ASSIGNMENT 1 FRONT SHEET

Qualification	**BTEC Level 5 HND Diploma in Computing**		
Unit number and title	Unit 20: Advanced Programming		
Submission date	24/06/2021	**Date Received 1st submission**	
Re-submission Date		**Date Received 2nd submission**	
Student Name	Duong Tien Thanh	**Student ID**	GCH190775
Class	GCH0803	**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	

Grading grid

P1	P2	M1	M2	D1	D2

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

Table of Contents

I. Introduction

In this assignment, I will clarify the concepts of object-oriented programming and design patterns, attributes and characteristics of each. Additionally, I will also mention the specific assumed scenarios applied in each feature, along with the class diagrams that help to generalize the protocols of each object. Finally, I will analyze the relationship between object-oriented programming and design pattern.

II. OOP general concepts

1. Definition

Object-oriented programming(OOP) combines a collection of variables (properties) and functions (methods) into a single entity known as an "object." These items are classified into classes, which allow individual objects to be grouped together. OOP may assist you in considering objects in a program's code and the many activities that could occur in connection to the objects.

This programming approach is prevalent in widely known programming languages such as Java, C++, and PHP. These languages aid in the organization and structure of software applications. When creating complicated applications, programmers frequently utilize OOP.

Figure 1: OOP's characteristics

2. Characteristic of OOP

- **Encapsulation:**
  The many objects within each application will attempt to communicate with one another automatically. If a programmer wishes to prevent items from interacting with one another, they must be contained in separate classes. Classes cannot alter or interact with an object's particular variables and functions due to the encapsulation process (Team, 2021).

The idea of encapsulation works in a digital fashion to build a protective barrier around the information that isolates it from the rest of the code, just as a pill "encapsulates" or retains the drug inside of its coating. This object can be replicated by programmers in other areas of the program or in other applications (Team, 2021).

- **Abstraction:**
  Abstraction is similar to encapsulation in that it conceals some attributes and functions from outside code in order to simplify the interface of the objects. Abstraction is used by programmers for a variety of reasons. Overall, abstraction helps to isolate the impact of code modifications so that if something goes wrong, the change only affects the variables presented and not the outside code (Team, 2021).
- **Inheritance:**
  Using this idea, programmers may expand the functionality of existing classes in the code to reduce redundant code. For example, HTML code components such as a text box, select field, and checkbox share particular attributes with certain methods.
  Rather than rewriting the attributes and methods for each type of HTML element, they may be defined once in a generic object. By naming that object "HTMLElement," other objects will inherit its attributes and functions, allowing you to minimize needless code (Team, 2021).
  The superclass is the primary object, and any objects that follow it are subclasses. Subclasses can have their own components while relying on the superclass for what they require (Team, 2021).
- **Polymorphism:**
  This method, which translates as "many forms or shapes," enables programmers to generate numerous HTML components based on the kind of item. This idea enables programmers to reimagine how something works by altering how it is done or the portions that are done. Overriding and overloading are polymorphism terms (Team, 2021).
- **Objects and Methods:**

  An object is a collection of data, procedures for manipulating the data, and functions that provide data-related information. Methods refer to both procedures and functions (Team, 2021).

- **Class:**

  A class is a crucial construct in many object-oriented languages. A class is a group of things that are grouped together based on their members. Classes, like objects, can be implemented in traditional programming languages by employing separate

compilation and structs for encapsulation. All members defined in the declarations will be present in each object in the class (Team, 2021).

3. Advantages of OOP

- OOP supports the modular framework for application development and offers it. Good definitions are implemented by concealing internal information, abstract data types are implemented
- The retention and modification of the current code is easy, whilst existing code may be changed to existing objects with little modifications.
- The OOP provides an excellent code library foundation for quickly adjusting and modifying program components.

III. OOP scenario

1. Scenario

The selling system will be used as an example for OOP. There are three actors in this system: the customer, the seller, and the administrator. Login, search products, check details, and buy products are all available to customers. The seller can log in, search products, enter data, and check for client information such as phone numbers, etc. The administrator can log in, search for and manage all client information. Admins can also add, update, and delete products, as well as change their prices based on market conditions. There will be a memory for all of the items that will keep track of all of the locations that they have added. Memory can be managed by the administrator, but not by customers, and memory data cannot be removed. Customers can only buy products after registering an account.

