VIETNAM GENERAL CONFEDERATION OF LABOR

**TON DUC THANG UNIVERSITY**

**FACULTY OF INFORMATION TECHNOLOGY**

A red and blue logo

Description automatically generated

**FINAL PROJECT**

**DESIGN PATTERN**

*Lecture*: **TS Vũ Đình Hồng**

*Implementer*: **Tạ Tiến Đạt – 521H0442**

**Tăng Thiệu Phong – 521H0136**

**Phan Lương Huy – 521H0453**

**Lương Viết An – 521H0434**

Class **: 21H50203-21H50202**

Year  **: 25**

**HO CHI MINH CITY, 2024**

VIETNAM GENERAL CONFEDERATION OF LABOR

**TON DUC THANG UNIVERSITY**

**FACULTY OF INFORMATION TECHNOLOGY**

A red and blue logo

Description automatically generated

**FINAL PROJECT**

**DESIGN PATTERN**

*Lecture*: **TS Vũ Đình Hồng**

*Implementer*: **Tạ Tiến Đạt – 521H0442**

**Tăng Thiệu Phong – 521H0136**

**Phan Lương Huy – 521H0453**

**Lương Viết An – 521H0434**

Class **: 21H50203-21H50202**

Year  **: 25**

HO CHI MINH CITY, 2024

THANK YOU

We want to extend our warm greetings to the Faculty of Information Technology, as well as the faculty and students from other faculties at Ton Duc Thang University. We are grateful for your support and direction in developing our study on discrete structures. We would like to express our sincere appreciation to Mr. Vu Dinh Hong, who provided specific supervision and guidance throughout the completion of this article. As our skills are still limited, we acknowledge that there may be some imperfections that we hope you will overlook. Furthermore, we welcome feedback from various sources to help us gain more knowledge and improve the quality of our graduation report.

**THE WORK IS COMPLETED**

**AT TON DUC THANG UNIVERSITY**

I hereby declare that this is my own research project and is under the scientific guidance of Dr. Vu Dinh Hong. The research content and results on this topic are honest and have not been published in any form before. The data in the tables for analysis, comments, and evaluation were collected by the author from different sources and clearly stated in the reference section.

In addition, the thesis also uses a number of comments, assessments as well as data from other authors and other organizations, all with citations and source notes.

If any fraud is discovered, I will take full responsibility for the content of my thesis. Ton Duc Thang University is not involved in copyright violations caused by me during the implementation process (if any).

*Ho Chi Minh City, April 28 2024*

*Author*

*(sign and write full name)*

*Tạ Tiến Đạt*

*Tăng Thiệu Phong*

*Phan Lương Huy*

*Lương Viết An*

SUMMARY

Our final project showcases design patterns implemented to solve key issues within a POS system. We'll delve into each pattern, explaining its purpose and how it tackles specific challenges to enhance the system's robustness and maintainability. Prepare to see how design patterns can streamline development and promote code reusability in aPOS context.

TABLE OF CONTENTS

[HO CHI MINH CITY, 2024 ii](#_Toc165112918)

[THANK YOU iii](#_Toc165112919)

[SUMMARY 1](#_Toc165112920)

[TABLE OF CONTENTS 2](#_Toc165112921)

[CHAPTER 1 – INTRODUCTION OF DESIGN PATTERN 7](#_Toc165112922)

[1.1 What is Design Pattern 7](#_Toc165112923)

[1.1.1 Introduction 7](#_Toc165112924)

[1.1.2 Constitution 7](#_Toc165112925)

[1.2 Introduction of Singleton Pattern 7](#_Toc165112926)

[1.2.1. Definition 7](#_Toc165112927)

[1.2.2 UML 7](#_Toc165112928)

[1.2.3 Problem 8](#_Toc165112929)

[1.2.4 Solution 8](#_Toc165112930)

[1.2.5 Consequence 8](#_Toc165112931)

[1.3 Introduction of Strategy Pattern 9](#_Toc165112932)

[1.3.1 Definition 9](#_Toc165112933)

[1.3.2 UML 9](#_Toc165112934)

[1.3.3 Problem 9](#_Toc165112935)

[1.3.4 Solution 10](#_Toc165112936)

[1.3.5 Consequence 10](#_Toc165112937)

[1.4 Introduction of Template Method Pattern 10](#_Toc165112938)

[1.4.1 Definition 10](#_Toc165112939)

[1.4.2 UML 11](#_Toc165112940)

[1.4.3 Problem 11](#_Toc165112941)

[1.4.4 Solution 11](#_Toc165112942)

[1.4.5 Consequence 12](#_Toc165112943)

[1.5 Introduction of Factory Pattern 12](#_Toc165112944)

[1.5.1 Definition 12](#_Toc165112945)

[1.5.2 UML 12](#_Toc165112946)

[1.5.3 Problem 13](#_Toc165112947)

[1.5.4 Solution 13](#_Toc165112948)

[1.5.5 Consequence 13](#_Toc165112949)

[1.6 Introduction of Command Pattern 14](#_Toc165112950)

[1.6.1 Definition 14](#_Toc165112951)

[1.6.2 UML 14](#_Toc165112952)

[1.6.3 Problem 14](#_Toc165112953)

[1.6.4 Solution 15](#_Toc165112954)

[1.6.5 Consequence 15](#_Toc165112955)

[1.7 Introduction of Adapter Pattern 15](#_Toc165112956)

[1.7.1 Definition 15](#_Toc165112957)

[1.7.2 UML 15](#_Toc165112958)

[1.7.3 Problem 16](#_Toc165112959)

[1.7.4 Solution 16](#_Toc165112960)

[1.7.5 Consequence 16](#_Toc165112961)

[1.8 Introduction of Decorator Pattern 17](#_Toc165112962)

[1.8.1 Definition 17](#_Toc165112963)

[1.8.2 UML 17](#_Toc165112964)

[1.8.3 Problem 18](#_Toc165112965)

[1.8.4 Solution 18](#_Toc165112966)

[1.8.5 Consequence 18](#_Toc165112967)

[1.9 Introduction of Builder Pattern 18](#_Toc165112968)

[1.9.1 Definition 18](#_Toc165112969)

[1.9.2 UML 19](#_Toc165112970)

[1.9.3 Problem 19](#_Toc165112971)

[1.9.4 Solution 20](#_Toc165112972)

[1.9.5 Consequence 20](#_Toc165112973)

[1.10 Introduction of Proxy Pattern 20](#_Toc165112974)

[1.10.1 Definition 20](#_Toc165112975)

[1.10.2 UML 20](#_Toc165112976)

[1.10.3 Problem 21](#_Toc165112977)

[1.10.4 Solution 21](#_Toc165112978)

[1.10.5 Consequence 22](#_Toc165112979)

[1.11 Introduction of ChainOfResponsibility Pattern 22](#_Toc165112980)

[1.11.1 Definition 22](#_Toc165112981)

[1.11.2 UML 22](#_Toc165112982)

[1.11.3 Problem 23](#_Toc165112983)

[1.11.4 Solution 23](#_Toc165112984)

[1.11.5 Consequence 24](#_Toc165112985)

[CHAPTER 2 – APPLICATION OF DESIGN PATTERN 25](#_Toc165112986)

[2.1 Application of Singleton Pattern 25](#_Toc165112987)

[2.1.1 Introduce 25](#_Toc165112988)

[2.1.2 UML 25](#_Toc165112989)

[2.1.2.1 CacheManager: 25](#_Toc165112990)

[2.1.2.2 Logger 26](#_Toc165112991)

[2.1.3 How it works 26](#_Toc165112992)

[2.1.3.1 Cache Manager 26](#_Toc165112993)

[2.1.3.2 LogManager 27](#_Toc165112994)

[2.1.4 Apply code 27](#_Toc165112995)

[2.1.4.1 CacheManger 27](#_Toc165112996)

[2.2 Application of Strategy Pattern 28](#_Toc165112997)

[2.2.1 Introduce 28](#_Toc165112998)

[2.2.2 UML 29](#_Toc165112999)

[2.2.3 How it works 29](#_Toc165113000)

[2.2.4 Apply code 30](#_Toc165113001)

[2.3 Application of Template Method Pattern 31](#_Toc165113002)

[2.3.1 Introduce 31](#_Toc165113003)

[2.3.2 UML 32](#_Toc165113004)

[2.3.3 How it works 33](#_Toc165113005)

[2.3.4 Apply code 35](#_Toc165113006)

[2.4 Application of Factory Pattern 39](#_Toc165113007)

[2.4.1 Introduce 39](#_Toc165113008)

[2.4.2 UML 40](#_Toc165113009)

[2.4.3 How it works 40](#_Toc165113010)

[2.4.4 Apply code 42](#_Toc165113011)

[2.5 Application of Command Pattern 44](#_Toc165113012)

[2.5.1 Introduce 44](#_Toc165113013)

[2.5.2 UML 45](#_Toc165113014)

[2.5.3 How it works 45](#_Toc165113015)

[2.5.4 Apply code 46](#_Toc165113016)

[2.6 Application of Adapter Pattern 49](#_Toc165113017)

[2.6.1 Introduce 49](#_Toc165113018)

[2.6.2 UML 50](#_Toc165113019)

[2.6.3 How it works 50](#_Toc165113020)

[2.6.4 Apply code 51](#_Toc165113021)

[2.7 Application of Decorator Pattern 54](#_Toc165113022)

[2.7.1 Introduce 54](#_Toc165113023)

[2.7.2 UML 54](#_Toc165113024)

[2.7.3 How it works 55](#_Toc165113025)

[2.7.4 Apply code 55](#_Toc165113026)

[2.8 Application of Builder Pattern 57](#_Toc165113027)

[2.8.1 Introduce 57](#_Toc165113028)

[2.8.2 UML 57](#_Toc165113029)

[2.8.3 How it works 58](#_Toc165113030)

[2.8.4 Apply code 59](#_Toc165113031)

[2.9 Application of Proxy Pattern 61](#_Toc165113032)

[2.9.1 Introduce 61](#_Toc165113033)

[2.9.2 UML 62](#_Toc165113034)

[2.9.3 How it works 62](#_Toc165113035)

[2.9.4 Apply code 63](#_Toc165113036)

[2.10 Application of ChainOfResponsibility Pattern 64](#_Toc165113037)

[2.10.1 Introduce 64](#_Toc165113038)

[2.10.2 UML 64](#_Toc165113039)

[2.10.3 How it works 64](#_Toc165113040)

[2.10.4 Apply code 65](#_Toc165113041)

[CHAPTER 3 – REFERENCES 68](#_Toc165113042)

CHAPTER 1 – INTRODUCTION OF DESIGN PATTERN

* 1. What is Design Pattern

1.1.1 Introduction

* Design patterns are reusable solutions for common software design problems.
* They act as **blueprints** that can be adapted to fit your specific code.
* Unlike functions or libraries, design patterns are not ready-made code snippets, but rather **general concepts** for solving problems.
* Design patterns differ from algorithms. Algorithms are step-by-step instructions, while design patterns are more **high-level** descriptions.
* The same design pattern implemented in different programs might result in different code.

