ASSIGNMENT 1 FRONT SHEET

Qualification	**BTEC Level 5 HND Diploma in Computing**		
Unit number and title	Unit 20: Advanced Programming		
Submission date	24/05/2021	**Date Received 1st submission**	
Re-submission Date		**Date Received 2nd submission**	
Student Name	Do Duc Tai	**Student ID**	GCH190834
Class	GCH0804	**Assessor name**	Doan Trung Tung

Student declaration

I certify that the assignment submission is entirely my own work and I fully understand the consequences of plagiarism. I understand that making a false declaration is a form of malpractice.

Student's signature	Tai

Grading grid

P1	P2	M1	M2	D1	D2

☼ Summative Feedback:		☼ Resubmission Feedback:
Grade:	Assessor Signature:	Date:
Lecturer Signature:		

Contents

Table of figures

1. Introduction

The author group is assigned to work on the basic concepts of OOP, in the report, the group will introduce some OOP concepts, describe the features of OOP, and provide concrete examples through a scenario, including use case diagrams and class diagrams. After that, we'll discuss and define a variety of design patterns, with specific examples for each situation and where they should be applied.

2. OOP general concepts

Object-oriented programming combines a group of variables (properties) and functions (methods) into a unit called an "object." These objects are organized into classes where individual objects can be grouped together. OOP can help you consider objects in a program's code and the different actions that could happen in relation to the objects. (Indeed, 2021)

This programming style widely exists in commonly used programming languages like Java, C++ and PHP. These languages help simplify the structure and organization of software programs. Programmers often use OOP when they need to create complex programs. (Indeed, 2021)

Constructor

A constructor is a special method of a class to create and initialize an object of that class. A constructor allows you to provide any custom initialization that must be performed before any other methods can be called on an instantiated object. (Mozilla, 2021).

Object

An object is an entity or instance of a class. The objects are mostly the physical entity but it can be a logical entity as well. Each object has state and behaviors. (Indeed, 2021)

Methods

Methods are functions defined inside the body of a class. They are used to perform operations with the attributes of our objects. Methods bring *modularity* to our programs. (Indeed, 2021)

Class

The class is a model or blueprint or prototype of an object that defines or specifies all the properties of the objects. Classes have the data and its associated function wrapped in it. The class defines the state and behaviors of an object. (Indeed, 2021)

What are the four basics of object-oriented programming?

Object-oriented programming has four basic concepts: encapsulation, abstraction, inheritance and polymorphism. Even if these concepts seem incredibly complex, understanding the general framework of how they work will help you understand the basics of a computer program. Here are the four basic theories and what they entail:

Encapsulation

The different objects inside of each program will try to communicate with each other automatically. If a programmer wants to stop objects from interacting with each other, they need to be encapsulated in individual classes. Through the process of encapsulation, classes cannot change or interact with the specific variables and functions of an object. (Indeed, 2021)

Just like a pill "encapsulates" or contains the medication inside of its coating, the principle of encapsulation works in a digital way to form a protective barrier around the information that separates it from the rest of the code. Programmers can replicate this object throughout different parts of the program or other programs. (Indeed, 2021)

Abstraction

Abstraction is like an extension of encapsulation because it hides certain properties and methods from the outside code to make the interface of the objects simpler. Programmers use abstraction for several beneficial reasons. Overall, abstraction helps isolate the impact of changes made to the code so that if something goes wrong, the change will only affect the variables shown and not the outside code. (Indeed, 2021)

Inheritance

Using this concept, programmers can extend the functionality of the code's existing classes to eliminate repetitive code. For instance, elements of HTML code that include a text box, select field and checkbox have certain properties in common with specific methods. (Indeed, 2021)

Instead of redefining the properties and methods for every type of HTML element, you can define them once in a generic object. Naming that object something like "HTMLElement" will cause other objects to inherit its properties and methods so you can reduce unnecessary code. (Indeed, 2021)

The main object is the superclass and all objects that follow it are subclasses. Subclasses can have separate elements while adding what they need from the superclass. (Indeed, 2021)

Polymorphism

This technique meaning "many forms or shapes" allows programmers to render multiple HTML elements depending on the type of object. This concept allows programmers to redefine the way something works by changing how it is done or by changing the parts in which it is done. Terms of polymorphism are called overriding and overloading. (Indeed, 2021)

3. OOP scenario

3.1 Scenario

Vietnam is on the verge of strong development, along with that the need for people movement also increases. We are a group of young startups who plan to open a Car showroom for selling renting car. In order for our store to be developed in the best way, we are planning to build up a management car system for our showroom. Building a car showroom management system including product information management, customer information management, product rental and return management:

- Users want to access the system to perform functions must first register an account, the account information will contain name, phone number, email address, address, password and role (there are 3 types of accounts: admin, customers and seller.)
- Users must log in before do system.
- The showroom needs sellers, sellers will have access to customer information (view information, modify customer information if it is wrong, and manage car sales and rentals in the showroom.
- The showroom needs an administrator, who manages both seller and customer accounts, and has access to modify vehicle information.
- Customers must pay first and then print the invoice to make the order.

- Sellers will manage the rental and return of the car, if the customer returns the car later than the deadline, an additional fee will be charged.

