ASSIGNMENT #5

RITIKA KUMARI

A20414073

Assignment #1

Project :

Build an application for the GameSpeed retailer that will allow its customers to buy/trade-in products from the retailer either in-store or online.

Deliverables:

GameSpeed Retailer

1. **List of actors**
   1. Customer
   2. Salesman
   3. StoreManager

Sl. No.	Name(Actors)	Role
1.	Customer	• The customer can pre-order products • The customer can trade-in products • The customer can place an order online, check the status of the order, or cancel the order. • The customer can purchase products from store. • The customer can pay for products in cash, check, or credit card. • The customer can enroll for PowerMember with annual membership fee of $100 and receive 5% discount on every item purchased. • The customer can rent game console with various plans.
2.	Salesman	• Salesman can create Customer accounts • Salesman can Create/Cancel/Update customers order/transaction.
3.	StoreManager	• StoreManager can Add/Delete/Update products.

2. <u>Lists of Use-Cases:</u>

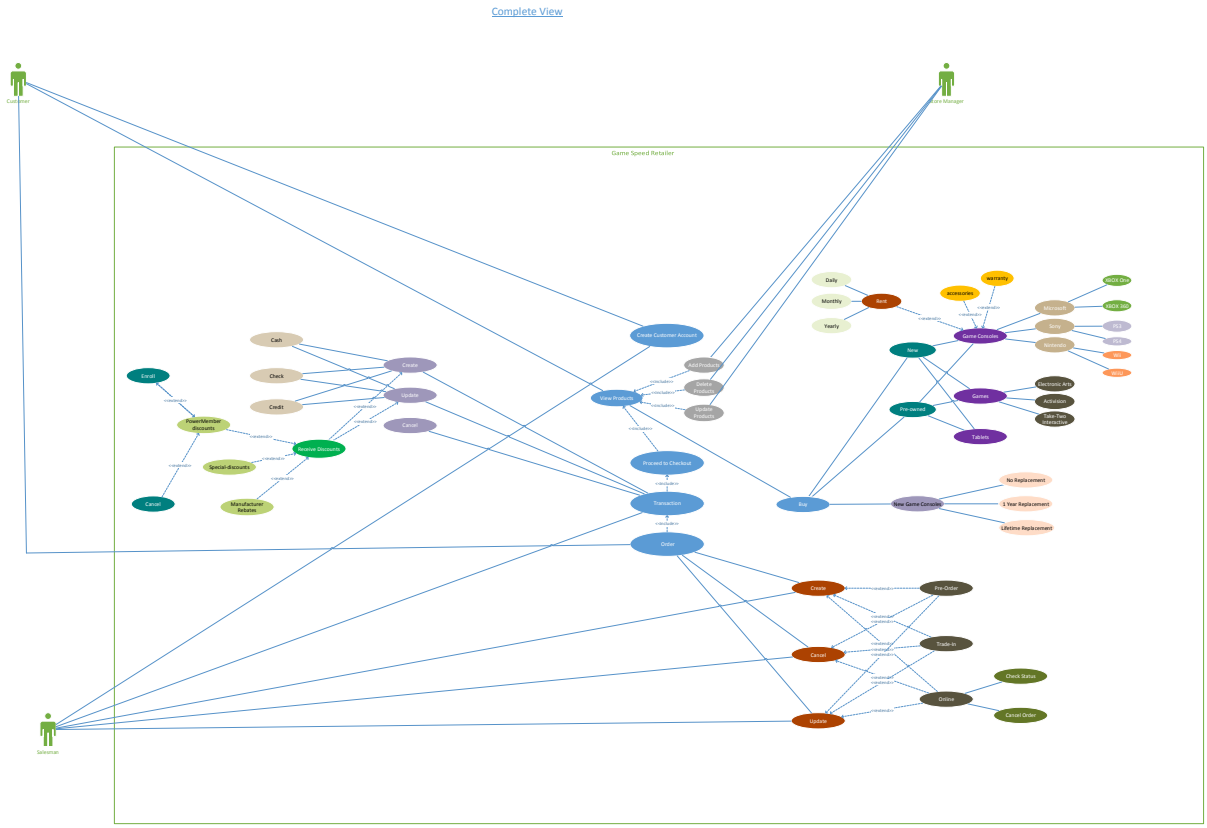
Sl. No.	List of Use cases	Description
1.	Create Customer account	-Customer can create or Update his/her account based on existing or new user. -Salesman will be responsible to verify the customer authentication.
2.	View Products a. Add Products b. Delete Products c. Update Products	- Store manager will be responsible to add/delete/update products to the website/warehouse based on customer requirements. -Store manager can update products and services which are already offered to customers. (either by offering discounts or by increasing the price)
3.	Proceed with checkout	Customer will proceed to checkout once he/she found the required products in the warehouse.
4.	Transaction a. action: Create/Cancel/Update b. types: Cash, Check, Credit c. Discounts for create/update 1. PowerMember(Enroll/Cancel) 2. Special-discounts 3. Manufacturer Rebates	-Salesman is responsible to Create/Cancel/Update customers transaction. -Cash, Check and credits are types available to make the transaction for customers. - Under Create/Update transaction, three types of discounts are provided by Salesman to Customer.
5.	View New Products a. Game Consoles 1. Buy a. No Replacement b. 1 Year Replacement c. Lifetime Replacement 2. Rent a. Daily b. Monthly c. Yearly 3. Accessories 4. Warranty 5. Types: a. Microsoft • XBOXOne	-Customer can view the new products which is Game Consoles, Games and Tablets. -Customer can either buy or rent the new Game consoles. -Customer can buy the accessories or warranty if he/she wants. -Buying the new product will give the option to get the types of replacement by the salesman.

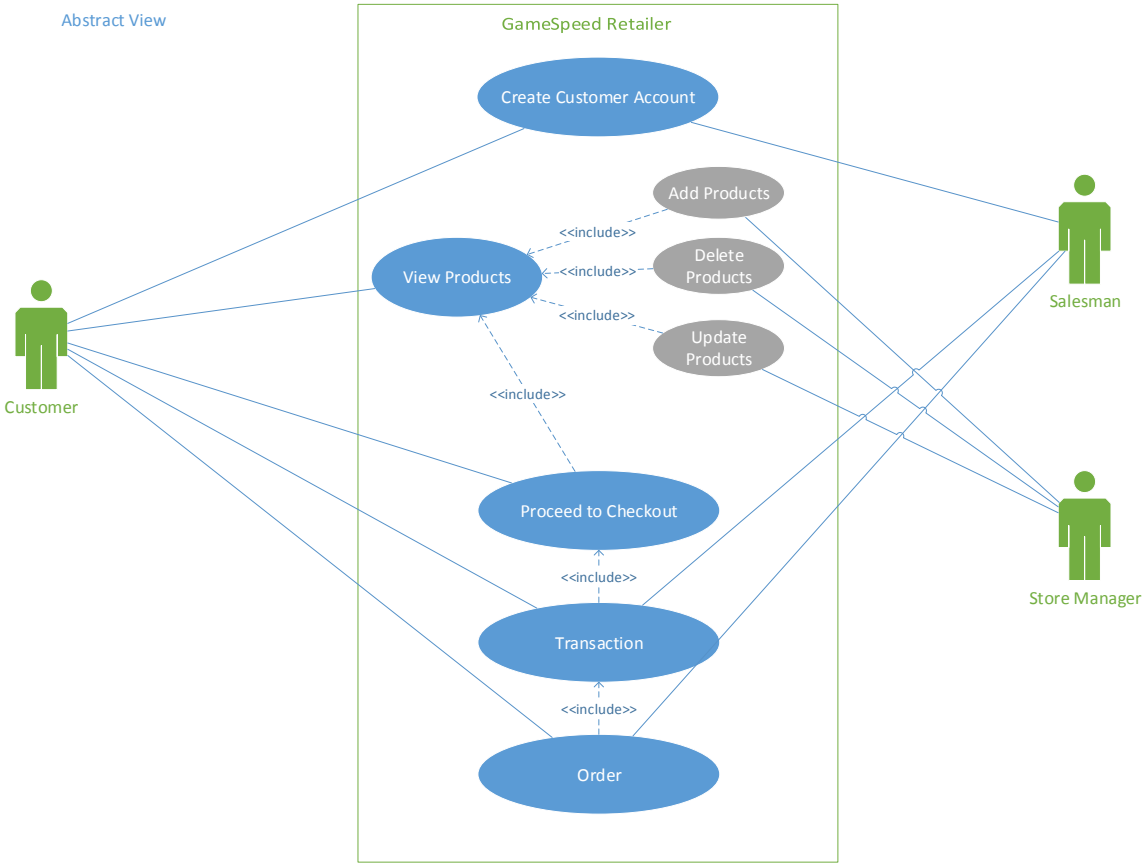
	• XBOX 360 b. Sony • PS3 • PS4 c. Nintendo • Wii • WiiU b. Games 1. Electronic Arts 2. Activision 3. Take-Two Interactive c. Tablets	

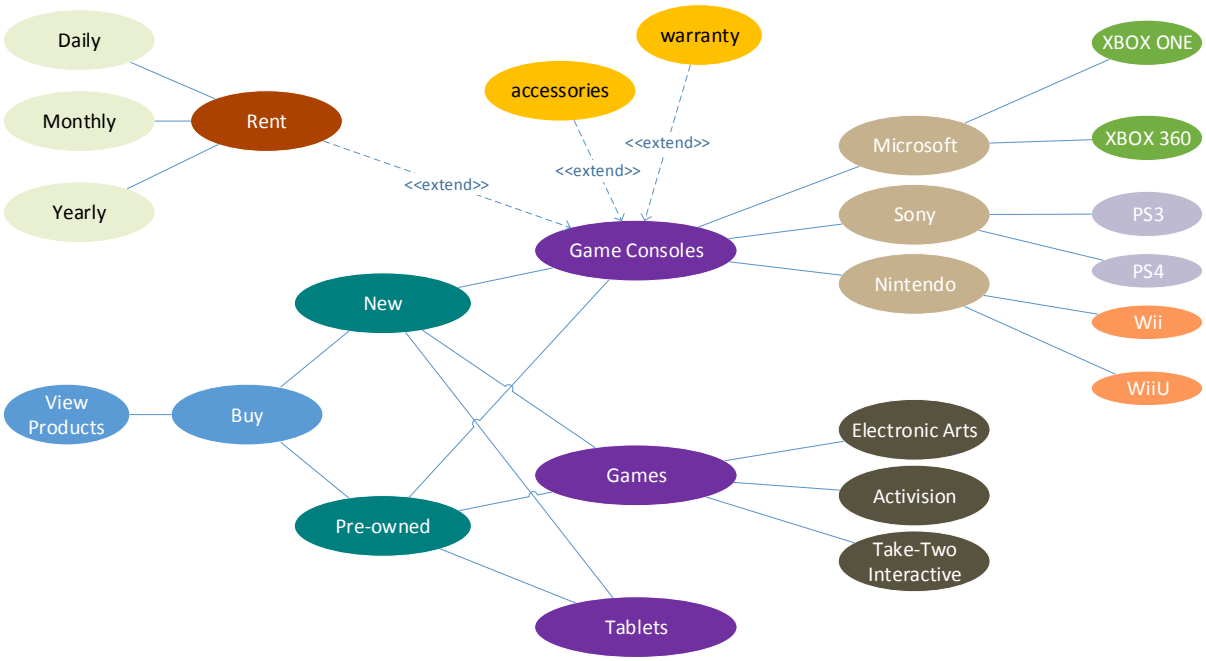
6.	View Pre-Owned Products a. Game Consoles 1. Buy 2. Rent a Daily b Monthly c Yearly 3. Accessories 4. Warranty 5. Types: a. Microsoft • XBOXOne • XBOX 360 b. Sony • PS3 • PS4 c. Nintendo • Wii • WiiU b. Games 1. Electronic Arts 2. Activision 3. Take-Two Interactive c. Tablets	-Customer can view the pre-owned products which is Game Consoles, Games and Tablets. -Customer can either buy or rent the pre-owned Game consoles. -Customer can buy the accessories or warranty if he/she wants.

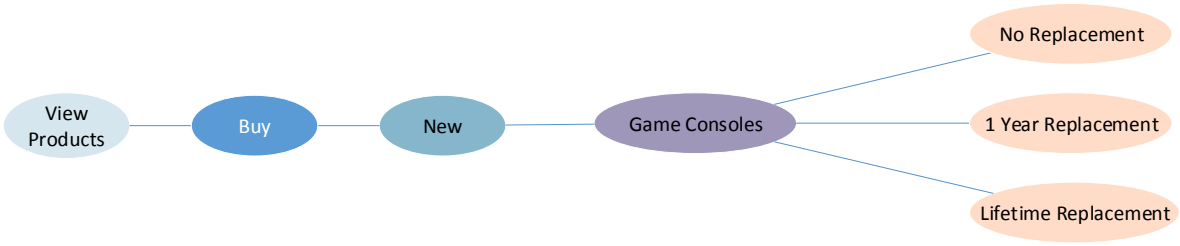
7.	Order 1. Create a. Pre-Order b. Trade-In c. Online 2. Cancel a. Pre-Order b. Trade-In c. Online 3. Update a. Pre-Order b. Trade-In c. Online 1. Check Status 2. Cancel Order	-Salesman is responsible to Create/Cancel/Update customers order. - There are three types (Pre-Order, Trade-In, Online) of order facilitates by the salesman under Create/Cancel/Update to customer.

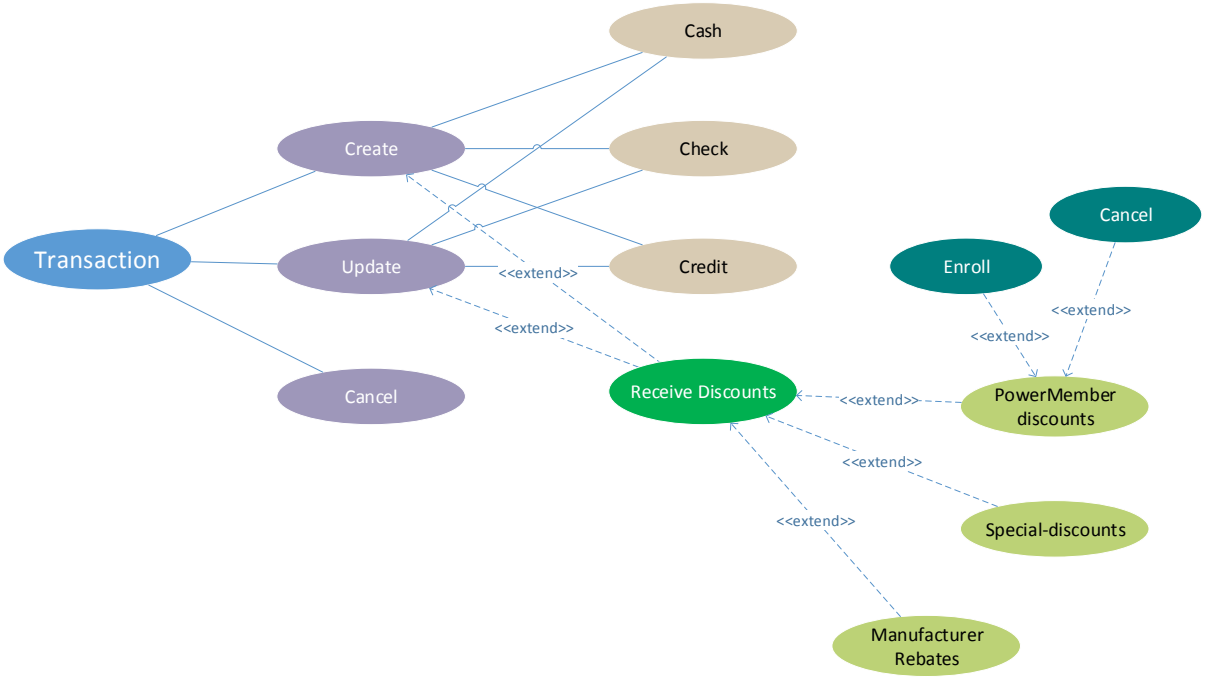
3. UML use case Diagram

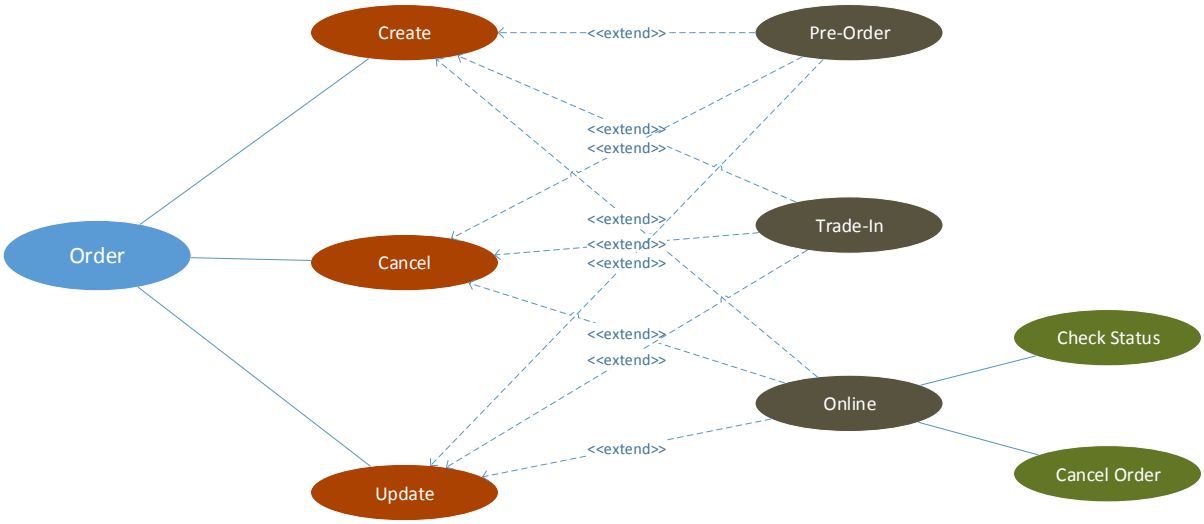



Abstract View
GameSpeed Retailer
Create Customer Account
Add Products
Delete Products
Update Products
View Products
<<include>>
<<include>>
<<include>>
<<include>>
Proceed to Checkout
<<include>>
Transaction
<<include>>
Order
Customer
Salesman
Store Manager

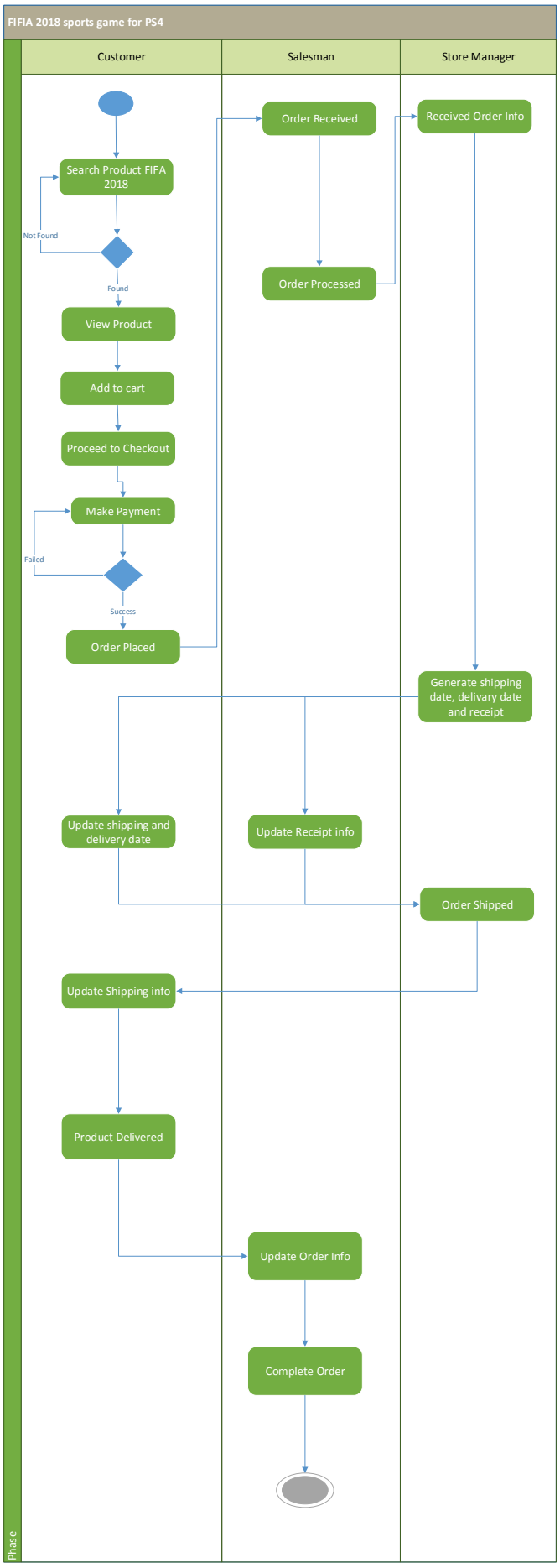

Daily
Monthly
Yearly
Rent
warranty
accessories
<<extend>>
<<extend>>
<<extend>>
XBOX ONE
XBOX 360
Microsoft
Sony
PS3
PS4
Nintendo
Wii
WiiU
Game Consoles
New
View Products
Buy
Pre-owned
Games
Electronic Arts
Activision
Take-Two Interactive
Tablets


View Products
Buy
New
Game Consoles
No Replacement
1 Year Replacement
Lifetime Replacement


Cash
Create
Check
Transaction
Update
Credit
<<extend>>
<<extend>>
Cancel
Cancel
Enroll
<<extend>>
<<extend>>
Receive Discounts
<<extend>>
PowerMember discounts
<<extend>>
Special-discounts
<<extend>>
Manufacturer Rebates



4.Activity diagram for placing an online order to preorder FIFA 2018 sports game for PS4.


