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	December 13th, 2022	**Date Received 1st submission**	December 13th, 2022
Re-submission Date		**Date Received 2nd submission**	
Student Name	Nguyen Tien Thinh	**Student ID**	GCH200796
Class	GCH1002	**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. I understand that making a false declaration is a form of malpractice.

Student's signature	[illegible]

Grading grid

P1	P2	M1	M2	D1	D2

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

Contents

Table of figures

I. Introduction

In this report, we provide an example of a hypothetical situation to use object-oriented design and analysis to explore the fundamental elements of the object-oriented programming paradigm (OOP). It may explain the many OOP properties, such as abstraction, polymorphism, inheritance, and encapsulation. We provided each wallet's diagram and code based on its OOP features. A good example will help you comprehend. In the second assignment, you will be expected to introduce a number of UML class diagrams for object-oriented analysis and design, including use case diagrams, class diagrams, sequence diagrams, activity diagrams, state diagrams, and flowcharts. We continued by giving examples of each form of chart we had previously mentioned and described. The audience is then given the task of learning about various design patterns, which are divided into three types: creativity, structure, and behavior. The related patterns are then shown using diagrams in real-world scenarios. UML classes and C# programming techniques for implementing them.

II. OOP general concepts

1. Introduction of OOP

a. History of OOP.

According to Purdum, J. (2008), Many people feel that OOP is a result of the 1980s and Bjarne Stroustrup's work in transforming the C language into an object-oriented language by establishing the C++ language. Actually, the first object-oriented languages were SIMULA 1 (1962) and SIMULA 67 (1967). Ole-John Dahl and Kristen Nygaard worked on the Simula languages at the Norwegian Computing Center in Oslo, Norway. While most of the benefits of OOP were accessible in previous Simula languages, it wasn't until C++ became entrenched in the 1990s that OOP really took off.

b. Definition of OOP.

According to Shet, P. (2019), Object-oriented programming is considered as a design method for building software that is not rigid. In OOPS, our jobs will be represented as objects and written in a logical way to accomplish. Problems will be broken down into small work units represented through objects and their functions. We build functions around objects. The four basic principles of object-oriented programming are Abstraction, Encapsulation, Inheritance and Polymorphism.

2. General concepts of OOP

According to website learn.microsoft.com (n.d.), some importance concepts of OOP are as follows:

a. Class.

In OOP, a class is an object description plan. Or is it the blueprint for how the object is represented. Primarily a class will consist of names, properties, and operations. There are some OOPS guidelines that need to be met while creating a class. This principle is called SOLID where each letter has some specification: Single Responsibility Principle (SRP), Open Closure Principle (OCP), Liskov Substitution Principle (LSP), Interface Separation Principle (ISP), Dependency Inversion Principle (DIP).

b. Objects.

An object can be thought of as any real-world entity that may have some characteristics or can perform some tasks. This object is also known as an entity copy in programming languages. An Object is an instance of a Class. Each object contains data and code to manipulate the data.

c. Properties and fields

Fields and properties represent the information an object contains. Fields can be read or set directly just like variables. Do properties provide more control over how values are set or returned by get and set procedures.

d. Methods.

A method is an action that an object can perform. In object-oriented programming, a method is the equivalent of a function. A noun is what a verb is to a variable, and methods are the actions that execute operations on variables. When a method is invoked on an object, it receives parameters as arguments, manipulates them, and then returns an output. Methods are analogous to functions, but in the class design, methods are additionally categorized based on their purpose. Variables are referred to as attributes in classes, hence methods frequently act on attributes.

e. Constructors.

Constructors are class methods that are automatically run when an object of a specific type is created. Constructors often initialize the new object's data members. When a class is formed, the constructor can only be called once. Furthermore, the constructor code is always executed before any other code in a class.

f. Access modifiers and access levels

All classes and class members can specify what access level they provide to other classes by using access modifiers. The following access modifiers are available:

Visual Basic Modifier	Definition
Public	The type or member can be accessed by any other code in the same assembly or another assembly that references it.
Private	The type or member can only be accessed by code in the same class.
Protected	The type or member can only be accessed by code in the same class or in a derived class.

Table 1. Access modifiers and access levels

Example:

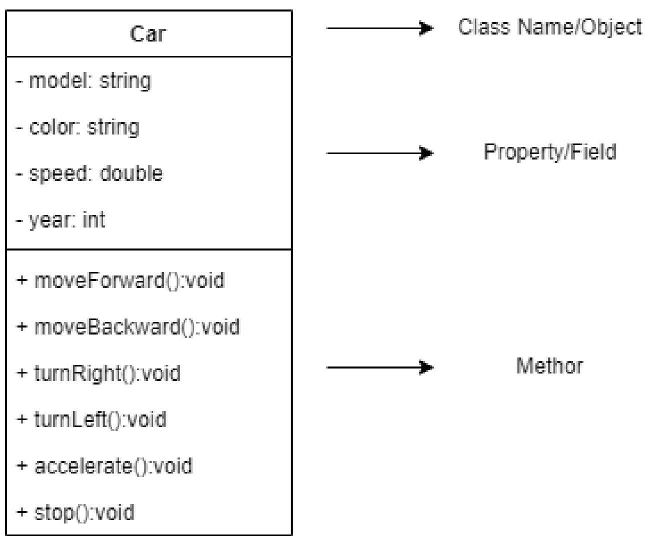


Figure 1. General concepts of OOP

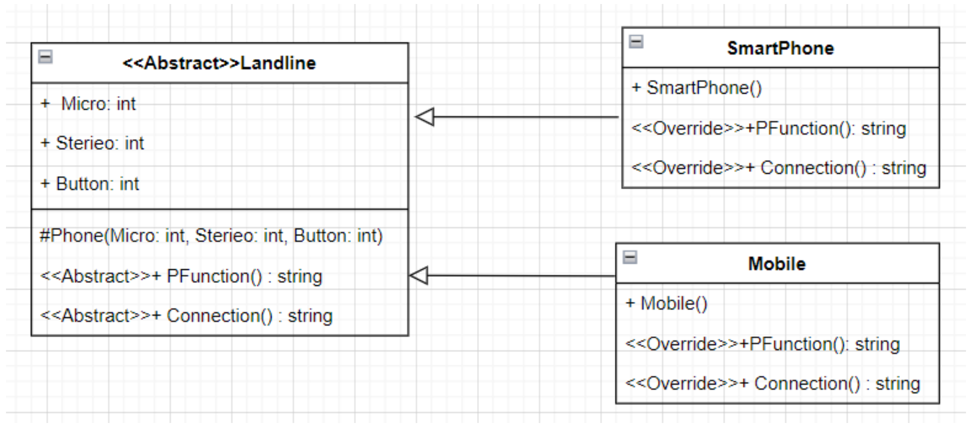
3. APIE concept in OOP.

a. Abstraction

The abstraction indicates that the object being edited has either a missing or incomplete implementation. The abstraction supports classes, methods, attributes, indexers, and events. Use the abstraction in the class definition to indicate that a class is only intended to act as the base class for other classes and cannot be instantiated independently. Non-abstract classes that descended from the abstract class must implement the abstract members.

Example:

In the past, phones were born to serve us in connecting to talk over long distances and faster than sending mail. Therefore, the desk phone was born to meet the needs of making calls. but desk phones are not flexible when moving. Therefore, mobile phones were born to serve the needs of users to be compact and portable, but mobile phones were too rudimentary and had very few functions. Later, when people's needs are increasing, smart phones have been born to serve people in addition to calling and mobile needs, it can also help people entertain and convenient.



<<Abstract>>Landline
+ Micro: int
+ Sterieo: int
+ Button: int
#Phone(Micro: int, Sterieo: int, Button: int)
<<Abstract>>+ PFunction() : string
<<Abstract>>+ Connection() : string
SmartPhone
+ SmartPhone()
<<Override>>+PFunction(): string
<<Override>>+ Connection() : string
Mobile
+ Mobile()
<<Override>>+PFunction(): string
<<Override>>+ Connection() : string

Figure 2. Abstraction

b. Polymorphism

Polymorphism is a Greek word, meaning "one name many forms". In other words, one object has many forms or has one name with multiple functionalities. "Poly" means many and "morph" means forms. Polymorphism provides the ability to a class to have multiple implementations with the same name. It is one of the core principles of Object-Oriented Programming after encapsulation and inheritance. In this article, you'll learn what polymorphism is, how it works, and how to implement polymorphism in C#.

Example:

In fact, we can see a lot of polymorphic things. An example is blood group. We have cold blood group and warm blood group. The cold blood group is often found in reptiles, specifically snakes. and the specific warm-blooded type is human.

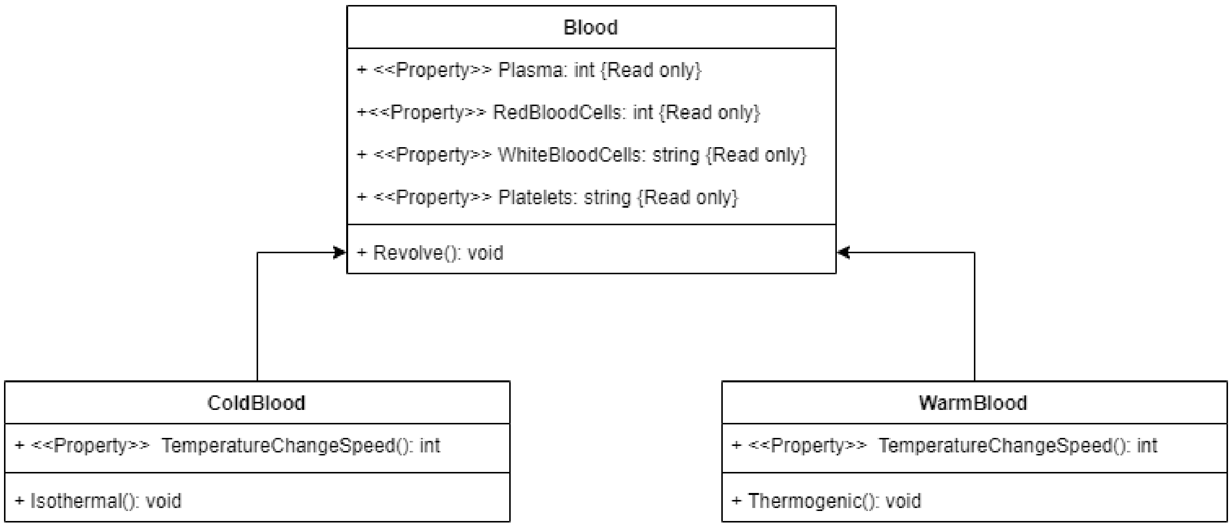

Blood
+ <<Property>> Plasma: int {Read only}
+<<Property>> RedBloodCells: int {Read only}
+ <<Property>> WhiteBloodCells: string {Read only}
+ <<Property>> Platelets: string {Read only}
+ Revolve(): void
ColdBlood
+ <<Property>> TemperatureChangeSpeed(): int
+ Isothermal(): void
WarmBlood
+ <<Property>> TemperatureChangeSpeed(): int
+ Thermogenic(): void

Figure 3. Polymorphism

c. Inheritance

The ability of one thing to acquire some or all of the properties of another object is known as inheritance. For instance, a child acquires the characteristics of his or her parents. Reusability is a major benefit of inheritance. The current class's fields and methods can be used again. Inheritance is not only limited to just one subclass that inherits a superclass, but also many subclasses that inherit properties from a superclass. However, a subclass cannot inherit from multiple super classes because in case both super classes have the same properties, it will cause an error. So, except for C++ programming language, other programming languages do not support multiple inheritance.

Example: Since Apple is a fruit, let's imagine that there is a class called Fruit and an Apple subclass within it. The characteristics of the Fruit class are added to our Apple. Other categories can include grape, pear, mango, etc. Fruit refers to a group of foods that are fleshy, ripe plant ovaries that may or may not contain a big seed. The subclass of Apple, which differs from other subclasses of Fruit by being red, spherical, and having a depression at the top, derives these characteristics from Fruit.

