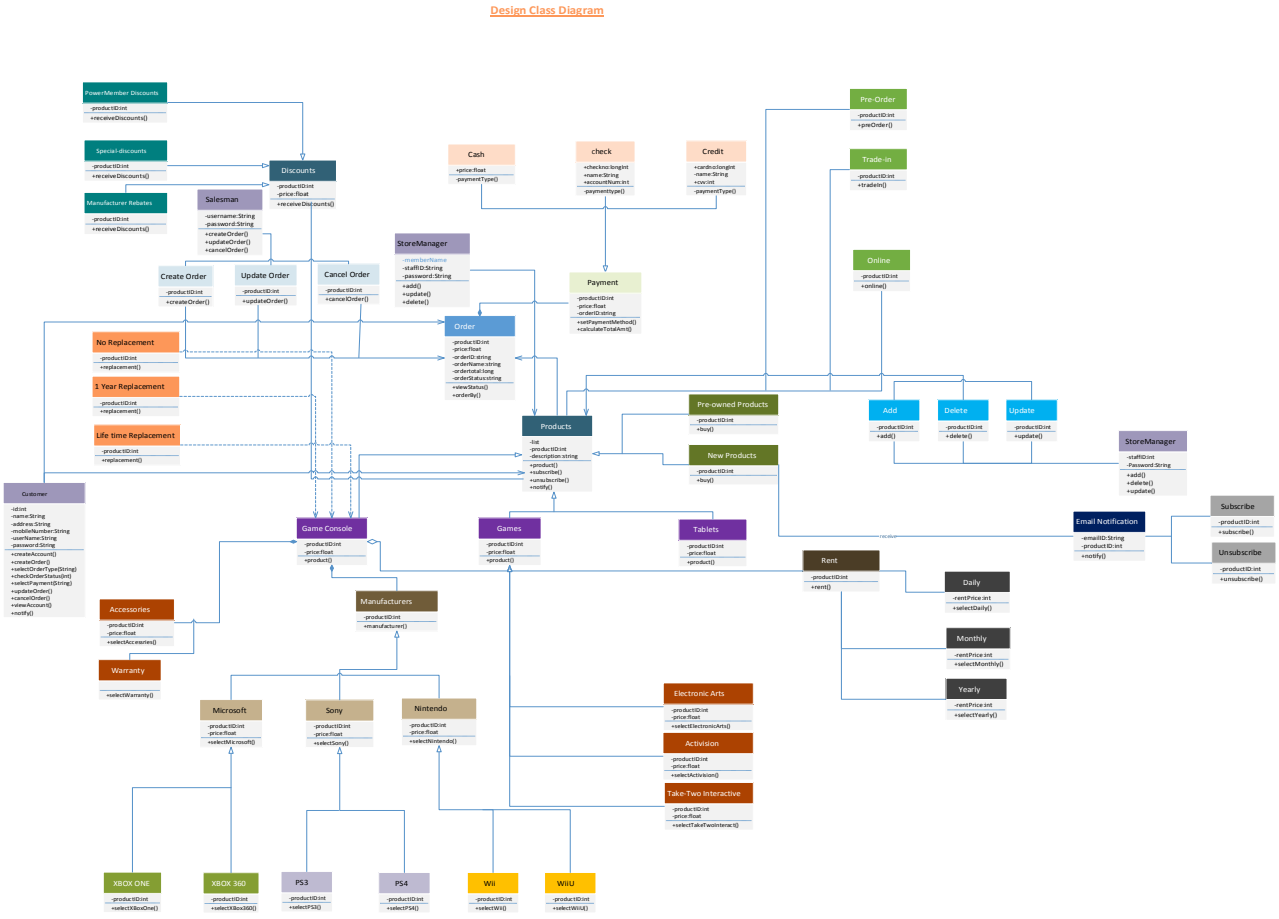
ASSIGNMENT #4

RITIKA KUMARI

A20414073

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). Composite Design Pattern:

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 behavioral 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). <u>Template Method Design Pattern:</u>

