ASSIGNMENT 1

Qualification	**BTEC Level 5 HND Diploma in Computing**		
Unit number and title	Unit 20: Advanced Programming		
Submission date		**Date Received 1st submission**	
Re-submission Date		**Date Received 2nd submission**	
Student Name	Nguyen Duc Cuong	**Student ID**	GCH18641
Class	GCH0705	**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

LIST OF FIGURES

1 INTRODUCTION

This report discusses the foundations of OOP, along with its main attributes: Encapsulation, Inheritance, Polymorphism and Abstraction. Besides, I will also give the scenarios as well as design a class diagram to solve a specific problem. Finally, the report will outline knowledge of three types of Design Patterns: Creational, Structural and Behavioral Pattern. For each type of template, we will give the definition and usage along with the scenarios to solve specific problems.

2 OOP GENERAL CONCEPTS

Object-Oriented Programming (OOP) is a computer programming paradigm that organizes software design around data or objects, rather than functions and logic. An object can be defined as a data field with unique properties and behaviors. The properties in object-oriented programming are Encapsulation, Inheritance, Polymorphism and Abstract. (Lain D. Craig, 2007)

2.1 ENCAPSULATION

Encapsulation is the state of the object that is protected from external code access such as changes in state or direct view. It is up to the coder to allow the external environment to influence the internal data of an object in any way. This is the property to ensure the integrity and security of the object. In addition, related classes can be grouped together into packages. (Lain D. Craig, 2007)

2.2 INHERITANCE

Inheritance is the ability to allow us to build a new class based on the definitions of an existing class. An existing class is called a Parent class, a newly arising class is called a Child class and naturally inherits all the components of a Parent class, and can share or extend the existing properties without having to redefine. (Lain D. Craig, 2007)

2.3 POLYMORPHISM

When a task is performed in different ways it is called polymorphism. Polymorphism provides the ability for the programmer to call an object's method in advance, although it has not been determined whether the object has the method to be called. Until the execution (run-time), the program can identify the object and call the corresponding method of that object. (Lain D. Craig, 2007)

2.4 ABSTRACTION

Abstraction is the process of hiding implementer details and exposing the feature only to the user. Abstraction allows you to eliminate the complexity of an object by showing only the necessary properties

and methods of the object in programming. Abstraction helps you to focus on the essentials of the object instead of how it does it. (Lain D. Craig, 2007)

3 OOP SCENARIO

3.1 SCENARIO

A company hired us to develop a simple entertaining game for them. This game will be integrated into the company's website so that users can play when their computers cannot connect to the network. This is basically an obstacle avoidance game where the user controls a running dinosaur and avoids obstacles in front of him. Furthermore, it is possible to choose colors for dinosaurs. The obstacles will include the cactus and the pipe. To avoid the cactus users need to press the button to help the dinosaur jump over and vice versa, the user needs to press the down button to help the dinosaur roll through the pipes. The game will not only like that but the speed of the dinosaur will increase over time ie it will run faster and faster. The game will stop when the dinosaur hits a cactus or a pipe, and the system will save the longest distance the dinosaur can run.

3.2 USE CASE DIAGRAM

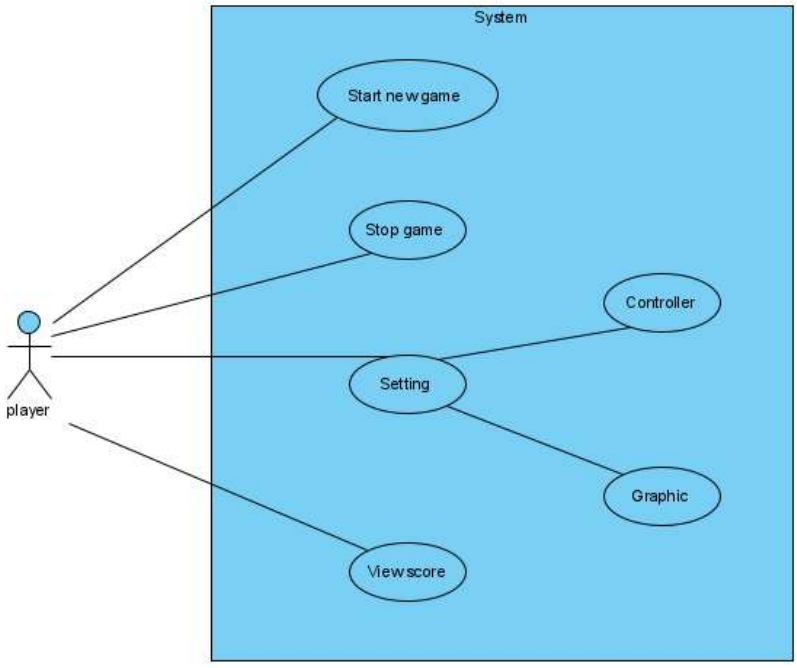

figure 1: Use Case Diagram for OOP

This game is basically an obstacle avoidance game. The player controls the dinosaur to avoid 2 obstacles, the cactus and the pipe. The game has 4 main functions: Start new game, Stop game, setting and view score. In the setting function will include keyboard control, graphics settings such as changing the color of dinosaurs. And players will see their highest score at the view score.

