ASSIGNMENT 1 FRONT SHEET

Qualification	**BTEC Level 5 HND Diploma in Computing**		
Unit number and title	Unit 20: Advanced Programming		
Submission date	June 25, 2021	**Date Received 1st submission**	
Re-submission Date		**Date Received 2nd submission**	
Student Name	Nguyen Tien Hoc	**Student ID**	GCH190844
Class	GCH0805	**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	Hoc

Grading grid

P1	P2	M1	M2	D1	D2

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

Table of Contents

1 Introduction

My team and I have to explain the characteristics of the object-oriented programming model by applying object-oriented analysis and design on a given (hypothetical) scenario. Our scenario must exhibit various characteristics of OOP (such as: encapsulation, inheritance, polymorphism, overriding, overloading, etc.). The second task we have to introduce some design patterns (including 3 types: creative, structural and behavioral) to the audience by giving real situations, the corresponding patterns are illustrated by UML class diagram.

2 OOP general concepts

2.1 Defignition OOP

Object-oriented programming is a method of software development in which the software's structure is built around objects that interact with one another to complete a goal. Messages are passed back and forth between the objects in this interaction. An object can take action in response to a message.
You interact in an object-oriented environment if you look at how your complete things in the world around you. You just start the series of events by using the key to execute the start method of the ignition object. Then you wait for a response (message) indicating whether you were successful or not (Clark, 2013).

2.2 The Characteristics of OOP

2.2.1 Objects

We live in an object-oriented world, as I mentioned previously. You are a thing. Other items interact with you. In reality, you are a data entity with information like your height and hair color. So, what exactly are objects? A structure for combining data and the procedures for interacting with that data is referred to as an object in OOP. If you wanted to include printing capabilities in your app, you'd use a printer object, which is in charge of the data and methods for interacting with printers (Clark, 2013).

2.2.2 Abstraction

When you engage with objects in the real world, you're usually just interested in a small portion of their capabilities. You wouldn't be able to digest the avalanche of information and focus on the work at hand if you didn't have this capacity to abstract or filter out the unnecessary attributes of objects.
When two persons interact with the same item, they frequently deal with a distinct subset of properties as a consequence of abstraction. This idea of abstraction should be considered while creating objects in OOP applications. Only the information that is important in the context of the application is included in the objects. You'd create a product object with characteristics like size and weight if you were making a shipping application. The item's color would be considered irrelevant and would be ignored. Color, on the other hand, may be crucial when creating an order-entry application and would be included as a product object attribute (Clark, 2013).

2.2.3 Encapsulation

Encapsulation is another fundamental characteristic of OOP. Encapsulation is the technique of preventing direct access to data and instead hiding it. You must interact with the object that is accountable for the data in order to have access to it. If you wanted to examine or edit product information in the previous inventory example, you'd have to go through the product object. You would send a message to the product object to read the data. The product object

would then read the value and respond with a message stating what it is. The product object specifies which actions on the product data are possible. If you submit a message to edit data and the product object recognizes it as a legitimate request, it will conduct the action and return a message with the outcome.
Encapsulating data makes your system's data safer and more trustworthy. You understand how the data is accessed and what actions are done on it. This makes program maintenance a lot easier, and it also makes debugging a lot easier. You may also change the methods that deal with the data, and you don't have to change the other objects that utilize the method if you don't change how the method is asked or the sort of answer it returns (Clark, 2013).

2.2.4 Polymorphism
Polymorphism refers to the capacity of two distinct things to react to the same request message in various ways. What does this have to do with OOP? You can design objects with different implementations that react to the same message. For example, you could send a print message to a printer object that would print the text on a printer, and you could send the same message to a screen object that would print the text to a window on your computer screen.
Overloading is a technique used in OOP to achieve this form of polymorphism. Different methods of an object with the same name can be implemented. The object may then determine which method to use based on the message's context (i.e., the number and type of arguments given) (Clark, 2013).

2.2.5 Inheritance
The majority of real-world items may be categorized into hierarchies. For example, all dogs may be grouped together since they share some qualities, such as four legs and hair. Their breeds further divide them into subcategories based on characteristics like size and attitude.
In OOP, you utilize inheritance to organize your objects into groups based on their shared properties and functions. Working with the things becomes easier and more natural as a result of this. It also simplifies programming by allowing you to aggregate generic qualities into a parent object, which can then be inherited by child objects. You may create an employee object, for example, that describes all of the general features of your company's workers. You can then create a manager object that inherits the features of the employee object while also adding features specific to your company's managers. Because of inheritance, any changes to the employee object's characteristics will immediately be reflected in the manager object (Clark, 2013).

2.2.6 Aggregation
When an item is aggregated, it is made up of a collection of other items that function together. Aggregation is a significant element in OOP that allows you to precisely define and implement business processes in your systems (Clark, 2013)

2.2.7 Constructor
In C#, a constructor is a class member. It's a class method that is called when a class object is created. The initialization code is usually placed in the constructor. The constructor's name is always the same as the class's name. A public or private constructor in C# can be used. Multiple overloaded constructors are possible in a class (S. Neeraj, 2020) (3).

2.2.8 Attributes
In C#, attributes are used to transmit declarative information or metadata about different code elements including methods, assemblies, properties, types, and so on. Attributes are introduced

to the code using a declarative tag that is placed on top of the needed code element using square brackets ([]) (GeeksforGeeks, 2019).

2.2.9 Methods

Methods are the blocks of code or statements in a program that allow the user to reuse the same code, which saves memory, saves time, and, most significantly, improves code readability. A method is simply a set of statements that execute a specified task and provide the outcome to the caller. Without returning anything, a method might execute a specified purpose (GeeksforGeeks, 2019).