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

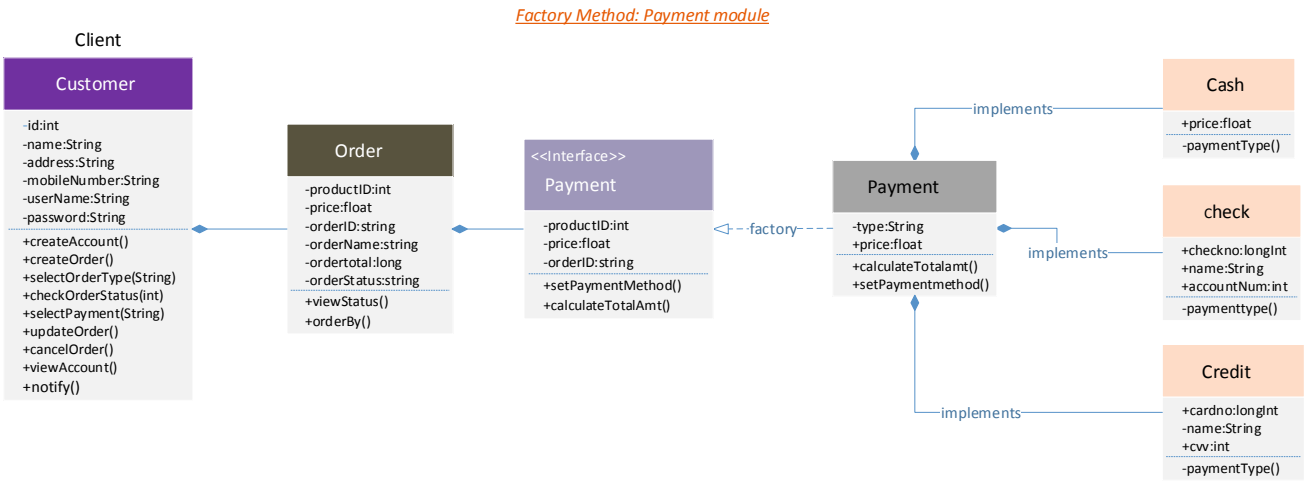
This pattern has two main parts, and typically uses object-oriented programming:

- 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:

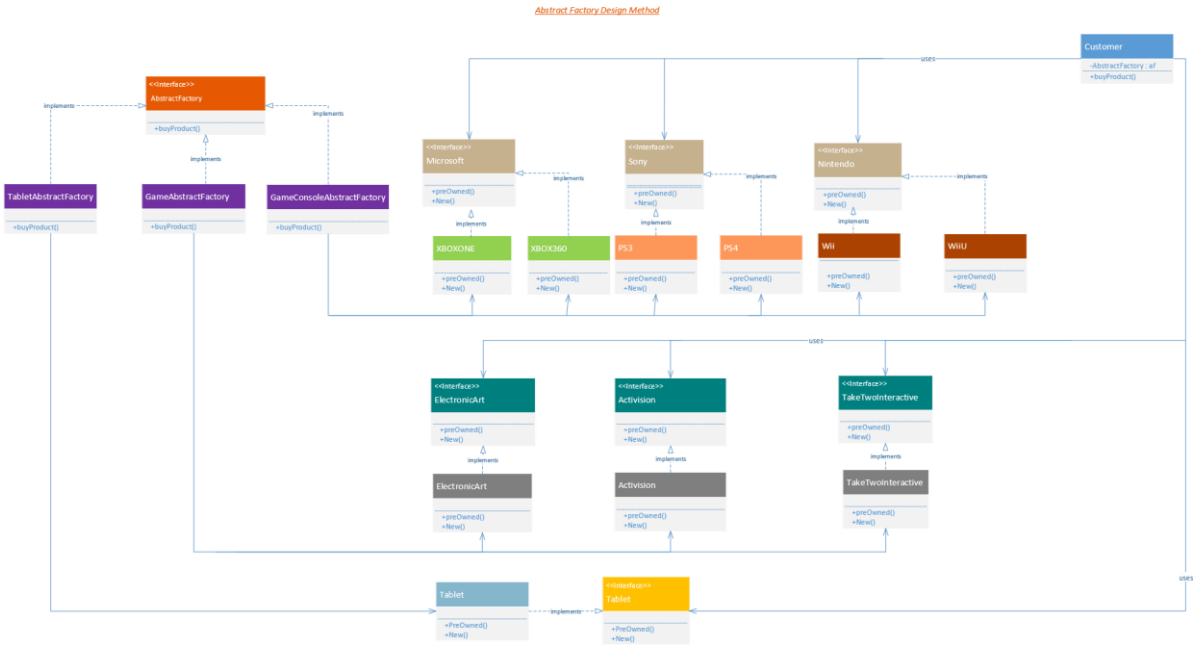
a. Factory Method Design Pattern:



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.