3.3 CLASS DIAGRAM

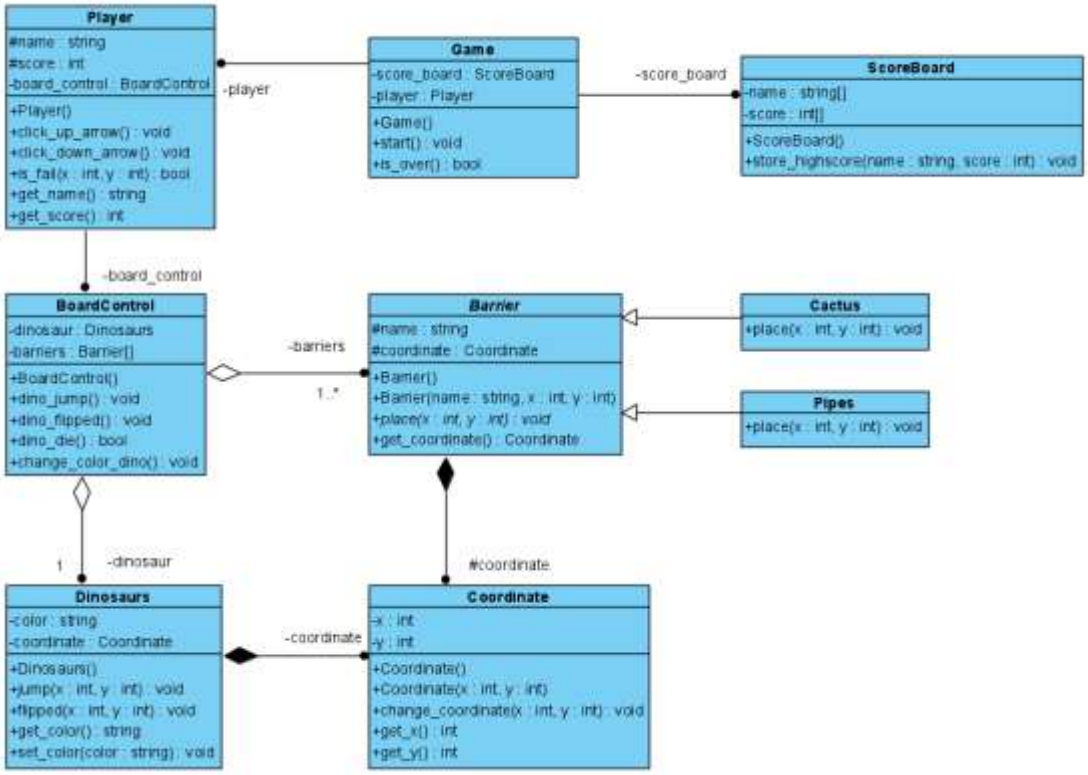

figure 2: Class diagram

- **Polymorphism**
  - Override: the cactus and pipes are override from barrier
  - Virtual: place () setting as a method can be inheritance and overridden to facilitate dynamic coordination (from cactus and pipes).
  - Pointer: in the barrier class has initialized a pointer name coordinate from Coordinate class for memory address.
  - Inheritance: the cactus and pipes class will inherit all the features from barrier class.

- **Abstract class:** the barrier class will be initialized as an abstract class such as place() so that the subfunctions like cactus and pipes class can override.
- **Destructor:** when pointer is used, so that the barrier have to create a function to delete the data store in heap.
- **Relationship:** association, aggregation (boardControl contains Dinosaur and Barrier), generalization (cactus and pipes is a part of barrier) and composition (if Dinosaur or Barrier is deleted, Coordinate is also deleted).

4 DESIGN PATTERN

4.1 DEFINITION

Design patterns are overall solutions that have been optimized, reused for common software design problems that we encounter on a daily basis. This is the set of thought-out solutions that have been resolved in a particular situation. Programmers can apply this solution to solve similar problems. The problems you encounter may be able to come up with a solution on your own, but they may not be optimal (Gamma et al., 1994)

It is important to understand that the Design pattern is not a specific language. Design patterns can be performed in most programming languages. It helps you to solve the problem in the best way, provides you with solutions in object-oriented programming (OOP). (Gamma et al., 1994)

4.2 TYPES OF DESIGN PATTERN

The design pattern system is divided into 3 groups: Creational group (5 models), Structural group (7 models) and Behavioral group (11 models):

- **Creational Patterns:**

Creational design patterns are design patterns that deal with object creation mechanisms, trying to create objects in a manner suitable to the situation. The basic form of object creation could result in design problems or added complexity to the design. Creational design patterns solve this problem by somehow controlling this object creation (Dinesh Rajput, 2017).

Creational Pattern includes 5 models:

  - Abstract Factory
  - Builder

- Factory
- Prototype
- Singleton

- **Structural Patterns**

Structure design patterns are design patterns that make it easy to design by defining a simple way to realize relationships between entities. Structure design patterns concern how classes and objects can be constructed, to form larger structures. In addition, these patterns focus on, how classes inherit from each other, and how they are composed of other classes (Dinesh Rajput, 2017).

Structural Pattern includes 7 models:

- Adapter
- Bridge
- Composite
- Decorator
- Façade
- Flyweight
- Proxy

- **Behavioral Patterns**

Behavioral design patterns are design patterns that define common communication patterns between objects. That way, these templates increase flexibility in the implementation of communication (Dinesh Rajput, 2017)

Behavioral Pattern includes 11 models:

- Chain of responsibility
- Command
- Interpreter
- Iterator
- Mediator
- Memento
- Observer
- State
- Strategy
- Template method
- Visitor

4.3 CREATIONAL DESIGN PATTERN

4.3.1 SINGLETON PATTERN

The Singleton pattern is a creational design pattern, it is one of the simplest design patterns. According to the singleton design pattern, the class provides the same single object for each call--that is, it is restricting the instantiation of a class to one object and provides a global point of access to that class. So, the class is responsible for creating an object and also ensures that only a single object should be created for each client call for this object. This class doesn't allow a direct instantiation of an object of this class. It allows you to get an object instance only by an exposed static method (Dinesh Rajput, 2017).

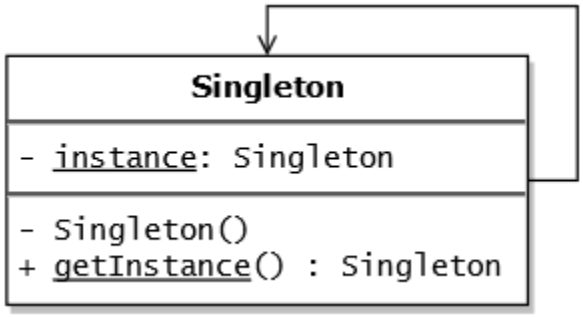

figure 3: Singleton Pattern

4.3.1.1 SCENARIO

When we shop online, it can be seen that each buyer will always have a separate shopping cart that stores the products that the buyer has chosen to buy. Furthermore, the buyer can also edit the number of selected products and remove it from the shopping cart. When you meet your requirements, you will pay.