2. Usecase Diagram

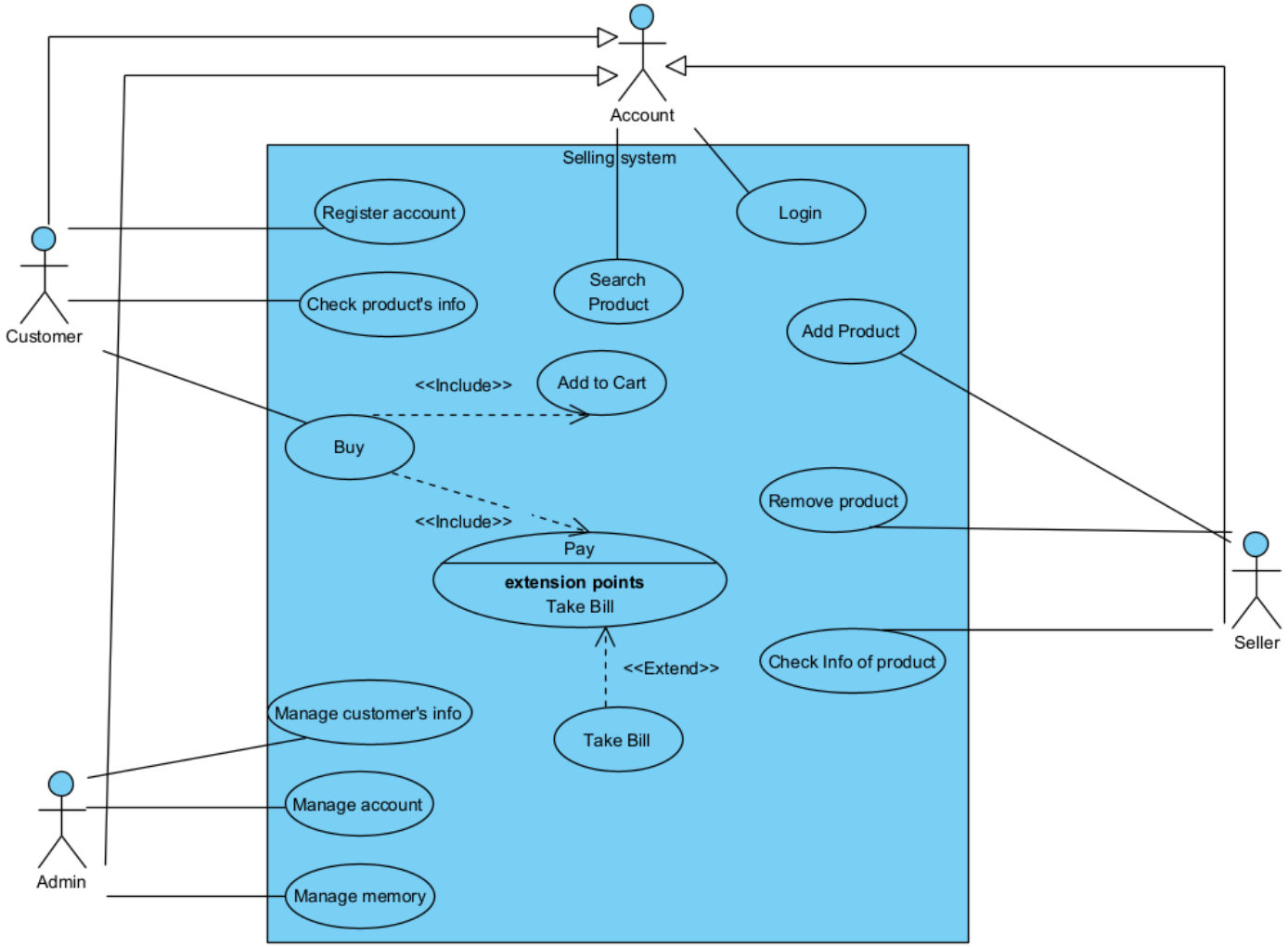

Figure 2: Usecase Diagram

In the use case diagram applied to the scenario that consists of three actors:

- Customer: Login, search products, check details products, and buy products are all available to customers. Customers can only search for products, view information of products, and buy them after registering an account, completing adding to cart and paying. Besides, customer also can take bill or not after paying for product that he/she just buy.
- Seller: Seller can log in, search products, add products or remove them and check for product's information such as quantities,author, etc.
- Administrator: Administrator can log in, search for and manage all customer's information.Admin can change it base on customer's request. Admins can also change product's prices based on market conditions. Memory can be managed by the administrator.

