ASSIGNMENT 1 FRONT SHEET

Qualification	**BTEC Level 5 HND Diploma in Computing**		
Unit number and title	Unit 20: Advanced Programming		
Submission date		**DateReceived1stsubmission**	
Re-submissionDate		**DateReceived2ndsubmission**	
Student Name	Pham Dang Linh	**Student ID**	GCH17374
Class	GCH0704	**Assessor name**	

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	xuyen

Grading grid

P1	P2	M1	M2	D1	D2

UNIVERSITY of GREENWICH
Alliance with FPT Education

BTEC

UNIVERSITY of GREENWICH

Alliance with FPT Education

Assignment 1 : ADVANCED PROGRAMING

Group Student:

Nguyen Minh Khanh – GCH17370

Nguyen Huy Ha Xuyen - GCH17380

Pham Dang Linh – GCH17374

Vu Trung Kien – GCH17536

Tran Nam Dan – GCH17205

Class :GCH0704

Tutor : Doan Trung Tung

Contents

I – Introduction

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

II – Object–Oriented Programming (OOP) Introduction

1 – OOP definition and characteristics

According to (Alexander Petkov, 2018), I have conducted to research that object-oriented programming is considered a model of a programming language, in which objects and data are more focused instead of logic or functions. An object in OOP is considered a data field that includes unique attributes and behaviors.

The first thing when implementing OOP is that we need to identify all the objects we want to work with and how they relate to each other. This step is often called the "data modeling" step. Next, once we have identified a specific object, we need to determine the type of data it contains and any logical sequence of commands that can be executed with it. We can call a separate logic sequence as a method and objects can communicate with well-defined interfaces called messages.

There are four principles of object-oriented programming are (Alexander Petkov, 2018) :

Encapsulation

This principle helps the actions or states of objects to be kept private within a defined boundary or class. Other objects will not have direct access to these states, instead, they can only communicate with the object through the methods provided. This feature provides data security for large programs and avoids corruption of data from outside

Abstraction

This principle is considered a natural extension of the principle of packaging. It eases the pressure of maintaining a large code base when objects have to communicate a lot (especially large objects). In other words, this mechanism will hide internal implementation details, which makes it easier for developers to make changes and additions over time.

Inheritance

In OOP, objects will often be very similar, sharing common logic, but not quite the same. Therefore, this principle helps us to reuse common logic and extract it into a separate class. This means that the programmer will create a class (child) and derive from another (parent) class and they will have a hierarchy.

This attribute of OOP forces programmers to analyze data more thoroughly and ensure a higher level of accuracy

Polymorphism

This principle provides a way to re-use the same classes as the parent class without having to confuse the mix types. In other words, the objects are allowed to perform more than one form, then the program will determine which uses are needed for each execution, to reduce the need to copy the code.

2 – Advantages and disadvantages of OOP

Basically, OOP operates in a form that focuses on objects rather than necessary logic rules. This operation is very suitable for large programs with many different entities and is constantly updated. In addition, OOP has several other benefits such as code re-use, scalability and efficiency. However, OOP has also been criticized for many reasons, one of the biggest reasons is that overemphasizes the data component of software development and does not focus enough on computation or algorithmsIn. In addition, OOP will be more complicated to design and more time-consuming to compile

III – Assumed Scenario for OOP

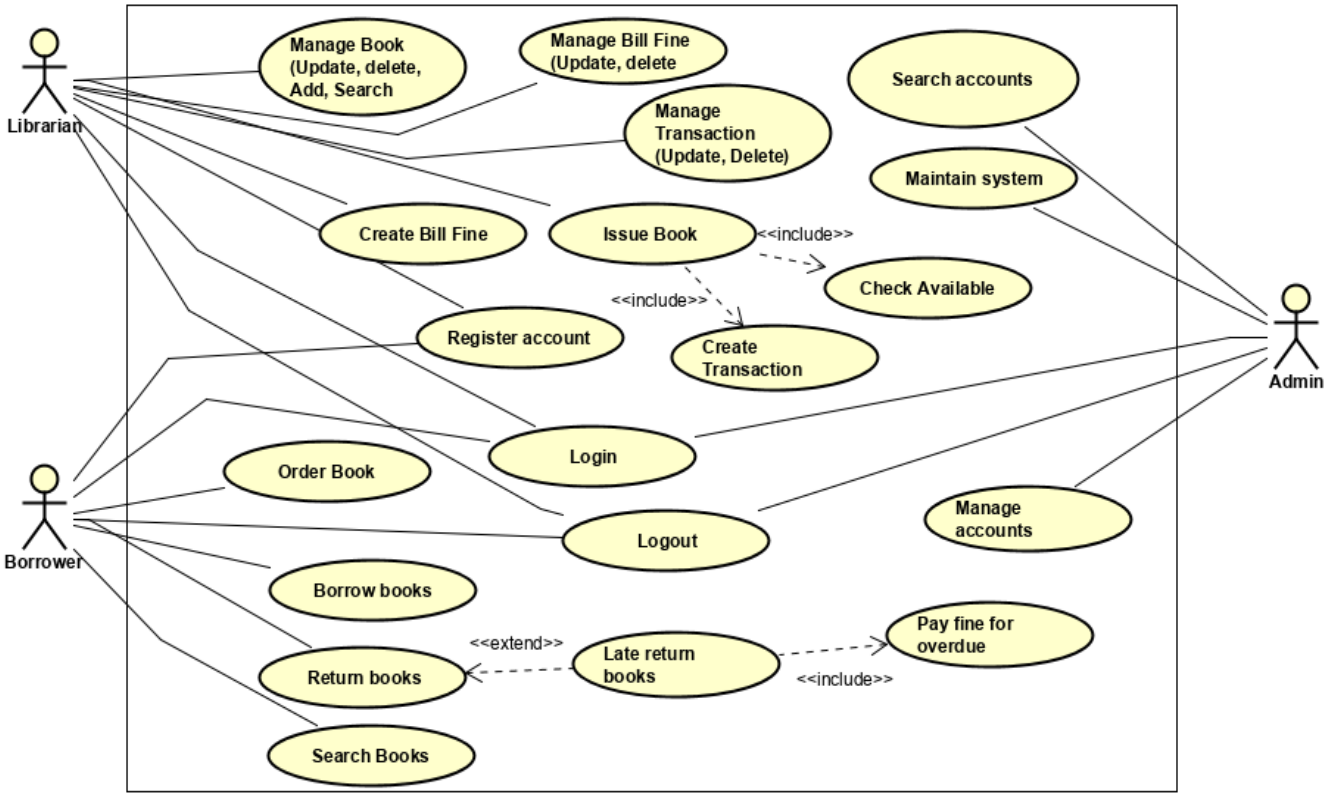
1 – Main script

We want to use the university library system as an example for OOP. This system contains 3 actors: borrower, librarian and admin. The borrower has abilities to login, search books, borrow books, order books, return books and pay fine if borrow books overdue. The librarian has abilities to login, search books, insert data, check for browed book and statistic borrowers. The admin has abilities to login, search for all users' information and manage them.

2 – UML diagrams

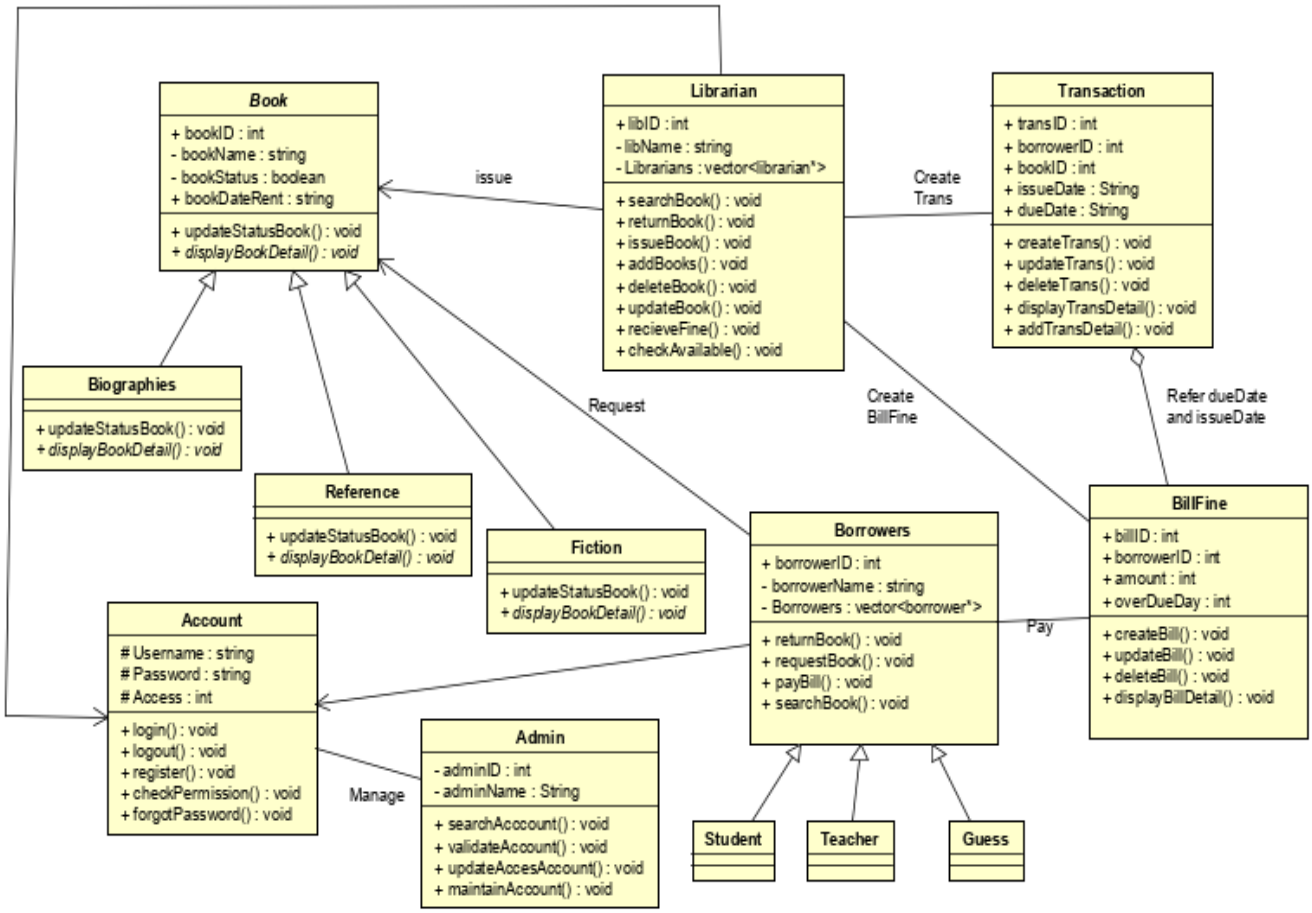
UML is an abbreviation for Unified Model Language, it is an approach to modeling documents. It works based on diagram representations of software, a type of UML that can be used to replace diagrams. They provide a way to model work as well as a wider range of features to improve readability and efficacy. There are many types of UML diagrams, each of which is used for different purposes. The two most broad categories that encompass all other types are Behavioral UML diagram and Structural UML diagram. There are also many other types of small diagrams in these two main UML categories. In this report, I will design some typical diagrams.