4.3.1.2 CLASS DIAGRAM

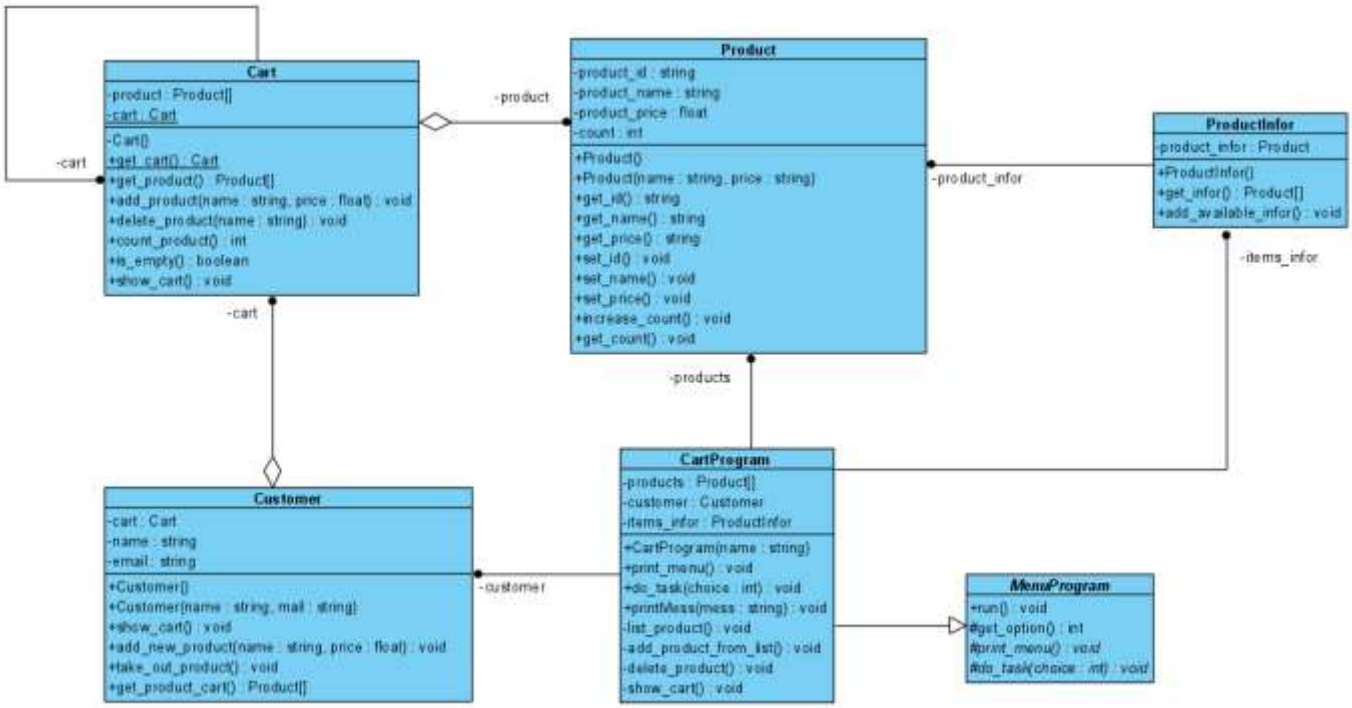

figure 4: Class Diagram for Singleton sceanario

At **Customer Class**, it can be seen that contains objects of Cart class and this object will be declared by pointer, if "***Cart***" object in Customer Class is destroyed, object in Cart class can still exist. The properties "***name***" and "***email***" are initialized to store some basic information of the user. The "**add_new_product ()**" method is used to help the user add products from the list to the cart. Besides, users also have the ability to delete the products that they added with the "**take_out_product ()**" method. The "**show_cart ()**" method is used to show on-screen the products the user has added from the list to the shopping cart.

The **Cart Class** will include a Product vector to store the product, a private static pointer basket to provide access to the class's properties. We will have one more private constructor that will ensure that there is only one "***cart***" and the user cannot create any more objects. Besides, this class has a public static method "**get_cart ()**" that returns the static pointer "**cart**".

The Product class includes some basic product information such as the id, product name, and price of the product along with getter / setter methods to assign and return values to variables.

The relationship defined between the **Customer class** and the **Product class** is *Aggregation* which means that the Cart is owned by the Customer (if there are no customers or the customers leave, the Cart will not be instantiated or deleted), and the Relationship between the **Cart class** and the **Product class** is also *Aggregation* similar to the **Customer class** and the **Cart class**.

Besides, we have some classes used to interact with the user, namely **ProductInfor** and **CartProgram**. Basically, the **ProductInfor class** will also provide a list of currently available products for the user to choose from. **CartProgram** is a class that provides a menu interface for the user to interact with the program and this class is inherited from the **MenuProgram class**. The **MenuProgram class** includes a number of methods such as displaying menus, performing tasks.

