UNIVERSITY of GREENWICH
Alliance with FPT Education

BTEC

ASSIGNMENT 1 FRONT SHEET

Qualification	**BTEC Level 5 HND Diploma in Computing**		
Unit number and title	Unit 20: Advanced Programming		
Submission date	8/12/2023	**Date Received 1st submission**	8/12/2023
Re-submission Date		**Date Received 2nd submission**	
Student Name	Mai Trung Duc	**Student ID**	GCH211004
Class	GCH1106	**Assessor name**	Le Viet Bach
Student declaration I certify that the assignment submission is entirely my own work and I fully understand the consequences of plagiarism. Iunderstand that making a false declaration is a form of malpractice.			
		Student's signature	Duc

Grading grid

P1	P2	M1	M2	D1	D2

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

Table Content

1 Introduction 4
2 OOP general concepts 4
2.1 Introduce OOP, general concepts of OOP 4
3 OOP scenario 15
3.1 Scenario 15
Description of OOP scenario: 15
3.2 Usecase Diagram 17
3.3 Class Diagram 23
Explain 23
4 Design Patterns 24
Introduce design pattern, general concepts of Design Pattern 24
4.1 Creational pattern 26
Introduce creational pattern 26
Description of a creational scenario 34
4.2 Structural pattern 39
Introduce creational pattern 39
Description of a structural scenario 51
4.3 Behavioral pattern 54
Introduce creational pattern 54
Description of a behavioral scenario 74

5 Design Pattern vs OOP 76
6 Conclusion 77
References 78

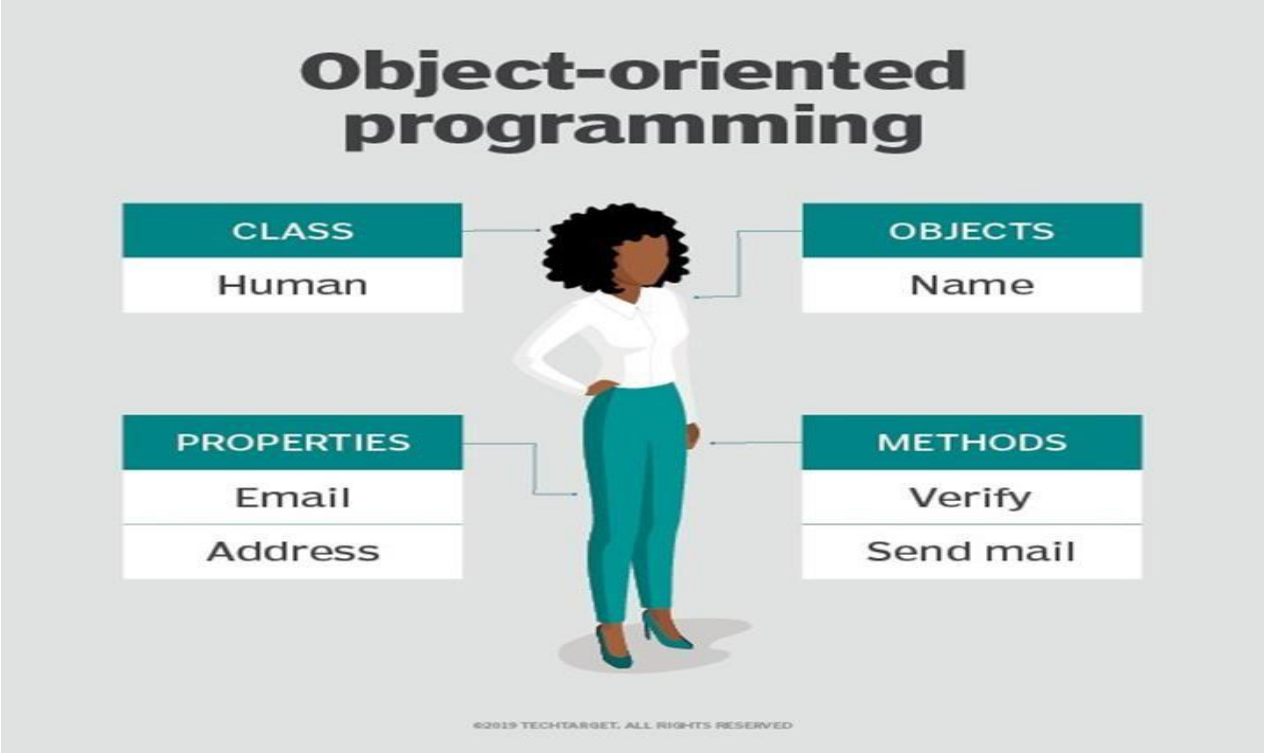
1. Introduction

In order for readers to comprehend object-oriented programming (OOP) before the project that the author is modeling for implementation is presented, this report attempts to provide them a fundamental understanding of OOP by elucidating the ideas included in the paradigm. Basic OOP concepts like Abstraction, Polymorphism, Inheritance & Encapsulation (APIE), Single Responsibility Principle, Open Closure Principle, Liskov Substitution Principle, Interface Separation Principle, Dependency Inversion Principle (SOLID), diagrams, and design patterns are covered in this section. In addition, this study provides readers with an easy-to-understand scenario and class diagram for each design pattern, which includes behavioural, structural, and creative design patterns.

2. OOP general concepts

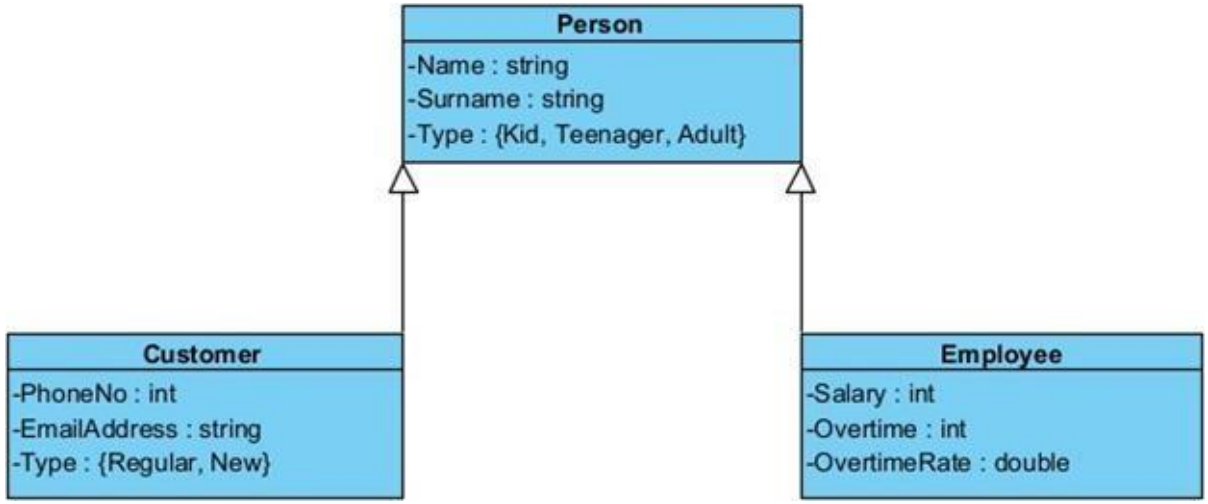
a. Introduce OOP, general concepts of OOP

Object-oriented programming is a way of writing code in which programs are organized as groups of objects that work together. Each object is an instance of a class, and the classes of the objects are all connected through inheritance relationships to form a hierarchy.



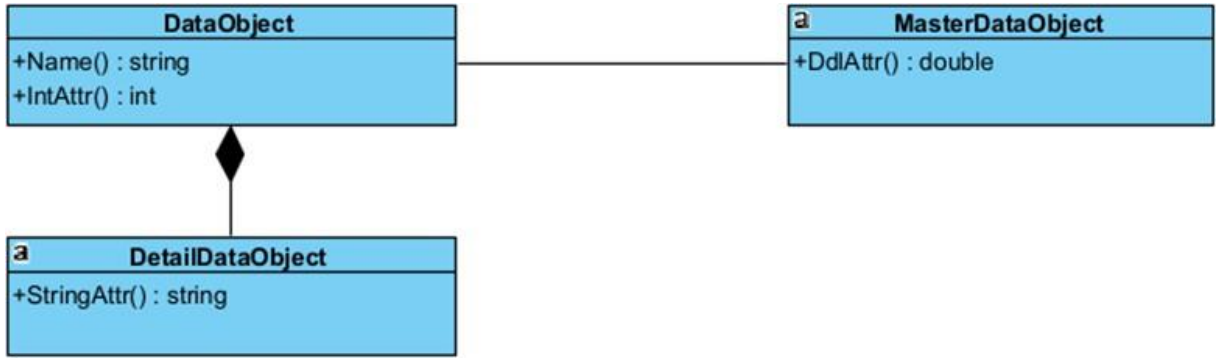
A, Class

In object-oriented programming (OOP), a class is a blueprint or template that defines the structure, attributes, and behaviors of objects. It serves as a blueprint from which individual objects are created. A class encapsulates data (attributes) and behavior (methods) related to a specific concept or entity.


Person
-Name : string
-Surname : string
-Type : {Kid, Teenager, Adult}
Customer
-PhoneNo : int
-EmailAddress : string
-Type : {Regular, New}
Employee
-Salary : int
-Overtime : int
-OvertimeRate : double

B, Object

Are instances of a class created with specifically defined data. Objects can correspond to real-world objects or an abstract entity. When a class is initially defined, the description is the only object that is defined.



C, Constructor

In object-oriented programming, a constructor is a unique method of a class or structure that initializes a newly formed object of that kind. Automatically, the constructor is invoked whenever an object is created. Similar to an instance method, a constructor sets the member values of an object to user-defined or default values. It typically shares the same name as the class. A constructor is not a legitimate method, despite its resemblance, because it lacks a return type. The constructor initializes the object; it cannot be static, final, abstract, or synchronized. Instead, it completes a task by running code.

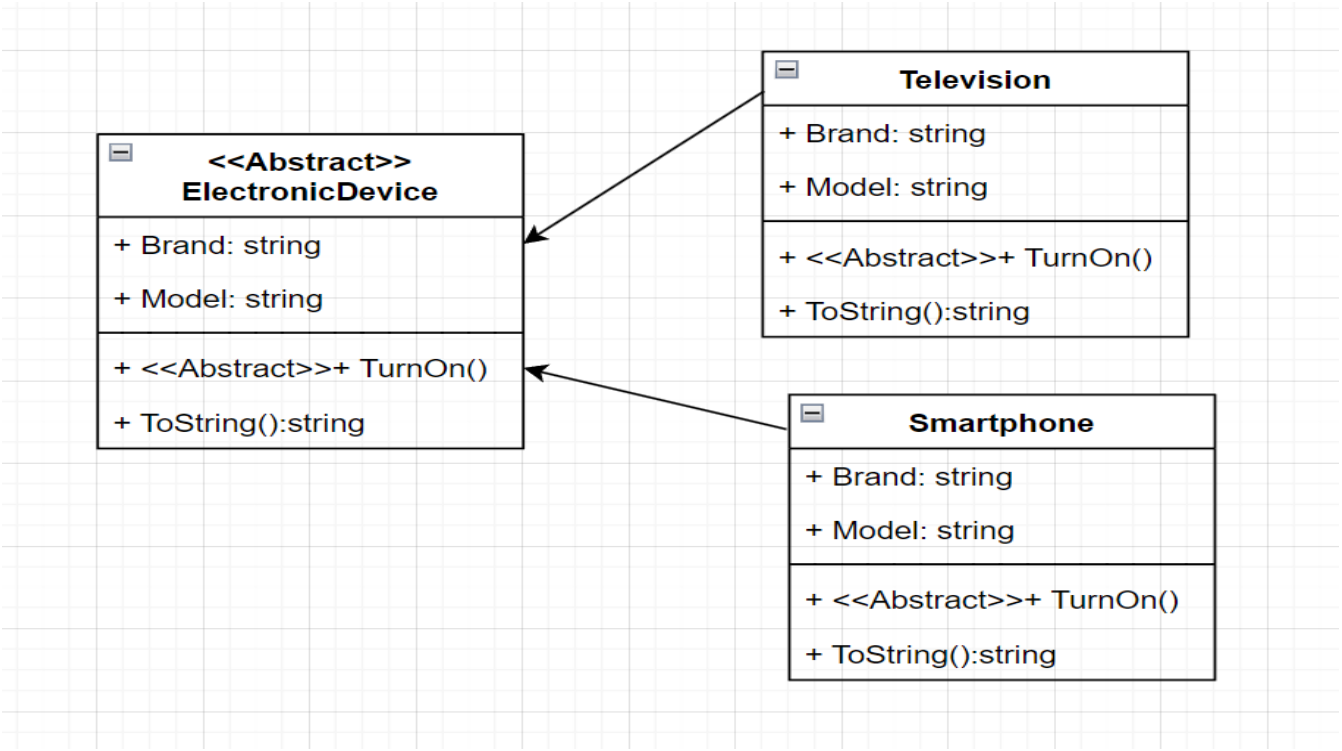
```
using System;

// Define the Person class
class Person
{
    // Properties
    public string Name { get; set; }
    public int Age { get; set; }

    // Constructor
    public Person(string name, int age)
    {
        Name = name;
        Age = age;
    }

    // Method to introduce the person
    public void Introduce()
    {
        Console.WriteLine($"Hello, my name is {Name} and I am {Age} years old.");
    }
}
```

Abstraction

Abstraction is a principle that focuses on providing simplified representations of complex systems. It allows the creation of abstract classes or interfaces that define the essential characteristics and behaviors without providing the implementation details.

Abstraction helps in managing complexity by hiding unnecessary details and exposing only what is necessary for interaction. It allows developers to focus on high-level concepts and ignore low-level implementation complexities.

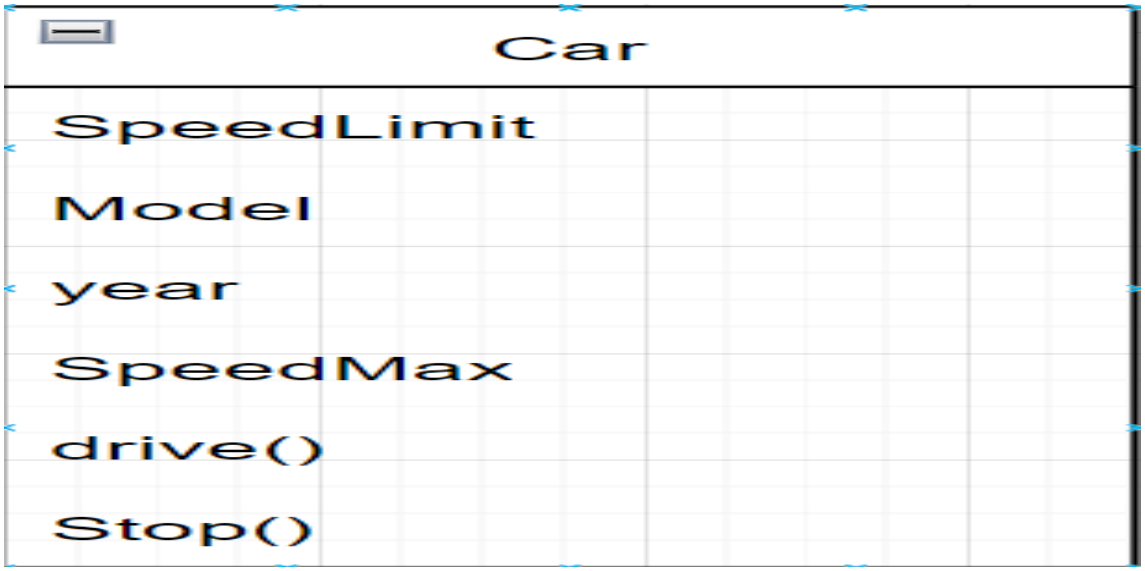
In many programming languages, abstract classes or interfaces cannot be instantiated directly but serve as blueprints for derived classes. Derived classes inherit the abstract class's structure and provide their own implementation.



Encapsulation

Encapsulation is one of the core principles of object-oriented programming. It refers to the bundling of data and methods within an object and the restriction of access to the internal details of the object from the outside. This provides data abstraction, where the internal state of an object is hidden and only a well-defined interface is exposed to interact with the object.

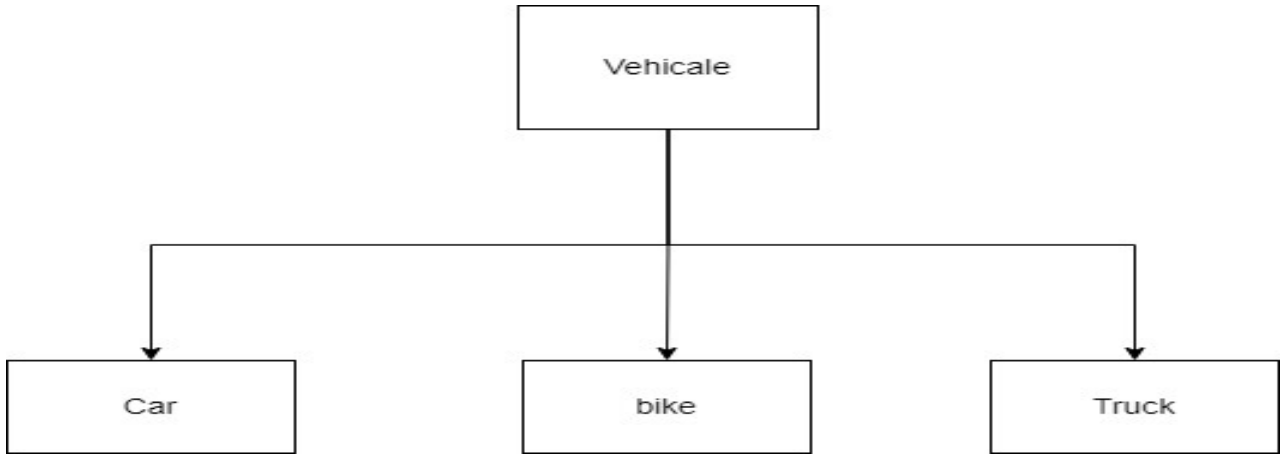
Encapsulation helps in achieving data security and integrity. By hiding the implementation details, the object can control how its data is accessed and modified. This prevents unauthorized access and ensures that the object's data is consistent and valid.



Inheritance

Inheritance is a fundamental principle of OOP that allows the creation of new classes (derived classes) based on existing classes (base or parent classes). The derived classes inherit the attributes and methods of the base class and can also add new attributes or override inherited methods.

Inheritance promotes code reuse and provides a way to model hierarchical relationships between classes. It allows common attributes and behaviors to be defined in a base class and shared among multiple derived classes. This reduces code duplication and makes the codebase more maintainable and modular.



Polymorphism

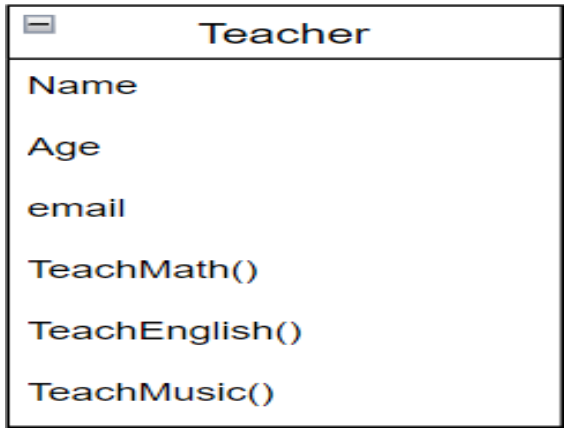
Polymorphism is the ability of objects of different classes to be used interchangeably, based on a common interface. It allows objects to be treated as instances of their base class or any of their derived classes. Polymorphism enables code to be written in a generic way, making it more flexible and reusable. Polymorphism is often achieved through method overriding and method overloading. Method overriding occurs when a derived class provides its own implementation of a method that is already defined in the base class. Method overloading, on the other hand, allows multiple methods with the same name but different parameters to be defined within a class.