2.1 Usecase diagram



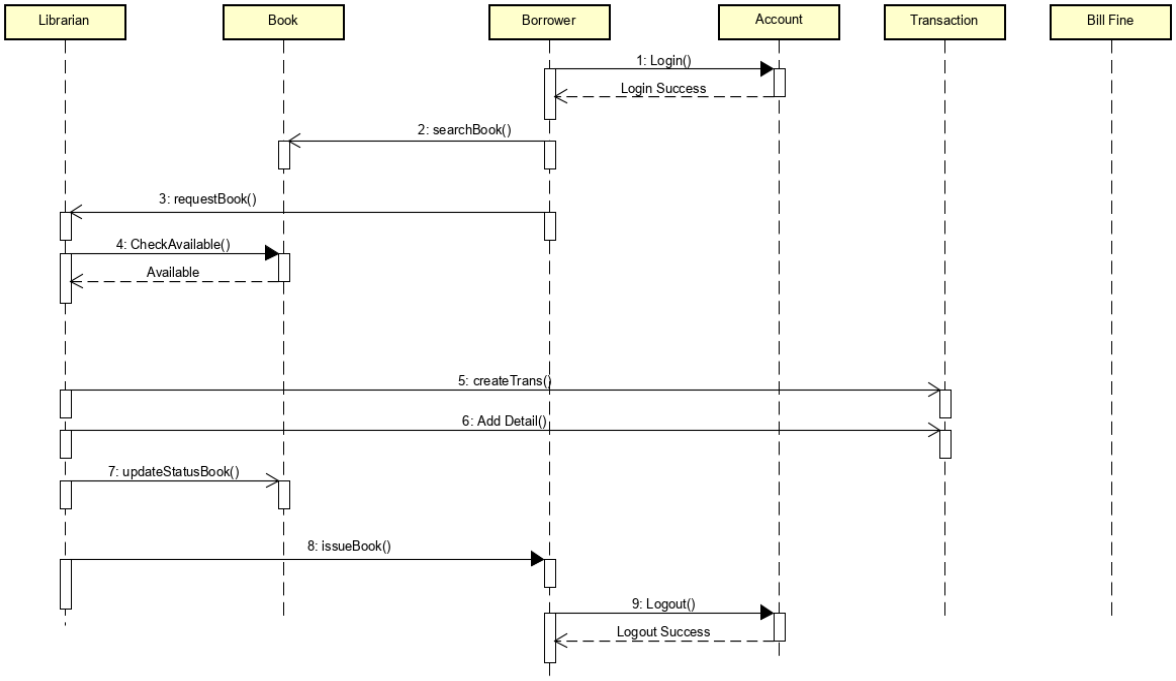
The diagram above includes three main entities: borrowers, librarians and admin. Admin will be responsible for managing the entire system and managing all accounts. He also has the right to check account information of librarian if necessary. Librarians and borrowers who want to use the library will have to have an account, log into the system to be able to search for books

2.2 Class diagram

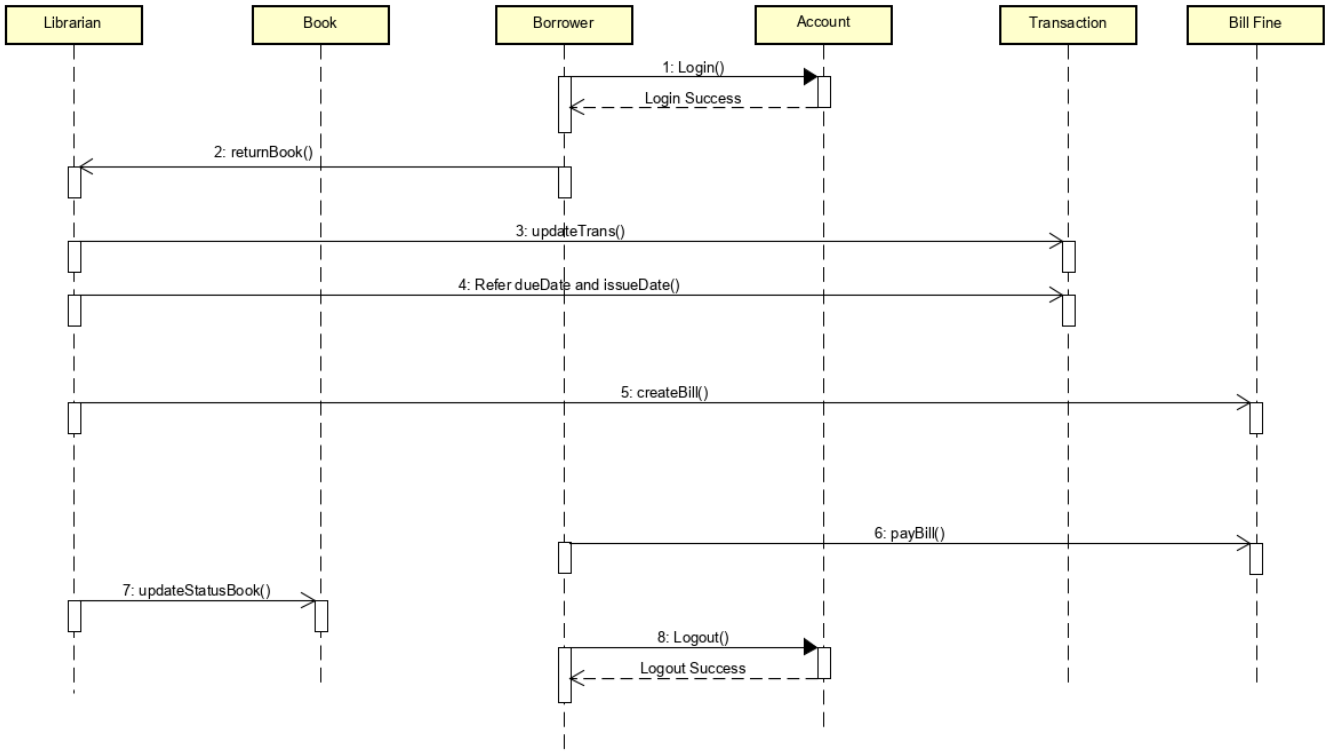


2.3 Sequence diagram

Borrow Book Diagram



Return Book Late Diagram



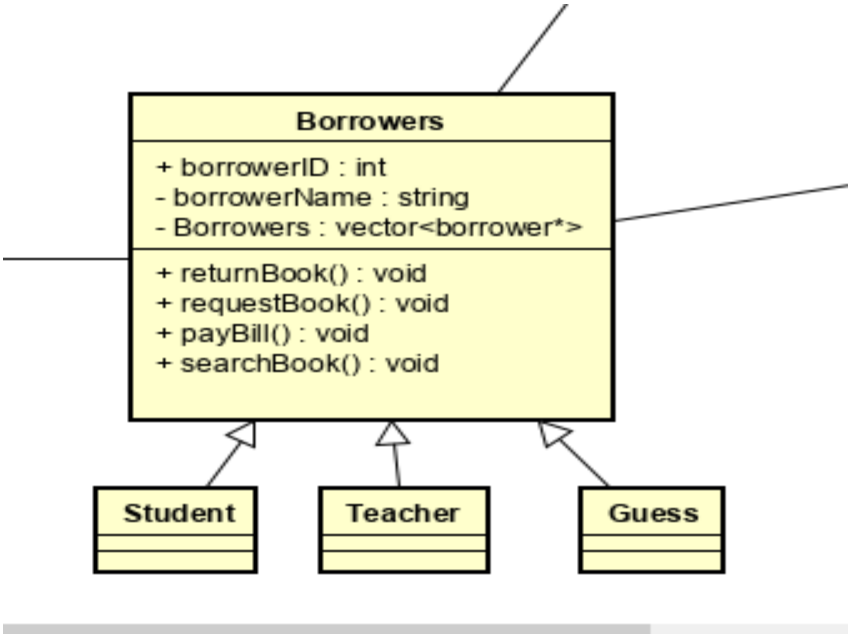
3 – OOP characteristics analysis

Encapsulation analysis

Encapsulation is most clearly expressed in this case, three separate entities, borrowers, librarians and admin will have different special functions. For example, borrowers or librarians will not be able to access the system or view account data at will.

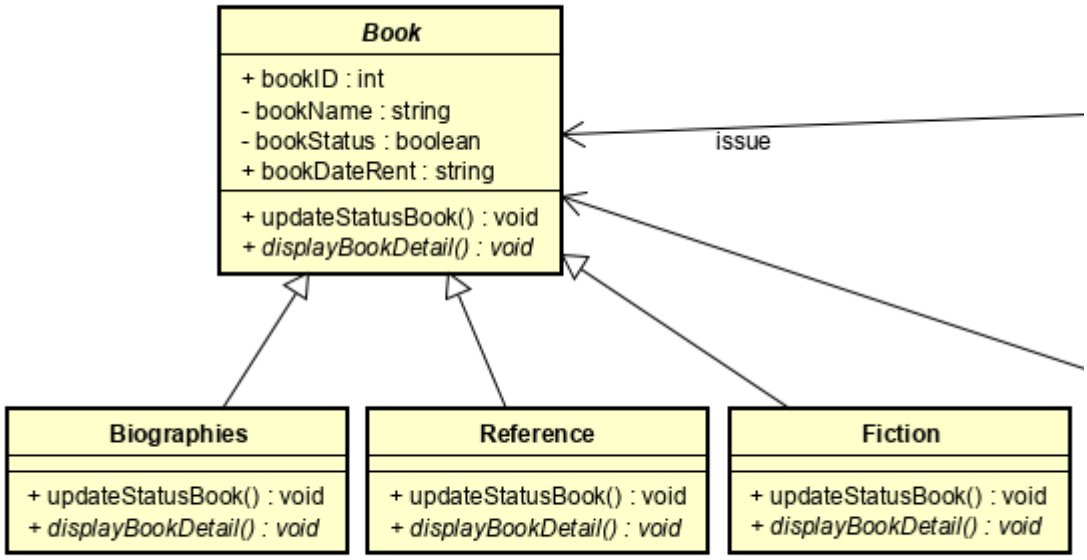
Inheritance analysis

This relationship is also known as the "is a" relation, in this attribute, the subclass inherits all properties from the parent class. As in the example below, borrowers can be anyone of three objects, guests, teachers and students. And each object has the right to inherit rights such as renting books, return books and paying bills if they are overdue



Abstracts and Polymorphism analysis

Polymorphism is clearly shown in the "Book" layer, which has the function "displayBookDetail ()" and “updateStatusBook()”, meaning that each book will carry this function. But depending on the type of book, this function will work differently because each book has a different characteristic. The book class will be the abstracts class and the displayBookDetails() is abstracts method.



