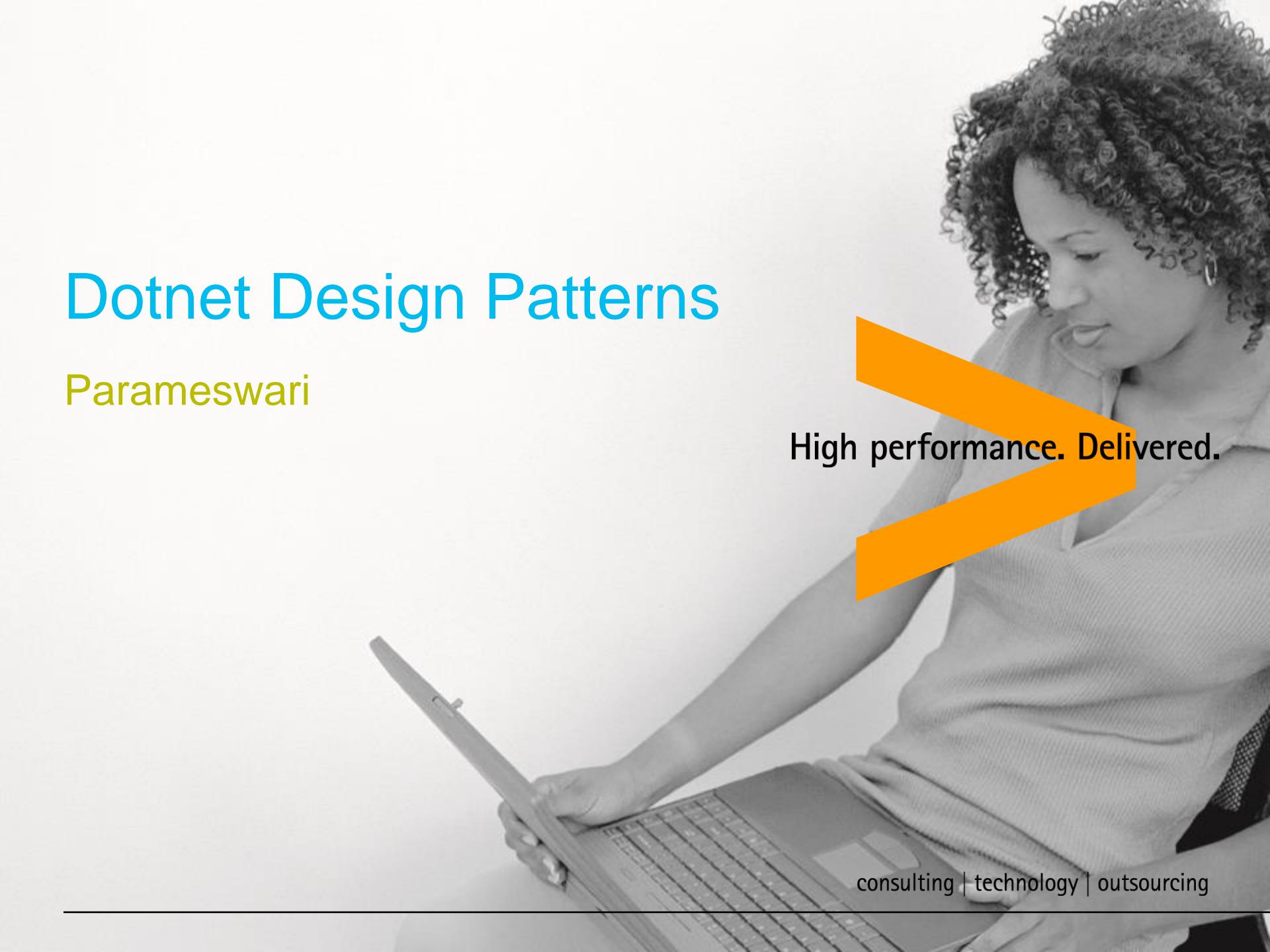


Dotnet Design Patterns

Parameswari

A black and white photograph of a woman with curly hair, wearing a light-colored shirt, sitting at a desk and working on a laptop. She is looking down at the screen. The background is plain.