1.1.2 Constitution

* **Intent**: This concisely explains the problem the pattern addresses and the general solution it offers.
* **Motivation**: Here, the problem is elaborated on further, along with the benefits and advantages of using the specific design pattern as a solution.
* **Structure**: This section outlines the different components involved in the design pattern and how they interact with each other. It essentially provides a blueprint of the pattern's structure.

1.2 Introduction of Singleton Pattern

1.2.1. Definition

* [Singleton](https://en.wikipedia.org/wiki/Singleton_pattern) is a creational design pattern that lets you ensure that a class has only one instance and provide a global access point to this instance.
* **Singleton** is one of the 5 design patterns of the **Creational Design Pattern** group.

1.2.2 UML



1.2.3 Problem

* When you need to access data or functions of a class from anywhere in the system.
* When it is necessary to ensure that all parts of the system use the same state or data
* When needing to optimize resource usage such as memory or database connections
* When you need to strictly control the creation and access of instances of a class

1.2.4 Solution

* Make the default constructor private, to prevent other objects from using the new operator with the Singleton class.
* Create a static creation method that acts as a constructor. Under the hood, this method calls the private constructor to create an object and saves it in a static field. All following calls to this method return the cached object.

1.2.5 Consequence

* Violates the Single Responsibility Principle. The pattern solves two problems at the time.
* The Singleton pattern can mask bad design, for instance, when the components of the program know too much about each other.
* The pattern requires special treatment in a multithreaded environment so that multiple threads won’t create a singleton object several times.
* It may be difficult to unit test the client code of the Singleton because many test frameworks rely on inheritance when producing mock objects

1.3 Introduction of Strategy Pattern

1.3.1 Definition

* Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from the clients that use it.
* Strategy Pattern is one of the Patterns in the Behavioral Pattern group. It allows for defining a set of algorithms, packaging each algorithm, and easily and flexibly changing the algorithms within the object. Strategy allows algorithms to change independently as users use them.

1.3.2 UML



1.3.3 Problem

* Each time you added a new routing algorithm, the main class of the navigator doubled in size. At some point, the program became too hard to maintain.
* Any change to one of the algorithms, whether it was a simple bug fix or a slight adjustment of the street score, affected the whole class, increasing the chance of creating an error in already-working code.
* In addition, teamwork became inefficient. Your teammates, who had been hired right after the successful release, complain that they spend too much time resolving merge conflicts. Implementing a new feature requires you to change the same huge class, conflicting with the code produced by other people.

1.3.4 Solution

* The Strategy pattern suggests that you take a class that does something specific in a lot of different ways and extract all these algorithms into separate classes called strategies.
* The original class, called context, must have a field for storing a reference to one of the strategies. The context delegates the work to a linked strategy object instead of executing it on its own.
* This way the context becomes independent of concrete strategies, so you can add new algorithms or modify existing ones without changing the code of the context or other strategies.

1.3.5 Consequence

* If you only have a couple of algorithms and they rarely change, there’s no real reason to overcomplicate the program with new classes and interfaces that come along with the pattern.
* Clients must be aware of the differences between strategies to be able to select a proper one.
* A lot of modern programming languages have functional type support that lets you implement different versions of an algorithm inside a set of anonymous functions. Then you could use these functions exactly as you’d have used the strategy objects, but without bloating your code with extra classes and interfaces

1.4 Introduction of Template Method Pattern

1.4.1 Definition

* Define the skeleton of an algorithm in an operation, deferring some steps to client subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm’s structure.
* **Template Method Pattern:** A behavioral design pattern that defines a general structure for an algorithm.
* **Key concept:** Subclasses can customize specific steps within the overall algorithm structure.
* **Application:** Often used with abstract and concrete classes.
  + Abstract class defines the template with specific steps (can be implemented or abstract).
  + Concrete classes implement or override steps to define their specific behavior.

1.4.2 UML



1.4.3 Problem

* At some point, you noticed that all classes have a lot of similar code. While the code for dealing with various data formats was entirely different in all classes, the code for data processing and analysis is almost identical.
* There was another problem related to the client code that used these classes. It had lots of conditionals that picked a proper course of action depending on the class of the processing object. If all three processing classes had a common interface or a base class, you’d be able to eliminate the conditionals in client code and use polymorphism when calling methods on a processing object.

1.4.4 Solution

* The Template Method pattern suggests that you break down an algorithm into a series of steps, turn these steps into methods, and put a series of calls to these methods inside a single template method.
* At first, we can declare all steps **abstract**, forcing the subclasses to provide their implementations for these methods.
  + **Abstract steps** must be implemented by every subclass.
  + **Optional steps** already have some default implementation but still can be overridden if needed.
* There’s another type of step, called **hooks**. A hook is an optional step with an empty body. A template method would work even if a hook isn’t overridden. Usually, hooks are placed before and after crucial steps of algorithms, providing subclasses with additional extension points for an algorithm.

1.4.5 Consequence

* Some clients may be limited by the provided skeleton of an algorithm.
* You might violate the **Liskov Substitution Principle** by suppressing a default step implementation via a subclass.
* Template methods tend to be harder to maintain the more steps they have.

1.5 Introduction of Factory Pattern

1.5.1 Definition

* [Factory Method](https://en.wikipedia.org/wiki/Factory_method_pattern) is a creational design pattern that Defines an interface for creating an object but lets subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.
* **Factory Method Design Pattern**, simply called **Factory Pattern,** is one of the Patterns in the **Creational Design Pattern** group. The Factory Pattern's task is to manage and return objects on request, helping to create object creation more flexibly.
* **Factory Pattern** is a **factory**, and this factory will " **produce** " objects according to our requirements

1.5.2 UML



1.5.3 Problem

* The client code might become tightly coupled to the concrete classes being created. This can make it difficult to introduce new product types without modifying the client code.
* Testing the factory logic itself can be more complex than testing code that directly creates objects. This is because the factory might involve conditional logic to decide which concrete class to instantiate.

1.5.4 Solution

* The Factory Method pattern suggests that you replace direct object construction calls (using the newoperator) with calls to a special factorymethod. Don’t worry: the objects are still created via the newoperator, but it’s being called from within the factory method. Objects returned by a factory method are often referred to as products.
* There’s a slight limitation though: subclasses may return different types of products only if these products have a common base class or interface. Also, the factory method in the base class should have its return type declared as this interface.

1.5.5 Consequence

The code may become more complicated since you need to introduce a lot of new subclasses to implement the pattern. The best-case scenario is when you’re introducing the pattern into an existing hierarchy of creator classes.

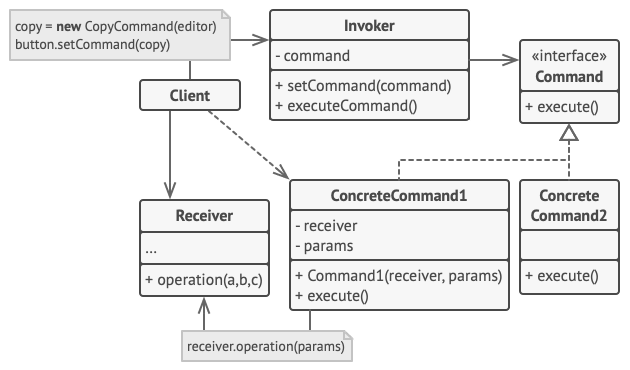
1.6 Introduction of Command Pattern

1.6.1 Definition

- Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.

**- Command Pattern** is one of the Patterns in the Behavioral Pattern group. It allows turning requests into independent objects, which can be used to parameterize objects with different requests such as log, queue (undo/redo), and translation.

1.6.2 UML



1.6.3 Problem

* Introducing the command pattern adds more classes and objects to your codebase. This can make the code initially more complex to understand and maintain.
* If not implemented carefully, the command pattern can lead to tight coupling. The invoker (the object triggering the command) might become reliant on specific concrete command classes.

1.6.4 Solution

* Good software design is often based on the **principle of separation of concerns**, which usually results in breaking an app into layers. The most common example: a layer for the graphical user interface and another layer for the business logic. The GUI layer is responsible for rendering a beautiful picture on the screen, capturing any input and showing results of what the user and the app are doing. However, when it comes to doing something important, like calculating the trajectory of the moon or composing an annual report, the GUI layer delegates the work to the underlying layer of business logic..
* As a result, commands become a convenient middle layer that reduces the coupling between the GUI and business logic layers. And that’s only a fraction of the benefits that the Command pattern can offer.