Name of Use Case:	Search product
Description:	User can search a product by keyword
Actors:	Customer, Seller, Admin
Preconditions:	1. User needs to Login first or stay in logged in state
Postconditions:	No post condition
Flow:	1. User enter a keyword 2. System search in data for keyword 3. Display a list of products corresponding to that keyword
Alternative Flows:	2.1. No product found
Exceptions:	No exception
Requirements:	

Name of Use Case:	Remove product
Description:	User can remove product
Actors:	Seller
Preconditions:	1. User needs to Login first or stay in logged in state 2. User must be Seller
Postconditions:	No post condition
Flow:	1. User enter keyword (product name) 2. System search in data for keyword 3. Display a list of products corresponding to that keyword 4. Find product user want to remove 5. Select remove function to remove product
Alternative Flows:	2.1. No product found
Exceptions:	No exception
Requirements:	

Name of Use Case:	Manage account
Description:	User can add, update or delete account
Actors:	Admin
Preconditions:	1. User needs to Login first or stay in logged in state 2. User must be Admin
Postconditions:	No post condition
Flow:	1. User go to the manage account and select add, update or delete function 2. Enter information of account user want to add or update, find account user want to remove 3. Save information account when user added or updated
Alternative Flows:	No alternatives
Exceptions:	No exception
Requirements:	

3. Class Diagram

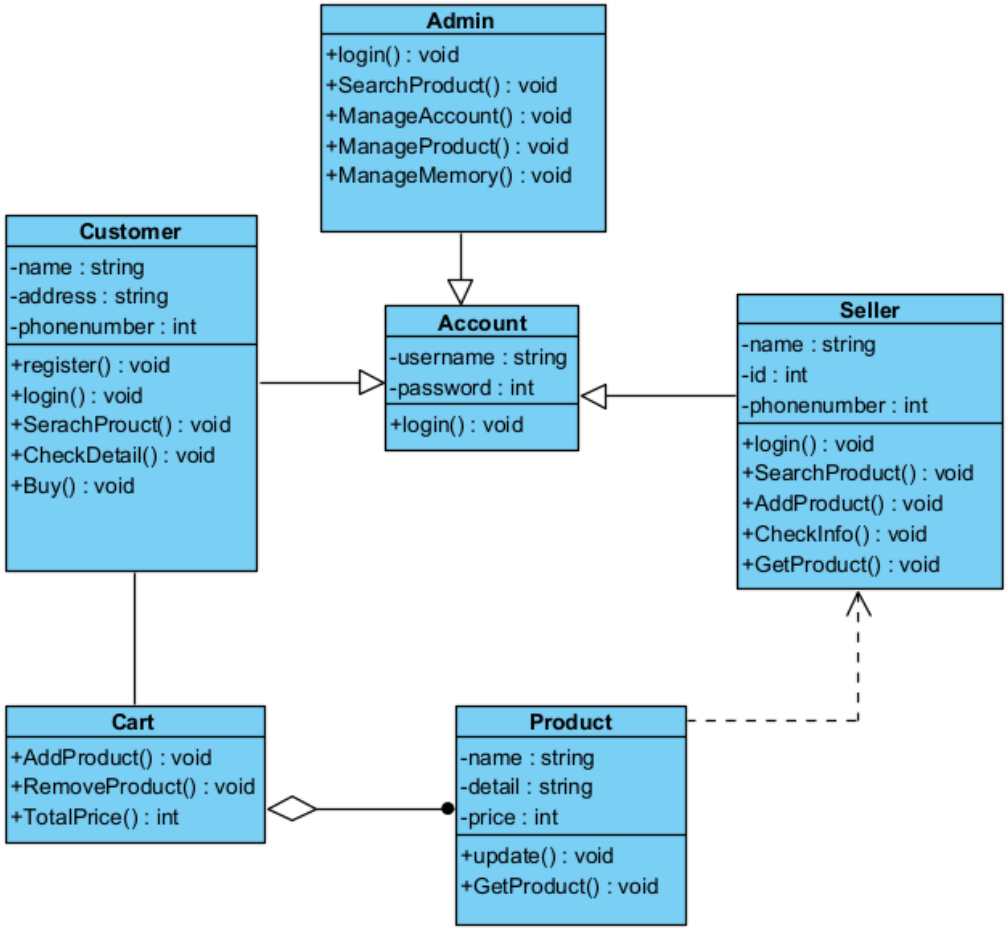

Figure 3: Class Diagram

Explain:

- Customer, Seller and Admin also inherit from Account clas with function login.
- Customer can add product to Cart and also remove it. Cart will help custom to calculate total price of all products.
- There is aggregation relationship between Cart and Product. It means that if Cart is removed, Product is not affected.
- There is dependency relationship between Product class and Seller class. It means that Seller changes something, it will affect to Product such as forgetting to add product or removing product.