FIFIA 2018 sports game for PS4
Customer
Salesman
Store Manager
Search Product FIFA 2018
Not Found
Found
View Product
Add to cart
Proceed to Checkout
Make Payment
Failed
Success
Order Placed
Order Received
Order Processed
Received Order Info
Generate shipping date, delivary date and receipt
Update shipping and delivery date
Update Receipt info
Order Shipped
Update Shipping info
Product Delivered
Update Order Info
Complete Order
Phase

5. Fully dressed use cases:

1. Use Case Section	Comment
Name	View Products
Scope	Game speed Retailer
Level	User goal
Primary Actor	Customer
Stakeholders and Interests	Customer: Wants purchase and fast service with minimum effort. Wants easy search for entered products and prices. Wants to see the varieties of items for games consoles, games and tablets. Salesman: Wants to create Customer accounts successfully. Wants to create/Cancel/Update customers order. Wants to create/Cancel/Update customer transactions without any issue. Storage Manager: Wants to Add or Delete or Update products successfully.
Preconditions	Product should be available in the warehouse. Product price should be listed properly.
Postconditions	Able to add the product in the cart. Able to make payment successfully. Product is delivered successfully.
Main Success Scenario	-Customer visits the website. -Customer searches for the products he/she wants to buy and then view the product for more details. -Customer adds the product to the cart and enter the shipping address. -Customer makes the payment with all available options. - Order is placed successfully.
Special Requirement	-Pre-Ordering of products is available. -Notifications and recommendations for the product is available. -1-click checkout option is available.
Variations in Technology and Data	-Cash payment for Customer is available. -Able to fetch the entered credit card information. -Searching product through voice is available.
Frequency of Occurrence	Could be nearly continuous.
Miscellaneous	-What are the tax low variations? -What should be done if the product found defective?

2. Use Case Section	**Comment**
Name	Transaction
Scope	Game speed Retailer
Level	User goal
Primary Actor	Salesman
Stakeholders and Interests	Customer: Wants easy mode of transaction. Wants different mode of transaction available. Salesman: Wants to create/Cancel/Update customer transactions without any issue.
Preconditions	Debit/credit Card should be working properly. System should be working fine.
Postconditions	Able to make payment successfully. Able to get the receipt of the payment.
Main Success Scenario	-Customer visits the website. -Customer searches for the products he/she wants to buy and then view the product for more details. -Customer adds the product to the cart and enter the shipping address. -Customer makes the payment with all available options. - Order is placed successfully.
Special Requirement	-Cash on delivery of products is available.
Frequency of Occurrence	Could be nearly continuous.

3. Use Case Section	**Comment**
Name	Checkout
Scope	Game speed Retailer
Level	User goal
Primary Actor	Customer
Stakeholders and Interests	Customer: Wants easy checkout with minimum effort. Wants to add or delete products easily for checkout.
Preconditions	Items should be added in the cart properly. System should be working fine.
Postconditions	Able to make payment successfully. Product is delivered successfully.
Main Success Scenario	-Customer visits the website. -Customer searches for the products he/she wants to buy and then view the product for more details. -Customer adds the product to the cart and enter the shipping address. -Customer makes the payment with all available options. - Order is placed successfully.
Special Requirement	-Pre-Ordering of products is available. -1-click checkout option is available.
Frequency of Occurrence	Could be nearly continuous.

Assignment #2

Project Overview Statement:

Develop an application for the GameSpeed retailer that will allow its customers to buy/trade-in products from the retailer either in-store or online.

Deliverables:

1. Any Three System Sequence Diagrams for the use-case diagram

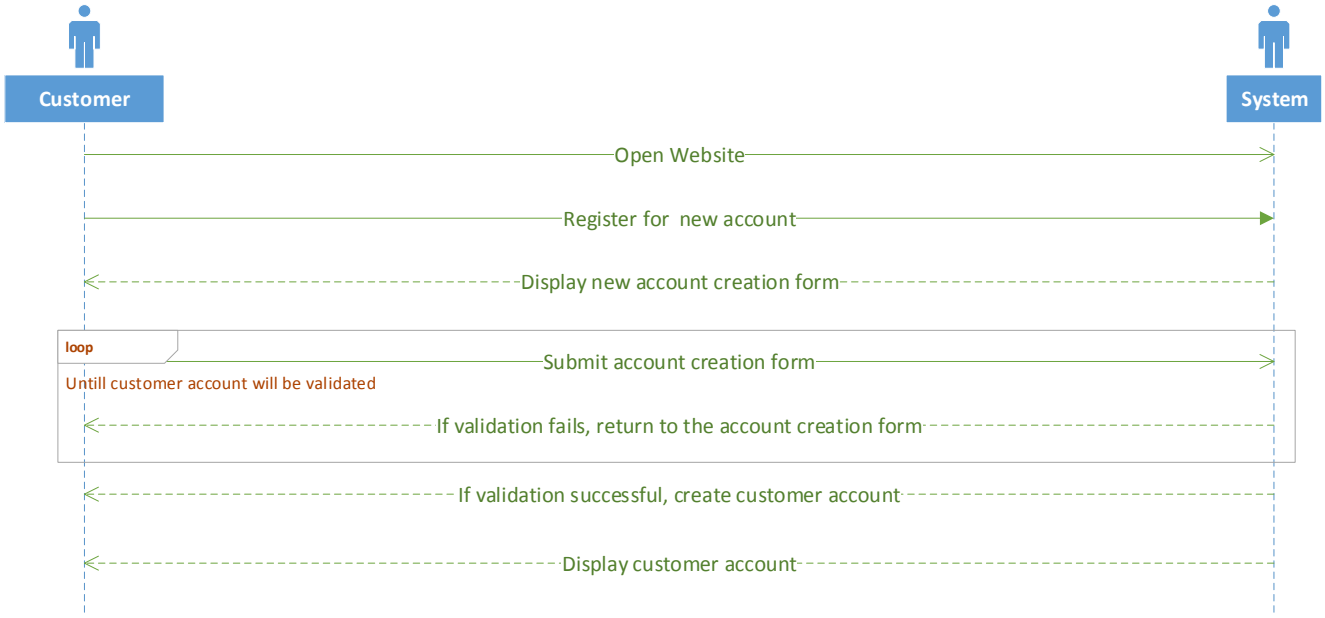
A system sequence diagram (SSD) illustrates events sequentially input from an external source to the system

- A sequence diagram is a good way to visualize and validate various runtime scenarios.
- An SSD shows – for one scenario of a use case –
    - the events that external actors generate,
    - their order, and
    - inter-system events
- The system is treated as a black-box
- System sequence diagrams are a timeline drawing of an expanded use case.

A) System Sequence Diagram for Use Case: "CreateCustomerAccount"

- Customer will go to the website and request for register a new account
- System displays a registration form to the customer
- Customer fills the registration form and submit the form
- System will validate the form using loop (loop: if validation fails)
- Customer account gets created after validation and acknowledgement is sent by system.

System Sequence Diagram for Create Customer Account



B) System Sequence Diagram for Use Case: "Order"

- Customer opens the website and login to his account.
- System validates the credential details. If it is successful, it will allow customer to login. If the attempt is unsuccessful, it will go back to the login page.
- Customer requests for search items.
- System returns the requested items with descriptions.
- Customer add items to the shopping cart and requests for the cart info.
- System displays the cart info to the Customer.
- Customer enters the payment details. System validates the payment details. If it is successful, order will be placed, and system will send a notification to customer. If it is unsuccessful, System asks the Customer to re- enter the payment details.
- Finally, Customer logs out from his account.

System Sequence Diagram for Order

C) System Sequence Diagram for Use Case: "ViewProducts"

- Store Manager logs in to his account.
- System validates the credential details. If it is successful, it will allow Store Manager to login. If the attempt is unsuccessful, it will go back to the login page.
- Store manager requests to view the products' page and system displays the requested products' page.
- Store manager will add new products to the products page and the page is updated by the system.
- Store manager will update discounts for the products to the products page and the page is updated by the system.
- Store manager deletes products from the page and it is updated by the system.
- Store manager updates the product info and it is updated by the system.
- Store manager logs out from his account after doing all the operations.

2. The Class diagram for your analysis model

The class diagram is the main building block of object-oriented modelling. It is used for general conceptual modelling of the systematic of the application, and for detailed modelling translating the models into programming code.

In the diagram, classes are represented with boxes that contain three compartments:

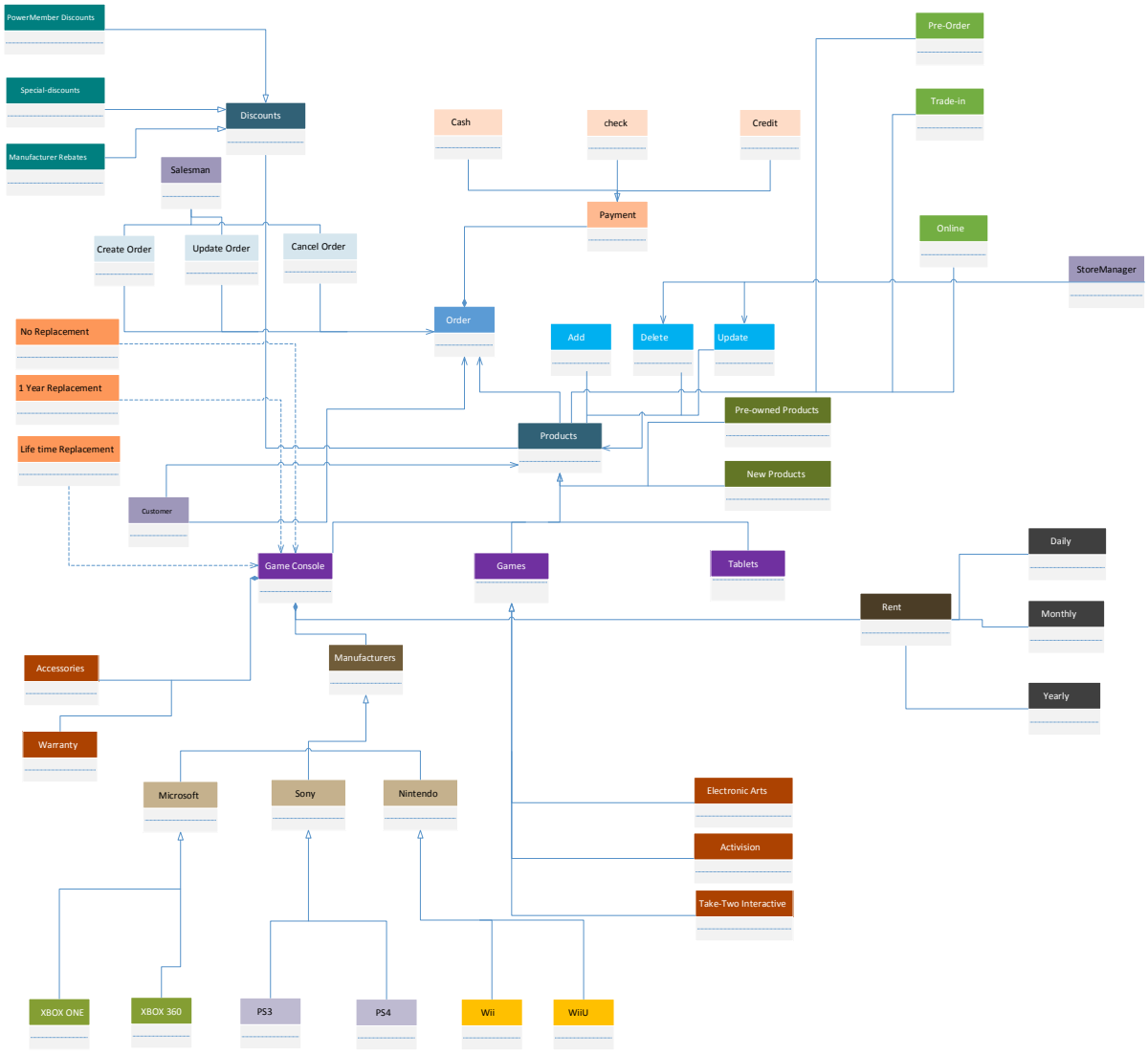
- The top compartment contains the name of the class. It is printed in bold and cantered, and the first letter is capitalized.
- The middle compartment contains the attributes of the class. They are left-aligned, and the first letter is lowercase. The bottom compartment contains the operations the class can execute. They are also left aligned, and the first letter is lowercase.

Sl No.	Class name
1	Customer
2	Products a) Pre-Owned Products b) New Products
3	Order
4	Payment a) Cash b) Check c) Credit
5	Discounts a) PowerMember discounts b) Special-discounts c) Manufacturer rebates
6	Salesman a) Create Order b) Update Order c) Delete Order
7	StoreManager d) Add Products e) Update Products f) Delete Products
8	Games a) Electronic Arts b) Activision c) Take-two Interactive
9	Game consoles
10	Tablets
11	Accessories

12	Warranty
13	Manufacturers a) Microsoft 1. XBOXONE 2. XBOX360 b) Sony 1. PS3 2. PS4 c) Nintendo 1. Wii 2. WiiU
14	Rent a) Daily b) Monthly c) Yearly
15	PreOrder
16	Trade in
17	Online

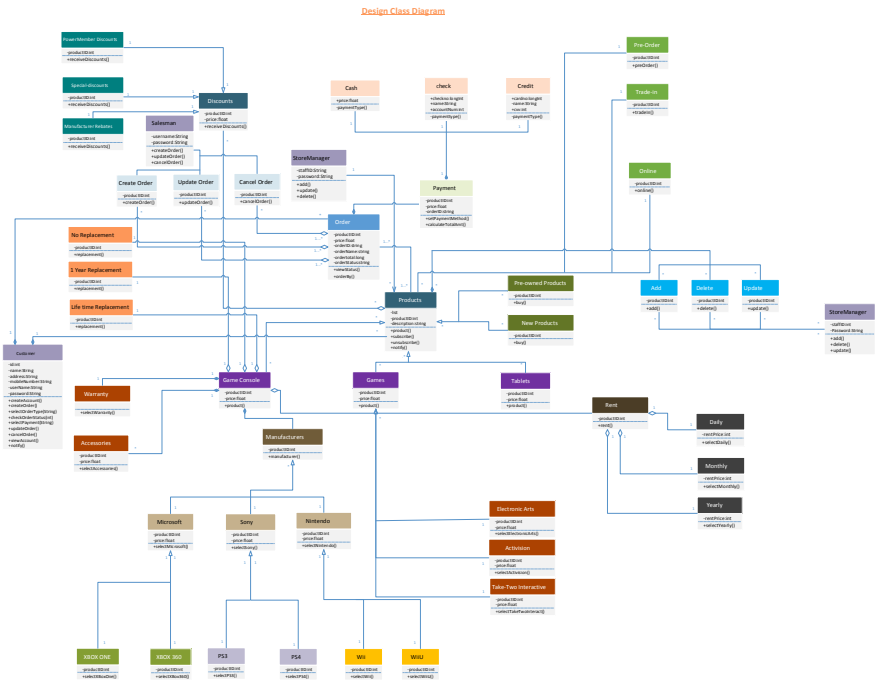
CLASS DIAGRAM-ANALYSIS MODEL

Analysis Class Diagram



3. The Class diagram and package diagram for your design model

A. CLASS DIAGRAM-DESIGN MODEL



B. PACKAGE DIAGRAM-DESIGN MODEL

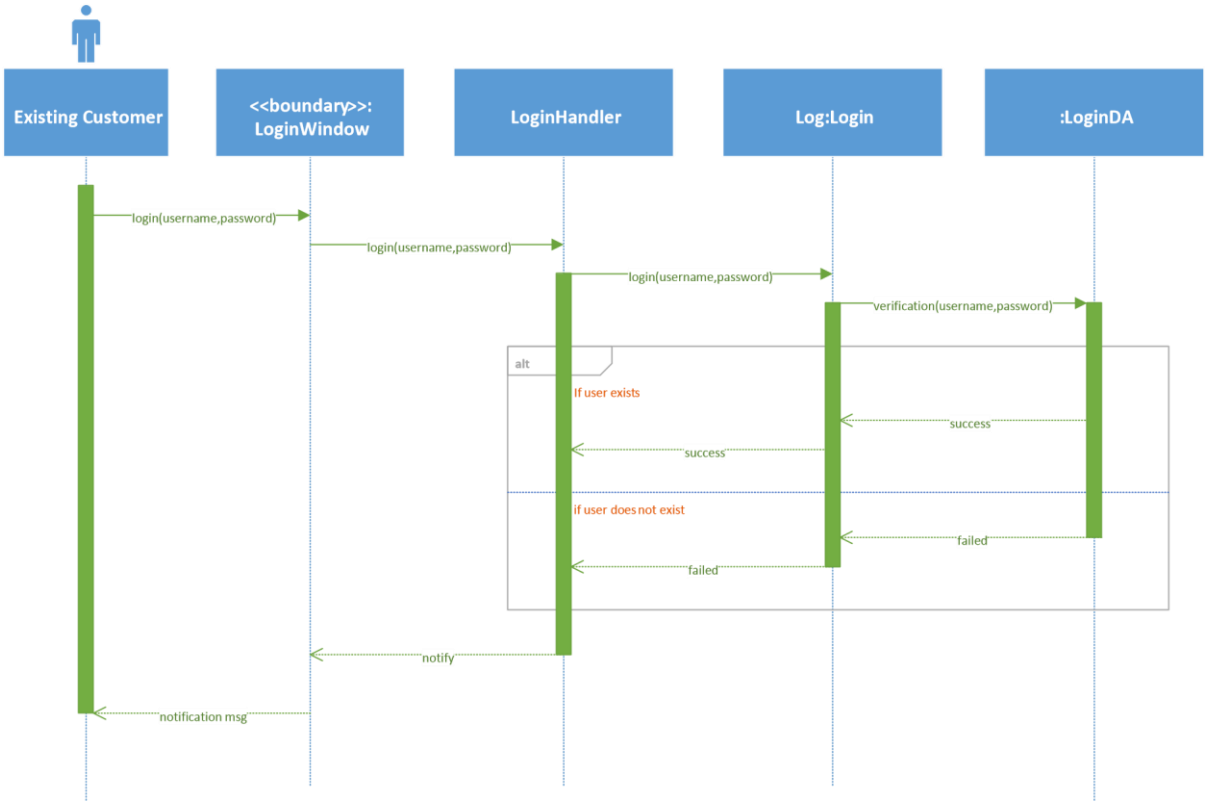
Package diagram is UML structure diagram which shows packages and dependencies between the packages.

4. **Any Five sequence interaction diagrams that you can pick form the list of potential sequence interaction diagrams for your design model.**

Sequence diagrams describe interactions among classes in terms of an exchange of messages over time. They're also called event diagrams. A sequence diagram is a good way to visualize and validate various runtime scenarios. These can help to predict how a system will behave and to discover responsibilities a class may need to have in the process of modelling a new system.