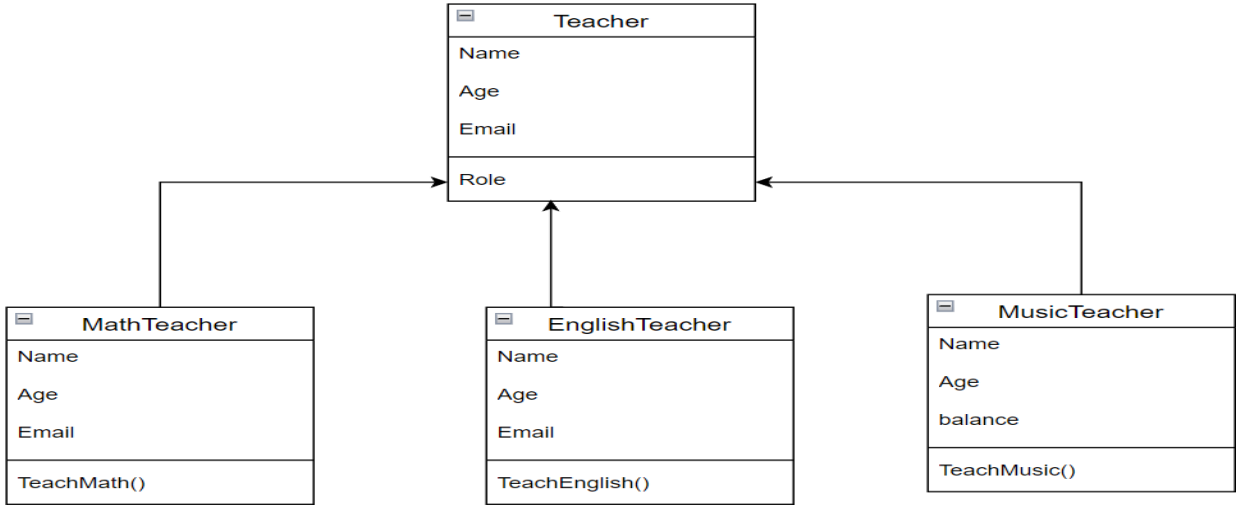
SOLID

SOLID are the five basic principles that help in creating good software architecture. SOLID is an acronym where S stands for SRP (Single Responsibility Principle).
O stands for OCP (Open Closed Principle) and L stands for LSP (Liskov Substitution Principle). I stand for the ISP (Interface Segregation Principle).
D stands for DIP (Dependency Inversion Principle).

i. Single Responsibility Principle

A key tenet of object-oriented programming is the Single Reason Principle (SRP), which asserts that a class should only have one change cause. It implies that a class should not be in charge of several unconnected tasks but rather should have a single duty or goal.

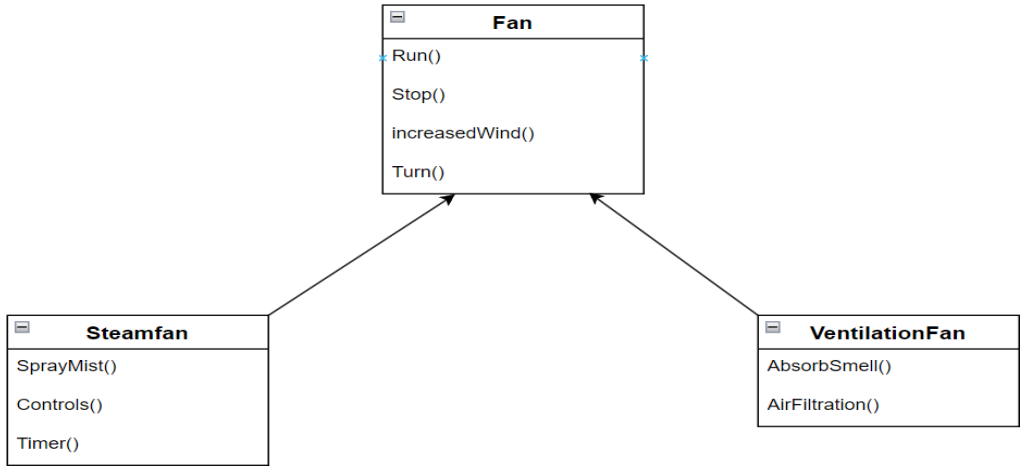




Teaching class contains up to 3 responsibilities, so I divided into 3 classes: math, English and music teachers

ii. Open Closed Principle

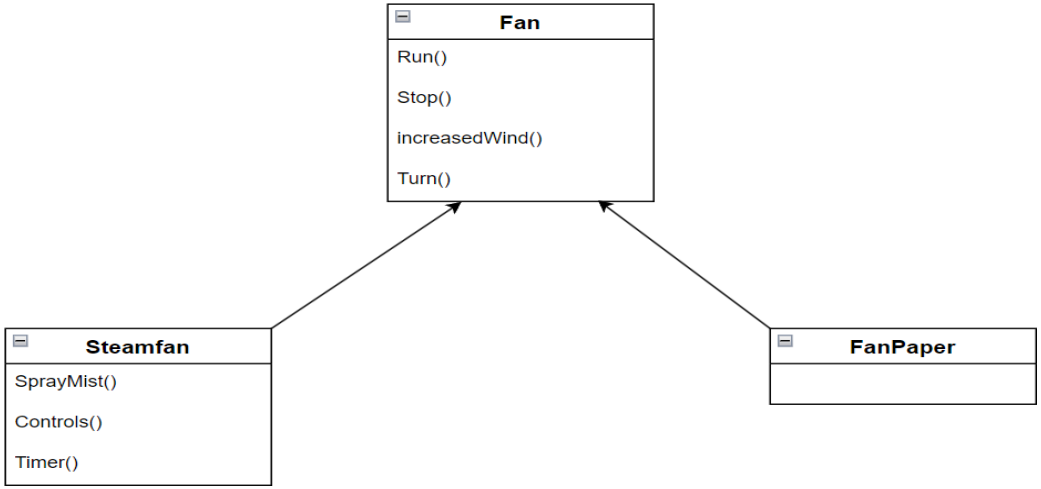
The Open/Closed Principle (OCP) states that software entities (classes, modules, functions, etc.) should be open for extension but closed for modification. It suggests that we should design our code in a way that allows for easy extension without modifying existing code.



We can declare a new class called Steamfan and VentilationFan that inherits the features of the original Fan class and adds new features, rather than changing the Fan class, which does not fully suit our needs.

iii. Liskov Substitution Principle

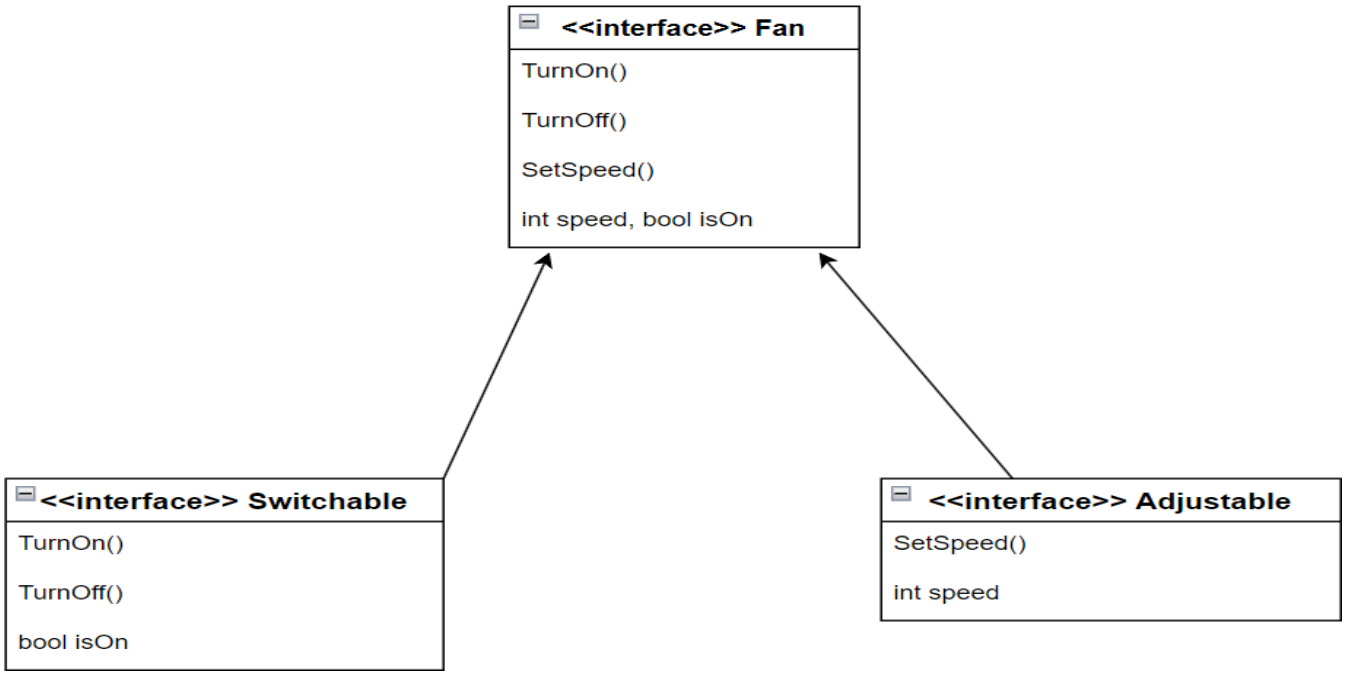
The Liskov Substitution Principle (LSP) states that objects belonging to a superclass should be interchangeable with objects belonging to a subclass without compromising the program's validity.



We have a fan class. The steam fan class can inherit this class. However, paper fans need to have wind or power to run or theywill cause errors. So it violates the liskov substitution principle.

iv. Interface Segregation Principle

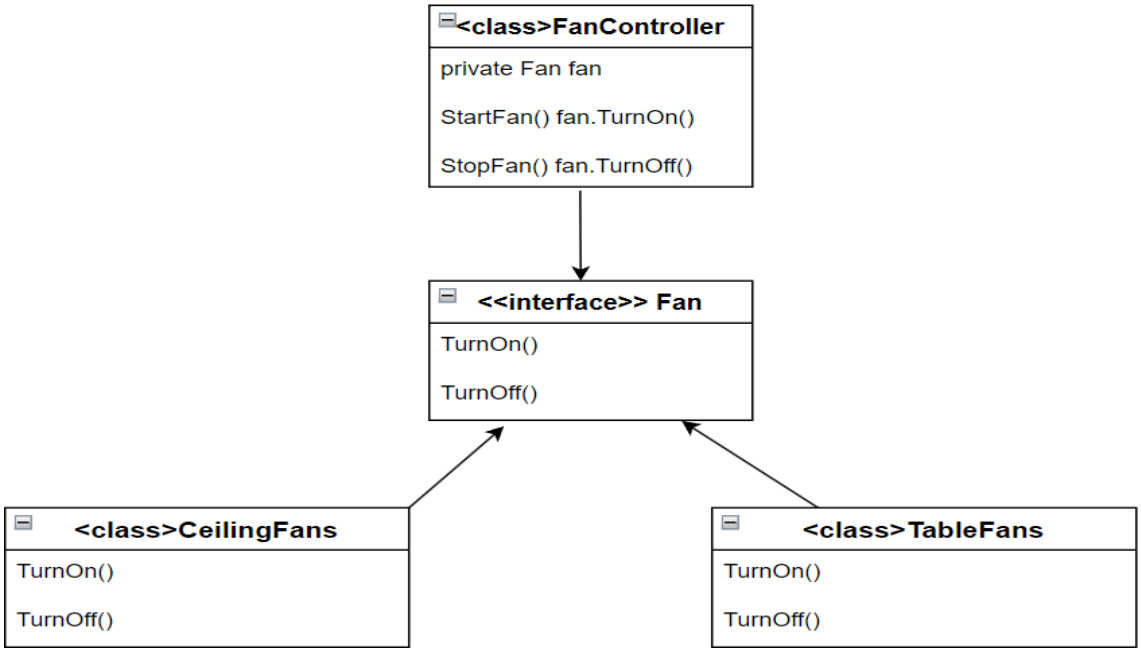
A class shouldn't be compelled to implement interfaces it doesn't utilize, according to the Interface Segregation Principle (ISP). This principle deals with designing customized interfaces for customized client requirements.



Instead of using one big interface for the Fan, we should separate it into many small interfaces that can be switched and adjusted, for many specific purposes, the speed can be turned off or adjusted.

v. Dependency inversion principle

Dependency Inversion Principle (DIP): Both high-level and low-level modules should rely on abstractions rather than the other way around. Details should not depend on abstractions; rather, abstractions should depend on details.



Fan Controller is a high-level module that does not depend on low-module Table Fans, or Ceiling Fans. Both depend on the Fan abstract class. The Ceiling Fan and Desk Fan classes are low-level modules that implement the Fan interface.

3. OOP scenario

a. Scenario

Description of OOP scenario:

Nowadays refrigerators play a very important role in human life. It stores food and preserves it for long-term use. There are many different types of refrigerators on the market. Our group decided to make a Scenario about refrigerators so everyone can better understand it. The purpose of the system is to help users clearly understand the structure and functions of refrigerators.

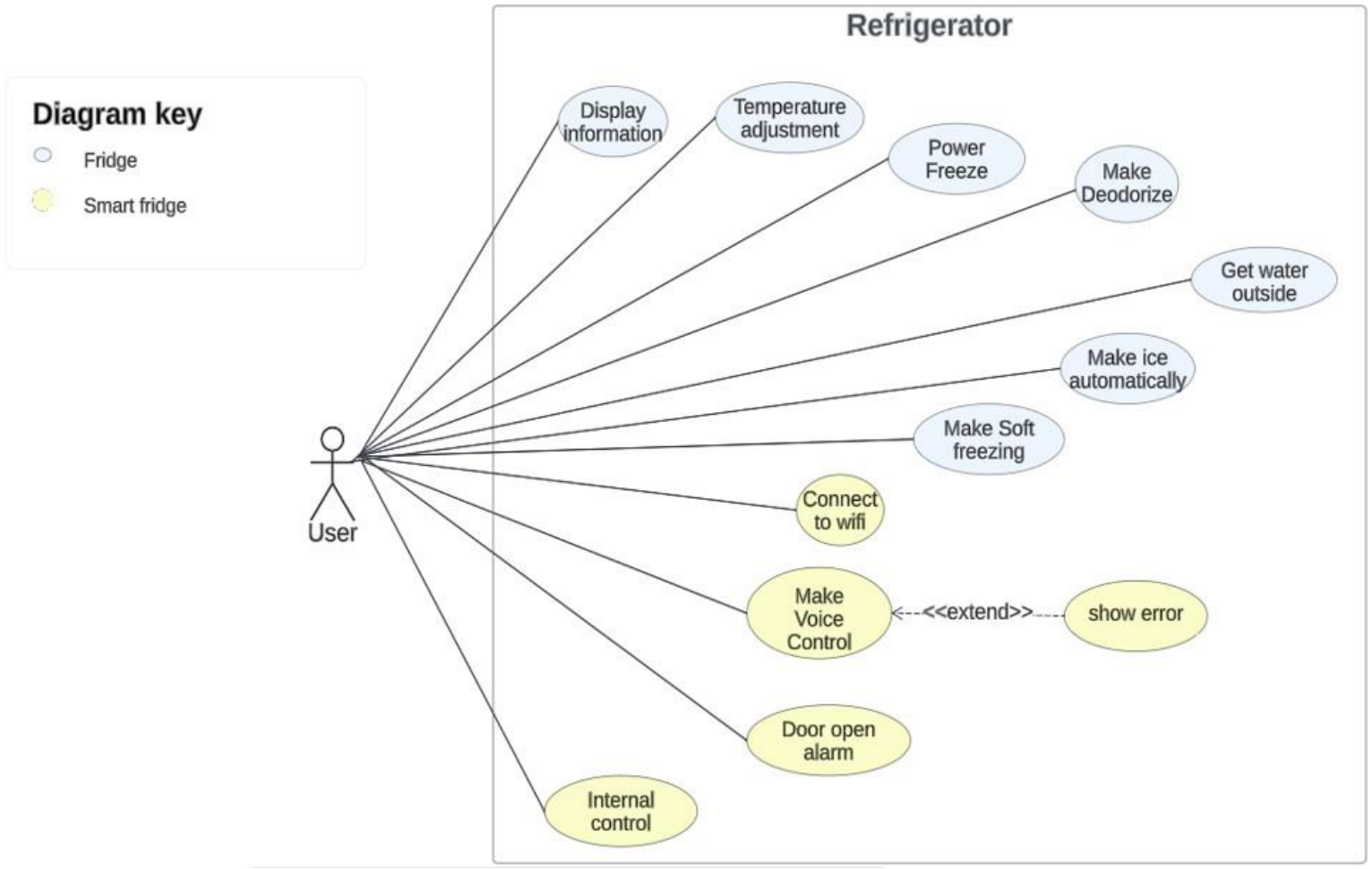
The refrigerator has a simple structure consisting of 3 main parts:

- Compressor: helps maintain low temperatures inside the refrigerator.

- Condenser: removes absorbed heat from inside the refrigerator, allowing refrigerant to return to the system.

- Evaporator: The evaporator facilitates cooling of the refrigerator by absorbing heat from the inside. Users need to be able to use the system in a variety of ways via web-based or mobile interfaces.
- Display information: Cabinet type, Usable capacity, Freezer capacity, Refrigerator capacity, Refrigerator door material, Refrigerator tray material, Material of gas pipe, indoor unit, Year of launch, Manufactured in, length, width, brand.
- Temperature adjustment: 7 different adjustment levels (number displayed from 1 - 7). They have the effect of adjusting the temperature in both compartments (refrigerator and freezer) of the refrigerator. If you want to hold the ice tighter, adjust the knob or dial toward "Min" or reduce the temperature. For more warmth, adjust toward "Max" or increase the temperature.
- Power Freeze: Power Freeze blows powerful cold air directly into the freezer to help freeze food or create ice cubes quickly.
- Deodorize: Activated carbon filter has the ability to remove many types of bacteria, mold, fishy odors, and dust.
- External water dispenser: The refrigerator is designed with an internal water tray and an external water dispenser. Just put water in the tray and you will be provided with cool water as soon as you need it with just the push of a button without needing to open the refrigerator a lot causing cold air loss.
- Automatic ice maker: In the container, there will be a power pump. When you add water, the water will be pumped to the automatic ice maker, and after making ice will fall into the tray.
- Soft freezing: only lightly freezes the surface but still maintains the softness inside. In addition, the system also describes a smart refrigerator that has the characteristics of a regular refrigerator in terms of additional features.
- Wifi connection for remote control: control the refrigerator temperature or activate quick cooling in case of problems.
- Door open alarm: opening the refrigerator door for too long or accidentally not closing the refrigerator, the refrigerator will automatically emit a warning bell to help you promptly close the door.
- Voice control: easy voice control helps adjust the temperature and monitor it during operation.
- Internal control: The camera can take pictures inside the cabinet, helping you check the amount of food left without opening the door.

b. Usecase Diagram



Name of use case	Refrigerator

Created by:	Cường	Last Update By:	Group8
Date Created:	25/11/2023		
Description:	This use case illustrates the process of operating a refrigerator with both basic and advanced functions. Users can adjust the temperature and use various features.		
Actor:	User		
Precondition:	The refrigerator is connected to a power source.		
Postcondition:	The user has successfully adjusted the refrigerator temperature, utilized basic and advanced functions, and ensured the proper functioning of the refrigerator.		

Flow	- Display information: Users can see information about their refrigerator. - Temperature adjustment: Users select the desired temperature level for both the refrigerator and freezer compartments using a scale from 1 to 7. - Power Freeze: This function is used to quickly freeze food and water. - Deodorizing: Eliminates bad odors of food and does not let food odors mix with each other. -External water dispenser: Users can get cold water and ice outside the refrigerator door instead of having to open the refrigerator door and take it out. - Automatic ice making: The user adds water to the ice tray, then ice is produced and falls into the storage tray. - Soft freezing: Store food that still retains its soft state

Alternative Flow:	Every time the user selects a function, The system will notify you to confirm the successful activation of that function.
Exception:	No exceptions
Requirements:	The refrigerator must maintain consistent performance and deliver the specified functions reliably. The refrigerator must have choice before use others function.