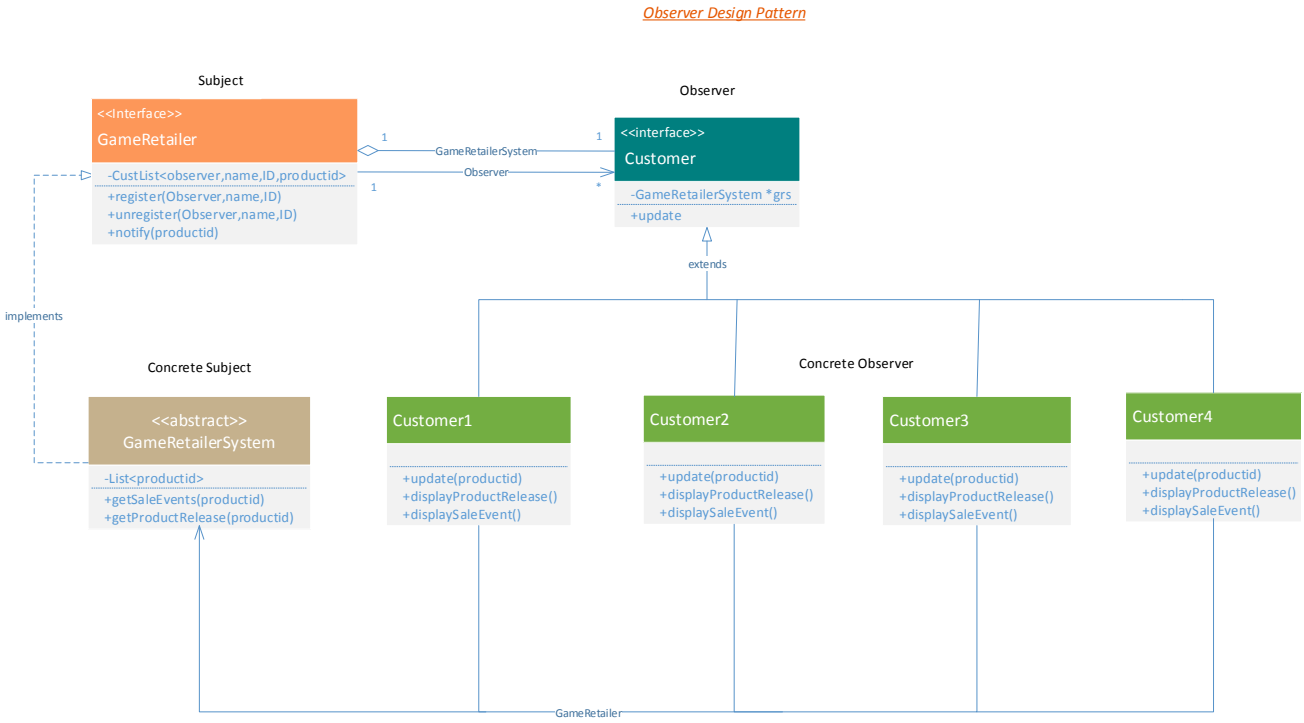
b. Abstract Factory Design Pattern:



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:



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.

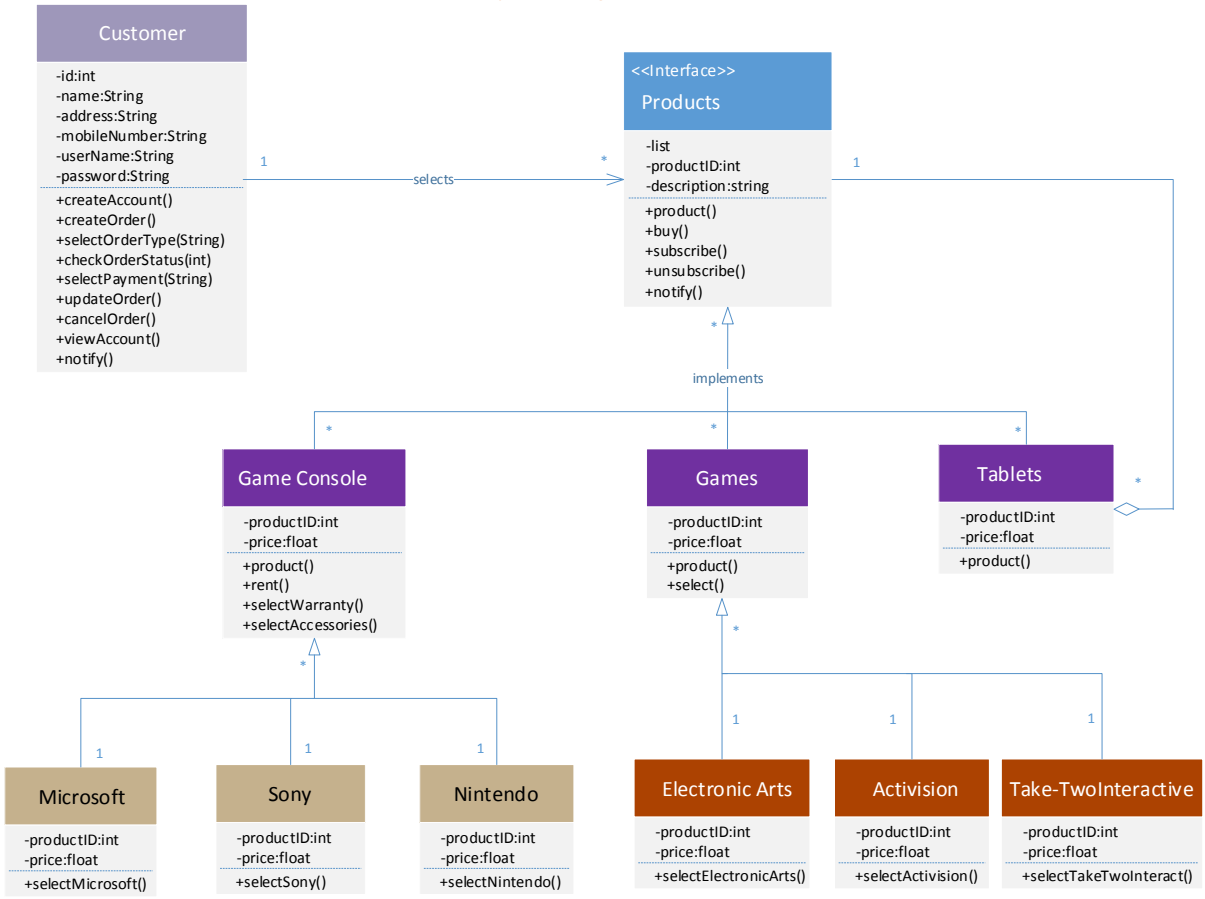
Here, this design pattern is used in the Customer Class. Customer can subscribe to email notifications for the game. This pattern generally represents a one to many relationships between observer and the many objects, i.e. the Customer and its subclasses. Also, the customer here can either choose to 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