High performance. Delivered.

consulting | technology | outsourcing

Design Patterns





Goals

- Introduction to Patterns
- MVC,MVP and MVVM
- Language and DSL Pattern
- Observer Pattern
- Iterator Pattern
- Singleton Pattern
- Patterns Relating to Factories
- Strategy Pattern
- Command Pattern



Goals

- Proxy Pattern
- Template Method Pattern
- Decorator and Adapter Pattern
- Observer Pattern
- Visitor Pattern
- State Pattern
- ASP.NET Core Architecture
- ASP.NET Core Application Development
- ASP.NET Core 2.0 Tag Helpers



Goals

- Structuring ASP.NET Application
- Dependency Injection, Configuring and Entity Framework
- TDD
- Custom Tag Helpers
- Navigation
- State Management
- Creating Restful Services using WEBAPI
- Industry best practices for .NET especially CORE based applications
- Efficient way of building project architecture – Do's & Don'ts
- ML applications/use cases (if time permits)



Introduction to Patterns

The Pattern for successful applications

- John Lennon once wrote, “There are no problems, only solutions.”
- Now, Mr. Lennon did much in the way of ASP.NET programming;
- Still, what he said is extremely relevant in the realm of software development and probably humanity.
- The job as software developers involves solving problems — problems that other developers have had to solve countless times.



Design Patterns Explained

- Design patterns are high-level abstract solution templates.
 - It is blueprint for solutions rather than the solutions themselves.
 - It is difficult to find a framework that one can simply apply to your application;
 - Instead, we will typically arrive at design patterns through refactoring your code and generalizing your problem.
 - Design patterns can be found in all areas of life from engineering to architecture.
 - In fact, it was the architect Christopher Alexander who introduced the idea of patterns in 1970 to build a common vocabulary for design discussion.



Origins

- The origins of the design patterns that are prevalent in software architecture today were born from the experiences and knowledge of programmers over many years of using object-oriented programming languages.
- A set of the most common patterns were catalogued in a book entitled *Design Patterns*:
- *Elements of Reusable Object-Oriented Software*, more affectionately known as the *Design Patterns Bible*.
- This book was written by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, better known as the Gang of Four

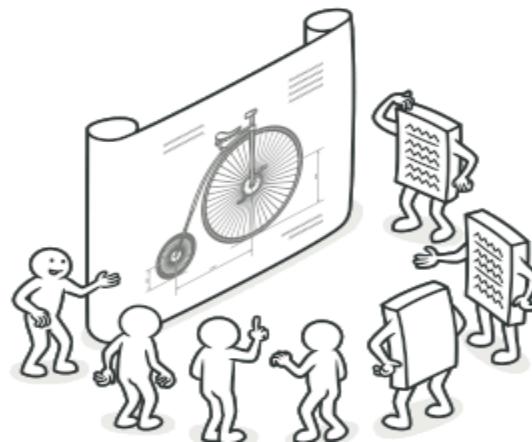


Origins

- They collected 23 design patterns and organized them into 3 groups:
- **Creational Patterns:** These deal with object construction and referencing.
- **Structural Patterns:** These deal with the relationships between objects and how they interact with each other to form larger complex objects.
- **Behavioral Patterns:** These deal with the communication between objects, especially in terms of responsibility and algorithms.

Necessity

- Patterns are essential to software design and development.
- Patterns promote the use of good object-oriented software design, as they are built around solid object-oriented design principles.





Common Design Principles

- Design pattern is one of the common design patterns.
- It has become best practice use design patterns when enterprise-level and maintainable software built.

Keep It Simple Stupid (KISS)

- An all-too-common issue in software programming is the need to overcomplicate a solution.
- The goal of the KISS principle is concerned with the need to keep code simple but not simplistic, thus avoiding any unnecessary complexities.



Common Design Principles

Don't Repeat Yourself (DRY)

- The DRY principle aims to avoid repetition of any part of a system by abstracting out things that are common and placing those things in a single location.
- This principle is not only concerned with code but any logic that is duplicated in a system;
- Ultimately there should only be one representation for every piece of knowledge in a system.