4.3.1.3 ACTIVITY DIAGRAM

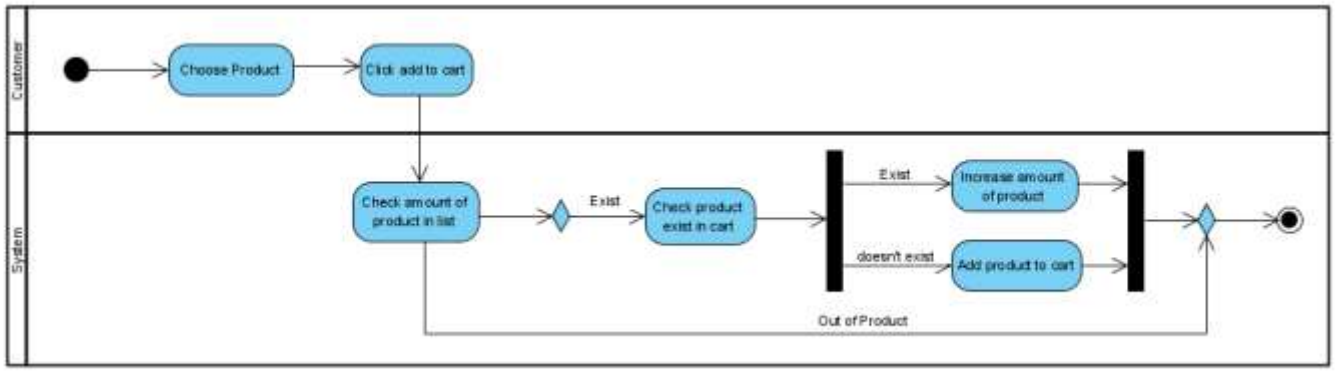

figure 5: Activity Diagram

This is the Activity Diagram for adding products from a list to the cart. The diagram will work as follows:

When a user adds a product to a shopping cart, the system before the name will check if the item has enough quantity or not, if out of stock, it will stop working. If the product is still available, the system will next check to see if the user has selected this product before - that is, if the customer has selected this product, the system will increase the quantity in the cart. If not, then add new products to cart.

4.3.2 ABSTRACT FACTORY PATTERN

Abstract factory is one of the best ways to create an object. An abstract factory provides an interface for creating families of related objects without specifying their concrete classes. This pattern is also a hierarchy that encapsulates: many possible "platforms", and the construction of a suite of "products". (Dinesh Rajput, 2017)

- When to **use** the abstract factory:
  - The system needs to be independent from the way the products it works with are created.
  - The system is or should be configured to work with multiple families of products.
  - A family of products is designed to work only all together.

- The creation of a library of products is needed, for which is relevant only the interface, not the implementation, too.

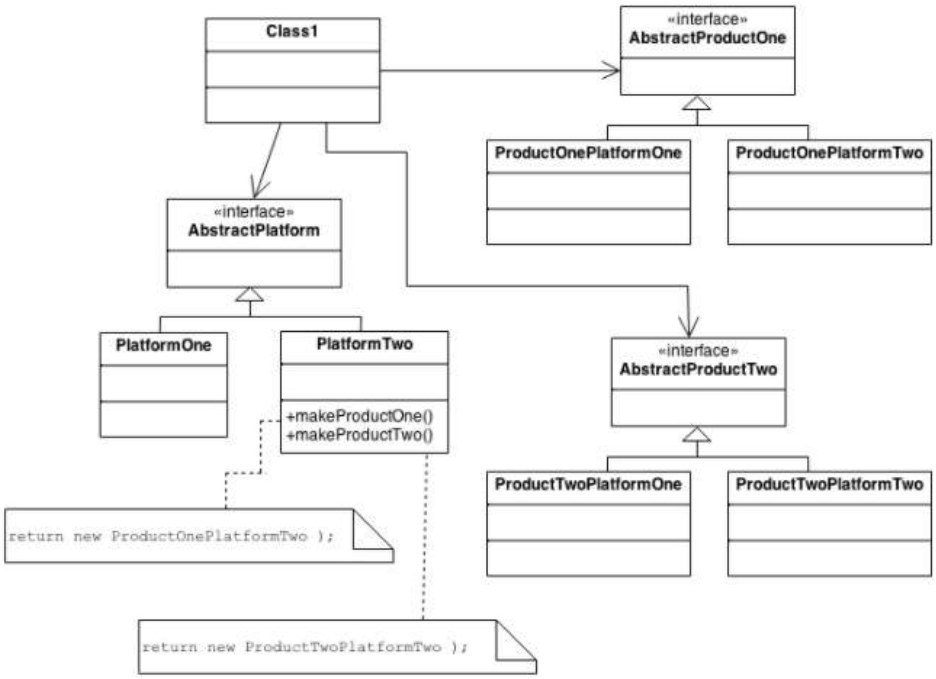

figure 6: Structure of Abtract Factory

4.3.2.1 SCENARIO

A furniture company (Furniture) DaiDuong specializes in manufacturing chairs: traditional chairs (Traditional Chair) and modern chairs (Modern Chair). With the business situation like sitting on the goldmine, the company decided to open more production of tables (tables). Since we have previous experience in manufacturing chairs (Traditional Chair and Modern Chair) but the process of manufacturing tables and chairs according to each material is different, so the company will have a factory (Factory): 1 specialized in manufacturing traditional furniture (Factory Traditional) and one specialized in manufacturing modern furniture (Factory Modern) and both factories can manufacture chairs and tables (Abstract Factory). When customers need something just go to the store to buy. Then each material goods will be transferred to the respective workshop for production (Abstract Factory) to the table (table) and the corresponding chair (Chair).