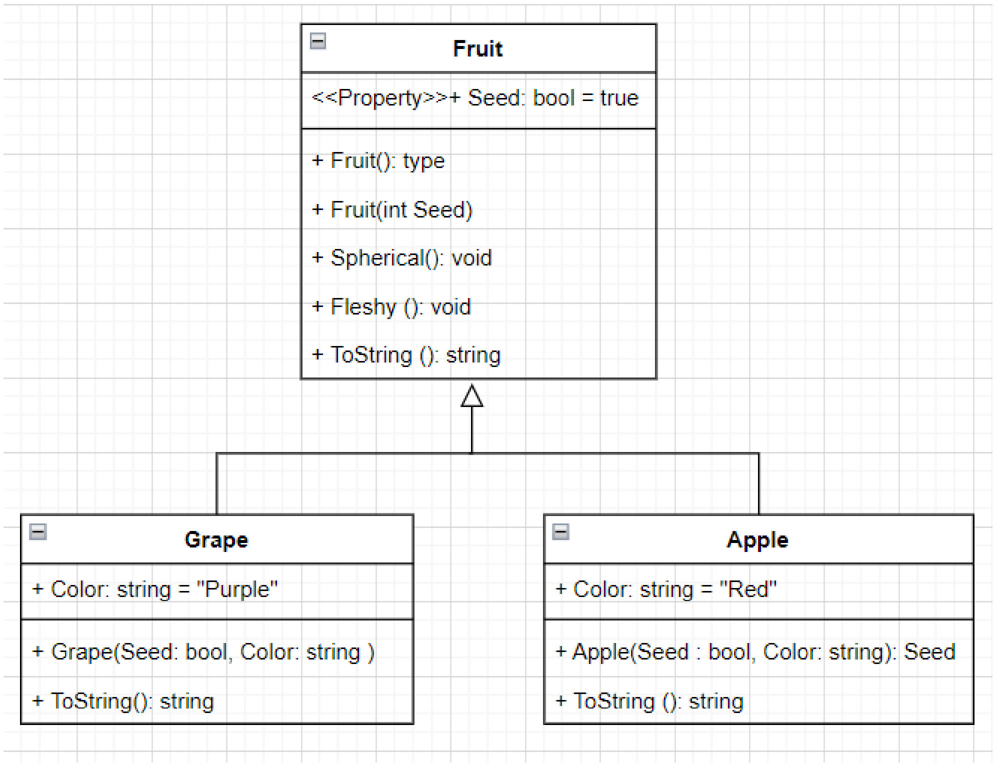


Figure 4. Inheritance

d. Encapsulation

The encapsulation is the process of grouping or wrapping up of data and functions to perform actions on the data into the single unit. The single unit is called a class. Encapsulation is like enclosing in a capsule. That is enclosing the related operations and data related to an object into that object. It keeps the data and the code safe from external interference. The main purpose or use of the encapsulation is to provide the security to the data of a class.

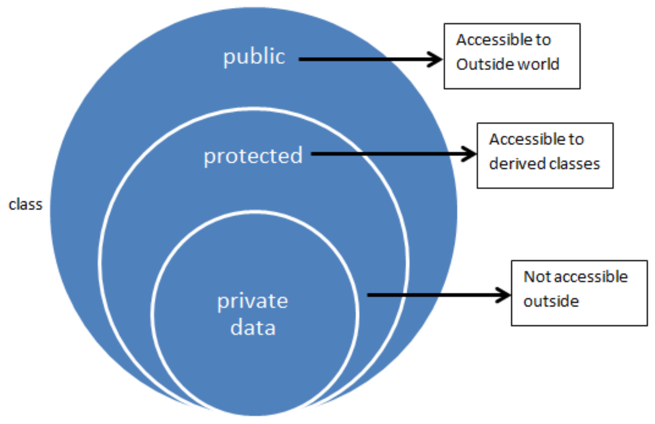
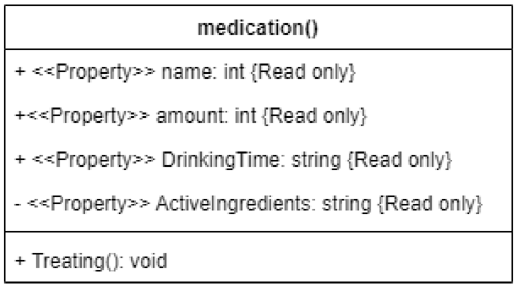


Figure 5. Access level

Example: We visit the hospital for a checkup when we are ill. After examination the doctor said we had a cold and handed us a medicine, which included a list of medications along with instructions on how much to take of each and when to take it. Reading medicine, we only know that there are those drugs and the dose of each drug, how to take it at what time. If we take the right medicine, the right dose at the right time, we will get rid of the flu, and specifically what active ingredients are inside each drug, why do we have to take such a dose, why should we take it. We also do not know at all.



medication()
+ <<Property>> name: int {Read only}
+<<Property>> amount: int {Read only}
+ <<Property>> DrinkingTime: string {Read only}
- <<Property>> ActiveIngredients: string {Read only}
+ Treating(): void

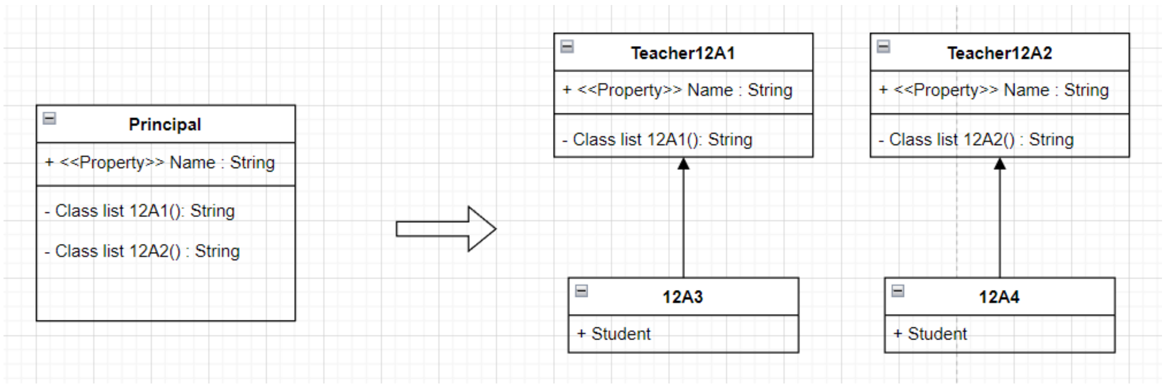
Figure 6. Encapsulation

4. SOLID concept in OOP

a. Single responsibility principle

Each software module or class should have only one reason to change. we can say that each module or class should have only one responsibility to do. So, we need to design the software in such a way that everything in a class or module has to do with a single responsibility. Clear SRP will make the layers smaller, cleaner and therefore easier to maintain.

Example: At school, if all students are only managed by the principal, it will lead to overcrowding and uncontrollability for students. So, students will be divided into different classes and only one homeroom teacher will manage their students. This helps to better manage students.



Principal
+ <<Property>> Name : String
- Class list 12A1(): String
- Class list 12A2() : String
Teacher12A1
+ <<Property>> Name : String
- Class list 12A1(): String
Teacher12A2
+ <<Property>> Name : String
- Class list 12A2() : String
12A3
+ Student
12A4
+ Student

Figure 7. Single responsibility principle

b. Open/closed principle

The Open Closure Principle (OCP) is the SOLID principle that states that software entities should be open for expansion but closed for modification. We have to create software modules and classes that can easily accommodate new responsibilities and functions as they become necessary. This is called extensibility for extensions. On the contrary, closed for modification indicates that we should not make any changes to the class or module unless an error is detected.

Example: In a dedicated camera, a lens is mounted to the camera to record landscapes. As customers request more portrait features, we openly expand the camera set by adding other lenses, such as those made expressly for portraiture, rather than removing the present lens to address the problem.

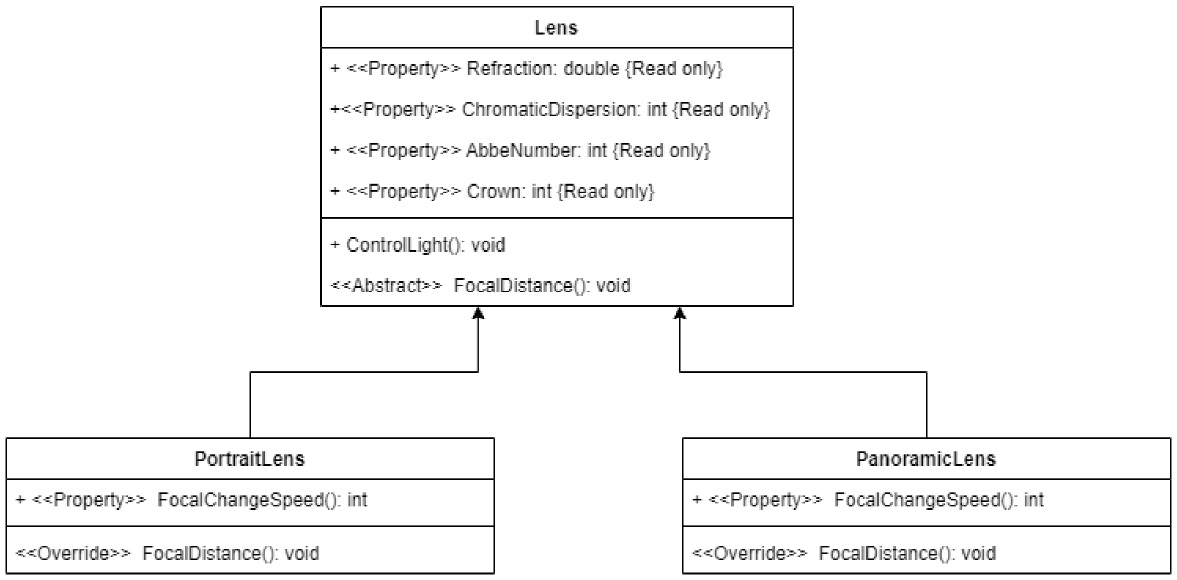

Lens
+ <<Property>> Refraction: double {Read only}
+<<Property>> ChromaticDispersion: int {Read only}
+ <<Property>> AbbeNumber: int {Read only}
+ <<Property>> Crown: int {Read only}
+ ControlLight(): void
<<Abstract>> FocalDistance(): void
PortraitLens
+ <<Property>> FocalChangeSpeed(): int
<<Override>> FocalDistance(): void
PanoramicLens
+ <<Property>> FocalChangeSpeed(): int
<<Override>> FocalDistance(): void

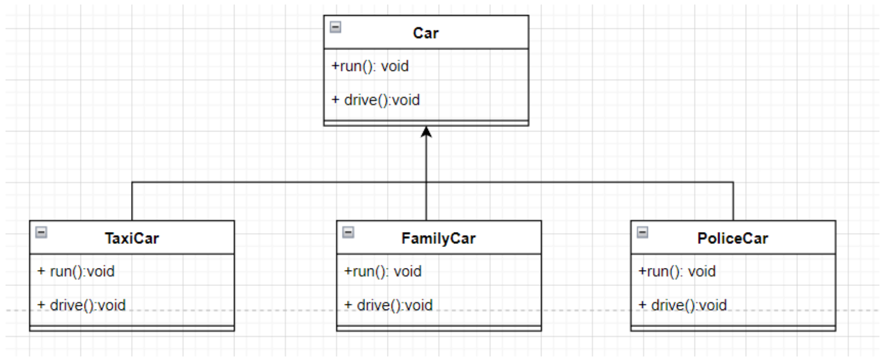
Figure 8. Open/closed principle

c. Liskov substitution principle

When we have inheritance relationship between base class and subclass, then objects of subclass can replace objects of superclass without affecting the integrity of application. Basically, this means that we should strive to create such derived class objects so that we can replace objects of the base class without modifying its behavior. then it is said in the Liskov Substitution Principle.

Example: From a car, we can turn it into a taxi or a police car, a family car. but ordinary people will not be able to drive but only police can drive.

Violation of Liskov substitution principle:



Not violation of Liskov substitution principle:

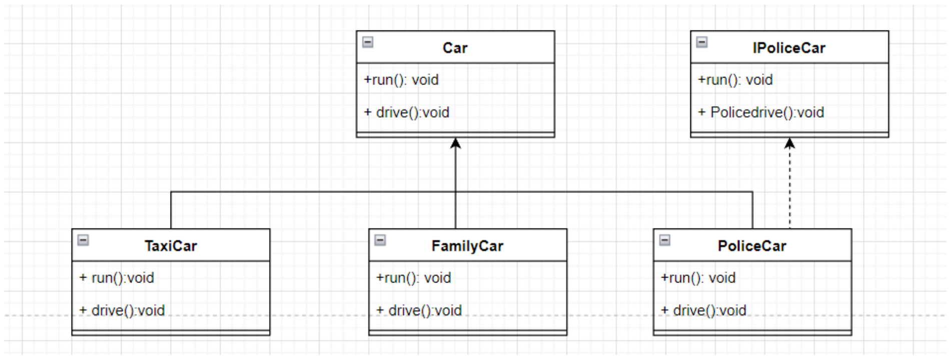


Figure 9. Liskov substitution principle

d. Interface segregation principle

Each interface should have a specific purpose. You shouldn't be forced to implement an interface when your object doesn't share that purpose. By extrapolation, the larger the interface, the more likely it includes methods that not all implementers can achieve. That's the essence of the Interface Segregation Principle

Example: In a vegetable market, the distributor divides the stalls for each type of vegetable they sell. 2 types of vegetables are not allowed to be sold in the same store

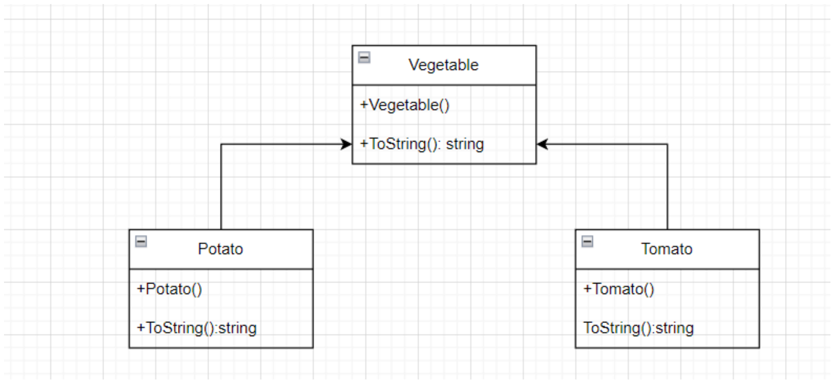


Figure 10. Interface segregation principle

e. Dependency inversion principle

The Dependency Inversion Principle (DIP) states that high-level modules/classes should not depend on low-level modules/classes. Both should depend upon abstractions. Secondly, abstractions should not depend upon details. Details should depend upon abstractions.