IV – Design Pattern Introduction And Assumed Scenarios

1 – Design pattern definition and catalogs

Definition

According to (Erich Gamma et al, 1994), I have researched that design pattern is a design pattern that describes a problem that happens many times in our environment. Since then it has designed a common solution for that problem. Each model has a core solution specification applied. In addition, It describes how to solve a problem that can be used in many different situations.

Based on (Erich Gamma et al, 1994), design pattern has been divided into three main catalogs, which are:

Creational

These design patterns help to abstract the initialization process. They make the system independent of its objects. A class creational pattern uses inheritance to vary the class that's instantiated. and meanwhile, this design pattern will authorize initialization to another object. It will be important when developing systems depend on object components rather than inheriting old classes. Therefore, creating objects with specific behaviors will require more than just starting a class. Here are the designs of Creational

- Abstract Factory
- Factory Method

- Object Pool
- Prototype
- Singleton

Structural

This design pattern uses inheritance to compose settings or interfaces. Structural patterns are concerned with how classes and objects are composed, from that it will form larger structures. These templates include all Class or Object components. Besides, instead of composing interfaces or deploying. They describe ways to compose objects so that a new function can be identified. Here are the designs of Structural

- Adapter
- Bridge
- Composite
- Decorator
- Facade
- Flyweight
- Private Class Data
- Proxy

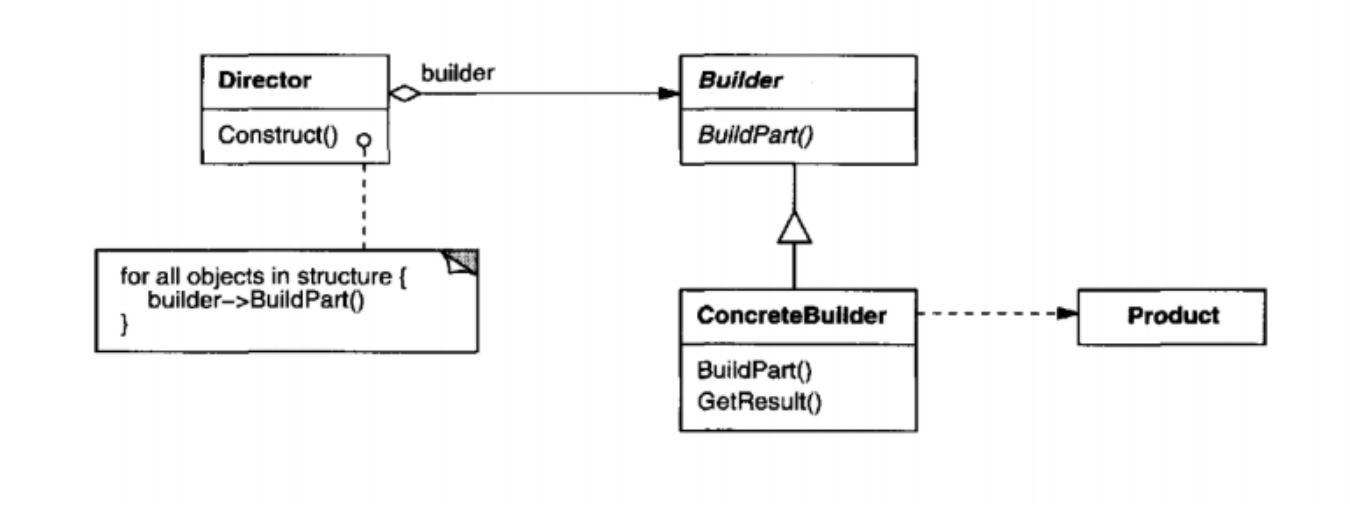
Behavioral

This design pattern related to algorithms and assignments between objects. It not only describes the patterns of objects but also focuses on the communication between them. The characteristic of this design is that the control line is complex and difficult to track at runtime. Behavioral class patterns use inheritance to distribute behavior between classes. Here are the designs of Behavioral

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

2 – Builder design pattern and assumed scenario

2.1 Builder design definition



Builder separates the construction of a complex object from its representation so that the same construction process can create different representations. This design is used to analyze a complex representation, creating one of several goals. Here are some participants of Builder

Builder (TextConverter)

Specifies an abstract interface for creating parts of a product object.

ConcreteBuilder (ASCIIConverter, TeXConverter, TextWidgetConverter)

Perform the assembly of parts through the Builder, identify and track the representatives that it creates

Director (RTFReader)

Constructs an object using theBuilderinterface.

Product (ASCIIText, TeXText, TextWidget)

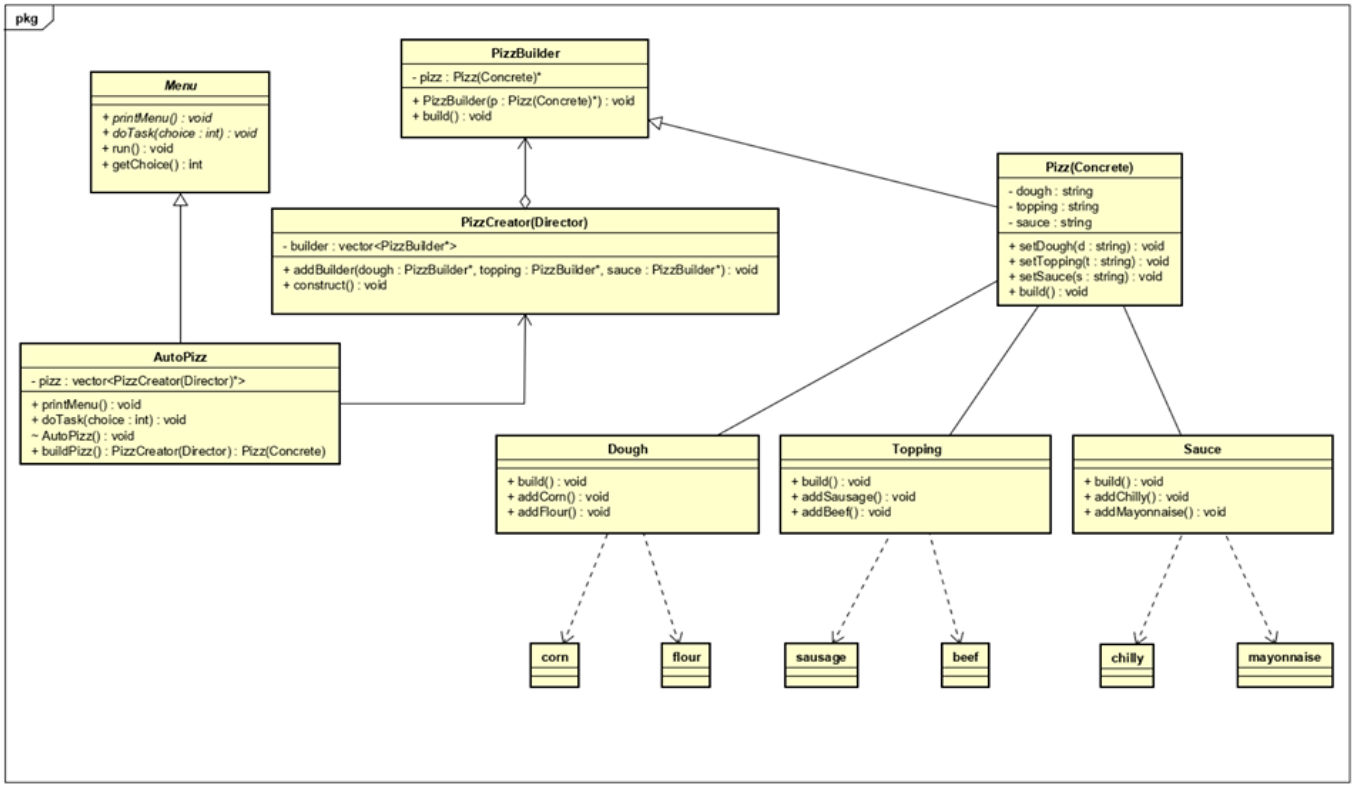
It represents the complex object under construction. ConcreteBuilder builds the product's internal representation and definesthe process by which it's assembled. And includes classes that define the constituent parts, including interfaces for assembling the parts into the final result.

2.2 Assumed scenario

Currently fast food is becoming more and more popular thanks to its usability, especially pizza. Therefore, the demand for pizza is increasing, which leads to people having to spend more time waiting to buy a pizza, along with expensive travel expenses if the pizza shop is too far.

To solve this problem, AutoPizz was born with the purpose of bringing the most convenience. AutoPizz is a series of automated pizza production and sales systems according to customer needs. This system will be located in many places to provide maximum convenience both in time and cost. First, the system will display the ingredients to make pizza from bottom to top in order (dough, topping, sauce). After that, customers will be free to choose materials suitable for their desired pizza. Finally, after choosing, customers will press the word "pizza making" to start the pizza making process

2.3 UML class diagram



Above is the UML Class Diagram diagram of the AutoPizz system. The system works according to Builder design in Creational. It describes how to create a pizza from user-selected ingredients. The system consists of eight main entities:

Menu : This is an abstract class, used to create menus and execute tasks

AutoPizz : This class is used to display menu lists, as well as other necessary functions for customers to use

Pizz : In the Builder method, this is also called the Director class, which has the function of creating a complete pizza

PizzCreator : This is a class called Concrete Builder, which has the function of combining ingredients to make a pizza, monitor process through the Builder class

PizzBuilder : This is Builder class, specifies an abstract interface for creating parts of a product object.