3.2. Use case Diagram

Building a general use case chart for the showroom management system:

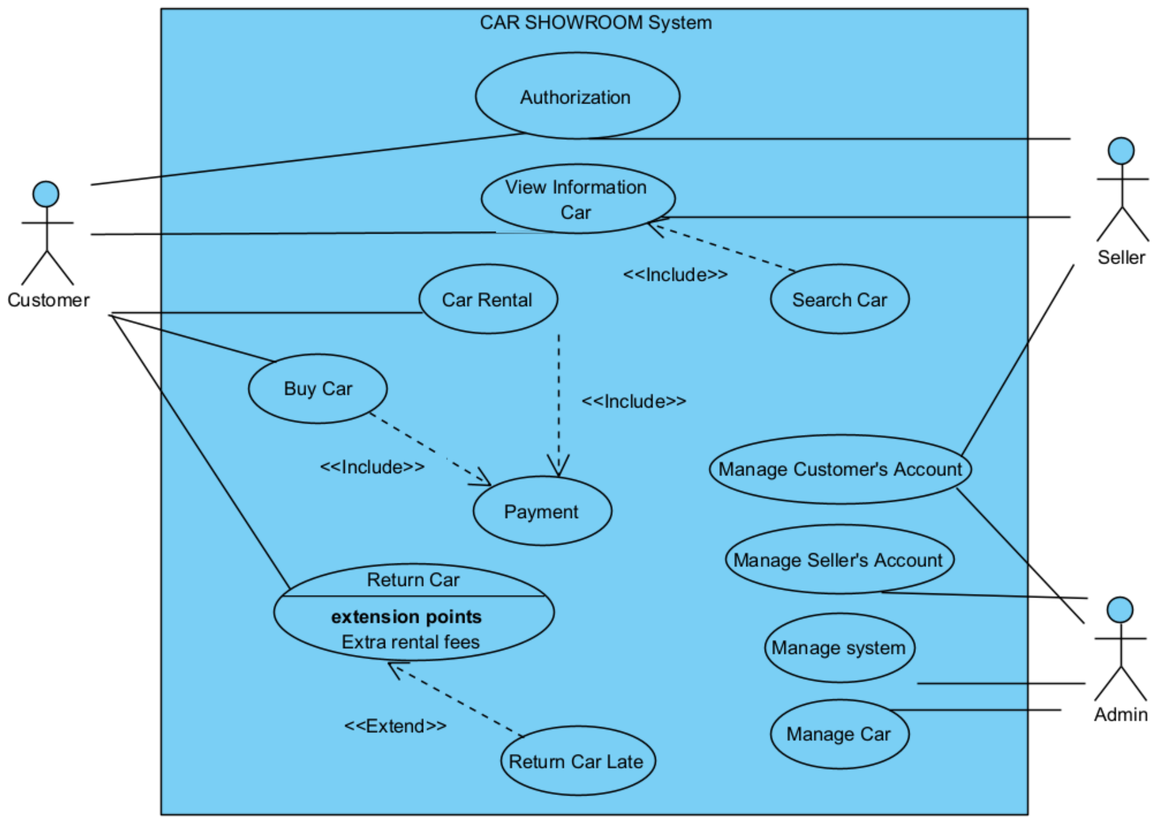

Figure 1-Use Case Diagram

Explanation:

- The main components in the use case diagram:

- System Boundary: Car Showroom System.
- Actors: Customer, Seller, Admin
- Relationships: Association, Include, Extend.
- Use cases.

- From the main Actors determine the general Use Cases:

- Admin: Manage Customer’s Account, Manage Seller’s Account, Manage Car, Manage System.
- Customer: Authorization, View Information Car, Buy Car, Car Rental, Return Car.
- Seller: View Information Car, Manage Customer’s Account, Confirm payment and make order.

- Relationships between use case – use case:

- Include: Include means a must-have relationship between use cases. In the diagram, the include relationship is concatenated from the use cases "Buy Car" and "Car Rental" to "Payment". Obviously, to buy a car or rent a car, you have to pay, so it is reasonable to put the include relationship here. After the customer has paid, the seller will confirm the payment and print the invoice and make the order. Similarly, to be able to see information about car products, users must "Search" first.
- Extend: Extend means an extended relationship between use cases. If include is a required relationship, then extend represents an optional relationship. In the diagram, "Return Car Late" is a use case that has an Extend relationship with "Return Car". That is, "Return Car Late" is just a use case that may or may not happen. There is one thing that comes with the Extend is the Extension Point, it is the condition where the Extend use case occurs, I set it to "Over-due". That means the customer is overdue to return the car => Extend => "Return Car Late". Customer returns the car on time => No Extend.

-Use Case Specification:

Name of Use Case:	Authorization
Description:	As a user, I want to login to the app to use services from the system.
Actors:	Customer, Seller, Admin.
Preconditions:	1. User accounts already created. 2. User account has been authorized.

	3. The user's device is already connected to the internet when logging in.
Postconditions:	1. User successfully logged into the application 2. The system records successful logging in Activity Log.
Flow:	1. User enter a password 2. The system validates the credentials successfully and allows the user to access the application 3. The system records successful logging in Activity Log.
Alternative Flows:	No Alternative Flows.
Exceptions:	No exception.

Name of Use Case:	View Information Car
Description:	Show all product information.
Actors:	Customer, Seller.
Preconditions:	1. Login successful. 2. Search Car.
Postconditions:	1. Show all product information.
Flow:	1. User enter the keyword about product they want to view. 2. Received keyword processing system. 3. Show product follow keyword.