Example: In a family, we have a lot of brothers, we can consider it a low-level module. high-level modules are their parents

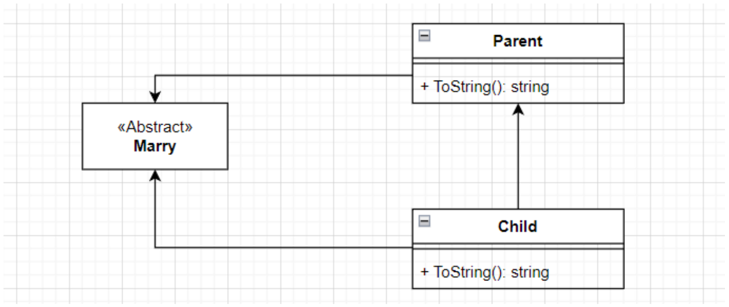


Figure 11. Dependency inversion principle

III. OOP scenario

1. Scenario

FitnessAlpha is an old gym that has just been renovated after being acquired by John. John invited Paul a fitness trainer to work at the gym with the aim of implementing a private gym management model that only accepts a certain number of members and focuses on training them.

As the main trainer of the gym, Paul will be in charge of the training process of the members. From adjusting the workout schedule, to the member's diet plan and taking responsibility during the workout. Paul will have to monitor and guide the members carefully to get the optimal training results for the members.

Members who register for the training course will have to adhere to the plan that the coach has set for them along with the members will be able to use the equipment in the gym for training purposes. During the training process, the members will be supported by the coach so that they can exercise properly and advise on an effective diet plan. During the training course, students will be tested for health to get information about the results of their training and will be adjusted to a training plan stating that the previous plan did not achieve the desired effect.

This is a self-sufficient management process of two people John and Paul when they discuss and divide the work among themselves.

- In order to facilitate the management process, personal information of trainer and members will be collected. With this, coaches and members can conveniently exchange training plans with each other.
- When members register for a training course, the coach will provide a suitable exercise plan for each member (depending on the member's height, weight, and physical condition).
- Trainers and members both receive course information such as course validity period, course schedule, course cost, trainer information.
- The members will have to follow the exercise and diet plan that the coach has set. During practice, students can ask for support from the coach.
- Coaches and members are free to use equipment in the gym for training purposes.
- After a period of time, the members will report to the coach about their body condition so that the trainer can understand the results of the training process.
- The trainer is responsible for adjusting the members' training plans to match the training results reported by the members.

2. Use case Diagram

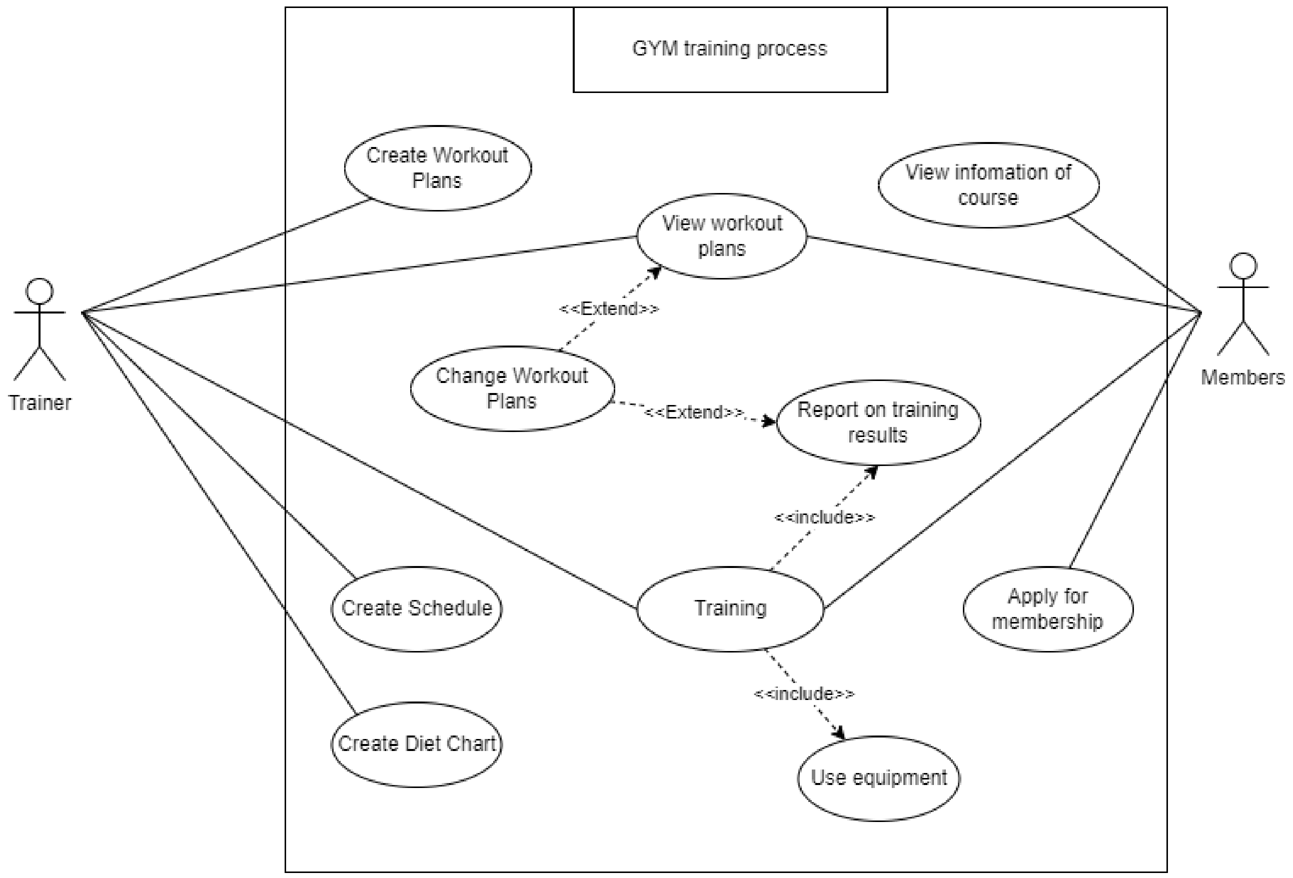


Figure 12. Use case Diagram

Explain gym manager system:

Name of Use Case	**Create Schedule**		
Created By	Group 3	Last Update By	Nguyen Quang Truong
Date Created	26/11	Last Revision Date	
Description	Trainer can create and schedule courses for them		
Actors	Trainer		
Preconditions	The course must have at least one customer		
Postconditions			
Flow	Register for the course See information about the training time		
Alternative Flows			
Exceptions	Do not register for the course		
Requirements	Paid for the course with Gym owner		

Name of Use Case	**Create Diet Chart**		
Created By	Group 3	Last Update By	Nguyen Quang Truong
Date Created	26/11	Last Revision Date	
Description	Trainers can create science-based diet charts		
Actors	Trainer		
Preconditions	Attended the course		

Postconditions	
Flow	Attended the coach's course Eat according to the given diet chart
Alternative Flows	
Exceptions	Students do not follow the diet chart
Requirements	Students are required to follow the diet chart

Name of Use Case	**Training**		
Created By	Group 3	Last Update By	Nguyen Quang Truong
Date Created	26/11	Last Revision Date	
Description	The trainer will guide his students to practice the exercises in each course. Give feedback and report back on your training results		
Actors	Trainer		
Preconditions	The trainer will not be allowed to be absent		
Postconditions			
Flow	After choosing the course, the trainer begins to guide the students to practice Then record and report the results of the exercise		
Alternative Flows			
Exceptions			
Requirements	Students practice hard, and eat according to the given chart		

Name of Use Case	**View Workout Plan**		
Created By	Group 3	Last Update By	Nguyen Quang Truong
Date Created	26/11	Last Revision Date	
Description	The trainer will observe the student's practice and can change the exercise if it is not suitable		
Actors	Trainer		
Preconditions	The trainer must be meticulous in observing and evaluating		
Postconditions			
Flow	Students need to practice for the trainer to make an assessment		
Alternative Flows			
Exceptions			
Requirements	Trainers must have techniques and focus on observing and evaluating their students properly		

Name of Use Case	**Create Workout Plans**		
Created By	Group 3	Last Update By	Nguyen Quang Truong
Date Created	26/11	Last Revision Date	
Description	The trainer creates a workout plan to achieve the desired member results		
Actors	Trainer		
Preconditions	Technical and scientific elements are a must-have in a trainer		

Postconditions	
Flow	Apply the knowledge of the trainer to give appropriate lessons
Alternative Flows	
Exceptions	Inappropriate workout plan
Requirements	Ask students to practice according to the lesson that has been given

Name of Use Case	**Members's Training Results**		
Created By	Group 3	Last Update By	Nguyen Quang Truong
Date Created	26/11	Last Revision Date	
Description	After a period of practice, the student's change has met the initial requirements, if it is not effective, it is necessary to consider and change the exercises to be more appropriate.		
Actors	Trainer		
Preconditions	Students always follow and practice according to the exercises given by the trainer		
Postconditions			
Flow	After the training period, it is necessary to give the results and evaluate the status.		
Alternative Flows			
Exceptions	Member does not follow the exercises		
Requirements	If the exercise doesn't work, the trainer should give you a more suitable exercise		

Name of Use Case	**View Information of Course**		
Created By	Group 3	Last Update By	Nguyen Quang Truong
Date Created	26/11	Last Revision Date	
Description	Allows users to view the gym's courses so that they can choose a course for themselves		
Actors	Member		
Preconditions	Have learned about gym courses		
Postconditions			
Flow	Refer to the training courses Choose the right course		
Alternative Flows			
Exceptions	I have researched very carefully about the gym and its courses before		
Requirements			

Name of Use Case	**Apply for Membership**		
Created By	Group 3	Last Update By	Nguyen Quang Truong
Date Created	26/11	Last Revision Date	
Description	If you become a new member of the gym, you will need to pay fees.		
Actors	Member		
Preconditions	Must have money and excitement		
Postconditions			
Flow	Register new member		

	Pay the fees for the courses
Alternative Flows	
Exceptions	
Requirements	Pay the fees and train hard

3. Class Diagram

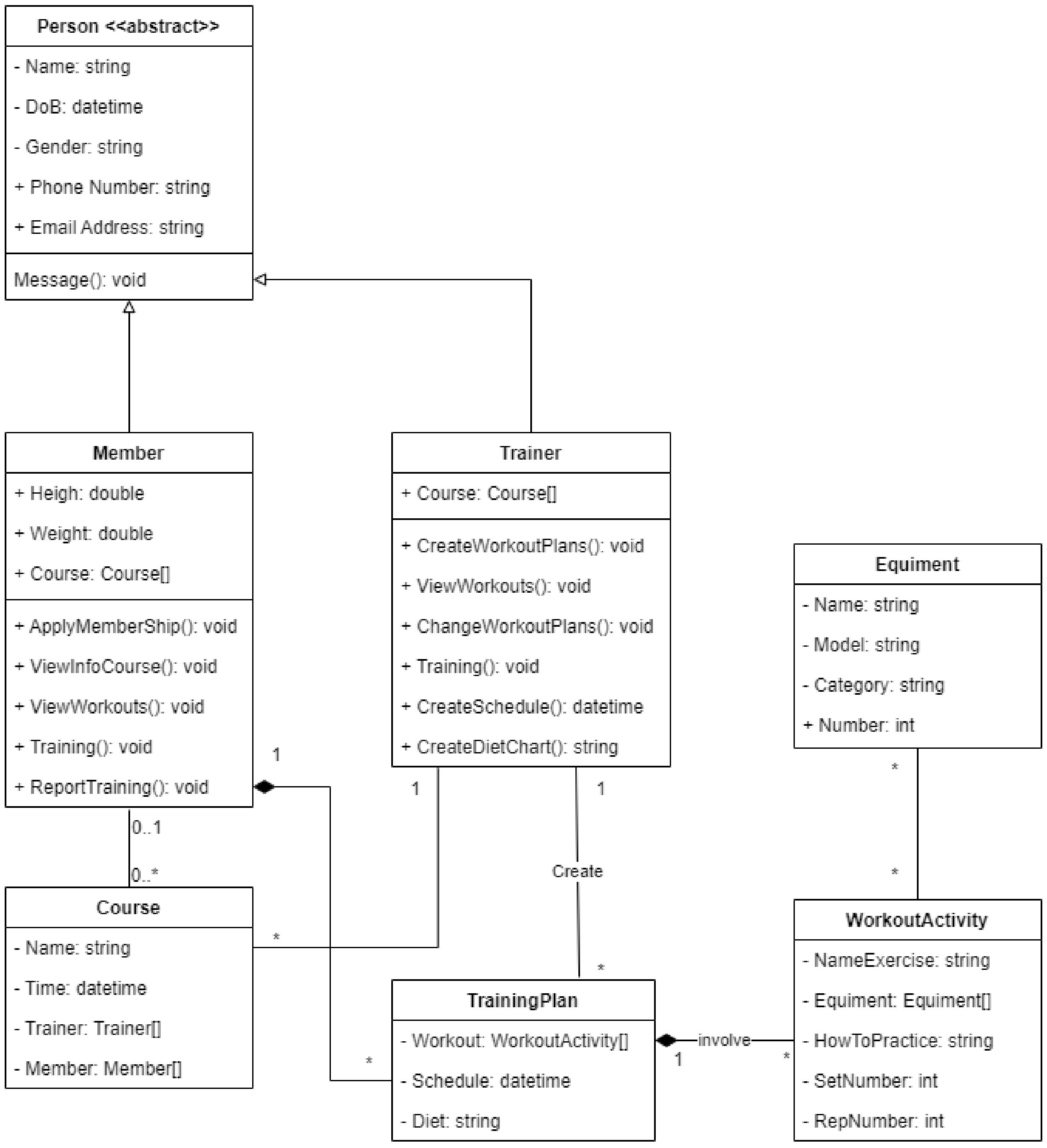


Figure 13. Class Diagram 1

Explain:

The first is the Person class, this will be the class that stores the user's information including name, gender, episode time, phone number and email. this will be the SuperClass of the Member class and the Trainer class.