3 OOP scenario

3.1 Scenario

For our project, we work on a banking software. This software will have the features of trading, saving money, borrowing money. Objects in Bank are admin with customers. In the Bank software, every time a customer wants to know about their information in the Bank, they need to log in to an account of theirs that has been added by the system.

When they use the Deposit Savings feature, they enter an amount that they want to deposit there. The number they enter will be saved in their information, where we use abstraction. We use polymorphism so that the information will be saved and invoked when the user wants to see their information. Because we use encapsulation so that customers can only see their information, not edit it. The software after receiving the amount they deposited into the Bank, it will calculate the amount of interest for a certain period of time in Get Interest at the discretion of the user such as 1 month, 1 year, etc.

When they use the Borrow Money feature, they also have to enter how long they want to borrow for the term. The software will transfer the data to Pay Interest to calculate the interest they have to pay in 1 month. When it is due to return the borrowed amount but has not been paid in full, Borrow Money will send a notice of overdue payment to the screen.
When they want to Check Transaction history and balance, the program will show their transactions and their current balance.

For accountants, they can Check Bill and Check Transaction history and balance of the customer, to use the feature, they must also log in to the account as the customer. When they check the bill, they can see the customer's information.

As for the Admin, they have the Manage Account feature. Manage Account will allow them to add, edit, delete accounts. Manage Account will inherit the class that stores the user's information and calls them to correct or delete their information.

3.2 Usecase Diagram

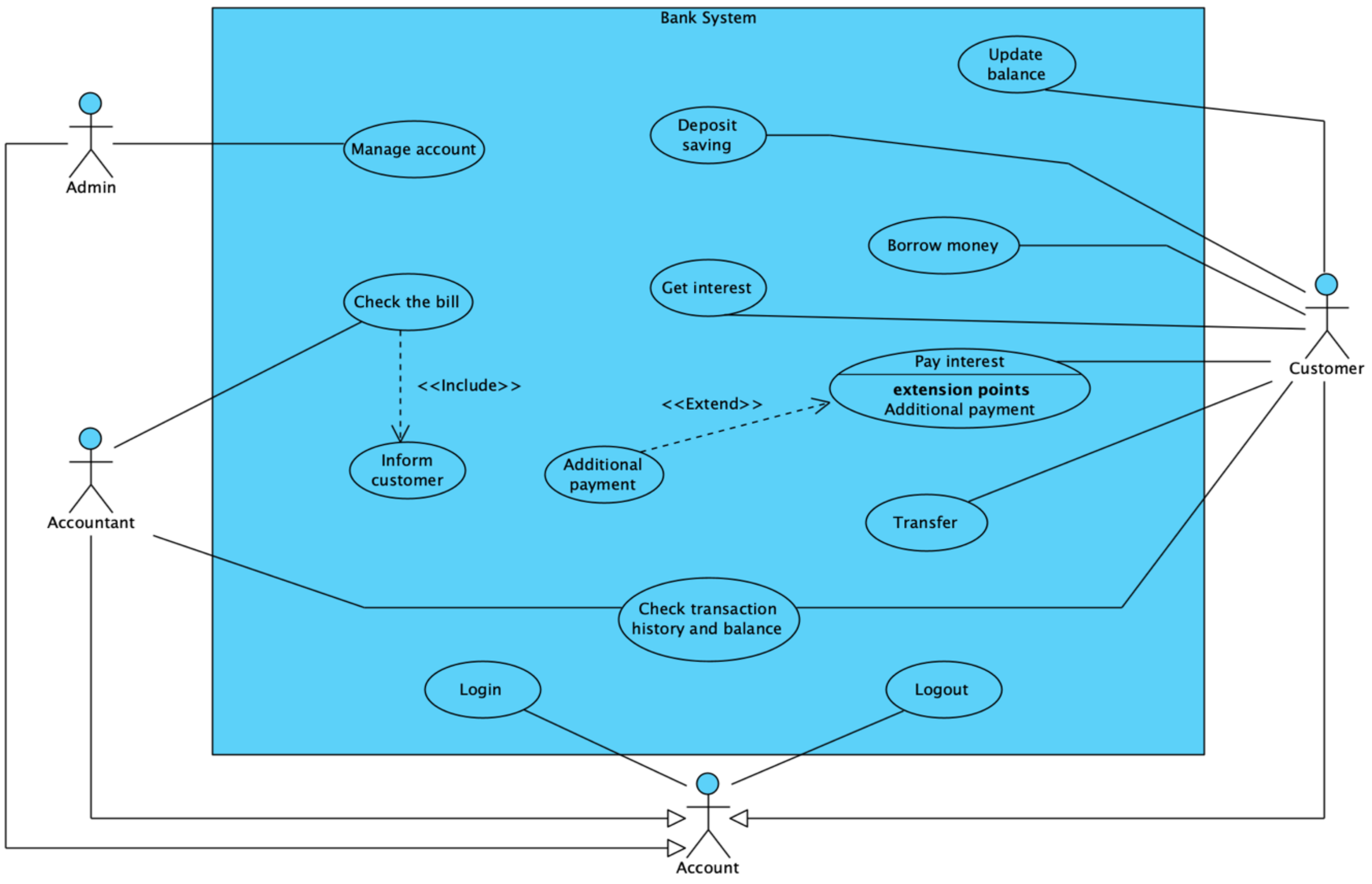

Figure 1: Usecase Diagram OOP

For our project, we work on a banking software. This software will have the features of trading, saving money, borrowing money. Objects in Bank are admin with customers. In the Bank software, every time a customer wants to know about their information in the Bank, they need to log in to an account of theirs that has been added by the system.