Alternative Flows:	1. The keyword entered by the user is not in the system.
Exceptions:	No exception.

Name of Use Case:	Buy Car, Rental Car
Description:	After viewing the product, customers can choose to buy or rent a car.
Actors:	Customer.
Preconditions:	1. Show all product information. 2. Click Buy or Rental Car.
Postconditions:	1. Show interface payment.
Flow:	1. User presses rent or buy. 2. Select a payment method. 3. Check balance. 4. Transfer information to Seller.
Alternative Flows:	1. The amount in the account is not enough.
Exceptions:	No exception.

Name of Use Case:	Return Car

Description:	Customer's car return activity
Actors:	Customer.
Preconditions:	Have a previous car loan contract
Postconditions:	Car return successful.
Flow:	1. Check the car loan term. 2. Check vehicle facilities. 3. Invoice incurred expenses. 4. Return the car successfully.
Alternative Flows:	No
Exceptions:	No exception.

Name of Use Case:	Confirm payment and make order
Description:	Finalize the product and hand it over to the customer.
Actors:	Seller.
Preconditions:	1. Customer pay successful.
Postconditions:	1. Make order successful.
Flow:	1.Make an invoice. 2.Check the bill.

	3.Print invoice. 4.Car handover.
Alternative Flows:	No
Exceptions:	No exception.

Name of Use Case:	Manage Customer's Account, Seller's Account, Car, System
Description:	Administrator has the ability to run the entire system.
Actors:	Admin.
Preconditions:	1. Login successful.
Postconditions:	1. Show interface of Admin.
Flow:	Admin can edit, update, add, remove information of all.
Alternative Flows:	No.
Exceptions:	No exception.

3.3 Class Diagram and Explanation

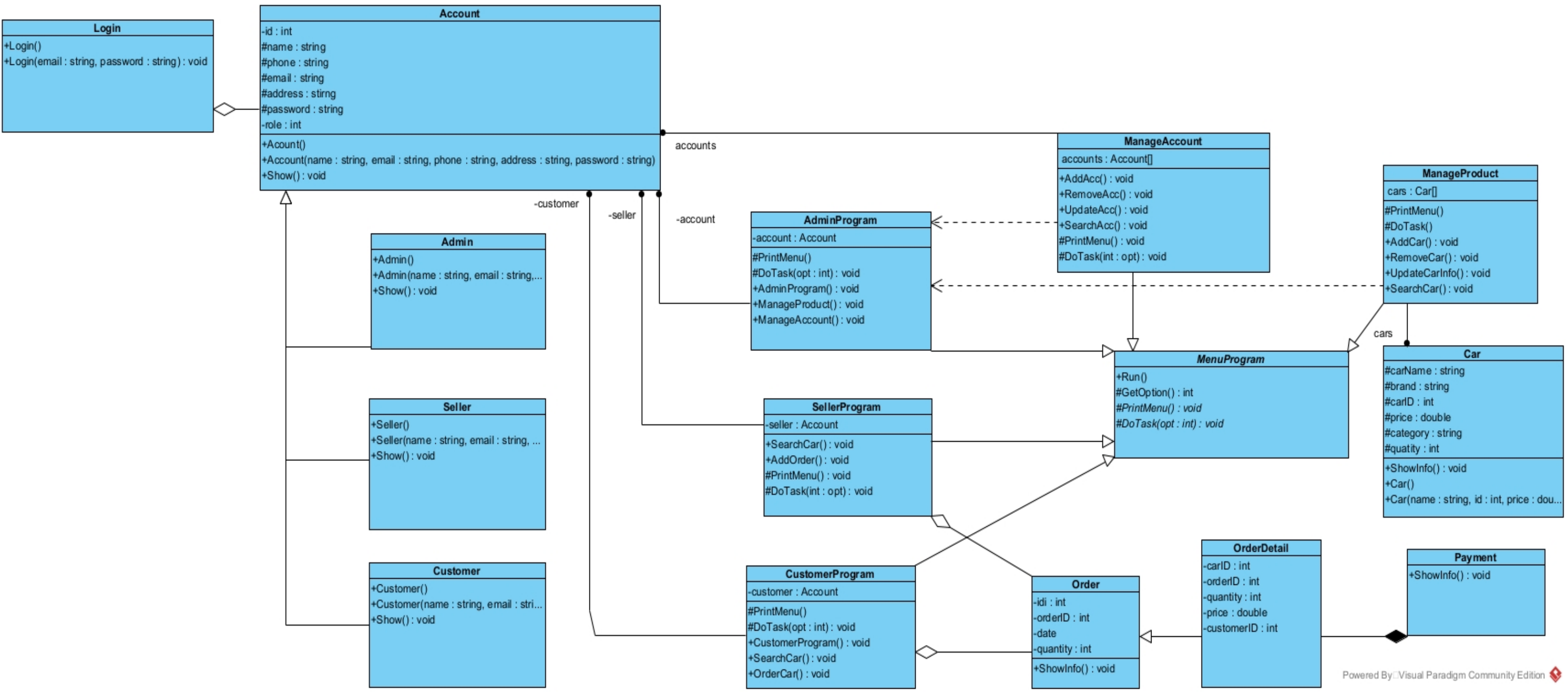

Figure 2- Class Diagram

Explanation:

Constructor: The constructor, formally, always has the same name as the class and has no output type. Parameter lists and function bodies are similar to member methods. For example, in Car class, default constructor is Car() with no parameters, the custom constructor is Car(name, id, price…) with parameters.