Common Design Principles

Tell, Don't Ask

- The Tell, Don't Ask principle is closely aligned with encapsulation and the assigning of responsibilities to their correct classes.
- The principle states that we should tell objects what actions we want them to perform rather than asking questions about the state of the object.
- Then deciding ourselves on what action you want to perform.
- This helps to align the responsibilities and avoid tight coupling between classes.



Common Design Principles

You Ain't Gonna Need It (YAGNI)

- The YAGNI principle refers to the need to only include functionality that is necessary for the application and put off any temptation to add other features that you may think you need.
- A design methodology that adheres to YAGNI is test-driven development (TDD).
- TDD is all about writing tests that prove the functionality of a system and then writing only the code to get the test to pass.



Common Design Principles

Separation of Concerns (SoC)

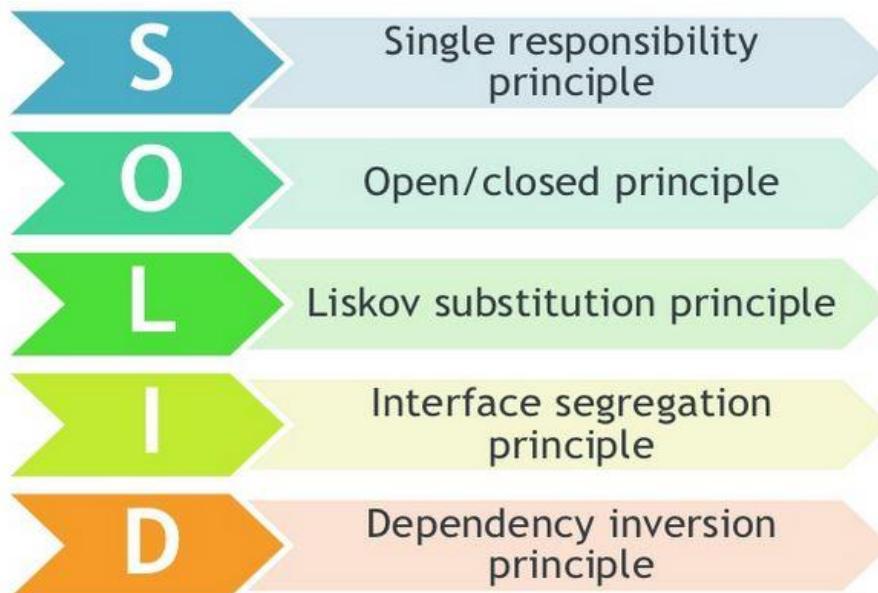
- SoC is the process of dissecting a piece of software into distinct features that encapsulate unique behavior and data that can be used by other classes.
- Generally, a concern represents a feature or behavior of a class.
- The act of separating a program into discrete responsibilities significantly increases code reuse, maintenance, and testability.



Common Design Principles

The S.O.L.I.D. Design Principles

- The S.O.L.I.D. design principles are a collection of best practices for object-oriented design.
- All the Gang of Four design patterns adhere to these principles in one form or another.





Single Responsibility Principle(SRP)

- The principle of SRP is closely aligned with SoC(Separation of Concern).
- It states that every object should only have one reason to change and a single focus of responsibility.
- By adhering to this principle, you avoid the problem of monolithic class design that is the software equivalent of a Swiss army knife.
- By having concise objects, you again increase the readability and maintenance of a system.
- SRP is a way to divide the whole problem into small parts and each part will be dealt with any separate class.



Open/Closed Principle(OCP)

- The OCP states that classes should be open for extension and closed for modification.
- In that we should be able to add new features and extend a class without changing its internal behavior.
- Once we create the class and other parts of the application start using it, we should not change it.
- If we change the class, it will have impact on the working application and it will break.
- If we require additional features, we should extend that class rather than modifying it.
- This way the existing system won't get any impact from the new change.



Liskov Substitution Principle (LSP)

- The LSP dictates that you should be able to use any derived class in place of a parent class.
- It should behave in the same manner without modification.
- This principle is in line with OCP in that it ensures that a derived class does not affect the behavior of a parent class.
- Put another way, derived classes must be substitutable for their base classes.