4.3.2.2 CLASS DIAGRAM

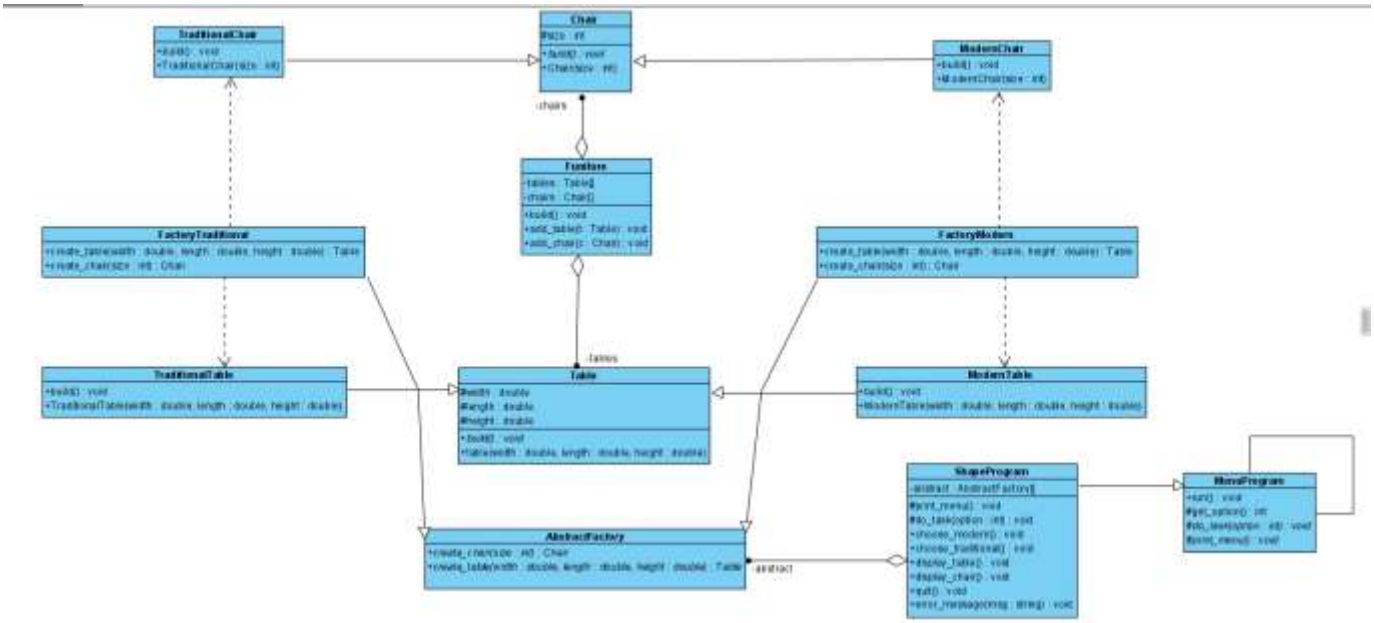

figure 7: Class Diagram for AbstractFactory

The class diagram is based on the abstract factory pattern, this aims of abstract factory is to divide and conquer, each factory (FactoryTraditional and FactoryModern) will have its own product (Table and Chair). The big one is AbstractFactory – it will initialize interface or abstract class that contains method to create abstract objects. In the product class – setting concrete objects that is initialized in factory (FactoryTraditional and FactoryModern).

4.3.2.3 ACTIVITY DIAGRAM

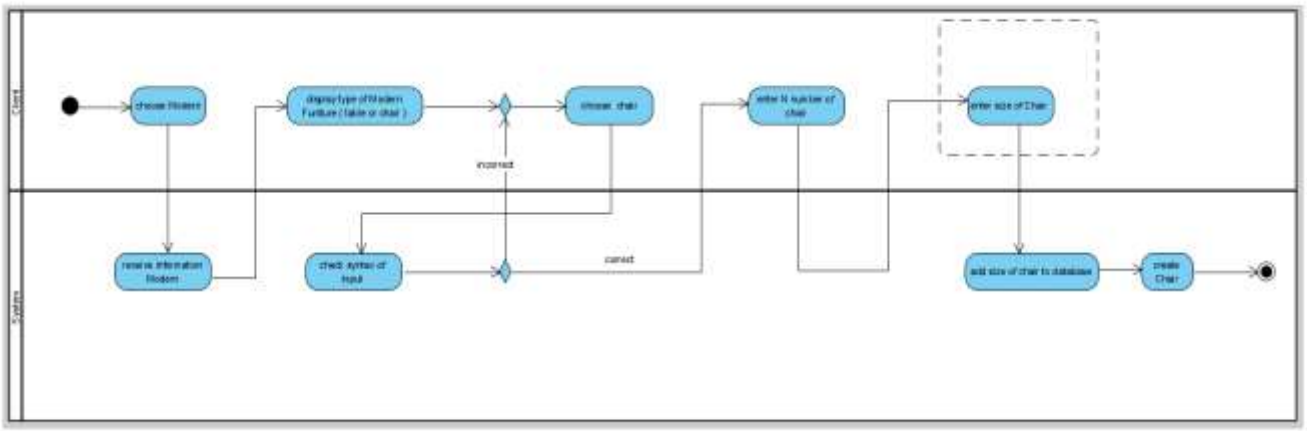

figure 8: Activity Diagram

The activity diagram above is showing about user actions and how the program behaves when user choose Modern.

At first, after users choose Modern, the program will display the screen tell user to choose table or chair. If users enter wrong number, the program will make user to re-enter until it right. If enter right number such as 1. chair, the program continues asking for enter N Chair of furniture and enter the size of its. When user is finishing enter syntax, the program will create chair inside the database and finish program.

4.4 STRUCTURE DESIGN PATTERN: COMPOSITE PATTERN

Composite pattern is a partitioning design pattern and describes a group of objects that is treated the same way as a single instance of the same type of object (Dinesh Rajput, 2017)

- **Composite pattern has fours elements**:
  - Component – Component declares the interface for objects in the composition and for accessing and managing its child components. It also implements default behavior for the interface common to all classes as appropriate.
  - Leaf – Leaf defines behavior for primitive objects in the composition. It represents leaf objects in the composition.
  - Composite – Composite stores child components and implements child related operations in the component interface.
  - Client – Client manipulates the objects in the composition through the component interface.

Client use the component class interface to interact with objects in the composition structure. If recipient is a leaf, then request is handled directly. If recipient is a composite, then it usually forwards request to its child components, possibly performing additional operations before and after forwarding (Dinesh Rajput, 2017).