Class: A class is a data type consisting of predefined properties and methods. For example, Account is an class including properties such as: email, password, name… It has a method is Show(), a default constructor and custom constructor Account() and Account(name, email, address, password)

Method: Method/function is action that the object can perform. For example, AdminProgram class has many method such as PrintMenu(), DoTask(), ManageProduct(), ManageAccount() these methods can do several actions depends on how developer want to use them. AdminProgram() is a constructor a special method of a class for creating and initializing an object of that class.

Inheritance: It allows to build a new class based on existing class definitions. This means that the parent class can share data and methods with the child classes. Subclasses do not have to be redefined, in addition, they can extend inherited components and add new ones. For example: the parent class is MenuProgram and child classes (subclasses) are AdminProgram, SellerProgram, CustomerProgram… Each class represents a different type of user program but has the same properties such as PrintMenu(), GetpOption(), DoTask(). Instead of copying these properties, we should put them in a common class called the parent class. We can define a parent class – in this case Menu, and have subclasses that inherit from it, creating a parent/child relationship.

Encapsulation: Encapsulation is demonstrated when each object is private within a class and other objects cannot access this scope directly. Instead they can only call public scoped functions called methods. For example, in class AdminProgram, related data (accounts) and methods are encapsulated into the class for ease of management and use to perform a class-specific set of functions. In addition, packaging also hides some information and details of internal settings so that the outside cannot see, the list -accounts is set as private so user cannot access this private data from outside of that class, user can access the data only through public/protected method that give acceptation such as ManageAccount().

Polymorphism: Polymorphism is a concept where two or more classes have the same methods but can implement them in different ways. For example, in CustomerProgram and SellerProgram both have DoTask() method but int CustomerProgram class, it is used to search car and order car, while in SellerProgram, it is used to search car and add an order from customer requires.

Abstraction: Abstraction help removes the unnecessary complexity of the object and focuses only on what is essential and important. For example: ManageAccount is only interested in information such as: name, email, phone, address… without having to manage additional information such as: hair's color, hobbies, height… In CustomerProgram, the action is to search cars to order, the search function is to represent the abstraction. Users only need to know that when search the cars, the system found them, no matter how it works inside the function.

4. Design Patterns

In software engineering, a design pattern is a general repeatable solution to a problem that often occurs in software design. A design pattern is not a complete design that can be converted directly into code. It is a description or a pattern of how to solve a problem that can be used in many different situations (Gamma, 1994).

Programmers can apply this solution to solve similar problems. The problems programmer faces can be solved by themselves, but it is not necessarily optimal (Gamma, 1994).

Design pattern is not language specific. Design patterns are possible in most programming languages. It helps to solve problems in the most optimal way, providing with solutions in OOP (Gamma, 1994).

4.1 Creation Design Pattern.

4.1.1 Introduction Creation Design Pattern.

In software engineering, creation design patterns are design patterns that deal with object creation mechanisms, trying to create an object in a way that is appropriate for the situation. The basic form of object creation can lead to design problems or add complexity to the design. Creative design patterns that solve this problem somehow control the creation of this object.

Creational patterns provide various object creation mechanisms, which increase flexibility and reuse of existing code.

- **Abstract Factory** is a creational design pattern that provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created.
- **Builder** is a creational design pattern that lets you construct complex objects step by step. The pattern allows you to produce different types and representations of an object using the same construction code.
- **Factory Method** is a creational design pattern that provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created.
- **Prototype** is a creational design pattern that lets you copy existing objects without making your code dependent on their classes.
- **Singleton** is a creational design pattern that lets you ensure that a class has only one instance, while providing a global access point to this instance.

4.1.2. Description of Creation Scenario (Singleton Pattern)

Scenario: A showroom system has come to the final steps to complete. The manager wants to use a system to manage all information of categories and products (data). So the showroom wants to hire author to build a system to manage all this information and it is just for the manager to use. The system must have some specific functions such as: Add, show, search product and category. In this case, the manager wants to only one object manager who control the system and each time the manager had any changes in the system he/she can access the same data from the beginning time he/she uses the program without re-initialize many objects for many behaviors leading to re-create many same data. It may cost many resources and makes the system slower.

