	1,080,000	Pass
6	1500	0	0	Not Allowed	0	Pass
7	1500	-97	Invalid value	Invalid Value	Not Valid - Exception	Pass
8	1500	“hi”	NaN	Invalid Number	Not Valid - Exception	Pass

It should be noted that the price of each item is already added in the system and the Operator cannot change it during the ‘Checkout’ process. Hence for the items that are selected, their prices will be queried from the database and above mentioned discount rules will be applied.

For the sake of building test cases, I have used the items with particular prices and the quantities are used according to the representative values in the Equivalence Classes.

EC-coverage = (number of tested EC / total number of EC) x 100%
EC-coverage = (7 / 8) x 100% = 87.5%

Integrated Module: TransactionsManagement

	Parameter					
Test Case	**Item ID**	**Item Price p**	**Quantity x**	**Total Price T = (p * x)**	**Discount% d**	**Result r = T – T*d**
1	5	1500	9	13,500	0%	13,500
2	-20	INVALID	25	INVALID	INVALID	INVALID
3	0	INVALID	80	INVALID	INVALID	INVALID
4	400	NOT EXIST	360	INVALID	INVALID	INVALID
5	“to”	NaN	900	INVALID	INVALID	INVALID
6	5	1500	25	37,500	2%	36,750
7	5	1500	80	120,000	5%	114,000
8	5	1500	360	540,000	10%	486,000
9	5	1500	900	1,350,000	20%	1,080,000
10	5	1500	0	0	Not Allowed	0
11	5	1500	-97	INVALID	INVALID	INVALID
12	5	1500	“hi”	NaN	INVALID	INVALID

From the above table, Test Cases 1 – 5 and 9 will be run. This is because test cases 6 – 12 were already tested during the Component Testing of checkOutCustomer. Running those tests would be a redundant activity.

EC-coverage = (number of tested EC / total number of EC) x 100%
EC-coverage = (10 / 12) x 100% = 83.3%

Hochschule Fulda
University of Applied Sciences

Static Test

Test Object: Point of Sales System, Saad Abdullah Gondal, 30.03.2017

Goal: To test the completeness of Unit Requirements and Code Standard

Review schedule: 30.03.2017, 15:00 – 14:00, 5411

Moderator: Muneeb Noor

List of reviewers:

Role	Person	Scribe	Time (h) spent for preparation	Remarks
System Architect(Maintainability, Design, code quality etc)	Intesar Haider	Hassaan Ahmed Rana	60 mins	- Java coding standards are properly followed. - Meaningful comments and variable/function names.
Business Analyst – Required Functionality (Verification and validation)	Zain ul Abedeen		35 mins	- Business functionality majorly is working fine. - Components are correctly integrated.
Critical Paths, Code completeness and functioning.	Lloyd Djokoto		1hr 30 mins	- Uncover coding errors and inefficiencies. - Memory leaks

Kick-off Meeting: 21.11.2016, 18:00 – 19:00, G111

Hochschule Fulda
University of Applied Sciences

Before the review starts:

	Yes	No	
Code runs without compiler warnings?		X	
Reviewers are well prepared?	X		
Reference documents available?		X	
Scribe is named?	X		

After the review:

	Yes	No	
Is the list of review findings available?	X		
Time spent for preparation filled in above?	20mins		
Result agreed by reviewers?	X		

Result: Accepted and advised the developer to perform high priority changes.

List of findings:

No.	Location	Raised by	Priority	fixed	remarks
1	ItemsManagement.java	Intesar Haider	Medium	No	Remove unused imports
2	TransactionsManagement.java	Intesar Haider	High	No	Write function definitions for all methods
3	Check it through-out the application within different classes	Lloyd Djokoto	High	No	Close all database connections.