When they use the Deposit Savings feature, they enter an amount that they want to deposit there. The number they enter will be saved in their information, where we use abstraction. We use polymorphism so that the information will be saved and invoked when the user wants to see their information. Because we use encapsulation so that customers can only see their information, not edit it. The software after receiving the amount they deposited into the Bank, it will calculate the amount of interest for a certain period of time in Get Interest at the discretion of the user such as 1 month, 1 year, etc.

When they use the Borrow Money feature, they also have to enter how long they want to borrow for the term. The software will transfer the data to Pay Interest to calculate the interest they have to pay in 1 month. When it is due to return the borrowed amount but has not been paid in full, Borrow Money will send a notice of overdue payment to the screen.

When they want to Check Transaction history and balance, the program will show their transactions and their current balance.

For accountants, they can Check Bill and Check Transaction history and balance of the customer, to use the feature, they must also log in to the account as the customer. When they check the bill, they can see the customer's information.

As for the Admin, they have the Manage Account feature. Manage Account will allow them to add, edit, delete accounts. Manage Account will inherit the class that stores the user's information and calls them to correct or delete their information.

Name of Use Case:	Check transaction history and balance		
Created By:	Ho Quang Minh	**Last Updated By:**	Ho Quang Minh
Date Created:	10/06/2021	**Last Revision Date:**	15/06/2021
Description:	With this function, users can check transaction history and account balance		
Actors:	Accountant, Customer		
Precondition:	Users must be logged in or have an account to use this function		
Postconditions:	No post condition		
Flow:	1. User clicks on "Transaction history" or "balance" on the system 2. The system will look up the transaction history or balance of an account in the database 3. After that, the system will display the results corresponding to that account		
Alternative Flows:	3.1 If the balance of that account is lower than the initial limit, a message will be displayed reminding the customer to pay money		
Exceptions:	No exception		
Requirements:			

Name of Use Case:	Check the bill		
Created By:	Ho Quang Minh	**Last Updated By:**	Ho Quang Minh
Date Created:	10/06/2021	**Last Revision Date:**	15/06/2021
Description:	The accountant can check the customer's payment bill		
Actors:	Accountant		
Precondition:	Accountants need to Login first or continue to login		
Postconditions:	Accountants can send information to users so that they know the bill they have paid		
Flow:	1. The accountant presses the "CHECK BILL" button on the system 2. Then the accountant can send the payment bills to the customer 3. The system will receive the request and send a notification to the customer's phone		

Alternative Flows:	2.1 Most bills will automatically be sent to customers, only bills with large amounts of money will be checked and sent by the accountant for the customer to confirm one more time.
Exceptions:	No exception
Requirements:	

Name of Use Case:	Pay interest		
Created By:	Ho Quang Minh	**Last Updated By:**	Ho Quang Minh
Date Created:	10/06/2021	**Last Revision Date:**	15/06/2021
Description:	The customer must pay the interest rate corresponding to the amount borrowed from the bank		
Actors:	Customer		
Precondition:	1. Customers must log in for the first time or continue to log in 2. Customers need to pay the full amount of monthly payment		
Postconditions:	Need to borrow money first		
Flow:	1. Click "Pay interest" on the system 2. Enter the amount you pay 3. Confirm with customer's PIN		
Alternative Flows:	No alternatives		
Exceptions:	2.1 If interest has not been paid in the previous month, additional payment must be made		
Requirements:			