4.1.3. Diagram and Explanation

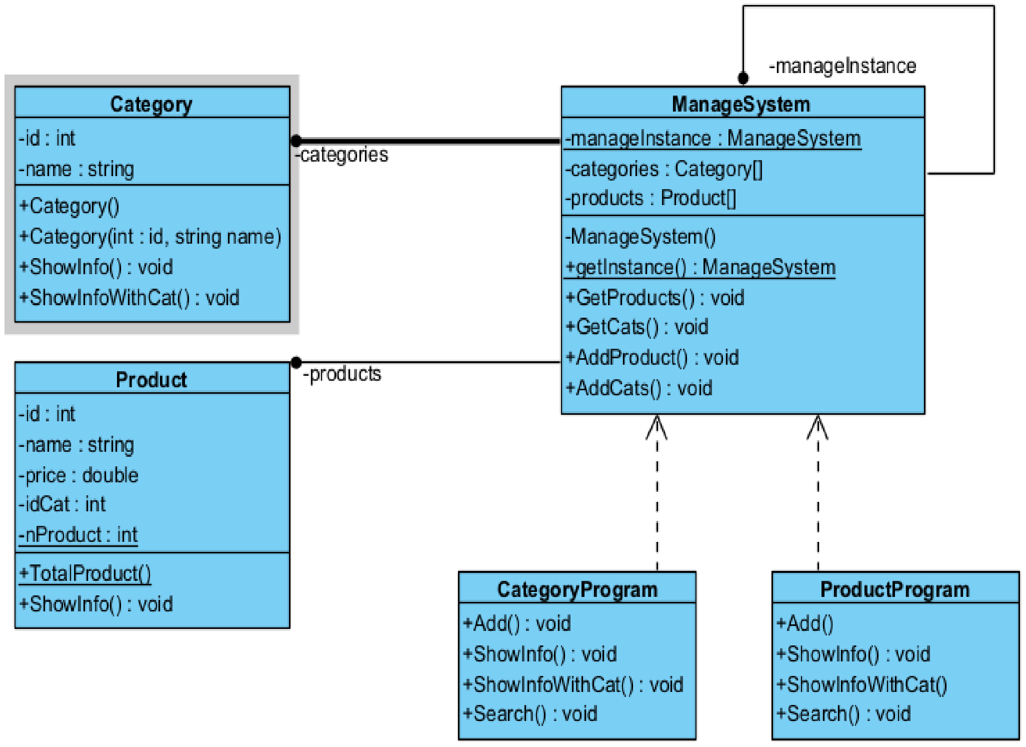

Figure 3-Singleton Design Pattern

Explanation:

According to showroom's manager's requirement, author had used Singleton Design Pattern. Because in whole system, user only uses one single object (instance) to implement many methods. In this way, user can use the instance many time in many class method and no matter how many time users use this instance, the objects always pointing the same instance that initialized from the beginning and they are always have the same value instance.

Normally, each time user wants to change anything in the system they have to create an different object such as Add() method, every time manager wants to add many products he/she has to create many objects, they cannot access to the same Product array and each time an object was created this issues takes times and resources. Using Singleton Design Pattern, user only uses one object manager instance to do variety classes method and it can access with the same data Product array. When the manager access to application, the instance will be saved until the manager finished the job without initialize another new object leading to heavy and long-process system. Therefore, if user wants to use a single object and a single data then he/she can use Singleton Design Pattern. This Design Pattern makes user easier to manage shared data and makes the system be more secured.

Here is explanation about the diagram: First of all, create class ManageSystem which contains a private static manageInstance as an instance and a public static method getInstance() and a private constructor ManageSystem().

The constructor ManageSystem() must be initialized as private to prevent user create many object outside the class. To initialize the object, the method getInstance() is used to return the object manageInstance now is the a object of ManageSystem, the object is only initialized through this method and when run the program, it only calls ManageSystem() one time and all objects called from getInstance() are always the same one object pointing to one manageInstance because of static modifier.

Product and Category class are owned by ManageSystem class to get list of products and categories with GetProducts() and GetCats(). ProductProgram and CategoryProgram use the instance from ManageSystem to perform their function.

4.2. Structural Design Pattern.

4.2.1. Introduction Structural Design Pattern.

In software engineering, Structural Design Patterns are design patterns that ease the design by identifying a simple way to realize the relationship among entities.

This design pattern is all about class and object composition. Structural class creation patterns use inheritance to compose interfaces. The structural object patterns define ways to compose objects to obtain new functionality.

The structural patterns describe how objects and classes can be combined to form larger structures.

Examples of Structural Design Patterns:

- **Adapter Design Pattern** is a software design pattern that allows the interface of an existing class to be used as another interface. It is often used to make existing classes work with others without modifying their source code.
- **Facade Design Pattern** is a software-design pattern commonly used in object-oriented programming. Analogous to a facade in architecture, a facade is an object that serves as a front-facing interface masking more complex underlying or structural code.
- **Decorator Design Pattern** is a design pattern that allows behavior to be added to an individual object, dynamically, without affecting the behavior of other objects from the same class.
- **Bridge Design Pattern** is a design pattern used in software engineering that is meant to "decouple an abstraction from its implementation so that the two can vary independently.
- **Composite Design Pattern** is a partitioning design pattern. The composite pattern describes a group of objects that are 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.
- **Proxy Design Pattern** is a software design pattern. A proxy, in its most general form, is a class functioning as an interface to something else.
- **Flyweight Design Pattern** primarily used to reduce the number of objects created and to decrease memory footprint and increase performance. This type of design pattern comes under structural pattern as this pattern provides ways to decrease object count thus improving the object structure of application.

4.2.2. Description of Structural Scenario (Composite Pattern)

Scenario: showroom system has come to the final steps to complete. Features such as search, view, payment, buy car, return car, ... have all been completed. Until the test run, the developers encountered a problem, that when clicking on product view, all products were displayed, not in any order, looking messy. Immediately, everyone thought of organizing everything like the files in a computer.

4.2.3. Diagram and Explanation

The Composite Pattern allows to combine objects into tree structures to represent an entire hierarchy. Composite allows clients to treat individual objects and their components in a unified manner.

The Composite Pattern allows us to build structures of objects in the form of trees that contain both collections of objects and individual objects as nodes.