1.6.5 Consequence

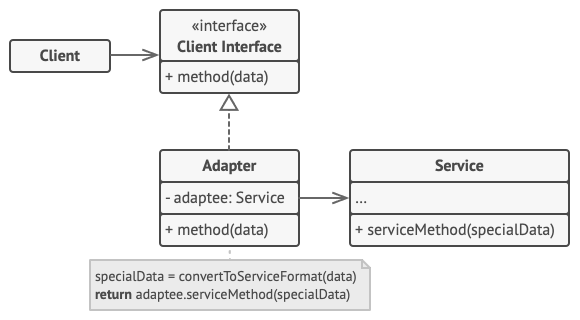
The code may become more complicated since you’re introducing a whole new layer between senders and receivers.

1.7 Introduction of Adapter Pattern

1.7.1 Definition

* Adapter (wrapper) is a design pattern belonging to the Structural Pattern group design patterns for structural design
* A design pattern that converts the interface of a class into another template that the client wants. Allows two unrelated templates to work together.
* Adapter Pattern plays an intermediary role between two classes, converting the interface of one or more existing classes into another interface, allowing classes with different interfaces to easily communicate well with each other through the intermediate interface., no need to change the code of the existing class or the class being written.
* Adapter Pattern is also called Wrapper Pattern because it provides a compatible "outer" interface for an existing system, has appropriate data and behavior but has an interface that is incompatible with the class being written.

1.7.2 UML



1.7.3 Problem

* In cases where we want to convert data from many different sources to a common standard without editing the old code, we should use this pattern.
* When you want to use some subclasses, but those subclasses are missing some methods and the methods cannot be included in the parent class.
* Want to use some available classes but their interfaces are not compatible with the current code
* Adapters can not only convert data into many different formats but can also help objects with different interfaces collaborate.

1.7.4 Solution

* An adapter wraps one of the objects to hide the complexity of conversion happening behind the scenes. The wrapped object isn’t even aware of the adapter
* The adapter has a compatible interface with one of the existing objects.
* Using this interface, the existing object can safely call the adapter's methods.
* When the call is received, the adapter passes the request to the second object, but in the format and order expected by the second object.

1.7.5 Consequence

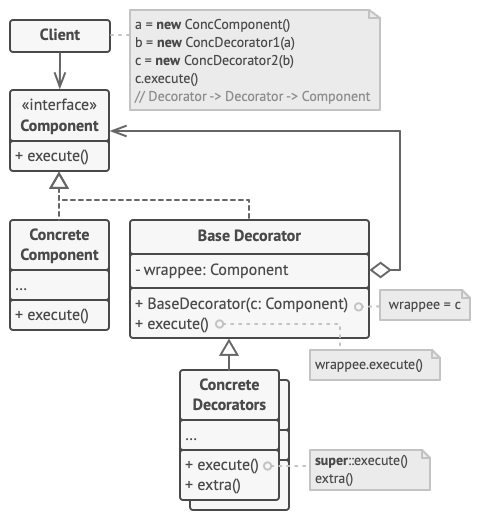
Introducing an adapter adds another layer of abstraction to your code. While it translates interfaces, it can make the overall code structure more complex, especially with many adapters.

1.8 Introduction of Decorator Pattern

1.8.1 Definition

Decorator is a structural design pattern that lets you attach new behaviors to objects, either statically or dynamically, without affecting the behavior of other objects from the same class.

1.8.2 UML



1.8.3 Problem

You want to add behavior or state to individual objects at run-time. Inheritance is not feasible because it is static and applies to an entire class.

1.8.4 Solution

* A "wrapper" serves as an alternative term for the Decorator pattern, conveying its central concept effectively. Essentially, a wrapper is an object that can be associated with a target object. It mirrors the target's methods and delegates all requests it receives to it. However, the wrapper can modify the result by performing actions either before or after passing the request to the target.
* A simple wrapper evolves into a true decorator when it adheres to the same interface as the wrapped object. Consequently, from the client's viewpoint, these objects appear identical. By allowing the wrapper's reference field to accept any object conforming to that interface, you enable the stacking of multiple wrappers around an object, thereby incorporating the combined behavior of all the wrappers.
* The client code would need to wrap a basic notifier object into a set of decorators that match the client’s preferences. The resulting objects will be structured as a stack.
* The final decorator in the stack becomes the object with which the client interacts. Given that all decorators adhere to the same interface as the base notifier, the client code remains agnostic as to whether it operates with the "pure" notifier object or the decorated one.

1.8.5 Consequence

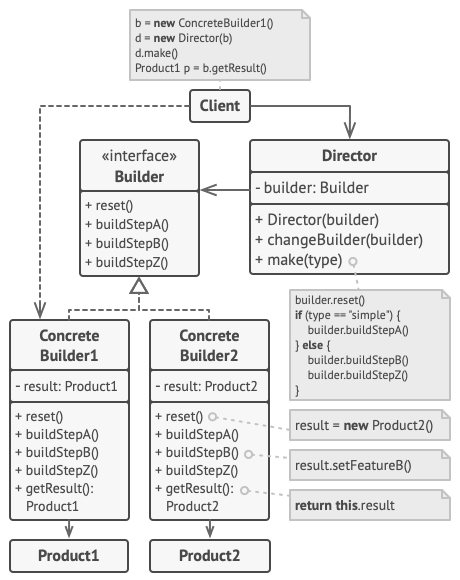
* Removing a specific wrapper from the stack of wrappers can be challenging.
* Implementing a decorator in a manner that its behavior remains independent of the order in the decorator's stack can be difficult.
* The initial configuration code for layers might appear messy.

1.9 Introduction of Builder Pattern

1.9.1 Definition

* The builder pattern is one of the Creational patterns that lack designs for initializing class objects.
* Builder pattern was created to build a complex object using simple objects and uses a step-by-step approach, building objects independently of other objects.

1.9.2 UML



1.9.3 Problem

* When an object initializes too many parameters or has many parameters with complex data types or when there are many parameters that we do not need to pass into the object, we will use the Builder Pattern.
* In addition, to limit the overloading of Product classes we can also use the Builder Pattern

1.9.4 Solution

* The Builder pattern suggests that you extract the object construction code out of its class and move it to separate objects called builders.
* This pattern organizes object construction into a set of steps. To create an object, you perform this series of steps on the generator object. The important part is that you don't have to follow all the steps. You can call only those steps necessary to create a particular configuration of an object.
* Some of the construction steps might require different implementations when you need to build various representations of the product.

1.9.5 Consequence

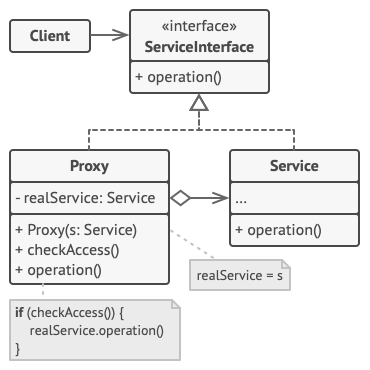
The overall complexity of the code increases since the pattern requires creating multiple new classes.

1.10 Introduction of Proxy Pattern

1.10.1 Definition

* Proxy Pattern is a Design Pattern belonging to the Structural Design Pattern group related to the structure and structure of objects in object-oriented programming.
* Proxy is a structural design pattern that allows you to provide a substitute or placeholder for another object. Proxies control access to the original object, allowing you to do something before or after the request is passed to the original object.

1.10.2 UML



1.10.3 Problem

* When you want to control access to the actual object and apply checks before allowing access.
* When actual object creation is expensive and you want to delay object creation until it is necessary
* When you want to direct requests from the client to many different objects according to specific conditions or rules.
* When you want to add functionality to an actual object without changing its interface.

1.10.4 Solution

Proxy Pattern says that we need to create a new class representing the service class with the same interface, this class is called proxy. Then when updating the application, it will pass the proxy object to all clients on the original object. When receiving a request from the client, the proxy creates a real service and delegates all tasks to it.

1.10.5 Consequence

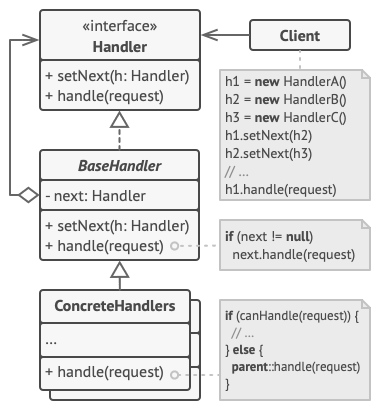
* The code can become more complicated because you need to add new classes.
* Response from the service may be delayed.

1.11 Introduction of ChainOfResponsibility Pattern

1.11.1 Definition

The Chain of Responsibility pattern is a behavioral design pattern that allows an object to pass a request along a chain of handlers. Each handler in the chain either handles the request or passes it to the next handler in the chain. It is represented in UML diagrams with a chain of handler objects.

1.11.2 UML



1.11.3 Problem

There is a potentially variable number of "handler" or "processing element" or "node" objects, and a stream of requests that must be handled. Need to efficiently process the requests without hard-wiring handler relationships and precedence, or request-to-handler mappings.

1.11.4 Solution

* Similar to other behavioral design patterns, the Chain of Responsibility involves converting specific behaviors into independent objects known as handlers. In our scenario, each check is abstracted into its own class, housing a single method responsible for executing the check. The request, along with its associated data, is passed as an argument to this method.
* The pattern advocates for linking these handlers into a chain. Each handler in the chain holds a reference to the next handler. Besides processing the request, handlers also pass it along the chain. The request traverses the chain until all handlers have had an opportunity to process it.
* The key advantage lies in a handler's ability to choose whether to pass the request further down the chain, thereby halting any subsequent processing.

1.11.5 Consequence

Some requests may end up unhandled.