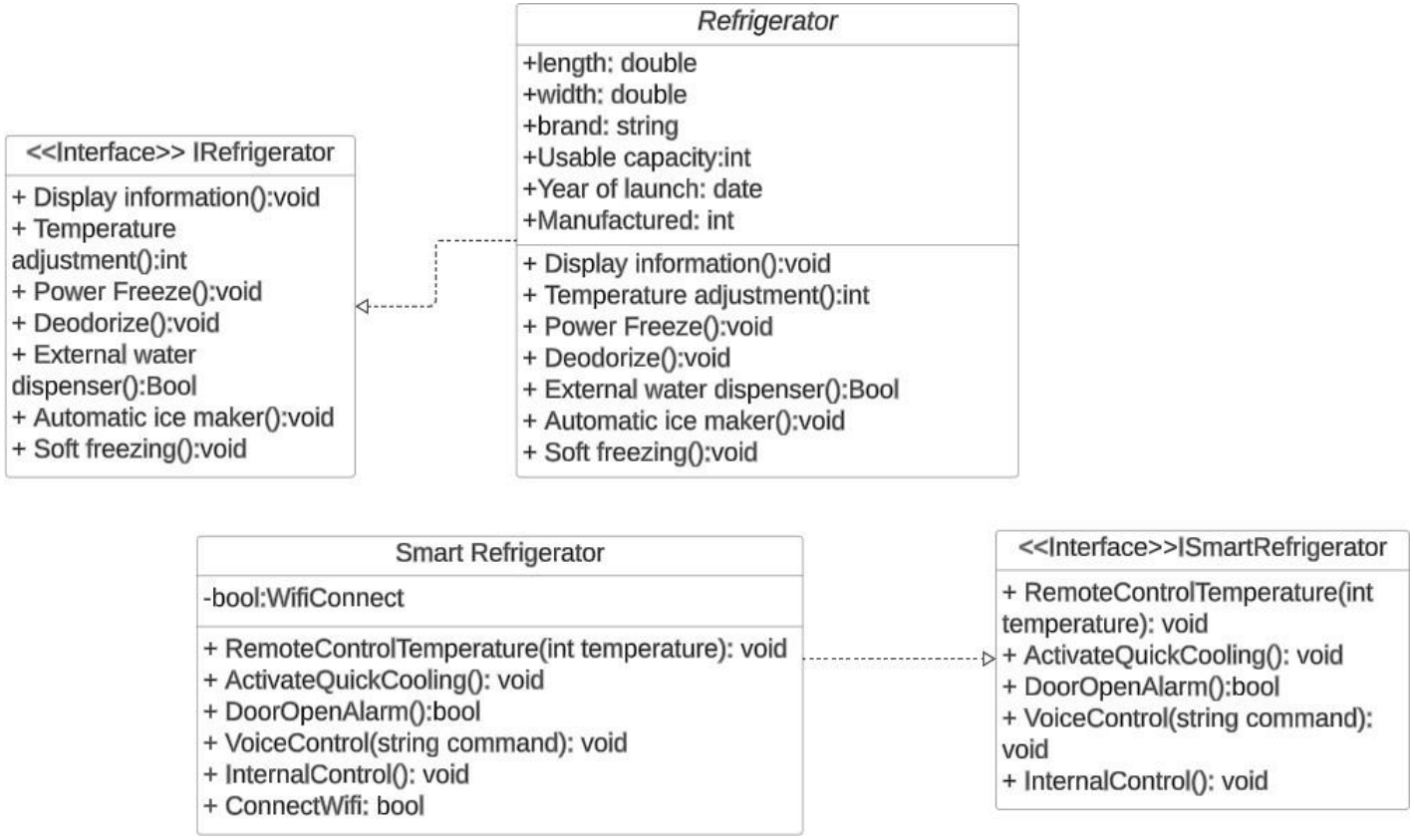
Name of use case	Smart Refrigerator		
Created by:	Group8	Last Update By:	Group8
Date Created:	27/11/2023		
Description:	This use case illustrates the process of operating a refrigerator with both basic and advanced functions. Users can adjust the temperature and use various features.		
Actor:	User		
Precondition:	The refrigerator is connected to a power source. The user has a stable internet connection.		

Postcondition :	The user has successfully adjusted the refrigerator temperature, utilized basic and advanced functions, and ensured the proper functioning of the refrigerator.

Flow	Wifi connection: Users connect to wifi to control temperature and select functions remotely Internal control: Users can view inside the refrigerator using the camera inside the refrigerator

Alternative Flow:	Voice control: Users control temperature and functions via voice. Every time the user selects a function, The system will notify you to confirm the successful activation of that function.
Exception:	Door open warning: Sounds a warning if the user opens the door for too long.
Requirements:	The refrigerator must have choice before use others function. Temperature adjustments must be available for both the refrigerator and freezer compartments. Confirmation messages must be provided for successful activation of functions. Voice control must be implemented for hands-free operation. The system must have a door open alarm to alert the user if the refrigerator door is left open. The refrigerator must maintain consistent performance and deliver the specified functions reliably.

c. Class Diagram



Explain:

Class User: represents the user's name and shows their own list of refrigerators and it has a one-to-many relationship with the refrigerator class because one person can own many refrigerators

Class Refrigerators: The refrigerator class shows basic information such as length, width, brand, year of lunch, manufacture and Usable capacity including functions such as displayinginformation(), Temperatureadjustment(), Powerfreeze(), Deodorize(), External water dispenser(), Automatic ice maker()
Class Smartrefrigerators: is a subclass created from the refrigerator class and it inherits all the properties of the refrigerator class it has functions such as: ActivateQuickCooling(), DoorOpenAlarm(), VoiceControl(), InternalControl() and it can ConnectWifi
Class <<interface>> IRefrigerators: Defines a set of methods that any class representing a refrigerator should implement. This includes displaying information, adjusting temperature, power freezing, deodorizing, using an external water dispenser, and activating an automatic ice maker.
Class <<interface>>ISmartRefrigerators: Extends IRefrigerators and adds methods for features specific to smart refrigerators, such as quick cooling activation, door open alarm, voice control, internal control, and Wi-Fi connectivity.

4. Design Patterns

Introduce design pattern, general concepts of Design Pattern

A. General concept of Design Pattern

Design patterns are typical solutions to commonly occurring problems in software design. They are like pre-made blueprints that you can customize to solve a recurring design problem in your code.
You can't just find a pattern and copy it into your program, the way you can with off-the-shelf functions or libraries. The pattern is not a specific piece of code, but a general concept for solving a particular problem. You can follow the pattern details and implement a solution that suits the realities of your own program.
Patterns are often confused with algorithms, because both concepts describe typical solutions to some known problems. While an algorithm always defines a clear set of actions that can achieve some goal, a pattern is a more high-level description of a solution. The code of the same pattern applied to two different programs may be different.
An analogy to an algorithm is a cooking recipe: both have clear steps to achieve a goal. On the other

hand, a pattern is more like a blueprint: you can see what the result and its features are, but the exact order of implementation is up to you.

B. Introduce about design pattern

Most patterns are described very formally so people can reproduce them in many contexts. Here are the sections that are usually present in a pattern description:

- Intent of the pattern briefly describes both the problem and the solution.
- Motivation further explains the problem and the solution the pattern makes possible.
- Structure of classes shows each part of the pattern and how they are related.
- Code example in one of the popular programming languages makes it easier to grasp the idea behind the pattern.

Some pattern catalogs list other useful details, such as applicability of the pattern, implementation steps and relations with other patterns

Creational Design Patterns:

- Factory Method
- Singleton
- Abstract Factory
- Builder
- Prototype
- Singleton

Structural Design Patterns:

- Adapter
- Bridge
- Composite
- Decorator
- Facade
- Flyweight
- Proxy

Behavioral Design Patterns:

- Chain of Responsibility
- Command
- Iterator
- Mediator
- Mementor
- Observer
- State
- Strategy

- Template Method
- Visitor

a. Creational pattern

Introduce creational

pattern

A, Factory method

Factory Method is to creating objects as Template Method is to implementing an algorithm. A superclass specifies all standard and generic behaviour (using pure virtual "placeholders" for creation steps), and then delegates the creation details to subclasses that are supplied by the client.
Factory Method makes a design more customizable and only a little more complicated. Other design patterns require new classes, whereas Factory Method only requires a new operation.
People often use Factory Method as the standard way to create objects; but it isn't necessary if: the class that's instantiated never changes, or instantiation takes place in an operation that subclasses can easily override (such as an initialization operation).
Factory Method is similar to Abstract Factory but without the emphasis on families.
Factory Methods are routinely specified by an architectural framework, and then implemented by the user of the framework.

Problem:

I'm facing a situation where your logistics management application is becoming less and less flexible due to its tight coupling with the Truck class. To solve this problem and make your code more modular and scalable, you can use Strategy Patterns. Your logistics management application initially focuses on truck transportation, leading to a close association with the Truck class. Adding support for ocean freight is challenging and can result in a codebase full of conditionals.

Solution:

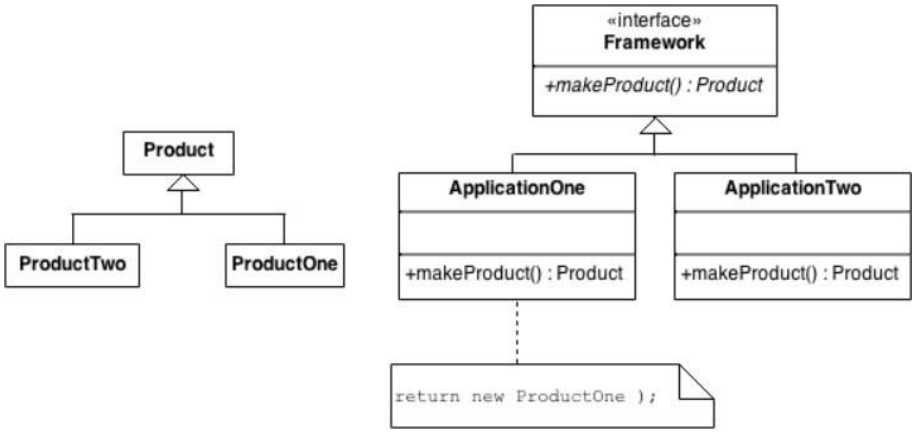
Define transport interface:
Create an interface, let's call it TransportationStrategy, that declares a freight

transportation method. How to use in your application:
Now you can easily switch between shipping methods without significantly modifying existing code.

Benefits:
The strategy pattern allows you to encapsulate the shipping algorithm separately from the client code. You can easily add new shipping strategies without modifying existing code. Code becomes more modular and follows the Open/Closed

Principle, making extension easier. By adopting the Strategy Pattern, you can make your logistics management application more flexible, easier to maintain, and ready to incorporate new transportation modes in the future.

Structure:



B,Abstract Factory

Abstract Method is a concept in object-oriented programming, especially in Java and other programming languages that support abstraction. It refers to a method that is only declared in an abstract class or an interface, without any concrete implementation. Subclasses of the abstract class or classes implementing that interface will have to provide implementation of this abstract method.

Problem:

In the scenario of creating a furniture shop simulator with various furniture products and their style variants, you can address the need for creating matching furniture objects without modifying existing code by applying the Abstract Factory Pattern.

You have a family of related products (Chair, Sofa, CoffeeTable) available in different style variants (Modern, Victorian, ArtDeco). Ensuring that individual furniture objects match others of the same family and style is crucial. Additionally, you want to accommodate new products or families without altering existing code.

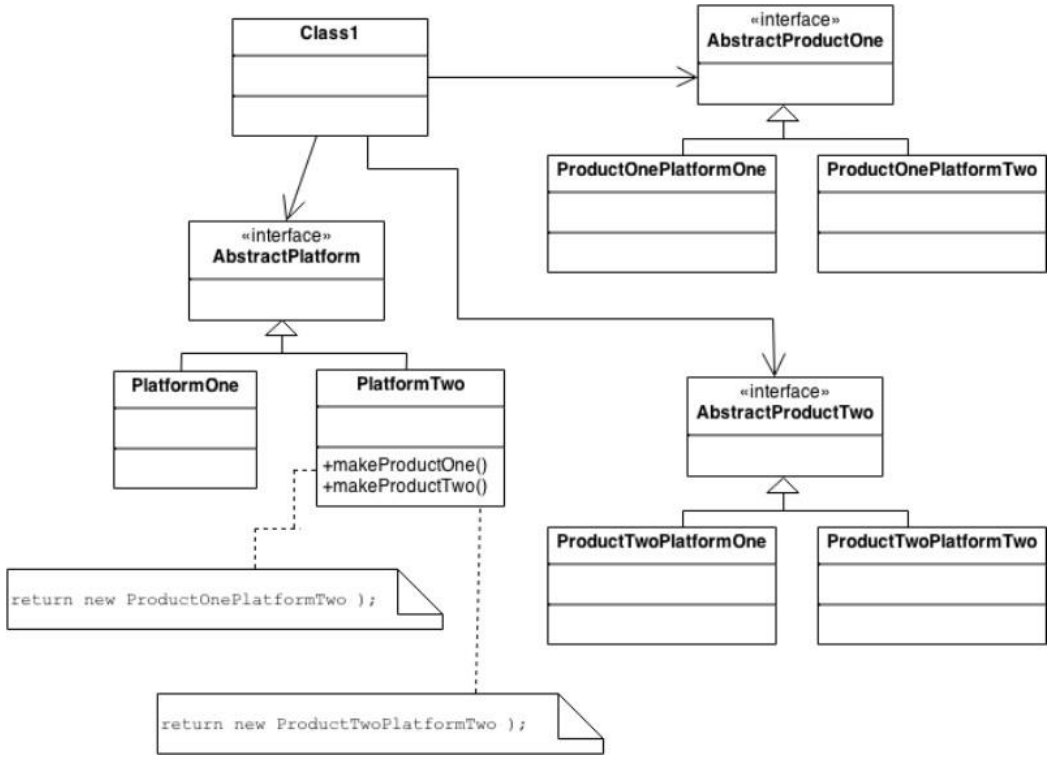
Solution:

Define Abstract Product Interfaces:
Create interfaces for each product in the family (Chair, Sofa,
CoffeeTable). Create Concrete Product Implementations:
Implement concrete classes for each product in different style variants (Modern,
Victorian, ArtDeco). Define Abstract Factory Interface:
Create an abstract factory interface that declares methods for creating
each product. Create Concrete Factory Implementations:
Implement concrete factories for each
style variant. Usage in Your Application:

Benefit:

Use the abstract factory to create families of related products without worrying about their individual implementations. The Abstract Factory Pattern ensures that products created by a factory are compatible with each other.
Adding new products or families is facilitated by creating new concrete factories without changing existing code. The code is modular, adhering to the Open/Closed Principle, making it easy to extend for new styles or products.

Structure:



A. Builder

Builder pattern builds a complex object using simple objects and using a step by step approach. This type of design pattern comes under creational pattern as this pattern provides one of the best ways to create an object.

A Builder class builds the final object step by step. This builder is independent of other objects.

Problem:

In the process of developing a human resource management application, the challenge lies in creating a flexible system for constructing Employee objects. Employees may have a variety of properties, and different instances may require different sets of attributes. Additionally, optional attributes further complicate the construction process. A robust solution is needed to handle these variations without

cluttering the codebase and making the construction process more manageable.

Solution:

EmployeeBuilder Interface:

Define an interface, EmployeeBuilder, that outlines methods responsible for constructing various properties of an Employee (e.g., setName, setAge, setGender, etc.). Additionally, include methods to accommodate optional attributes, ensuring the construction process is adaptable to different scenarios.

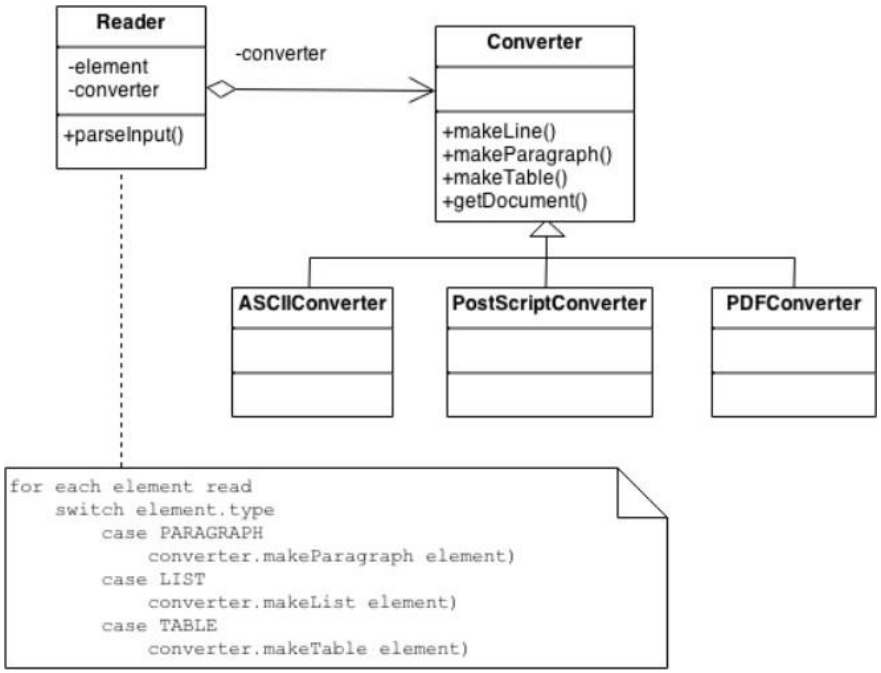
Concrete EmployeeBuilder Class:

Implement the EmployeeBuilder interface with a concrete class, such as DefaultEmployeeBuilder. Provide specific implementations for each method within the interface to set the corresponding properties of the Employee. Allow flexibility for setting optional attributes based on the specific needs of the Employee instance being constructed.

HRManager Class:

Create an HRManager class tasked with managing the construction of Employee objects. The HRManager should accept an instance of the EmployeeBuilder interface. Delegate the construction process to the provided EmployeeBuilder, allowing the HRManager to remain oblivious to the specifics of how an Employee is constructed.

Structure:


Reader
-element
-converter
+parseInput()
-converter
Converter
+makeLine()
+makeParagraph()
+makeTable()
+getDocument()
ASCIIConverter
PostScriptConverter
PDFConverter
for each element read
switch element.type
case PARAGRAPH
converter.makeParagraph element)
case LIST
converter.makeList element)
case TABLE
converter.makeTable element)

B. Prototype

The prototype design pattern is used to create a duplicate or clone of the current object to improve performance. When the creation of an object is expensive or complex, this pattern is used. For example, an object is to be created following a time- consuming database operation. We can cache the object, return its clone on the next request, and update the database as needed, resulting in fewer database calls (Chauhan, 2022).

Problem:

While developing a system for handling geometric objects like squares and circles, there's a need to efficiently create multiple instances of similar objects without compromising performance and resource utilization. A dynamic approach that involves creating copies of existing objects to generate new ones is considered for an optimal solution.

Solution

To address this requirement, a prototype-based design is suggested. Here are the steps to implement this solution: Shape Interface:

Introduce a Shape interface that includes a clone method, enabling geometric objects to replicate themselves.

This common interface ensures that all geometric objects, regardless of type (e.g., Square, Circle), can support the cloning mechanism.

Concrete Classes (Square and Circle):

Implement concrete classes that adhere to the Shape interface, such as Square and Circle.

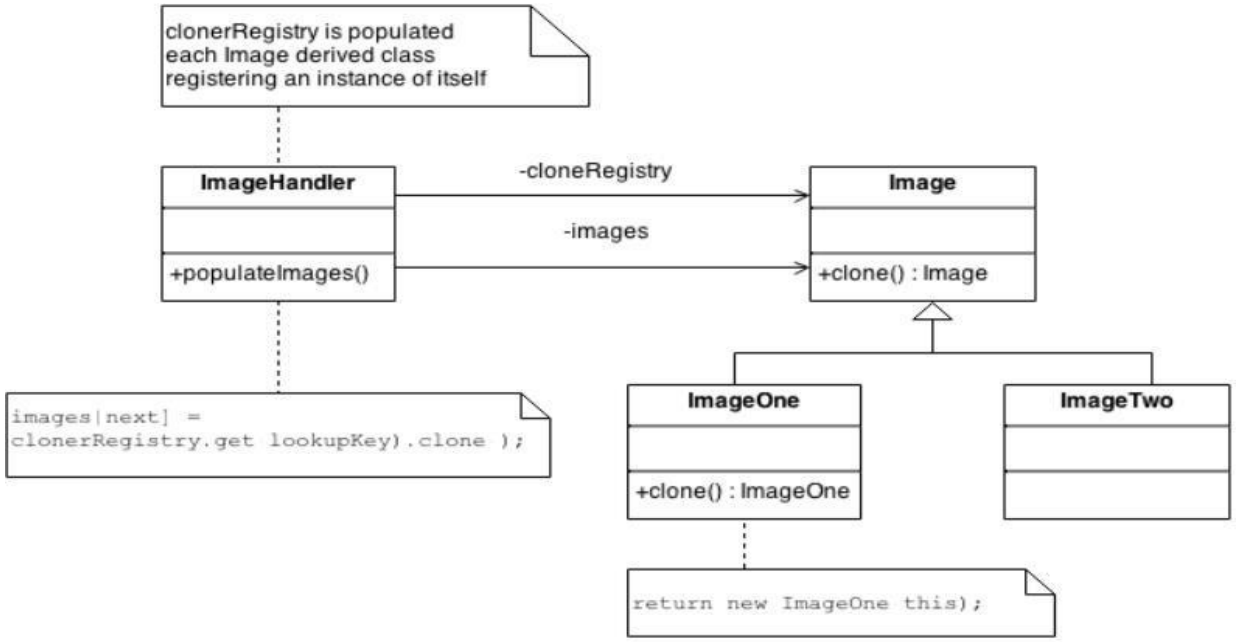
Each concrete class should provide its own implementation of the clone method, allowing instances of the class to be efficiently duplicated.