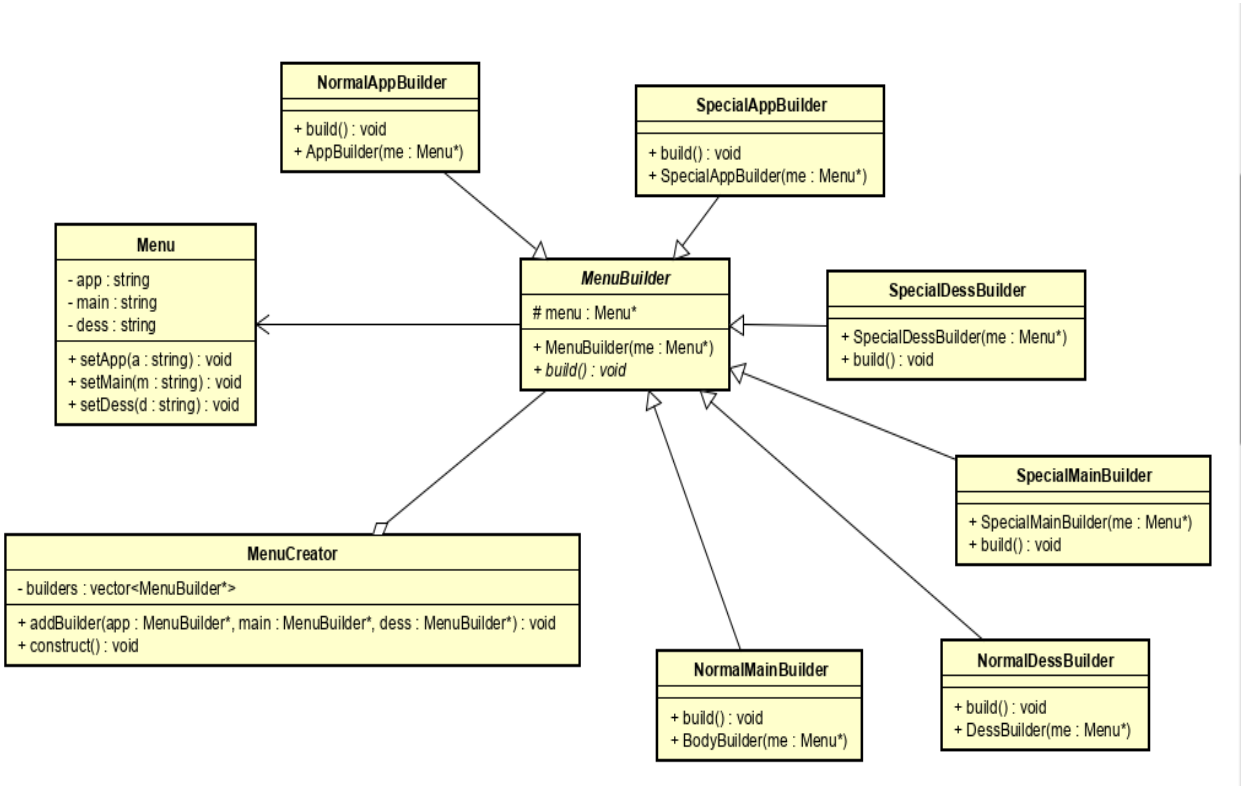
Ingredients : This layer includes products, here is understood as ingredients of pizza including dough, topping and sauce. In each product there will be smaller components to diversify the user's taste

3 – The second builder design pattern and assumed scenario

3.1 Assumed scenario

A restaurant divides their menu into two separate menu lists. An ordinary and special menu includes 3 dishes: appetizers, main courses, and desserts, and it's just different about the dishes in it. They want us to create a software that helps customers choose Menu by entering from the keyboard. We decided to use Design Pattern with Builder Pattern to create a user interface built from small layers. Below is the program's Class Diagram:

3.2 UML class diagram



In my Class Diagram there are 9 classes they include:

MenuBuilder: Is an abstract class, it's includes protected Menu, build() function, and public MenuBuilder. Get referenced by Aggregation relationship with MenuCreator. In other words MenuCreator is part of MenuBuilder.

Menu Class : Includes 3 private string properties: app, main, dess. And setApp functions, setMain, void setDess. Has a-has relationship with MenuBuilder.

NormalAppBuilder, NormalMainBuilder, NormalDessBuilder, SpecialAppBuilder, SpecialMainBuilder, SpecialDessBuilder class: These three classes all have inheritance relations with MenuBuilder because they use the build method of this class.

They all include the build () method, which is inheritance from MenuBuilder.

MenuCreator Class: Includes a private vector that points to MenuBuilder, 2 public addBuilder and construct. This class has part of MenuBuilder thanks to Aggregation relationship.

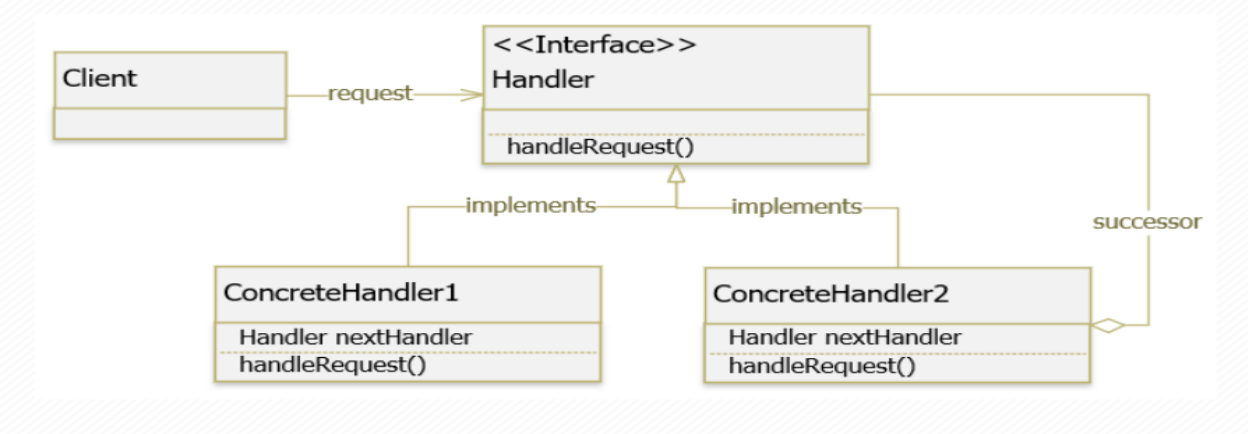
4 – Chain of responsibility design pattern and assumed scenario

4.1 Chain of responsibility definition

Definition

Chain of Responsibility is a behavioral design pattern that lets you avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.

Using Chain of Responsibility



Handler : It declares interface. It contains one or more method(s) for handling request

ConcreteHandler : It contains the code that executes requests. Upon receiving a request each handler must decide whether to process it or pass it to the next Handler.

The Client : create requests and the requests will be sent to the chain

Why we use Chain of Responsibility

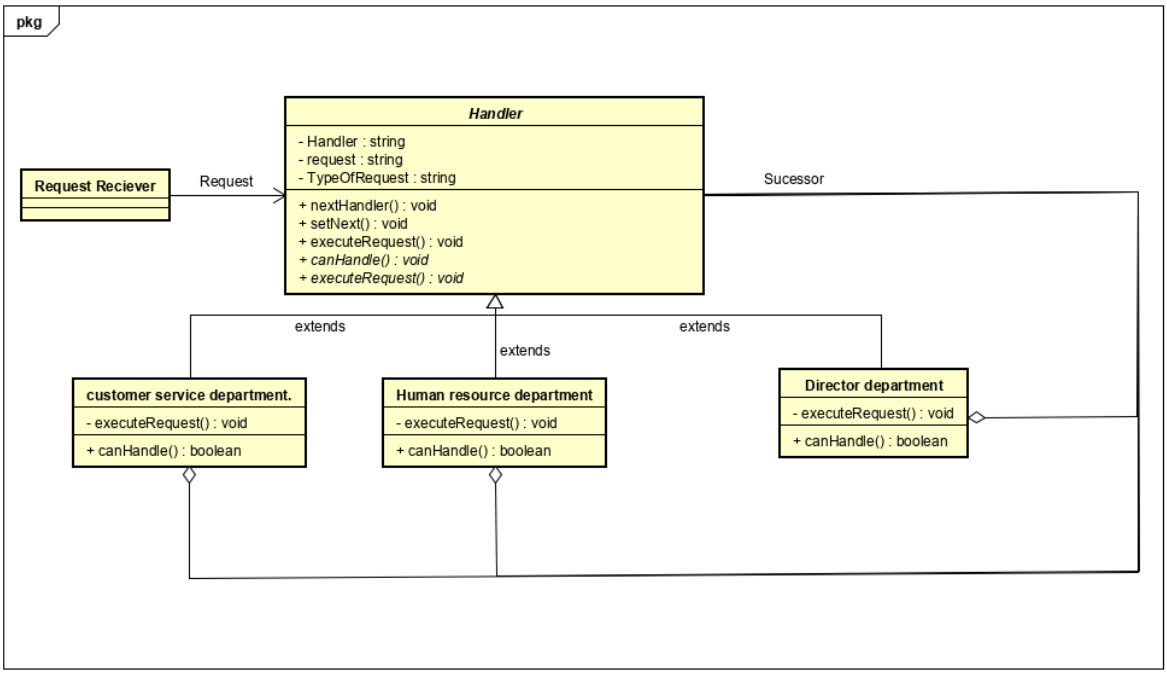
By using the Chain of Responsibility, we can control the order of request handling, easy modify our app by adding new handlers into the app without breaking the existing client code and it devides responsibilities for objects: make sure Single Responsibility Principle.

4.2 Assumed scenario

In programing, we quite familiar with if- else command. This command is simple and easy to use. But let's imagine we have a scenario like this: A client comes to a company and give a request. Not every request can be handle by 1 department, but each department con only handle a field of work. If the request is about investment, it will come to director department, if the request is a job applicant, it will come to Human resource department, if it is a complaint or question, it will come to customer service department.

When we have a situation like this, it's instinctive to write a massive lines of if- else command code. But the problem with this kind code is the more ladder you add to the code, the more complexity you are increasing for your code and it will become very hard to debug or actually read it. One of our job in writing code is to let other can understand it and make modification to it. If we write code that people are not able to read it, it will really hard to maintain. So one of the solution is using Chain of Responsibility.