3.3 Class Diagram

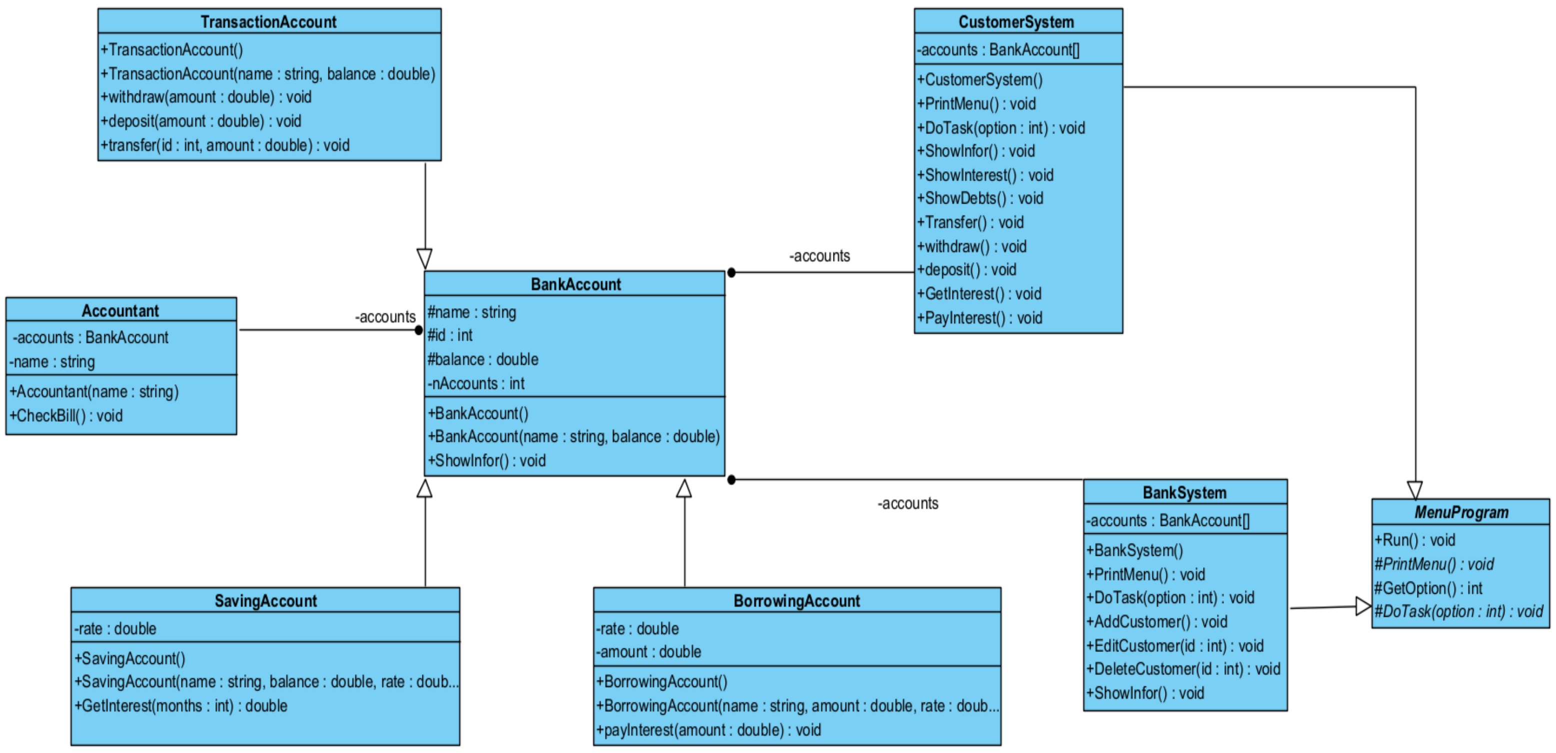

Figure 2: Class Diagram OOP

The program has 8 classes and uses the features of OOP. In the BankAccount class, the customer information properties (name, id, balance) are created in the type of access modifier as protected - this is the encapsulation of OOP. In this class, there are also public functions so that other classes can call and use them directly. nAccounts attribute to store the number of customer accounts, this attribute has the access type is private because it is related to the customer's ID and is the bank's confidential information. The methods set to Get and Set are suitable for object-oriented programming specifically for the purpose of encapsulation, properties with private access type need to be hidden so that users cannot manipulate them directly.

There are 3 classes inherited from BankAccount which are TransactionMoney, SavingMoney and BorrowingMoney. Inheritance is used here for the purpose so that the 3 subclasses can inherit the properties of the BankAccount class, thereby using these data to process and calculate the individual functions of each subclass. The TransactionMoney class needs to use the data of the #name, #balance and #id attribute. Inheritance is also used for the purpose of reusing the functions of the parent class, the CustomerSystem and BankSystem classes inherit from an abstract class that is MenuProgram.

In the abstract class MenuProgram has 2 abstract methods and 2 non-abstract methods Since MenuProgram is an abstract class, two subclasses need to override abstract methods, namely

PrintMenu() and DoTask(). Abstract methods are methods that have no implementation and are used as a basis, when needed, they will be detailed in the derived classes.

Overriding is here so that the methods are more suitable for each class, because the subclasses have different behavior. Polymorphism in OOP is when subclasses override the method inherited from the base class.

4 Design Patterns

Design patterns are the result of experts' documented experience in designing object-oriented software. In object-oriented systems, each design pattern methodically names, describes, and evaluates an essential and recurrent design. The goal is to capture design experience in a form that people can use effectively (Gamma, Helm, Johnson and Vlissides).

Successful designs and architectures can be reused more easily with design patterns. Proven techniques become more accessible to new system developers when they are expressed as design patterns. Design patterns assist you in selecting reusable design alternatives and avoiding alternatives that compromise reusability. By providing a precise explanation of class and object interactions and their underlying intent, design patterns can even improve the documentation and maintenance of existing systems. Simply put, design patterns assist a designer in getting a design "correct" more quickly (Gamma, Helm, Johnson and Vlissides).

A design pattern identifies the fundamental characteristics of a common design structure that make it helpful for building reusable object-oriented designs by naming, abstracting, and naming them. The involved classes and instances, their roles and collaborations, and the division of responsibilities are all identified by the design pattern. Each design pattern focuses on a specific problem or difficulty in object-oriented design. It explains whether it applies, if it can be used due to other design constraints, and the implications and trade-offs of doing so (Gamma, Helm, Johnson and Vlissides).

In summary, Design patterns are repurposed, optimized total solutions to common software design problems that programmers face every day. This is a set of solutions that have been thought about, solved in a specific situation. The system of design patterns is divided into 3 groups: Creational group (5 patterns), Structural group (7 patterns) and Behavioral group (11 patterns).

4.1 Creational pattern

4.1.1 Introduce Creational pattern

Creational design patterns abstract the instantiation process. They assist make a system independent of how its items are produced, constructed, and displayed. An object creational pattern will delegate instantiation to another object, but a class creational pattern will use inheritance to vary the class that is created.
As systems grow to rely more on object composition than class inheritance, creational patterns become increasingly essential. As a result, the focus moves from hardcoding a fixed set of behaviors to defining a smaller set of core behaviors that may be combined to create

any number of more sophisticated actions. As a result, constructing objects with specific behaviors necessitates more than just instantiating a class (Gamma, Helm, Johnson and Vlissides).

- ✓ Sometimes creational patterns
- Prototype: is an early sample, model, or release of a product built to test a concept or process.
- Abstract Factory: is a creational design pattern that lets you produce families of related objects without specifying their concrete classes.
- Builder: is a creational design pattern, which allows constructing complex objects step by step.
- Factory Method: Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.
- Singleton: is a creational design pattern, which ensures that only one object of its kind exists and provides a single point of access to it for any other code.

4.1.2 Description of a Creational scenario

4.1.2.1 Prototype

When a system should be independent of how its products are generated, combined, and represented, use the Prototype pattern.

- when the classes to instantiate are defined at runtime, such as through dynamic loading; or
- to avoid creating a factory class structure that mirrors the product class hierarchy; or
- When a class's instances can only have a few possible state combinations. Installing a matching number of prototypes and cloning them may be more convenient than manually instantiating the class with the proper state each time.