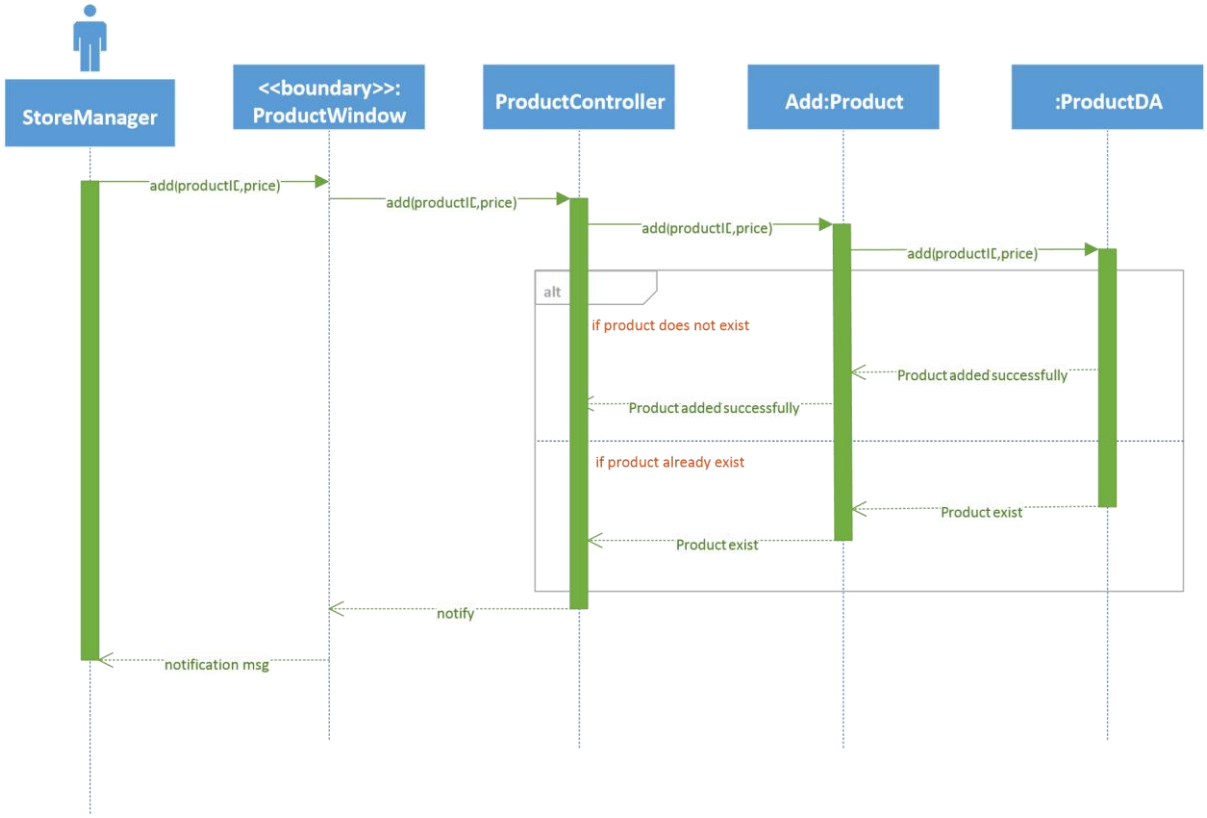
1. Login

Login



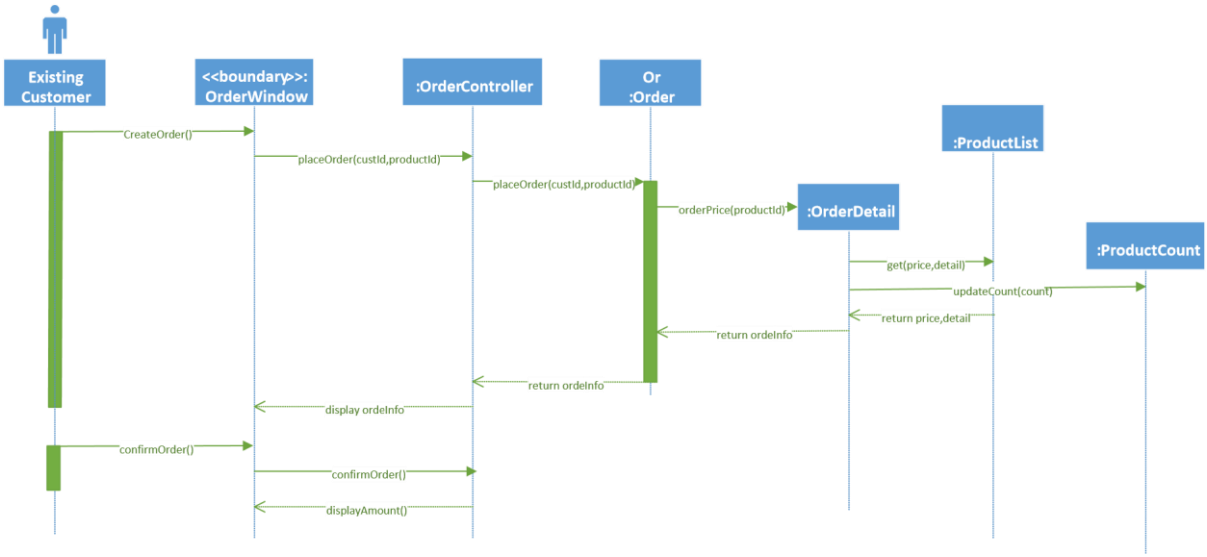
2. Add Product

Add Product



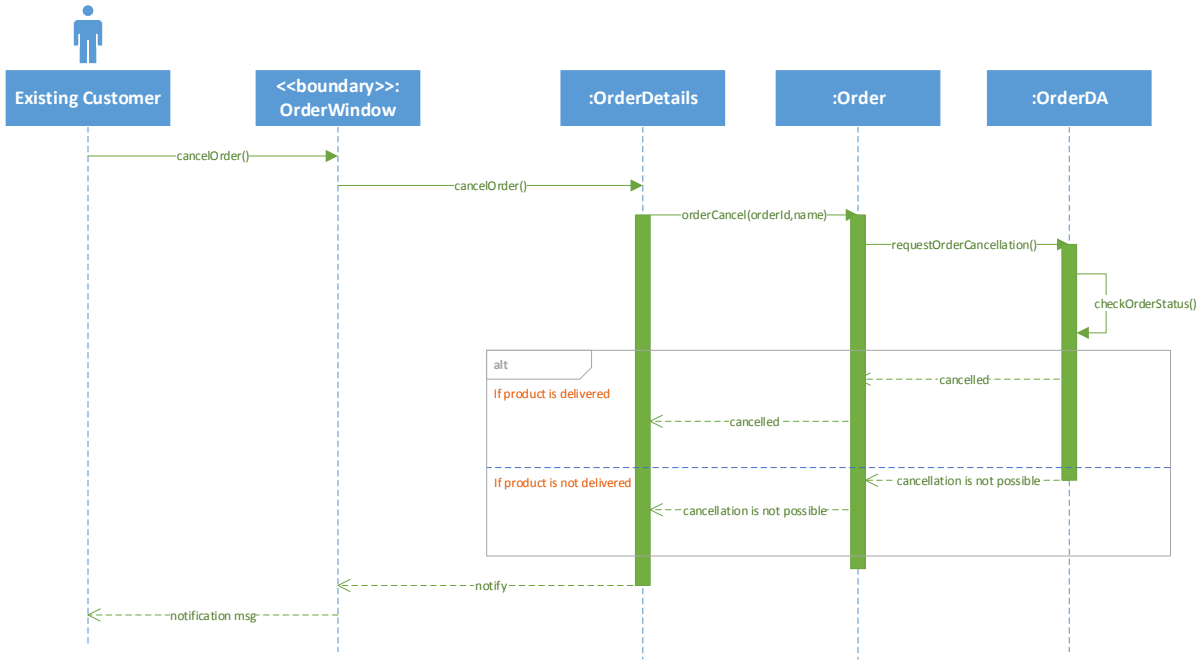
3. Place Order

Place Order



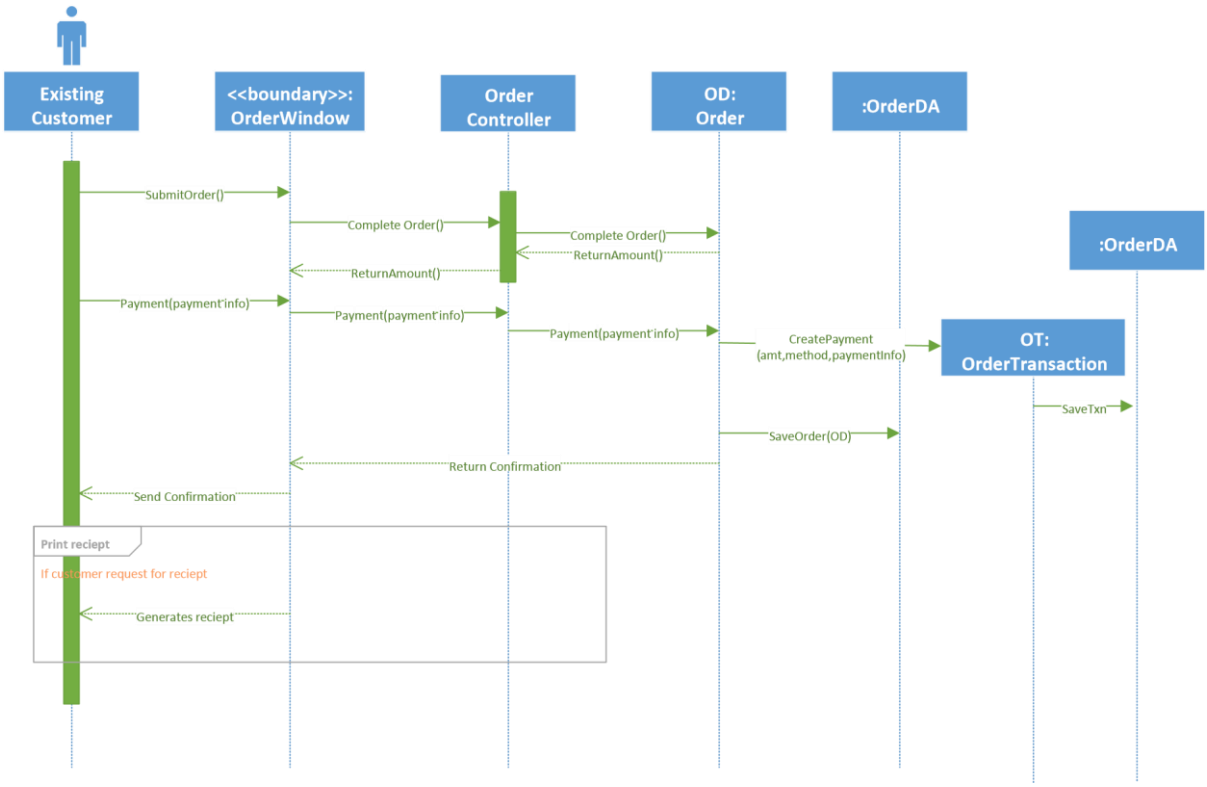
4.Cancel Order

Cancel Order



5. Payment

Payment



Assignment #3

Project Overview Statement:

Build an application for the GameSpeed retailer that will allow its customers to buy/trade-in products from the retailer either in-store or online.

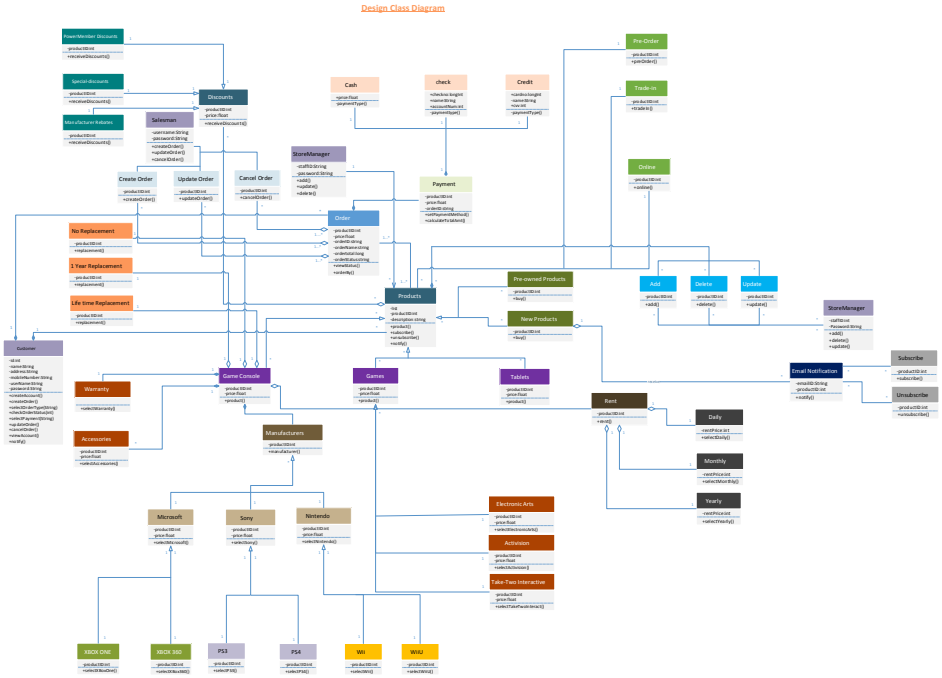
Deliverables:

1. **Complete list of classes used in your design**

Sl No.	Class name
1	Customer
2	Products a) Pre-Owned Products b) New Products
3	Order
4	Payment a) Cash b) Check c) Credit
5	Discounts a) Power Member discounts b) Special-discounts c) Manufacturer rebates
6	Salesman a) Create Order b) Update Order c) Delete Order
7	Store Manager d) Add Products e) Update Products f) Delete Products
8	Games a) Electronic Arts b) Activision c) Take-two Interactive
9	Game consoles
10	Tablets
11	Accessories
12	Warranty

13	Manufacturers a) Microsoft 1. XBOXONE 2. XBOX360 b) Sony 1. PS3 2. PS4
	c) Nintendo 1. Wii 2. WiiU
14	Rent a) Daily b) Monthly c) Yearly
15	PreOrder
16	Trade in
17	Online
18.	Email Notification a) Subscribe b) Unsubscribe

2. Complete UML Design Model/class diagram



3. List of the Design pattern(s) that you have used

The following are the design patterns which I have used.

a. Factory Method Design Pattern: Factory method pattern is a creational pattern that uses factory methods to deal with the problem of creating objects without having to specify the exact class of the object that will be created. This is done by creating objects by calling a factory method—either specified in an interface and implemented by child classes or implemented in a base class and optionally overridden by derived classes—rather than by calling a constructor.

The Factory Method design pattern solves problems like:

- How can an object be created so that subclasses can redefine which class to instantiate?
- How can a class defer instantiation to subclasses?

Creating an object directly within the class that requires (uses) the object is inflexible because it commits the class to an object and makes it impossible to change the instantiation independently from (without having to change) the class.

b. Abstract Factory Design Pattern: The abstract factory pattern provides a way to encapsulate a group of individual factories that have a common theme without specifying their concrete classes. Client software creates a concrete implementation of the abstract factory and then uses the generic interface of the factory to create the concrete objects that are part of the theme. This pattern separates the details of implementation of a set of objects from their general usage and relies on object composition, as object creation is implemented in methods exposed in the factory interface.

The Abstract Factory design pattern solves problems like:

- How can an application be independent of how its objects are created?
- How can a class be independent of how the objects it requires are created?
- How can families of related or dependent objects be created?

Creating objects directly within the class that requires the objects is inflexible because it commits the class to objects and makes it impossible to change the instantiation later independently from (without having to change) the class. It stops the class from being reusable if other objects are required, and it makes the class hard to test because real objects can't be replaced with mock objects.

c. Observer Design Pattern: The observer pattern is a software design pattern in which an object, called the subject, maintains a list of its dependents, called observers, and notifies them automatically of any state changes, usually by calling one of their methods.

The Observer pattern addresses the following three problems:

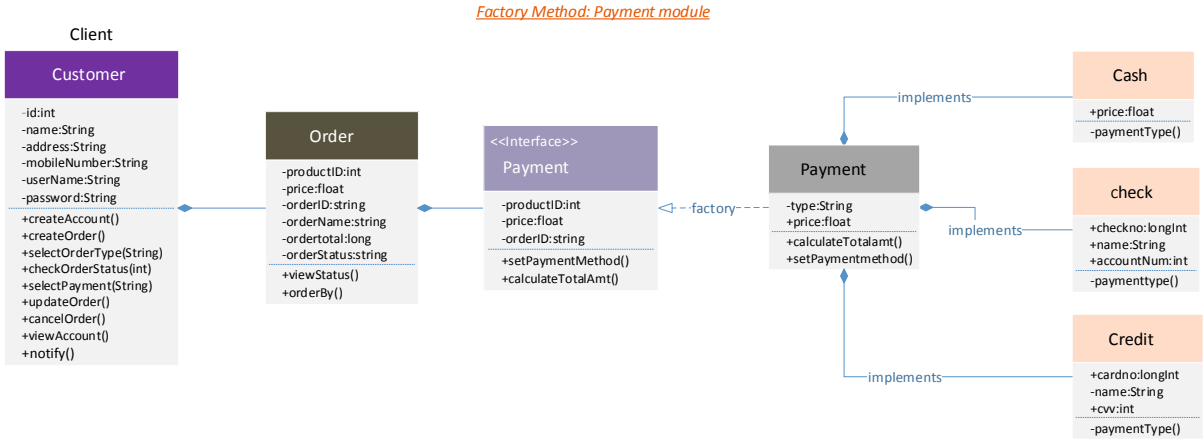
- A one-to-many dependency between objects should be defined without making the objects tightly coupled.
- It should be ensured that when one object changes state an open-ended number of dependent objects are updated automatically.
- It should be possible that one object can notify an open-ended number of other objects.

The observer pattern is implemented with the "subject" (which is being "observed") being part of the object whose state change is being observed, to be communicated to the observers upon occurrence.

4. Documentation how these design patterns are used in your design

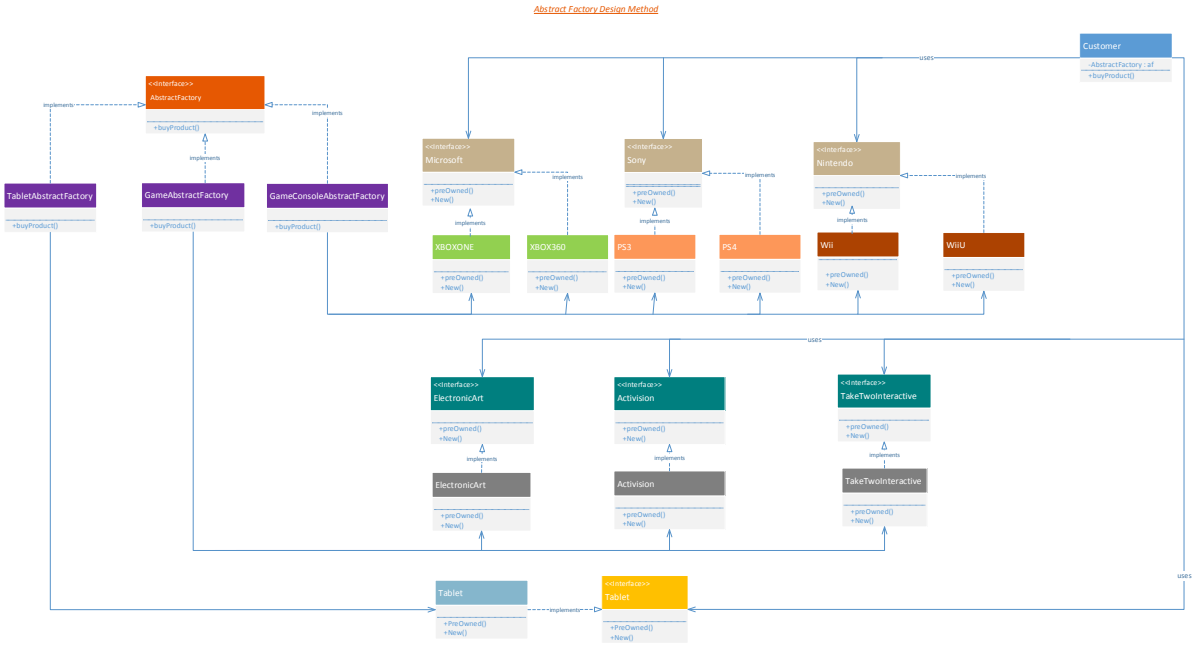
The usage of Design Pattern are as follows:

a. Factory Method Design Pattern:



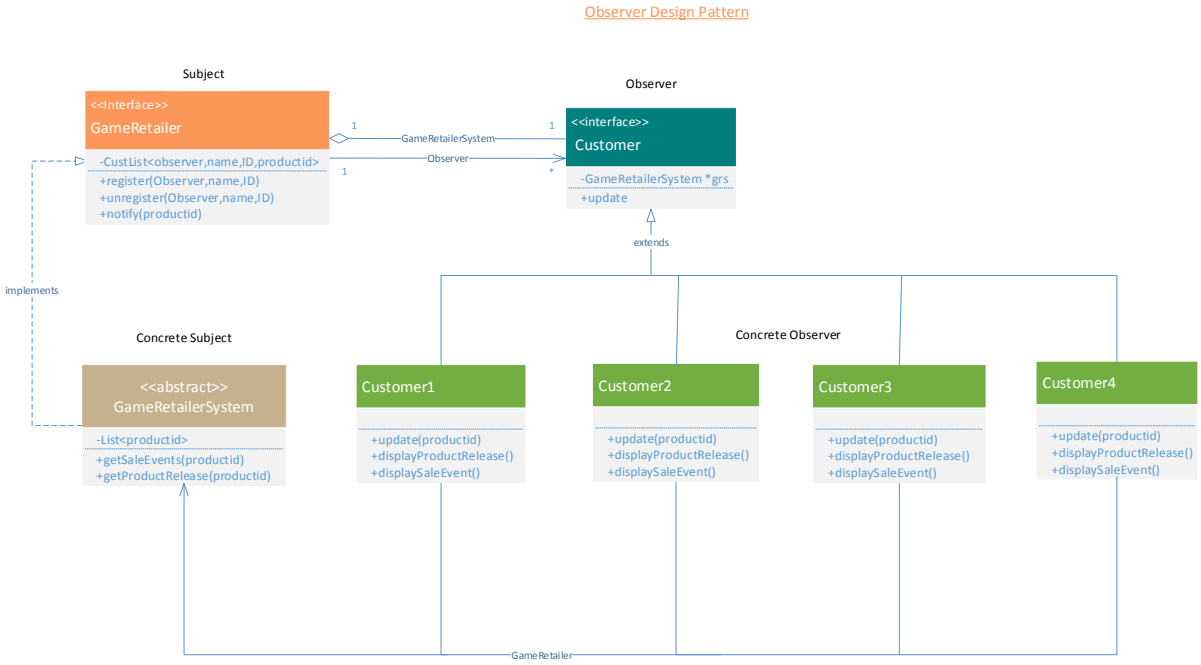
On Payment module, I have implemented factory method. Here we have different methods to pay for the order for example payment by cash, check, and credit. Each payment type has its own creator class connected to them. All the methods are provided by the Payment method which is super class acting as an interface for them. Customer uses the desired method to pay for the order. The Order class is declared with the Payment method statically. This is the factory where the customer process payment.
Customer has various option to pay for the services he has availed based on the monthly billing. All Payment methods have creator and they are connected to each of them. Payment Method is the Interface which provide various interface of payment type. Customer use this interface to select appropriate payment method suited best to his situation. Hence, the Factory design pattern is used in this case.

b. Abstract Factory Design Pattern:



On Product module, I have implemented abstract factory design pattern. The abstract factory pattern provides a way to encapsulate a group of individual factories (Table, Game Console and Games) that have a common method without specifying their concrete classes. This pattern separates the details of implementation of a set of objects from their general usage and relies on object composition, as object creation is implemented in methods exposed in the factory interface.
Hence, Abstract Factory Design Pattern is being used in this case.

c. Observer Design Pattern:



In Email subscription module observer pattern is used. Here the customer can subscribe / unsubscribe for email notifications when a new product is released and when there are any special sale events. In our case, Customer is the observer who use the interface subscription to use that offer. To subscribe / unsubscribe for email notification is being provided by the email notification interface.