Coming to the Member class, the purpose of this class is to store the user's information and perform some exclusive functions of this class. In addition to storing information like SuperClass, it also stores the user's height, weight, and class information. In class Members will be able to perform a number of functions such as registering for classes, viewing class information, training plans, performing workouts and reporting training status.

Next is the Trainer class that can create classes and timetables for people to sign up for. The trainer will have to come up with a workout and eating plan for the class members. The trainer will also be the one to guide everyone to practice as well as change the exercises that are not suitable for class members.

Next is the Course class, this class will store the information of class members, training time, trainer.

TrainingPlan class will save exercises, timetables, and information about suitable diets for members.

WorkoutActivity class makes exercises and equipment suitable for everyone.

Finally, the Equipment class will perform the function of managing training equipment as well as the number and type of equipment.

IV. Design Patterns

Design patterns are used to represent some of the best practices adapted by experienced object-oriented software developers. A design pattern systematically names, motivates, and explains a general design that addresses a recurring design problem in object-oriented systems. It describes the problem, the solution, when to apply the solution, and its consequences. It also gives implementation hints and examples.

Types of Design Patterns:

- **Creational Patterns:** These design patterns provide a way to create objects while hiding the creation logic, rather than instantiating objects directly using new operator. This gives program more flexibility in deciding which objects need to be created for a given use case.

- **Structural Patterns:** These design patterns concern class and object composition. Concept of inheritance is used to compose interfaces and define ways to compose objects to obtain new functionalities.
- **Behavioral Patterns:** These design patterns are specifically concerned with communication between objects.

1. Creational pattern

a. General definition

Creational Pattern (initial group - 5 templates) including: Factory Method, Abstract Factory, Builder, Prototype, Singleton. Design patterns of this type provide a solution for creating objects and hiding the logic of its creation, rather than creating the object directly using the new method. This makes the program more flexible in deciding which objects need to be created in given situations (Erich Gamma, 2014).

b. Factory Method:

Factory Method Pattern defines an interface for creating an object, but let's subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

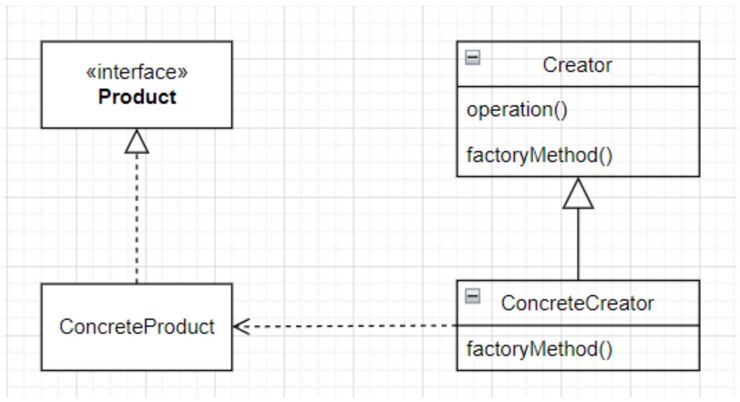


Figure 14. Factory Method

Applicability:

Use the Factory Method pattern when

- A class can’t anticipate the class of objects it must create.
- A class wants its subclasses to specify the objects it creates.
- Classes delegate responsibility to one of several helper subclasses, and you want to localize the knowledge of which helper subclass is the delegate

Pros	Cons
+ Allows you to hide implementation of an application seam (the core interfaces that make up your application) + Allows you to easily test the seam of an application (that is to mock/stub) certain parts of your application so you can build and test the other parts + Allows you to change the design of your application more readily, this is known as loose coupling	+ Makes code more difficult to read as all of your code is behind an abstraction that may in turn hide abstractions. + Can be classed as an anti-pattern when it is incorrectly used, for example some people use it to wire up a whole application when using an IOC container, instead use Dependency Injection.

c. Abstract Factory:

Abstract Factory Pattern provides an interface for creating families of related or dependent objects without specifying their concrete classes.

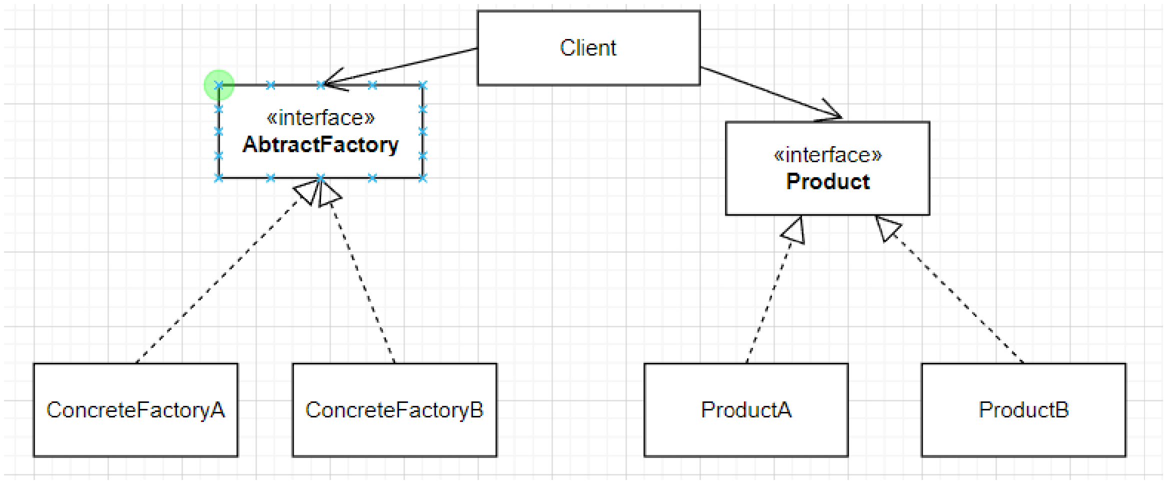


Figure 15. Abstract Factory

Applicability:

Use the Abstract Factory pattern when

- A system should be independent of how its products are created, composed, and represented.
- A system should be configured with one of multiple families of products.
- A family of related product objects is designed to be used together, and you need to enforce this constraint.
- You want to provide a class library of products, and you want to reveal just their interfaces, not their implementations.

Pros	Cons
+ Products are compatible with each other, because they all use the same interface. + You avoid tight coupling between the concrete objects and client code. + Single Responsibility Principle: you put the object creation all into one class. Hence making it easier to maintain. + Open Closed Principle: to add new variants you don't have to change existing code.	We added so many classes and interfaces, making the code more complicated.

d. Builder:

Builder Pattern to encapsulate the construction of a product and allow it to be constructed in steps.

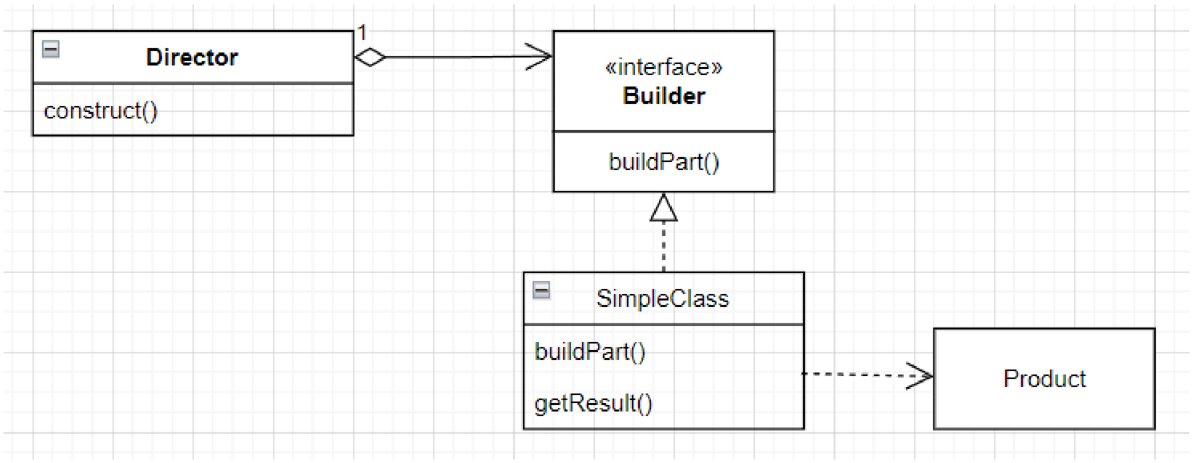


Figure 16. Builder

Applicability:

Use the Builder pattern when

- The algorithm for creating a complex object should be independent of the parts that make up the object and how they're assembled.
- The construction process must allow different representations for the object that's constructed

Pros	Cons
+ Encapsulates the code for object construction from the client. provides great control over the construction process of the object. + Creating complex objects in an easy and neat way. + Gets rid of a chain of constructors to build complex objects. Also, we need a constructor only for mandatory arguments making it less complex. provides great flexibility and readability, though it increases the number of lines (fortunately that also can be reduced using the Lombok library). + Allows us to build immutable objects with little complex logic.	+ Increased number of lines. + Not all the data members are guaranteed to be initialized. + Challenging to support dependency injection.

e. Prototype:

Prototype Pattern when creating an instance of a given class is either expensive or complicated.

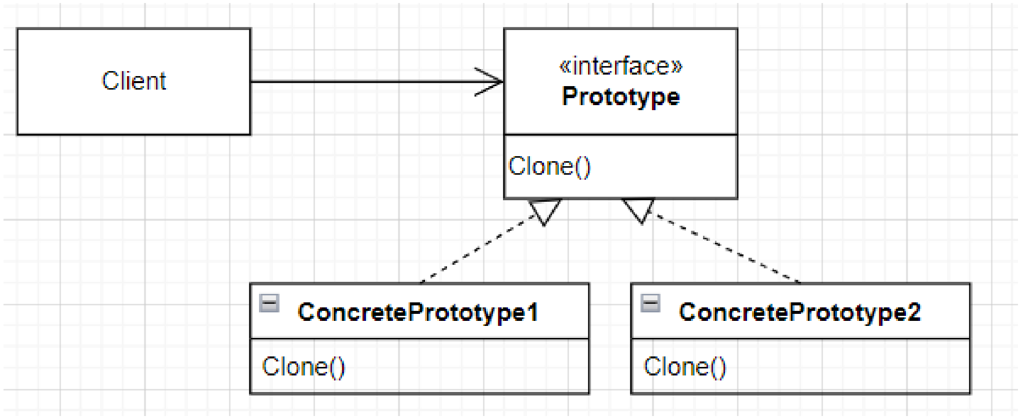


Figure 17. Prototype

Applicability:

Use the Prototype pattern when a system should be independent of how its products are created, composed, and represented.

- When the classes to instantiate are specified at run-time, for example, by dynamic loading
- To avoid building a class hierarchy of factories that parallels the class hierarchy of products

- When instances of a class can have one of only a few different combinations of state. It may be more convenient to install a corresponding number of prototypes and clone them rather than instantiating the class manually, each time with the appropriate state.

Pros	Cons
+ Cloned objects can be duplicated independently of their concrete classes. + Complex objects can be produced more easily.	Circularly referenced complicated items can be exceedingly difficult to duplicate.

f. Singleton:

Singleton Pattern ensures a class has only one instance and provides a global point of access to it.

