RAMAIAH UNIVERSITY OF APPLIED SCIENCES

ASSIGNMENT

Course Code	CSC204A
Course Name	Advanced Programming Concepts
Programme	B.Tech
Department	CSE
Faculty	FET

Name of the Student	Satyajit Ghana
Reg. No	17ETCS002159
Semester/Year	03/2018
Course Leader/s	V.S. Yerragudi

Declaration Sheet			
Student Name	Satyajit Ghana		
Reg. No	17ETCS002159		
Programme	B.Tech	Semester/Year	03/2018
Course Code	CSC204A		
Course Title	Advanced Programming Concepts		
Course Date		to	
Course Leader	V.S. Yerragudi		

Declaration

The assignment submitted herewith is a result of my own investigations and that I have conformed to the guidelines against plagiarism as laid out in the Student Handbook. All sections of the text and results, which have been obtained from other sources, are fully referenced. I understand that cheating and plagiarism constitute a breach of University regulations and will be dealt with accordingly.

Signature of the Student		Date	
Submission date stamp (by Examination & Assessment Section)			

Signature of the Course Leader and date	Signature of the Reviewer and date

Question No. 1

Solution to Question No. 1 Part A:

A 1.1 Introduction with relevance of the discussion in paper:

The paper *"Encapsulation and Inheritance in Object-Oriented Programming Languages"* by A.Snyder discusses about how the introduction of inheritance severely compromises encapsulation which is one of the most important feature of Object-Oriented Programming. The prime feature of this paradigm is *data abstraction*, which is the ability to define objects with behaviors that are defined abstractly, i.e its main goal is to handle complexity by hiding the unnecessary details of the internal implementation from the user. This enables the user to add more complex login on top of it without worrying about the inner complexity of it. As long as I know which methods do what form of operation with the objects, I'm good to go. So, then what's the problem with inheritance? Well, it pretty much tried to break this abstraction, or these methods that act on the data we call it encapsulation, by providing access of the inherited instance variables that clearly breaks the main aim of OOP. The paper also discusses the challenges involved in solving this issue and proved some viable solutions to it.

This paradigm of language in itself encourages the form of modular design and code reusability through inheritance, though I believe that this is a less understood fact and often misunderstood.

A 1.2 Summary of the paper:

The paper is very well written and raises a valid dispute of Encapsulation and Inheritance.

It talks about how Inheritance is trying to break Encapsulation and therefore trying to break the one of the main concepts of OOP.

Let's look at why this problem is raised in the first place, Inheritance is the mechanism of basing an object or class upon another object (prototypical inheritance) or class (class-based inheritance), retaining similar implementation. In most class-based object-oriented languages, an object created through inheritance acquires all the properties and behaviors of the parent object. Inheritance allows programmers to create classes that are built upon existing classes, to specify a new implementation while maintaining the same behaviors (realizing an interface), to reuse code and to independently extend original software via public classes and interfaces. **(Johnson and Ralph, 1991)**

When inheritance is used carefully (some will say properly), all classes derived from an abstract class will share its interface. This implies that a subclass merely adds or overrides operations and does not hide operations of the parent class. All subclasses can then respond to the requests in the interface of this abstract class, making them all subtypes of the abstract class. **(Design Patterns, 1994)**

Referring to this above lines we can clearly see the where the problem arises, the child or subclass has access to its parents interface, this external interface is just as important as the external interface provided to the

created objects of this class, and severely limits the changes that the designer can make to this class. The designer then can no longer safely modify, remove or reinterpret the instance variables without adversely affecting the descendent classes that depend on that instance variable. Some Languages overcome this issue by using access modifiers that limits the exposure of the interfaces to the classes than inherit the root class, thus then the designer has more control over which instances can be exposed to its subclasses.

Alan Snyder identifies two areas where most object oriented language are deficient in their support for encapsulation, one area that we discussed was the encapsulation of instance variables and the other challenges whether inheritance should be visible in itself, i.e. whether a class should be able to know if it inherits another class itself. One advantage of this would be that suppose the designer has decided that the behavior inherited from the parent is not good enough then he can easily modify the child's behavior in a completely new implementation without making any changes to the clients. This feature of making inheritance "invisible" further goes about excluding operation and the use of subtyping.

There may be times when the child class inherits an interface that is not required from its parent, and thus can be excluded from it, but even if the child excludes the inherited operation, several languages provide the ability to directly invoke a parent interface therefore again breaking encapsulation.