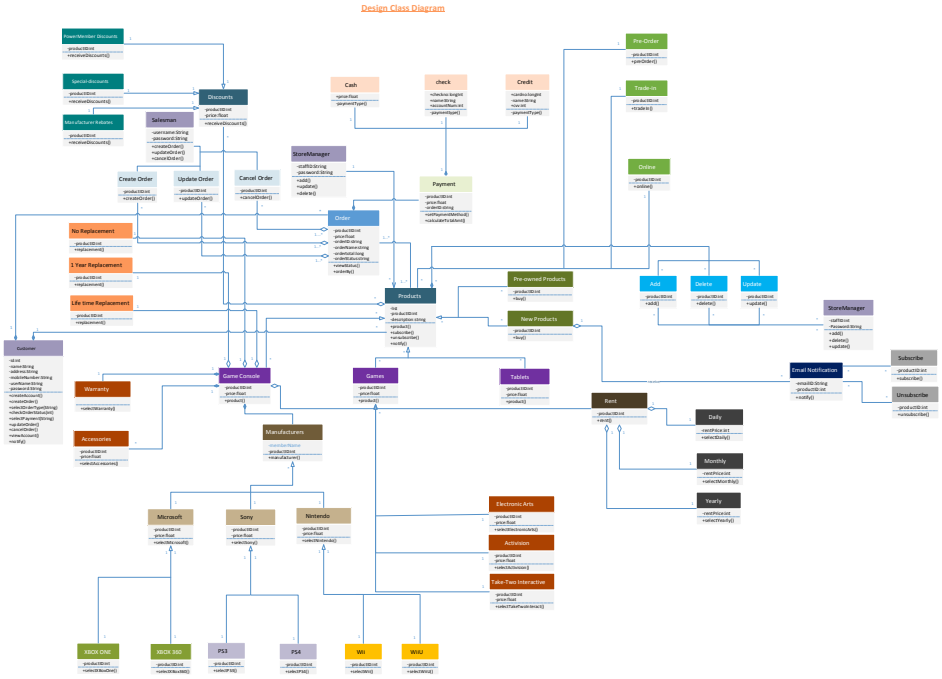
Here, this design pattern is used in the Customer Class. Customer can subscribe to email notifications for the game. This pattern generally represents a one to many relationships between observer and the many objects, i.e. the Customer and its subclasses. Also, the customer here can either choose to subscribe for updates from the customer class.
If there is any kind of change in the Customer Class irrespective of the reason, the classes will be notified. For example, if any new products will come, the customer will get email notification for that products and at the same time, customer can subscribe or unsubscribe the notification.

Hence, the Observer Design Pattern is used in this case.

Assignment #4

Assignment Deliverables:

1. UML Design Model/Class Diagram



2. List of the Design pattern(s) that you have used

The following are the design patterns which I have used in assignment #3.

a. **Factory Method Design Pattern:** Factory method pattern is a creational pattern that uses factory methods to deal with the problem of creating objects without having to specify the exact class of the object that will be created. This is done by creating objects by calling a factory method-either specified in an interface and implemented by child classes or implemented in a base class and optionally overridden by derived classes rather than by calling a constructor.

The Factory Method design pattern solves problems like:

- How can an object be created so that subclasses can redefine which class to instantiate?
- How can a class defer instantiation to subclasses?

Creating an object directly within the class that requires (uses) the object is inflexible because it commits the class to a object and makes

it impossible to change the instantiation independently from (without having to change) the class.

b. **Abstract Factory Design Pattern:** The abstract factory pattern provides a way to encapsulate a group of individual factories that have a common theme without specifying their concrete classes. Client software creates a concrete implementation of the abstract factory and then uses the generic interface of the factory to create the concrete objects that are part of the theme. This pattern separates the details of implementation of a set of objects from their general usage and relies on object composition, as object creation is implemented in methods exposed in the factory interface.

   The Abstract Factory design pattern solves problems like:

   - How can an application be independent of how its objects are created?
   - How can a class be independent of how the objects it requires are created?
   - How can families of related or dependent objects be created?

   Creating objects directly within the class that requires the objects is inflexible because it commits the class to particular objects and makes it impossible to change the instantiation later independently from (without having to change) the class. It stops the class from being reusable if other objects are required, and it makes the class hard to test because real objects can't be replaced with mock objects.

c. **Observer Design Pattern:** The observer pattern is a software design pattern in which an object, called the subject, maintains a list of its dependents, called observers, and notifies them automatically of any state changes, usually by calling one of their methods.

   The Observer pattern addresses the following three problems:

   - A one-to-many dependency between objects should be defined without making the objects tightly coupled.
   - It should be ensured that when one object changes state an open-ended number of dependent objects are updated automatically.
   - It should be possible that one object can notify an open-ended number of other objects.

   The observer pattern is implemented with the "subject" (which is being "observed") being part of the object whose state change is being observed, to be communicated to the observers upon occurrence.

The following design patterns I have used in Assignment #4:

(d). <u>Composite Design Pattern:</u>

The composite pattern describes a group of objects that is treated the same way as a single instance of the same type of object. The intent of a composite is to "compose" objects into tree structures to represent part-whole hierarchies.

Problems the Composite design pattern solve:

- A part-whole hierarchy should be represented so that clients can treat part and whole objects uniformly.
- A part-whole hierarchy should be represented as tree structure.

Solution the Composite design pattern describe:

- Define a unified Component interface for both part (Leaf) objects and whole (Composite) objects.
- Individual Leaf objects implement the Component interface directly, and Composite objects forward requests to their child components.

This enables clients to work through the Component interface to treat Leaf and Composite objects uniformly: Leaf objects perform a request directly, and Composite objects forward the request to their child components recursively downwards the tree structure. This makes client classes easier to implement, change, test, and reuse.

(e). <u>Strategy Design Pattern:</u>

Strategy Design Pattern is a behavioural software design pattern that enables selecting an algorithm at runtime. Instead of implementing a single algorithm directly, code receives run-time instructions as to which in a family of algorithms to use. The strategy pattern consists of three basic components:

- Strategy – A interfaced implementation of the core algorithm.
- Concrete Strategy – An actual implementation of the core algorithm, to be passed to the Client.
- Client – Stores a local Strategy instance, which is used to perform the core algorithm of that strategy.

The goal of the strategy design pattern is to allow the Client to perform the core algorithm, based on the locally-selected Strategy. In so doing, this allows different objects or data to use different strategies, independently of one another.

(f). Template Method Design Pattern:

Template method pattern is a behavioural design pattern that defines the program skeleton of an algorithm in an operation, deferring some steps to subclasses.

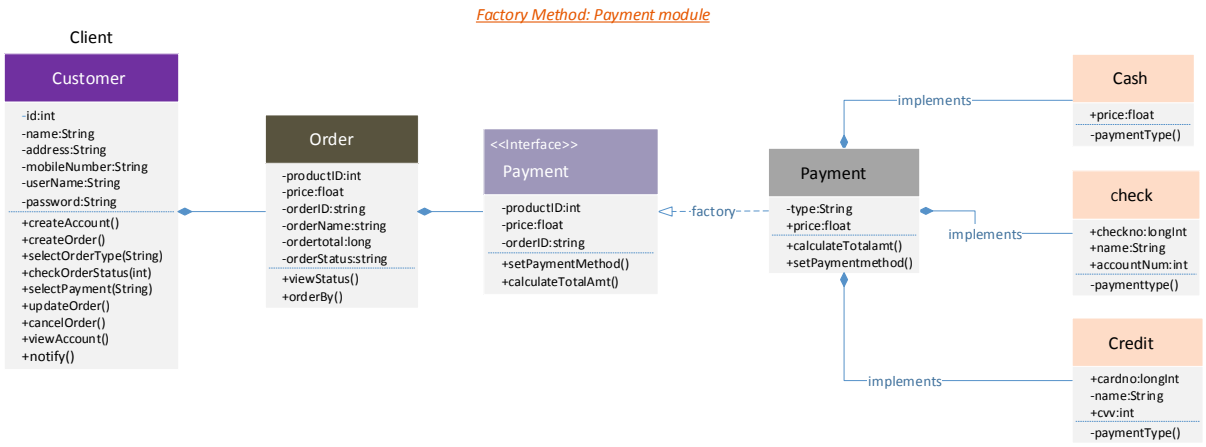
This pattern has two main parts, and typically uses object-oriented programming:

- The "template method", implemented as a base class which contains shared code and parts of the overall algorithm which are invariant. The template ensures that the overarching algorithm is always followed. In this class, "variant" portions are given a default implementation, or none.
- Concrete implementations of the abstract class, which fill in the empty or "variant" parts of the "template" with specific algorithms that vary from implementation to implementation.

3. Documentation on how these design patterns are used in your design.

Assignment #3 Design Patterns:

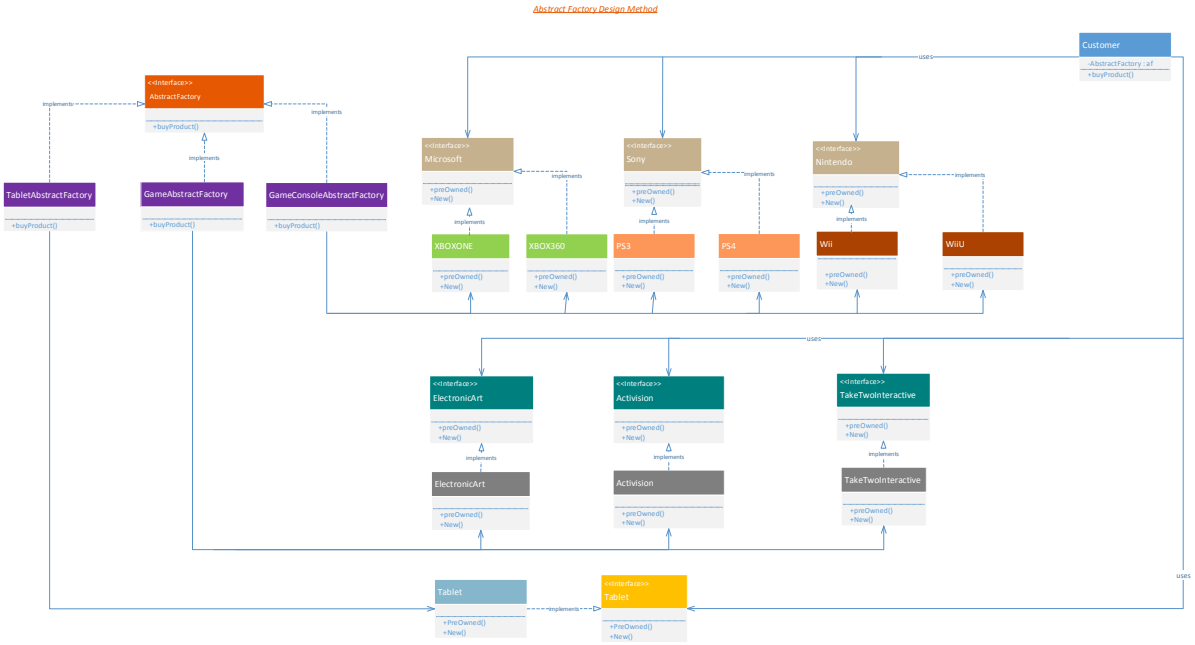
a. Factory Method Design Pattern:



On Payment module, I have implemented factory method. Here we have different methods to pay for the order for example payment by cash, check, and credit. Each payment type has its own creator class connected to them. All the methods are provided by the Payment method which is super class acting as an interface for them. Customer uses the desired method to pay for the order. The Order class is declared with the Payment method statically. This is the factory where the customer process payment.

Customer has various option to pay for the services he has availed based on the monthly billing. All Payment methods have creator and they are connected to each of them. Payment Method is the Interface which provide various interface of payment type. Customer use this interface to select appropriate payment method suited best to his situation. Hence, the Factory design pattern is used in this case.

b. Abstract Factory Design Pattern:

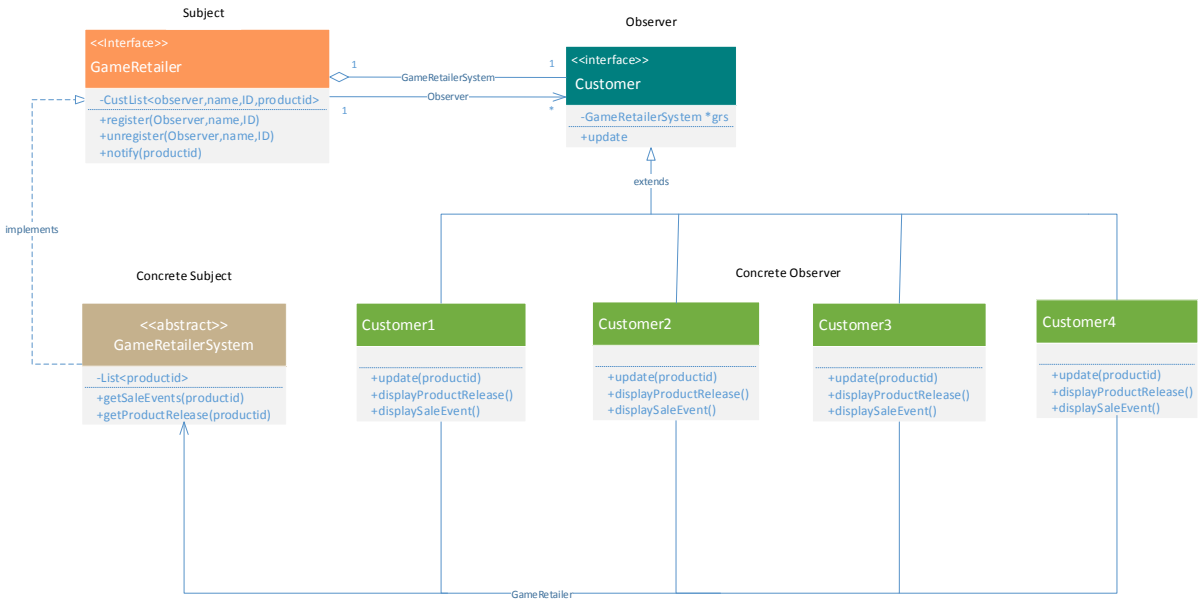


On Product module, I have implemented abstract factory design pattern. The abstract factory pattern provides a way to encapsulate a group of individual factories (Table, Game Console and Games) that have a common method without specifying their concrete classes. This pattern separates the details of implementation of a set of objects from their general usage and relies on object composition, as object creation is implemented in methods exposed in the factory interface.

Hence, Abstract Factory Design Pattern is being used in this case.

c. Observer Design Pattern:

Observer Design Pattern



In Email subscription module observer pattern is used. Here the customer can subscribe / unsubscribe for email notifications when a new product is released and when there are any special sale events. In our case, Customer is the observer who use the interface subscription to use that offer. To subscribe / unsubscribe for email notification is being provided by the email notification interface.

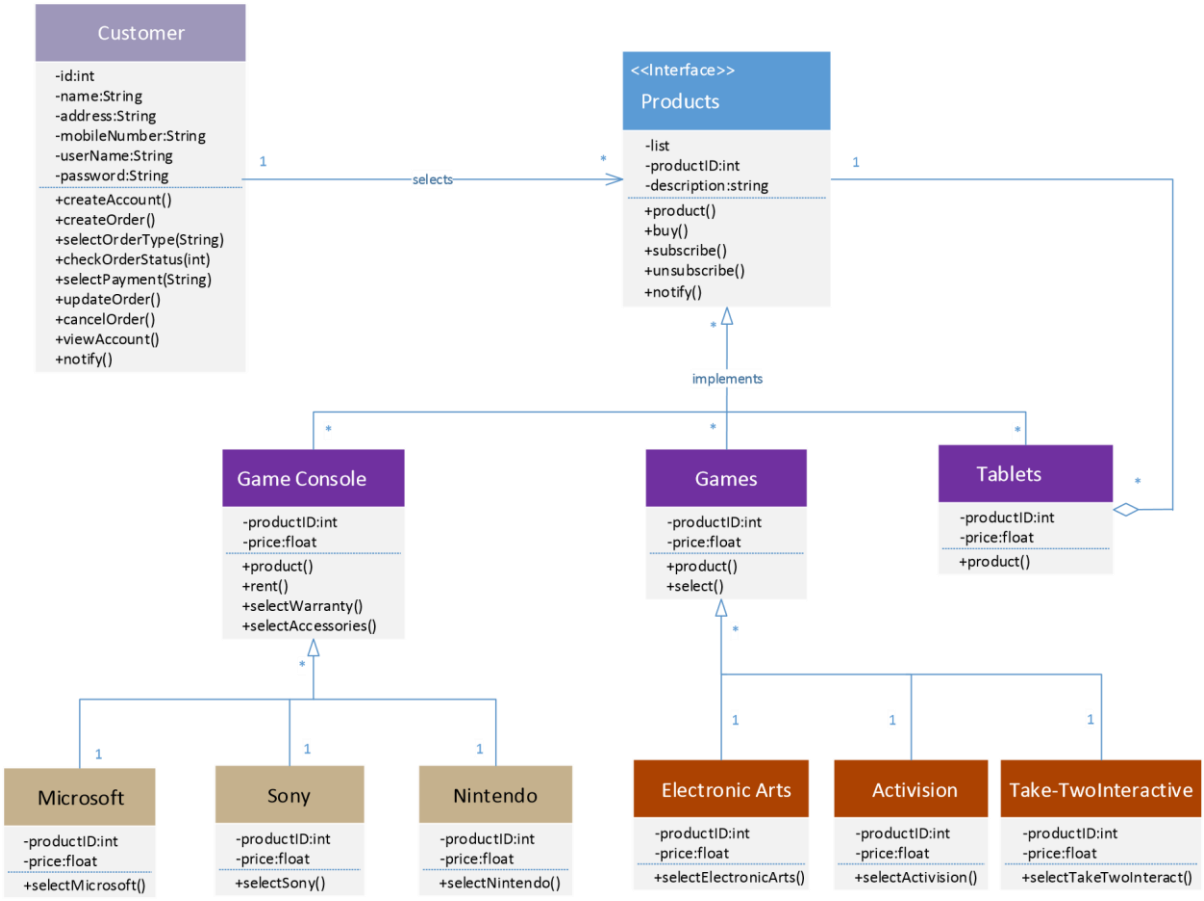
Here, this design pattern is used in the Customer Class. Customer can subscribe to email notifications for the game. This pattern generally represents a one to many relationships between observer and the many objects, i.e. the Customer and its subclasses. Also, the customer here can either choose to subscribe for updates from the customer class. If there is any kind of change in the Customer Class irrespective of the reason, the classes will be notified. For example, if any new products will come, the customer will get email notification for that products and at the same time, customer can subscribe or unsubscribe the notification.

Hence, the Observer Design Pattern is used in this case.