Prototype Usage:

Utilize the prototype pattern to quickly replicate and create objects.

Instead of creating new objects through conventional constructors, clone an existing object to produce similar ones. This approach enhances performance by avoiding redundant setup processes

Structure:



C. Singleton

The Singleton Pattern is one of the simplest design patterns. Sometimes we need to have only one instance of our class for example a single DB connection shared by multiple objects as creating a separate DB connection for every object may be costly. Similarly, there can be a single configuration manager or error manager in an application that handles all problems instead of creating multiple managers. The singleton pattern is a design pattern that restricts the instantiation of a class to one object. Let's see various design options for implementing such a class. If you have a good handle on static class variables and access modifiers this should not be a difficult task.

Problem

In certain scenarios during application development, it becomes crucial to ensure that a particular class has only a single instance, and that instance is easily accessible by every component in the system. This is especially relevant for classes managing configuration, logging, or connections to a database where having only one instance is desirable for consistency and resource management.

Solution:

To address this, the Singleton pattern is suggested as a solution. The Singleton pattern ensures that a class has only one instance and provides a global point of access to that instance throughout the application.

Singleton Class:

Create a class designed to be a Singleton. This class typically contains private constructors to prevent direct instantiation from outside the class.

Implement a private static instance variable within the class to hold the single instance of the class. Static Method for Instance Retrieval:

Design a static method, often named getInstance(), within the Singleton class to provide a way to access the single instance. Ensure that this method is responsible for creating the instance if it doesn't exist yet and returning the existing instance otherwise.

Ensure Single Instance:

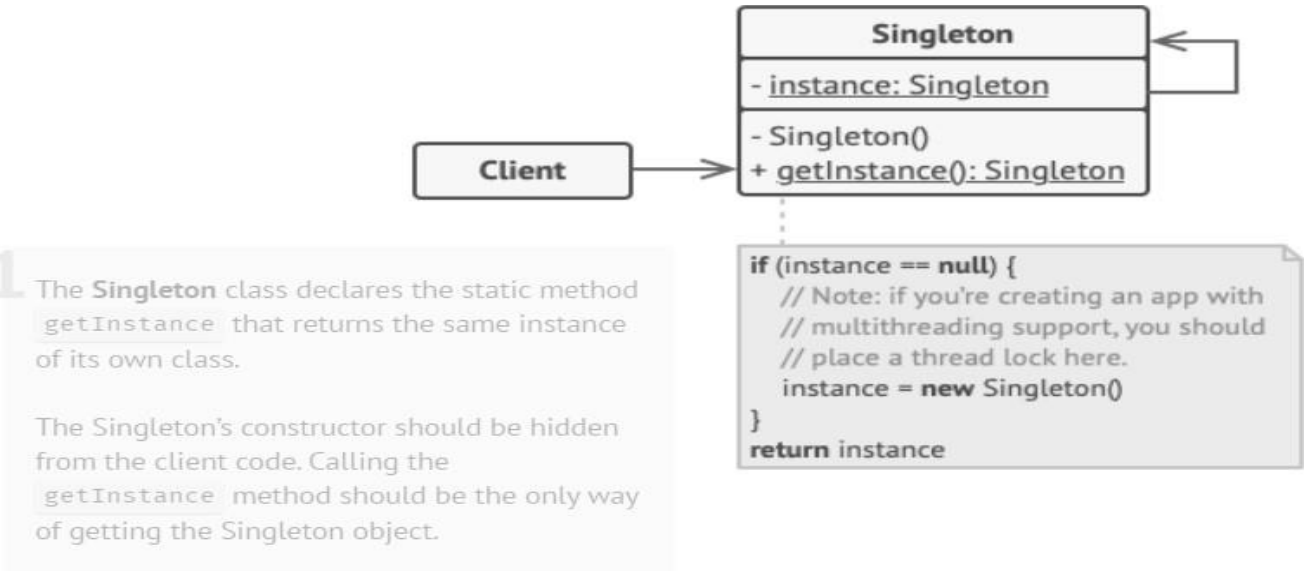
Guard against multiple instantiation by restricting access to the constructor, either by making it private or protected.

Use a mechanism, such as lazy initialization or double-checked locking, to ensure that only one instance is created when needed.

Access and Utilize the Singleton:

Use the getInstance() method wherever access to the Singleton instance is required in the application. Call the necessary methods or properties on the Singleton instance as needed

Structure:



Description of a creational scenario:

In one city, there is a famous decorative lamp company called "XZY", which specializes in producing high-quality lamps. The company produces three main types of lights: Chandeliers, LED String Lights and Stage Lights. Each type of lamp has its own characteristics and functions, serving the diverse lighting needs of customers. The light includes parts such as:

- Chip: Chips can affect factors such as brightness, color temperature, and energy efficiency in LED lighting fixtures.
- Power: Power is a measure of the electrical energy consumed by a device. It shows how much energy a light bulb or fixture uses.
- Light bulb type: Common types include incandescent bulbs, fluorescent lights, halogen bulbs,

and LED lights. Each type has different characteristics, including energy efficiency, longevity and color temperature.

- Number of bulbs: Number of bulbs in 1 lamp.

Chandeliers:

Bulb type: Halogen Bulb.

Frame structure: Made of metal, stainless steel, copper or aluminum.

Lamp arm: Attached to the frame to contain the light bulb to form a beam

shape. Reflector: Integrated into the frame structure to connect to the

bulb.

- Adjust color: turn the knob with 3 steps: Step 1 light turns white, Step 2 light turns light yellow, Step 3 light turns yellow.
- Removing bulbs: There are 6 bulbs that can light up, removing from 1 to 5 bulbs can still light up.

Led string lights:

Bulb type: LED

Material: flexible silicone, water

resistant. High elasticity: able to bend.

- Light mode: Mode 1 LED light is basic white, Mode 2 LED light is red, Mode 3 LED light is orange yellow, Mode 4 lights change orange yellow, red white, white continuous.
- Timer mode: This feature allows you to schedule the lights to automatically turn on and off at specific times.

Stage light:

Motor: The Moving Head Beam light has a Pan (horizontal rotation) and Tilt (vertical rotation) motor. This line of lights is very smart and the light can rotate 540 degrees horizontally and 270 degrees

vertically.

Ball type: osram 230 ball.

Function

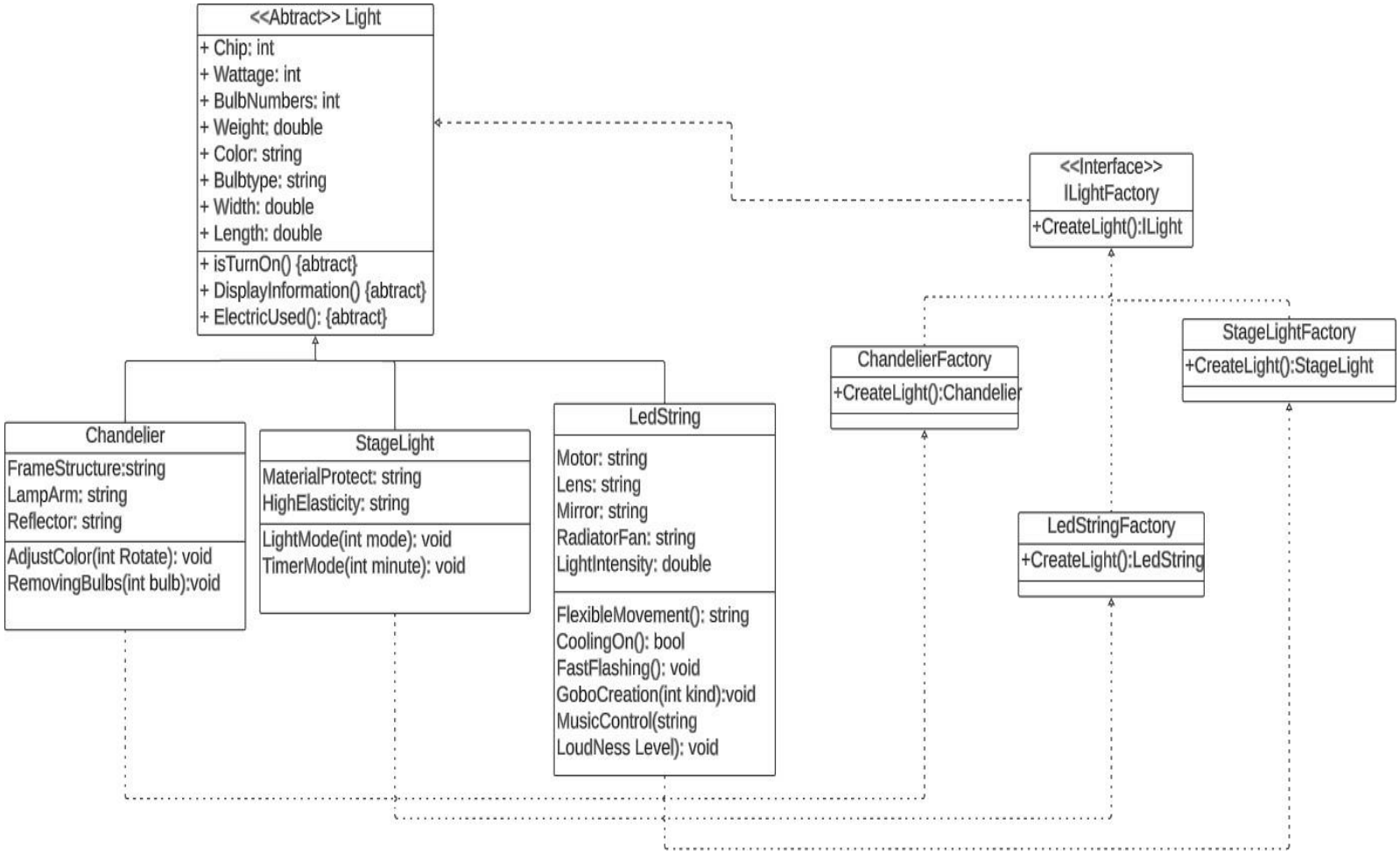
Flexible Movement: Ability to rotate and tilt the light in many different directions, creating dynamic lighting effects. Radiator fan: The lamp is usually very hot, a radiator fan will be used to cool it down Fast flashing: The light becomes stronger for 1 second then returns to normal brightness. Gobo creation: Light creates all kinds of shapes:

- Star Shape: Gobos are star shaped to create a shimmering star effect.
- Heart Shape: Heart-shaped gobos are often used in romantic or love-related events.
- Flower Image: Gobo with images of flowers to create a fresh and comfortable atmosphere. Music Control: Some lights have a feature that responds to music.

- Low Music: When the music is low or there is no sound, the light can keep at a low brightness level
- Rapid sound: Effects such as rotation, tilt, rapid and sudden flashes can appear according to the beat of the music.
- Loud Music: create a light gobo with an image and stronger light.

The company established a light factory for the purpose of producing lights. Each factory is responsible for creating its specific type of lamp. Chandelier factory specializes in manufacturing chandeliers. LED string light factory specializes in manufacturing LED string lights. Stage lights factory specializes in producing stage lights.

Diagram + explanation



Explain:
Light (Class):
Attributes:
Chip, Wattage, BulbNumbers, Weight, Color, Bulbtype, Width, Length: Basic characteristics of a light source. Methods:
isTurnOn(): bool (abstract): Checks if the light is turned on.
DisplayInformation(): void (abstract): Displays information about the light.

ElectricUsed(): double (abstract): Calculates the amount of electricity used. Chandelier (Class):
Inherits from ILight (Interface): Inherits the common attributes and methods of the ILight interface. Attributes:
FrameStructure, LampArm, Reflector: Specific attributes related to the structure of a chandelier. Methods:
AdjustColor(int Rotate): void: Adjusts the color of the chandelier.
RemovingBulbs(int bulb): void: Removes bulbs from the chandelier. StageLight (Class):
Inherits from ILight (Interface): Inherits the common attributes and methods of the ILight interface. Attributes:
MaterialProtect, HighElasticity, Motor, Lens, Mirror, RadiatorFan, LightIntensity: Attributes related to the structure and functionality of a stage light.
Methods:
LightMode(int mode): void: Sets the light mode for stage performance. TimerMode(int minute): void: Sets a timer mode for the stage light.
FlexibleMovement(): string: Allows flexible movement of the stage light. CoolingOn(): bool: Turns on the cooling system.
FastFlashing(): void: Triggers a fast flashing mode. GoboCreation(int kind): void: Creates a Gobo effect.
MusicControl(string LoudNess Level): void: Controls the light based on music loudness. LEDString (Class):
Inherits from ILight (Interface): Inherits the common attributes and methods of the ILight interface. No additional attributes or methods mentioned in the provided text.
Purpose: Represents a type of light source with characteristics similar to generic lights but with no additional specialized features mentioned.
ILight (Interface):
Methods:
isTurnOn(): bool
DisplayInformation():

void ElectricUsed():
double
Purpose: Defines a common set of attributes and methods that any light class (Chandelier, StageLight, LEDString) should implement, ensuring a consistent interface.
ILightFactory
(Interface): Methods:

CreateLight(): ILight
Purpose: Declares a method for creating instances of lights. This interface allows different factories to produce various types of lights.
ChandelierFactory, StageLightFactory, LedStringFactory (Factories):
Methods:
CreateLight(): Chandelier / StageLight / LedString
Purpose: Implement the ILightFactory interface to create specific instances of lights (Chandeliers, StageLights, LEDStrings). Each factory specializes in creating a particular type of light.
Relationships:
Chandeliers, StageLights, LEDStringLights all inherit from the ILight interface. ChandelierFactory, StageLightFactory, LedStringFactory all implement the ILightFactory interface.

b. Structural pattern

Introduce creational

pattern

A. Adapter

Definition: The structural design pattern known as an adaptor is what makes it possible for things that have interfaces that are incompatible to work together.

Problem:

You are now working on building an application for monitoring the stock market that is capable of downloading stock data in XML format from a variety of sources. Additionally, you want to incorporate a third-party analytics library in order to improve the app; however, the library only allows data in the JSON format. If you do not have access to the library's source code, modifying it so that it is compatible with XML might potentially cause disruption to the code that is already in place.

Solution:

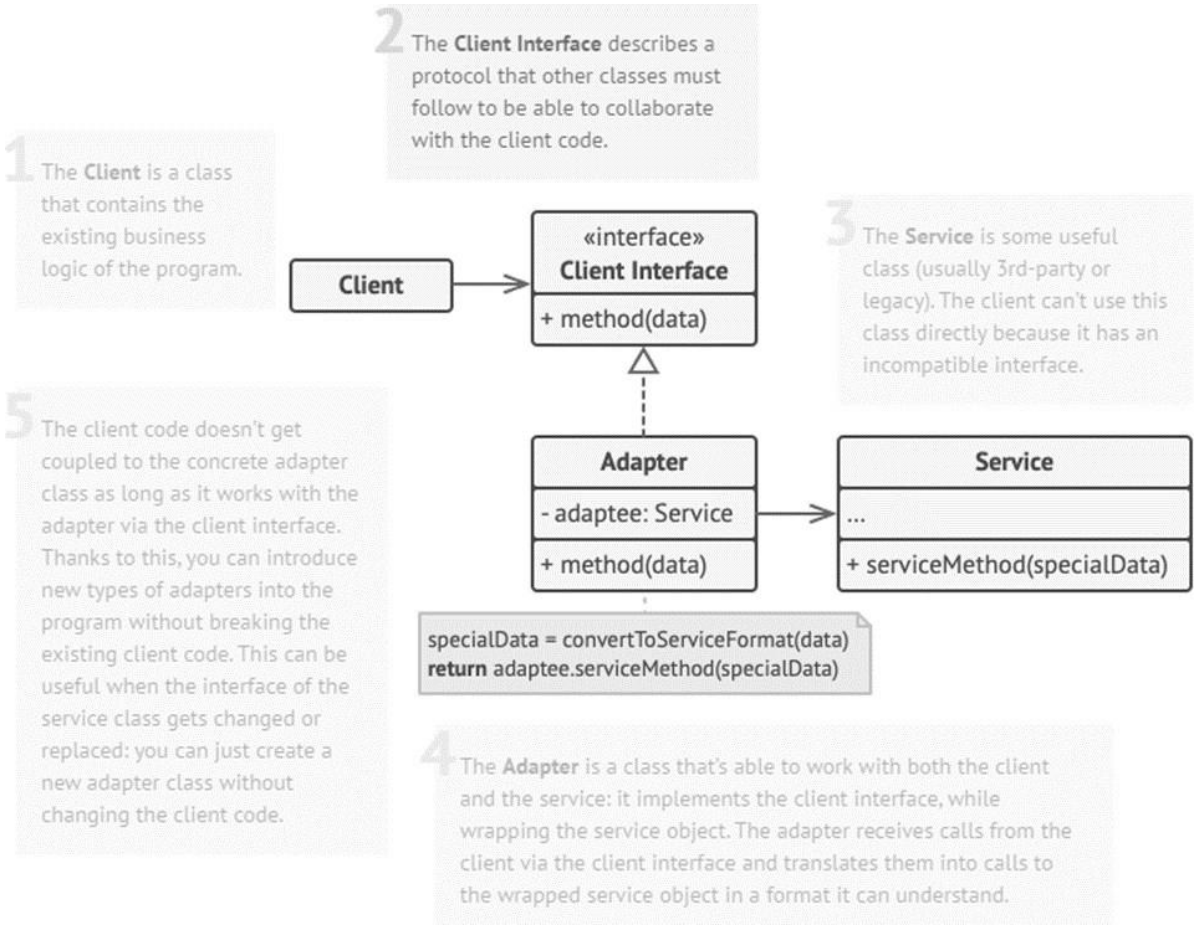
Develop an adapter, which is a specialized entity that transforms the interface of one object to make it

compatible with another. The adaptor hides the intricacies of the conversion process from the wrapped object, which stays oblivious to its existence. For example, an adapter has the capability to transform data expressed in metric units such as meters and kilometers into imperial units such as feet and miles.

Adapters serve the purpose of converting data formats and enabling cooperation between objects that have disparate interfaces. The technique entails acquiring an interface that is compatible with an already existing object, so enabling the object to safely invoke the methods of the adapter. Upon receiving a call, the adapter transmits the request to the second object in the specified format and sequence.

To address the problem of incompatible formats in our stock market app, we may handle it by developing XML-to-JSON adapters for each analytics library class that is directly used in your code. Revise your code to only interact with the library via these adapters. Upon receiving a request, the adapter converts the XML data into a JSON format and then routes the call to the relevant methods of the wrapped analytics object.

Structure:



B. Bridge

Definition:

The Bridge pattern is a design pattern that allows for the division of a big class or a group of closely related classes into two distinct hierarchies: abstraction and implementation. These hierarchies may be created separately from each other.

Problem:

Assume that you have created a task management application that includes fundamental classes such as Tasks and Projects. You want to expand the system's capabilities to monitor the progress of work based on various priorities, such as High Priority and Low Priority. As a result, subclasses like TaskHighPriority and ProjectLowPriority were developed to merge task type and priority.

The inclusion of additional task types and priorities in the hierarchy results in the creation of several related subclasses, such as LowPriorityProject and HighPriorityTask. The complexity and difficulty of organising and maintaining the source code increase with each new combination, making the process more intricate.