CHAPTER 2 – APPLICATION OF DESIGN PATTERN

2.1 Application of Singleton Pattern

2.1.1 Introduce

* In the department's application, we have implemented the singleton pattern for 2 places that are caching and storing login and logout information for the application.
* With caching, helps minimize the number of queries to the database by storing cached data in memory. By reducing the number of queries to the database, using a cache saves costs on data access and storage.
* With the storing login and logout information, recording user login and logout times helps system administrators analyze usage behavior, detect unusual accesses, and improve user experience.
* Thanks to this, data stored and accessed is always consistent, avoiding conflicts or discrepancies caused by many different instances operating at the same time. In addition, it is also possible to avoid wasting system resources.

2.1.2 UML

2.1.2.1 CacheManager:

A screenshot of a computer program

Description automatically generated

2.1.2.2 Logger

A diagram of a log manager

Description automatically generated

2.1.3 How it works

2.1.3.1 Cache Manager

**CacheManager:** Here we will create a uniqueInstance and a constructor with the private keyword to prevent direct access from the user, then we create the getInstance() method so the user can access and get the method. initialization of the CacheManager class. Finally, create the necessary methods and properties for caching

2.1.3.2 LogManager

**LogManager:** Here we will create a uniqueInstance and a constructor with the private keyword to prevent direct access from the user, then we create the getInstance() method so the user can access and get the method. initialization of the LogManager class. Finally, create the necessary methods and properties for setting the log

2.1.4 Apply code

2.1.4.1 CacheManger

//CacheManager.java

public class CacheManager {

**private** **volatile** **static** CacheManager uniqueInstance;

**private** Map<String, Object> cache;

private CacheManager(){

cache = **new** HashMap<>();

}

public static CacheManager getInstance(){

**if**(uniqueInstance == **null**){

**synchronized** (CacheManager.class){

**if** (uniqueInstance == **null**){

uniqueInstance = **new** CacheManager();

}

}

}

**return** uniqueInstance;

}

**public** **synchronized** **void** **setCache**(String key, Object value){

cache.put(key, value);

}

**public** **synchronized** Object **getCache**(String key){

**return** cache.get(key);

}

public synchronized void removeCache(String key){

cache.remove(key);

}

}

2.1.4.2 LogManager

//LogManager.java

@Service

public class LogManager {

**private** **volatile** **static** LogManager uniqueInstance;

@Autowired

**private** LogHistoryService logHistoryService;

private LogManager() {}

@Bean

public static LogManager getInstance() {

**if**(uniqueInstance == **null**){

**synchronized** (LogManager.class){

**if** (uniqueInstance == **null**){

uniqueInstance = **new** LogManager();

}

}

}

**return** uniqueInstance;

}

public synchronized void setLog(LogHistory log){

logHistoryService.addLogHistory(log);

}

}

2.2 Application of Strategy Pattern

2.2.1 Introduce

* In the department's application, the Strategy Pattern is applied to the "Add voucher" function when paying, here we will have 3 more voucher options corresponding to the possibility they can receive, this will be based on the points they have accumulated through their purchases.
* This way we can easily expand the system to add new strategies in the future if necessary, for example, adding new coupon types based on other criteria. Suppose the customer has ordered 10 times, it means they will get 10 points, this will also correspond to them receiving 1 HAPPY10 voucher. This will be the same for HAPPY20 and HAPPY30 vouchers.
* The same Strategy class can be used for many different types of vouchers, helping to reduce duplicate code and increase system maintainability.
* Using the Strategy pattern makes the code easier to read and understand since the voucher algorithm is separated from the rest of the program.

2.2.2 UML

A computer screen shot of a computer screen

Description automatically generated

2.2.3 How it works

* First, create an interface "IStrategy" containing the price calculation method, then create a class "VoucherStrategy" and call the method inside the interface
* Next, create ConcreateStrategy classes "HAPPY10Voucher", "HAPPY20Voucher", and "HAPPY30Voucher", these classes will implement "IStrategy" to implement the same method, in the calculation method of each ConcreateStrategy will be each different calculation methods :
  + **IStrategy**: In this class, we create a method caculateVoucher with 2 parameters including price and discount
  + **VoucherStrategy**: In this class, we create 1 attribute with the data type is IStrategy, then create 1 setStrategy() method to set the kind of voucher for customers, after that, we create calculateVoucher() method to calculate the corresponding voucher
  + **HAPPY10Voucher:** In this class, we implements to IStrategy to Override the calculateVoucher() method, then we do the calculation in this method
  + **HAPPY20Voucher:** In this class, we implements to IStrategy to Override the calculateVoucher method, then we do the calculation in this method
  + **HAPPY30Voucher:** In this class, we implements to IStrategy to Override the calculateVoucher method, then we do the calculation in this method

2.2.4 Apply code

//IStrategy.java

@FunctionalInterface

public interface IStrategy {

double calculateVoucher(double price);

}

//VoucherStrategy.java

@NoArgsConstructor

@Service

public class VoucherStrategy {

**private** IStrategy iStrategy;

public VoucherStrategy(){}

**public** **void** **setStrategy**(IStrategy strategy){

**this**.iStrategy = strategy;

}

public double calculateVoucher(double price){

**return** iStrategy.calculateVoucher(price);

}

}

//HAPPY10Voucher.java

public class HAPPY10Voucher implements IStrategy {

@Override

public double calculateVoucher(double price){

**return** price - (price \* **10** / **100**);

}

}

//HAPPY20Voucher.java

public class HAPPY20Voucher implements IStrategy {

@Override

public double calculateVoucher(double price) {

**return** price - (price \* **20** / **100**);

}

}

//HAPPY30Voucher.java

public class HAPPY30Voucher implements IStrategy {

@Override

public double calculateVoucher(double price) {

**return** price - (price \* **30** / **100**);

}

}

2.3 Application of Template Method Pattern

2.3.1 Introduce

* The provided code leverages the template method pattern to establish a structured and reusable approach to creating VAT reports in various formats.
* Here's a breakdown of the main benefits:
  + **Reduce code duplication**: The base ReportExporter class centralizes common functionality such as retrieving order details and logging actions. This eliminates the need to duplicate this logic in subclasses focused on specific report formats (PDF, XML). This promotes a cleaner codebase and simplifies maintenance efforts.
  + **Flexibility**: Subclasses such as “PDFReportExporter” and “XMLReportExporter” extend the base class and implement the abstract method “export”. This allows you to tailor the specific format (PDF, XML) and presentation of data to your individual needs. The core logic for data collection and reporting remains centralized, but customization occurs at the final presentation layer.
  + **Standardized structure**: The template method pattern enforces a consistent report structure across different export types. The base class determines the order of operations and ensures that important information such as order details and VAT data are always included. This improves consistency and simplifies the interpretation of reports produced by different subclasses.
  + **Maintainability and Extensibility**: Separating core functionality from export-specific logic improves code maintainability.If the underlying data retrieval process changes, you only need to make the change in the base class and the change will be propagated to all subclasses. It also makes it easier to introduce new report formats such as CSV. A new subclass can be created to implement the “Export” method in the desired format according to the provided template.

2.3.2 UML

A black screen with white text

Description automatically generated

2.3.3 How it works

* HTTP Request Handling*:*
  + When a client sends an HTTP GET request to the “/report/VAT” endpoint, this method is invoked.
  + The “HttpServletRequest” object (“req”) contains information about the request, such as parameters.
  + The “HttpServletResponse” object (“resp”) is used to manipulate the response to be sent back to the client.
* Request Parameter Retrieval*:*
  + The method retrieves two parameters from the request:
  + “vat-type”: Specifies the type of VAT report to export (PDF or XML).
  + “order-id-vat”: Specifies the ID of the order for which the VAT report is to be generated.
* Exporter Instantiation*:*
  + Based on the value of the “vat-type” parameter, the method decides which type of “ReportExporter” to instantiate:
  + If “vat-type” is "PDF", it creates an instance of “PDFReportExporter”.
  + If “vat-type” is not "PDF", it creates an instance of “XMLReportExporter”.
  + The appropriate “orderID” and “ordersService” are passed to the constructor of the chosen exporter.
* Export VAT Report*:*
  + Once the exporter is instantiated, the method calls its “exportFileVAT()” method.
  + This method within the chosen exporter generates the VAT report:
  + For “PDFReportExporter”, it creates a PDF report.
  + For “XMLReportExporter”, it creates an XML report.
  + The method returns a “ResponseEntity<byte[]>“ containing the exported file data.
* Response Handling*:*
  + The method returns the “ResponseEntity<byte[]>“ returned by the exporter's “exportFileVAT()” method.
  + This response contains the exported file data (PDF or XML) along with appropriate headers for content type and disposition.
  + The client receives this response, which triggers a download of the exported VAT report file.