Interface Segregation Principle (ISP)

- The ISP is all about splitting the methods of a contract into groups of responsibility.
- Hence, client need not to implement one large interface.
- No need to host methods that they do not use.
- The purpose behind this is so that classes wanting to use the same interfaces only need to implement a specific set of methods as opposed to a monolithic interface of methods.
- Incase class has ten methods – five are needed by desktop and five are needed by mobile clients.



Interface Segregation Principle (ISP)

- Thus, the same interface consisting of ten methods is being used by both desktop and mobile clients.
- If desktop application requires some additional methods, then we must add that at interface level and both the application must be updated to new version.
- ISP suggests we avoid such situations, create two separate interfaces one for desktop and another for mobile clients.



Dependency Inversion Principle (DIP)

- The DIP is all about isolating our classes from concrete implementations and having them depend on abstract classes or interfaces.
- It promotes the mantra of coding to an interface rather than an implementation.
- It increases flexibility within a system by ensuring you are not tightly coupled to one implementation.

Dependency Inversion (DI) and Inversion of Control (IoC)



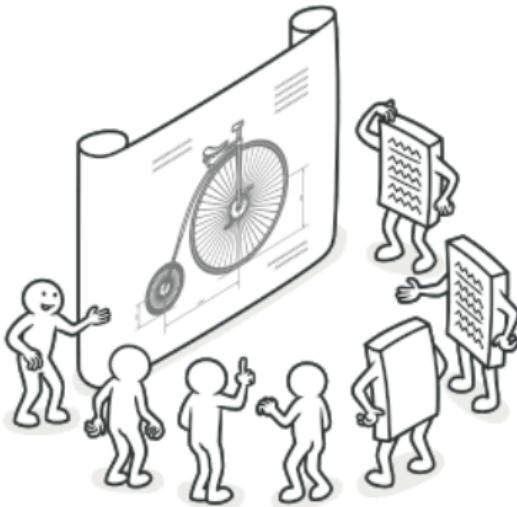
- Closely linked to the DIP are the DI principle and the IOC principle.
- DI is the act of supplying a low level or dependent class via a constructor, method, or property.
- Used in conjunction with DI, these dependent classes can be inverted to interfaces or abstract classes.
- It will lead to loosely coupled systems that are highly testable and easy to change.

Dependency Injection (DI) and Inversion of Control (IoC)



- In IoC, a system's flow of control is inverted compared to procedural programming.
- An example of this is an IoC container, whose purpose is to inject services into client code without having the client code specifying the concrete implementation.
- The control in this instance that is being inverted is the act of the client obtaining the service.

Design Patterns



DESIGN PATTERNS

Design patterns are typical solutions to common problems in software design. Each pattern is like a blueprint that you can customize to solve a particular design problem in your code.

[What's a design pattern?](#)



Benefits of patterns

Patterns are a toolkit of solutions to common problems in software design. They define a common language that helps your team communicate more efficiently.

Catalog of patterns

List of 22 classic design patterns, grouped by their intent.