4.1.2.2 Scenario (Prototype)

I will create an application based on a prototype that allows users to reuse existing prototypes for ease of use and we can reuse it many times and follow the form correctly and most complete. Creating a ready-made prototype helps readers easily visualize what information needs to be filled in a given form, and can inherit the layout of the form easily. when the user wants to switch topics or want to write another one.
For example, in my post. I have created a form for the complete letter and through that when the user uses it the user can follow the existing layout that the software has provided so that the user will not have to rewrite the layout when want to write a letter with a different subject.

4.1.3 Diagram and explanation

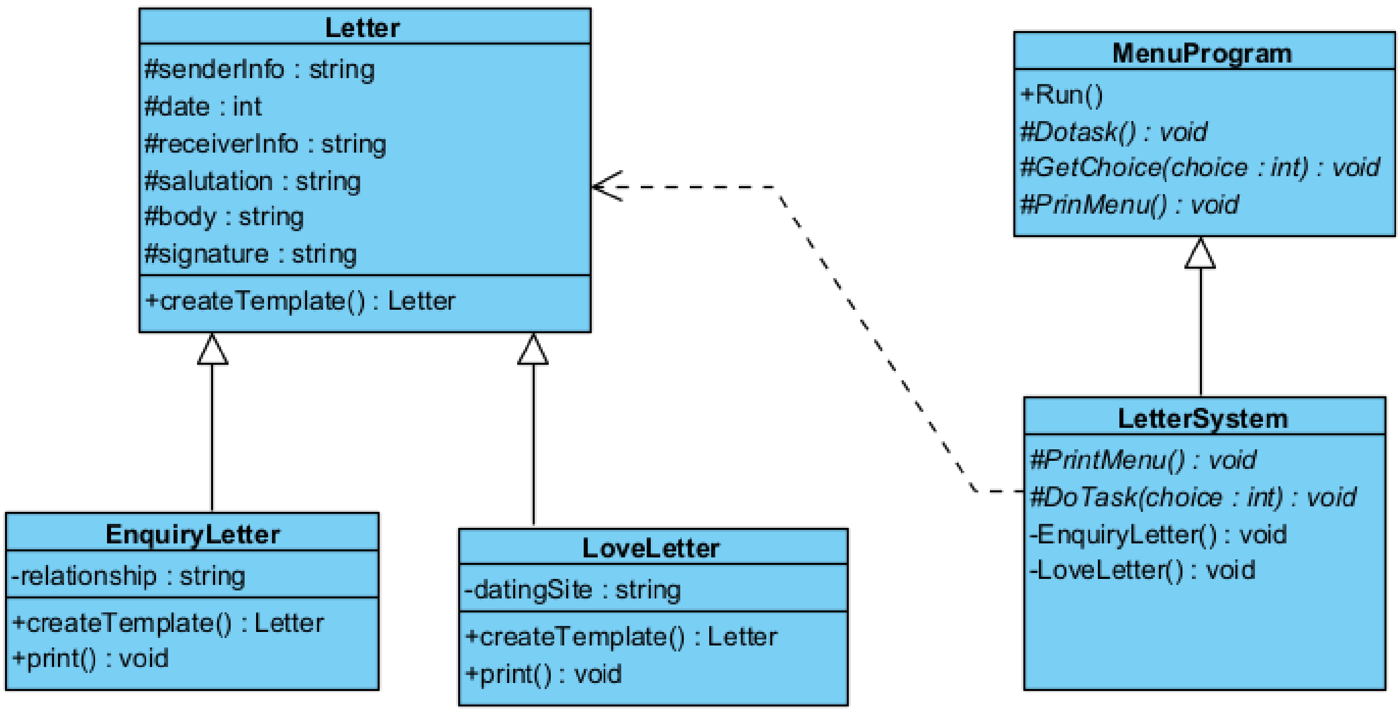

Figure 3: Class Diagram Prototype

In my diagram there are 3 classes: Letter, EnquiryLetter , Loveletter.
The first is the Letter class: I will declare the senderInfo variable and its property as protected and it returns a string and the variable date has an integer return value. Similarly, the receiverInfo, salutation, body and signature variables like senderInfo all return a character string.
second is the EnquiryLetter class: I will declare the variable Who and its property as private and I will have two operations() is createTemplate() and print(). createTemplate() returns the value of the Letter table because I used Generalization so it can inherit the properties from the parent class(Letter) completely. Finally, print() will print out all the information of the parent class (Letter) to the screen to interact with the reader.
Finally there is the LoveLetter class: similar to the EnquiryLetter.

4.2 Structural pattern

4.2.1 Introduce creational pattern

Structural patterns are concerned with how classes and objects are composed to form larger structures. Structural class patterns use inheritance to compose interfaces or implementations. This pattern will make library classes to be developed independently. Inheritance helps libraries to be homogenized through structural patterns (Gamma, Helm, Johnson and Vlissides):

- Adapter: Convert the interface of a class into another interface clients expect. Adapter let's classes work together that wouldn't otherwise because of incompatible interfaces.
- Bridge: Decouple an abstraction from its implementation so that the two can vary independently.

- Composite: Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.
- Decorator: Attach Additional responsibilities to an object dynamically. Decorators provide a flexible alternative to sub classing for extending functionality.
- Façade: Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.
- Flyweight: Use sharing to support large numbers of fine-grained objects efficiently.
- Proxy: Provide a surrogate or placeholder for another object to control access to it.

4.2.2 Description of a structural scenario (Adapter Pattern)

Adapter Pattern (Converter) is one of the Patterns belonging to the structural group (Structural Pattern). The Adapter Pattern allows unrelated interfaces to work together. It saves users from having to redesign interfaces that are incompatible with each other and still work together. It acts as a middleman that takes output from one client and gives it to another client after porting. converted into the format expected by the customer (Gamma, Helm, Johnson and Vlissides).