2.3.4 Apply code

//ReportExporter.java

public abstract class ReportExporter {

**private** **final** String orderID;

**private** **final** CustomerService.OrdersService ordersService;

**public** **ReportExporter**(String orderID, CustomerService.OrdersService ordersService) {

**this**.orderID = orderID;

**this**.ordersService = ordersService;

}

**public** ResponseEntity<**byte**[]> **exportFileVAT**() {

VatReport vatReport = prepareVATInformation();

loggingAction(orderID);

**return** **export**(vatReport);

}

public VatReport prepareVATInformation() {

List<Object[]> odtList = ordersService.getAllOrderListDetails(orderID);

Object[] odtDetail = (Object[]) ordersService.getOrderByOrderID(orderID);

Timestamp timestamp = (Timestamp) odtDetail[**1**];

String cusName = (String) odtDetail[**2**];

String posFirstName = (String) odtDetail[**3**];

String posLastName = (String) odtDetail[**4**];

String status = (String) odtDetail[**5**];

**float** totalAmount = (**float**) odtDetail[**6**];

List<VatReportItem> items = **new** ArrayList<>();

**for** (Object[] data : odtList) {

VatReportItem item = **new** VatReportItem();

item.setProductName((String) data[**0**]);

item.setQuantity((**int**) data[**2**]);

item.setChangeGiven(Double.parseDouble(String.valueOf(data[**3**])));

item.setDescription((String) data[**4**]);

items.add(item);

}

VatReport vatReport = **new** VatReport();

vatReport.setTimestamp(timestamp);

vatReport.setCusName(cusName);

vatReport.setPosFirstName(posFirstName);

vatReport.setPosLastName(posLastName);

vatReport.setStatus(status);

vatReport.setItems(items);

vatReport.setTotalAmount(totalAmount);

**return** vatReport;

}

**public** **abstract** ResponseEntity<**byte**[]> **export**(VatReport vatReport);

public abstract void loggingAction(String orderID);

}

//XMLReporter.java

public class XMLReportExporter extends ReportExporter {

**public** **XMLReportExporter**(String orderID, CustomerService.OrdersService ordersService) {

**super**(orderID, ordersService);

}

@Override

**public** ResponseEntity<**byte**[]> **export**(VatReport vatReport) {

// Use JAXB to convert VAT report object to XML string

JAXBContext jaxbContext = **null**;

**try** {

jaxbContext = JAXBContext.newInstance(VatReport.class);

Marshaller marshaller = jaxbContext.createMarshaller();

marshaller.setProperty(Marshaller.JAXB\_FORMATTED\_OUTPUT, **true**); // Formatted output for readability

ByteArrayOutputStream outputStream = **new** ByteArrayOutputStream();

marshaller.marshal(vatReport, outputStream);

**byte**[] xmlBytes = outputStream.toByteArray();

// Set response headers for XML content

HttpHeaders headers = **new** HttpHeaders();

headers.setContentType(MediaType.APPLICATION\_XML);

headers.setContentDisposition(ContentDisposition.attachment()

.filename("vat\_report\_" + vatReport.getTimestamp() + ".xml")

.build());

**return** **new** ResponseEntity<>(xmlBytes, headers, HttpStatus.OK);

} **catch** (JAXBException e) {

throw new RuntimeException(e);

}

}

@Override

**public** **void** **loggingAction**(String orderID) {

**final** Logger logger = (Logger) LoggerFactory.getLogger(XMLReportExporter.class);

logger.info("Logging orderID: {}", orderID);

}

}

//PDFExporter.java

public class PDFReportExporter extends ReportExporter {

**public** **PDFReportExporter**(String orderID, CustomerService.OrdersService ordersService) {

**super**(orderID, ordersService);

}

@Override

**public** ResponseEntity<**byte**[]> **export**(VatReport vatReport) {

**try** {

ByteArrayOutputStream baos = createInvoicePdf(vatReport);

HttpHeaders headers = **new** HttpHeaders();

headers.setContentType(MediaType.APPLICATION\_PDF);

headers.setContentDispositionFormData("inline", "vat\_report\_" + vatReport.getTimestamp() + ".pdf");

headers.setCacheControl("must-revalidate, post-check=0, pre-check=0");

// Download invoice PDF

**return** ResponseEntity.ok()

.headers(headers)

.body(baos.toByteArray());

} **catch** (NumberFormatException e) {

e.printStackTrace();

} **catch** (DocumentException e) {

throw new RuntimeException(e);

}

**return** **null**;

}

@Override

**public** **void** **loggingAction**(String orderID) {

**final** Logger logger = (Logger) LoggerFactory.getLogger(PDFReportExporter.class);

logger.info("Logging orderID: {}", orderID);

}

**private** ByteArrayOutputStream **createInvoicePdf**(VatReport vatReport) **throws** DocumentException {

ByteArrayOutputStream baos = **new** ByteArrayOutputStream();

Document document = **new** Document();

PdfWriter.getInstance(document, baos);

document.open();

Font titleFont = FontFactory.getFont(FontFactory.TIMES\_ROMAN, **20**, BaseColor.BLACK);

Paragraph title = **new** Paragraph("VAT REPORT", titleFont);

title.setAlignment(Element.ALIGN\_CENTER);

document.add(title);

Font contentFont = FontFactory.getFont(FontFactory.TIMES\_ROMAN, **13**, BaseColor.BLACK);

document.add(**new** Chunk("Customer name: " + vatReport.getCusName() + "\n", contentFont));

document.add(**new** Chunk("Date created: " + vatReport.getTimestamp().toString() + "\n", contentFont));

document.add(**new** Chunk("Sealer: " + vatReport.getPosFirstName() + ' ' + vatReport.getPosLastName() + "\n", contentFont));

document.add(**new** Chunk("Status: " + vatReport.getStatus() + "\n\n", contentFont));

Paragraph paragraph = **new** Paragraph();

paragraph.add(**new** Chunk("SN ", contentFont));

paragraph.add(**new** Chunk("Product name", contentFont));

paragraph.add(**new** Chunk(**new** VerticalPositionMark()));

paragraph.add(**new** Chunk("Quantity ", contentFont));

document.add(paragraph);

**int** i = **1**;

**for** (VatReportItem item : vatReport.getItems()) {

document.add(createDetailParagraph(i + " ", item.getProductName(), "" + item.getQuantity(), contentFont));

i++;

}

document.add(**new** Paragraph("----------------------------------------------------------------------------------------------------------------------------------"));

document.add(**new** Paragraph("Total Amount: " + String.format("%.2f", vatReport.getTotalAmount()) + "$"));

document.add(**new** Paragraph("Customer Given: " + String.format("%.2f", vatReport.getTotalAmount()) + "$"));

document.close();

**return** baos;

}

**private** Paragraph **createDetailParagraph**(String index, String productName, String quantity, Font contentFont) {

Paragraph paragraph = **new** Paragraph();

paragraph.add(**new** Chunk(index + " " + productName, contentFont));

paragraph.add(**new** Chunk(**new** VerticalPositionMark()));

paragraph.add(**new** Chunk(quantity, contentFont));

**return** paragraph;

}

}

//Maincode

**@GetMapping**("/report/VAT")

**public** ResponseEntity<**byte**[]> **exportFileVAT**(HttpServletRequest req, HttpServletResponse resp) **throws** JAXBException, IOException {

String vatType = req.getParameter("vat-type");

String orderID = req.getParameter("order-id-vat");

ReportExporter exporter;

**if** ("PDF".equals(vatType)) {

exporter = **new** PDFReportExporter(orderID,ordersService);

}

**else** {

exporter = **new** XMLReportExporter(orderID,ordersService);

}

**return** exporter.exportFileVAT();

}

2.4 Application of Factory Pattern

2.4.1 Introduce

* This code uses the Factory pattern to provide a centralized way to obtain an instance of PaymentProcessor based on a specified payment method.
* Here's why the factory pattern is useful in this context:
  + **Encapsulating object creation logic**: The factory encapsulates the logic that creates instances of PaymentProcessor based on the provided methods. This helps separate your client code from the instantiation details of the various payment processors.
  + **Promote modularity and extensibility**: Centralizing the creation of PaymentProcessor instances within a factory makes it easier to add support for new payment methods in the future. Simply add a new PaymentProcessor implementation and update the factory without changing your existing client code.
  + **Separate client code from concrete implementation**: Client code that requires a "payment processor" only needs to interact with the factory to provide the desired payment method. There is no need to know specific implementation classes such as "PaypalProcessor" or "CashProcessor", which promotes loose coupling and improves code maintainability.
  + **Easier dependency injection**: Using dependency injection (in this case @Autowired) to inject instances of PaypalProcessor and CashProcessor into a factory makes managing dependencies easier.
  + **Error Handling and Validation**: A factory can perform validation and error handling when retrieving a PaymentProcessor instance.

For example, if an unsupported payment method is requested, the factory can throw an exception and provide clear feedback to the client code

2.4.2 UML

A black screen with white text

Description automatically generated

2.4.3 How it works

* PaymentProcessor Interface*:*
  + Defines the contract for processing payments. It declares a method “processPayment(double amount)” that all payment processor implementations must implement.
* CashProcessor and PaypalProcessor Classes*:*
  + Implement the “PaymentProcessor” interface, providing concrete implementations for processing payments.
  + “CashProcessor” processes cash payments, while “PaypalProcessor” processes payments using the PayPal API.
* RedirectView Class*:*
  + Represents a view for redirection after a payment is processed. It's used to redirect users to different URLs based on the outcome of the payment process.
* PaymentFactory Class*:*
  + Acts as a factory for creating instances of “PaymentProcessor” based on the specified payment method.
  + It contains references to “CashProcessor” and “PaypalProcessor” instances, which are injected using “@Autowired” annotation.
  + The “getPaymentProcessor(String method)” method retrieves the appropriate “PaymentProcessor” instance based on the specified payment method. If the method is not supported, it throws an “IllegalArgumentException”.
* Usage*:*
  + In the application code, when there's a need to process a payment, the appropriate payment processor instance is obtained from the “PaymentFactory” based on the payment method.
  + For example, if a cash payment is to be processed, “PaymentFactory.getPaymentProcessor("cash")” will return an instance of “CashProcessor”. Similarly, if a PayPal payment is to be processed, “PaymentFactory.getPaymentProcessor("paypal")” will return an instance of “PaypalProcessor”.
  + Once the payment processor instance is obtained, the “processPayment(double amount)” method is called to process the payment

2.4.4 Apply code

//PaymentProcessor.java

@FunctionalInterface

public interface PaymentProcessor {

RedirectView **processPayment**(**double** amount);

}

//PayPalProcessor.java

@Service

@RequiredArgsConstructor

public class PaypalProcessor implements PaymentProcessor {

**private** **final** APIContext apiContext;

public Payment createPayment(

Double total,

String currency,

String method,

String intent,

String description,

String cancelUrl,

String successUrl

) **throws** PayPalRESTException {

Amount amount = **new** Amount();

amount.setCurrency(currency);

amount.setTotal(String.format(Locale.forLanguageTag(currency), "%.2f", total)); // 9.99$ - 9,99€