Assignment #4 Design Patterns:

d. Composite Design Pattern:

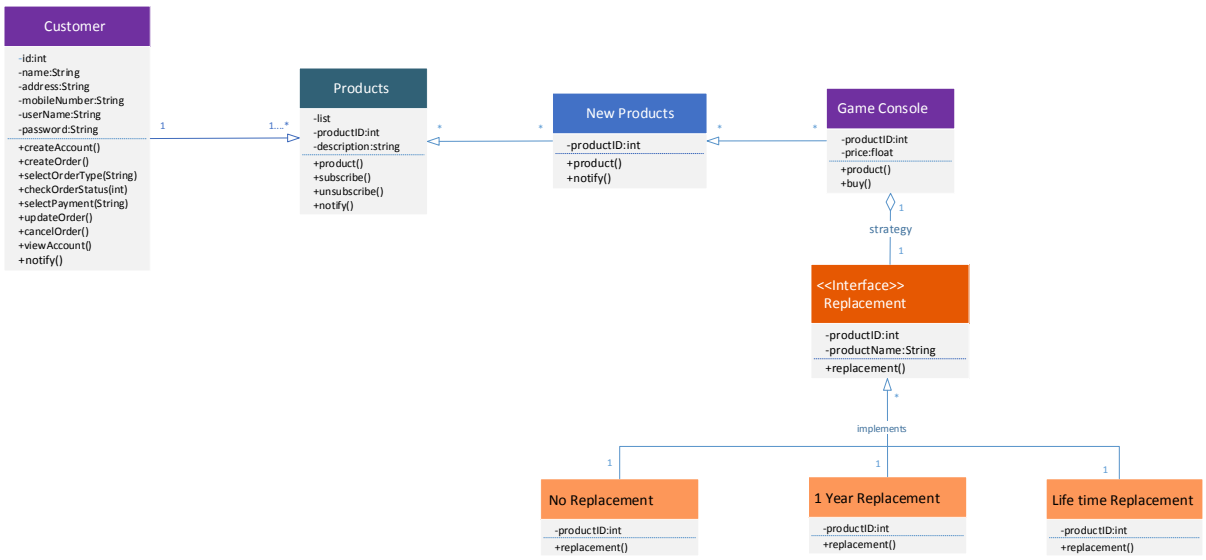
Composite Design Pattern: Products



Here, Composite design pattern is used in the Products Class. It consists of three subclasses:
GameConsole, Games and Tablets. Where GameConsole has three product types as Microsoft, Sony, and Nintendo. This design pattern is generally used, when a hierarchy is used during the coding process. Using this design pattern, we can apply same operations over both the composite and individual object i.e. the GameConsole object and Microsoft, Sony & Nintendo which are individual objects. Both these classes will inherit the methods of the Products class. The customer here can choose any product types.
This pattern allows us to create part to whole hierarchies. It also allows clients to treat both the individual object i.e. Microsoft, Sony & Nintendo as well as the composite object 'GameConsole' with uniformity i.e. both will implement all the attributes and methods of the composite object. There is no difference in the properties of both these product types except for the fact that one is selected by the customer if they want to make use of Microsoft, Sony or Nintendo inside GameConsole.
Hence, the Composite Design Pattern is used in this case.

(e). Strategy Design Pattern:

Strategy Design Pattern: Game Console

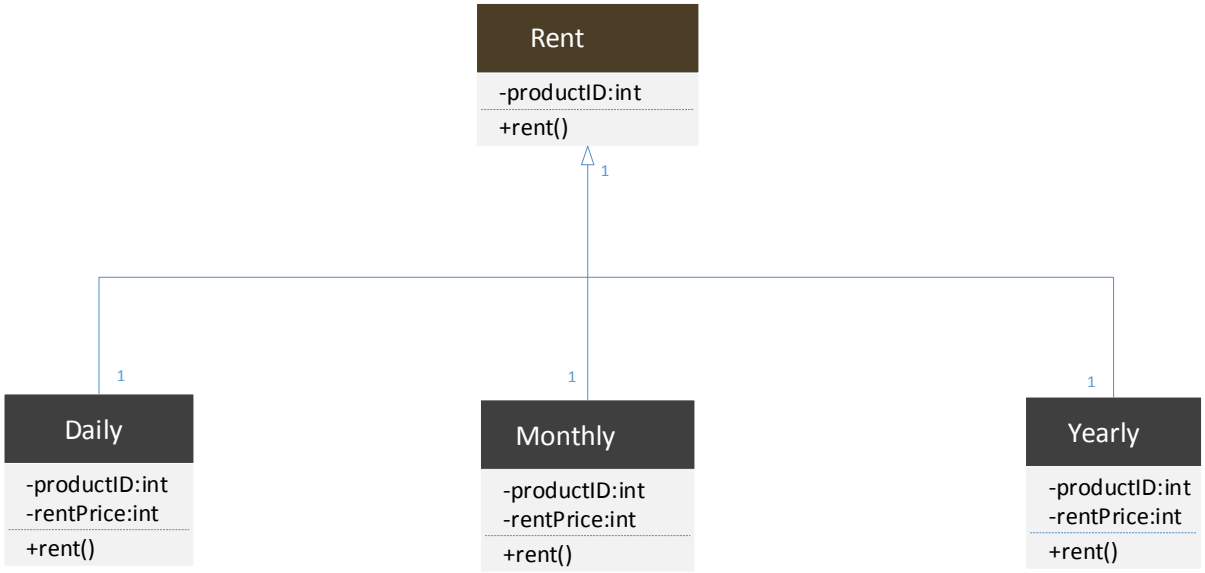


Here Customer can choose any of the replacement services which help customer to replace the products based on their choice. I have used strategy pattern to show the proper strategy for choosing the replacement options which includes the replacement strategy as per customer need.
A Strategy defines a set of algorithms that can be used interchangeably. Replacement is an example of a Strategy. Several options exist such as NoReplacement, 1YearReplacement and LifeTimeReplacement. Any of these Replacements will have many choices to select. The customer must choose the Strategy based on Replacement between cost, convenience, and time.

In Strategy pattern, a class behaviour or its algorithm can be changed at run time. This type of design pattern comes under behaviour pattern. We create objects which represent various strategies and a context object whose behaviour varies as per its strategy object. The strategy object changes the executing algorithm of the context object. Hence the Strategy design pattern is used in this case.

(f). Template Method Design Pattern:

Template Design Pattern: Rent



In Template pattern, an abstract class exposes defined template to execute its methods. Its subclasses can override the method implementation as per need, but the invocation is to be in the same way as defined by an abstract class. This pattern comes under behaviour pattern category.

In Rent, the customer can get three types of rents- Daily, Monthly, Yearly plans. Here, Rent is the abstract class for Daily, Monthly and Yearly plans.

Assignment #5

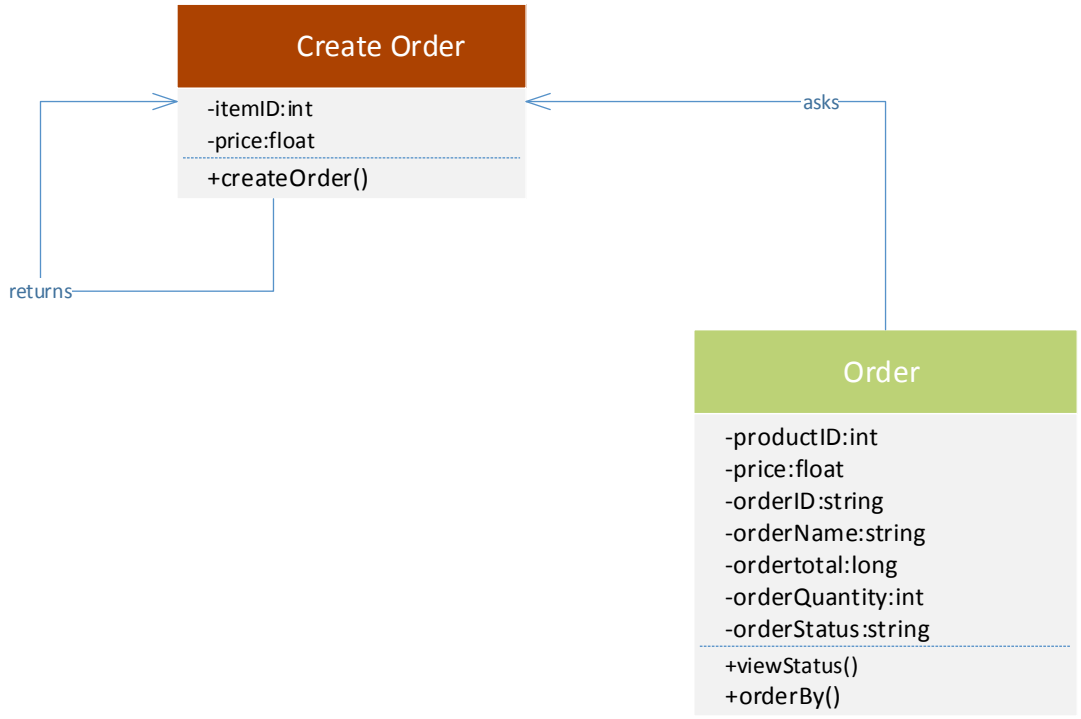
Deliverables:

1. Two new design pattern used

a. **Singleton Pattern:** Used on Create Order module

Singleton pattern is a software design pattern that restricts the instantiation of a class to one object. This is useful when exactly one object is needed to coordinate actions across the system. The concept is sometimes generalized to systems that operate more efficiently when only one object exists, or that restrict the instantiation to a certain number of objects. The term comes from the mathematical concept of a singleton.

Singleton Design Pattern



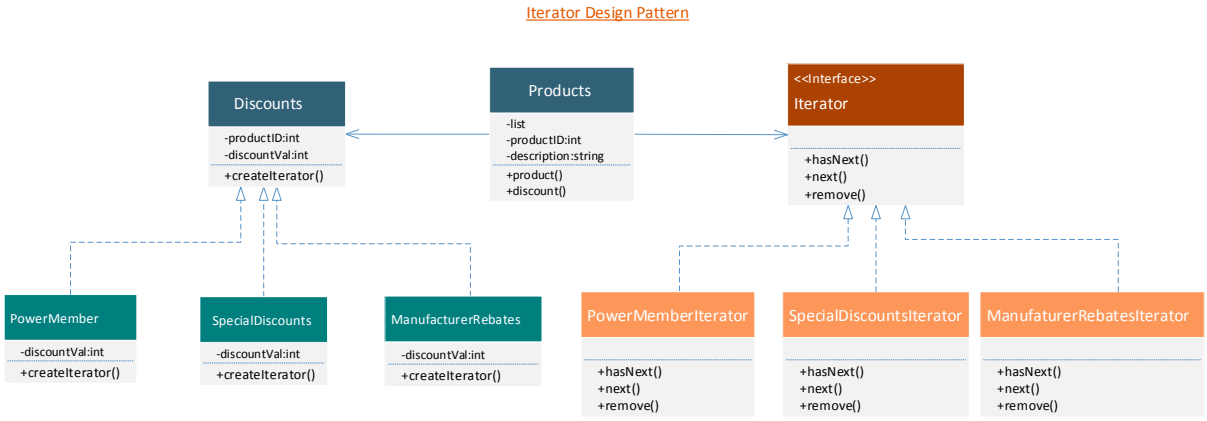
Here, this Design Pattern is used in the Create Order class. This ensures that this class has only one instance and provides a global point of access to it. Using the variable createOrder, which is of the type Boolean, the entire system can know that whether the order has been created. Once, these are created, a message of created order gets notified.
This class has made sure that the create order method allows us to instantiate the CreateOrder class singlehandedly and to return an instance of it. Hence, the Singleton Design Pattern is used in this case.

This pattern involves a single class which is responsible to create an object while making sure that only single object gets created. This class provides a way to access its only object which can be accessed directly without need to instantiate the object of the class.

b. **Iterator Pattern:** Used on Discounts module

Iterator pattern is a design pattern in which an iterator is used to traverse a container and access the container's elements. The iterator pattern decouples algorithms from containers; in some cases, algorithms are necessarily container-specific and thus cannot be decoupled. The abstraction provided by the iterator pattern allows you to modify the collection implementation without making any changes outside of collection.

Here, I have used Iterator design pattern on discount module given on each product. While purchasing a product, discount is given on each product. However, product is decoupled from discount mechanism and have nothing to with the implantation of discount given.

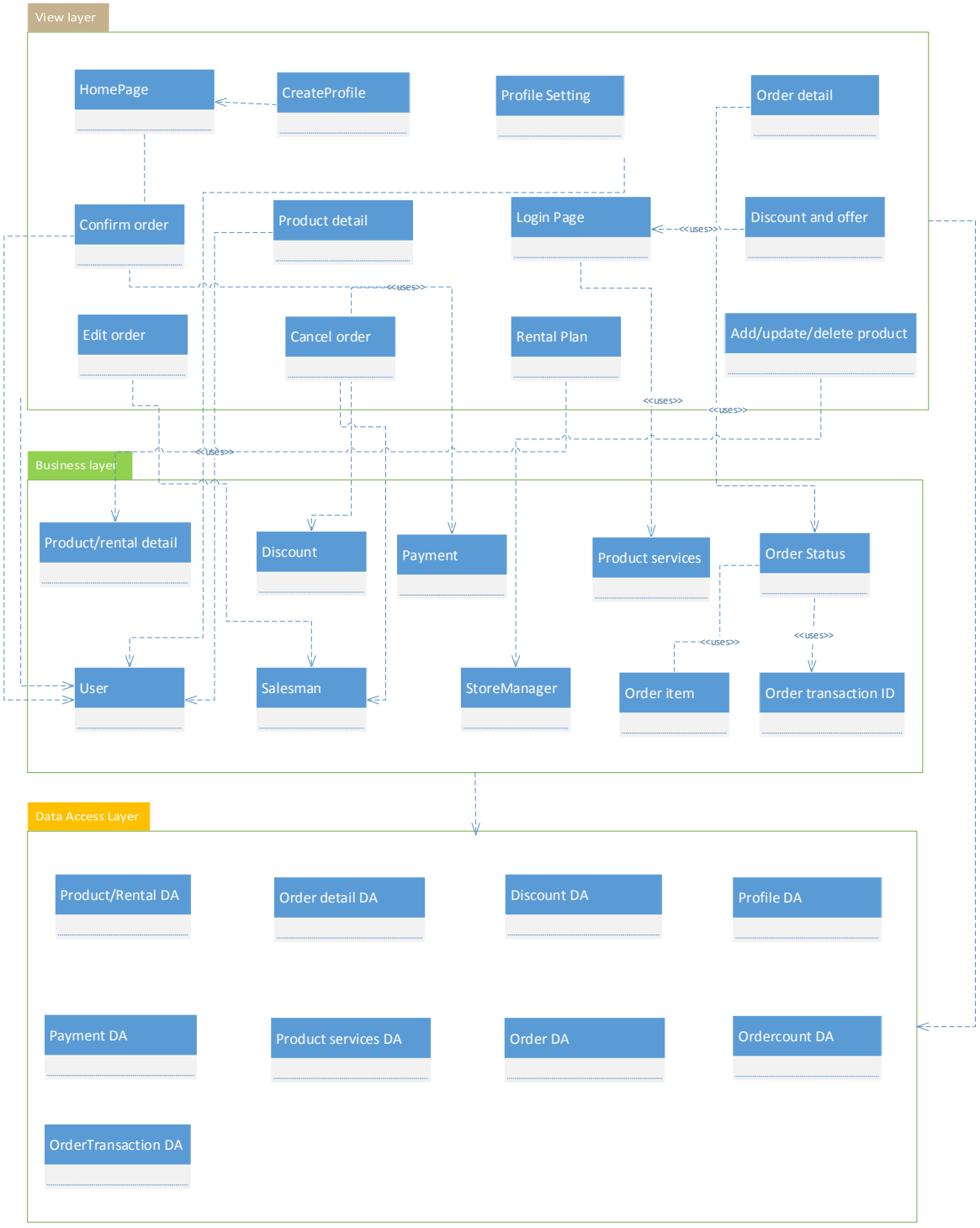


2. MVC architectural pattern to model and document all artifacts into the appropriate levels of MVC

MVC Pattern stands for Model-View-Controller Pattern. MVC is an architectural pattern commonly used for developing user interfaces that divides an application into three interconnected parts. This is done to separate internal representations of information from the ways information is presented to and accepted from the user. It has three major components:

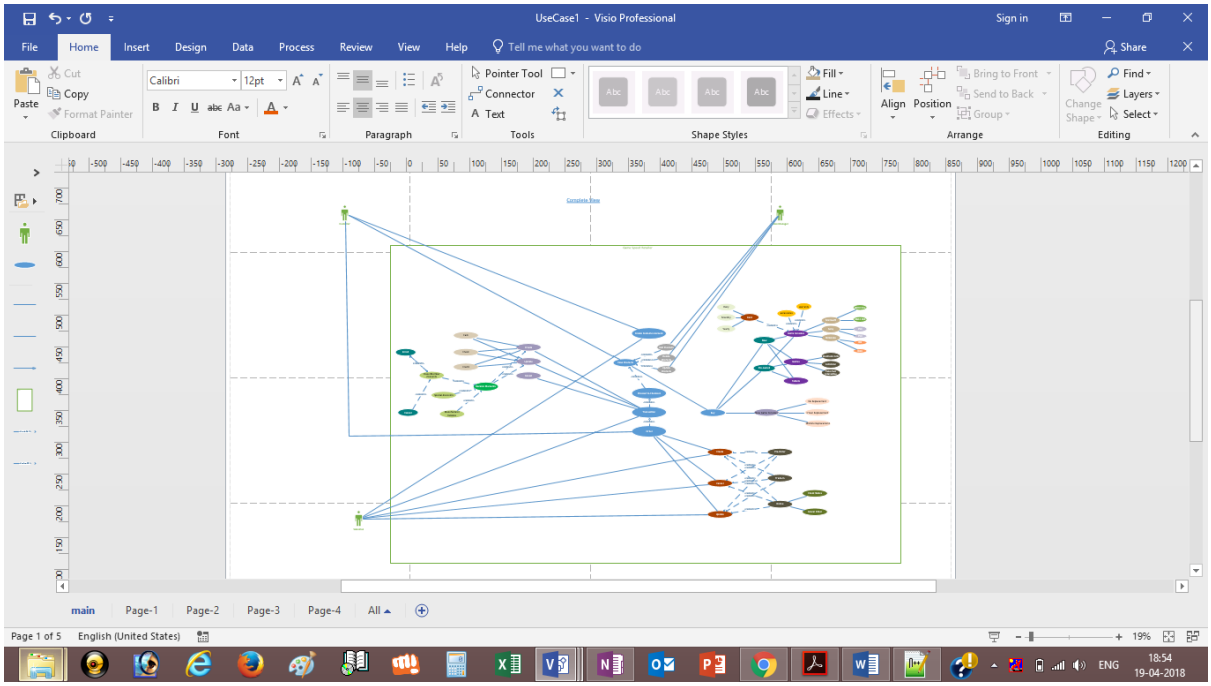
- **Model** - Model represents an object or JAVA POJO carrying data. It can also have logic to update controller if its data changes.
- **View** - View represents the visualization of the data that model contains.
- **Controller** - Controller acts on both model and view. It controls the data flow into model object and updates the view whenever data changes. It keeps view and model separate.