Another issue that arises is during multiple inheritance, about how inheritance must work in such conditions, it tries to solve this issue with Graph-Oriented, where the structure is altered, Linear Solution where the Inheritance is flattened out to a linear chain, but this breaks the ability to communicate with a child's real parents, and the problem of multiple invocations of the same interface, unlike the graph oriented languages, linear solution does not allow a class to designate an operation by the name of the defining parent. These problems are solved in the Tree Solution, it differs from the previous two solutions in (1) an attempt to inherit an operation from more than one parent is always an error, regardless of the source operation (2) each parent of each class defines a completely separate set of inherited variables.

A 1.3 Conclusions drawn from the paper:

Inheritance surely can break the Object Oriented-ness of a Language if not implemented properly, but for any language to be truly OO, it needs to support Inheritance, because it helps in mapping interfaces with the real world itself. At the end this conflict tries to resolve the problems of scalability of software, if there are not strict bounds, the maintenance of the software becomes difficult. If there are strict rules in place from the beginning then it would be easier for the Designers to have fine grain control over the overall development of the software, and would reduce code conflicts between different levels of designing.

Question No. 2

Solution to Question No. 1 Part B:

B 1.1 Introduction to problem:

The problem is a simple, or seems to be at least at first glance, there's a Grocery Shop "ACME Buy N Save" in ACME Land, and as all grocery shop needs is a Billing System for managing the customer purchases and keeping track of the bills, The bill thus generated is to be printed and should contain the list of purchased items with the quantity and subtotal, the bill should also show the total price to be paid by the customer along with the savings made by him, the savings part is necessary as it's a business logic that the customer would be delighted to see the savings made on each purchase, the savings are a result of the discounts given on each product. Since it's a grocery shop it contains both Perishable and Non-Perishable items, and the Shop shouldn't be able to sell an expired item to the customer, hence there should be checking for that, also this would require an inventory manager to manage and remove the expired items and add more items in its place.

B 1.2 UI and Programming Language Selection:

Since we are trying to implement a real world things, into our implementation, we could imagine all of the items as *"Objects"*, with specific behavior and state. Anything in our scenario can be thought of instance of a class, which is like a blue print of an object. Since we would be making a GUI based Program, a good approach for it would be Event-Driven, or Object Action Approach, where the user will be presented with various options and he/she has to choose from them and then specify the action related to it. Buttons will be binded with functions that perform the action.

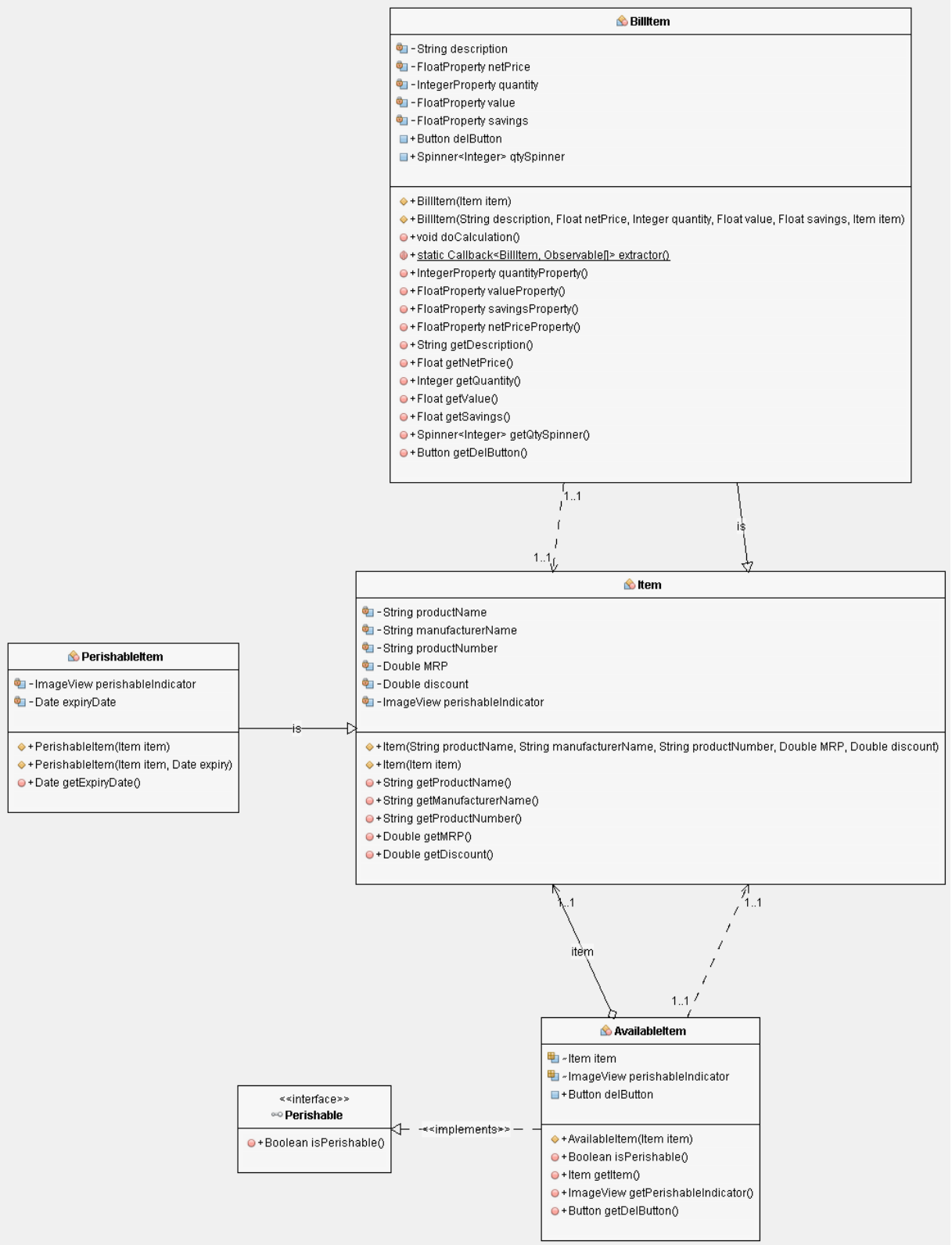
A very good Programming language would be *Java*, since it's highly Object Oriented and also has a huge community support. It has been there since decades and has been refined to very fine levels of almost perfection, everything and anything has a Java Library, which is the biggest feature when it comes to Java, moreover Java is backwards compatible, i.e. even the oldest java code runs on newer JVM.