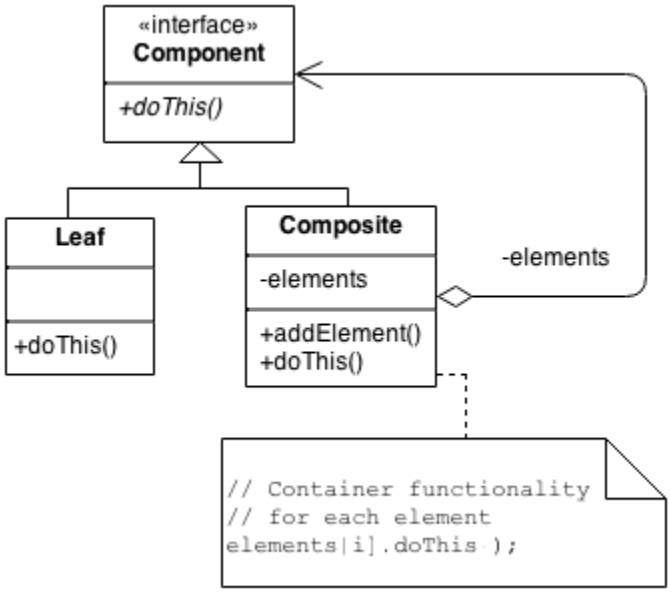

figure 9: Structure of Composite

4.4.1 SCENARIO

A shoes shop is looking to build their own shop management system. The management system will be designed according to Composite Pattern. The system will be developed following the structure of a tree. In the shoes shop will be divided into areas according to different brands (such as Adidas, Nike, Vans, Gucci, etc). Each area will have a certain brand of shoes (ID, style, size, price). Management system according to Compostie Pattern will make product management easier.

4.4.2 CLASS DIAGRAM

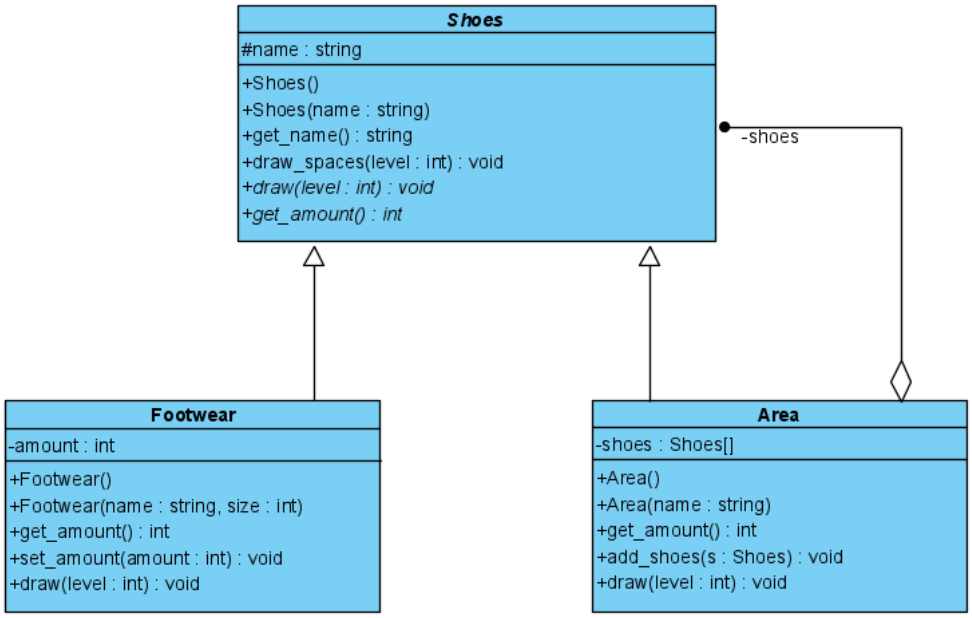

figure 10: Class Diagram for Composite pattern

- This diagram is based on the Composite Pattern. So it includes:
  - Class Shoes is Component. It contains methods to the Footwear and Area classes inheriting.
  - Class Area is Composite. It will manage the Footwear Class through the Shoes Class. Area Class is like a superclass that manages subclasses.
  - Class Footwear is Leaf. It contains child objects of Area Class
  - get_amount () is used to count the number of shoes
  - draw_spaces () is used to represent each area containing the shoes in that area as a description of a folder containing a small file.

4.4.3 ACTIVITY DIAGRAM

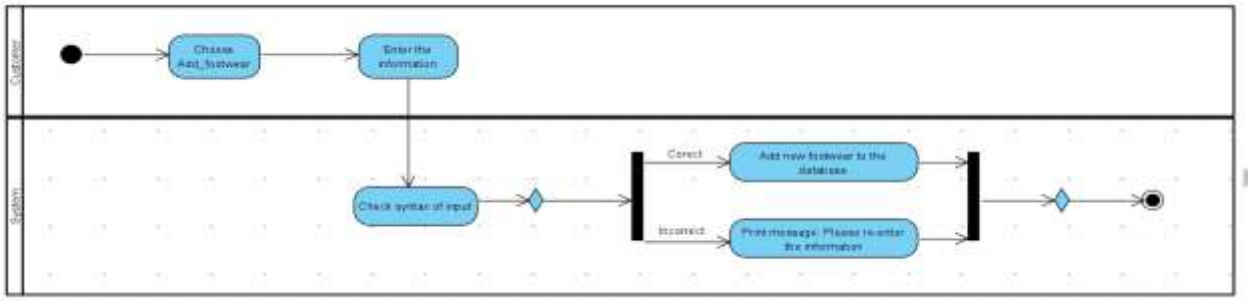

figure 11: Activity Diagram

The figure above shows the activity when the user adds a new footwear in the database with the following steps:

- Select the option Add_footwear.
- Enter new footwear information.
- The system will check if the user's entered information is syntactically correct or not.
- If correct, the system will add new footwear in the database. If it is incorrect, the system will print the message: "Please fill in information !!!".

4.5 BEHAVIORAL DESIGN PATTERN: OBSERVER PATTERN

Observer pattern will define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically. Encapsulate the core (or common or engine) components in a Subject abstraction, and the variable (or optional or user interface) components in an Observer hierarchy (Dinesh Rajput, 2017)

- **Applicability:**
  - When an abstraction has two aspects, one dependent on the other. Encapsulating these aspects in separate objects lets you vary and reuse them independently.
  - When a change to one object requires changing others, and you don't know how many objects need to be changed.
  - When an object should be able to notify other objects without making assumptions about who these objects are. In other words, you don't want these objects tightly coupled.