IV. Design Patterns

Definition:

Design Patterns are a technique in object-oriented programming. Design Pattern is used frequently in OOP languages. It provides you with “design patterns”, solutions to common problems, commonly encountered in programming. The problems you face may be your own to come up with a solution, but it may not be optimal. Design Pattern helps you solve problems in the most optimal way, providing you with solutions in OOP programming (Coder, 2018).

According to (Beasley, 2017) **pros and cons of design pattern**

Pros:

- Easy to adapt to predictable changes in business needs.
- Easy to unit test and validate individual components.
- Can provide organization and structure when business requirements become very complicated.

Cons:

- Beginner engineers may not understand them.
- Oftentimes used improperly without a realistic understanding of how the software is likely to change.
- Can add memory and processing overhead, so sometimes not appropriate for applications such as low level systems programming or certain embedded systems.

1. Creational pattern

Definition:

These design patterns are all about class instantiation. This pattern can be further divided into class-creation patterns and object-creational patterns. While class-creation patterns use inheritance effectively in the instantiation process, object-creation patterns use delegation effectively to get the job done (Making, 2021).

- **Abstract Factory:** Creates an instance of several families of classes (Making, 2021).
- **Builder:** Separates object construction from its representation (Making, 2021).
- **Factory Method:** Creates an instance of several derived classes (Making, 2021).
- **Object Pool:** Avoid expensive acquisition and release of resources by recycling objects that are no longer in use (Making, 2021).
- **Prototype:** A fully initialized instance to be copied or cloned (Making, 2021).
- **Singleton:** A class of which only a single instance can exist (Making, 2021).

2. Structural pattern

Definition:

These design patterns are all about Class and Object composition. Structural class-creation patterns use inheritance to compose interfaces. Structural object-patterns define ways to compose objects to obtain new functionality (Making, 2021).

- **Adapter:** Match interfaces of different classes (Making, 2021).
- **Bridge:** Separates an object's interface from its implementation (Making, 2021).
- **Composite:** A tree structure of simple and composite objects (Making, 2021).
- **Decorator:** Add responsibilities to objects dynamically (Making, 2021).
- **Façade:** A single class that represents an entire subsystem (Making, 2021).
- **Flyweight:** A fine-grained instance used for efficient sharing (Making, 2021).
- **Private Class Data:** Restricts accessor/mutator access (Making, 2021).
- **Proxy:** An object representing another object (Making, 2021).

a. Adapter Design Pattern

Definition:

Adapter Pattern is one of the Patterns in the Structural Pattern group. The Adapter Pattern allows unrelated interfaces to work together without having to modify them directly. The object that helps to connect the interfaces is called Adapter (Coder, 2018).

An Adapter Pattern consists of the following basic components:

- Client: class that uses objects with the Target interface.
- Target: an interface containing the functions used by the Client.
- Adaptee: interface definition is not compatible, needs to be integrated.
- Adapter: integration class, helping incompatible interfaces integrate with the working interface. Perform interface conversion for Adaptee and connect Adaptee to Client.

When to use the Adapter Design Pattern:

By using the adapter, we don't need to spend much of time to restructure the old resource. It will be acceptable when we restructure the resource the first time, but it will be very waste of time when we want to restructure the resource more than one time. This is the reason why the adapter is used.