Transaction transaction = **new** Transaction();

transaction.setDescription(description);

transaction.setAmount(amount);

List<Transaction> transactions = **new** ArrayList<>();

transactions.add(transaction);

Payer payer = **new** Payer();

payer.setPaymentMethod(method);

Payment payment = **new** Payment();

payment.setIntent(intent);

payment.setPayer(payer);

payment.setTransactions(transactions);

RedirectUrls redirectUrls = **new** RedirectUrls();

redirectUrls.setCancelUrl(cancelUrl);

redirectUrls.setReturnUrl(successUrl);

payment.setRedirectUrls(redirectUrls);

**return** payment.create(apiContext);

}

public Payment executePayment(

String paymentId,

String payerId

) **throws** PayPalRESTException {

Payment payment = **new** Payment();

payment.setId(paymentId);

PaymentExecution paymentExecution = **new** PaymentExecution();

paymentExecution.setPayerId(payerId);

**return** payment.execute(apiContext, paymentExecution);

}

**public** RedirectView **paymentCreate**(String amount) {

**try** {

String cancelUrl = "http://localhost:8080/site/payment\_failed";

String successUrl = "http://localhost:8080/site/payment\_success";

String method = "Paypal";

String currency = "USD";

String description = "";

Payment payment = createPayment(

Double.valueOf(amount),

currency,

method,

"sale",

description,

cancelUrl,

successUrl

);

**for** (Links **links:** payment.getLinks()) {

**if** (links.getRel().equals("approval\_url")) {

**return** **new** **RedirectView**(links.getHref());

}

}

} **catch** (PayPalRESTException ignored) {

**return** **new** **RedirectView**("/site/payment\_failed");

}

**return** **new** **RedirectView**("/site/payment\_failed");

}

@Override

**public** RedirectView **processPayment**(**double** amount) {

**return** **paymentCreate**(String.valueOf(amount));

}

}

//CashProcessor.java

@Service

public class CashProcessor implements PaymentProcessor {

@Override

**public** RedirectView **processPayment**(**double** amount) {

System.out.println("Processing cash payment for $" + amount);

**return** **new** **RedirectView**("http://localhost:8080/site/payment\_success"); // Assuming successful transaction

}

}

//PaymentFactory.java

@Component

public class PaymentFactory {

@Autowired

**public** PaypalProcessor paypalProcessor;

@Autowired

**public** CashProcessor cashProcessor;

**public** PaymentProcessor **getPaymentProcessor**(String method) {

Map<String, PaymentProcessor> processors = **new** HashMap<>();

processors.put("paypal", paypalProcessor);

processors.put("cash", cashProcessor);

PaymentProcessor processor = processors.get(method);

**if** (processor == **null**) {

**throw** **new** **IllegalArgumentException**("Unsupported payment method: " + method);

}

**return** processor;

}

}

2.5 Application of Command Pattern

2.5.1 Introduce

In the department's application, the command pattern is applied to the javaMailSender library in sending verification emails when creating an account and sending order confirmation emails when paying at the counter. This will also make it easier for us to add new email types without having to change existing code, and they will also easily adapt to changing requirements in the future. Besides, we will also easily change the logic of email types without affecting the rest of the code.

2.5.2 UML

A screenshot of a computer

Description automatically generated

2.5.3 How it works

* **ICommand:** We create the “ICommand” interface andcreate the method execute here.
* **VerifyAccountMailSenderCommand:** This is an implementation of ICommand, used to send account confirmation emails. There is a constructor that accepts arguments like JavaMailSender, User, url, and verifyCode. These arguments are used to create the email body and send an email to the user to confirm the account. The execute() method is implemented to create and send emails.
* **ConfirmOrderMailSenderCommand:** This is another implementation of ICommand, used to send order confirmation emails. There is also a constructor that accepts arguments such as JavaMailSender, Customer, voucher\_name, total\_amount, and orderListCus. These arguments are used to create email content and send order confirmation emails to customers. The execute() method is implemented to create and send emails.
* **MailSenderInvoker:** This is a class to execute commands sequentially. There is a queue to store commands. The setCommand() method is used to add commands to the queue. The execute() method is used to iterate through the commands in the queue and call the execute() method of each command to execute them.

2.5.4 Apply code

ICommand.java

@FunctionalInterface

public interface ICommand {

void execute();

}

//VerifyAccountMailSenderCommand.java

public class VerifyAccountMailSenderCommand implements ICommand{

**private** **final** JavaMailSender javaMailSender;

**private** **final** User user;

**private** **final** String url;

**private** **final** String verifyCode;

**public** **VerifyAccountMailSenderCommand**(JavaMailSender mailSender, User user, String url, String verifyCode){

**this**.javaMailSender = mailSender;

**this**.user = user;

**this**.url = url;

**this**.verifyCode = verifyCode;

}

@Override

public void execute() {

String from = "phanluonghuy4623@gmail.com";

String to = **this**.user.getEmail();

String subject = "Account Verification";

String content = "Dear [[name]],<br>" + "Please click the link below to verify your registration:<br>"

+ "<h3><a href=\"[[URL]]\" target=\"\_self\">VERIFY</a></h3>" + "Thank you,<br>" + "Final Project with love <3";

**try** {

MimeMessage message = **this**.javaMailSender.createMimeMessage();

MimeMessageHelper helper = **new** MimeMessageHelper(message);

helper.setFrom(from, "Final Project");

helper.setTo(to);

helper.setSubject(subject);

content = content.replace("[[name]]", user.getFirst\_name() + " " + user.getLast\_name());

String siteUrl = **this**.url + "/log/verify?code=" + **this**.verifyCode;

System.out.println(siteUrl);

content = content.replace("[[URL]]", siteUrl);

helper.setText(content, **true**);

javaMailSender.send(message);

} **catch** (Exception e) {

e.printStackTrace();

}

}

}

//ConfirmOrderMailSenderCommand.java

public class ConfirmOrderMailSenderCommand implements ICommand{

**private** **final** JavaMailSender javaMailSender;

**private** **final** Customer customer;

**private** **final** String voucher\_name;

**private** **final** String total\_amount;

**private** **final** List<Object[]> orderListCus;

**public** **ConfirmOrderMailSenderCommand**(JavaMailSender mailSender, Customer customer, String voucher\_name, String total\_amount, List<Object[]> orderListCus){

**this**.javaMailSender = mailSender;

**this**.customer = customer;

**this**.voucher\_name = voucher\_name;

**this**.total\_amount = total\_amount;

**this**.orderListCus = orderListCus;

}

@Override

public void execute() {

**try** {

String from = "phanluonghuy4623@gmail.com";

String to = **this**.customer.getCustomer\_email();

String subject = "Invoice Verification";

MimeMessage message = **this**.javaMailSender.createMimeMessage();

MimeMessageHelper helper = **new** MimeMessageHelper(message);

helper.setFrom(from, "Final Project");

helper.setTo(to);

helper.setSubject(subject);

String content = "Dear [[name]],<br>" + "Please check your invoice:<br>";

content = content.replace("[[name]]", **this**.customer.getCustomer\_name() /\*+ " " + user.getLast\_name()\*/);

**for** (Object[] **olc:** **this**.orderListCus){

content += "<p style='margin-bottom: 0;'>"+ olc[**0**].toString()+" x "+olc[**2**].toString()+": "+olc[**3**].toString()+"$" +"</p>";

}

**if** (!**this**.voucher\_name.isEmpty()){

content += "<p style='margin-bottom: 0;'>Voucher: "+ **this**.voucher\_name +"</p>";

}

content += "Total bill: " + total\_amount + "$<br>Thank you,<br>" + "Final Project with love <3";

helper.setText(content, **true**);

**this**.javaMailSender.send(message);

} **catch** (Exception e) {

e.printStackTrace();

}

}

}

//MailSenderInvoker.java

public class MailSenderInvoker {

**private** **final** Queue<ICommand> commands;

public MailSenderInvoker(){

**this**.commands = **new** LinkedList<>();

}

**public** **void** **setCommand**(ICommand cmd){

commands.add(cmd);

}

public void execute(){

**while** (!commands.isEmpty()){

ICommand cmd = commands.poll();

cmd.execute();

}

}

}

2.6 Application of Adapter Pattern

2.6.1 Introduce

* In the department's application, the Adapter Pattern is applied to the "Export User" function. Instead of initially exporting an "EXCEL" file with the information of "User", when applying the Adapter pattern we will change the system by exporting a "User" information file in both "CSV" and XLSX file format.
* Using the Adapter Pattern helps integrate the "CSV" file export system into the old system without changing the code of the old system, and provides users with more file export options (XLSX and CSV). Additionally, the Adapter Pattern makes it easy to maintain the code because it separates the file export logic from the rest of the application.