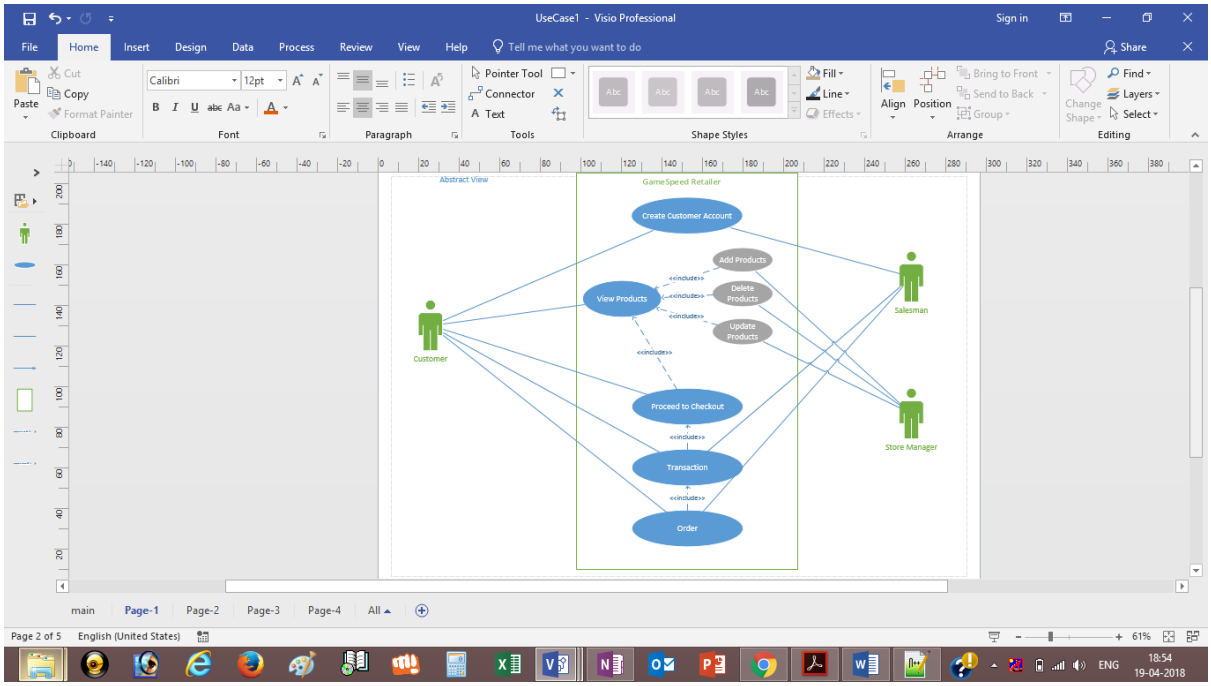
MVC Architectural Pattern

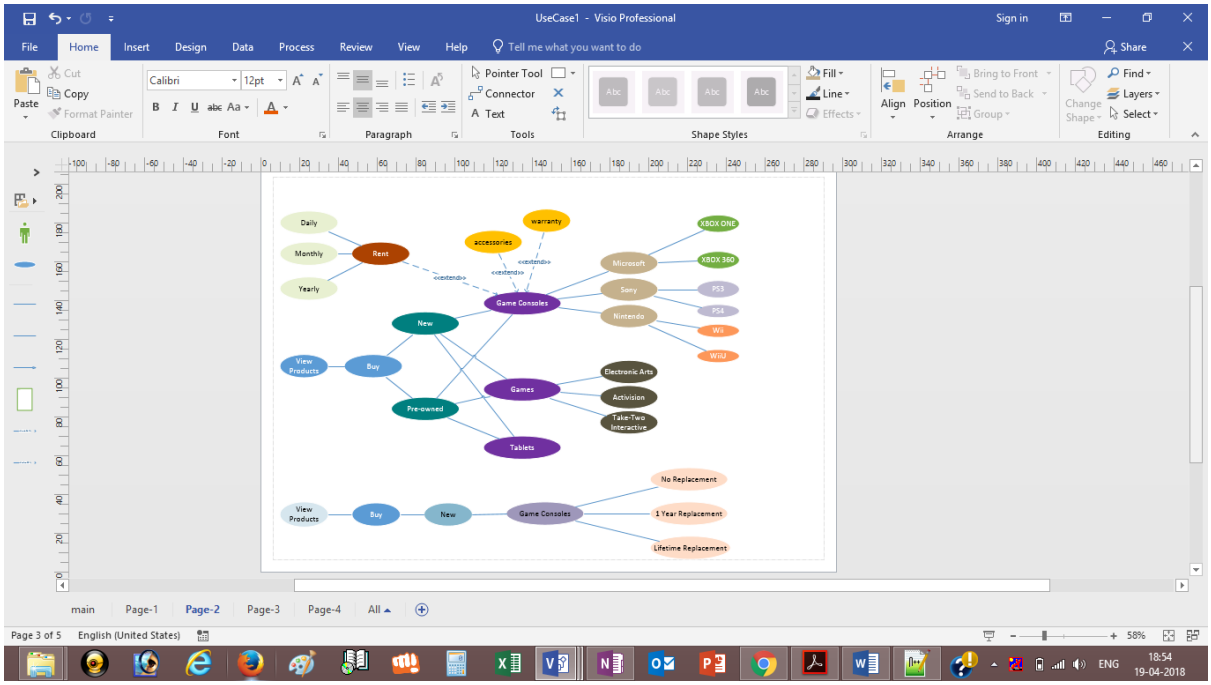


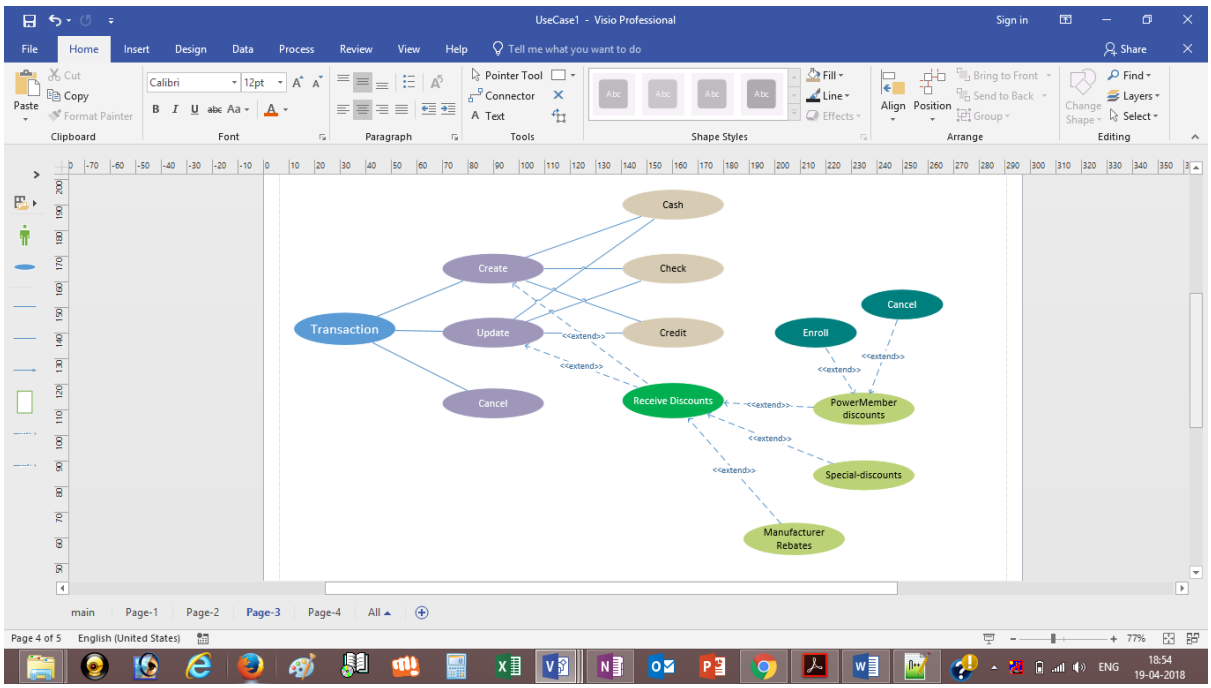
All Screenshots

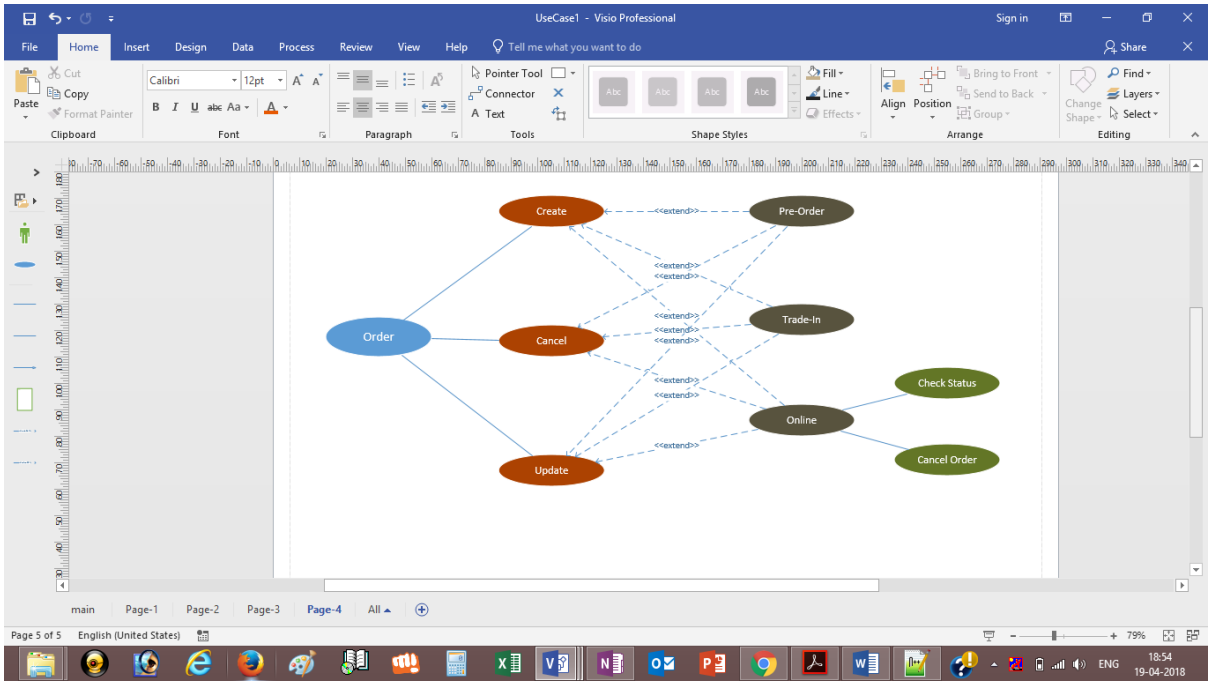
a. Screenshot of Usecase Diagrams



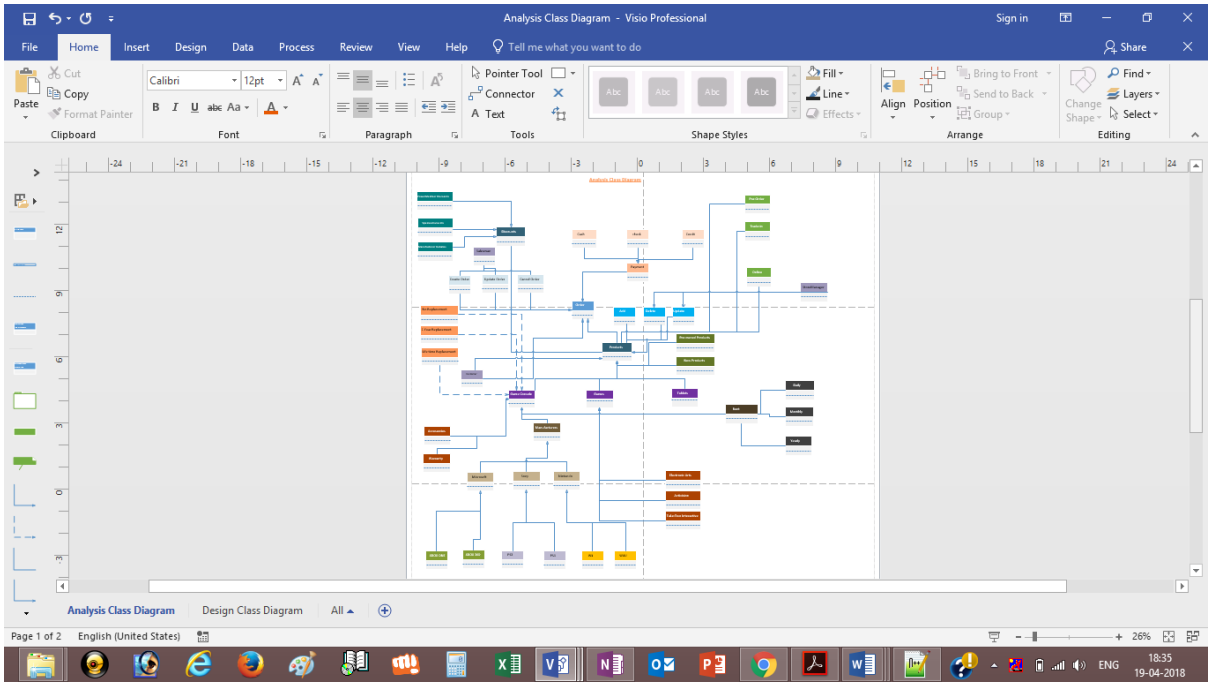



UseCase1 - Visio Professional
Daily
Monthly
Yearly
Rent
warranty
accessories
<<extend>>
Microsoft
Sony
Nintendo
XBOX ONE
XBOX 360
PS3
PS4
Wii
WiiU
Game Consoles
New
View Products
Buy
Pre-owned
Electronic Arts
Activision
Take-Two Interactive
Games
Tablets
No Replacement
1 Year Replacement
Lifetime Replacement
main
Page-1
Page-2
Page-3
Page-4
All

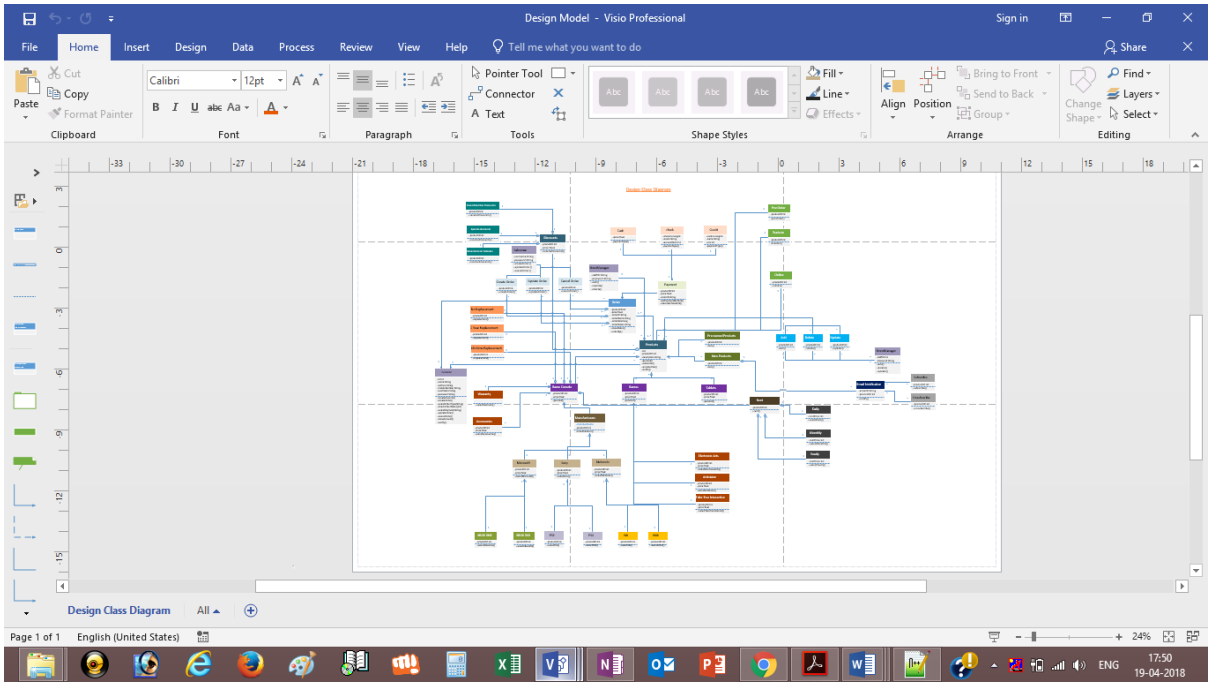

UseCase1 - Visio Professional
Cash
Create
Check
Transaction
Update
Credit
Cancel
Enroll
<<extend>>
Receive Discounts
PowerMember discounts
Special-discounts
Manufacturer Rebates
main
Page-1
Page-2
Page-3
Page-4
All