Using a composite structure, we can apply the same operations on both composite objects and individual objects. In other words, in most cases we can ignore the difference between composites of objects and individual objects.

To clarify for choosing the Composite Pattern to make the interface for the system. The author gives the following main reasons:

- It defines class hierarchies that contain primitive and complex objects : The showroom's auto products are currently in a jumbled arrangement, so the Composite pattern can stratify large objects such as automakers, car manufacturers, and vehicles. will contain individual accessories for each vehicle.

- It makes it easier for you to add new types of ingredients: When the seller wants to add products, brands, and accessories, it is very easy to add it to its correct menu, avoiding the case of adding a new element and then repeating it again. not found.

- It provides flexibility of structure with class or manageable interface: The interface is open, clear presentation, easy to use by customers and sellers, increasing work efficiency.

This is the tree structure that illustrates a small part of the showroom's products:

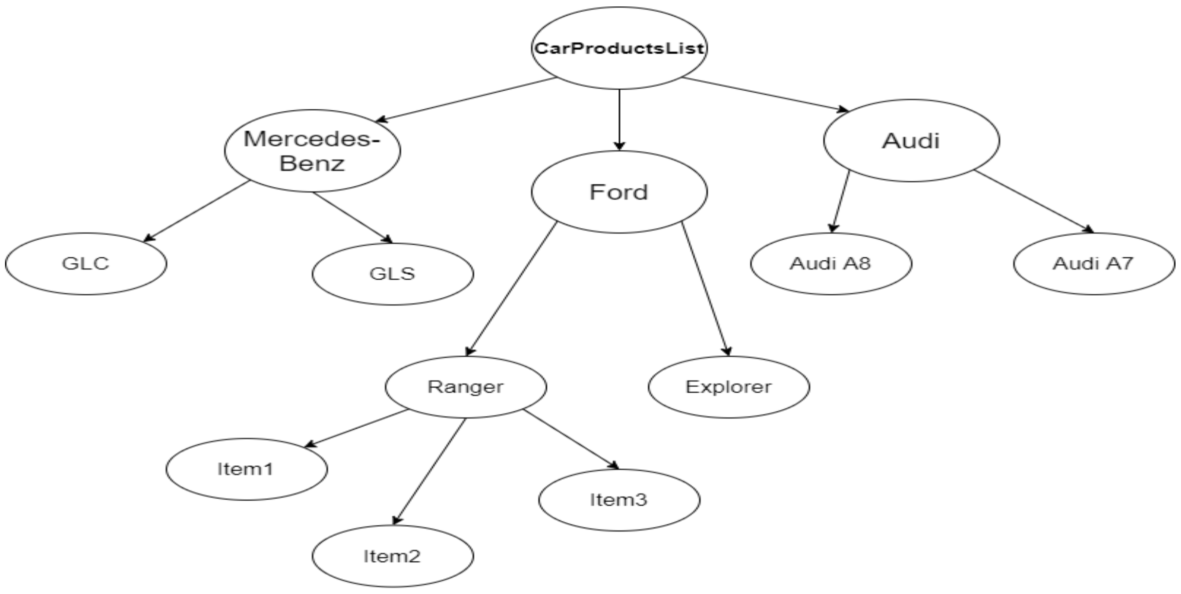

Figure 4-Tree structure of Composite Pattern

- The element containing the child element is called Node. As the showroom structure below we can see: CarProduct, Mercedes-Benz, Ford, Audi, Ranger.

- The element that does not contain child elements will be Leaf: GLC, GLS, Explorer, Item1, Item2...

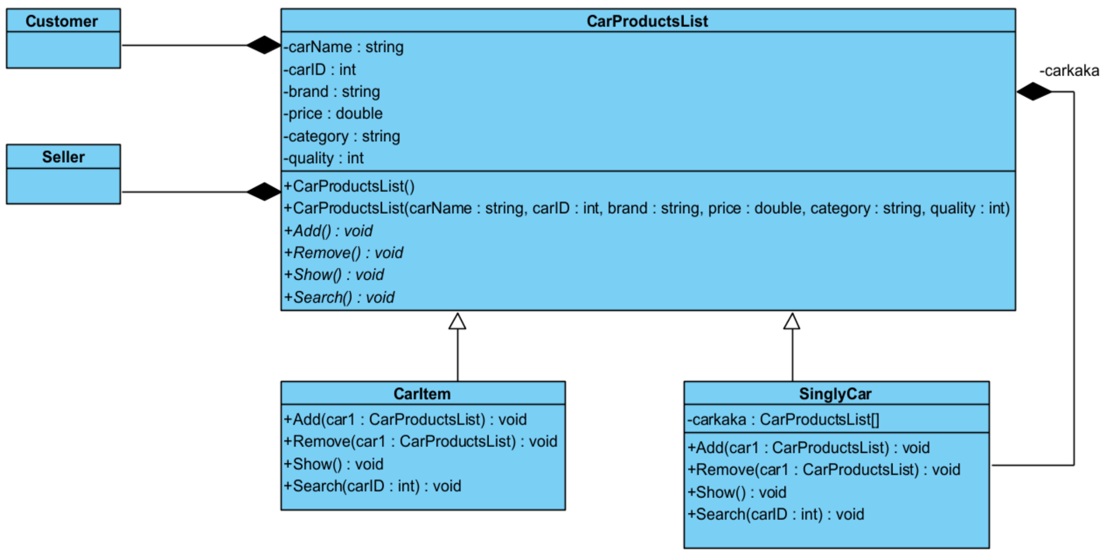
Customer
Seller
CarProductsList
-carName : string
-carID : int
-brand : string
-price : double
-category : string
-quality : int
+CarProductsList()
+CarProductsList(carName : string, carID : int, brand : string, price : double, category : string, quality : int)
+Add() : void
+Remove() : void
+Show() : void
+Search() : void
-carkaka
CarItem
+Add(car1 : CarProductsList) : void
+Remove(car1 : CarProductsList) : void
+Show() : void
+Search(carID : int) : void
SinglyCar
-carkaka : CarProductsList[]
+Add(car1 : CarProductsList) : void
+Remove(car1 : CarProductsList) : void
+Show() : void
+Search(carID : int) : void