B 1.3 Object Decomposition:

Let's first imagine the Grocery Shop, There are two types of Items, Perishable and Non-Perishable, Perishable Items can be imagined as Non-Perishable Items which now has a expiry date, so one of our Class would be "*Item"*, and *"Perishable Item"* is a subclass of Item, it inherits the exposed methods of Item. Perishable is an Interface that can be implemented by any Class that has a/ can have a Perishable Item.
Customers can buy items that are Available, i.e. the Inventory List, which I would call *"Available Item"*, these will contain Item, and so this is a composition. Available Item has Item, and can contain 1..1 Items.

To keep all the Available Items, we need a Cart sort of thing, which we will call BillItem, since now the item is in our bill, BillItem Is also an Item, so it is a subclass of Item, and comes with some extra states such as netPrice, quantity, value and savings.

What's a better way to represent this Object Decomposition than a UML Diagram? A picture can say a thousand word can't it?



Explanation:

Item is the main super class which contains the basic state of every item such as its product name, manufacturer, mrp, discount etc. these properties are private and only accessible to this class, and the getter methods are public though.

Perishable Item inherits from Item and obtains all the states that Item has along with its methods. It has one extra state called expiryDate and a method to check if it's perishable.

Available Item doesn't not inherit from anything but is a composition, it contains an Item object and is a 1:1 relationship. Since it can contain a Perishable item, it implements the interface Perishable and has to override isPerishable, which checks if the item that is contained in AvailableItem is perishable or not.

Bill Item inherits Item and has some properties such as doCalculation, and states like subtotal, savings, item count, etc.

B 1.4 Conclusion

The Grocery Shopping Process is easily converted into an Object Oriented Procedure with an Instance of each thing. Object Decomposition is done before actually starting to write any piece of code. This gives a clear picture/idea about the whole project and makes it easier for implementation. *Think twice, Code once*. To represent the real world situation in terms of Objects using UML diagrams creates better relationships between different classes and makes it easier to imagine the whole scenario.

Object Decomposition breaks down a large system down into smaller classes or objects that perform some operations which is a part of the problem domain.

Question No. 3

Solution to Question No. 2 Part B:

B 2.1 Introduction to development:

The development process is a little more work than the design, but since our design is well, it can be well implemented. I chose to do this using Netbeans IDE, just because it provides good code completion and the UML diagram maker easily converts a UML structure into Classes, thus reducing man hours on this. And as they say *"If you can't make it good, at least make it look good"*, so I chose JavaFX with a good UI Library, since this is a desktop application, it needs to look windows-y, JMetro by PedroDV is a good looking UI Library for this purpose.