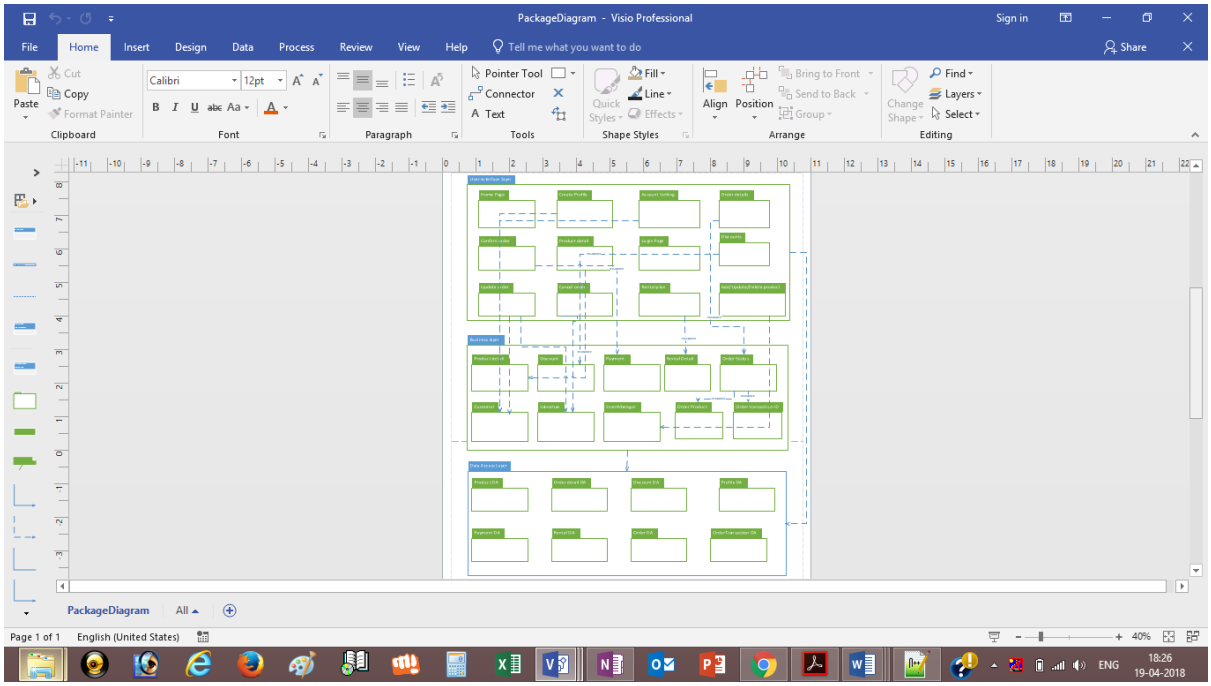
b. Screenshot of Analysis Class Diagram



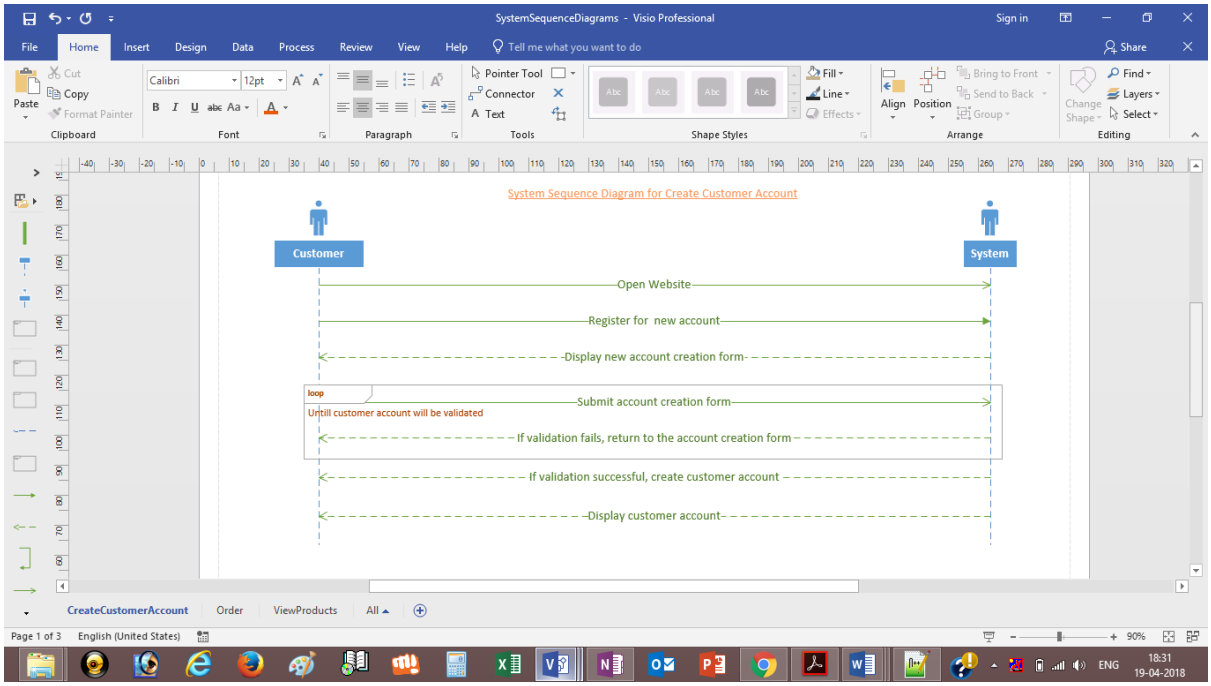
c. Screenshot of Design Class Diagram

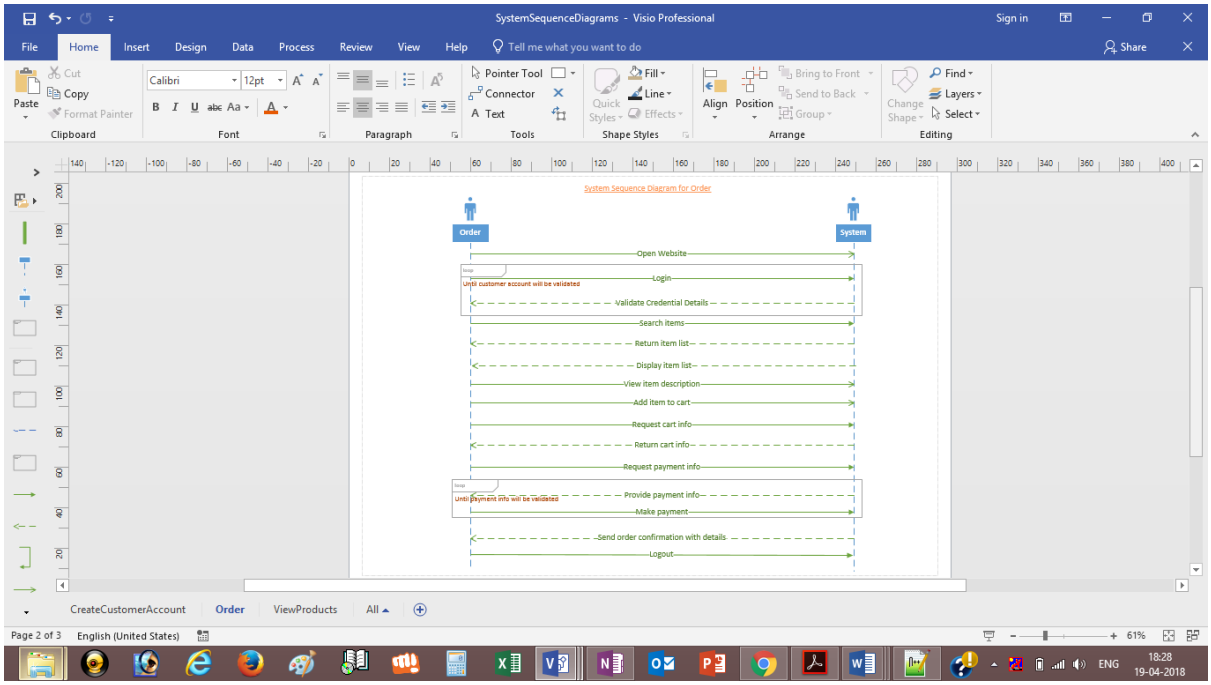


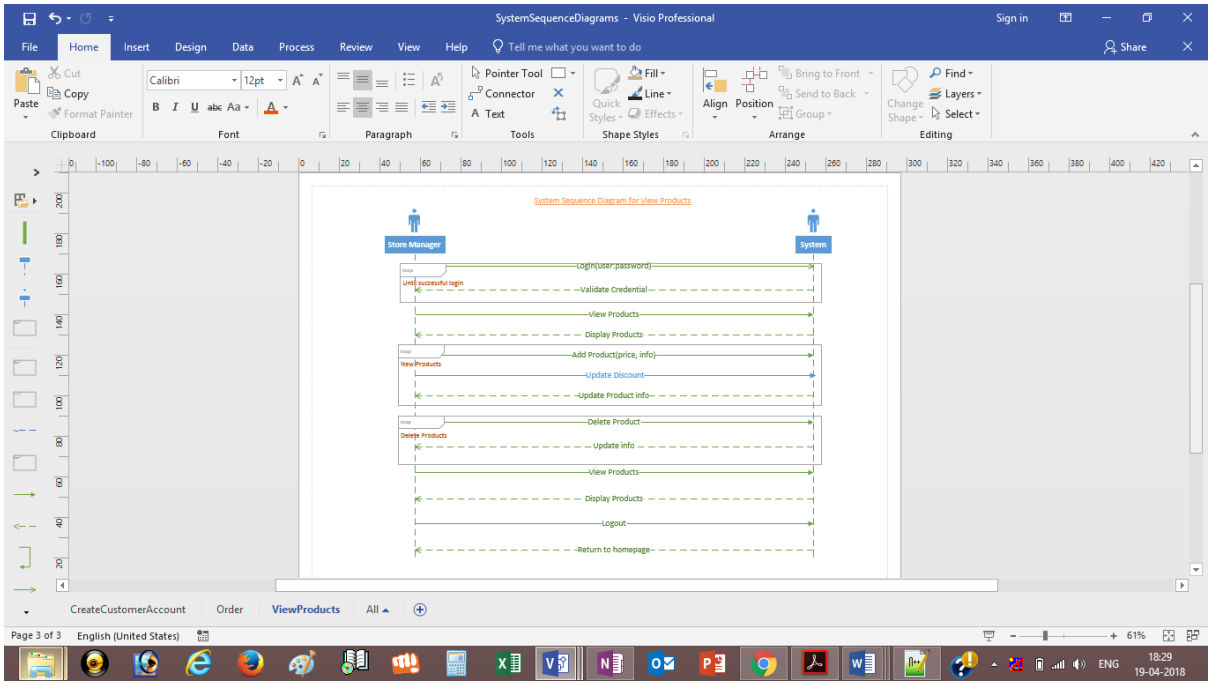
d. Screenshot of Package Diagram



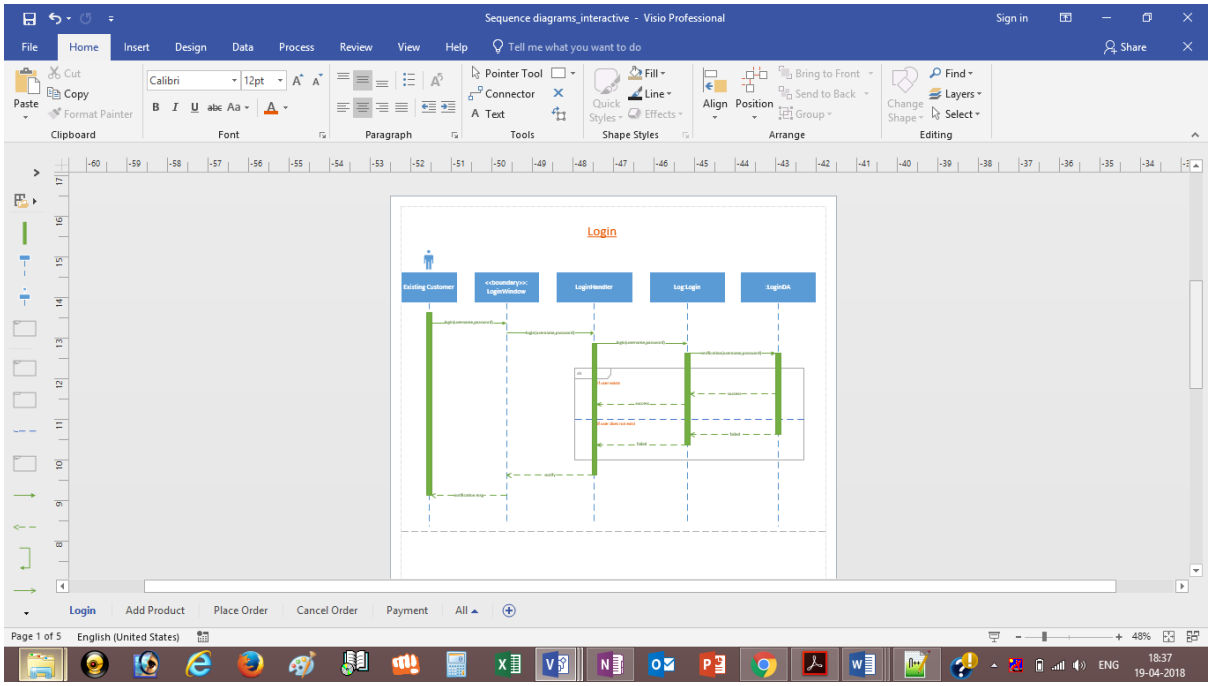
e. Screenshot of System Sequence Diagrams

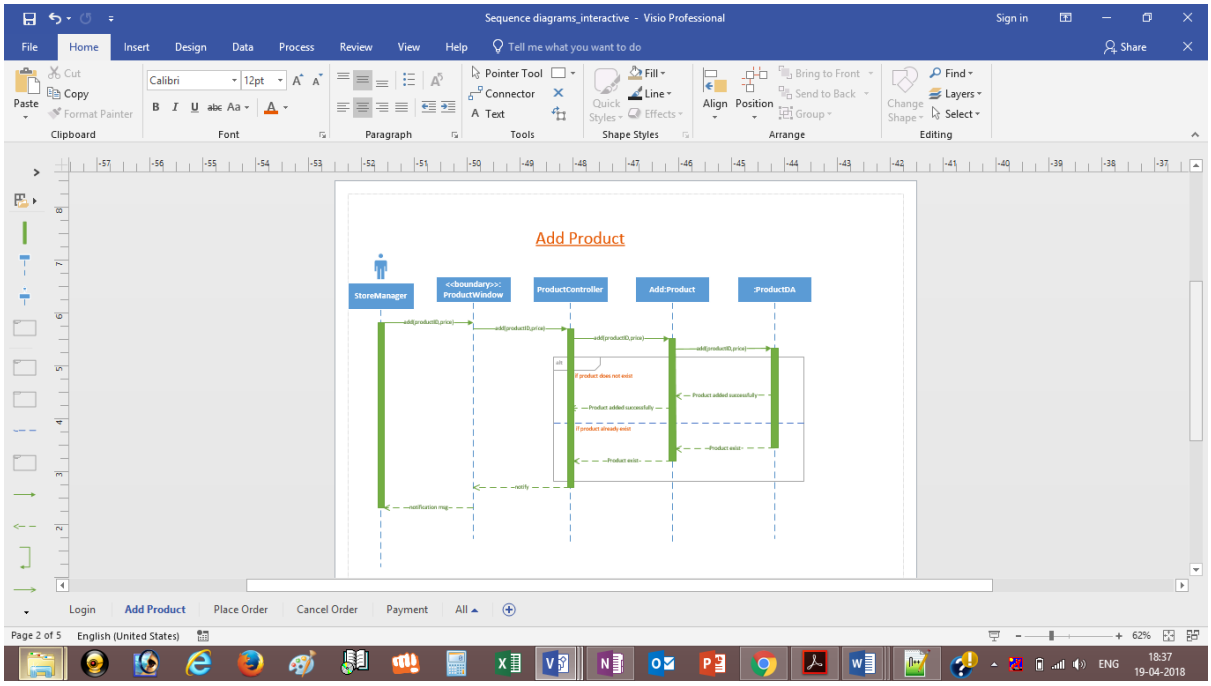


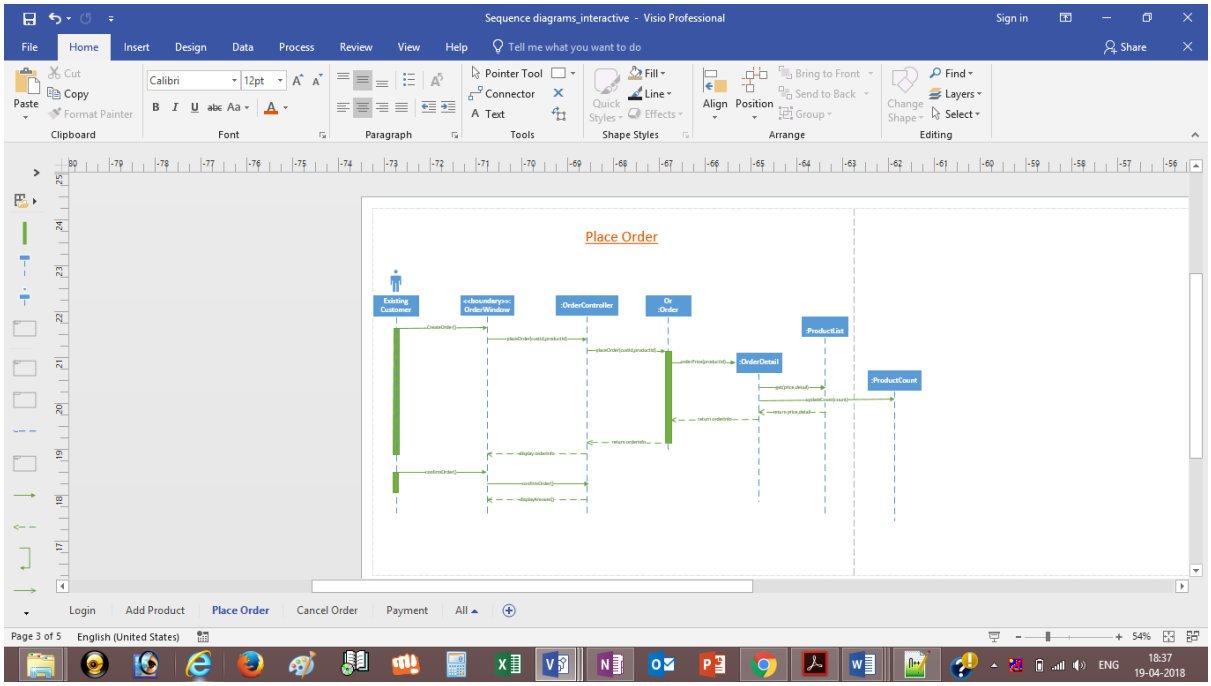


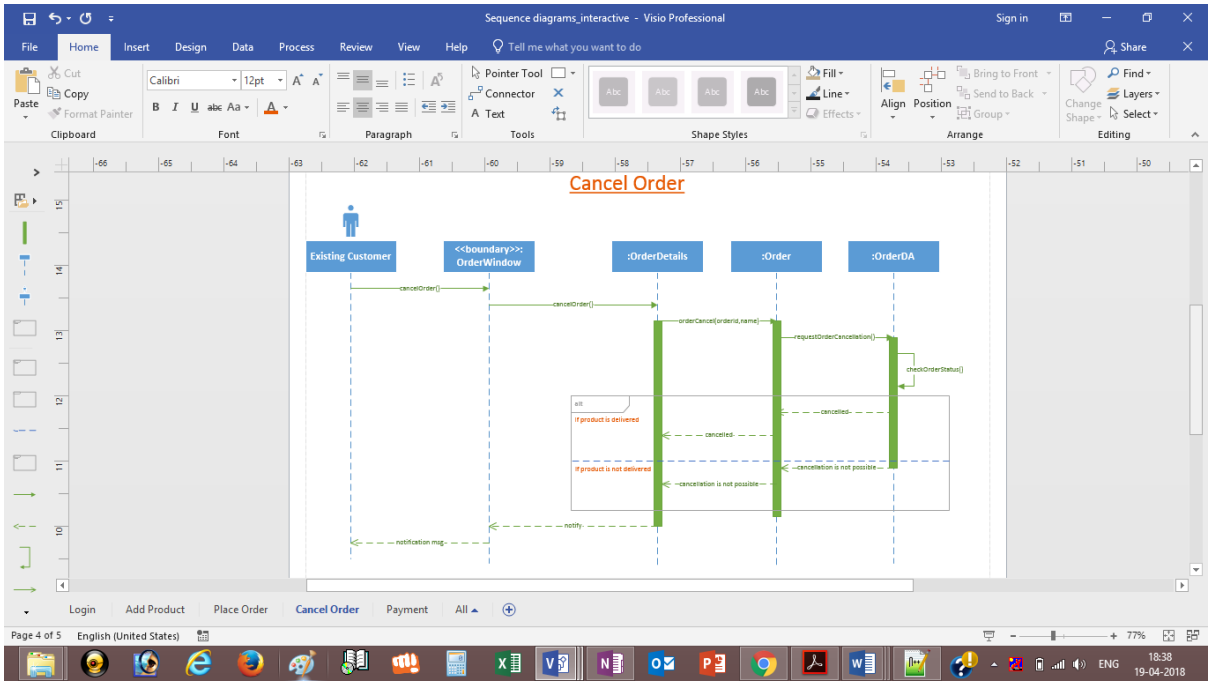


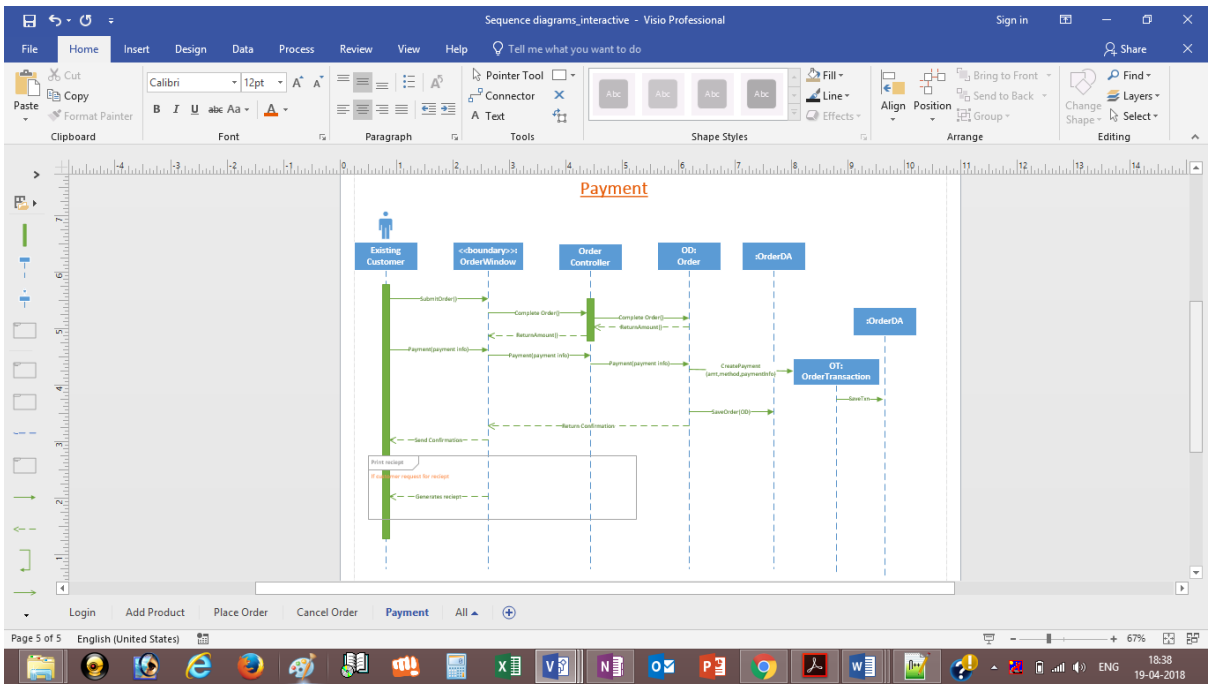
f. Screenshot of System Interaction Diagrams



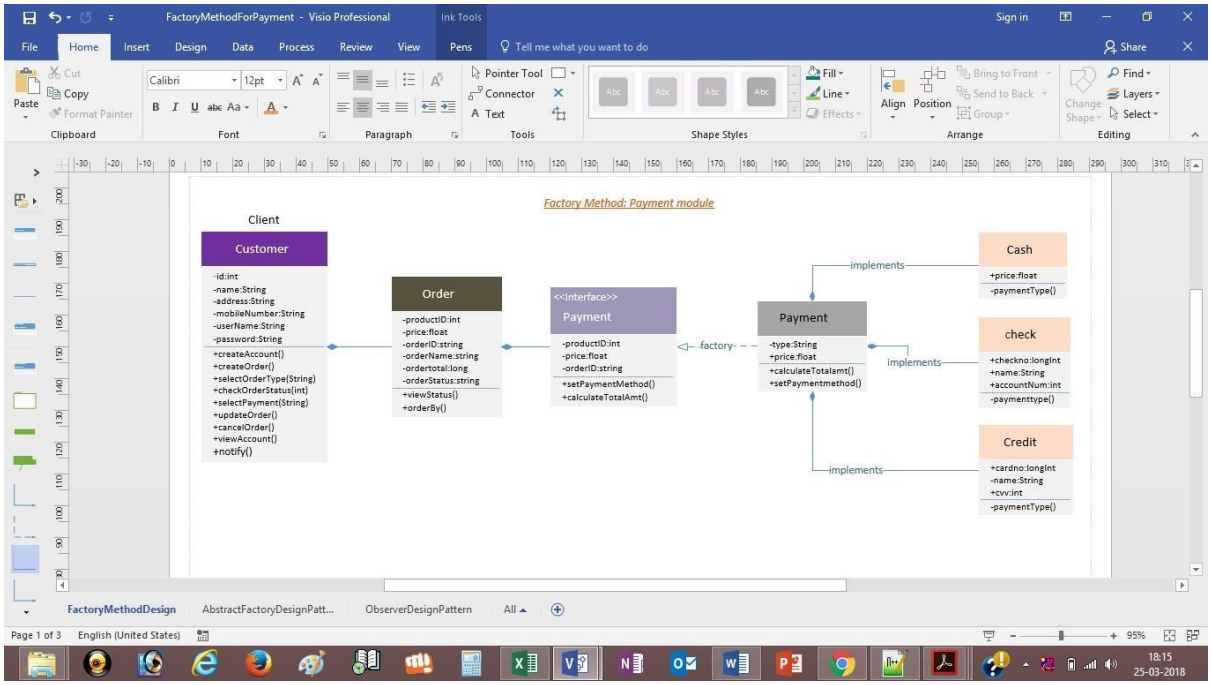

Sequence diagrams_interactive - Visio Professional
Add Product
StoreManager
<<boundary>>: ProductWindow
ProductController
Add:Product
:ProductDA
Login
Add Product
Place Order
Cancel Order
Payment
All
Page 2 of 5
English (United States)
62%
ENG
18:37
19-04-2018