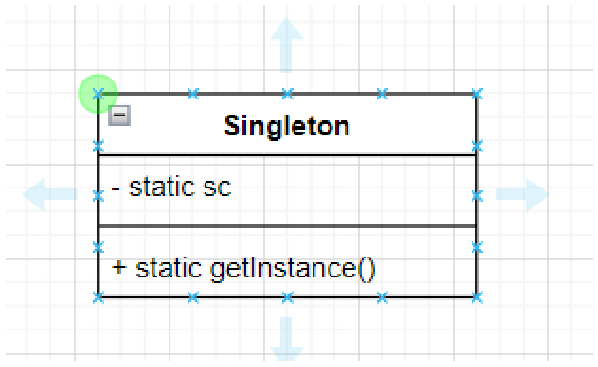


Figure 18. Singleton

Applicability:

- There must be exactly one instance of a class, and it must be accessible to clients from a well-known access point.
- When the sole instance should be extensible by sub-classing, and clients should be able to use an extended instance without modifying their code.

Pros	Cons
There is only one instance of a class, you can be certain of that. The instance becomes accessible from anywhere in the world.	In a multi - threading context, the pattern needs particular handling to prevent several threads from repeatedly creating a singleton object. When program components know too much about one another for instance, the Singleton pattern can hide poor design.

g. Scenario

GreenBus company has an application about designing buses for the city, initially the application was born only to create diesel-powered buses. Recently, the City People's Committee is developing a plan to use green energy sources, which many customers know and there is a passenger transport company that proposes to cooperate, with the vehicle being an Electric Bus. But with the original design of the application only serving diesel-powered Buses, the program's transmission line is closely tied to the vehicle it is. Therefore, to serve new customers you need to expand the chain to be able to accommodate them. The newly created diesel and electric buses will have the same components, that is the number of fan systems, chassis, speakers, and seats. However, the difference will be in the engine part of the Bus.

- In diesel-powered buses, the engine will be a traditional internal combustion engine and the fuel for them to operate is oil.
- In electric buses, the engine will be a rechargeable battery system.

h. Diagram:

With the requirements stated and after considering the suitability, we will choose the Factor Methor pattern to build a program for this problem.

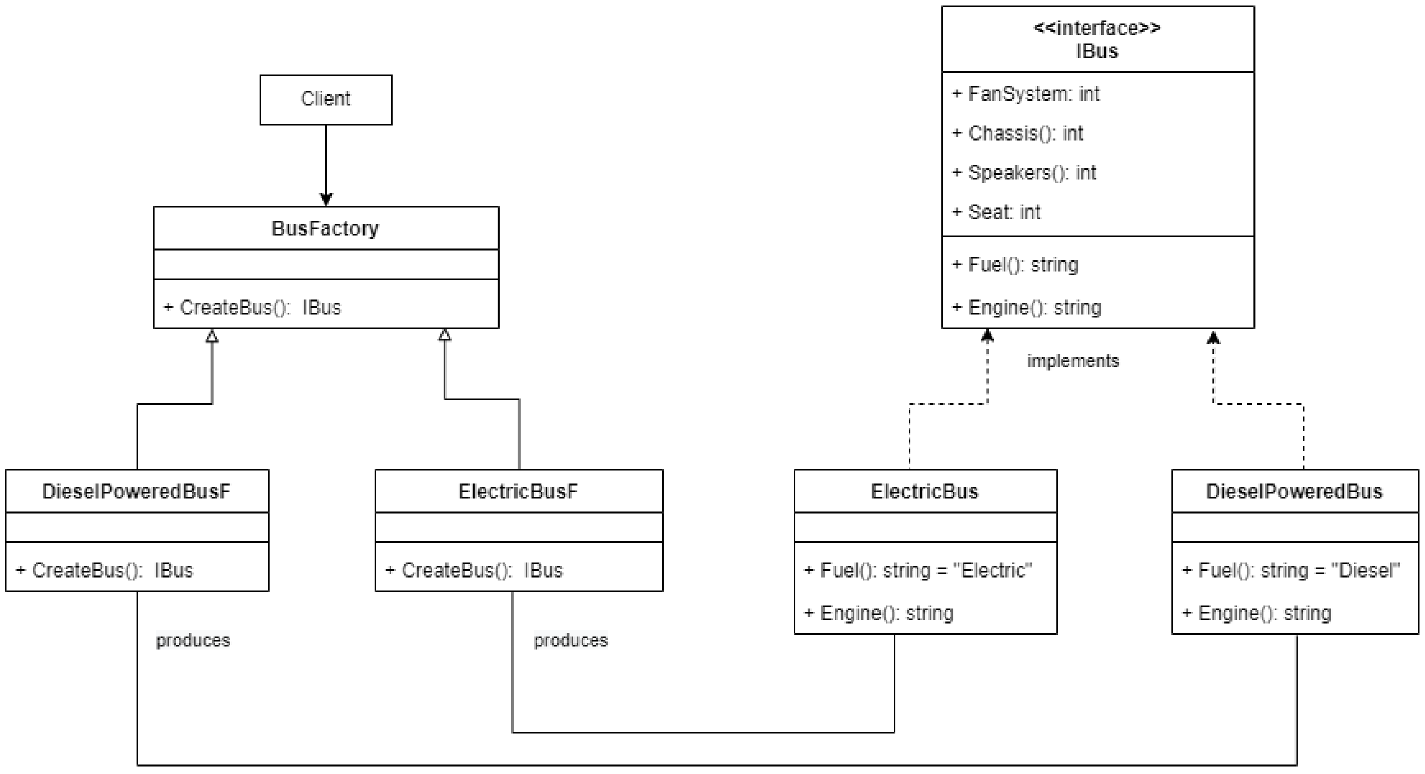


Figure 19. Class diagram 2

Explain:

- The IBus declares the interface, which is common to all objects that can be produced and its subclasses.
- ElectricBus and DieselPoweredBus are different implementations of the product interface.
- The BusFactory class declares the factory method that returns new product objects. The return type of this method matches the product interface.
- ElectricBusF and DieselPoweredBusF override base factory method and returns a different type of bus.

2. Structural pattern

a. General definition

Structural patterns are concerned with how classes and objects are composed to form larger structures. Structural class patterns use inheritance to compose interfaces or implementations.

b. Adapter pattern:

Adapter pattern is about creating an intermediary abstraction that translates, or maps, the old component to the new system. Clients call methods on the Adapter object which redirects them into calls to the legacy component. This strategy can be implemented either with inheritance or with aggregation. Adapter functions as a wrapper or modifier of an existing class. It provides a different or translated view of that class.

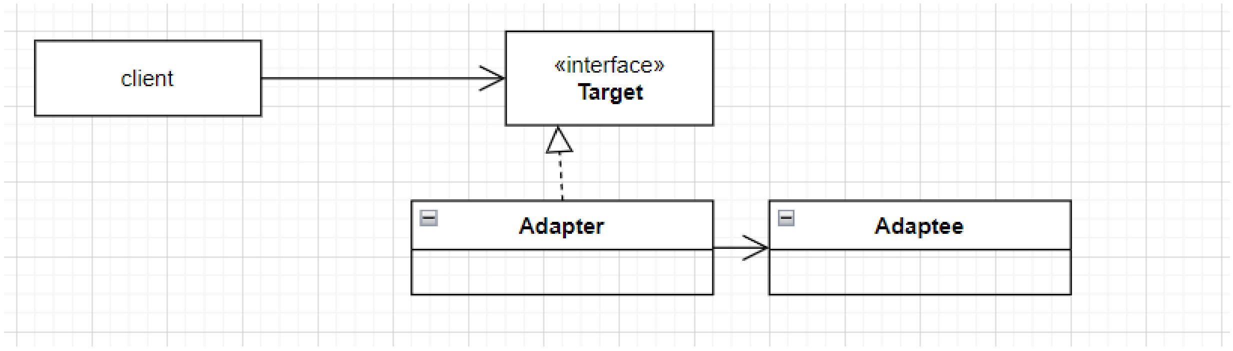


Figure 20. Adapter pattern

Applicability: The Adapter pattern should be used when:

- There is an existing class, and its interface does not match the one you need.
- You want to create a reusable class that cooperates with unrelated or unforeseen classes
- There are several existing subclasses to be use, but it’s impractical to adapt their interface by subclassing everyone

Pros	Cons
Adapter can add functionality to many Adaptees. CustomerAdapter can be more abstract and adapter more than just customer object.	Harder to override Adaptee behavior. Customer object behavior can't be changed without subclassing it.

c. Bridge pattern:

Bridge pattern is a structural design pattern that lets you split a large class or a set of closely related classes into two separate hierarchies—abstraction and implementation—which can be developed independently of each other.

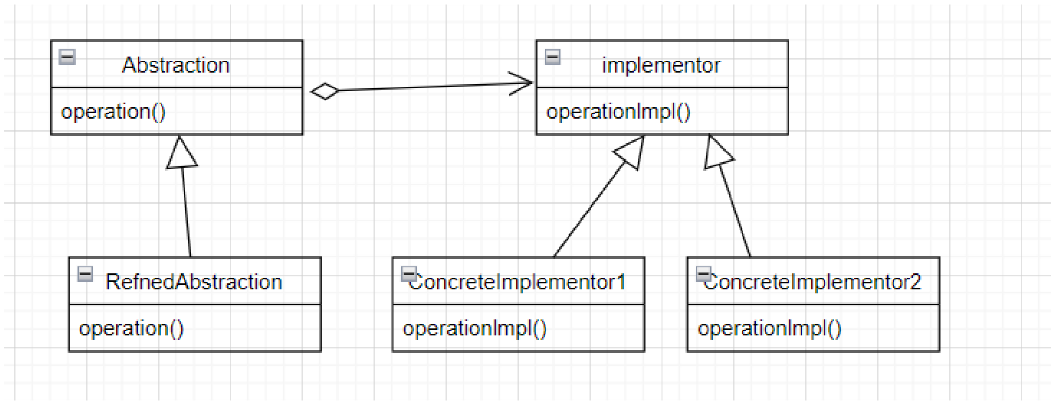


Figure 21. Bridge pattern

Applicability:

You should use the Bridge Pattern when:

- You want to avoid a permanent binding between an abstraction and its implementation
- Both the abstractions and their implementations should be extensible by sub-classing.
- You want to share an implementation among multiple objects (perhaps using reference counting), and this fact should be hidden from the client.

Pros	Cons
+ You can develop classes and applications that are platform-independent. + The client code uses high-level abstractions to function. It is not made aware of the platform specifics.	Applying the pattern to a class with a high degree of cohesiveness could result in the code being more complex.

d. Composite pattern:

Composite pattern allows you to compose objects into tree structures to represent part whole hierarchies. Composite lets clients to treat individual objects and compositions of objects uniformly.

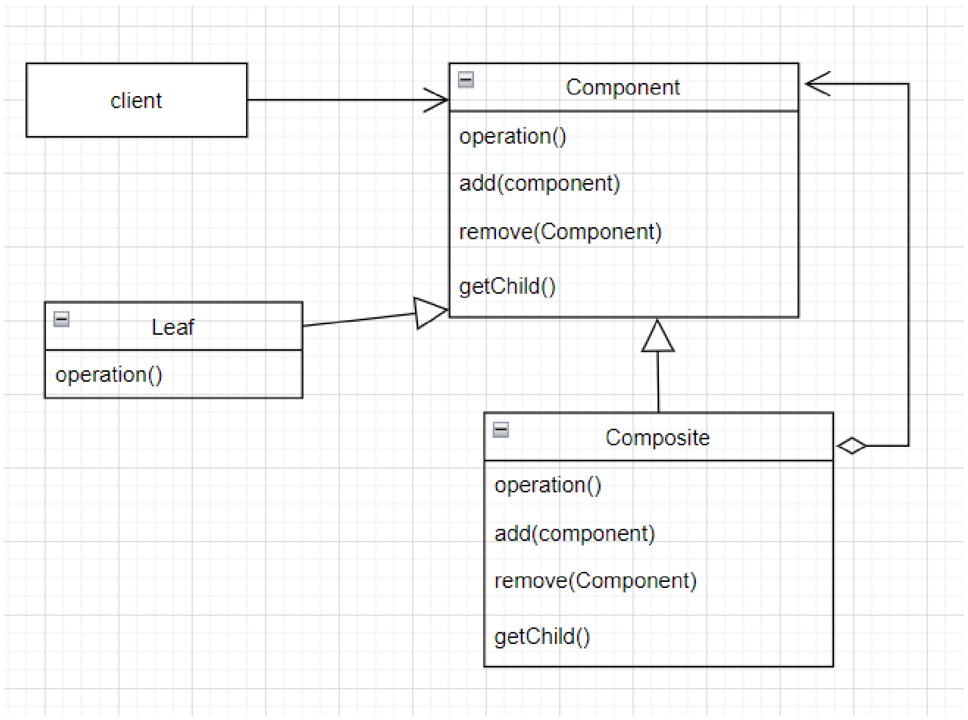


Figure 22. Composite pattern

Applicability:

- The composite pattern should be used when:
- When you want to represent part-whole hierarchies of objects.
- When you want clients to be able to ignore the difference between compositions of objects and individual objects. Clients will treat all objects in the composite structure uniformly

Pros	**Cons**
You can work with complex tree structures more easily - use polymorphism and recursion to your advantage.	Offering a common interface for classes with too many functional differences could be challenging. In other circumstances, you would have to overgeneralize the component interface, which would make it more difficult to understand.

e. Facade Pattern:

Facade Pattern makes a complex interface easier to use, using a Facade class. The Facade Pattern provides a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.