2.6.2 UML

A screenshot of a computer

Description automatically generated

2.6.3 How it works

* First, we create a CSV file export system by creating an interface "ICSVReport" and a class "CSVReport", class "CSVReport" will implement the interface "ICSVReport"
* Next, we create a class "CSVReportAdapter", this class will implement our old system (interface "IXLSXReport")
* Then, "CSVReportAdapter" will convert the necessary data before passing them into the new system's method to ensure that the method can operate normally and still be able to integrate into the system. old in a flexible way:
  + **IXLSXReport:** In this class, we create a method export with a parameter that has a data type List<User> for the old system, which just exports files by XLSX file
  + **XLSXReport:** In this class, we implements to IXLSXReport and Override an export method, then we do an exportation in that method
  + **ICSVReport:** In this class, we create a method export with a parament have a data type is List<String[]> to export CSV file
  + **CSVReport:** In this class, we implements to ICSVReport and Override an export method, then we do an exportation in that method
  + **CSVReportAdapter:** In this class, we implements IXLSXReport(old system), then Override an exportation of that interface, and then in that method, we convert List<User> data type to List<String[]> data type to use an exportation of the new system

2.6.4 Apply code

//IXLSXReport.java

@FunctionalInterface

public interface IXLSXReport {

ResponseEntity<**byte**[]> **exportReportXLSXMethod**(List<User> userList);

}

//XLSXReport.java

@Service

public class XLSXReport implements IXLSXReport {

@Override

**public** ResponseEntity<**byte**[]> **exportReportXLSXMethod**(List<User> userList) {

**byte**[] excelBytes = createExcelBytes(userList);

HttpHeaders headers = **new** HttpHeaders();

headers.add(HttpHeaders.CONTENT\_DISPOSITION, "attachment; filename=user\_data.xlsx");

**return** ResponseEntity.ok()

.headers(headers)

.contentType(MediaType.parseMediaType("application/vnd.openxmlformats-officedocument.spreadsheetml.sheet"))

.body(excelBytes);

}

**private** **byte**[] **createExcelBytes**(List<User> userList) {

**try** (Workbook workbook = **new** XSSFWorkbook();

ByteArrayOutputStream outputStream = **new** ByteArrayOutputStream()) {

Sheet sheet = workbook.createSheet("User Data");

Row headerRow = sheet.createRow(**0**);

String[] headers = {"ID", "First name", "Last name", "Email", "Phone number", "Address", "Birthday", "Gender"};

**for** (**int** i = **0**; i < headers.length; i++) {

Cell cell = headerRow.createCell(i);

cell.setCellValue(headers[i]);

}

**int** rowNum = **1**;

**for** (User user : userList) {

Row row = sheet.createRow(rowNum++);

row.createCell(**0**).setCellValue(user.getUser\_id());

row.createCell(**1**).setCellValue(user.getFirst\_name());

row.createCell(**2**).setCellValue(user.getLast\_name());

row.createCell(**3**).setCellValue(user.getEmail());

row.createCell(**4**).setCellValue(user.getPhone\_number());

row.createCell(**5**).setCellValue(user.getAddress());

row.createCell(**6**).setCellValue(user.getBirthday().toString());

row.createCell(**7**).setCellValue(user.getGender());

}

workbook.write(outputStream);

**return** outputStream.toByteArray();

} **catch** (IOException e) {

e.printStackTrace();

**return** **null**;

}

}

//ICSVReport.java

@FunctionalInterface

public interface ICSVReport {

ResponseEntity<**byte**[]> **exportReportCSVMethod**(List<String[]> userList);

}

//CSVReport.java

@Service

public class CSVReport implements ICSVReport {

@Override

**public** ResponseEntity<**byte**[]> **exportReportCSVMethod**(List<String[]> userListData) {

List<String[]> data = **new** ArrayList<>();

data.add(**new** String[]{"ID", "First Name", "Last Name", "Email", "Phone Number", "Address", "Birthday", "Gender"});

data.addAll(userListData);

**byte**[] csvBytes = convertCsvToBytes(data);

HttpHeaders headers = **new** HttpHeaders();

headers.setContentType(MediaType.APPLICATION\_OCTET\_STREAM);

headers.setContentDispositionFormData("attachment", "user\_data.csv");

**return** ResponseEntity.ok()

.headers(headers)

.body(csvBytes);

}

**private** **byte**[] **convertCsvToBytes**(List<String[]> data) {

StringBuilder csvContent = **new** StringBuilder();

**for** (String[] rowData : data) {

**for** (String cellData : rowData) {

csvContent.append(cellData).append(",");

}

csvContent.deleteCharAt(csvContent.length() - **1**).append("\n");

}

**return** csvContent.toString().getBytes();

}

//CSVReportAdapter.java

@Service

public class CSVReportAdapter implements IXLSXReport {

**private** ICSVReport ICSVReport;

public CSVReportAdapter(){

**this**.ICSVReport = **new** CSVReport();

}

@Override

**public** ResponseEntity<**byte**[]> **exportReportXLSXMethod**(List<User> userList) {

List<String[]> userListData = **new** ArrayList<>();

**for** (User **user:** userList){

userListData.add(**new** String[]{

user.getUser\_id(),

user.getFirst\_name(),

user.getLast\_name(),

user.getEmail(),

user.getPhone\_number(),

user.getAddress(),

user.getBirthday().toString(),

user.getGender()

});

}

**return** ICSVReport.exportReportCSVMethod(userListData);

}

}

2.7 Application of Decorator Pattern

2.7.1 Introduce

In our project, we implemented the decorator pattern to enhance the export functionality of our product list. The export process is crucial for our application as it allows users to extract product data in various formats for analysis and sharing purposes. We recognized the need for a flexible and scalable solution that would accommodate future export requirements without introducing complexity or compromising code maintainability.

2.7.2 UML

A screenshot of a graph

Description automatically generated

2.7.3 How it works

* **Export Interface**: We defined an Export interface that serves as the contract for all export classes. This interface outlines the basic methods required for exporting product data, ensuring consistency across different export implementations.
* **Normal Export Class**: We implemented a concrete class called NormalExport that provides the baseline export functionality. This class handles the standard export process without any additional features.
* **Export Decorator**: We introduced an ExportDecorator abstract class that acts as the base class for all export decorators. This class implements the Export interface and maintains a reference to the underlying Export object, allowing it to augment or modify the export behavior.
* **Compress Decorator**: To demonstrate the utility of the decorator pattern, we created a specific decorator called CompressDecorator, which extends ExportDecorator. This decorator adds the capability to compress the exported data before saving it to a file or transmitting it over a network. By applying the CompressDecorator to an Export object, users can opt to compress the exported data without altering the core export functionality.

2.7.4 Apply code

//Export.java

**public** **interface** **Export** {

**byte**[] **export**(List<Product> productList);

}

//ExportDecorator.java

**public** **abstract** **class** **ExportDecorator** **implements** Export{

**protected** Export wrapObj;

**public** **ExportDecorator** (Export wrapObj) {

**this**.wrapObj = wrapObj;

}

**@Override**

**public** **byte**[] **export**(List<Product> productList) {

**return** wrapObj.export(productList);

}

}

//NormalExport.java

**public** **class** **NormalExport** **implements** Export {

**@Override**

**public** **byte**[] **export**(List<Product> productList) {

**return** **generateCSVData**(productList);

}

**private** **byte**[] **generateCSVData**(List<Product> productList) {

ByteArrayOutputStream byteArrayOutputStream = **new** ByteArrayOutputStream();

**try** (CSVWriter writer = **new** CSVWriter(**new** OutputStreamWriter(byteArrayOutputStream, StandardCharsets.UTF\_8))) {

writer.writeNext(**new** String[]{"Product name", "Import price", "Retail price", "Category", "Quantity",

"Date Created", "Description","Barcode"});

**for** (Product product : productList) {

String[] userData = {product.getProduct\_name(), String.valueOf(product.getProduct\_price()),

String.valueOf(product.getRetail\_price()), String.valueOf(product.getCategory\_id()),

String.valueOf(product.getQuantity\_stock()), String.valueOf(product.getDate\_created()),

String.valueOf(product.getDescription()), String.valueOf(product.getBarcode())};

writer.writeNext(userData);

}

} **catch** (IOException e) {

// Handle the exception, e.g., log it or return an error response

e.printStackTrace();

}

**return** byteArrayOutputStream.toByteArray();

**public** **class** **CompressDecorator** **extends** ExportDecorator{

**public** **CompressDecorator**(Export wrapObj) {

**super**(wrapObj);

}

**@Override**

**public** **byte**[] **export**(List<Product> productList) {

**byte**[] exportedData = wrapObj.export(productList);

// byte[] compressedData = compress(exportedData, userList);

**return** **compress**(exportedData);

}

**private** **byte**[] **compress**(**byte**[] data) {

ByteArrayOutputStream byteArrayOutputStream = **new** ByteArrayOutputStream();

**try** (ZipOutputStream zipOutputStream = **new** ZipOutputStream(byteArrayOutputStream)) {

ZipEntry zipEntry = **new** ZipEntry("product\_report.csv");

zipOutputStream.putNextEntry(zipEntry);

zipOutputStream.write(data);

zipOutputStream.closeEntry();

} **catch** (IOException e) {

e.printStackTrace();

}

**return** byteArrayOutputStream.toByteArray();

}

}

}

2.8 Application of Builder Pattern

2.8.1 Introduce

* In the subject's application, the Builder Pattern is applied to the User object because that object has too many properties that need to be passed in, which helps us to control the object's properties more easily. Besides, it makes code easier to read and understand, especially for Objects with many complex Attributes.
* Builder Pattern simplifies the creation of Objects “User” with multiple Attributes by breaking the creation process into smaller steps. Additionally, Object construction steps are strictly controlled, helping to avoid errors caused by passing incorrect or missing values

2.8.2 UML

A screen shot of a computer

Description automatically generated

2.8.3 How it works

* First, we create a Builder interface class named “IBuilder”, here we will declare setter methods for the properties of the class “User” and a method “build()”
* Next, create a ConcreateBuilder class named "UserBuilder", class "UserBuilder" will implement the interface "IUserBuilder" and implement the methods of that interface class, the setter methods will always return IUserBuilder and the build method. () will return a User object.
  + **IUserBuilder**: create setter method of all attribute of the Object and method build()
  + **UserBuilder**: implements to “IUserBuilder” and Override all method of it, in here, with the setter method, we will set the corresponding attribute and return “this”(current Object), with method build(), we create a new Object then return to this Object.