4.3 UML class diagram



In this UML, I have created a request Receiver, where we receive the request from customer. The request will be sent to 1 of 3 Handler: Human resource department, Director department or Customer service department. Base on the type of request, each handler will analyze if it can handle the request or not, if

not the request will be passed to the next Handler. And if the handler can handle the request, it will execute the request.

5 – Adapter design pattern and assumed scenario

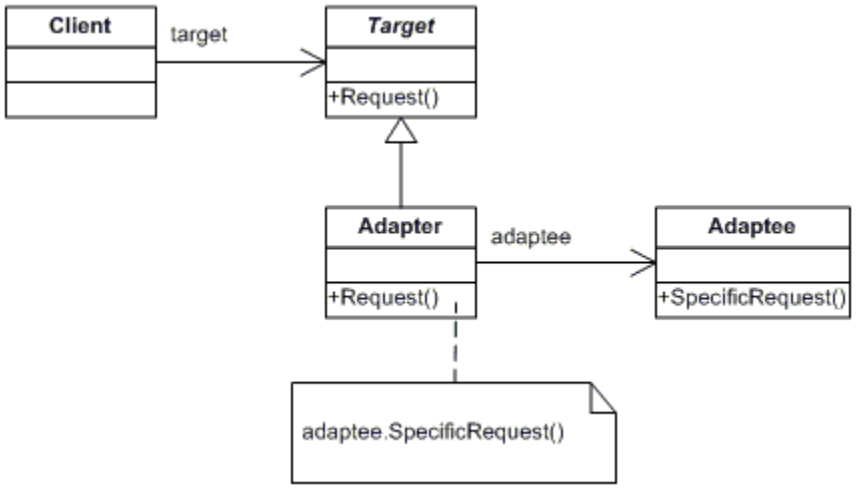
5.1 Adapter design pattern definition

Definition

The adapter pattern convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

Why we use

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.



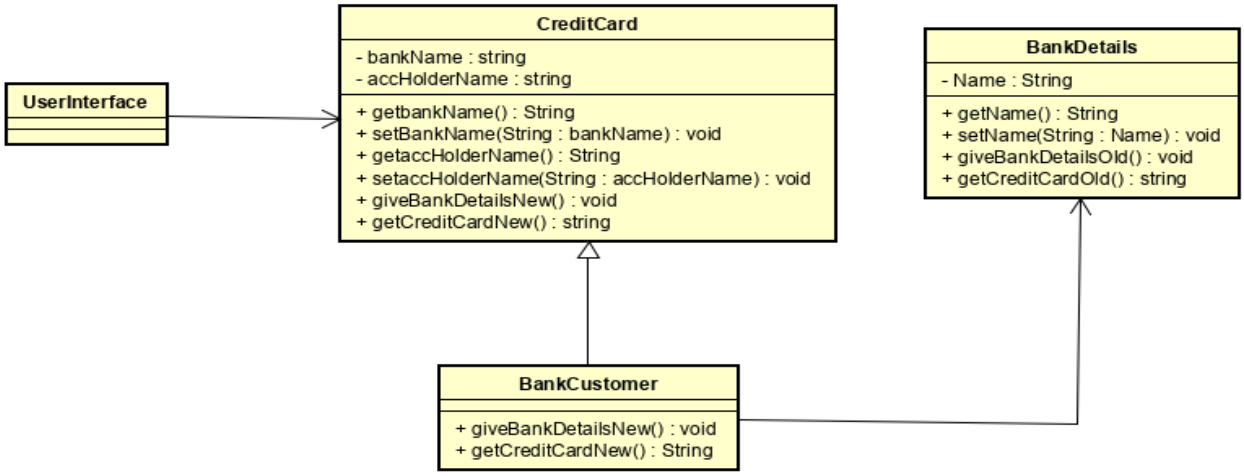
5.2 Assumed scenario

Scenario: A client bank account management application. Previously this application used user name to identify customers but now the company wants to upgrade the application so it is more clearly defined.

Problem: This application uses the account name to identify customers but only with these numbers will it be difficult to determine who is the owner of this account number and which bank type.

Solution: We will upgrade the application, divided into 3 parts: User name, account number and bank name. We need to use the Adapter to ensure that after the upgrade, the application and the old data will still be executed

5.3 UML class diagram



CreditCard (Target Interface): This is the desired interface class which will be used by the clients.

BankCustomer: This class is a wrapper class which implements the desired target interface and modifies the specific request available from the Adaptee class.

Bank Details: This is the class which is used by the Adapter class to reuse the existing functionality and modify them for desired use.

6 – Singleton design pattern and assumed scenario

6.1 Singleton design pattern definition

Definition

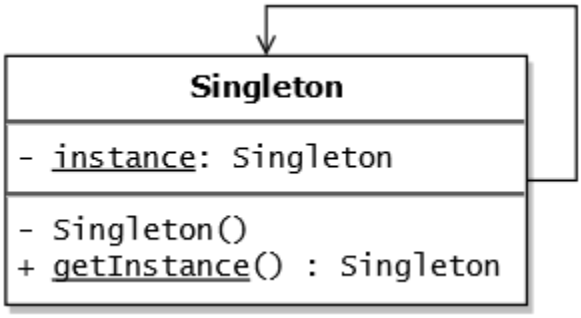
Singleton pattern is one of the simplest design patterns. Singleton Pattern is a design pattern is used to ensure that each class can only be an instance only and all interactions are through this expression. Example: file system, file manager, window manager, printer spooler.

This pattern involves a single class which is responsible to create an object while making sure that only single object gets created. This class provides a way to access its only object which can be accessed directly without need to instantiate the object of the class.

Why we use

When we see that how to initialize an instance of a class of very expensive performance and costs; just one and only one instance was enough for use in an application's lifecycle. Be used to design/create the

Logger object, Cache, Connection Pool, Thread Pool, ... (to be created once, no need to create one or more other instance to use). Be used in the template design pattern such as: Builder, Prototype, Facade, Abstract Factory, State ...



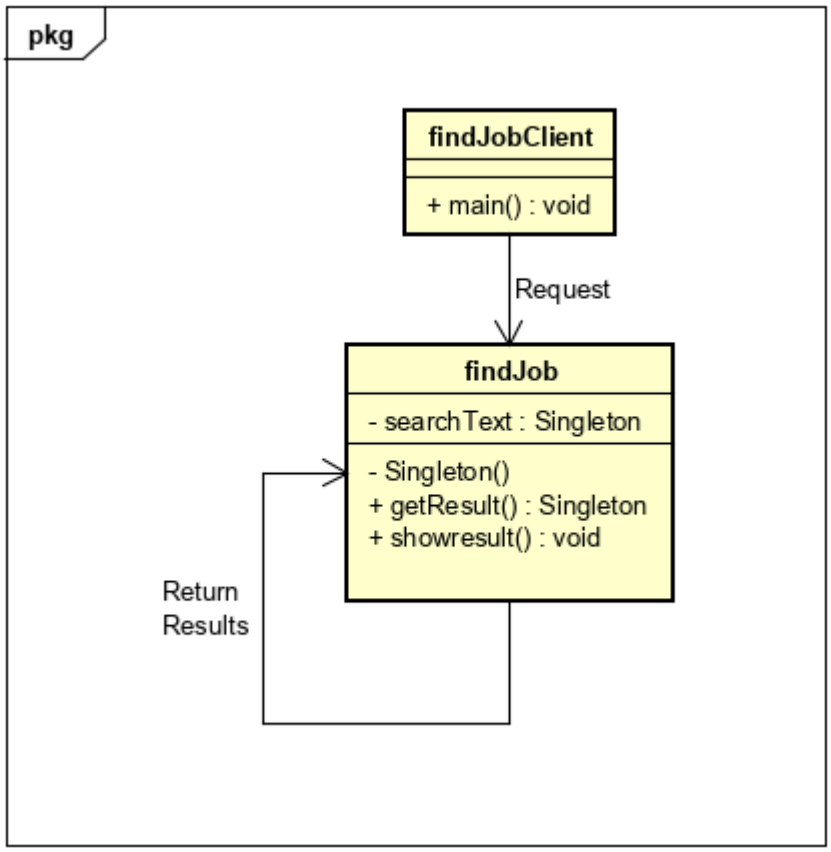
6.2 Assumed scenario

Scenario : Suppose I was director of a company Search Jobs. Our company employees will have to provide information about the current job for customers.

Problem : In the past, our old system can only store data on employment. If employees want to get data to serve customers must search manually. This is difficult when our system possesses a huge amount of data

Solution : Create a automatic job search function for our system

6.3 UML class diagram



V – Conclusion

The above report summarizes the content of object-oriented programming and design patterns including definitions, characteristics and classification. In addition, the report also includes a number of assumed scenarios and UML diagrams so readers can easily visualize the usage and importance of each feature of OOP and design patterns.

VI. Slides


Advanced Programming
Tutor : Doan Trung Tung
Assignment 1
Group Member :
Nguyen Minh Khanh
Nguyen Huy Ha Xuyen
Tran Nam Dan
Vu Trung Kien
Pham Dang Linh