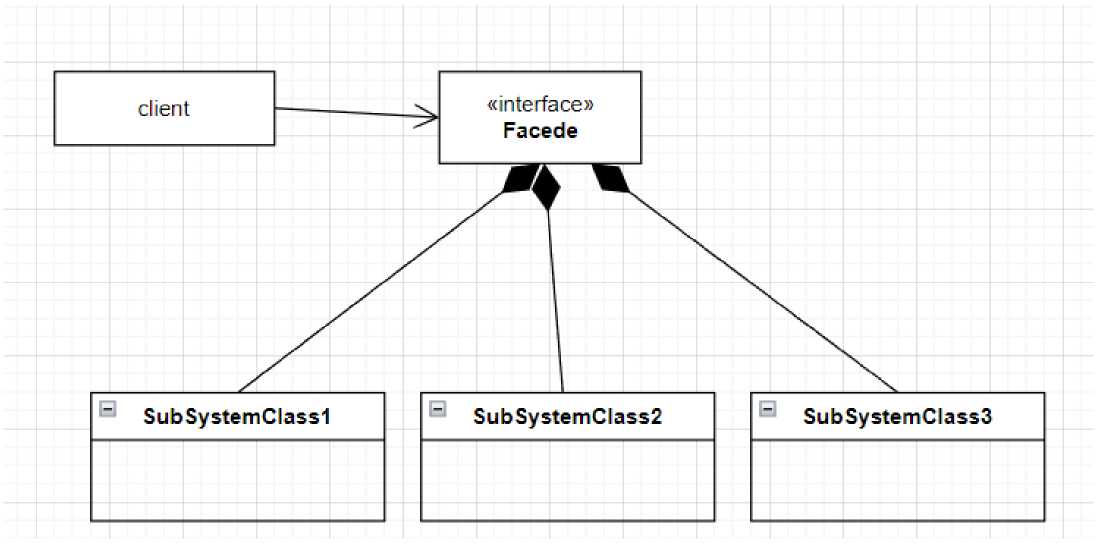


Figure 23. Facade Pattern

Applicability: Use the Facade Pattern, when:

- You want to provide a simple interface to a complex subsystem.
- There are many dependencies between clients and the implementation classes of an abstraction.
- You can layer your subsystems

Pros	Cons
You can isolate your code from the complexity of a subsystem.	A facade can become a god object coupled to all classes of an app

f. Flyweight Pattern:

Flyweight Pattern is designed to control such kind of object creation and provides you with a basic caching mechanism. It allows you to create one object per type (the type here differs by a property of that object), and if you ask for an object with the same property (already created), it will return you the same object instead of creating a new one.

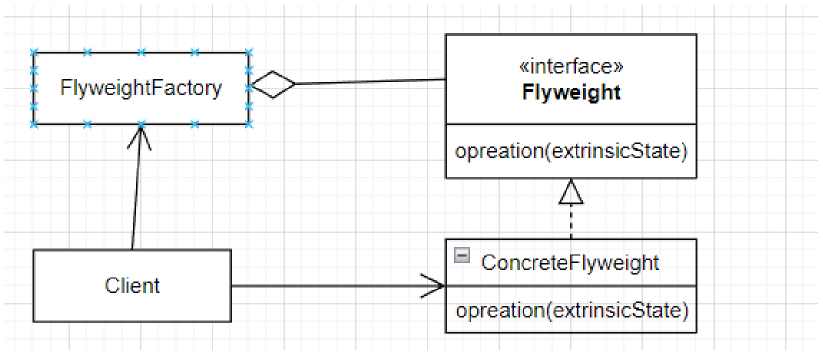


Figure 24. Flyweight Pattern

Applicability: Apply the Flyweight pattern when all of the following are true:

- An application uses a large number of objects.
- Storage costs are high because of the sheer quantity of objects.
- Most object state can be made extrinsic.
- Many groups of objects may be replaced by relatively few shared objects once extrinsic state is removed.
- The application doesn't depend on object identity. Since flyweight objects may be shared, identity tests will return true for conceptually distinct objects.

Pros	Cons
You can save a lot of RAM provided your program contains tons of similar objects.	You can swap RAM over CPU cycles if some of the context data needs to be recomputed every time someone calls a flyweight method.

g. Proxy Pattern:

Proxy Pattern is used to create a representative object that controls access to another object, which may be remote, expensive to create or in need of being secured.

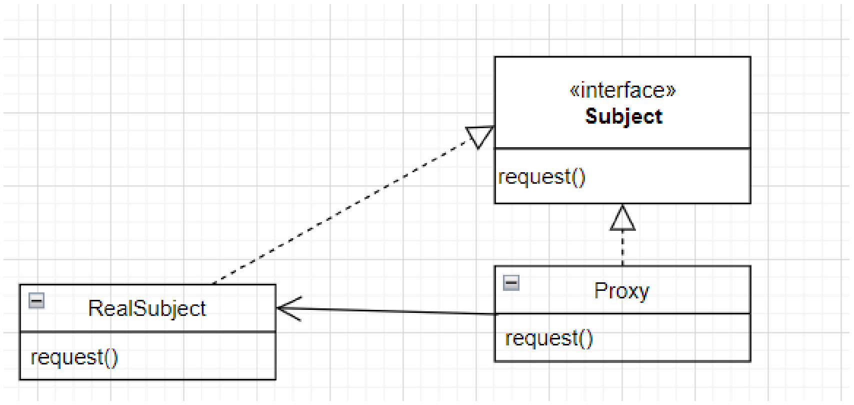


Figure 25. Proxy Pattern

Applicability:

- A remote proxy provides a local representative for an object in a different address space.
- A virtual proxy creates expensive objects on demand.
- A protection proxy controls access to the original object. Protection proxies are useful when objects should have different access rights.

Pros	Cons
+ You can control the service without any clients knowing about it. + You can control how long the service object lasts for when clients don't need it.	+ The code in this program can get a bit more complicated, because we need to introduce a lot of new classes. + The response from the service might get delayed

h. Scenario

DevKey is a start-up software design company. Currently, the company's human resources department is designing a salary calculation process for employees in the IT department. Because there are many roles and superiors require to divide the salary according to the employee's level, the employee's salary will be evaluated according to these two criteria.

- Employee information will be provided and specifically divided by roles: architect, developer, tester. Each role will have a base salary of $800, $700 and $650/month, respectively.
- Employees will be divided into 4 levels: leader, senior, junior, intern. Depending on the level, employees will have a corresponding salary coefficient from high to low, which is 2.2, 1.8, 1.4 and 0.5.
- The salary that employees receive will be considered according to their role, level and number of hours worked.

i. Diagram

With the requirements stated and after considering the suitability, we will choose the Bridge pattern to build a program for this problem

We will have 3 positions in the company: Architect, Developer and Tester respectively are 3 classes, each position will have the basic salary of each position passed in Class Position.

Respectively, there will be 4 classes of Leader, Senior, Junior and Intern, each position will have corresponding salary for each job level. These positions will all have salary contracts at Class LevelSalary.

The salary of the positions will be calculated in Class Salary.

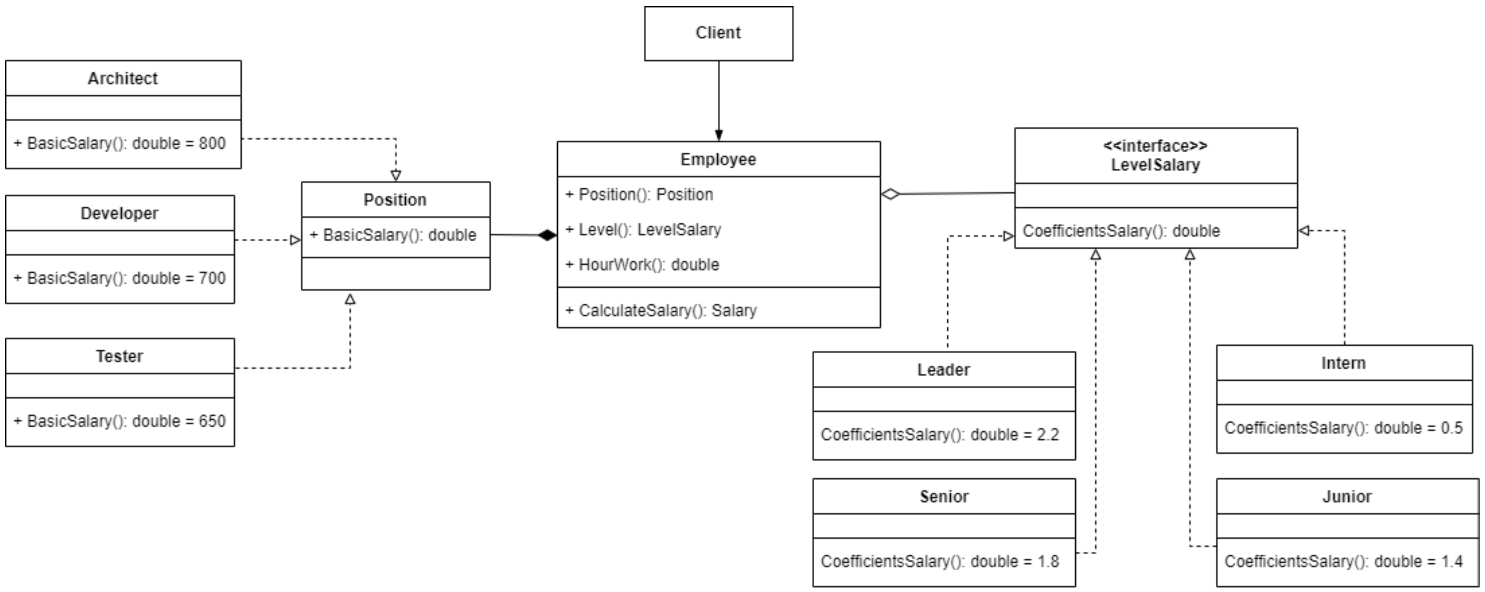

Client
Architect
+ BasicSalary(): double = 800
Developer
+ BasicSalary(): double = 700
Tester
+ BasicSalary(): double = 650
Position
+ BasicSalary(): double
Employee
+ Position(): Position
+ Level(): LevelSalary
+ HourWork(): double
+ CalculateSalary(): Salary
<<interface>>
LevelSalary
CoefficientsSalary(): double
Leader
CoefficientsSalary(): double = 2.2
Senior
CoefficientsSalary(): double = 1.8
Intern
CoefficientsSalary(): double = 0.5
Junior
CoefficientsSalary(): double = 1.4

Figure 26. Class diagram 3

3. Behavioral pattern

a. General definition

Behavioral patterns are concerned with algorithms and the assignment of responsibilities between objects are the focus of behavioral patterns. Not only do behavioral patterns represent patterns of objects or classes, but they also pattern of communication between them. Complex control flow that is difficult to follow at run-time is characterized by these patterns. They shift your focus away from flow of control to let you concentrate just on the way objects are interconnected. (Joshi, n.d.)

b. Chain of Responsibility pattern:

Chain of Responsibility pattern is a behavior pattern in which a group of objects is chained together in a sequence and a responsibility (a request) is provided in order to be handled by the group.

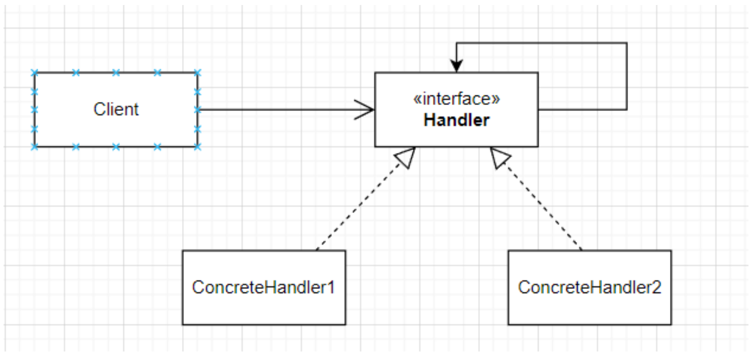


Figure 27. Chain of Responsibility pattern

Applicability: Use Chain of Responsibility when

- More than one objects may handle a request, and the handler isn't known a priori. The handler should be ascertained automatically.
- You want to issue a request to one of several objects without specifying the receiver explicitly.
- The set of objects that can handle a request should be specified dynamically.

Pros	Cons
You can control the order of request handling.	Some requests may end up unhandled.

c. **Command pattern:**

Command pattern is a behavioral design pattern and helps to decouples the invoker from the receiver of a request.