Basic Structure:

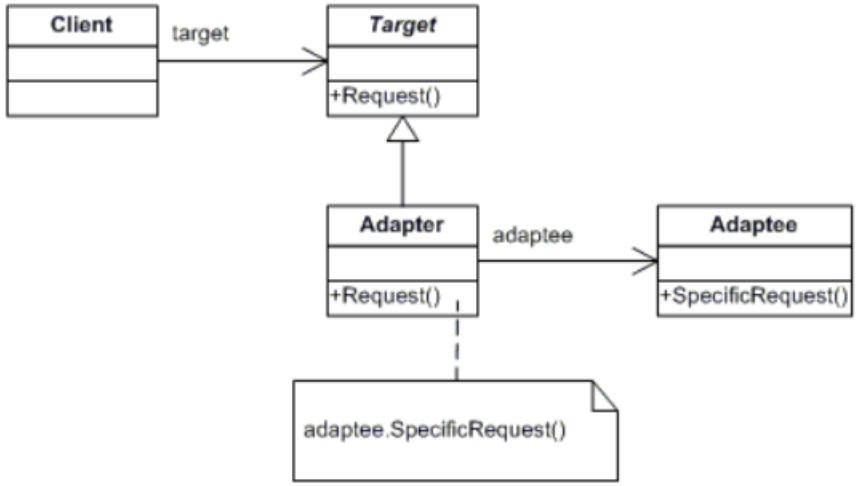

Figure 4: Basic Adapter Structural Diagram (dofactory)

Scenario:

TP Bank wants to build a new banking system. The former bank account had the account holder's name, account number, and the amount in the account. Currently TP Bank wants to merge the amount in the account with the account balance of OceanBank. We need to use an adapter to ensure that after merging the 2 banks, the old apps and data will still be executed.

UML class diagram:

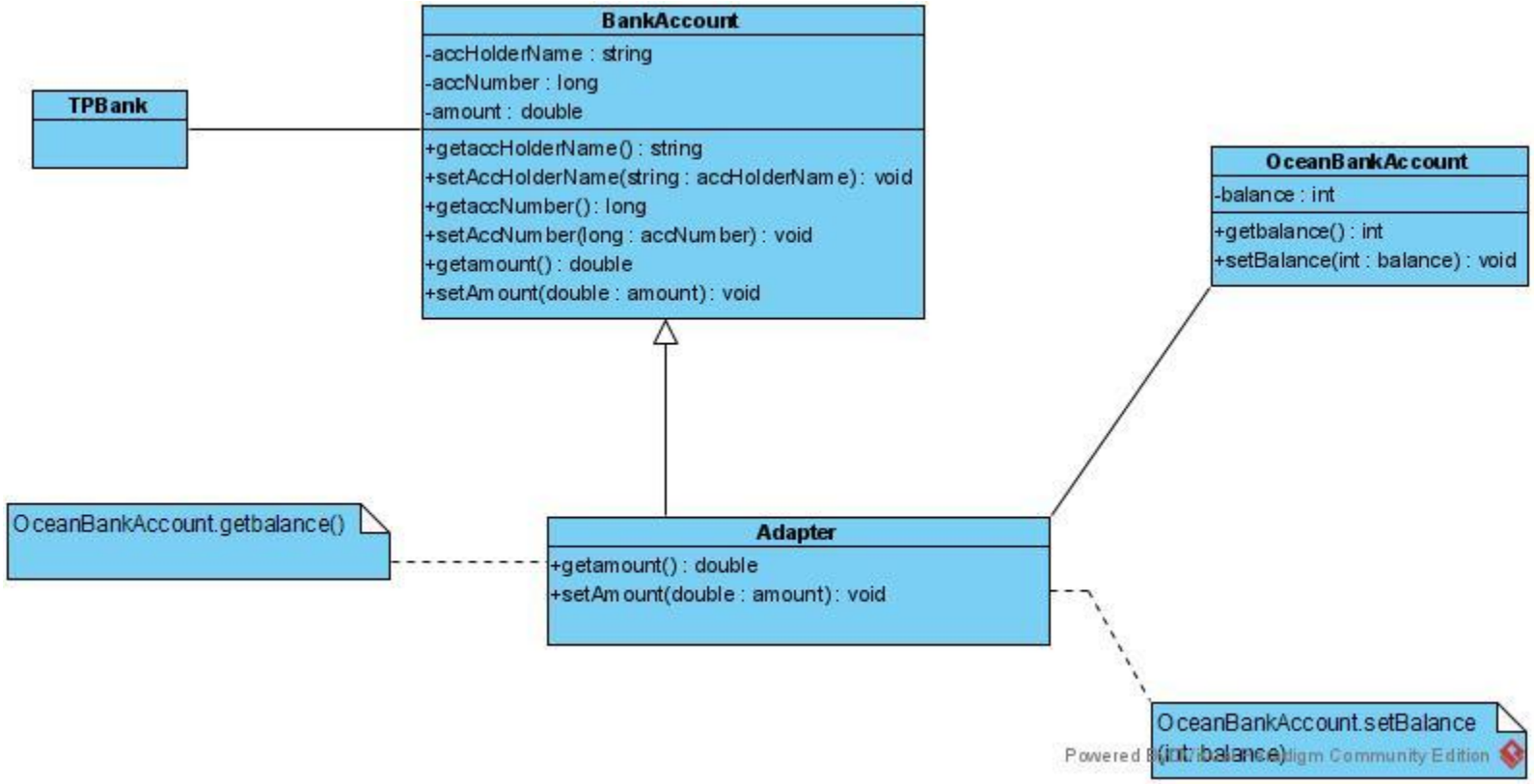

Figure 5: Adapter Diagram

UML explain:

There are four classes in my class diagram. The relationship between Client and CreditCard is an association relationship. And BankCustomer inherits BankDetails and then implements it in CreditCard class.

- Client: Customer is Client
- Target: CreditCard is Target Interface. This is the desired interface class which will be used by the clients namely the issuance of credit cards

- Adapter: BankCustomer is Adaper. This class is a wrapper class which implements the desired target interface and modifies the specific request available from the Adaptee class. Inherit the information in the BankDetails class and then implement it in the CreditCard class.
- Adaptee: BankDetails is Adaptee. This is the class which is used by the Adapter class to reuse the existing functionality and modify them for desired use.

b. Composite Design Pattern

Definition:

The composite pattern is a partitioning design pattern that defines a collection of items that are processed as if they were a single instance of the same type of object. A composite's purpose is to "compose" things into tree structures in order to express part-whole hierarchies. It allows you to create a tree structure and assign tasks to each node in the tree structure (GeeksforGeeks, 2018).

According to (Dofactory, 2021) , a composite pattern consists of the following basic components:

- Component**:**
  - Specifies the interface for the composition's objects.
  - As applicable, implements default behavior for the interface common to all classes.
  - Declares an access and management interface for its child components.
  - (optional) specifies and implements an interface for accessing a component's parent in the recursive structure, if applicable
- Leaf**:**
  - Represents the composition's leaf objects. There are no children for a leaf.
  - Determines the behavior of the composition's primitive items.
- Composite**:**
  - Describes the behavior of components that have children.
  - Child components are stored here.
  - In the Component interface, it implements child-related operations.

- Client:
  - Manipulates objects in the composition through the Component interface.

When to use the Composite Design Pattern:

- When we want to create objects in tree structures to represent the classification system.
- When we want the client to be able to ignore the difference between the object constructors and the object itself. Clients affect each object and its components uniformly.

Basic Structure:

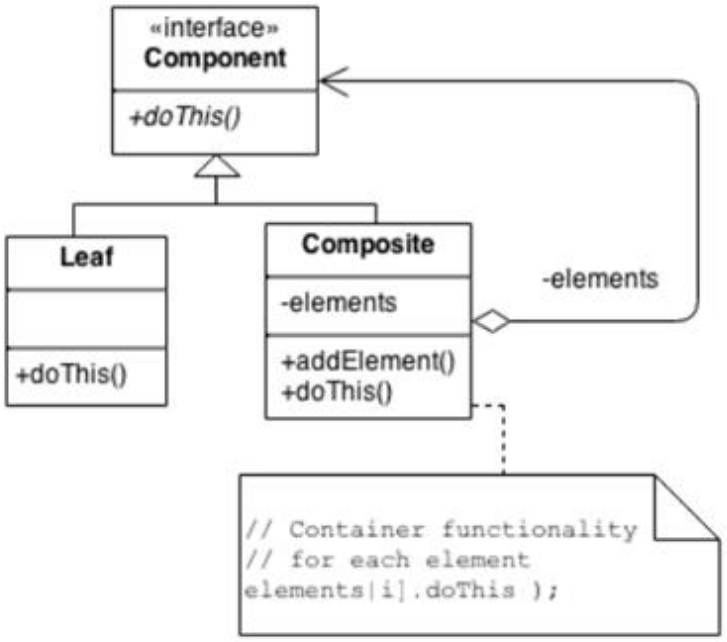

Figure 6: Basic Composite Structural Pattern

Scenario:

A product management system. Customer can add and delete products to buy and the system will summarize and give details of additional products, prices and total prices of all products. The Composite pattern will be used to specifically divide the work and properties of each component and object in the system.