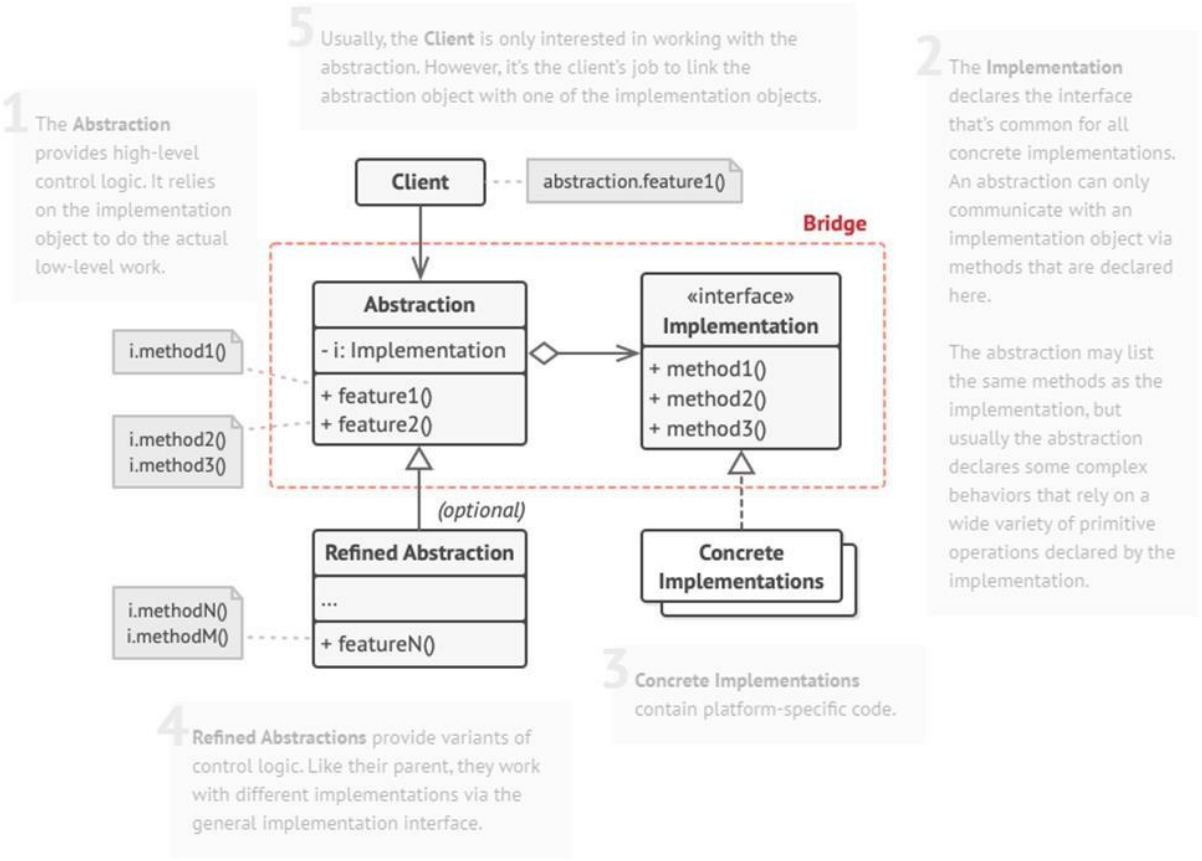
Solution:

In order to address the issue of expanding the task management system with varying priorities, we may use the Bridge design pattern.

The Bridge design addresses this issue by transitioning from inheritance to object composition. This entails isolating some elements and organizing them into a distinct class structure. To address this situation, we may create subclasses such as HighPriority and LowPriority to represent different levels of priority. Consequently, Project and Task will reference an instance of this updated hierarchy, rather than retaining all the data inside. The state and behavior of an object inside a class.

By using this method, we may effectively handle the order of tasks without adding intricacy to the system. The incorporation of additional job categories and priorities does not need any modifications to the existing hierarchy, hence minimizing intricacy and enhancing adaptability in code administration.

Structure:



C. Composite:

Definition:

The Composite pattern is a design pattern that allows for the composition of items into tree structures, enabling the manipulation of these structures as if they were singular things.

Problem:

The Composite pattern is applicable when the fundamental model of your application takes the shape of a hierarchical structure. Let's use two sorts of objects as an example: Products and Boxes. A box has the

ability to hold products or smaller

boxes, resulting in a hierarchical structure. Constructing an ordering system using these classes presents the difficulty of computing the overall cost of an order, since it necessitates the extraction of nested Boxes and Products. Implementing a straightforward method in a program becomes complex because it requires considering class types, nesting levels, and other subtle aspects, which might render it impractical or even unfeasible.

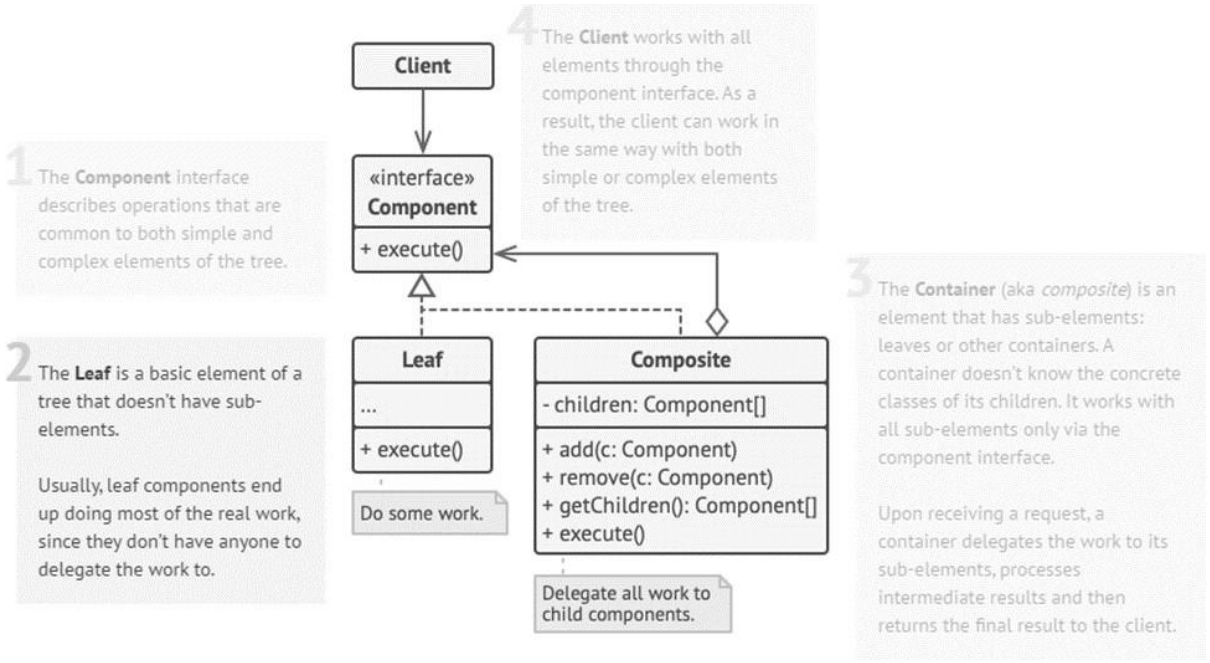
Solution:

The Composite design suggests that Products and Boxes should be accessed via a common interface, which includes a way for calculating the overall pricing.

This technique would directly retrieve the pricing of a Product. To process a Box, one must iterate over each item included inside the box, get its price, and then calculate the total value of the box. If any of these items are smaller boxes, the recursive procedure will continue until the costs of all inner components are calculated. Moreover, a box might incur additional fees, such as packing costs, which would be included in the final price.

An important benefit in this case is the ability to separate from certain classes inside the tree. There is no need to differentiate between a simple product and a complex packaging. Uniform treatment of all objects is possible via the use of a common interface. When a method is called, the objects propagate the request downwards via the tree structure.

Structure:



D. Decora

te

Definition:

The decorator pattern is a structural design pattern that allows for the addition of additional functionalities to objects by encapsulating them inside wrapper objects that possess these functionalities.

Problem:

You are creating a notification library that will enable program to inform consumers about significant occurrences. The first version depended on the Notifier class, which had fundamental attributes, a constructor, and a send function. Users set it up once to transmit emails. Nevertheless, the requests broadened to include SMS, Facebook, and Slack alerts. You augmented the Notifier class by creating subclasses for each sort of notification, with the intention that clients would choose and use one of them.

Nevertheless, consumers proposed the amalgamation of several notification categories, presenting a difficulty. Attempting togenerate subclasses for every possible combination was found to be unfeasible, resulting in excessive code in both the library

and the client. An innovative strategy is required to effectively arrange notification categories and avoid an excessive number of subclasses.

Solution:

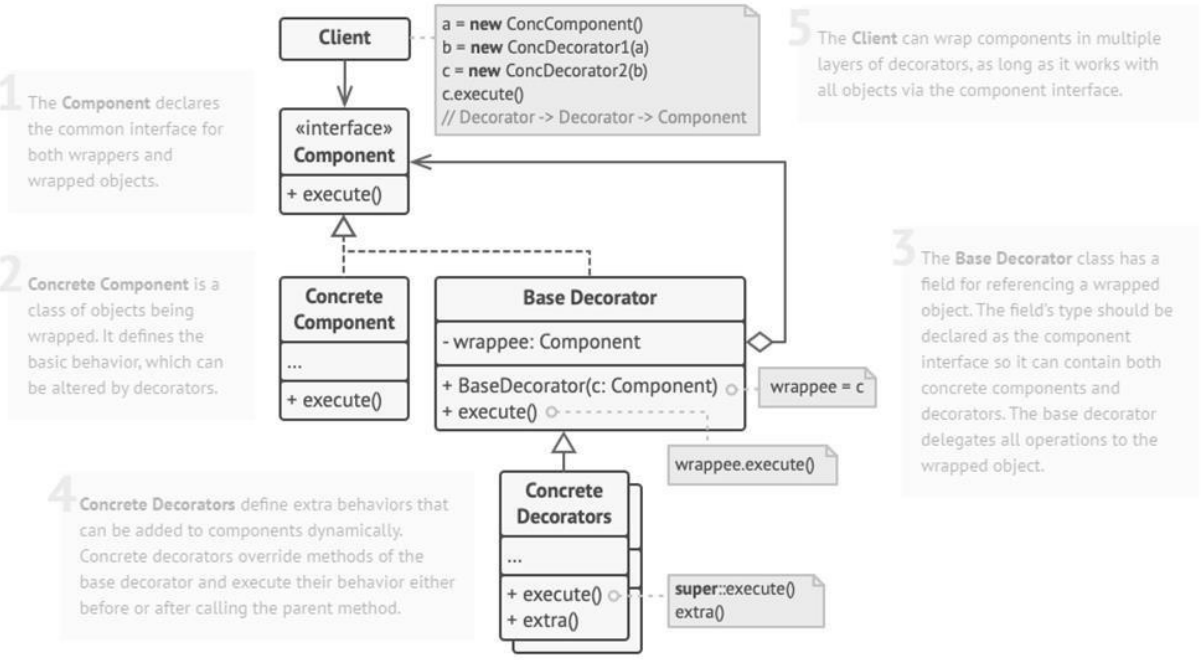
Extending a class is a common solution for modifying object behavior, but it comes with limitations. Inheritance is static, preventing runtime alterations to an object's behavior. Moreover, subclasses can have only one parent class, limiting inheritance to a single source.

Aggregation or Composition offers a flexible alternative, allowing an object to delegate work to another through references. This approach enables dynamic behavior changes at runtime by substituting linked "helper" objects. Objects can utilize behaviors from various classes, referencing multiple objects and delegating diverse tasks. This principle underlies design patterns like the Decorator.

In the Decorator pattern, a "wrapper" is an object linked to a target object, sharing the same methods and delegating requests. The wrapper can modify the result before or after passing the request to the target. To turn a simple wrapper into a decorator, ensure it implements the same interface as the wrapped object. This allows stacking multiple decorators, combining their behaviors.

In the notifications example, the base Notifier class retains simple email notification behavior, while other notification methods become decorators. Clients wrap a basic notifier with decorators based on their preferences, forming a stacked structure. The client interacts with the last decorator in the stack, unaware of whether it's the original notifier or a decorated version. This approach is applicable to various behaviors, enabling clients to decorate objects with custom decorators following the same interface.

Structure:



E. Faca

de

Definition

:

The facade pattern is a structural design pattern that offers a streamlined interface to a library, framework, or any other complex collection of classes.

Problem:

Imagine a scenario where you need to verify that your code is compatible with a wide variety of objects

from a sophisticated library or framework. Under normal conditions, it is necessary to initialize these objects, handle dependencies, guarantee appropriate sequencing of methods, and do other related tasks.

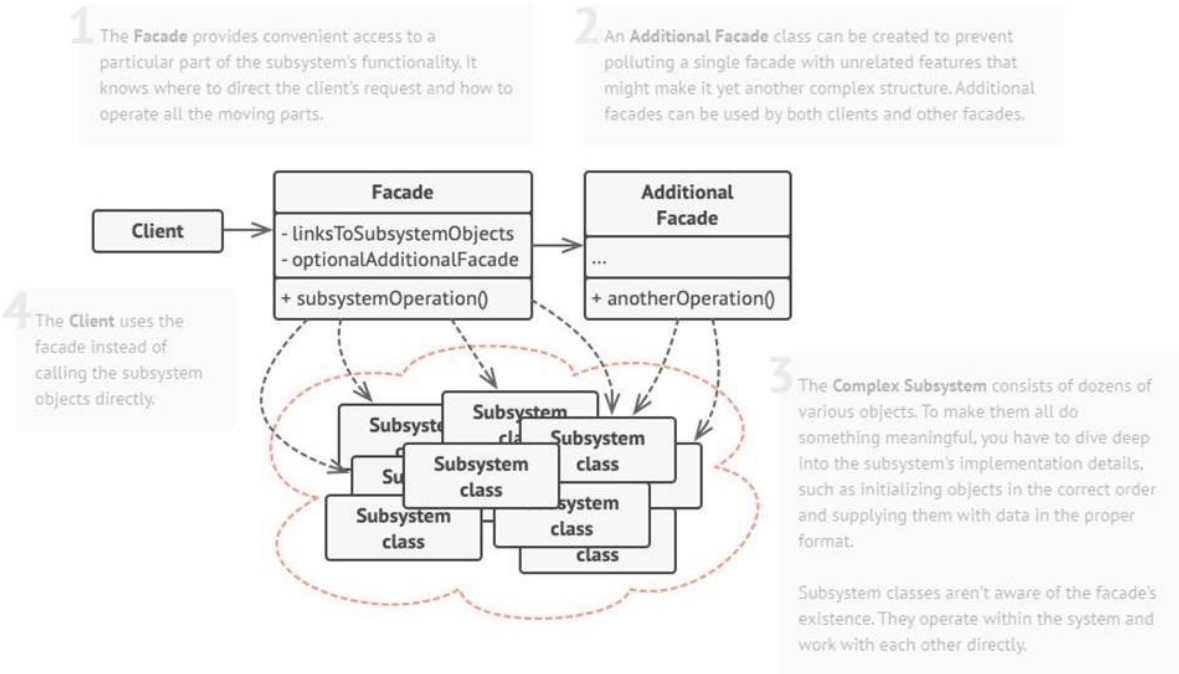
As a result, the operational logic of your classes would get closely intertwined with the unique workings of third-party classes, resulting in code that is difficult to comprehend and maintain.

Solution:

A facade is a software design pattern that provides a simplified interface to a complicated subsystem consisting of several components. Although a façade may have less capability than direct subsystem contact, it specifically concentrates on providing just the necessary functionalities for customers.

Using a facade is advantageous when integrating your application with a complex library that has many functionalities, but your needs are small. For example, let's suppose an application that uploads short, funny videos of cats to social media. This application may make use of a sophisticated video conversion library. However, all it really requires is a class that has just one function, namely encode(filename, format). By implementing this class and integrating it with the video conversion library, you construct a functional façade.

Structure:



F. Flyweight

Definition:

The Flyweight pattern is a structural design pattern that optimizes memory use by sharing common state across several objects, rather than duplicating the data in each object.

Problem:

In the realm of object-oriented programming, there are instances where generating a considerable quantity of akin objects becomes necessary. This circumstance can potentially give rise to challenges related to memory consumption, particularly when these objects harbor substantial amounts of state.

Solution:

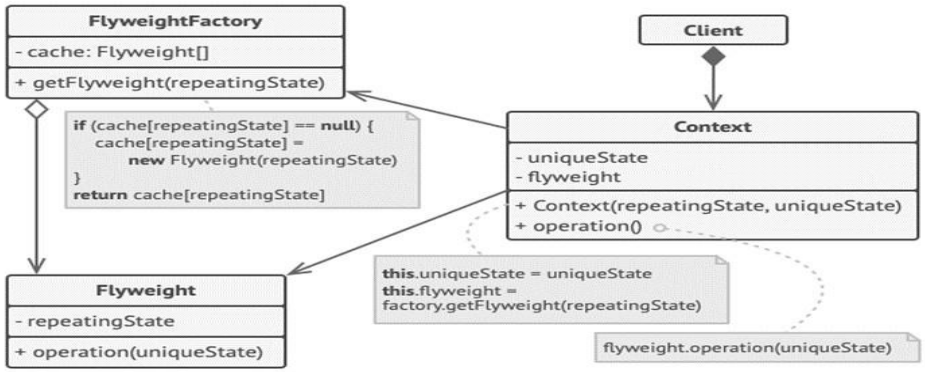
The flyweight pattern tackles this issue by enabling the sharing of identical object states. This is accomplished by storing the state in a distinct flyweight object and subsequently referencing that object from all instances requiring the same state.

Structure:

1 The Flyweight pattern is merely an optimization. Before applying it, make sure your program does have the RAM consumption problem related to having a massive number of similar objects in memory at the same time. Make sure that this problem can't be solved in any other meaningful way.

5 The **Client** calculates or stores the extrinsic state of flyweights. From the client's perspective, a flyweight is a template object which can be configured at runtime by passing some contextual data into parameters of its methods.

6 The **Flyweight Factory** manages a pool of existing flyweights. With the factory, clients don't create flyweights directly. Instead, they call the factory, passing it bits of the intrinsic state of the desired flyweight. The factory looks over previously created flyweights and either returns an existing one that matches search criteria or creates a new one if nothing is found.



2 The **Flyweight** class contains the portion of the original object's state that can be shared between multiple objects. The same flyweight object can be used in many different contexts. The state stored inside a flyweight is called *intrinsic*. The state passed to the flyweight's methods is called *extrinsic*.

3 The **Context** class contains the extrinsic state, unique across all original objects. When a context is paired with one of the flyweight objects, it represents the full state of the original object.

4 Usually, the behavior of the original object remains in the flyweight class. In this case, whoever calls a flyweight's method must also pass appropriate bits of the extrinsic state into the method's parameters. On the other hand, the behavior can be moved to the context class, which would use the linked flyweight merely as a data object.

G. Prox

y

Definitio

n:

The proxy pattern is a structural design pattern that enables the provision of a replacement or placeholder for another item. A proxy manages access to the actual object, enabling the execution of actions either before or after the request is sent to the underlying object.

Problem:

What may be the rationale for imposing restrictions on access to an object? Imagine this situation: you have a significant item that uses a large amount of system resources. Although you may need it sometimes, it is not a continuous need.

An effective strategy is to use lazy initialization, so the object is created only when it is required. Nevertheless, implementing this approach would probably lead to substantial code duplication, since every user of the object would be required to perform postponed initialization code.

Ideally, it would be more desirable to explicitly include this code into the class of the object. However, this may not always be possible, particularly if the class is integrated inside a proprietary third-party library.