There are two ways to go from here, either do rapid prototyping, i.e. make a very minimal but working version of the program first and then proceed to adding more features to it, or start with listing out all the features first and then implement all of them. Since we did a proper object decomposition and made the complete UML diagram we'll first implement all of that into our program.

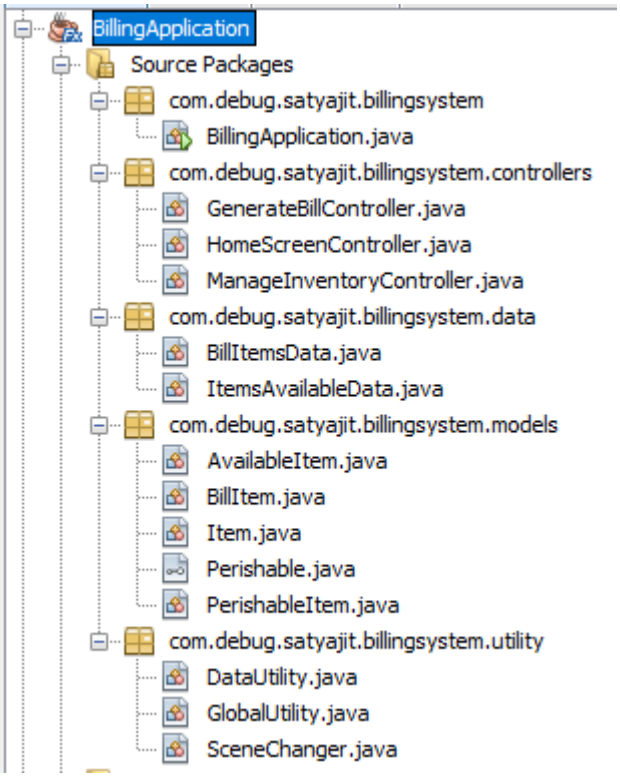
JavaFX in itself provides the necessary UI Elements required for this development, such as Buttons, Tables, Different form of Layouts, User Inputs, and Date Inputs. The data needs to be read and written to a file, and Java provides several methods to do so, the easy method is to go for a Library, since the data is going to be plain text, we would use CSV format to save and retrieve data from the file. An advantage of using CSV is that it can be easily edited using MS Excel. Using these the UI and the Data part is taken care of, although we still need to print the receipt. This can be done by writing out a pdf for each user with the bill details. Once again there are plenty of libraries for the same purpose, of which I chose iText 7, it's free and easy to use.

B 2.2 Implementation of methodology:

The first thing to do is the UI Design, because without this the structure of the code cannot be decided, so the UI was designed in SceneBuilder, a mockup can be made using AdobeXD, but since SceneBuilder itself makes it really easy to just drag and drop an external mockup isn't required.

After the design is done, the respective fx:id are set and the respective methods are bound to the respective elements. To display all the data, we've chosen a table from which the user can click to add an item, and the items then should get added to the bill.

The Program is then structured like this:



The main application name is Billing Application, and the rest are split into packages, the data models or the main data classes are stored in modls, the utility methods such as reading data from the CSV file, writing data are written in DataUtility, Global utility contains instances of DataUtility and SceneChanger, since we want only a single instance to exist that can be accessed throughout the program like a Singleton, but since we want it global singleton is not required.

SceneChanger contains the different scenes of the program, and any scene can be switched to seamlessly just by calling some static method in SceneChanger, this makes it easier to manage the whole program and add more scenes easily.

ItemsAvailableData as the name suggest contains those items when are available, these will be displayed on the right for the user to choose which product to buy from. Mostly the data package is meant to supervise the data from the models, like a container for them. Same is for BillItemsData.

```
public void calculateBill() {
    grandTotal.setValue(0f);
    totalSavings.setValue(0f);
    totalItems.setValue(0);
    billItems.forEach(billItem -> {
        grandTotal.setValue(grandTotal.getValue() + billItem.getNetPrice());
        totalSavings.setValue(totalSavings.getValue() + billItem.getSavings());
        totalItems.setValue(totalItems.getValue() + billItem.getQuantity());
    });
}
```

calculateBill is public method in BillItemsData, which contains all the *billItems*, this is the logic which calculates the bill whenever the GenerateBill button is clicked and sets the bill properties respectively.

```
public void doCalculation() {
    this.netPrice.set(this.quantity.getValue() * ((new Float(this.getMRP())) * new Float(1 - (this.getDiscount() / 100))));
    this.savings.set(new Float(this.quantity.getValue() * this.getMRP()) - this.netPrice.getValue());
}
```