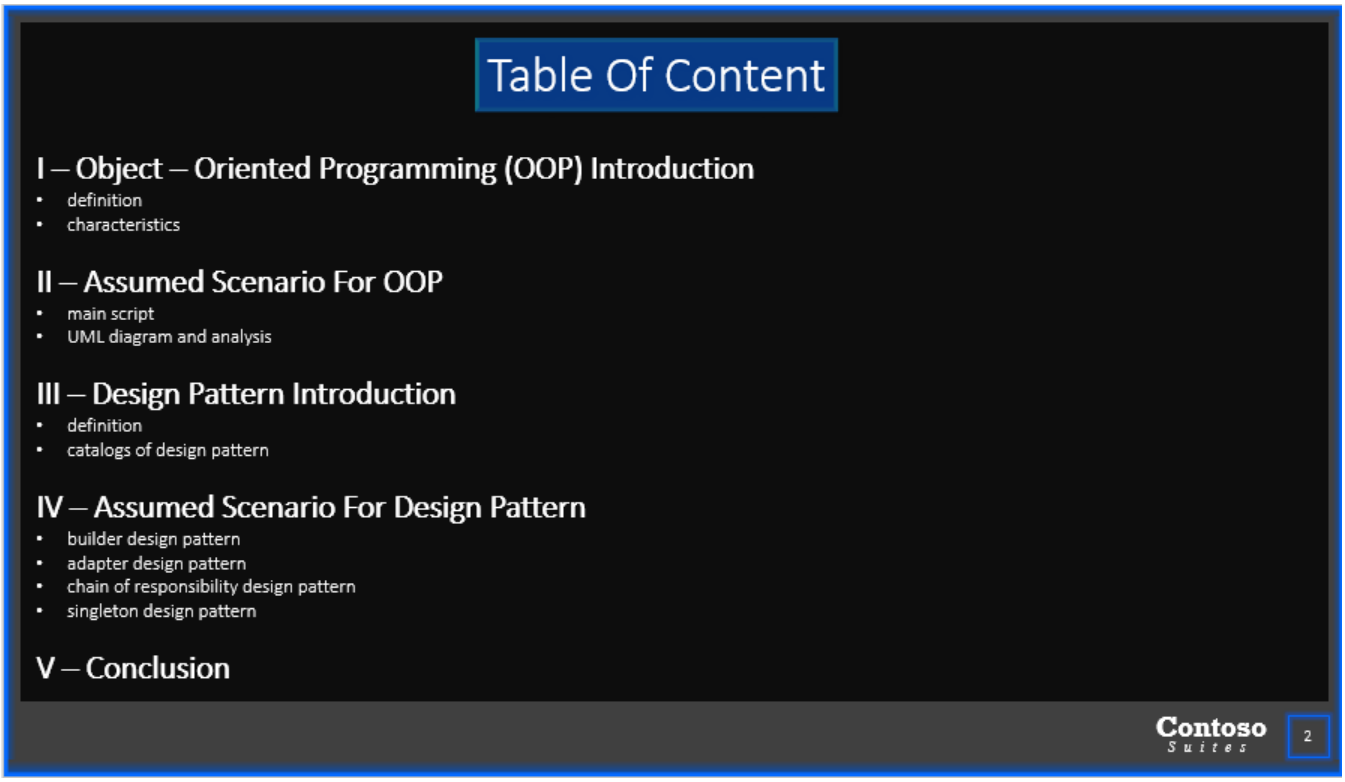
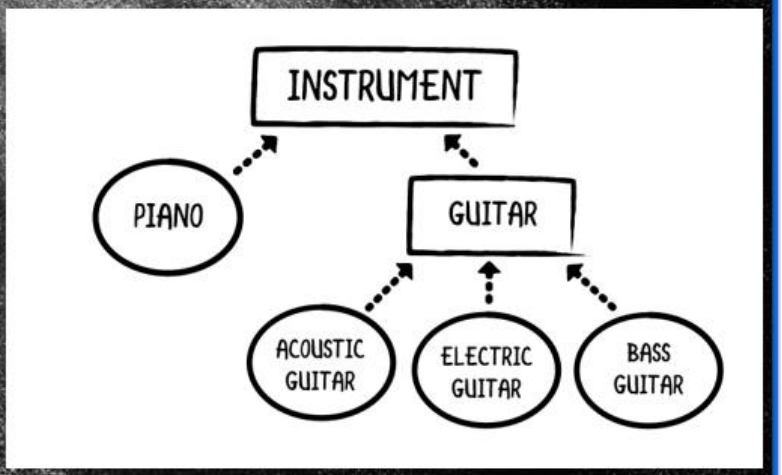


Table Of Content
I – Object – Oriented Programming (OOP) Introduction
definition
characteristics
II – Assumed Scenario For OOP
main script
UML diagram and analysis
III – Design Pattern Introduction
definition
catalogs of design pattern
IV – Assumed Scenario For Design Pattern
builder design pattern
adapter design pattern
chain of responsibility design pattern
singleton design pattern
V – Conclusion
Contoso Suites
2

I - Object – Oriented Programming Introduction

OOP Definition



I - Object – Oriented Programming Introduction

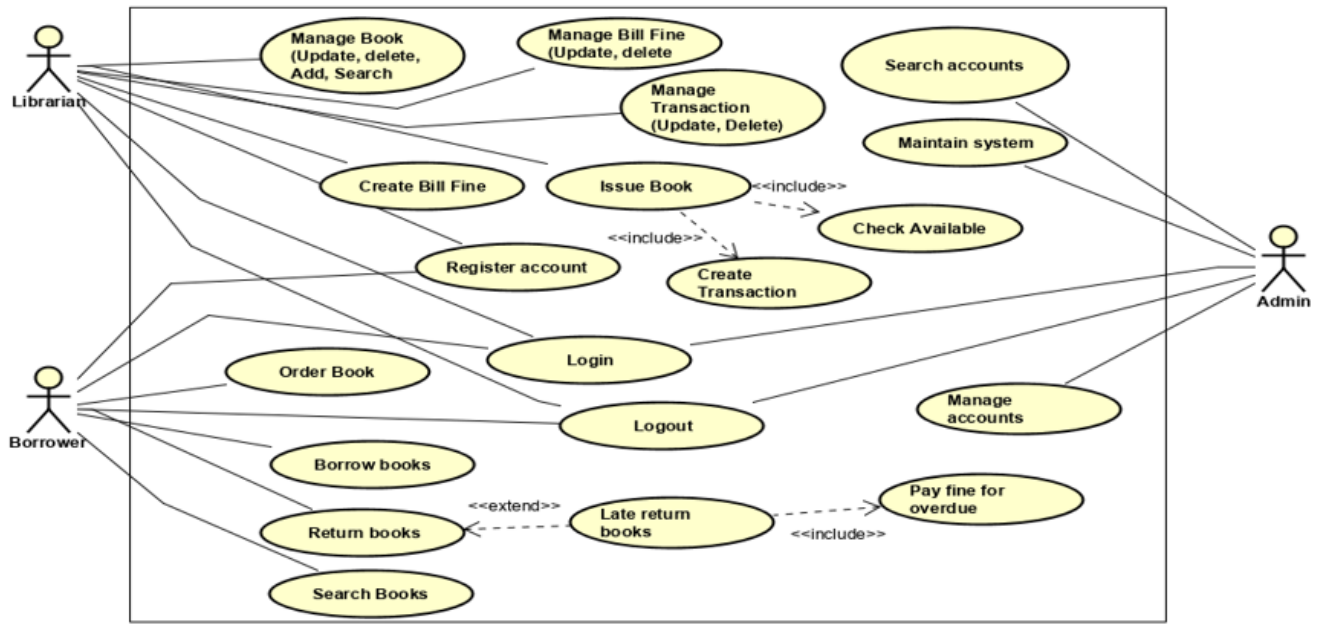
OOP Characteristics

Four main principles :