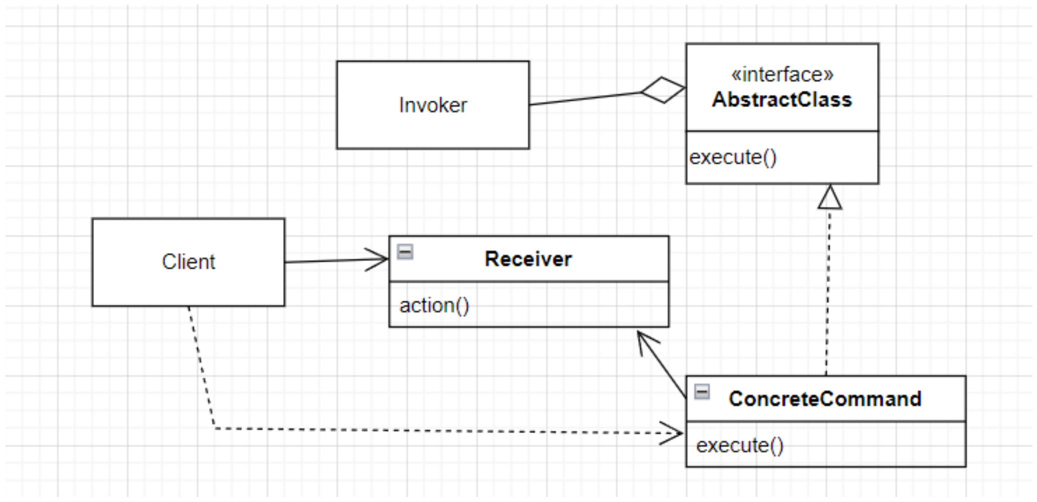


Figure 28. Command pattern

Applicability: Use the Command pattern when you want to:

- Parameterize objects by an action to perform.
- Specify, queue, and execute requests at different times. A Command object can have a lifetime independent of the original request.
- Support undo. The Command's Execute operation can store state for reversing its effects in the command itself.
- Support logging changes so that they can be reapplied in case of a system crash.
- Structure a system around high-level operations built on primitives' operations.

Pros	Cons
You can implement undo/redo. You can implement deferred execution of operations.	The code may become more complicated since you're introducing a whole new layer between senders and receivers.

d. Interpreter pattern:

Interpreter pattern is a heavy-duty pattern. It's all about putting together your own programming language, or handling an existing one, by creating an interpreter for that language.

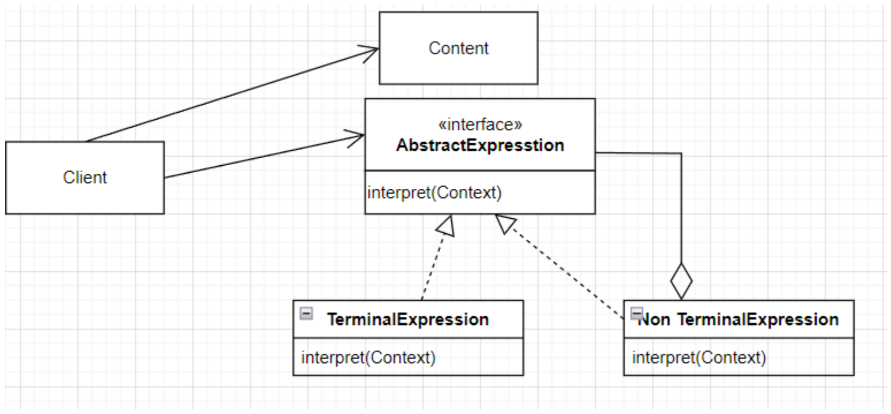


Figure 29. Interpreter pattern

Applicability: Use the Command pattern when

- The grammar is simple. For complex grammars, the class hierarchy for the grammar becomes large and unmanageable. Tools such as parser generators are a better alternative in such cases.
- Efficiency is not a critical concern. The most efficient interpreters are usually not implemented by interpreting parse trees directly but by first translating them into another form.

e. Iterator pattern:

Iterator pattern is to take the responsibility for access and traversal out of the list object and put it into an iterator object.

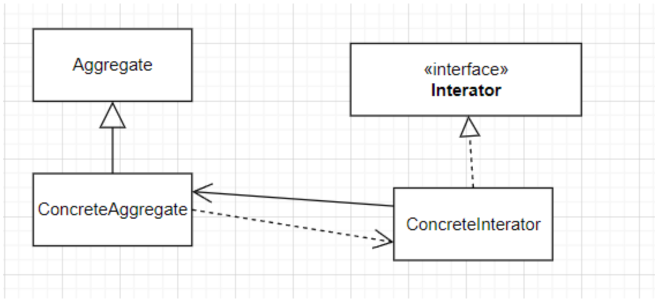


Figure 30. Iterator pattern

Applicability: Use the Iterator pattern:

- To access an aggregate object's contents without exposing its internal representation.
- To support multiple traversals of aggregate objects.
- To provide a uniform interface for traversing different aggregate structures (that is, to support polymorphic iteration).

Pros	**Cons**
+ It supports variations in the traversal of a collection + It simplifies the interface to the collection	+ Using the pattern can be excessive if your app only deals with simple collections. + Using an iterator may be less efficient than going through elements of some specialized collections directly.

f. Mediator pattern:

Mediator pattern defines an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.

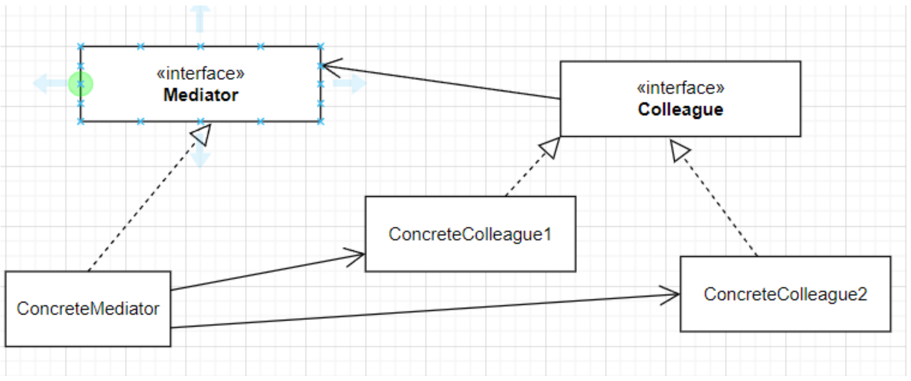


Figure 31. Mediator pattern

Applicability:

- A set of objects communicate in well-defined but complex ways. The resulting interdependencies are unstructured and difficult to understand.
- Reusing an object is difficult because it refers to and communicates with many other objects.
- A behavior that is distributed between several classes should be customizable without a lot of sub-classing.

Pros	Cons
+ It decouples the number of classes. + It simplifies object protocols and centralizes the control.	Over time an arbiter can advance into a God Protest

g. Memento pattern:

Memento pattern without violating encapsulation, to capture and externalize an object’s internal state so that the object can be restored to this state later

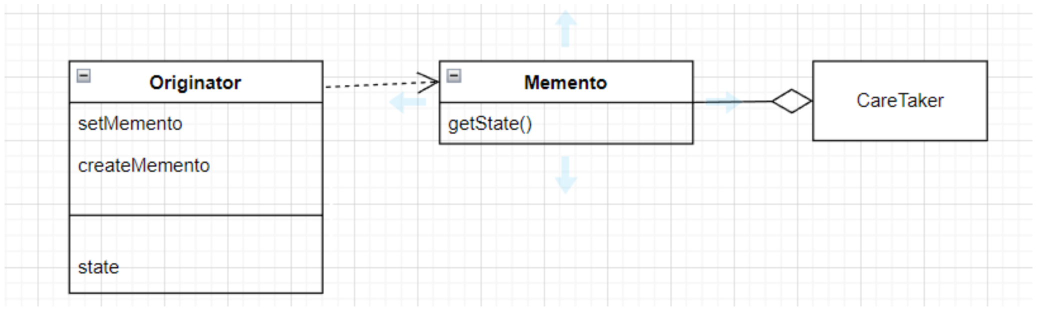


Figure 32. Memento pattern

Applicability:

Use the Memento Pattern in the following cases:

- A snapshot of an object’s state must be saved so that it can be restored to that state later
- A direct interface to obtaining the state would expose implementation details and break the object’s encapsulation.

Pros	Cons
It preserves encapsulation boundaries. It simplifies the originator	If users produce souvenirs too frequently, the application could use a lot of RAM.

h. Observer pattern:

Observer pattern defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

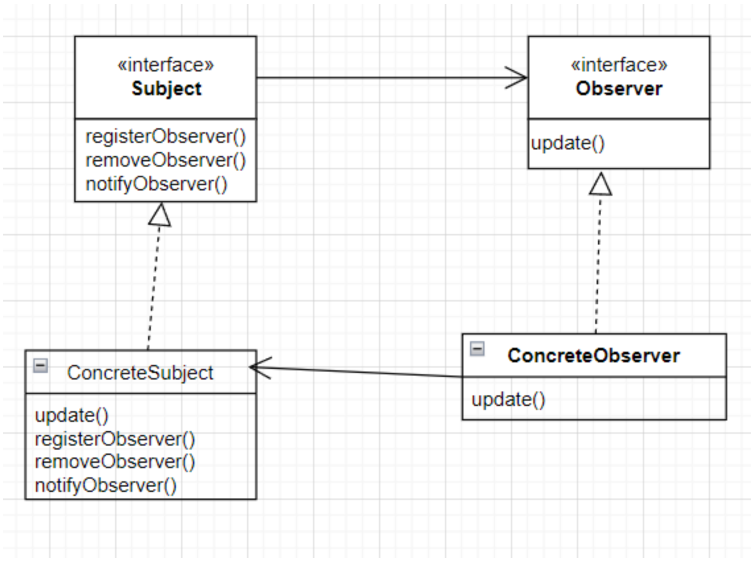


Figure 33. Observer pattern

Applicability:

Use the Observer pattern in any of the following situations:

- When an abstraction has two aspects, one dependent on the other. Encapsulating these aspects in separate objects lets you vary and reuse them independently.
- When a change to one object requires changing others, and you don’t know how many objects need to be changed.

- When an object should be able to notify other objects without making assumptions about who these objects are. In other words, you don’t want these objects tightly coupled.

Pros	Cons
You can establish relations between objects at runtime.	Subscribers are notified in random order.

i. State pattern:

State pattern allows an object to alter its behavior when its internal state changes. The object will appear to change. its class.

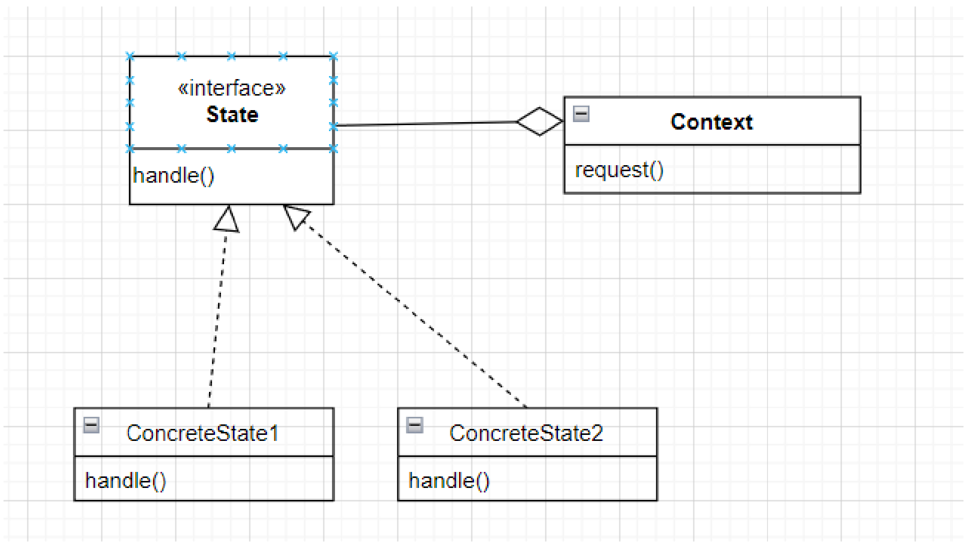


Figure 34. State pattern

Applicability:

Use the State pattern in either of the following cases:

- An object’s behavior depends on its state and change its behavior at run-time depending on that state.
- Operations have large, multipart conditional statements that depend on the object’s state.

j. Strategy pattern:

Strategy pattern defines a family of algorithms, encapsulating each one, and making them interchangeable. Strategy lets the algorithm vary independently from the clients that use it.