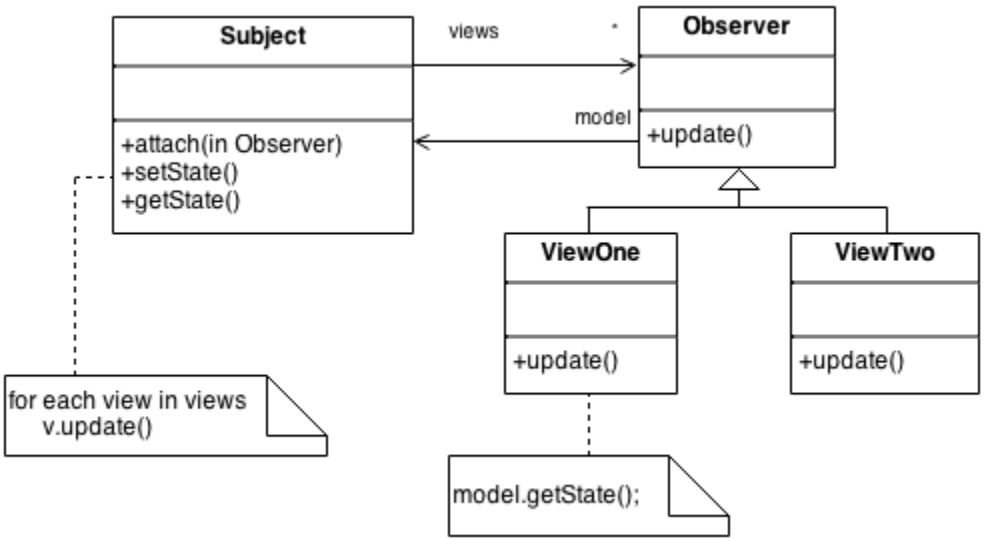

figure 12: Structure of observer

4.5.1 SCENARIO

A gold jewelry store specializing in buying and selling jewelry is designing a table to display the domestic gold rate through two ways: a display board and a bar chart. If the values in one table change, the other changes as well. Thereby, customers can rely on that to make decisions to buy or sell.

4.5.2 CLASS DIAGRAM

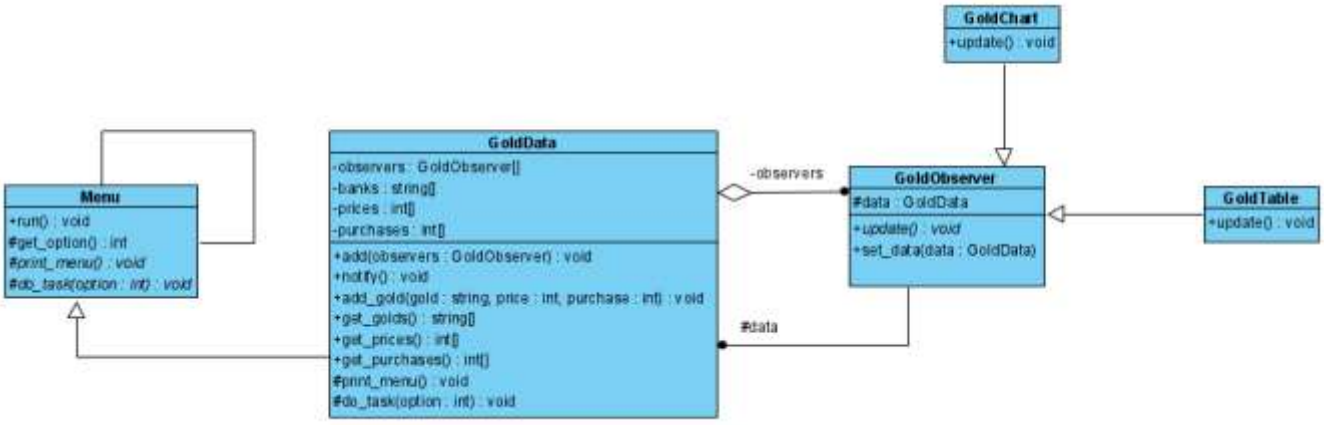

figure 13: Class Diagram for Observer

Subject is the GoldData class: This class includes an attribute which is a vector pointer array to store pointers to 2 observers(Table, Chart), constructor. In addition, it has two additional operators, add() and add_gold(), whose type of transmission is a pointer to the observer. The product class defines attributes about product information such as name, price, purchase and getter for those attributes. In addition, it has the operator notify(), which is used to notify its observers

Observer is GoldObserver class: This class is an abstract class that includes attribute name, the constructor and the notify (), set_data() function whose parameter is a pointer of the subject type. Class GoldObserver is a part-of class GoldData.

GoldChart and GoldTable classes are inherited from GoldObserver class through the update () function.