- Inheritance
- Encapsulation
- Abstraction
- Polymorphism

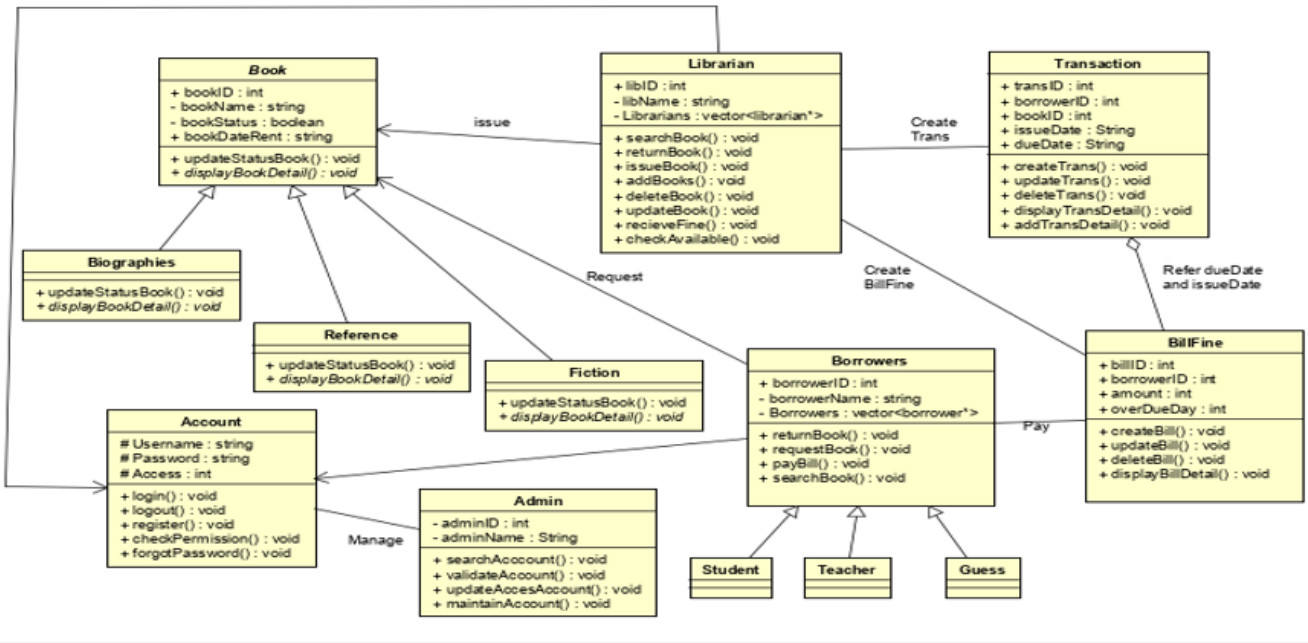
UML Diagram And Analysis

Usecase Diagram



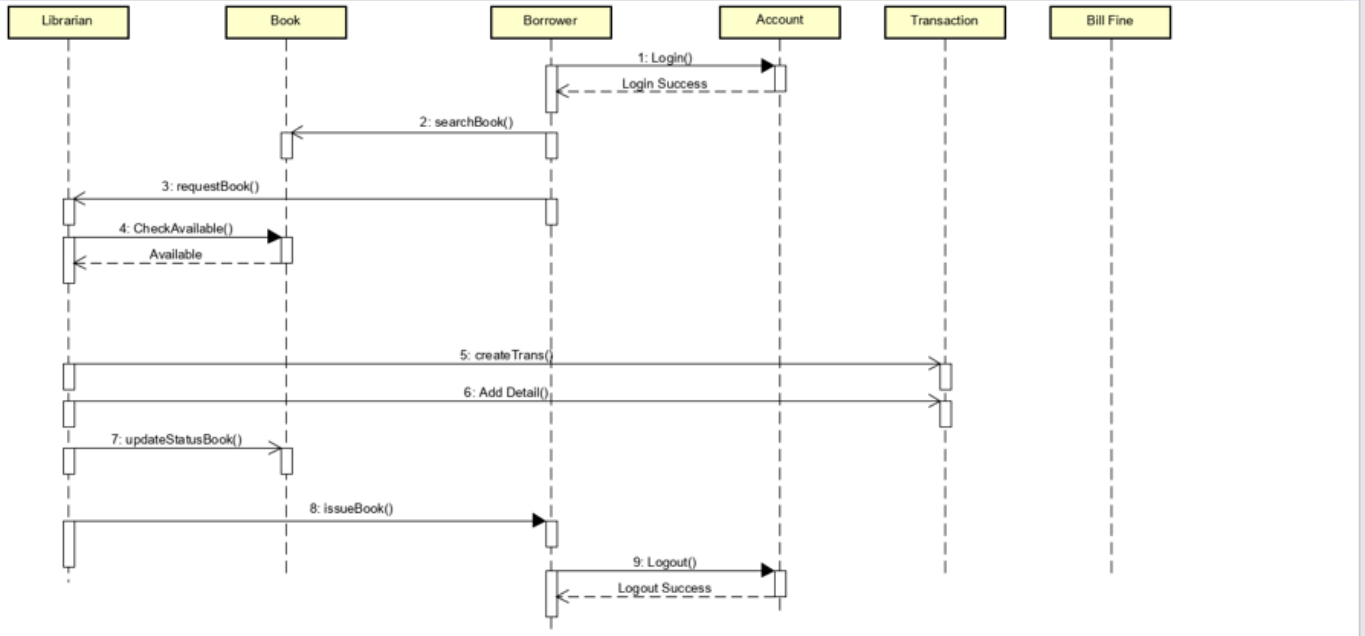
UML Diagram And Analysis

Class Diagram



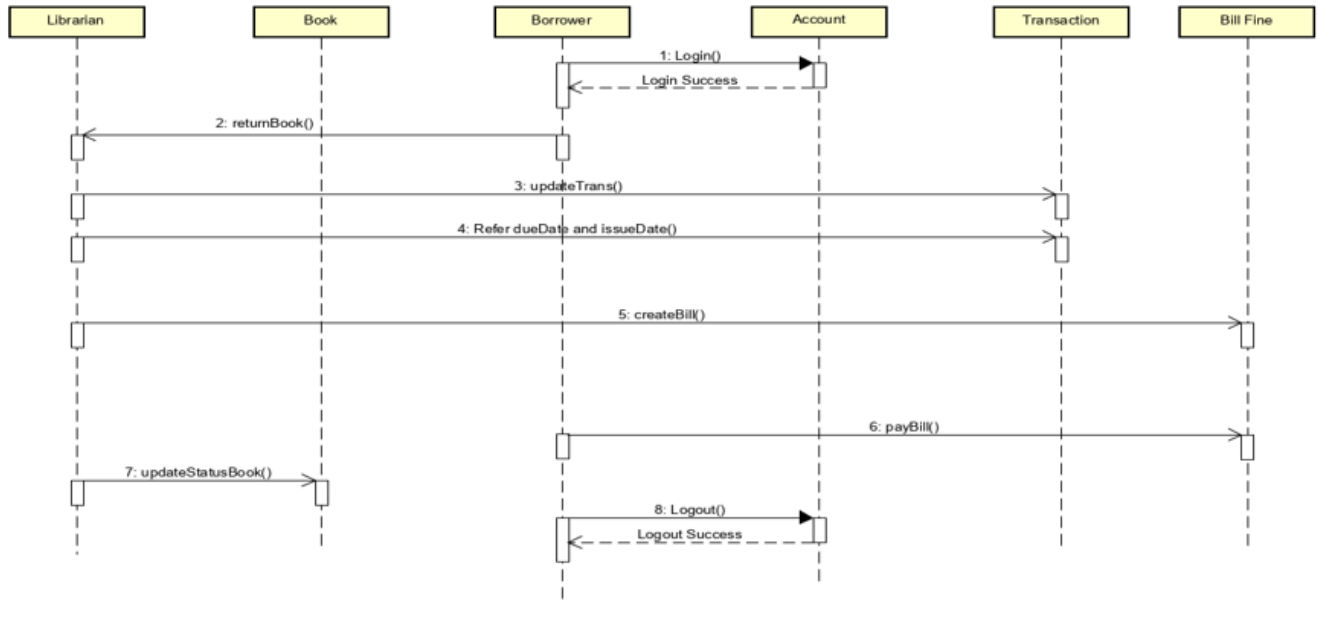
UML Diagram And Analysis

Sequence Diagram – Borrow Book



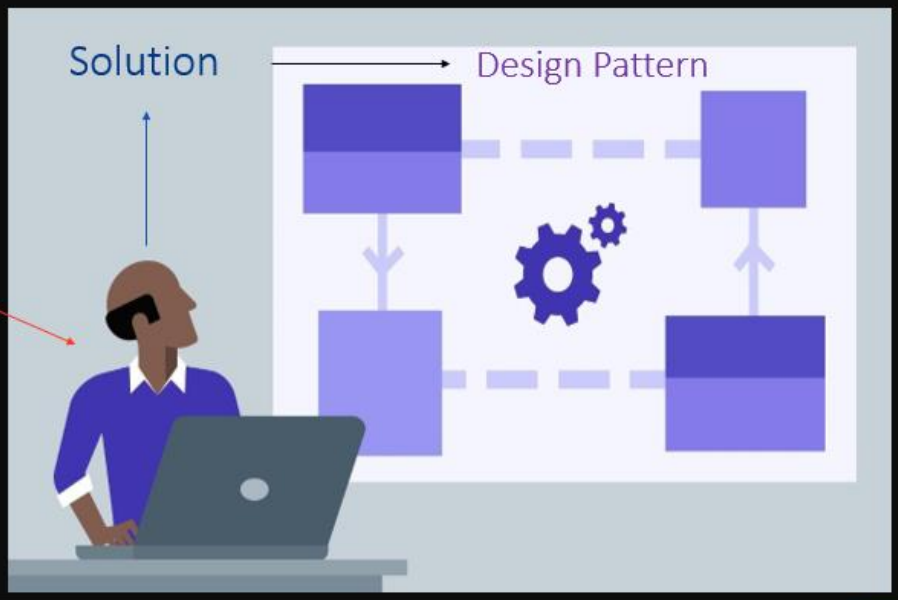
UML Diagram And Analysis

Sequence Diagram – Return Book Late



III – Design Pattern Introduction

Definition



III – Design Pattern Introduction

Catalogs of design pattern

		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	• Factory Method	• Adapter	• Interperter
	Object	• Abstract Factory • Builder • Prototype • Singleton	• Adapter • Bridge • Composite • Decorator • Facade • Flyweight • Proxy	• Chain of Responsibility • Command • Iterator • Mediator • Momento • Observer • State • Strategy • Vistor

IV – Assumed Scenario For Design Pattern

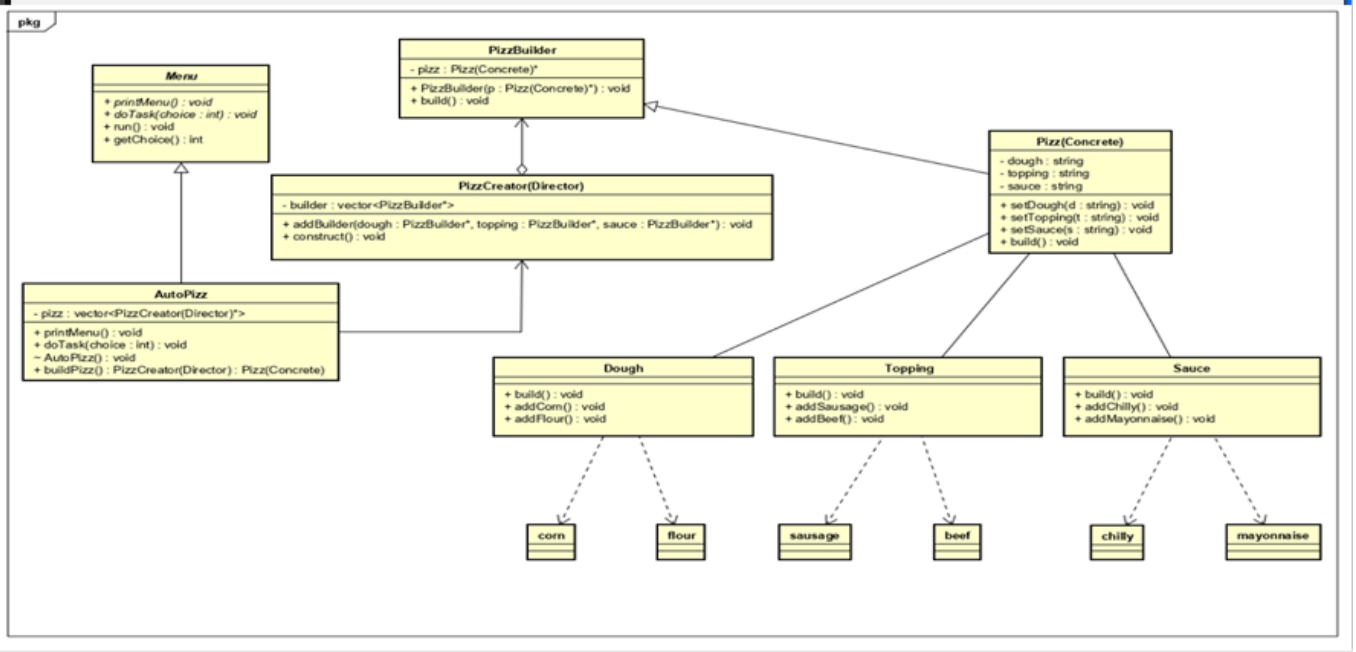
Builder Design Pattern (First) – Main Script :

Currently fast food is becoming more and more popular thanks to its usability, especially pizza. Therefore, the demand for pizza is increasing, which leads to people having to spend more time waiting to buy a pizza, along with expensive travel expenses if the pizza shop is too far.