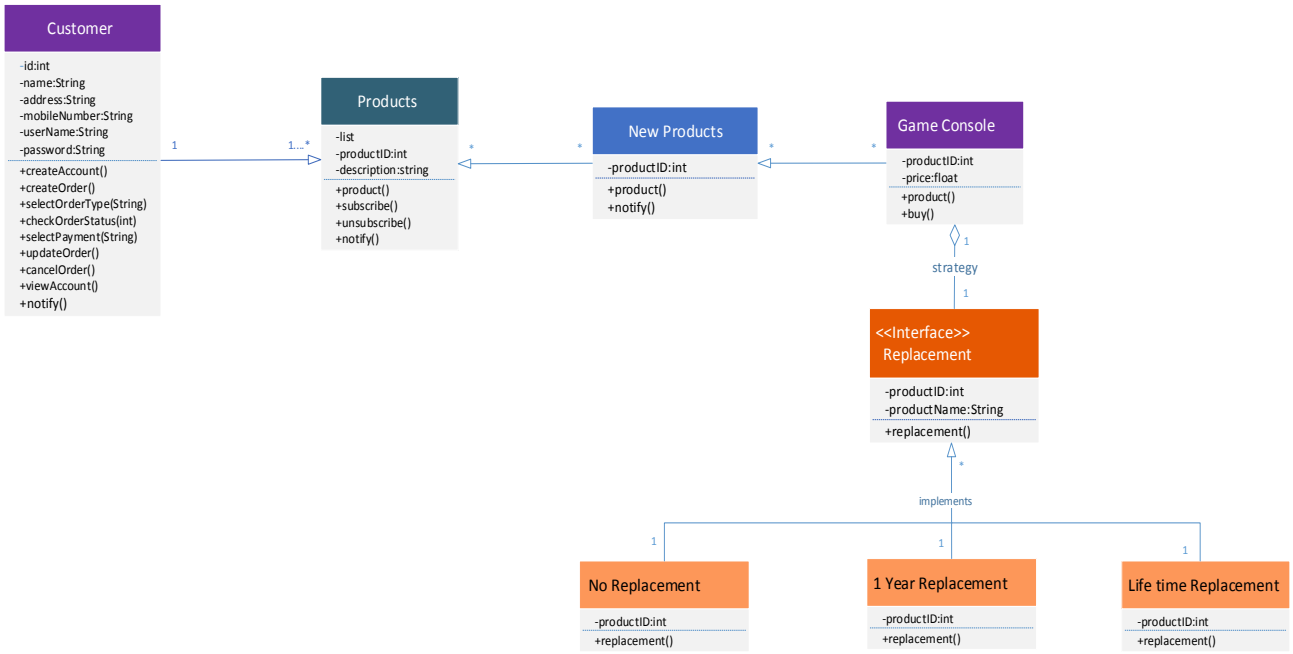
Here, Composite design pattern is used in the Products Class. It consists of three subclasses: GameConsole, Games and Tablets. Where GameConsole has three product types as Microsoft, Sony, and Nintendo. This design pattern is generally used, when a hierarchy is used during the coding process. Using this 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