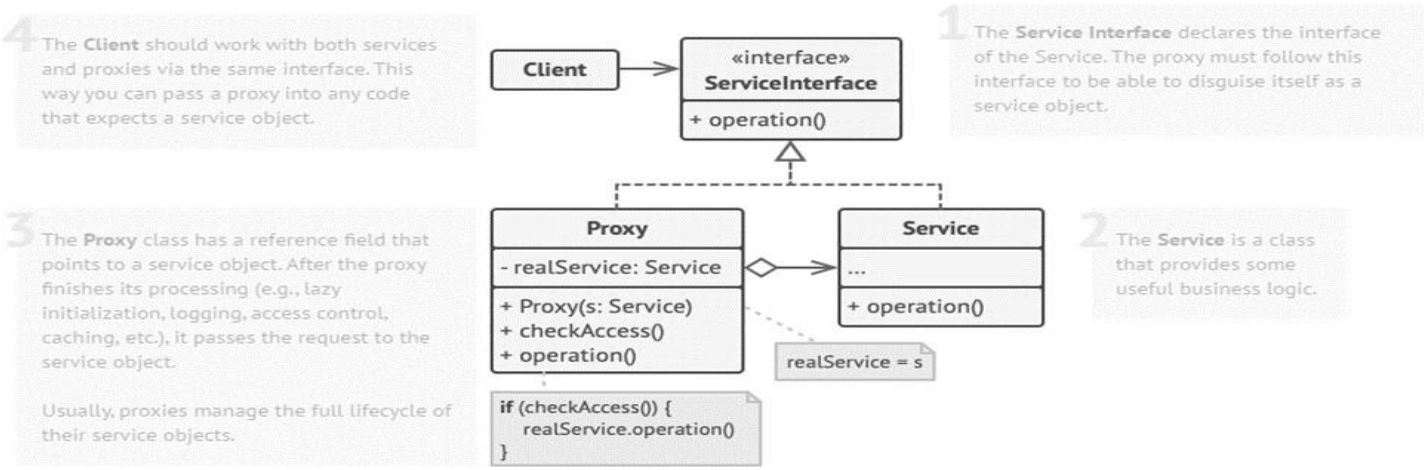
Solution:

The Proxy pattern suggests creating a new proxy class that has the same interface as the original service object. Afterwards, you make changes to your application such that the proxy object is given to all clients of the original object. Upon receiving a request from a client, the proxy creates a concrete service object and delegates all duties to it.

What is the benefit? If there is a need to execute actions before or after the main logic of the class, the

proxy enables you to do this without making any modifications to the class itself. Since the proxy follows the same interface as the original class, it can easily work with any client that expects a real service object.

Structure:



Description of a structural scenario

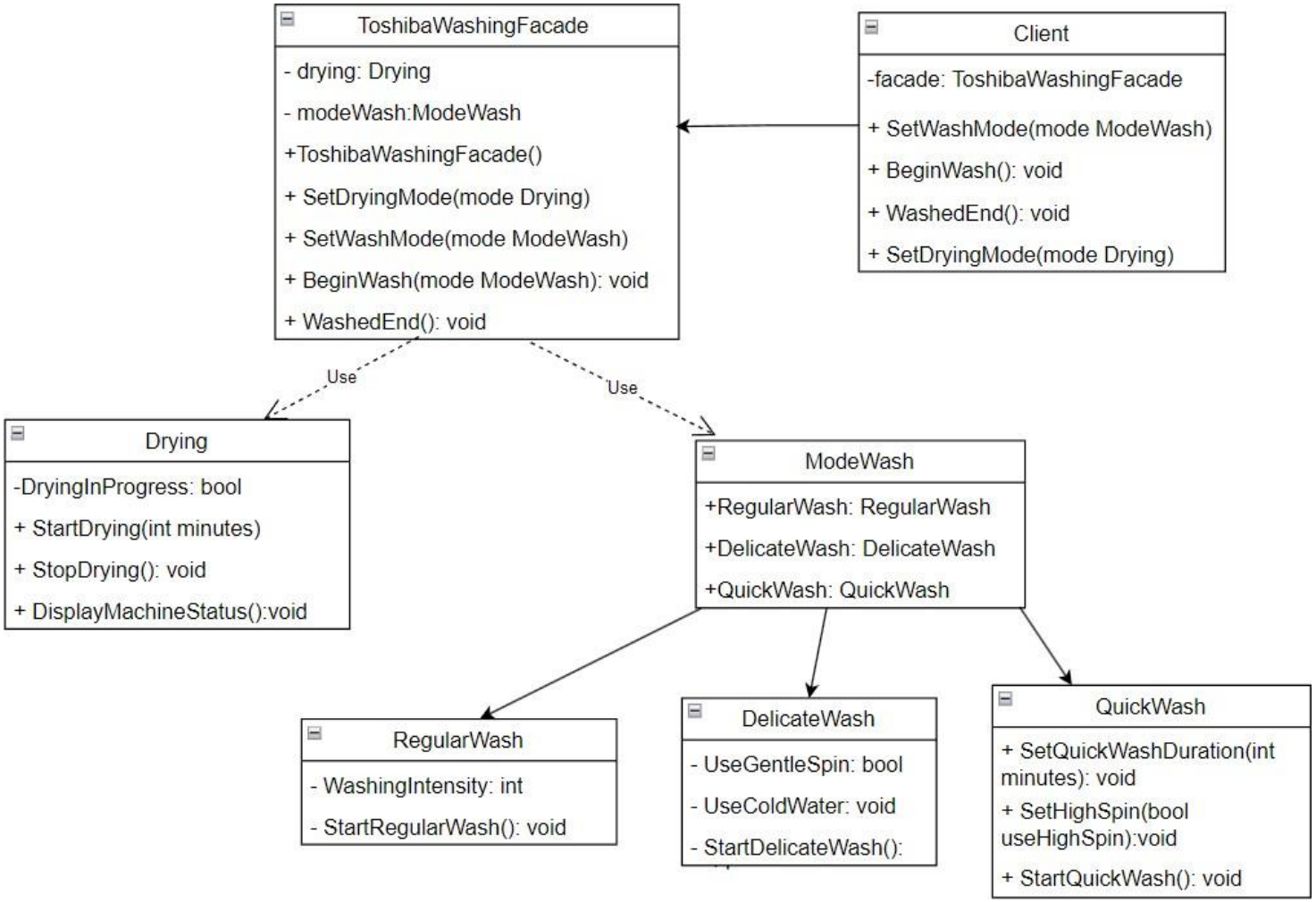
In a household, imagine a situation where ToshibaWashingFacade takes center stage, simplifying the laundry process with multiple modes. When you use ToshibaWashing, washing will be easier for you. Consider a situation in which a family's laundry needs vary widely, from regular clothes to delicate fabrics. You decide to start a comprehensive washing and drying cycle with the ToshibaWashingFacade. With a few simple commands, you set the washing intensity for regular clothes, select gentle spin and cold water for more sensitive items and select quick wash to save time.

When the washing machine starts operating, it seamlessly coordinates the activation of the washing components, ensuring that each stage complies with the specified parameters.

RegularWash starts with a defined strength, followed by DelicateWash, where the gentle spin and cold water settings are expertly applied. The QuickWash phase will then begin, with the selected duration and high spin priority applied. But ToshibaWashing doesn't stop there. It transitions seamlessly to the Drying subsystem, where you have specified drying times. Facade coordinates the drying process, ensuring your clothes are not only clean but perfectly dry.

In this laundry journey, ToshibaWashing plays an important role in providing a streamlined and user-friendly experience. The program ends by displaying the final machine status, giving you the assurance that your laundry is not only done, but done with precision and efficiency.

Diagram + explanation:



Drying Class:
Methods:
StartDrying(int minutes): Initiates the drying process for the specified duration. StopDrying(): Stops the drying process.
DisplayMachineStatus(): Displays the current drying status. RegularWash Class:

Attributes:
WashingIntensity (public int): Shows the level of
washing Methods:
StartRegularWash(): Initiates a regular wash with the specified
intensity. DelicateWash Class:
Methods:
StartDelicateWash(): Initiates a delicate wash with specified settings for gentle spin
and cold water. QuickWash Class:
Attributes:
QuickWashDuration
(public int) UseHighSpin
(public bool) Methods:
StartQuickWash(): Initiates a quick wash with the specified duration and
high spin settings. ModeWash Class:
Attributes:
RegularWash (public
RegularWash) DelicateWash
(public DelicateWash)
QuickWash (public
QuickWash)
ToshibaWashingFacade
Class:
Attributes: Use washing mode and drying
mode drying (private Drying)
modeWash (private
ModeWash) Methods:
SetDryingMode(int minutes): Initiates the drying process with a specified
duration. SetWashMode(string mode): Sets the wash mode based on the
provided string.
BeginWash(string mode): Begins the wash process for the
specified mode. WashedEnd(): Indicates the completion of
the washing process.

StopDrying(): Stops the drying process.
DisplayMachineStatus(): Displays the current machine status.

In this class diagram, the ToshibaWashingFacade acts as a higher-level interface, encapsulating the interaction between the drying subsystem (Drying) and the various washing modes (RegularWash, DelicateWash, QuickWash) through the ModeWash subsystem.

c. Behavioral pattern

Introduce creational pattern Chain of Responsibility

The Chain of Responsibility Pattern is a behavioral design pattern that enables an object to pass a request along a chain of potential handlers. Upon receiving a request, each handler decides either to process the request or to pass it to the next handler in the chain. This pattern decouples the sender and receiver objects, allowing multiple objects to handle the request without the sender needing to know which object will ultimately process it.

Problem

Imagine that you're working on an online ordering system. You want to restrict access to the system so only authenticated users can create orders. Also, users who have administrative permissions must have full access to all orders.

After a bit of planning, you realized that these checks must be performed sequentially. The application can attempt to authenticate a user to the system whenever it receives a request that contains the user's credentials. However, if those credentials aren't correct and authentication fails, there's no reason to proceed with any other checks.

During the next few months, you implemented several more of those sequential checks.

- One of your colleagues suggested that it's unsafe to pass raw data straight to the ordering system. So you added an extra validation step to sanitize the data in a request.
- Later, somebody noticed that the system is vulnerable to brute force password cracking. To negate this, you promptly added a check that filters repeated failed requests coming from the same IP address.
- Someone else suggested that you could speed up the system by returning cached results on repeated requests containing the same data. Hence, you added another check which lets the request pass through to the system only if there's no suitable cached response.

Solution

Like many other behavioral design patterns, the Chain of Responsibility relies on transforming particular

behaviors into stand-alone objects called *handlers*. In our case, each check should be extracted to its own class with a single method that performs the check. The request, along with its data, is passed to this method as an argument.

The pattern suggests that you link these handlers into a chain. Each linked handler has a field for storing a reference to the next handler in the chain. In addition to processing a request, handlers pass the request further along the chain. The request travels along the chain until all handlers have had a chance to process it.

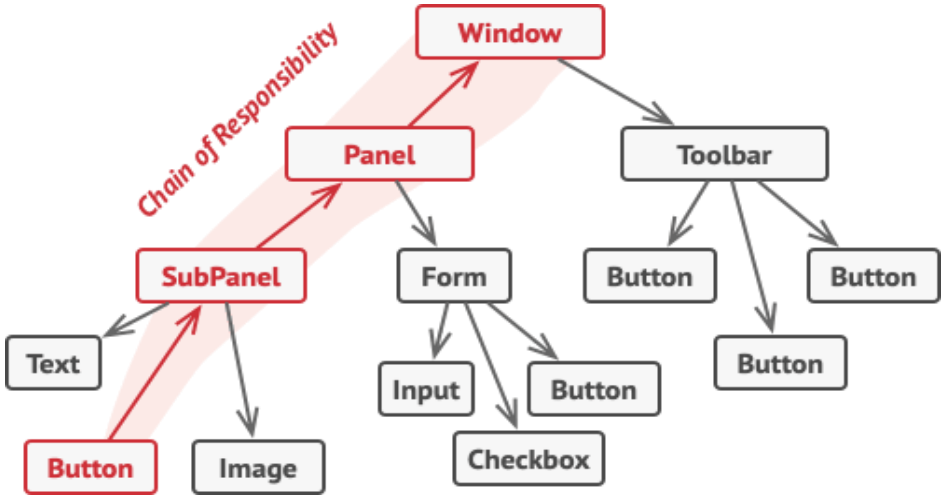
Here's the best part: a handler can decide not to pass the request further down the chain and effectively stop any further processing.

In our example with ordering systems, a handler performs the processing and then decides whether to pass the request further down the chain. Assuming the request contains the right data, all the handlers can execute their primary behavior, whether it's authentication checks or caching.

However, there's a slightly different approach (and it's a bit more canonical) in which, upon receiving a request, a handler decides whether it can process it. If it can, it doesn't pass the request any further. So it's either only one handler that processes the request or none at all. This approach is very common when dealing with events in stacks of elements within a graphical user interface.

For instance, when a user clicks a button, the event propagates through the chain of GUI elements that starts with the button, goes along its containers (like forms or panels), and ends up with the main application window. The event is processed by the first element in the chain that's capable of handling it. This example is also noteworthy because it shows that a chain can always be extracted from an object tree.

It's crucial that all handler classes implement the same interface. Each concrete handler should only care about the following one having the execute method. This way you can compose chains at runtime,



using various handlers without coupling your code to their concrete classes.

Command

Command is a behavioral design pattern that turns a request into a stand-alone object that contains all information about the request. This transformation lets you pass requests as a method arguments, delay

or queue a request’s execution, and support undoable operations.

Problem

Imagine that you're working on a new text-editor app. Your current task is to create a toolbar with a bunch of buttons for various operations of the editor. You created a very neat Button class that can be used for buttons on the toolbar, as well as for generic buttons in various dialogs.

While all of these buttons look similar, they're all supposed to do different things. Where would you put the code for the various click handlers of these buttons? The simplest solution is to create tons of subclasses for each place where the button is used. These subclasses would contain the code that would have to be executed on a button click.

Before long, you realize that this approach is deeply flawed. First, you have an enormous number of subclasses, and that would be okay if you weren't risking breaking the code in these subclasses each time you modify the base Button class. Put simply, your GUI code has become awkwardly dependent on the volatile code of the business logic.

And here's the ugliest part. Some operations, such as copying/pasting text, would need to be invoked from multiple places. For example, a user could click a small "Copy" button on the toolbar, or copy something via the context menu, or just hit Ctrl+C on the keyboard.

Initially, when our app only had the toolbar, it was okay to place the implementation of various operations into the button subclasses. In other words, having the code for copying text inside the CopyButton subclass was fine. But then, when you implement context menus, shortcuts, and other stuff, you have to either duplicate the operation's code in many classes or make menus dependent on buttons, which is an even worse option.

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.

In the code it might look like this: a GUI object calls a method of a business logic object, passing it some arguments. This process is usually described as one object sending another a *request*.

The Command pattern suggests that GUI objects shouldn't send these requests directly. Instead, you should extract all of the request details, such as the object being called, the name of the method and the list of arguments into a separate *command* class with a single method that triggers this request.

Command objects serve as links between various GUI and business logic objects. From now on, the GUI object doesn't need to know what business logic object will receive the request and how it'll be processed. The GUI object just triggers the command, which handles all the details.

The next step is to make your commands implement the same interface. Usually it has just a single execution method that takes no parameters. This interface lets you use various commands with the same request sender, without coupling it to

concrete classes of commands. As a bonus, now you can switch command objects linked to the sender, effectively changing the sender's behavior at runtime.

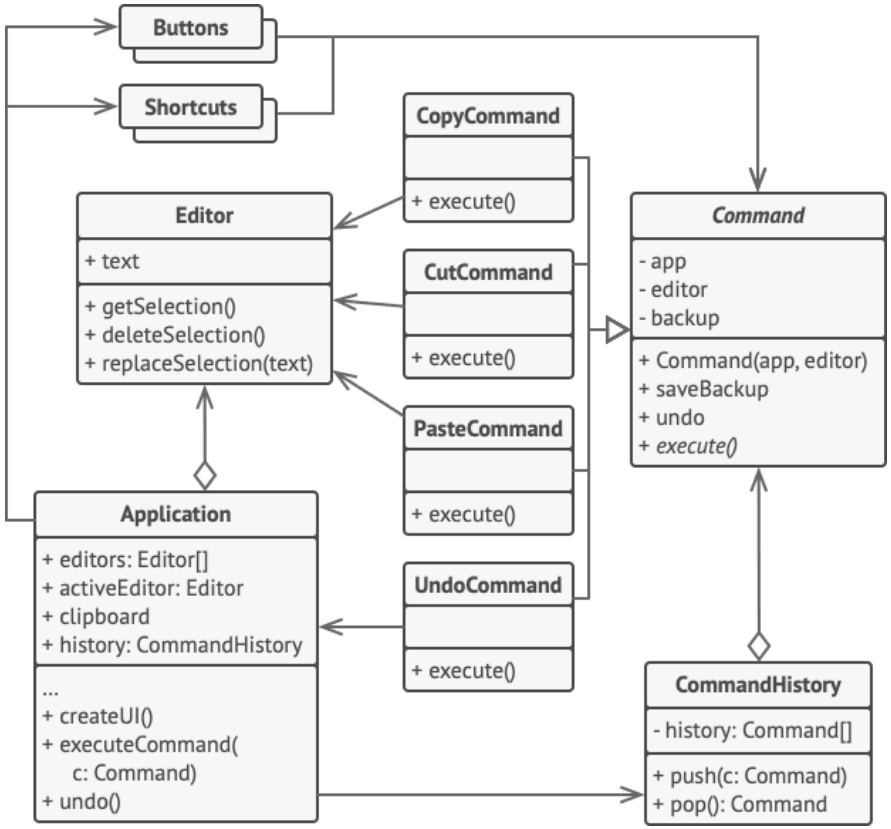
You might have noticed one missing piece of the puzzle, which is the request parameters. A GUI object might have supplied the business-layer object with some parameters. Since the command execution method doesn't have any parameters, how would we pass the request details to the receiver? It turns out the command should be either pre-configured with this data, or capable of getting it on its own.

Let's get back to our text editor. After we apply the Command pattern, we no longer need all those button subclasses to implement various click behaviors. It's enough to put a single field into the base Button class that stores a reference to a command object and make the button execute that command on a click.

You'll implement a bunch of command classes for every possible operation and link them with particular buttons, depending on the buttons' intended behavior.

Other GUI elements, such as menus, shortcuts or entire dialogs, can be implemented in the same way. They'll be linked to a command which gets executed when a user interacts with the GUI element. As you've probably guessed by now, the elements related to the same operations will be linked to the same commands, preventing any code duplication.

As a result, commands become a convenient middle layer that reduces coupling between the GUI and business logic layers. And that's only a fraction of the benefits that the Command pattern can offer!



Iterato

r

Intent

Iterator is a behavioral design pattern that lets you traverse elements of a collection without exposing its underlying representation (list, stack, tree, etc.).

Problem

Collections are one of the most used data types in programming. Nonetheless, a collection is just a container for a group of objects.

Most collections store their elements in simple lists. However, some of them are based on stacks,

trees, graphs and other complex data structures.

But no matter how a collection is structured, it must provide some way of accessing its elements so that other code can use these elements. There should be a way to go through each element of the collection without accessing the same elements over and over.

This may sound like an easy job if you have a collection based on a list. You just loop over all of the elements. But how do you sequentially traverse elements of a complex data structure, such as a tree? For example, one day you might be just fine with depth-first traversal of a tree. Yet the next day you might require breadth-first traversal. And the next week, you might need something else, like random access to the tree elements.

Adding more and more traversal algorithms to the collection gradually blurs its primary responsibility, which is efficient data storage. Additionally, some algorithms might be tailored for a specific application, so including them into a generic collection class would be weird.

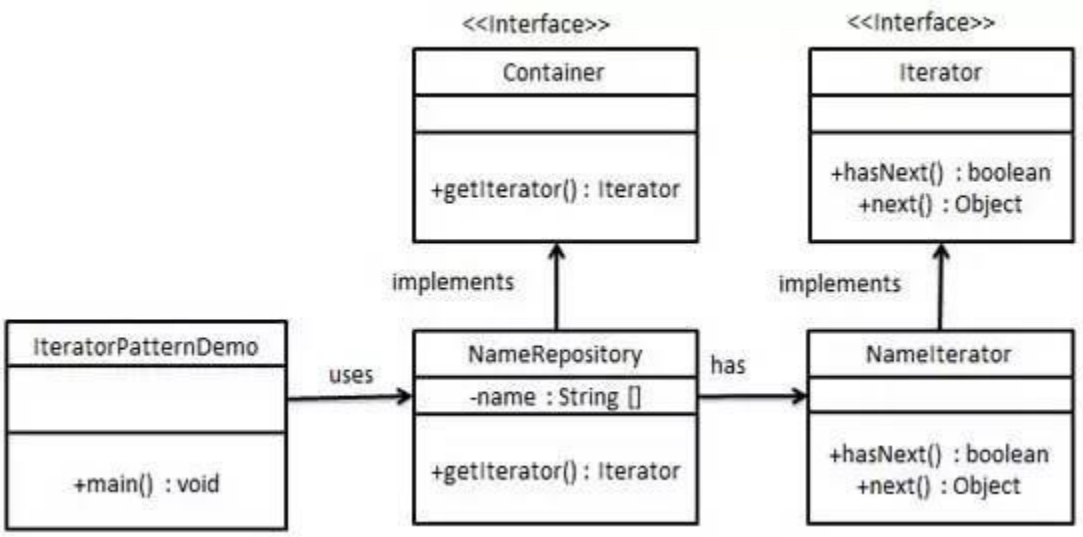
On the other hand, the client code that's supposed to work with various collections may not even care how they store their elements. However, since collections all provide different ways of accessing their elements, you have no option other than to couple your code to the specific collection classes.