Figure 5-Composite Pattern Class Diagram

- Customer and seller will use the CarProductsList interface to access both CarItem and SinglyCar.

- CarProductsList will represent the interface for both CarItem and SinglyCar. The author used an abstract class here because he wanted to provide default implementations for these methods.

- Some methods have added add(), remove(), show(), search(). These are methods to manipulate components, components here are CarItem and SinglyCar.

- SinglyCar also overrides add(), remove(), show(), search() methods to remove other SinglyCars in CarProductsList.

- CarItem will override only meaningful methods, will use the default implementations in CarProductsList.

4.2.4 Description of Structural Scenario (Adapter Pattern)

Scenario: VinGroup is a large corporation, in this group includes small corporations. To manually manage the revenue of subsidiaries is very difficult and complicated. Therefore, in order to manage the revenue of all subsidiaries, VinGroup desperately needs a software to be able to manage the revenue of the subsidiaries. Here, we will use the Adapter pattern to handle this.

4.2.5 Diagram and Explanation

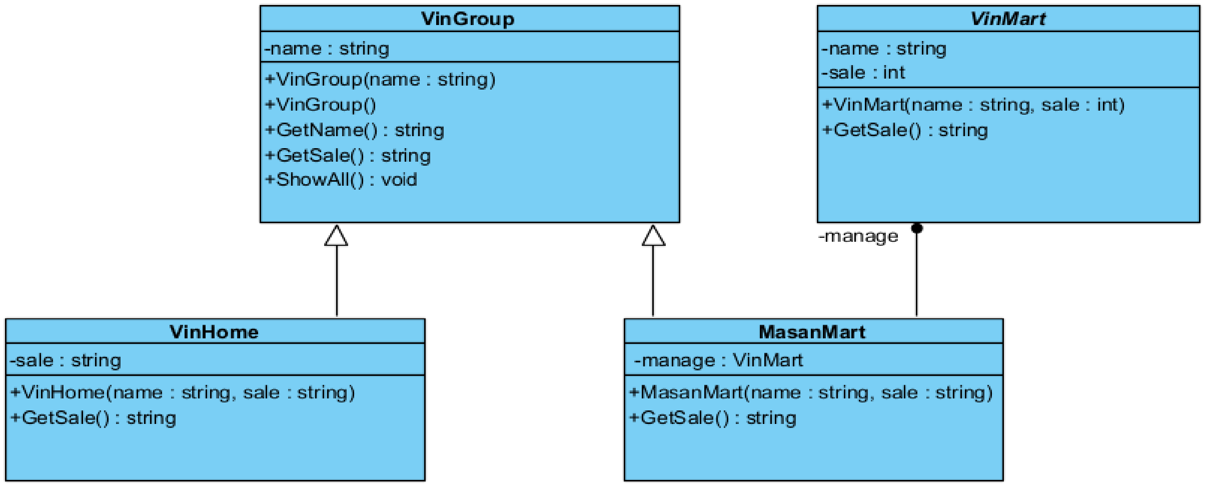

Figure 6- Adapter Design Pattern

Explain:

Author designed a revenue calculation software for VinGroup. Here, VinGroup is the class that calculates whether the revenue of the subsidiaries is enough to meet the target, VinGroup includes (name: string) constructors including (GetName: string, GetSale: string, ShowAll: Void). VinGroup function (name: string) is the constructor, (GetName: string, GetSale: string, ShowAll: Void) used to display the revenue of subsidiaries. In the VinHome class, which returns exactly how much the employee's revenue is, the VinMart class is the initialized adapter class (name: string, sale: int). Therefore, VinHome classes have a young value in terms of employee revenue. The VinMart class only returns the percentage they've reached through the :int type. The MassanMart class is a class that returns the data of the VinMart class: pass or fail

4.3. Behavioral Design Pattern.

4.3.1 Introduction of Behavioral Design Pattern

Behavioral patterns are concerned with algorithms and the assignment of responsibilities between objects. Behavioral patterns describe not just patterns of objects or classes but also the patterns of communication between them. These patterns characterize complex control flow that's difficult to follow at run-time. They shift your focus away from flow of control to let you concentrate just on the way objects are interconnected. (Gamma, 1994)

Behavioral class patterns use inheritance to distribute behavior between classes. This chapter includes two such patterns. Template Method (325) is the simpler and more common of the two. A template method is an abstract definition of an algorithm. It defines the algorithm step by step. Each step invokes either an abstract operation or a primitive operation. A subclass fleshes out the algorithm by defining the abstract operations. The other behavioral class pattern is Interpreter (243), which represents a grammar as a class hierarchy and implements an interpreter as an operation on instances of these classes. (Gamma, 1994)