doCalculation is a method in BillItem, this method should be called when every an item is added to the BillItemList, here the netprice and savings are calculated, since BillItem is an Item it can simply call its super class methods to get the data for the item BillItem has inherited.

```
private void updateBill() {
    grandTotal.setText(String.format("%4.2f", billItemsData.getGrandTotal().getValue()));
    totalSavings.setText(String.format("%4.2f", billItemsData.getTotalSavings().getValue()));
    totalItems.setText(String.format("%d",billItemsData.getTotalItems().getValue()));
}
```

updateBill is a private method in GenerateBillController which updates the UI Elements of TotalSavings, TotalItems and GrandTotal, whenever there is any change in the BillItems, such as an item being added to the bill or the number of items in the bill being changed, or an item from the bill being removed.

```
@FXML
private void addList(MouseEvent event) {
    Item _item = itemsAvailableTable.getSelectionModel().getSelectedItem().getItem();
    if (_item != null) {
        /* Check if the item is already in my Billing List */
        Optional<BillItem> result = billItemsData.getBillItems().stream().filter(x -> _item.getProductNumber().equals(x.getProductNumber())).findFirst();
        if (!result.isPresent()) {
            if (_item instanceof PerishableItem) {
                if (((PerishableItem)_item).getExpiryDate().before(new Date())) {
                    Alert alert = new Alert(Alert.AlertType.INFORMATION);
                    alert.setTitle("Expired Item");
                    alert.setContentText(_item.getProductName() + " has expired on " + ((PerishableItem) _item).getExpiryDate());
                    alert.setHeaderText("The Item has expired");
                    alert.showAndWait();
                    return;
                }
            }
            BillItem _billItem = new BillItem(_item);
            _billItem.delButton.setOnAction(_event -> {
                billItemsData.getBillItems().remove(_billItem);
            });
            billItemsData.getBillItems().add(_billItem);
        }
    }
}
```

addList is a private method in GenerateBillController that is binded to the onclick listener on the table, so whenever there is click on data on the table which is an item, that item is to be added to the billItems, getSelectionModel().getSelectedItem().getItem() fetched us the clicked item from the table. The main logic is here, check if the item is already there in the billItem or not, if it's not present then add it, here the perishable items are checked if they are expired or not, so the current system time is compared with the expiry date of the item, to check if the item clicked on is a perishable item or not we use the keyword ***instanceof*** which tells us exactly that, If it has expired then it's not allowed to be added to the billList and an alert box is displayed which contains information about the product and also the date on which it has expired.

```java
public ObservableList<Item> readItems() throws FileNotFoundException, IOException, ParseException {
    ObservableList<Item> items = FXCollections.observableArrayList();
    CSVReader reader = getCSVReader();
    String[] nextRecord;
    /* Read the Header */
    reader.readNext();
    while ((nextRecord = reader.readNext()) != null) {
        Item item = new Item(nextRecord[1], nextRecord[2], nextRecord[0], Double.parseDouble(nextRecord[3]), Double.parseDouble(nextRecord[4]));
        if (nextRecord.length > 5 && !nextRecord[5].equals("")) {
            PerishableItem perishableItem = new PerishableItem(item, dateFormat.parse(nextRecord[5]));
            items.add(perishableItem);
        } else {
            items.add(item);
        }
    }

    return items;
}
```

readItems is a public method that reads the data from a CSV file and return the data as an observable list. The logic here is that CSV files use the delimiter "," for separating data fields and "\n" for separating two objects, so we read line-by-line and split the line using that delimiter, the data is then read and a new Object of type is created, the type here depends upon if the item has a expiry date or not, if it has a expiry date then a new PerishableItem object is created, else an Item is created, these are then added to the observable list and the whole list is returned to the function callee.

```java
@FXML
private void search(KeyEvent event) {
    search.textProperty().addListener((observable, oldValue, newValue) -> {
        itemsAvailable.getFilter().setPredicate((Predicate<? super AvailableItem>) (AvailableItem item) -> {
            if (newValue.isEmpty() || newValue == null) {
                return true;
            } else if (item.getItem().getProductName().toLowerCase().contains(newValue.toLowerCase())
                    || item.getItem().getProductNumber().contains(newValue)) {
                return true;
            }
            return false;
        });
    });

    itemsAvailable.setSortedList(new SortedList(itemsAvailable.getFilter()));
    itemsAvailable.getSortedList().comparatorProperty().bind(itemsAvailableTable.comparatorProperty());
    itemsAvailableTable.setItems(itemsAvailable.getSortedList());
}
```

search is a private method that is bound to onKeyPressed listener in the textField Search, so any key typing on the search text box invokes this event, the textbox has a observable listener that keeps track of old and new values store in the textbox, we get the filter state of the itemsAvailable and set the predicate by using this newValue of the observer, if it's not empty then the new list needs to be made, the list is made by setting the filer to the sorted list state of the itemsAvailable, and then the sorted list is updated on the table.