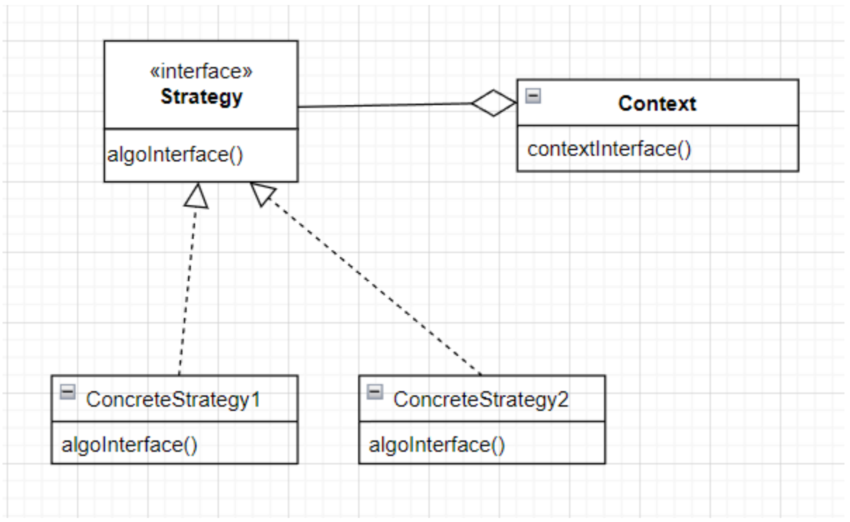


Figure 35. Strategy pattern

Applicability:

Use the Strategy pattern when:

- Many related classes differ only in their behavior. Strategies provide a way to configure a class with one of many behaviors.
- You need different variants of an algorithm. For example, you might define algorithms reflecting different space/time trade-offs. Strategies can be used when these variants are implemented as a class hierarchy of algorithms.
- An algorithm uses data that clients shouldn't know about. Use the Strategy pattern to avoid exposing complex, algorithm specific data structures.
- A class defines many behaviors, and these appear as multiple conditional statements in its operations. Instead of many conditionals, move related conditional branches into their own Strategy class.

Pros	Cons
You can swap algorithms used inside an object at runtime. You can replace inheritance with composition.	On the off chance that you simply have a few of calculations and they once in a while alter, there's no genuine reason to overcomplicate the program with unused classes and interfacing that come together with the design.

k. Template Method pattern:

defines the skeleton of an algorithm in an operation, deferring some steps to subclasses

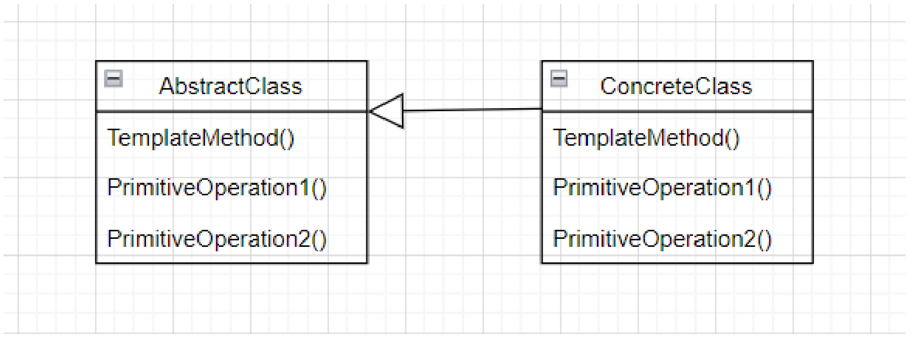


Figure 36. Template Method pattern

Applicability:

The Template Method pattern should be used in the following cases:

- To implement the invariant parts of an algorithm once and leave it up to subclasses to implement the behavior that can vary.
- When common behavior among subclasses should be factored and localized in a common class to avoid code duplication. You first identify the differences in the existing code and then separate the differences into new operations. Finally, you replace the differing code with a template method that calls one of these new operations.
- To control subclasses extensions. You can define a template method that calls "hook" operations (see Consequences) at specific points, thereby permitting extensions only at those points.

Pros	Cons
You can pull the duplicate code into a superclass	Some clients may be limited by the provided skeleton of an algorithm

l. Visitor pattern:

Visitor pattern is to represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.

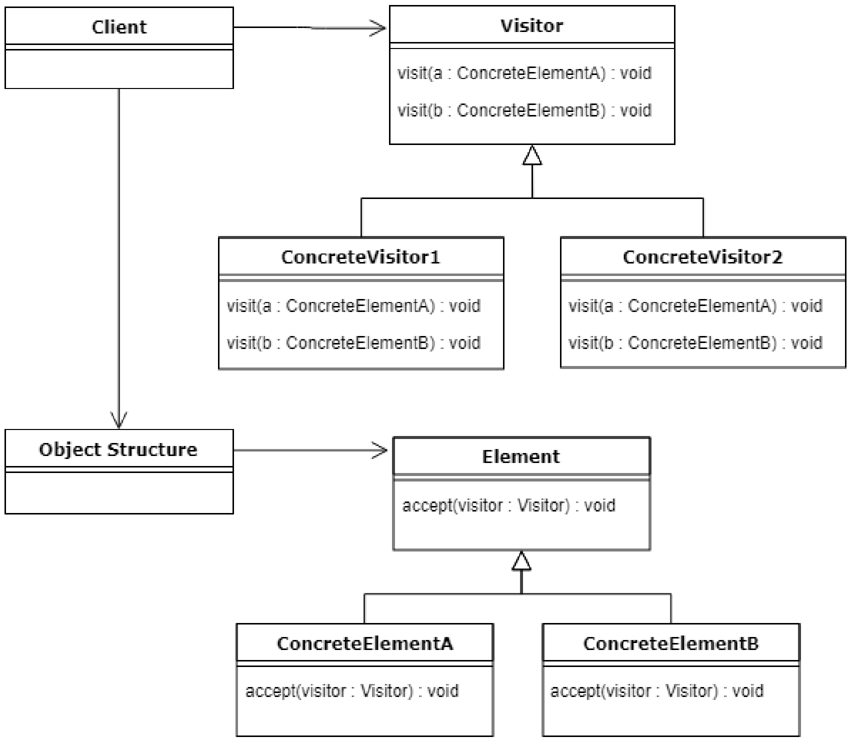

Client
Visitor
visit(a : ConcreteElementA) : void
visit(b : ConcreteElementB) : void
ConcreteVisitor1
visit(a : ConcreteElementA) : void
visit(b : ConcreteElementB) : void
ConcreteVisitor2
visit(a : ConcreteElementA) : void
visit(b : ConcreteElementB) : void
Object Structure
Element
accept(visitor : Visitor) : void
ConcreteElementA
accept(visitor : Visitor) : void
ConcreteElementB
accept(visitor : Visitor) : void

Figure 37. Visitor pattern

Applicability:

Use the Visitor pattern when:

- An object structure contains many classes of objects with differing interfaces, and you want to perform operations on these objects that depend on their concrete classes.
- Many distinct and unrelated operations need to be performed on objects in an object structure, and you want to avoid "polluting" their classes with these operations. Visitor lets you keep related operations together by defining them in one class. When the object structure is shared by many applications, use Visitor to put operations in just those applications that need them.
- The classes defining the object structure rarely change, but you often want to define new operations over the structure. Changing the object structure classes requires redefining the interface to all visitors, which is potentially costly. If the object structure classes change often, then it's probably better to define the operations in those classes.

Pros	Cons
You'll move different forms of the same behavior into the same course.	You wish to upgrade all guests each time a course gets included to or evacuated from the component pecking order.

m. Scenario

In order for members in the FitnessAlpha gym to exercise effectively, coach Paul has provided training instructions with equipment in the gym. Paul temporarily divided into 3 types of training: training without equipment, training using dumbbells, training using training equipment and will be distributed according to the number of reps (Number of continuous exercises in 1 set, calculated as follows: when you finish all movements completely)

- Equipment-free exercises are basic types of exercises where members do not need to use any equipment, just practice according to the set number of reps.
- The type of exercise that uses dumbbells will only use dumbbells to support training with different exercises according to the set number of reps.
- With the multifunctionality of gym equipment, with just one device, members can do a lot of exercises. Depending on the exercises, the number of reps will be customized according to Paul's adjustment.

n. Class diagram

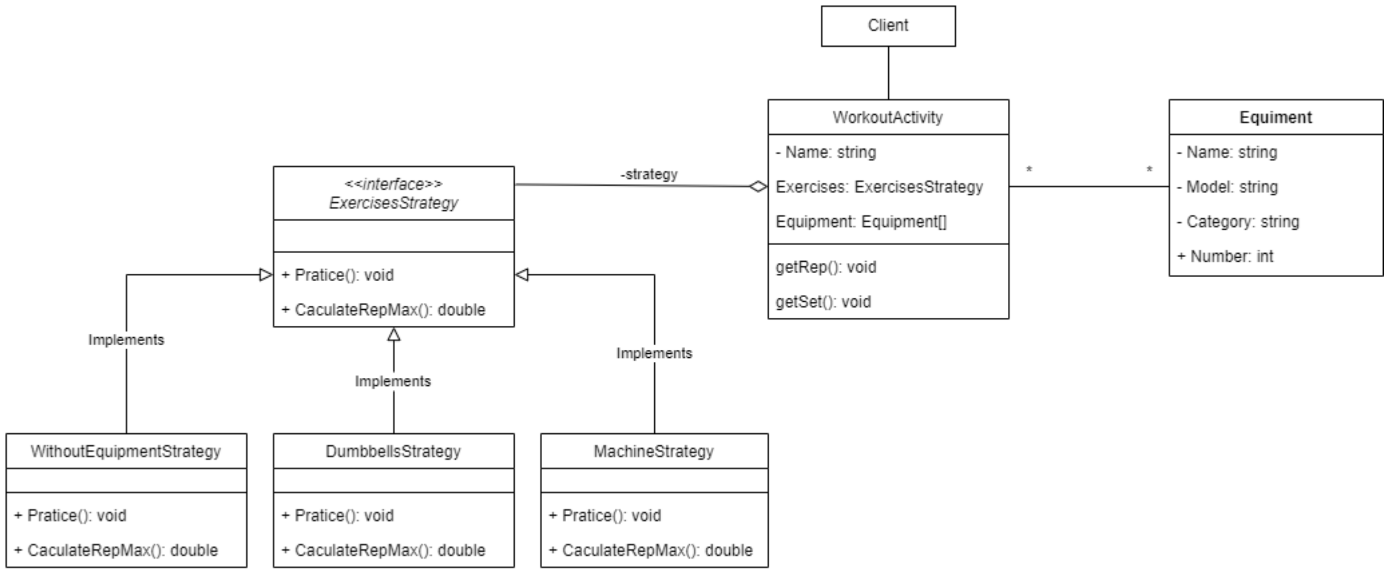


Figure 38. Class diagram 4

Explain:

Class WorkoutAtivity are exercises that have a route and types of equipment that will be used in that exercise. In the gym, there are a lot of equipment and also a lot of exercises, so the relationship of the WorkoutAtivity class with the Equipment class is many.

Besides, the exercises are also divided into many types without tools, with tools and machines, so there will be 3 classes respectively, class WithoutEquipmentStrategy, class DumbbellsStrategy and MachineStrategy This will be the Sub of Supper class ExercisesStrategy.

V. Design Pattern vs OOP

A design pattern is a tried and tested solution to a common programming problem. It's not necessarily an Object Oriented programming problem, but it's the most common one these days. Sometimes you have great ideas, but most of the time you don't really work. Usually, to be able to code effectively from the start, you need to have a lot of experience. Studying Design Patterns is learning good ways to build your program. You're basically reading advice from people who've been building things for decades. They have distilled their most common solutions into simple, easy-to-understand pieces of knowledge with easy-to-remember names. However, you can accept it and take the lead, or ignore it and repeat all their mistakes. In the scenarios outlined above, we can see that knowing and applying the right design pattern to problem solving is highly effective. Thanks to the sample diagrams from before, we were able to build a diagram specifically for the problem posed. This speed and convenience help the system planning work to save more time and manpower.

VI. Conclusion

I had the opportunity to revisit the fundamentals of OOP after finishing this paper. I am familiar with the fundamental OOP concepts of encapsulation, inheritance, polymorphism, abstraction, and the many UML class diagram types that are employed in system analysis and design. To make it simpler for you to comprehend, I explain the idea, symbolism, and usage of each graphic and provide particular examples. In addition, I learned the fundamentals of object-oriented coding and, when applicable, how to represent the organization of their code using a UML class diagram. I'm already familiar with the idea of several sorts of design patterns and can offer instances of each.

VII. Reference

1. Shet, P. (2019). *Object Oriented Programming With A Real-World Scenario*. [online] Available at: https://www.c-sharpcorner.com/UploadFile/cda5ba/object-oriented-programming-with-real-world-scenario/. (Accessed: 15 November 2022)

2. KathleenDollard (n.d.). *Object-oriented programming - Visual Basic*. [online] learn.microsoft.com. Available at: https://learn.microsoft.com/en-us/dotnet/visual-basic/programming-guide/concepts/object-oriented-programming. (Accessed: 15 November 2022)
3. Erich Gamma, R. H. R. J. J. V., 2014. Design Patterns: Elements of Reusable Object Oriented Software, Addison-Wesley. s.l.:s.n.
4. Joshi, R., n.d. *Java Design Patterns.* 2 ed. s.l.:Java Code Geeks.