Behavioral object patterns use object composition rather than inheritance. Some describe how a group of peer objects cooperate to perform a task that no single object can carry out by itself. An important issue here is how peer objects know about each other. Peers could maintain explicit references to each other, but that would increase their coupling. In the extreme, every object would know about every other. The Mediator (273) pattern avoids this by introducing a mediator object between peers. The mediator provides the indirection needed for loose coupling. (Gamma, 1994)

4.3.2 Scenario of behavioral pattern (Strategy Pattern)

Today, as the world grows, the need for human movement also increases. That's why a lot of car manufacturers were born. As you know, there are many types, there are many car manufacturers along with a lot of different prices for each car. In order for users to be less confused when choosing a car, we are implementing a system so that users can make the right car for themselves.

4.3.3 Diagram and explanation

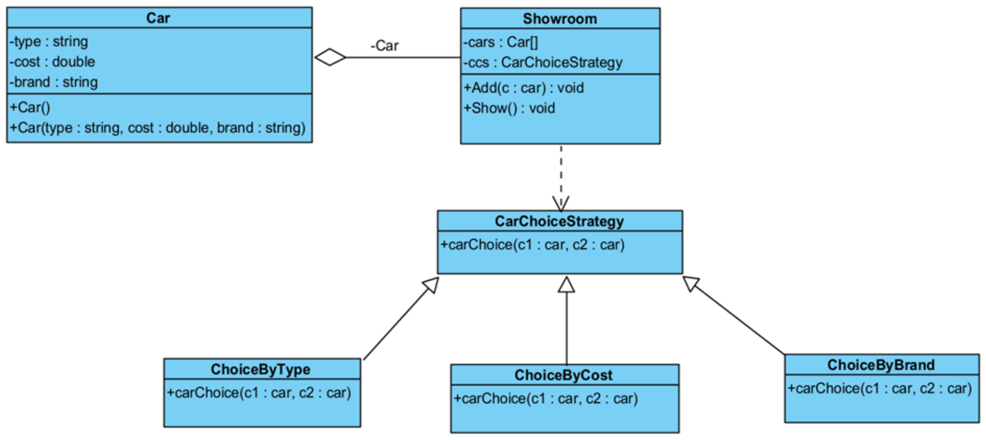

Figure 7- Strategy Design Pattern

This is the example of strategy pattern. I will create a car system where customer can choose car that they want. In this class diagram, I have Car class with type, cost and brand. If I implement CarChoice in Car class, it can't change if I want to change or add more choice for customer. That why I have an interface "CarChoiceStrategy", so the Car class doesn't need to know what method carChoice in CarChoiceStrategy do. When we want to change the customer's choice, we just need to set strategy in Car class.

Now, in this diagram, I only have 3 choices that is choice by type, choice by cost, choice by brand. If in the future, I want to add more options for customer I just need to add 1 implementation and arrange it.

5. Relationship between Design Pattern and OOP

In software engineering, a design pattern is a general repeatable solution to a problem that often occurs in software design. A design pattern is not a complete design that can be converted directly into code. It is a description or pattern of how to solve a problem that can be used in many different situations.

By providing verified, tested development models, design patterns can help to speed up the development process. For programmers and architects who are familiar with patterns, reusing design patterns can assist prevent subtle flaws that can lead to major issues and improve code readability.

Design Patterns and Object Oriented Programming are not necessarily related. That happens when a large number of design patterns are related to Object Oriented Programming. Design patterns are a commonly used approach to program creation. The approach of finding a common pattern language for a field can be extended to functional programming or building bridges or to where it begins, in architecture. OOP is a specific conceptual model, to which several programming pattern conform.

For example, in author's creation design pattern scenario in session 4, author used singleton pattern to handle the requirements from client, user can use the instance many time in many class method and no matter how many time users use this instance, the objects manager always pointing the same instance that initialized from the beginning and they are always have the same value instance and access to the same data that managed. By one instance, it can access every class and it were set as a private object this related to encapsulation concept in OOP. Singleton pattern can be used to solve many scenarios in programming, in this case it was used to solve a requirement in OOP.

Overall, Object-oriented programming is a programming method or concept of programming that organizes code into objects and the relationships of objects. The design pattern will suggest proven method/object construction methods to solve a certain situation in the program.

6. Conclusion

Through the group's presentation, everyone can understand more about the concepts of OOP, from these concepts, the group came up with a plan to create a scenario, from which the whole team created should be a complete application.

Bibliography

Gamma, E., 1994. *Design Patter-Elements of Reusalble Object-Oriented Solfware.* s.l.:s.n.

Indeed, 2021. [Online]
Available at: https://www.indeed.com/career-advice/career-development/what-is-object-oriented-programming

Mozilla, 2021. *constructor.* [Online]
Available at: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes/constructor
[Accessed 20 6 2021].

Gamma, E., 1994. *Design Patter-Elements of Reusalble Object-Oriented Solfware.* s.l.:s.n.

Indeed, 2021. [Online] Available at: https://www.indeed.com/career-advice/career-development/what-is-object-oriented-programming

Mozilla, 2021. *constructor.* [Online] Available at: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes/constructor [Accessed 20 6 2021].