B 2.3 Discussion of results:

The Program is pretty straight forward to use, the user is greeted with a welcome screen on running the application. The User is presented with two button Manage Inventory and GenerateBill, Manage Inventory is generally for the Manger to add and remove items from the inventory, for example if there is a Perishable Item that has expired.

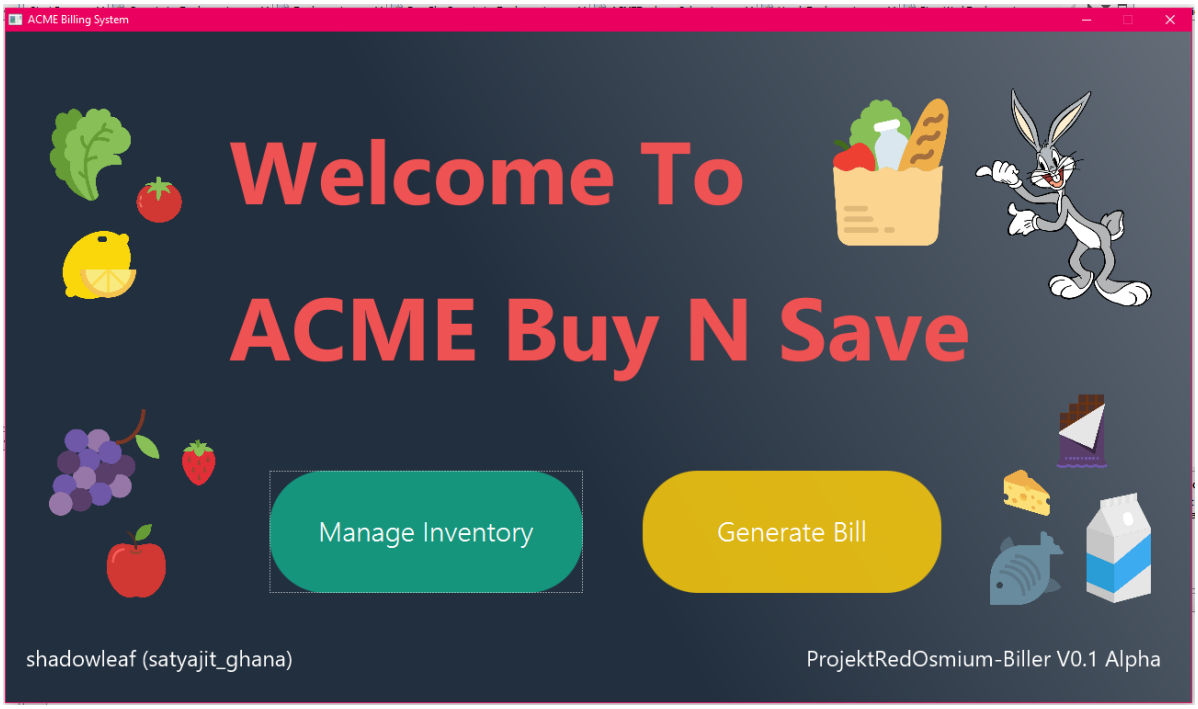


Figure B2.1 Output – Splash Screen

On Clicking the Generate Bill button, the respective scene is loaded into the screen, the user is presented with a list of items on the right, with the details of the products, on the top there is a placeholder for the current customer name, and followed by a search box for filtering items from the list. The Bill List is in the middle, to add items to the list the respective items are clicked from the available items and they are then added. The items quantity can be changed by clicking the spinner for the respective item from the list. The net value and savings are instantaneously updated as soon as there is any change in the quantity. On the right there is a display for the GrandTotal that the user has to pay along with the total savings he/she has made during this visit. The total Items are displayed along as well.