Scenario: Adapter Pattern (Converter) save users from having to redesign interfaces that are incompatible with each other and still work together. In the previous project, I worked on the battery sales system of a store, this system has a Battery class with the CalculateRevenue interface that I want to take advantage of in my new system, Company. The system works on the Company's sales of products such as chips and screens, I can ShowRevenue for chips and screens. I will use the adapter pattern to solve this problem.

4.2.2.1 Diagram + explanation

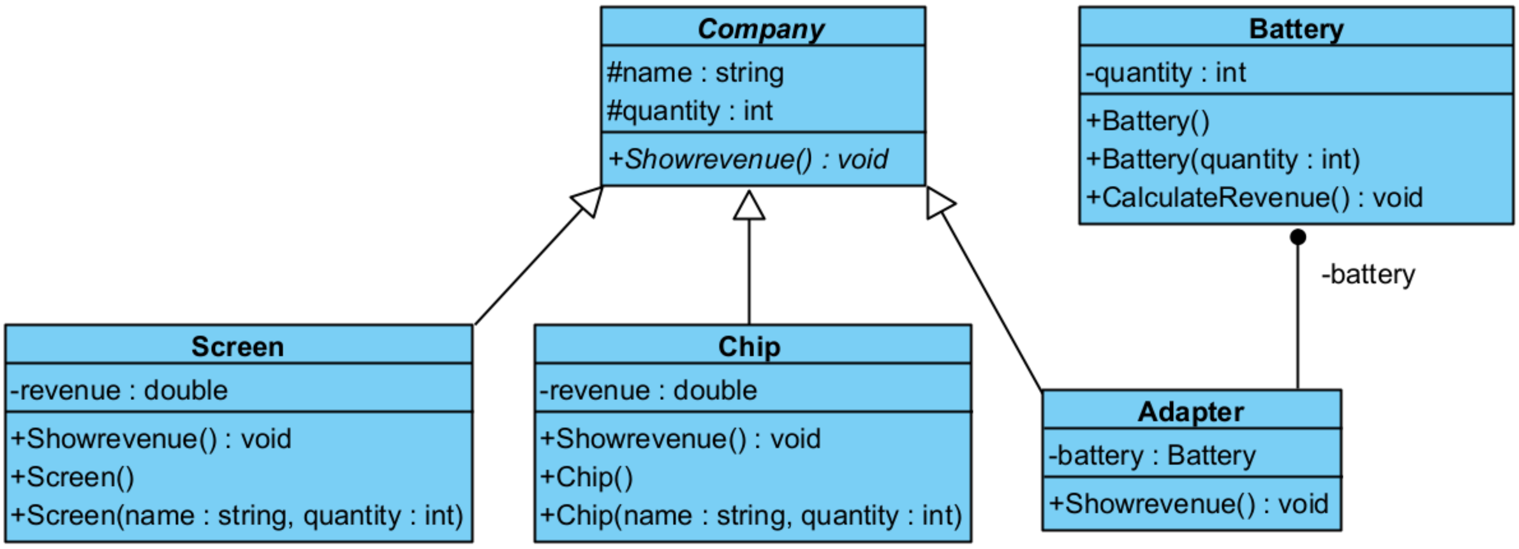

Figure 4: Class Diagram Adapter

Explain class diagram:

- Battery class is a class that has 1 method is CalculateRevenue that shows the average count and methods to implement the default behavior.

- Company class is an abstract class with an abstract method ShowRevenue shows the average quantity and methods to implement the default behavior.
- Chip and Screen classes inherit from Company class. They show the average number of products sold over a period of time.
- Adapter class inherits from the Company class and is associated with the Battery class.

4.2.3 Composite Pattern

Composite: To depict part-whole hierarchies, compose things into tree structures. Clients can use Composite to treat individual objects and object combinations uniformly. Use the Composite pattern when (Gamma, Helm, Johnson and Vlissides):

- user want to represent part-whole hierarchies of objects.
- user want clients to be able to ignore the difference between compositions of objects and individual objects. Clients will treat all objects in the composite structure uniformly.

Scenario: Composite Pattern is a pattern that can perform the same operations on an object or a group of objects. It allows to perform interactions with all objects in the same pattern. My problem is student management. Students in the school have their own information and schedule, each group of students will be placed in classes. Administrators can add or remove students from a class and view the class schedule. In addition, the administrator can view the information of all students.