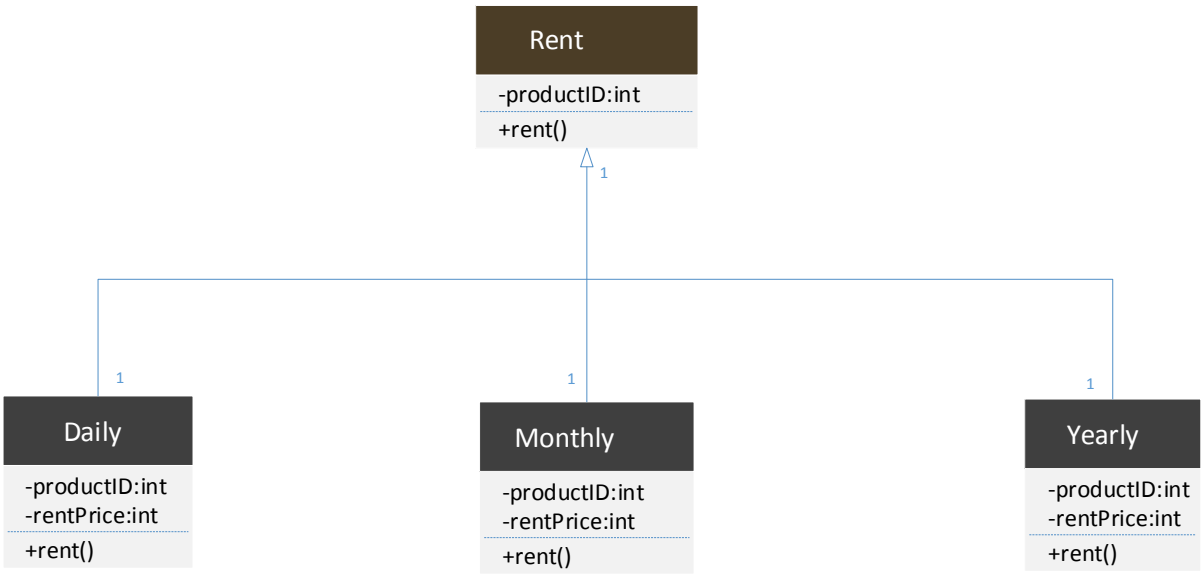


Here Customer can choose any of the replacement services which help customer to replace the products based on their choice. I have used strategy pattern to show the proper strategy for choosing the replacement options which includes the replacement strategy as per customer need.
A Strategy defines a 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 behavior or its algorithm can be changed at run time. This type of design pattern comes under behavior pattern. We create objects which represent various strategies and a context object whose behavior 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

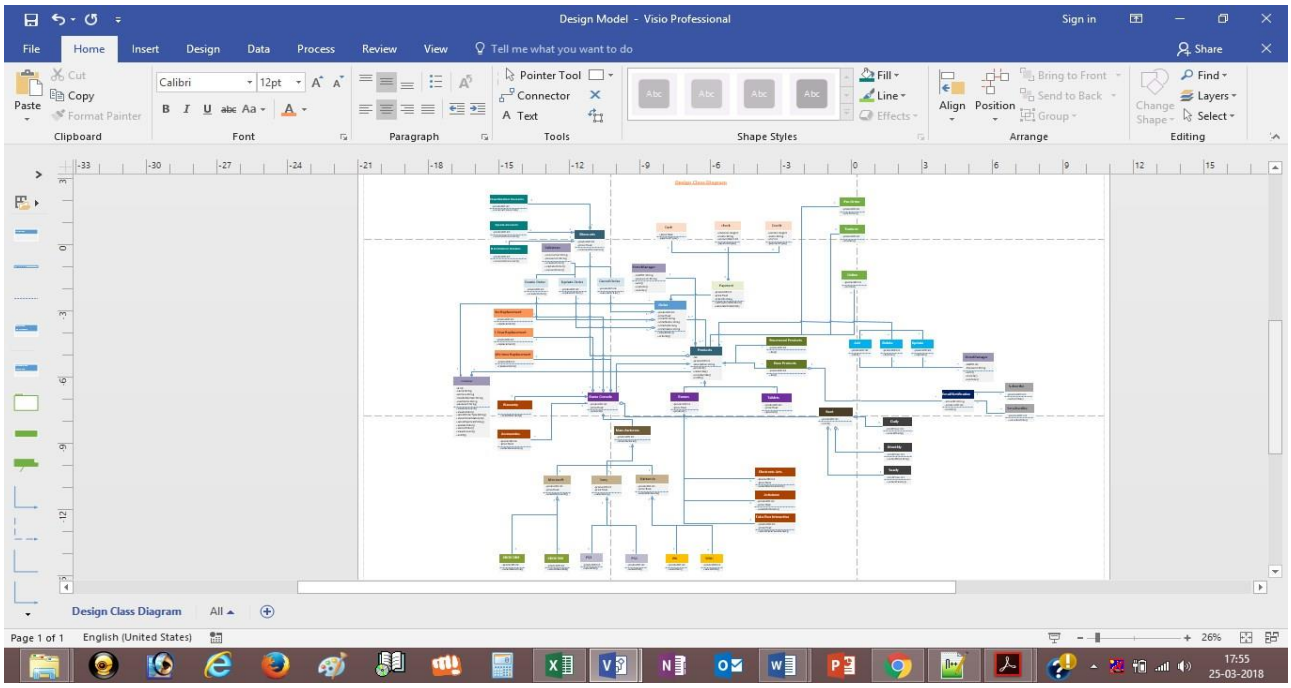


In Template pattern, an abstract class exposes defined template to execute its methods. Its subclasses can override the method implementation as per need, but the invocation is to be in the same way as defined by an abstract class. This pattern comes under behavior pattern category.