4.5.3 SEQUENCE DIAGRAM

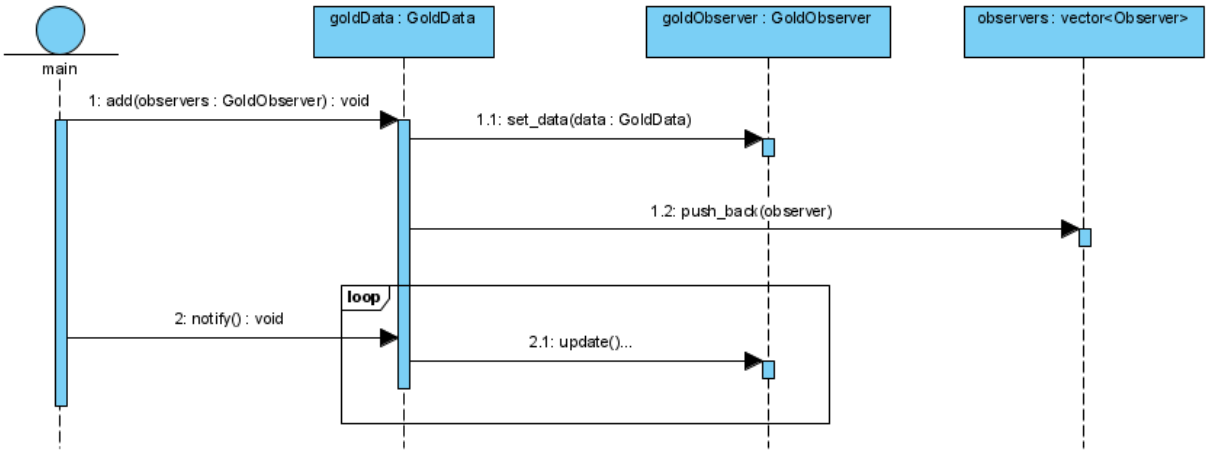

figure 14: Sequence Diagram

In this diagram I describe 4 main functions including main and 3 objects including goldData: GoldData, goldObserver: GoldObserver and observers: vector <Observer>. First, main sends a request to the goldData object with the add (observers: GoldObserver) function to add values to the system. After the add receives the request from main, the system executes the function set_data (data: GoldData). At this time, the system will execute the push_back (observer) function to add the value to the object observers: vector <Observer>. when the object changes, it calls the "notify ()" method to tell the Observers to "update ()", and observer need to get state of subject to update. The update () function is inserted in a loop and is executed by the goldData: GoldData object that sends an update request to the goldObserver: GoldObserver object.

5 DESIGN PATTERN AND OOP

To illustrate more clearly the relationship between OOP and design pattern, see the singleton design example given above.

And now, let's look at the OOP class diagram designed for this scenario:

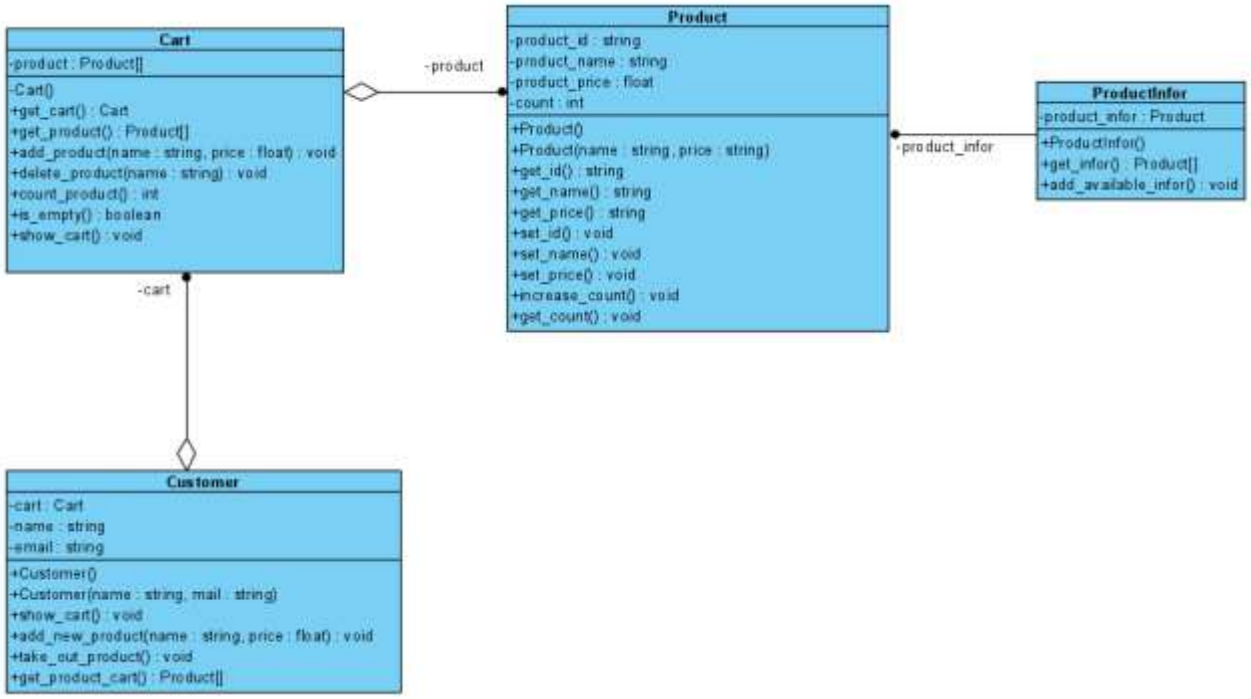

figure 15: Class Diagram OOP for singleton scenario

Here is the OOP class diagram designed for the singleton pattern scenario shown above. We will still have the same classes as the Singleton design, except that now we will no longer have the private static "cart" attribute in the Cart class, which will remove only one cart for each customer as before.

If according to the above design, when we want to create the shopping cart, simply access our constructor and create it, this is quite convenient. But if so, we can create as many shopping carts as we want and this is illogical. We can see this from online shopping page, each customer has only one shopping cart to manage products.

Design pattern is a tried and tested solution to a common programming problem. The design didn't have to be OOP, but it became popular. If you want to build an application, OOP should suffice. But if you want to build a good app, design is the way to go. The "design pattern" is mainly about OOP principles but they are handled separately since they are different objects.

6 CONCLUSION

Through the above article, the definitions and characteristics of object-oriented programming have been presented clearly and in the most detail, together with detailed scenarios for you to have the best overview of the application of object oriented programming for the real world. The UML diagrams used for that scenario also show how the system is run, and the design concepts along with its applications in

programming are detailed. Finally, I also analyzed the relationship between OOP vs Design Pattern and pointed out the strengths of both methods.

REFERENCES

Dinesh Rajput. (2017). *Spring 5 Design Patterns*. Packt Publishing Ltd.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Pearson Education.

Lain D. Craig. (2007). *Object-Oriented Programming Languages: Interpretation*. Springer Science & Business Media.