UML class diagram:

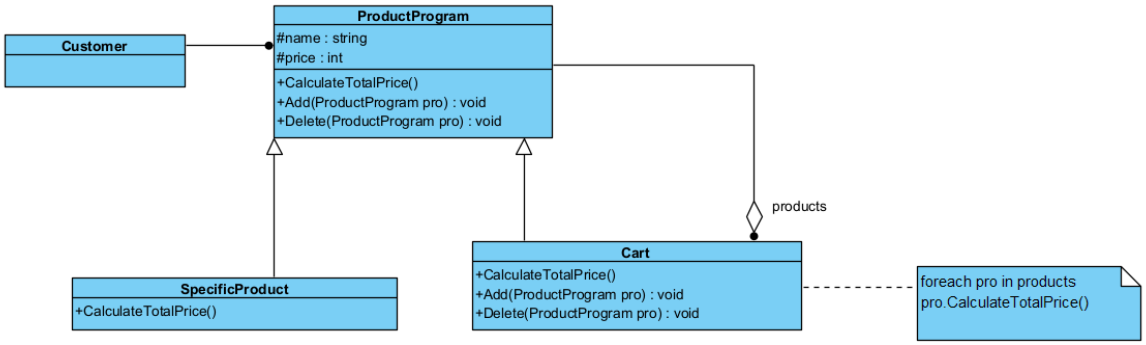

Figure 7: Composite Diagram

UML explain:

There are four classes in my class diagram. The relationship between Client and ProductProgram is an association relationship. SpecificProduct and CompositeProduct inherit ProductProgram. Besides, it's an aggregation relationship between ProductProgram and CompositeProduct.

- Client: Customer is client in this UML.
- Component: Class ProductProgram is Component. It contains activities that other classes need to perform.
- Leaf: SpecificProduct is Leaf and it can only do already task(s) without making more functions, …
- Composite: is CompositeProduct with inheriting from Component and it can make more activities, …

3. Behavioral pattern

Definition:

These design patterns are all about Class's objects communication. Behavioral patterns are those patterns that are most specifically concerned with communication between objects (Making, 2021).

- **Chain of responsibility:** A way of passing a request between a chain of objects (Making, 2021).
- **Command:** Encapsulate a command request as an object (Making, 2021).
- **Interpreter:** A way to include language elements in a program (Making, 2021).
- **Iterator:** Sequentially access the elements of a collection (Making, 2021).
- **Mediator:** Defines simplified communication between classes (Making, 2021).
- **Memento:** Capture and restore an object's internal state (Making, 2021).
- **Null Object:** Designed to act as a default value of an object (Making, 2021).
- **Observer:** A way of notifying change to a number of classes (Making, 2021).
- **State:** Alter an object's behavior when its state changes (Making, 2021).
- **Strategy:** Encapsulates an algorithm inside a class (Making, 2021).
- **Template method:** Defer the exact steps of an algorithm to a subclass (Making, 2021).
- **Visitor:** Defines a new operation to a class without change (Making, 2021).

References

Beasley, J., 2017. *What are some pros and cons of using Design Patterns to describe your business model?.* [Online]
Available at: https://www.quora.com/What-are-some-pros-and-cons-of-using-Design-Patterns-to-describe-your-business-model
[Accessed 20 June 2021].

Coder, G., 2018. *Giới thiệu Design Patterns.* [Online]
Available at: https://gpcoder.com/4164-gioi-thieu-design-patterns/
[Accessed 19 June 2021].

Coder, G., 2018. *Hướng dẫn Java Design Pattern – Adapter.* [Online]
Available at: https://gpcoder.com/4483-huong-dan-java-design-pattern-adapter/
[Accessed 19 June 2021].

Dofactory, 2021. *Composite Design Pattern in C# .NET.* [Online]
Available at: https://www.dofactory.com/net/composite-design-pattern
[Accessed 21 June 2021].

GeeksforGeeks, 2018. *Composite Design Pattern.* [Online]
Available at: https://www.geeksforgeeks.org/composite-design-pattern/
[Accessed 21 June 2021].

Making, S., 2021. *Design Patterns.* [Online]
Available at: https://sourcemaking.com/design_patterns
[Accessed 20 June 2021].

Team, I. E., 2021. *What Are the Four Basics of Object-Oriented Programming?.* [Online]
Available at: https://www.indeed.com/career-advice/career-development/what-is-object-oriented-programming
[Accessed 19 June 2021].