Solution

The main idea of the Iterator pattern is to extract the traversal behavior of a collection into a separate object called an *iterator*. In addition to implementing the algorithm itself, an iterator object encapsulates all of the traversal details, such as the current position and how many elements are left till the end. Because of this, several iterators can go through the same collection at the same time, independently of each other.

Usually, iterators provide one primary method for fetching elements of the collection. The client can keep running this method until it doesn't return anything, which means that the iterator has traversed all of the elements.

All iterators must implement the same interface. This makes the client code compatible with any collection type or any traversal algorithm as long as there's a proper iterator. If you need a special way to traverse a collection, you just create a new iterator class, without having to change the collection or the client.



Mediat

or

Intent

Mediator is a behavioral design pattern that lets you reduce chaotic dependencies between objects. The pattern restricts direct communications between the objects and forces them to collaborate only via a mediator object.

Problem

Say you have a dialog for creating and editing customer profiles. It consists of various form controls such as text fields, checkboxes, buttons, etc.

Some of the form elements may interact with others. For instance, selecting the “I have a dog” checkbox may reveal a hidden text field for entering the dog’s name. Another example is the submit button that has to validate values of all fields before saving the data.

By having this logic implemented directly inside the code of the form elements you make these elements’ classes much harder to reuse in other forms of the app. For example, you won’t be able to use that checkbox class inside another form, because it’s coupled to the dog’s text field. You can use either all the classes involved in rendering the profile form, or none at all.

Solution

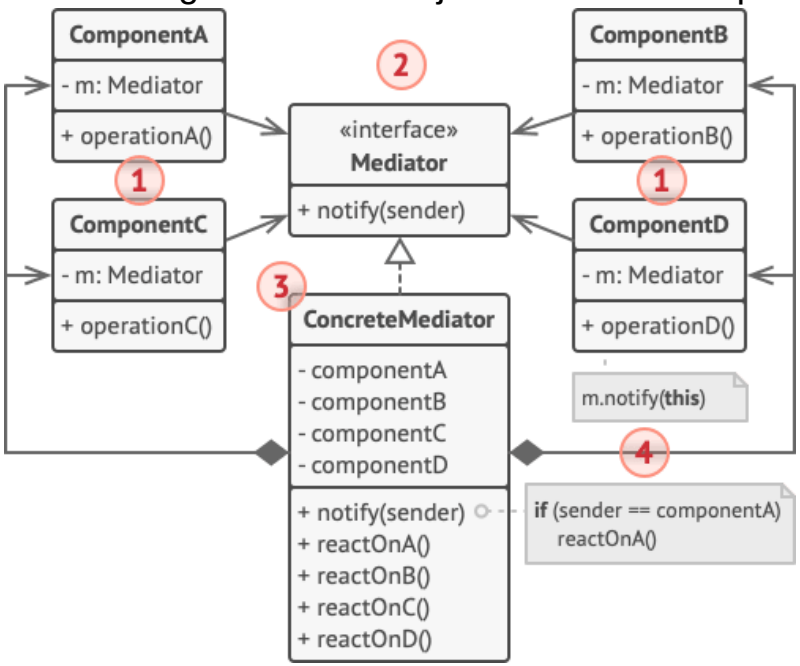
The Mediator pattern suggests that you should cease all direct communication between the components which you want to make independent of each other. Instead, these components must collaborate indirectly, by calling a special mediator object that redirects the calls to appropriate components. As a result, the components depend only on a single mediator class instead of being coupled to dozens of their colleagues.

In our example with the profile editing form, the dialog class itself may act as the mediator. Most likely, the dialog class is already aware of all of its sub-elements, so you won't even need to introduce new dependencies into this class.

The most significant change happens to the actual form elements. Let's consider the submit button. Previously, each time a user clicked the button, it had to validate the values of all individual form elements. Now its single job is to notify the dialog about the click. Upon receiving this notification, the dialog itself performs the validations or passes the task to the individual elements. Thus, instead of being tied to a dozen form elements, the button is only dependent on the dialog class.

You can go further and make the dependency even looser by extracting the common interface for all types of dialogs. The interface would declare the notification method which all form elements can use to notify the dialog about events happening to those elements. Thus, our submit button should now be able to work with any dialog that implements that interface.

This way, the Mediator pattern lets you encapsulate a complex web of relations between various objects inside a single mediator object. The fewer dependencies a class has, the easier it becomes to modify,



extend or reuse that class.

Mement

o Intent

Memento is a behavioral design pattern that lets you save and restore the previous state of an object without revealing the details of its implementation.

Problem

Imagine that you're creating a text editor app. In addition to simple text editing, your editor can format text, insert inline images, etc.

At some point, you decided to let users undo any operations carried out on the text. This feature has become so common over the years that nowadays people expect every app to have it. For the implementation, you chose to take the direct approach. Before performing any operation, the app records the state of all objects and saves it in some storage. Later, when a user decides to revert an action, the app fetches the latest snapshot from the history and uses it to restore the state of all objects. Let's think about those state snapshots. How exactly would you produce one? You'd probably need to go over all the fields in an object and copy their values into storage. However, this would only work if the object had quite relaxed access restrictions to its contents. Unfortunately, most real objects won't let others peek inside them that easily, hiding all significant data in private fields.

Ignore that problem for now and let's assume that our objects behave like hippies: preferring open relations and keeping their state public. While this approach would solve the immediate problem and let you produce snapshots of objects' states at will, it still has some serious issues. In the future, you might decide to refactor some of the editor classes, or add or remove some of the fields. Sounds easy, but this would also require changing the classes responsible for copying the state of the affected objects.

But there's more. Let's consider the actual "snapshots" of the editor's state. What data does it contain? At a bare minimum, it must contain the actual text, cursor coordinates, current scroll position, etc. To make a snapshot, you'd need to collect these values and put them into some kind of container.

Most likely, you're going to store lots of these container objects inside some list that would represent the history. Therefore the containers would probably end up being objects of one class. The class would have almost no methods, but lots of fields that mirror the editor's state. To allow other objects to write and read data to and from a snapshot, you'd probably need to make its fields public. That would expose all the editor's states, private or not. Other classes would become dependent on every little change to the snapshot class, which would otherwise happen within private fields and methods without affecting outer classes.

It looks like we've reached a dead end: you either expose all internal details of classes, making them

too fragile, or restrict access to their state, making it impossible to produce snapshots. Is there any other way to implement the "undo"?

Solution

All problems that we've just experienced are caused by broken encapsulation. Some objects try to do more than they are supposed to. To collect the data required to perform some action, they invade the private space of other objects instead of letting these objects perform the actual action.

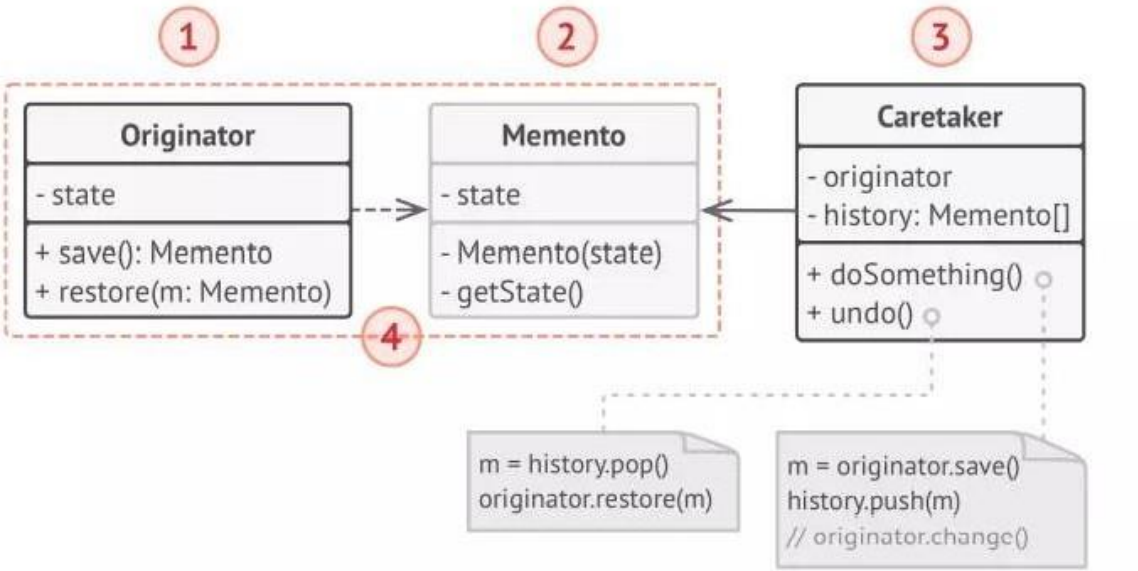
The Memento pattern delegates creating the state snapshots to the actual owner of that state, the *originator* object. Hence, instead of other objects trying to copy the editor's state from the "outside," the editor class itself can make the snapshot since it has full access to its own state.

The pattern suggests storing the copy of the object's state in a special object called *memento*. The contents of the memento aren't accessible to any other object except the one that produced it. Other objects must communicate with mementos using a limited interface which may allow fetching the snapshot's metadata (creation time, the name of the performed operation, etc.), but not the original object's state contained in the snapshot.

Such a restrictive policy lets you store mementos inside other objects, usually called *caretakers*. Since the caretaker works with the memento only via the limited interface, it's not able to tamper with the state stored inside the memento. At the same time, the originator has access to all fields inside the memento, allowing it to restore its previous state at will.

In our text editor example, we can create a separate history class to act as the caretaker. A stack of mementos stored inside the caretaker will grow each time the editor is about to execute an operation. You could even render this stack within the app's UI, displaying the history of previously performed operations to a user.

When a user triggers the undo, the history grabs the most recent memento from the stack and passes it back to the editor, requesting a roll-back. Since the editor has full access to the memento, it changes its own state with the values taken from the memento.



Observ

er

Intent

Observer is a behavioral design pattern that lets you define a subscription mechanism to notify multiple objects about anyevents that happen to the object they're observing.

Problem

Imagine that you have two types of objects: a Customer and a Store. The customer is very interested in a particular brand of product (say, it's a new model of the iPhone) which should become available in the store very soon.

The customer could visit the store every day and check product availability. But while the product is still en route, most of these trips would be pointless.

On the other hand, the store could send tons of emails (which might be considered spam) to all customers each time a new product becomes available. This would save some customers from endless trips to the store. At the same time, it'd upset other customers who aren't interested in new products.

It looks like we've got a conflict. Either the customer wastes time checking product availability or the

store wastes resources notifying the wrong customers.

Solution

The object that has some interesting state is often called *subject*, but since it's also going to notify other objects about the changes to its state, we'll call it *publisher*. All other objects that want to track changes to the publisher's state are called *subscribers*.

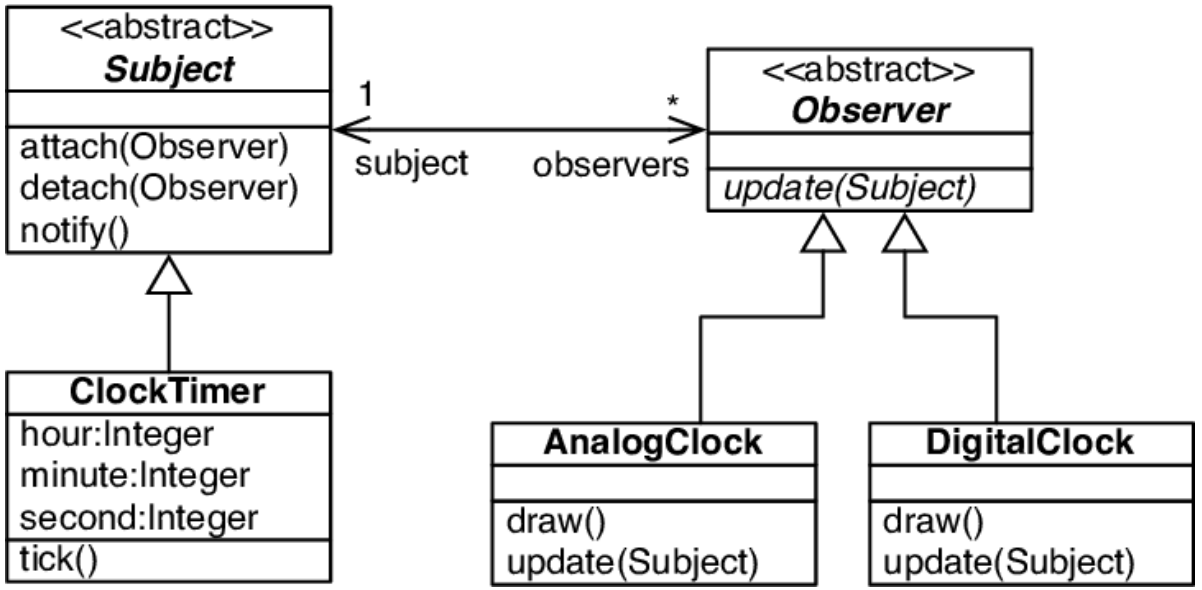
The Observer pattern suggests that you add a subscription mechanism to the publisher class so individual objects can subscribe to or unsubscribe from a stream of events coming from that publisher. Fear not! Everything isn't as complicated as it sounds. In reality, this mechanism consists of 1) an array field for storing a list of references to subscriber objects and 2) several public methods which allow adding subscribers to and removing them from that list.

Now, whenever an important event happens to the publisher, it goes over its subscribers and calls the specific notification method on their objects.

Real apps might have dozens of different subscriber classes that are interested in tracking events of the same publisher class. You wouldn't want to couple the publisher to all of those classes. Besides, you might not even know about some of them beforehand if your publisher class is supposed to be used by other people.

That's why it's crucial that all subscribers implement the same interface and that the publisher communicates with them only via that interface. This interface should declare the notification method along with a set of parameters that the publisher can use to pass some contextual data along with the notification.

If your app has several different types of publishers and you want to make your subscribers compatible with all of them, you can go even further and make all publishers follow the same interface. This interface would only need to describe a few subscription methods. The interface would allow subscribers to observe publishers' states without coupling to their concrete classes.



State

Intent

State is a behavioral design pattern that lets an object alter its behavior when its internal state changes. It appears as if the object changed its class.

Problem

The State pattern is closely related to the concept of a *Finite-State Machine* .

The main idea is that, at any given moment, there's a *finite* number of *states* which a program can be in. Within any unique state, the program behaves differently, and the program can be switched from one state to another instantaneously. However, depending on a current state, the program may or may not switch to certain other states. These switching rules, called *transitions*, are also finite and predetermined.

You can also apply this approach to objects. Imagine that we have a Document class. A document can be in one of three states: Draft, Moderation and Published. The publish method of the document works a little bit differently in each state:

- In Draft, it moves the document to moderation.
- In Moderation, it makes the document public, but only if the current user is an administrator.
- In Published, it doesn’t do anything at all.

State machines are usually implemented with lots of conditional statements (if or switch) that select the appropriate behavior depending on the current state of the object. Usually, this "state" is just a set of values of the object's fields. Even if you've never heard about finite-state machines before, you've probably implemented a state at least once. Does the following code structure ring a bell?

The biggest weakness of a state machine based on conditionals reveals itself once we start adding more and more states and state-dependent behaviors to the Document class. Most methods will contain monstrous conditionals that pick the proper behavior of a method according to the current state. Code like this is very difficult to maintain because any change to the transition logic may require changing state conditionals in every method.

The problem tends to get bigger as a project evolves. It's quite difficult to predict all possible states and transitions at the design stage. Hence, a lean state machine built with a limited set of conditionals can grow into a bloated mess over time.

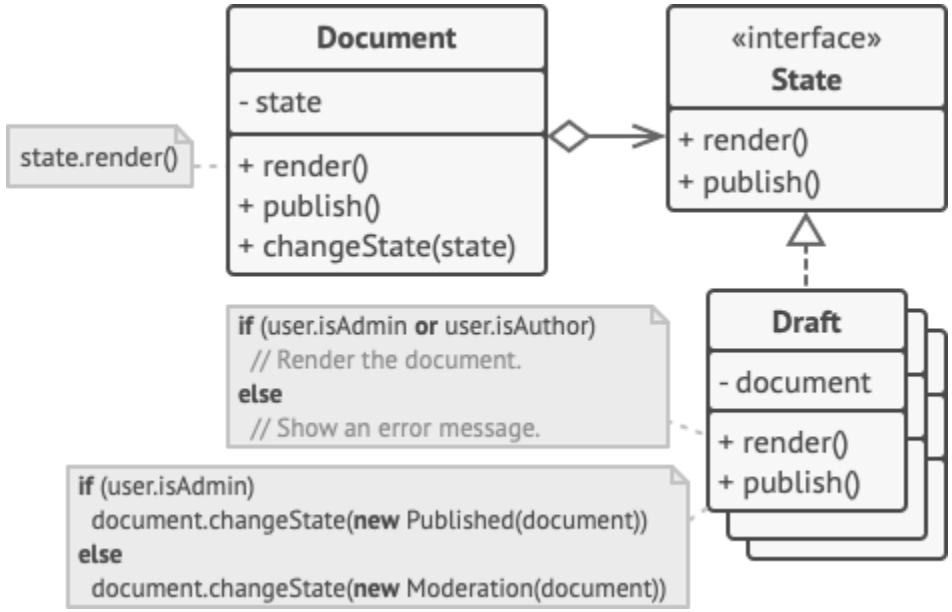
Solution

The State pattern suggests that you create new classes for all possible states of an object and extract all state-specific behaviors into these classes.

Instead of implementing all behaviors on its own, the original object, called *context*, stores a reference to one of the state objects that represents its current state, and delegates all the state-related work to that object.

To transition the context into another state, replace the active state object with another object that represents that new state. This is possible only if all state classes follow the same interface and the context itself works with these objects through that interface.

This structure may look similar to the **Strategy** pattern, but there's one key difference. In the State pattern, the particular states may be aware of each other and initiate transitions from one state to another, whereas strategies almost never know about each other.



Strategy

Intent

Strategy is a behavioral design pattern that lets you define a family of algorithms, put each of them into a separate class, and make their objects interchangeable.

Problem

One day you decided to create a navigation app for casual travelers. The app was centered around a beautiful map which helped users quickly orient themselves in any city.

One of the most requested features for the app was automatic route planning. A user should be able to enter an address and see the fastest route to that destination displayed on the map.

The first version of the app could only build the routes over roads. People who traveled by car were bursting with joy. But apparently, not everybody likes to drive on their vacation. So with the next update, you added an option to build walking routes. Right after that, you added another option to let people use public transport in their routes.

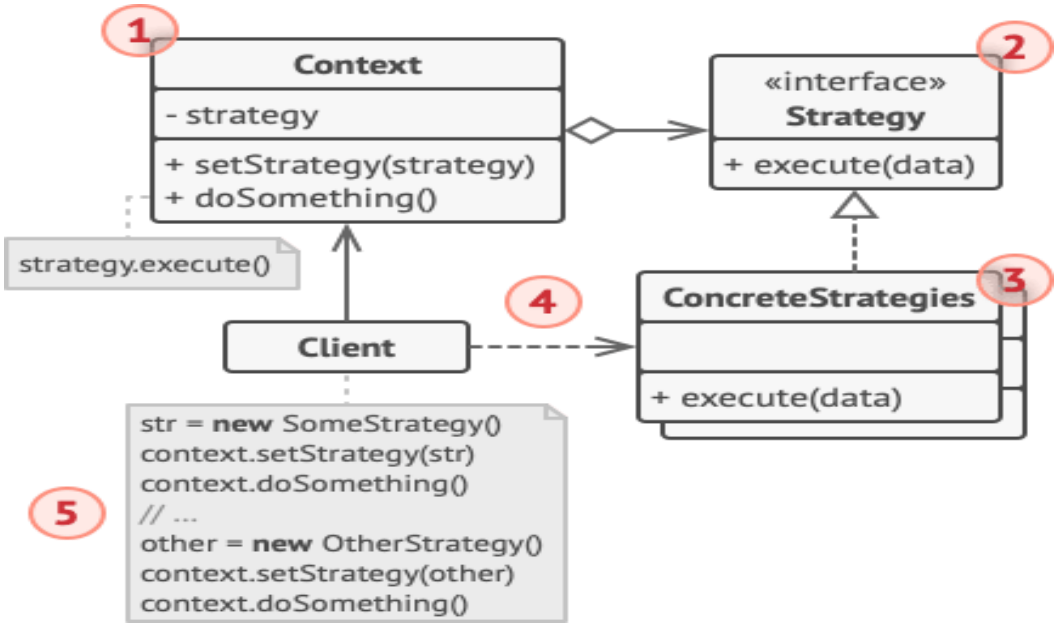
However, that was only the beginning. Later you planned to add route building for cyclists. And even later, another option for building routes through all of a city's tourist attractions.