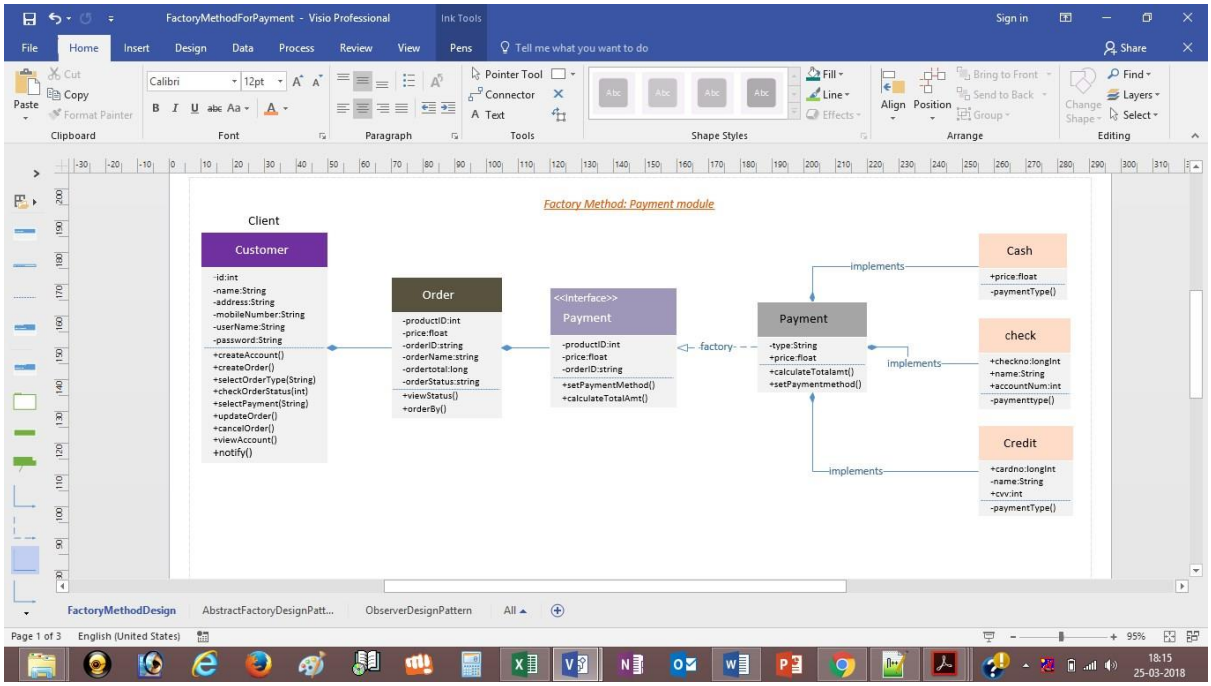
In Rent, the customer can get three types of rents- Daily, Monthly, Yearly plans. Here, Rent is the abstract class for Daily, Monthly and Yearly plans.

4. <u>Capture design model class diagram(s) (using shft+PrtScr) and save it/them as image(s) in the PDF file that you are submitting as the solution for this homework.</u>