To solve this problem, AutoPizz was born with the purpose of bringing the most convenience. AutoPizz is a series of automated pizza production and sales systems according to customer needs. This system will be located in many places to provide maximum convenience both in time and cost. First, the system will display the ingredients to make pizza from bottom to top in order (dough, topping, sauce). After that, customers will be free to choose materials suitable for their desired pizza. Finally, after choosing, customers will press the word "pizza making" to start the pizza making process

IV – Assumed Scenario For Design Pattern

Buidler Design Pattern (First) – Class Diagram :



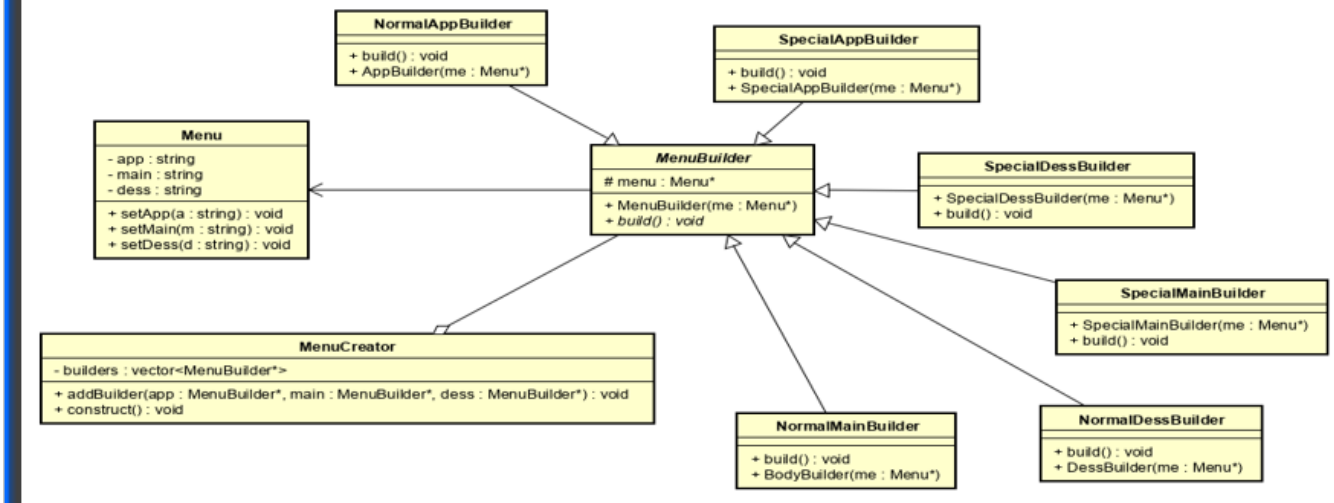
IV – Assumed Scenario For Design Pattern

Builder Design Pattern (Second) – Main Script:

A restaurant divides their menu into two separate menu lists. An ordinary and special menu includes 3 dishes: appetizers, main courses, and desserts, and it's just different about the dishes in it. They want us to create a software that helps customers choose Menu by entering from the keyboard. We decided to use Design Pattern with Builder Pattern to create a user interface built from small layers. Below is the program's Class Diagram:

IV – Assumed Scenario For Design Pattern

Builder Design Pattern (Second) – Class Diagram :



IV – Assumed Scenario For Design Pattern

Adapter Design Pattern – Main Script:

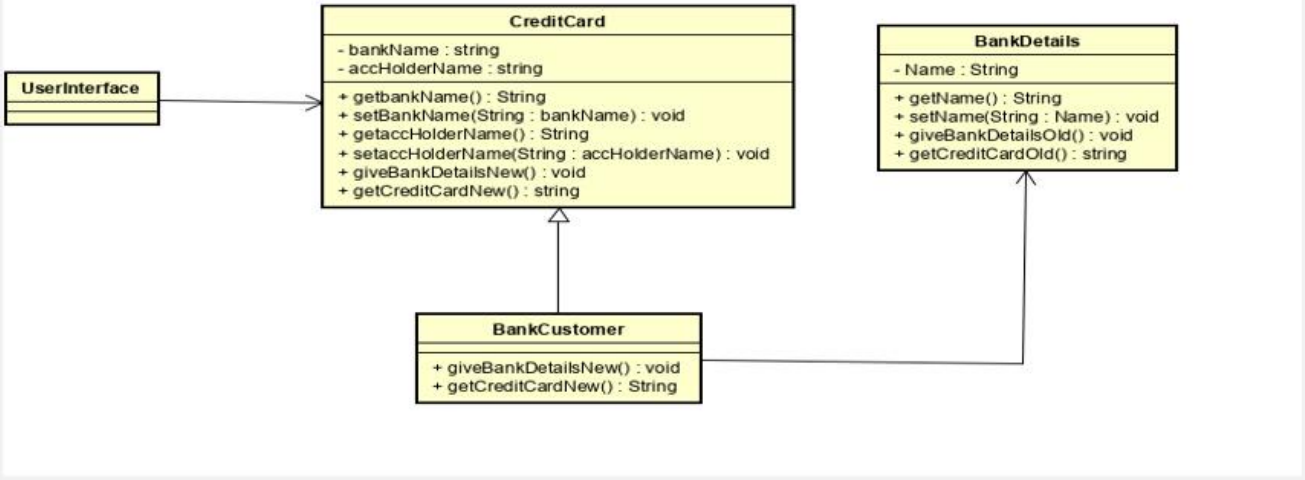
Definition: The adapter pattern convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

Scenario: A client bank account management application. Previously this application used user name to identify customers but now the company wants to upgrade the application so it is more clearly defined.

Solution: We will upgrade the application, divided name into 2 parts: User name and bank name. We need to use the Adapter to ensure that after the upgrade, the application and the old data will still be executed

IV – Assumed Scenario For Design Pattern

Adapter Design Pattern – Class Diagram :



IV – Assumed Scenario For Design Pattern

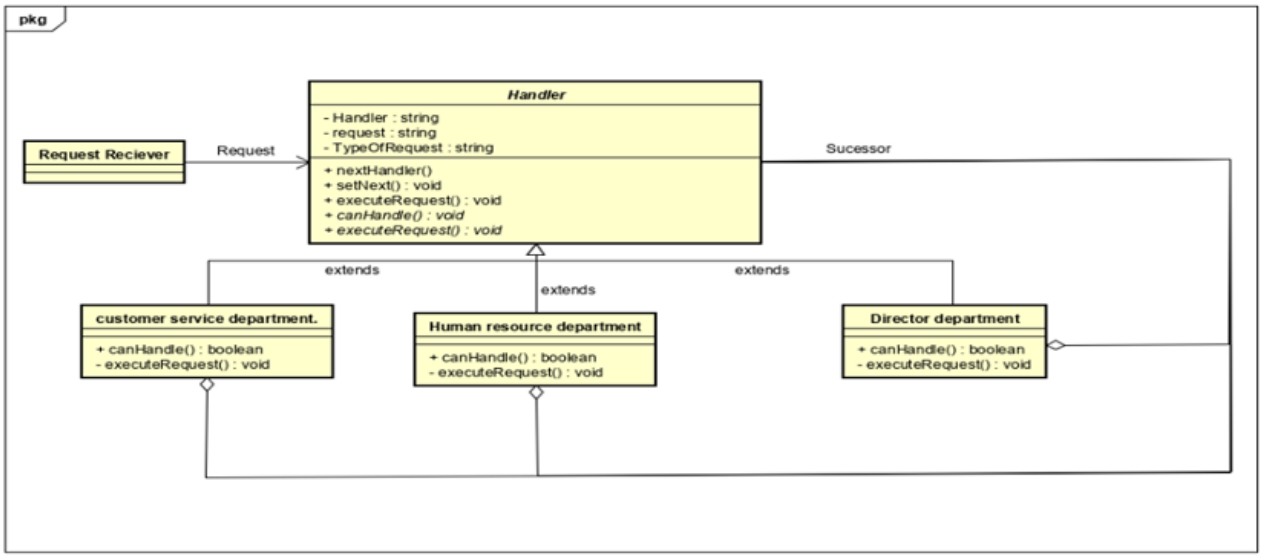
Chain of Responsibility Design Pattern – Main Script:

In programing, we quite familiar with if- else command. This command is simple and easy to use. But let's imagine we have a scenario like this: A client comes to a company and give a request. Not every request can be handle by 1 department, but each department con only handle a field of work. If the request is about investment, it will come to director department, if the request is a job applicant, it will come to Human resource department, if it is a complaint or question, it will come to customer service department.

When we have a situation like this, it's instinctive to write a massive lines of if- else command code. But the problem with this kind code is the more ladder you add to the code, the more complexity you are increasing for your code and it will become very hard to debug or actually read it. One of our job in writing code is to let other can understand it and make modification to it. If we write code that people are not able to read it, it will really hard to maintain. So one of the solution is using Chain of Responsibility.

IV – Assumed Scenario For Design Pattern

Chain of Responsibility Design Pattern – Class Diagram :



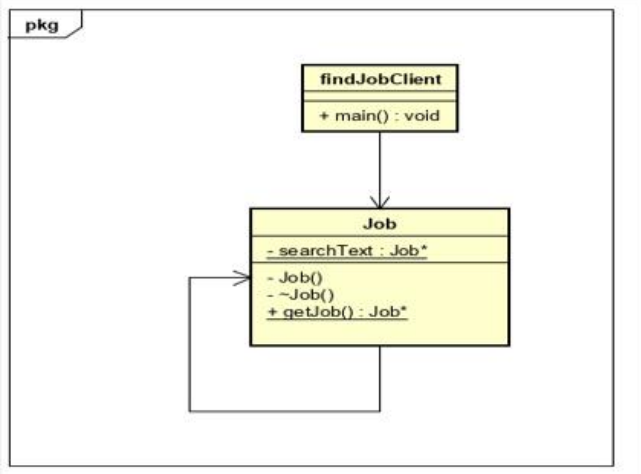
IV – Assumed Scenario For Design Pattern

Singleton Design Pattern – Main Script:

Suppose I was director of a company Search Jobs. Our company employees will have to provide information about the current job for customers. In the past, our old system can only store data on employment. If employees want to get data to serve customers must search manually. This is difficult when our system possesses a huge amount of data. So my solution is creating a automatic job search function for our system.

IV – Assumed Scenario For Design Pattern

Singleton Design Pattern – Class Diagram :



V - Conclusion

- Understand clearly about object-oriented programming and its characteristics
- Know how to apply OOP through a specific hypothetical case
- Understand the concept of Design Pattern and its main catalogs
- Know how to apply some small design patterns in hypothetical situations
- Know how to create diagrams, model data types thanks to the design of UML diagrams

Thank You So Much For Your Attention

We appreciate

Bibliography

Alexander Petkov, 2018. *FreeCodeCamp.* [Online]
Available at: https://www.freecodecamp.org/news/how-to-explain-object-oriented-programming-concepts-to-a-6-year-old-21bb035f7260/
[Accessed 22 June 2019].

Erich Gamma et al, 1994. *Design Patterns, Elements of Resuable Object-Oriented Software.* 1st ed. Canada: Addison-Wesley.

❒Summative Feedback: ❒ResubmissionFeedback:		
Grade:	**AssessorSignature:**	**Date:**
Lecturer Signature:		