While from a business perspective the app was a success, the technical part caused you many headaches. Each time you added a new routing algorithm, the main class of the navigator doubled in size. At some point, the beast 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.

Solution

The Strategy pattern suggests that you take a class that does something specific in a lot of different ways and extract all of 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.
The context isn't responsible for selecting an appropriate algorithm for the job. Instead, the client passes the desired strategy to the context. In fact, the context doesn't know much about strategies. It works with all strategies through the same generic interface, which only exposes a single method for triggering the algorithm encapsulated within the selected strategy.
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.
In our navigation app, each routing algorithm can be extracted to its own class with a single `buildRoute` method. The method accepts an origin and destination and returns a collection of the route's checkpoints.
Even though given the same arguments, each routing class might build a different route, the main navigator class doesn't really care which algorithm is selected since its primary job is to render a set of checkpoints on the map. The class has a method for switching the active routing strategy, so its clients, such as the buttons in the user interface, can replace the currently selected routing behavior with another one.



Template

Method Intent

Template Method is a behavioral design pattern that defines the skeleton of an algorithm in the superclass but lets subclasses override specific steps of the algorithm without changing its structure.

Problem

Imagine that you're creating a data mining application that analyzes corporate documents. Users feed the app documents in various formats (PDF, DOC, CSV), and it tries to extract meaningful data from these docs in a uniform format.

The first version of the app could work only with DOC files. In the following version, it was able to support CSV files. A month later, you "taught" it to extract data from PDF files.

At some point, you noticed that all three 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. Wouldn't it be great to get rid of the code duplication, leaving the algorithm structure intact?

There was another problem related to 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.

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.* The steps may either be abstract, or have some default implementation. To use the algorithm, the client is supposed to provide its own subclass, implement all abstract steps, and override some of the optional ones if needed (but not the template method itself).

Let's see how this will play out in our data mining app. We can create a base class for all three parsing algorithms. This class defines a template method consisting of a series of calls to various document-processing steps.

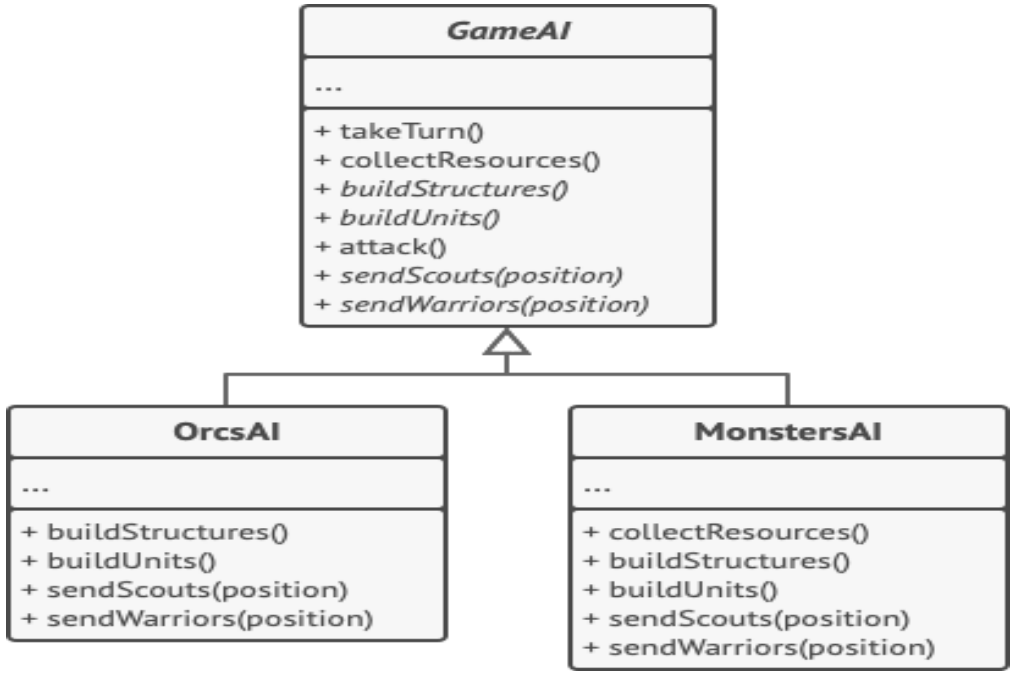
At first, we can declare all steps abstract, forcing the subclasses to provide their own implementations for these methods. In our case, subclasses already have all necessary implementations, so the only thing we might need to do is adjust signatures of the methods to match the methods of the superclass.

Now, let's see what we can do to get rid of the duplicate code. It looks like the code for opening/closing files and extracting/parsing data is different for various data formats, so there's no point in touching those methods. However, implementation of other steps, such as analyzing the raw data and composing reports, is very similar, so it can be pulled up into the base class, where subclasses can share that code.

As you can see, we've got two types of steps:

- *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.



Visitor

Intent

Visitor is a behavioral design pattern that lets you separate algorithms from the objects on which they operate.

Problem

Imagine that your team develops an app which works with geographic information structured as one colossal graph. Each node of the graph may represent a complex entity such as a city, but also more granular things like industries, sightseeing areas, etc. The nodes are connected with others if there's a road between the real objects that they represent. Under the hood, each node type is represented by its own class, while each specific node is an object.

At some point, you got a task to implement exporting the graph into XML format. At first, the job seemed pretty straightforward. You planned to add an export method to each node class and then leverage recursion to go over each node of the graph, executing the export method. The solution was simple and

elegant: thanks to polymorphism, you weren't coupling the code which called the export method to concrete classes of nodes.

Unfortunately, the system architect refused to allow you to alter existing node classes. He said that the code was already in production and he didn't want to risk breaking it because of a potential bug in your changes.

Besides, he questioned whether it makes sense to have the XML export code within the node classes. The primary job of these classes was to work with geodata. The XML export behavior would look alien there.

There was another reason for the refusal. It was highly likely that after this feature was implemented, someone from the marketing department would ask you to provide the ability to export into a different format, or request some other weird stuff. This would force you to change those precious and fragile classes again.

Solution

The Visitor pattern suggests that you place the new behavior into a separate class called *visitor*, instead of trying to integrate it into existing classes. The original object that had to perform the behavior is now passed to one of the visitor's methods as an argument, providing the method access to all necessary data contained within the object.

Now, what if that behavior can be executed over objects of different classes? For example, in our case with XML export, the actual implementation will probably be a little bit different across various node classes. Thus, the visitor class may define not one, but a set of methods, each of which could take arguments of different types, like this:

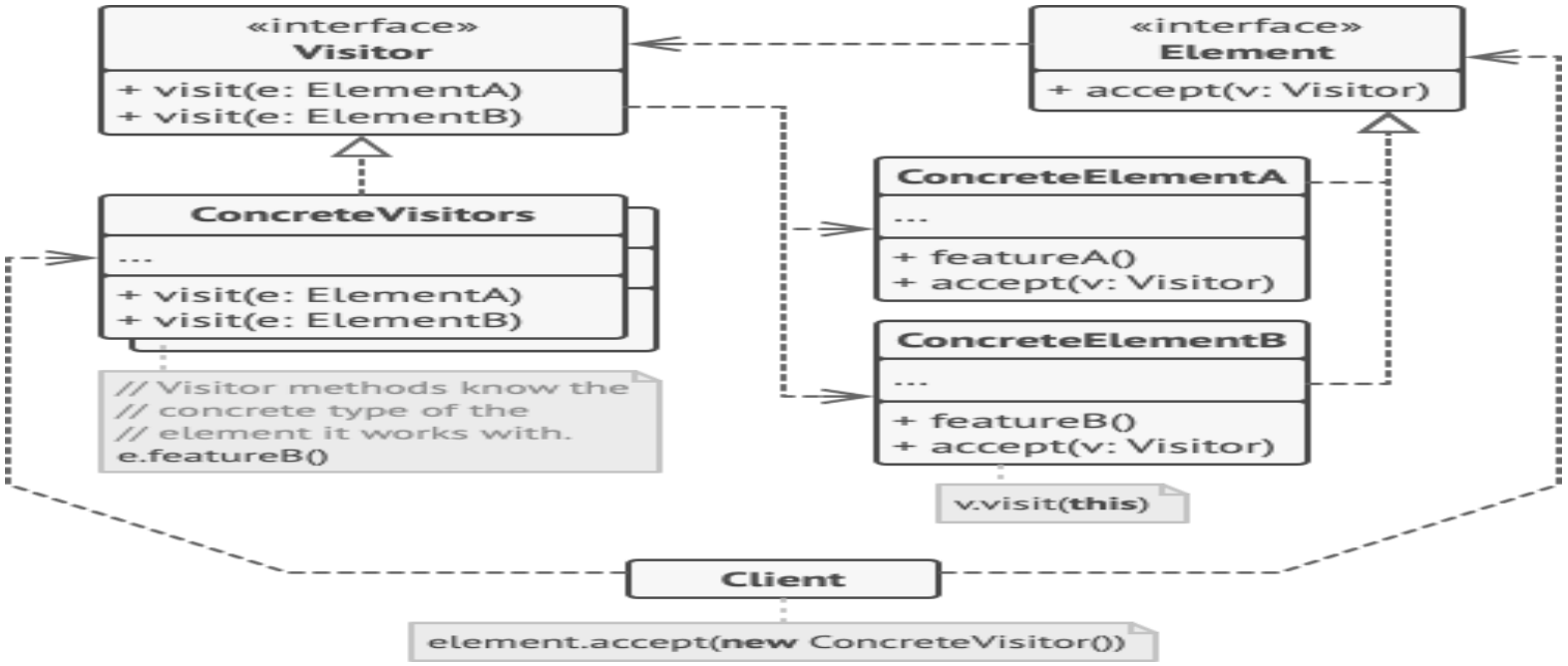
But how exactly would we call these methods, especially when dealing with the whole graph? These methods have different signatures, so we can't use polymorphism. To pick a proper visitor method that's able to process a given object, we'd need to check its class. Doesn't this sound like a nightmare?

You might ask, why don't we use method overloading? That's when you give all methods the same name, even if they support different sets of parameters. Unfortunately, even assuming that our programming language supports it at all (as Java and C# do), it won't help us. Since the exact class of a node object is unknown in advance, the overloading mechanism won't be able to determine the correct method to execute. It'll default to the method that takes an object of the base `Node` class.

However, the Visitor pattern addresses this problem. It uses a technique called **Double Dispatch**, which helps to execute the proper method on an object without cumbersome conditionals. Instead of letting the client select a proper version of the method to call, how about we delegate this choice to objects we're passing to the visitor as an argument? Since the objects know their own classes, they'll be able to pick a proper method on the visitor less awkwardly. They "accept" a visitor and tell it what visiting method should be executed.

I confess. We had to change the node classes after all. But at least the change is trivial and it lets us add further behaviors without altering the code once again.

Now, if we extract a common interface for all visitors, all existing nodes can work with any visitor you introduce into the app. If you find yourself introducing a new behavior related to nodes, all you have to do is implement a new visitor class.



Description of a behavioral scenario

A customer comes to shop at a computer accessories store and makes payment. The customer chooses the payment method from the available options: Bank transfer, PayPal or Cash.

Depending on the selected payment method, an instance of the corresponding payment strategy will be created:

For bank transfer, customers need to log in to the app with phone number and password. Then the number, when entering the recipient name account number will be displayed. The system authenticates login information, retrieves recipient account information, confirms bank transfer and processes payment. Finally, enter the OTP code to confirm successful payment. If it is not correct, payment will not be possible.

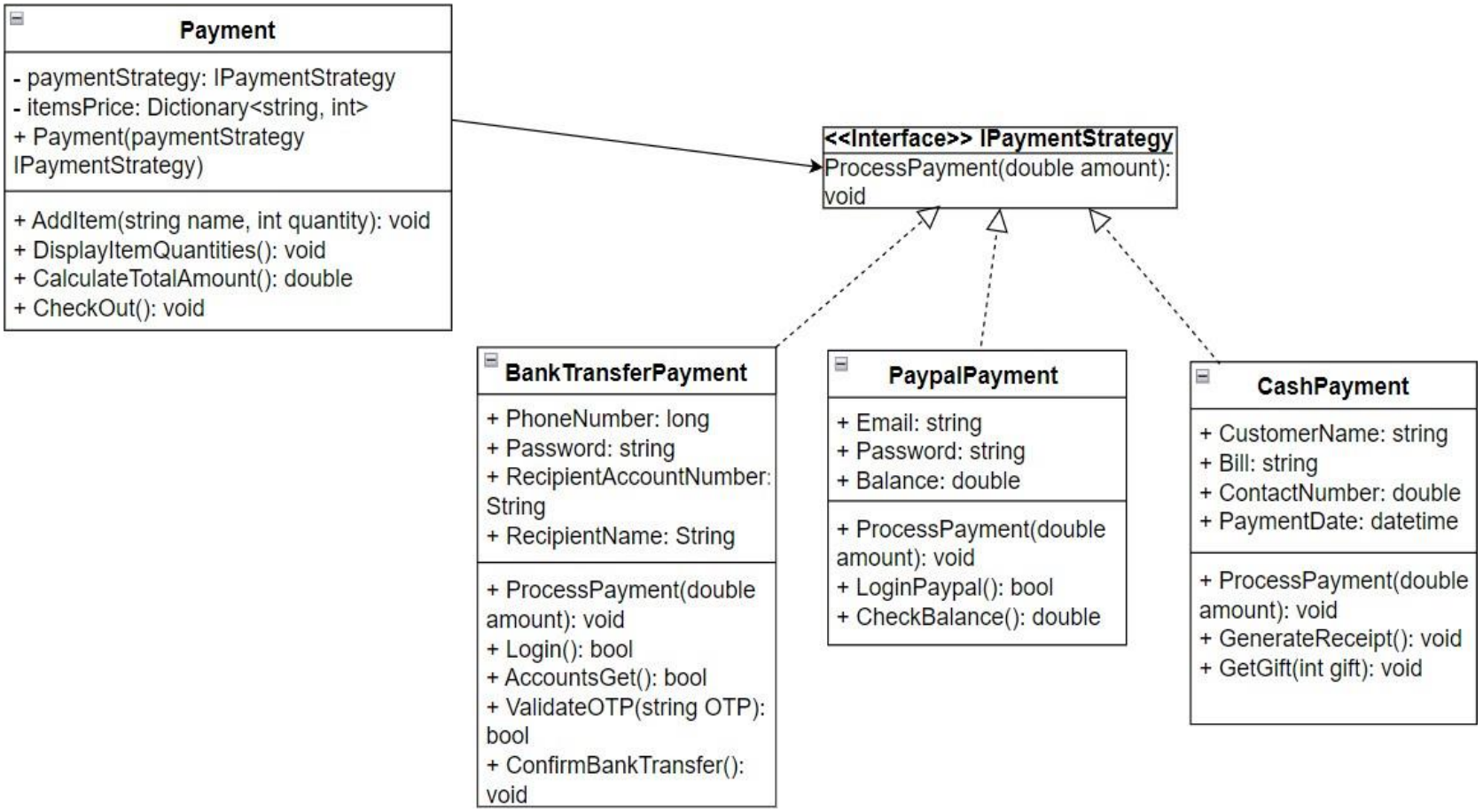
For PayPal, customers enter email and password to log in to their entire PayPal account. The system validates PayPal login information, checks balances, and processes payments. Customers can then check their account balance. Confirm payment to complete payment.

For Cash, customers provide their name, create an invoice, and enter contact information. The payment

date will be displayed on the bill. The system generates receipts and processes cash payments. The customer receives a confirmation message indicating whether the payment was successful or not. When paying in cash, you will receive a gift. There are 4 main types: Gift 1: keychain, Gift 2: Teddy bear, Gift 3: headphones, Gift 4: Power bank.

Customers add items to the list, specifying the quantity of each item. You can check the quantity of each type of goods and calculate their total value.

Diagram + explanation



The universal interface used by all payment methods is represented by the IPaymentStrategy (Interface). Declares a procedure called ProcessPayment (double amount), which needs to be put into practice through certain payment plans. Payment by Bank Transfer (Class using IPaymentStrategy): represents a bank transfer payment method. PhoneNumber, Password, RecipientAccountNumber, and RecipientName are examples of properties use the ProcessPayment method, which

includes account retrieval, validation, and confirmation of the OTP, along with login. PaypalPayment (Class: IPaymentStrategy Implementation): represents a PayPal payment method. Properties consist of Balance, Password, and Email carries out the ProcessPayment procedure, which includes processing payments, checking balances, and logging into PayPal. Cash Payment (Class: IPaymentStrategy Implementation): represents a cash transaction payment method. CustomerName, Bill, ContactNumber, and PaymentDate are examples of properties. Executes the ProcessPayment technique, which entails processing the cash payment and creating a receipt. GetGift(int gift) is another method that may be used to obtain a gift based on certain parameters. Payment (Class): Indicates the context class that has chosen a certain payment method. Has an IPaymentStrategy-class private field payment strategy that oversees an itemsPrice dictionary to record item quantities. The payment strategy is initialized by Constructor Payment (IPaymentStrategy paymentStrategy).
Techniques consist of:
AddItem: Adds a product to the list (string name, int quantity). DisplayItemQuantities(): Shows the items and quantities CalculateTotalAmount(): Determines the overall sum by using the prices and quantities of the items.
CheckOut(): Starts the checkout procedure, which includes utilizing the chosen strategy to process the payment.

5. Design Pattern vs OOP

Object-Oriented Programming (OOP) and Design Patterns are integral components of modern software development, each contributing distinct yet interconnected principles to the creation of robust and maintainable code.
Object-Oriented Programming (OOP):
OOP is a programming paradigm that revolves around the concept of objects, which are instances of classes encapsulating data and behavior. The four primary principles of OOP — encapsulation, inheritance, polymorphism, and abstraction — serve as the foundation for organizing code in a modular and reusable manner.
Encapsulation involves bundling data and methods within classes, promoting data security. Inheritance facilitates the creation of new classes by inheriting attributes and behaviors, fostering code reuse. Polymorphism enables flexibility by treating objects of different types as instances of a common base type. Abstraction simplifies complex systems by modeling classes based on their essential features.

OOP is not merely a set of guidelines but a paradigm that structures code around real-world entities, enhancing scalability and maintainability.

Design Patterns:

Design Patterns, introduced by the Gang of Four, are proven solutions to recurring design problems. They are categorized into creational, structural, and behavioral patterns. Creational patterns focus on object creation, structural patterns define class and object composition, and behavioral patterns address communication between objects.

Design Patterns offer a set of templates or blueprints for solving specific design challenges. Examples include the Singleton pattern ensuring a single instance of a class, the Observer pattern establishing dependencies, and the Factory pattern providing an interface for object instantiation.

6. Conclusion

The report begins by presenting readers with the fundamental concepts of Object-Oriented Programming (OOP), offering precise definitions and concrete examples. It proceeds to apply these concepts in a real-world context, accompanied by a use-case diagram for better illustration. The discussion then extends to the realm of design patterns, with a specific focus on behavioral patterns, showcasing scenarios and their practical implementations. Underscoring the interconnectedness of design patterns and OOP, the report highlights their role in enhancing the structural integrity, maintainability, and scalability of software. Serving as a conduit between theory and practical implementation, the report ensures that readers develop a comprehensive grasp of OOP principles and their hands-on application through design patterns.

References:

"\/." 2023. Learn OOP Basics - A Brief Guide.

https://www.google.com.vn/books/edition/Learn_OOP_Basics_A_Brief_Guide/pjDEEAAAQBAJ?hl=vi&gbpv=1&pg=PA1&printsec=frontcover.

Design Patterns. (n.d.). Refactoring.Guru. Retrieved December 8, 2023, from https://refactoring.guru/design-patterns