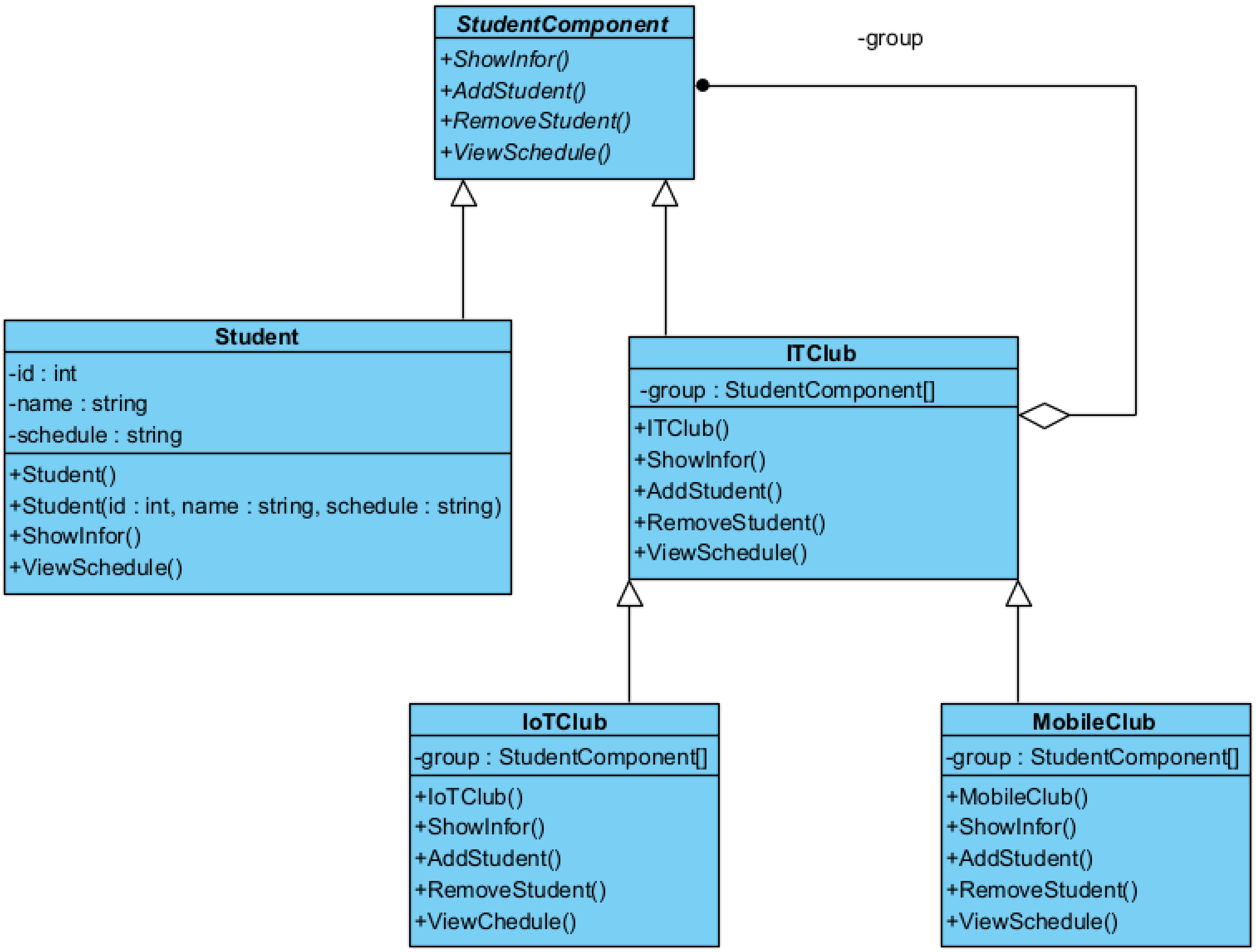

Figure 5: Class Diagram Composite

Explain class diagram: StudentComponent class is an abstract class with 4 abstract methods. This class is used to declare the interface for the objects in the component. And there are methods that implement the default behavior for the interface common to all classes. The other 2 classes that inherit from StudentComponent are the Student class and the ITClub class.

The Student class defines the behavior that can be handled on a Student object. And the ITClub class stores Student objects through the StudentComponent class. The ITClub class implements Student related operations in the StudentComponent interface. The relationship between ITClub and StudentComponent is aggregation because the ITClub class defines a collection of StudentComponent objects. The IoTClub class and MobileClub class inherit from ITClub and use a collection of student groups through the StudentComponent. Finally, the user or client will manipulate the objects in the component through the StudentComponent interface.

4.3 Behavioral pattern

4.3.1 Introduce creational pattern

Algorithms and the assignment of responsibilities between objects are the focus of behavioral patterns. Behavioral patterns encompass not just the patterns of objects or classes, but also the patterns of communication that exist between them. These patterns denote a complicated control flow that is difficult to follow in real time. They divert your attention away from the flow of control, allowing you to focus on the interconnections between items (Gamma, Helm, Johnson and Vlissides).

4.3.2 Description of a behavioral scenario

CHAIN OF RESPONSIBILITY

o Allowing more than one object to handle a request helps to avoid coupling the request's sender and receiver. Chain the receiving objects together and transmit the request down the chain until it is handled by one of them (Gamma, Helm, Johnson and Vlissides).

o Use Chain of Responsibility when:

- A request can be handled by multiple objects, and the handler isn't known ahead of time. The handler should be automatically determined.
- You wish to send a request to one of numerous objects without specifically selecting the receiver.
- The list of objects that can respond to a request should be dynamically defined.

Scenario: I will create an application based on the CoR principle that allows users to look up their test scores and see what levels they will achieve, such as passing the standard score, passing a specific faculty, and getting a scholarship if the score is high and the transcript is high. The user will enter the number of available test scores, and the system will check each item to see how much score can be obtained with that score.

For example, if you enter a test score of 21 and a transcript above 6.0, you will pass because the system's assigned benchmark is 21 or greater and 6.0 academic point or higher. However, you won't get higher faculties because the system's score for higher departments will be 29 points. To be eligible for the scholarship, you must have a grade of 30 and a transcript score of 9.0 or higher.