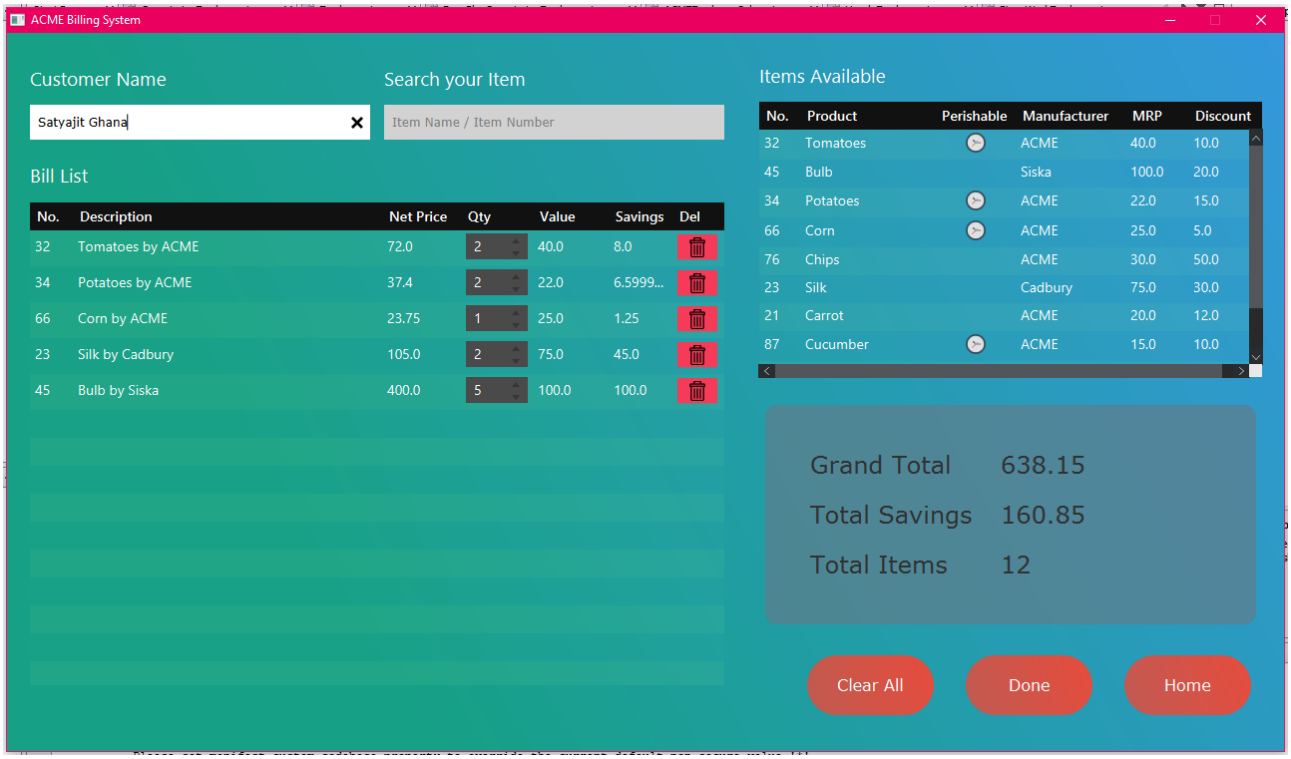


Figure B2.2 Output – Bill Generate Scene

After all this is done, when clicked on Done, the bill is generated in PDF format and saved along with the Customer Name and Date-Time of purchase, a printout of the same can be taken out easily.

ACME Buy N Save

ACME Land

Customer Name : Satyajit Ghana

Description	Net Price	Qty	Value	Savings
Tomatoes by ACME	72.00	2	40.00	8.00
Potatoes by ACME	37.40	2	22.00	6.60
Corn by ACME	23.75	1	25.00	1.25
Silk by Cadbury	105.00	2	75.00	45.00
Bulb by Siska	400.00	5	100.00	100.00

Balance Due: 638.15
Total Number of Items Sold: 12

You Saved : 160.85 !

Date Of Purchase : 2018-Sep-24 05:08:05

Thank You ! Please Visit Again

Figure B2.3 Output – Print Bill

The inventory Managers lets you add, delete, modify, and backup the inventory. The User/Manager in this case is presented with the current data from the data file, and presented with an option to add a new item to the current list, all of them being text fields, and a date pickers, each of which has a custom form validation and no chance of entering invalid input, this is really crucial since if wrong data is given it may mess up the actual data. Also to prevent accidental data loss backup button is provided that takes a snapshot of the data with current date and time, so whenever required the old data can be accessed. After the respective changes are made clicking on Save, writes all the objects to the data file.