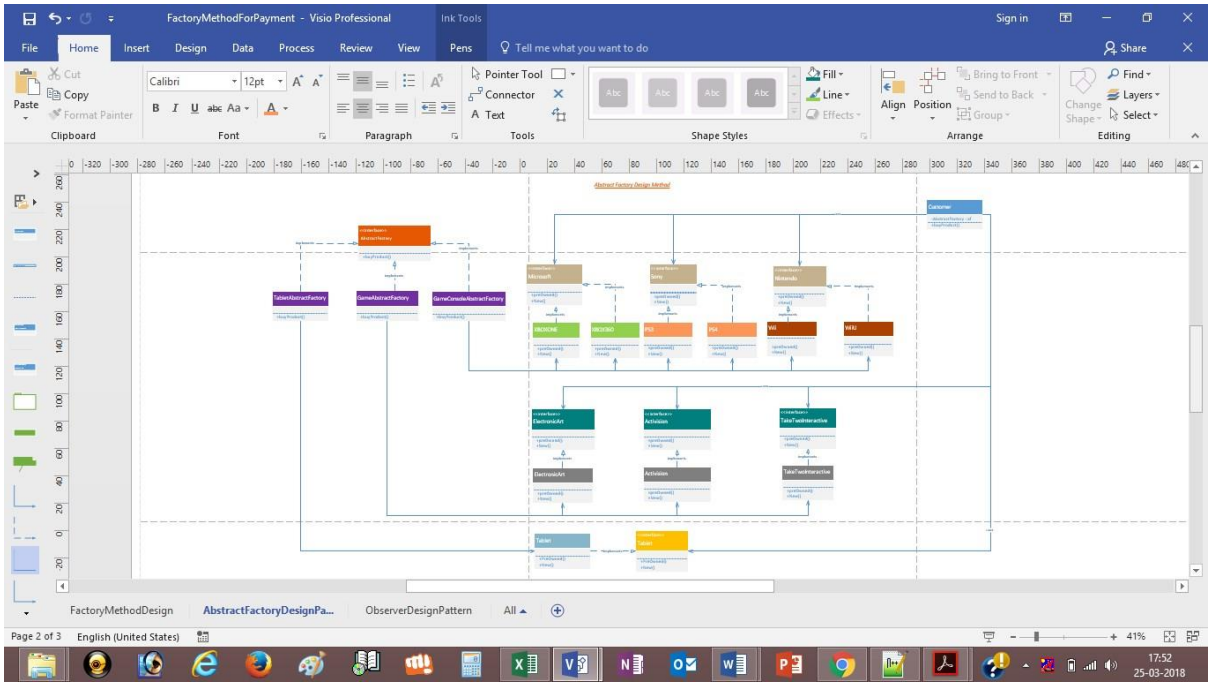
a. Screenshot of Design Class Diagram



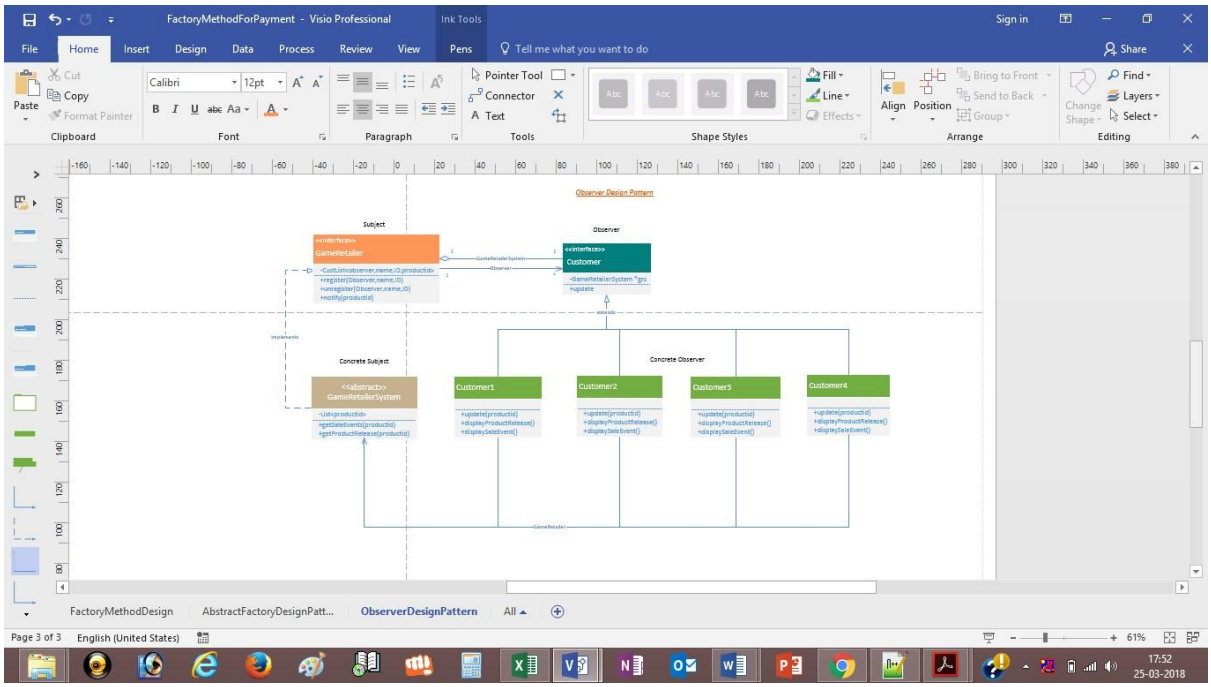
b. Screenshot of Factory Method Design Pattern