4.3.3 Diagram + explanation

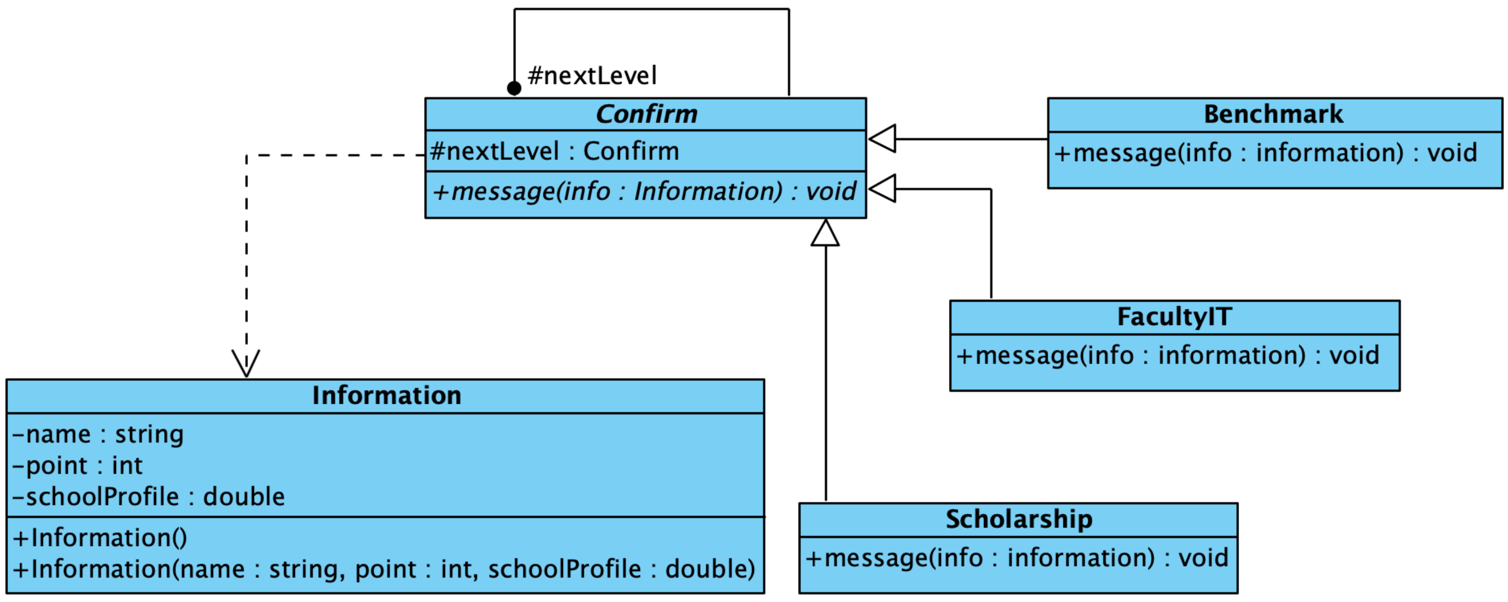
#nextLevel
Confirm
#nextLevel : Confirm
+message(info : Information) : void
Benchmark
+message(info : information) : void
FacultyIT
+message(info : information) : void
Information
-name : string
-point : int
-schoolProfile : double
+Information()
+Information(name : string, point : int, schoolProfile : double)
Scholarship
+message(info : information) : void

Figure 6: Class Diagram Chain of Responsibility

o **Confirm**: In this class, #nextLevel is a self association type (it has a relationship with the class itself), this is #successor in the original solution of the CoR pattern, this function will be used if the Client request it can handle then it will call the message function (which is the HandleRequest() function in CoR) to process, otherwise it will call the successorLevel function itself to forward it to other successors to properly handle the request.

o **Information**: This is the class used for the user to enter the name and score. Then the Confirm class uses the parameters obtained from this class to call the successors that handle the correct number of points. So the Confirm class needs to depend on the Information class to perform the next step.

o **Benchmark, FacultyIT, Scholarchip**: These are subclasses of Confirm class, it overrides Confirm class's message() function to modify the request, because Confirm class and message() function are abstract. In addition, the Scholarship class is the highest class, so the successor can be set to null and the other two classes must call the higher class to handle the request.

5 Design Pattern vs OOP

Design pattern is a technique in object-oriented programming. The problems that the user encounters may come up with a solution on their own, but it may not be optimal. Design Patterns help users solve problems in the most optimal way, providing users with solutions in OOP programming. They are a collection of optimized, proven solutions to problems in software engineering.

A large number of Design patterns are related to OOP but not all. Design patterns are approaches used to create programs. Programmers can still solve problems without Design pattern because OOP is inherently a general design pattern to solve programming problems.

So OOP is the general and Design patterns like sub problems inside. OOP is the foundation for creating Design patterns.

Design Patterns help users reuse code and easily extend the program. Design patterns help avoid potential problems that can cause major bugs, and are easy to upgrade and maintain later. Design patterns are also highly communicative so people can easily exchange code. In short, the use of Design patterns will help users reduce the time and effort to think of solutions to problems that have been solved.
Besides, Design patterns have some disadvantages. Design patterns can be difficult to approach for some inexperienced programmers. And not every problem can be solved with Design patterns so users should not be too dependent on them.

6 Conclusion

Our team explained the features of the object-oriented programming model by applying object-oriented analysis and design on a given (hypothetical) scenario. Our scenario demonstrates various characteristics of OOP (such as: encapsulation, inheritance, polymorphism, overriding, overloading, etc.). Our team introduced a number of design patterns (including 3 categories: creative, structural and behavioral) to the audience by presenting real-life situations, the corresponding patterns are illustrated with diagrams. UML class diagram. We have analyzed the relationship between the object-oriented paradigm and design patterns.

7 Reference

Clark, D., 2013. *Beginning C# Object-Oriented Programming*. Berkeley, CA: Apress.

Gamma, E, Helm, R, Johnson, R and Vlissides, J 1994, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley Professional, Boston.

C-sharpcorner.com. 2020. *Constructors In C#*. [online] Available at: <https://www.c-sharpcorner.com/article/constructors-in-C-Sharp/> [Accessed 25 June 2021].

GeeksforGeeks. 2019. *Attributes in C# - GeeksforGeeks*. [online] Available at: <https://www.geeksforgeeks.org/attributes-in-c-sharp/> [Accessed 25 June 2021].

GeeksforGeeks. 2019. *C# | Methods - GeeksforGeeks*. [online] Available at: <https://www.geeksforgeeks.org/c-sharp-methods/> [Accessed 25 June 2021].