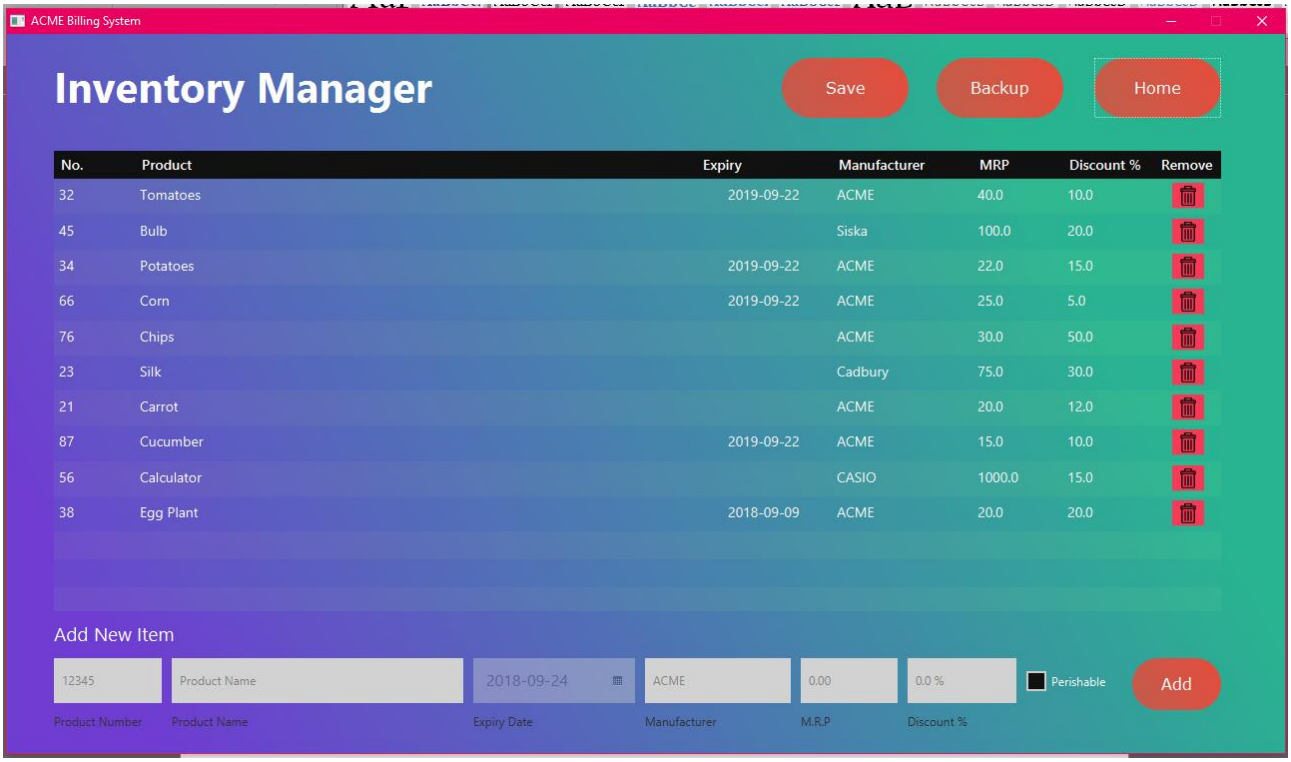


Figure B2.4 Output – Inventory Manager Scene

B 2.4 Conclusion:

Implementation is easy when a blueprint of the program is already made which was our object decomposition and the UML diagram, itty-bitty things can be looked up at the JavaDocs which provides a comprehensive detail to every method that ever existed in Java or JavaFX. Although the design implementation here is primitive, it's scalable but not to a large extend, there is not must security put into and cannot be used for commercial purposes, albeit does a pretty good job at what the question of this assignment states. The Development process is straight forward and exactly what the program specifications are.

Object Oriented Programming is good at representing real word objects in programming and simulating them with state and behavior makes it even better at it. Access specifiers must be used wisely as they are a very crucial part of the whole program.

There's another thing that I learnt, never use Inheritance *just* for code reuse, *never*!, granted that it does benefit from code reuse, but it doesn't mean that inheritance is the only option, composition could be another solution to it, just using inheritance is like calling for a banana but you receive the banana with a gorilla with a tiger and the whole jungle with it. This makes it less scalable and the reason Composition had to be used here. Inheritance doesn't make sense if used exclusively, real world objects have more kinds of relationships like "has-a" and aggregations should be used accordingly.

1. Johnson and Ralph (1991) *Designing Reusable Classes*, www.cse.msu.edu.
2. Design Patterns (1994) *Elements of Reusable Object Oriented Software*, Addison-Wesley.

For Complete Code refer to:
Satyajit Ghana (2018), https://github.com/satyajitghana/University-Work