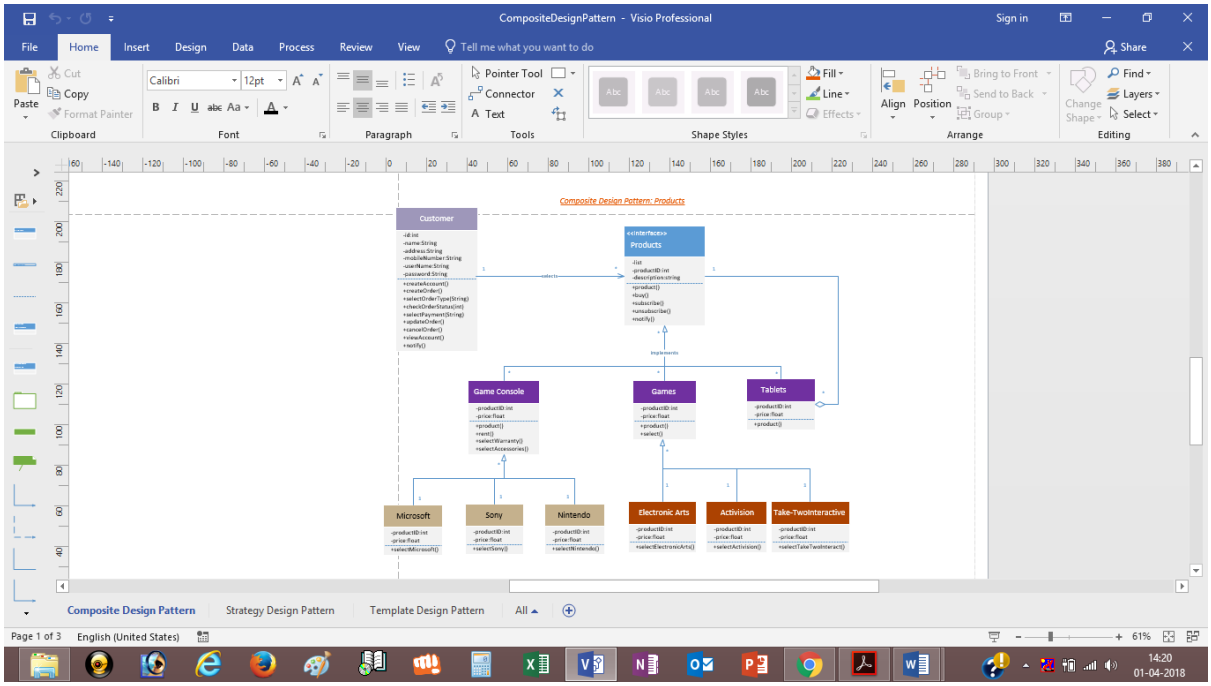
c. Screenshot of Abstract Factory Design Pattern



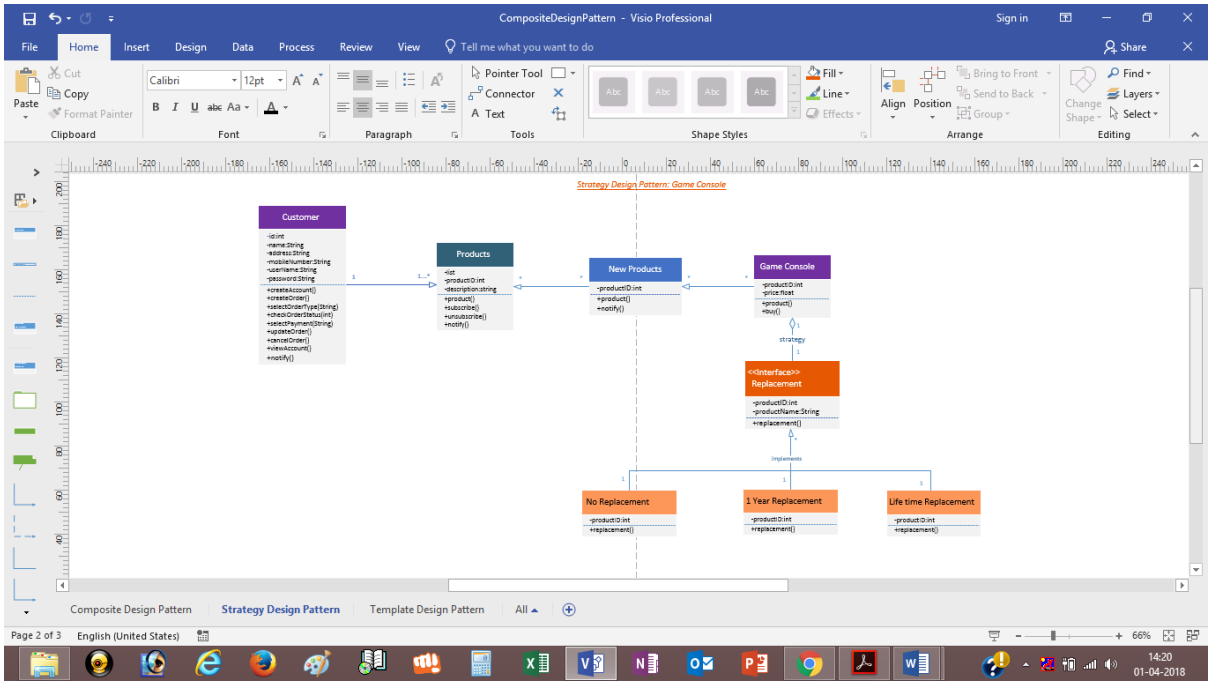
d. Screenshot of Observer Pattern



e. Screenshot of Composite Design Pattern



f. Screenshot of Strategy Design Pattern



g. Screenshot of Template Design Pattern