Sequence diagrams_interactive - Visio Professional
Place Order
Existing Customer
<<boundary>>: OrderWindow
:OrderController
Or :Order
:ProductList
:OrderDetail
:ProductCount
Login
Add Product
Place Order
Cancel Order
Payment
All
Page 3 of 5
English (United States)
54%
ENG
18:37
19-04-2018

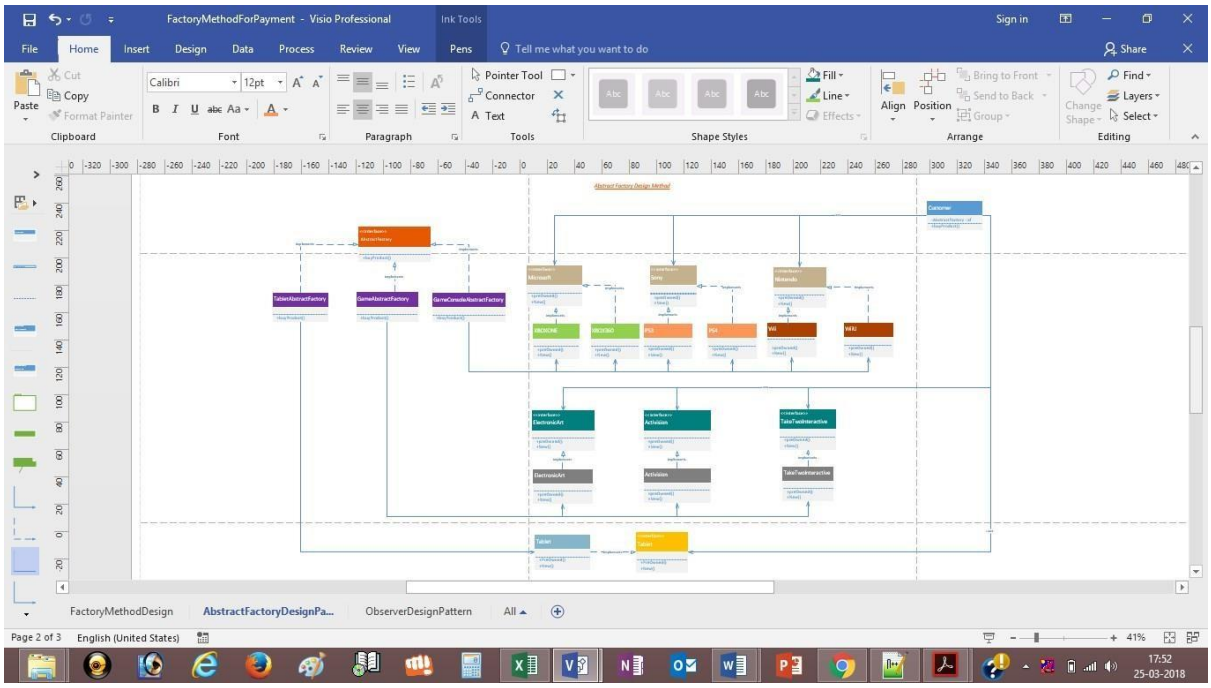

Sequence diagrams_interactive - Visio Professional
Cancel Order
Existing Customer
<<boundary>>: OrderWindow
:OrderDetails
:Order
:OrderDA
cancelOrder()
orderCancel(orderId,name)
requestOrderCancellation()
checkOrderStatus()
If product is delivered
If product is not delivered
cancelled
cancellation is not possible
notify
notification msg
Login
Add Product
Place Order
Cancel Order
Payment
All
Page 4 of 5
English (United States)


Sequence diagrams_interactive - Visio Professional
Payment
Existing Customer
<<boundary>>: OrderWindow
Order Controller
OD: Order
:OrderDA
OT: OrderTransaction
SubmitOrder()
CompleteOrder()
ReturnsAmount()
Payment(payment info)
SaveOrder(OD)
ReturnConfirmation
SendConfirmation
Login
Add Product
Place Order
Cancel Order
Payment
All
Page 5 of 5
English (United States)

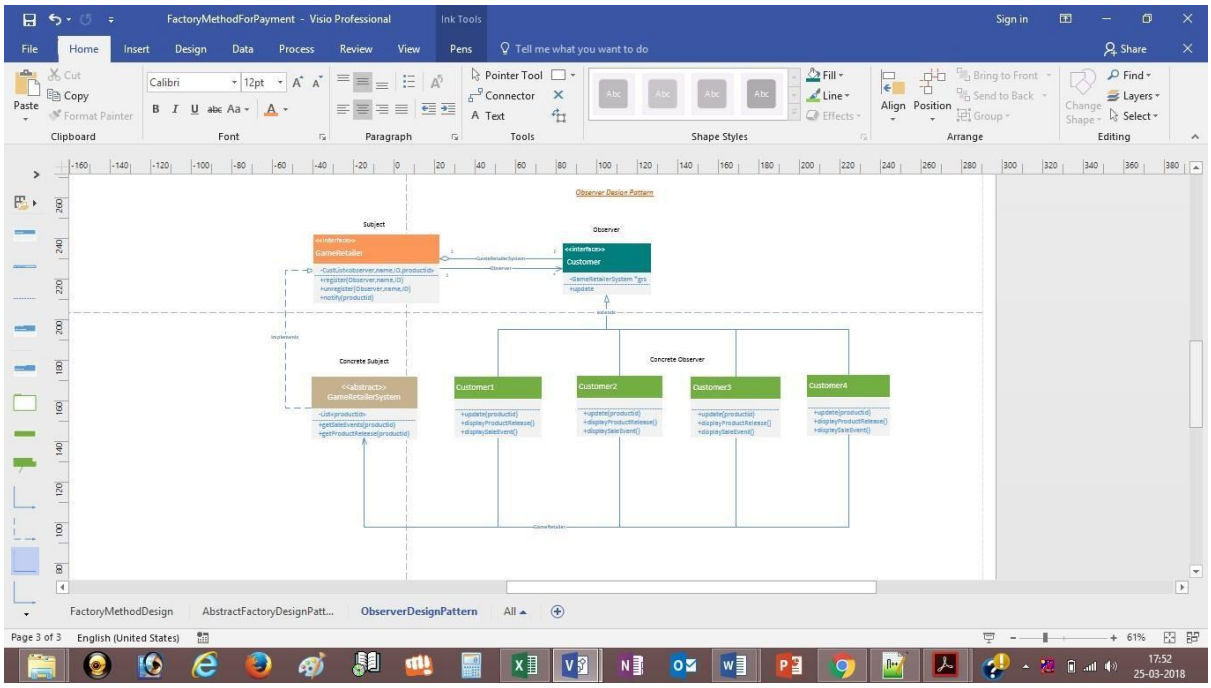
g. Screenshot of Factory Method Design Pattern



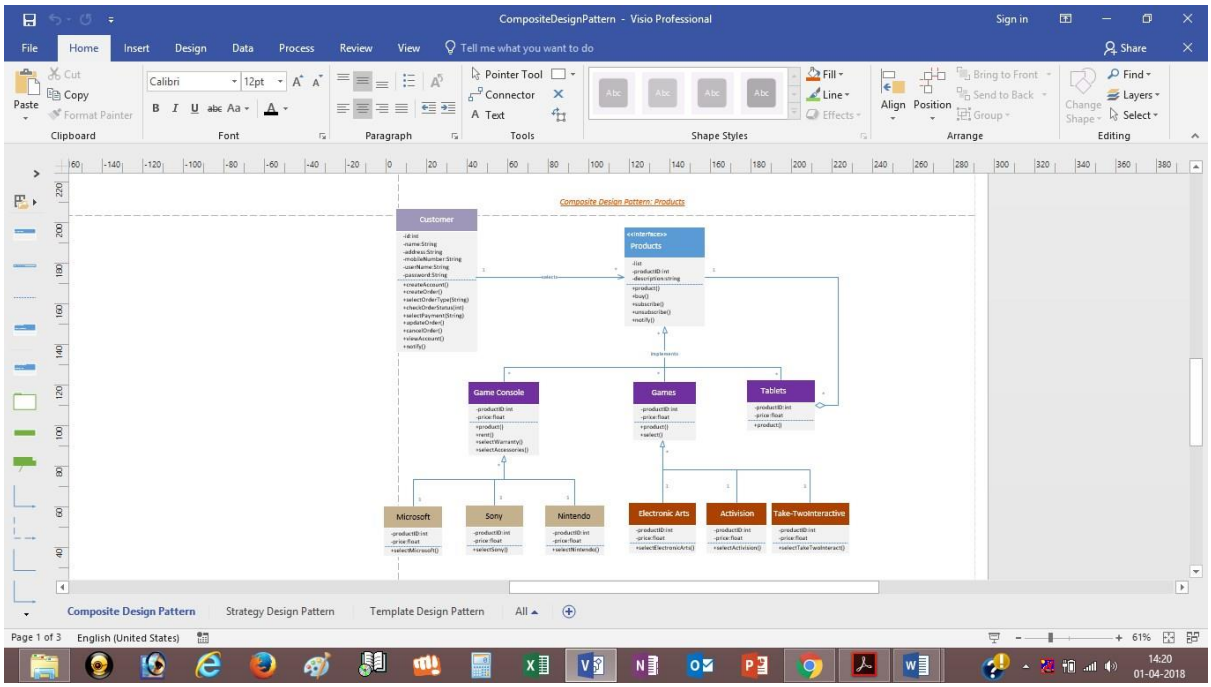
h. Screenshot of Abstract Factory Design Pattern



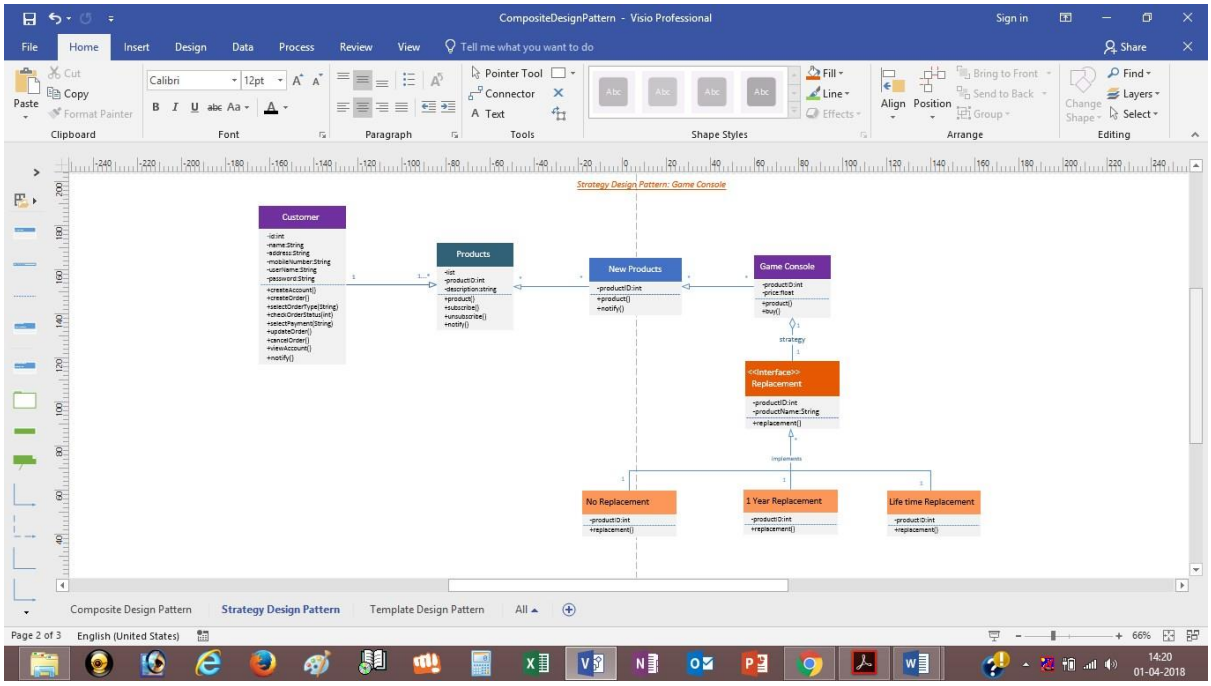
i. Screenshot of Observer Pattern



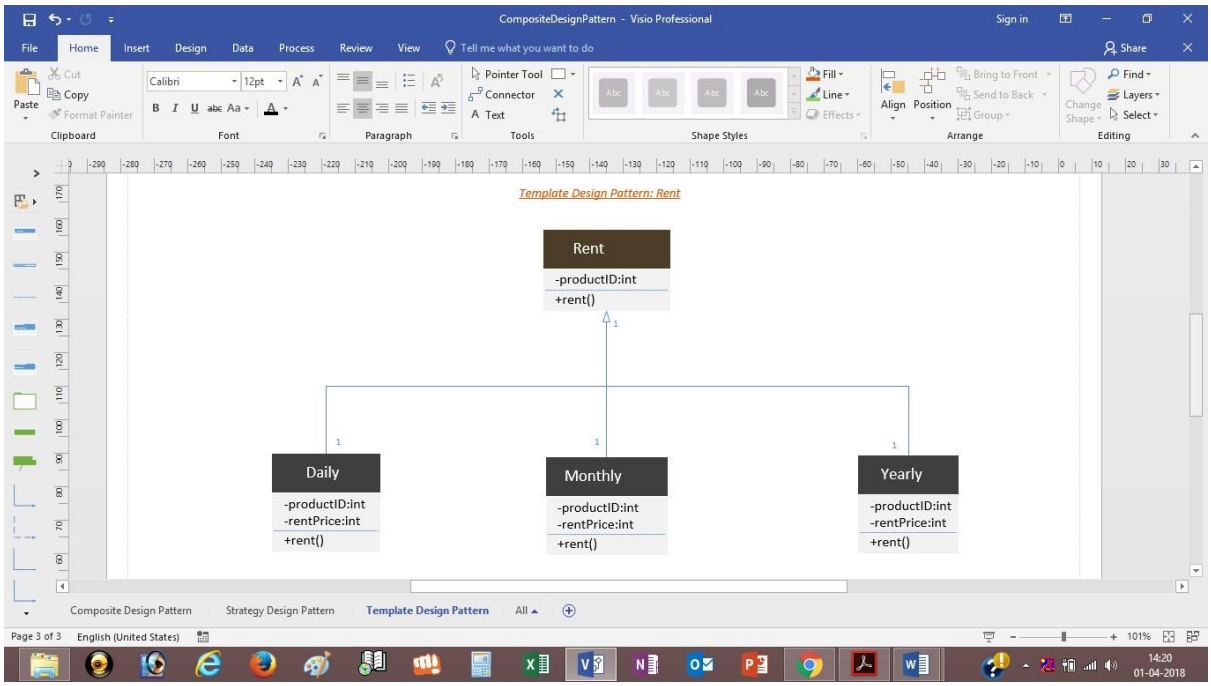
j. Screenshot of Composite Design Pattern



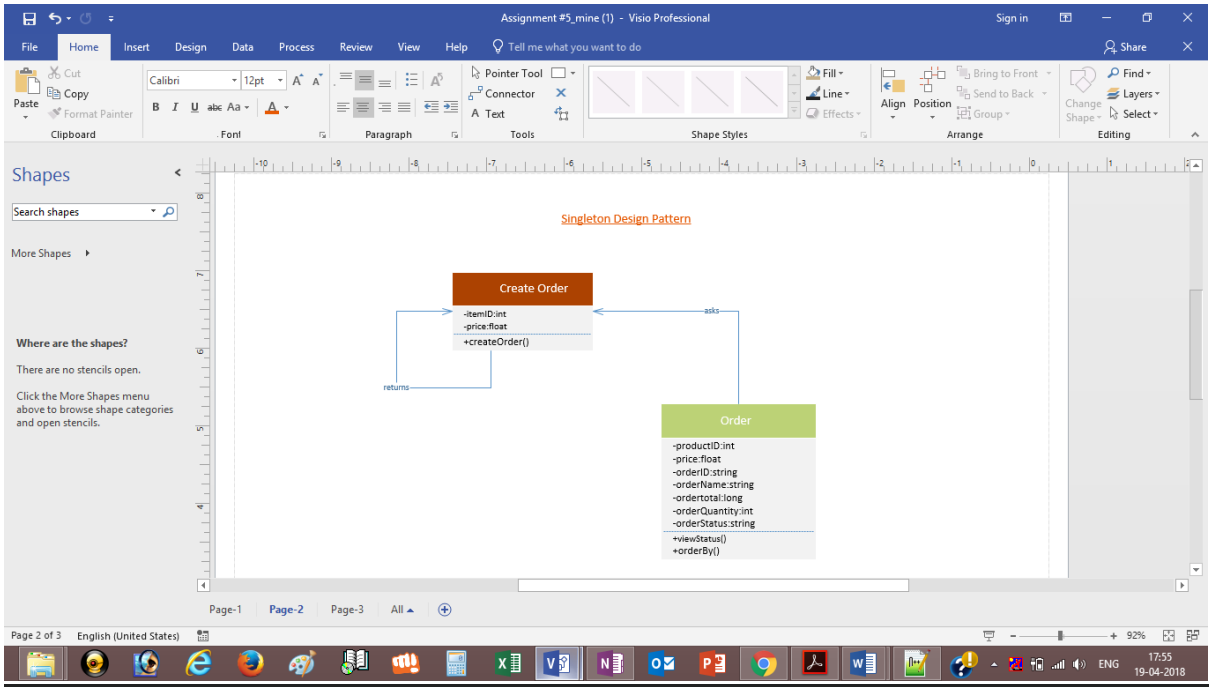
k. Screenshot of Strategy Design Pattern



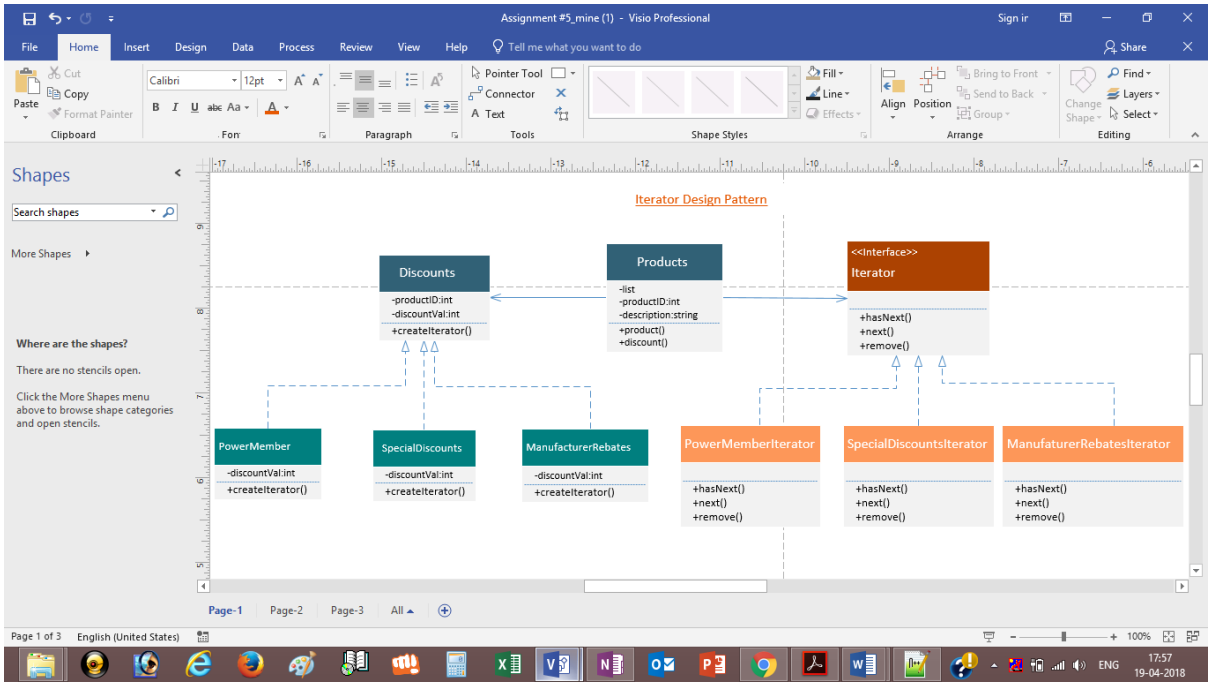
l. Screenshot of Template Design Pattern



m. Screenshot of Singleton Pattern



n. Screenshot of Iterator Pattern



o. Screenshot of MVC architectural pattern