Design Patterns

Classification

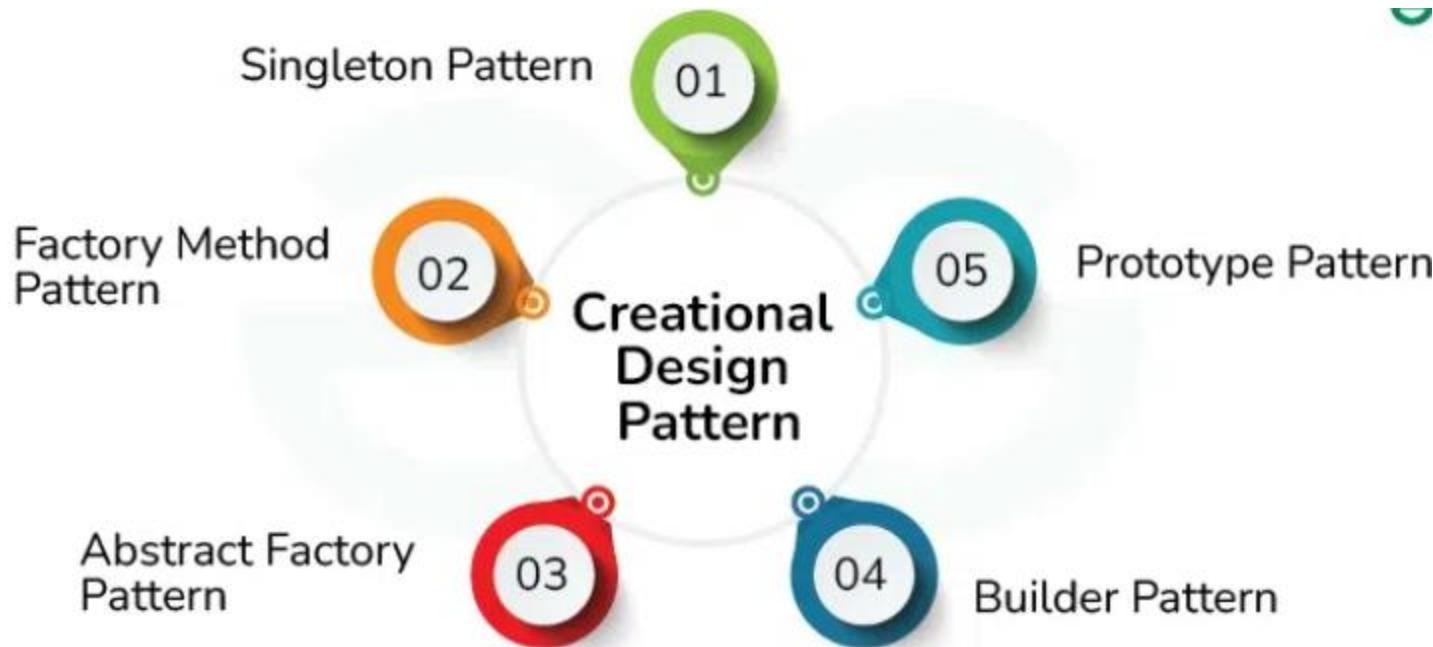
Design patterns differ by their complexity, level of detail and scale of applicability. In addition, they can be categorized by their intent and divided into three groups.



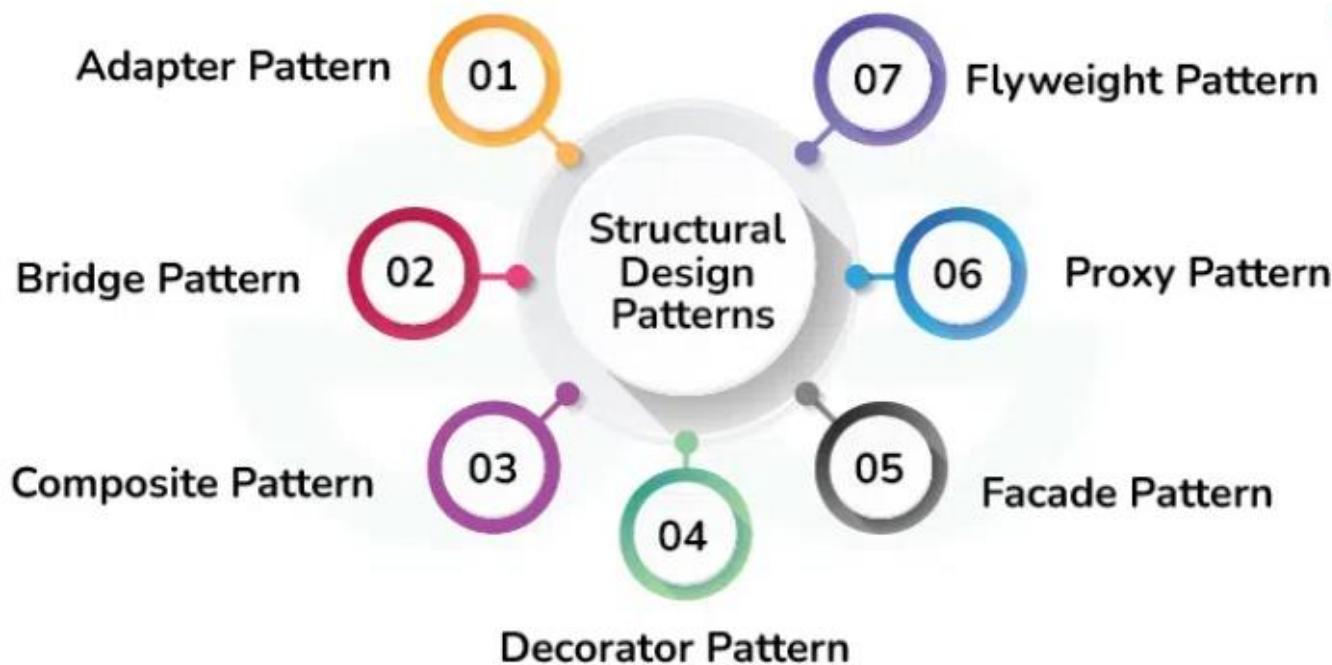
- **Creational patterns** provide object creation mechanisms that increase flexibility and reuse of existing code.
- **Structural patterns** explain how to assemble objects and classes into larger structures, while keeping these structures flexible and efficient.
- **Behavioral patterns** take care of effective communication and the assignment of responsibilities between objects.