2.8.4 Apply code

//IUserBuilder.java

**public** **interface** **IUserBuilder** {

IUserBuilder **setUserIdBuilder**(String user\_id);

IUserBuilder **setFirstNameBuilder**(String first\_name);

IUserBuilder **setLastNameBuilder**(String lastName);

IUserBuilder **setEmailBuilder**(String email);

IUserBuilder **setPhoneNumberBuilder**(String phone\_number);

IUserBuilder **setAddressBuilder**(String address);

IUserBuilder **setImgBuilder**(String img);

IUserBuilder **setAccIdBuilder**(String acc\_id);

IUserBuilder **setBirthdayBuilder**(Date dob);

IUserBuilder **setGenderBuilder**(String gender);

User **build**();

}

//UserBuilder.java

**public** **class** **UserBuilder** **implements** IUserBuilder{

**private** String user\_id;

**private** String first\_name;

**private** String last\_name;

**private** String email;

**private** String phone\_number;

**private** String address;

**private** String image;

**private** String account\_id;

**private** Date birthday;

**private** String gender;

**@Override**

**public** IUserBuilder **setUserIdBuilder**(String user\_id) {

**this**.user\_id = user\_id;

**return** **this**;

}

**@Override**

**public** IUserBuilder **setFirstNameBuilder**(String first\_name) {

**this**.first\_name = first\_name;

**return** **this**;

}

**@Override**

**public** IUserBuilder **setLastNameBuilder**(String lastName) {

**this**.last\_name = lastName;

**return** **this**;

}

**@Override**

**public** IUserBuilder **setEmailBuilder**(String email) {

**this**.email = email;

**return** **this**;

}

**@Override**

**public** IUserBuilder **setPhoneNumberBuilder**(String phone\_number) {

**this**.phone\_number = phone\_number;

**return** **this**;

}

**@Override**

**public** IUserBuilder **setAddressBuilder**(String address) {

**this**.address = address;

**return** **this**;

}

**@Override**

**public** IUserBuilder **setImgBuilder**(String img) {

**this**.image = img;

**return** **this**;

}

**@Override**

**public** IUserBuilder **setAccIdBuilder**(String acc\_id) {

**this**.account\_id = acc\_id;

**return** **this**;

}

**@Override**

**public** IUserBuilder **setBirthdayBuilder**(Date dob) {

**this**.birthday = dob;

**return** **this**;

}

**@Override**

**public** IUserBuilder **setGenderBuilder**(String gender) {

**this**.gender = gender;

**return** **this**;

}

**@Override**

**public** User **build**() {

**return** **new** **User**(user\_id, first\_name, last\_name, email, phone\_number, address, image, account\_id, birthday, gender);

}

}

//UserController.java

IUserBuilder userBuilder = **new** UserBuilder()

.setUserIdBuilder(maxID)

.setFirstNameBuilder(firstname)

.setLastNameBuilder(lastname)

.setEmailBuilder(email)

.setPhoneNumberBuilder(phone)

.setAddressBuilder(address)

.setImgBuilder("user\_profile.png")

.setAccIdBuilder(acc\_id)

.setBirthdayBuilder(date)

.setGenderBuilder(gender);

User user = userBuilder.build();

userService.addUser(user);

2.9 Application of Proxy Pattern

2.9.1 Introduce

* In the department's application, the Proxy Pattern is applied to the "addCustomer" function. Instead of initially combining the pre-test parts before adding a Customer object to the "CustomerService" class, now we will create an additional class named "ProxyCustomerService" to handle pre-checking tasks before calling the method to add a Customer object in the "CustomerService" class.
* Apply Proxy Pattern allows validation logic to be separated from customer add logic, making it easy to change and extend validation logic without affecting add customer logic. Additionally, it also allows the validation to be performed in a separate layer, helping to protect validation logic from unauthorized access.

2.9.2 UML

A screenshot of a diagram

Description automatically generated

2.9.3 How it works

* First, create an “ICustomerService”, this class can be abstract or interface and contains methods used in “ProxyCustomerService” and “CustomerService”
* Next, class "ProxyCustomerService" and implement "ICustomerService", in this class we will check the input data before passing them to "CustomerService"
  + **ICustomerService:** In this class, we create a method that is used in the proxy and real class
  + **ProxyCustomerService:** In this class, we implements to “ICustomerService” and then Override “addCustomer()” method, besides, we create a check validation function to check the valid input before adding the customer
  + **CustomerService:** In this class, we also implements “ICustomerService” and Override the “addCustomer” method, but here, we just implement the adding customer code

2.9.4 Apply code

//ICustomerService.java

**@FunctionalInterface**

**public** **interface** **ICustomerService** {

String **addCustomer**(Customer customer);

}

/ProxyCustomerService.java

**@Service**

**public** **class** **ProxyCustomerService** **implements** ICustomerService{

**@Autowired**

**private** CustomerService customerService;

**@Override**

**public** String **addCustomer**(Customer customer) {

// if (isValidCustomer(customer).equals("Success")){

// return customerService.addCustomer(customer);

// }

**return** **isValidCustomer**(customer).equals("Success") ? customerService.addCustomer(customer): isValidCustomer(customer);

}

**private** String **isValidCustomer**(Customer customer){

System.out.println(customerService.findCusByPhone(customer.getCustomer\_phone()));

**if** (customer.getCustomer\_name().isEmpty() || customer.getCustomer\_email().isEmpty()

|| customer.getCustomer\_address().isEmpty() || customer.getCustomer\_phone().isEmpty()){

**return** "You must fill in all fields !";

} **else** **if** (!customer.getCustomer\_email().contains("@")){

**return** "Invalid email format !";

} **else** **if** (customerService.findCusByPhone(customer.getCustomer\_phone()) != **null**) {

**return** "Phone is already existed !";

}

**return** "Success";

}

}

2.10 Application of ChainOfResponsibility Pattern

2.10.1 Introduce

* In our system, we have implemented the PasswordHandler interface along with three concrete implementations: EmptyPasswordHandler, MinLengthPasswordHandler, and PasswordMatchHandler.
* Using this pattern allows us to add new validation rules easily without modifying existing code. Additionally, each handler in the chain focuses on just one separate validation rule, making the code easier to understand and maintain.
* Besides, it helps create input change password validation code that is easy to understand, easy to maintain and extend

2.10.2 UML

A graph with a line and a line

Description automatically generated with medium confidence

2.10.3 How it works

* EmptyPasswordHandler:
  + This handler is responsible for checking if the password provided by the user is empty.
  + It ensures that the user cannot proceed with an empty password, thereby enhancing the security of our system.
  + The EmptyPasswordHandler implements the PasswordHandler interface and contains logic to verify if the password is empty. If it is, the handler stops further processing and returns an appropriate response to the client.
* MinLengthPasswordHandler:
* This handler enforces a minimum length requirement for the password.
  + It ensures that passwords meet a certain level of complexity, contributing to the overall security of our system.
  + The MinLengthPasswordHandler implements the PasswordHandler interface and includes logic to check if the length of the password meets the specified minimum requirement. If the password length is insufficient, the handler halts further processing and returns an appropriate response.
* PasswordMatchHandler:
  + This handler verifies if the user's password matches a predefined pattern or criteria.
  + It is commonly used for enforcing password policies such as requiring a combination of uppercase and lowercase letters, numbers, and special characters.
  + The PasswordMatchHandler implements the PasswordHandler interface and incorporates logic to validate if the password matches the specified criteria. If the password fails to meet the requirements, the handler terminates further processing and returns an appropriate response to the client.
* These implementations adhere to the Chain of Responsibility pattern, allowing for the sequential processing of password validation checks. Each handler encapsulates a specific validation rule, promoting modularity, flexibility, and maintainability in our system. Additionally, the use of interfaces and concrete implementations follows object-oriented design principles, facilitating extensibility and code reusability.

2.10.4 Apply code

//PasswordHandler.java

**public** **interface** **PasswordHandler** {

**boolean** **handleRequest**(String password, String confirmPassword, HttpSession session);

**void** **setNextHandler**(PasswordHandler nextHandler);

}

//PasswordMatchHandler.java

**public** **class** **PasswordMatchHandler** **implements** PasswordHandler {

**private** PasswordHandler nextHandler;

**@Override**

**public** **boolean** **handleRequest**(String password, String confirmPassword, HttpSession session) {

**if** (!password.equals(confirmPassword)) {

session.setAttribute("password", "Confirm password didn't match");

**return** **true**;

}

**return** nextHandler != **null** && nextHandler.handleRequest(password, confirmPassword, session);

}

**@Override**

**public** **void** **setNextHandler**(PasswordHandler nextHandler) {

**this**.nextHandler = nextHandler;

}

}

//MinLengthPasswordHandler.java

**public** **class** **MinLengthPasswordHandler** **implements** PasswordHandler {

**private** PasswordHandler nextHandler;

**@Override**

**public** **boolean** **handleRequest**(String password, String confirmPassword, HttpSession session) {

**if** (password.length() < **6**) {

session.setAttribute("password", "Password must have at least 6 characters");

**return** **true**;

}

**return** nextHandler != **null** && nextHandler.handleRequest(password, confirmPassword, session);

}

**@Override**

**public** **void** **setNextHandler**(PasswordHandler nextHandler) {

**this**.nextHandler = nextHandler;

}

}

//EmptyPasswordHandler.java

**public** **class** **EmptyPasswordHandler** **implements** PasswordHandler {

**private** PasswordHandler nextHandler;

**@Override**

**public** **boolean** **handleRequest**(String password, String confirmPassword, HttpSession session) {

**if** (password.isEmpty() || confirmPassword.isEmpty()) {

session.setAttribute("password", "You must fill in all fields");

**return** **true**;

}

**return** nextHandler != **null** && nextHandler.handleRequest(password, confirmPassword, session);

}

**@Override**

**public** **void** **setNextHandler**(PasswordHandler nextHandler) {

**this**.nextHandler = nextHandler;

}

}

CHAPTER 3 – REFERENCES

* Refactoring.Guru. Design patterns. Refactoring.Guru. https://refactoring.guru/design-patterns
* Design patterns and refactoring. https://sourcemaking.com/design\_patterns