Creational Design Patterns



Structural Design Patterns



Structural Design Patterns



Behavioral Design Patterns

01

Observer Pattern

02

Strategy Pattern

03

State Pattern

04

Command Pattern

05

Chain of Responsibility

Template Pattern

06

Interpreter Pattern

07

Visitor Pattern

08

Mediator Pattern

09

Memento Pattern

10



Design Pattern Groups

Factory Method	Abstract Factory	Adapter	Bridge	Chain of Responsibility	Command	Iterator	Mediator
Builder	Prototype	Composite	Decorator	Memento	Observer	State	Strategy
Singleton		Facade	Flyweight	Template method	Visitor		
			Proxy				



Creational

- Creational patterns deal with object construction and referencing.
- They abstract away the responsibility of instantiating instances of objects from the client.
- It keeps the code loosely coupled.
- The responsibility of creating complex objects in one place adhering to the Single Responsibility and Separation of Concerns principles.



Creational

- Following are the patterns in the Creational group:
- **Abstract Factory:** Provides an interface to create families of related objects.
- **Factory:** Enables a class to delegate the responsibility of creating a valid object.
- **Builder:** Enables various versions of an object to be constructed by separating the construction for the object itself.
- **Prototype:** Allows classes to be copied or cloned from a prototype instance rather than creating new instances.
- **Singleton:** Enables a class to be instantiated once with a single global point of access to it.



Structural

- Structural patterns deal with the composition and relationships of objects to fulfill the needs of larger systems.
- Following are the patterns in the Structural group:
- **Adapter:** Enables classes of incompatible interfaces to be used together.
- **Bridge:** As the name suggests, bridge design pattern is used to connect two pieces of code(Abstraction and Implementation). Also, sometimes we need to separate the code so that it is reusable and easy to maintain. And these separate pieces then can be joined together when needed by bridge design pattern.
- **Composite:** Allows a group of objects representing hierarchies to be treated in the same way as a single instance of an object.



Structural

- **Decorator:** Can dynamically surround a class and extend its behavior.
- **Facade:** Provides a simple interface and controls access to a series of complicated interfaces and subsystems.
- **Flyweight:** Provides a way to share data among many small classes in an efficient manner.
- **Proxy:** Provides a placeholder to a more complex class that is costly to instantiate.



Behavioral

- Behavioral patterns deal with the communication between objects in terms of responsibility and algorithms.
- The patterns in this group encapsulate complex behavior and abstract it away from the flow of control of a system, thus enabling complex systems to be easily understood and maintained.



Behavioral

- Following are the patterns in the Behavioral group:
- **Chain of Responsibility:** Allows commands to be chained together dynamically to handle a request.
- **Command:** Encapsulates a method as an object and separates the execution of a command from its invoker.
- **Interpreter:** Specifies how to evaluate sentences in a language.
- **Iterator:** Provides a way to navigate a collection in a formalized manner.
- **Mediator:** Defines an object that allows communication between two other objects without them knowing about one another.

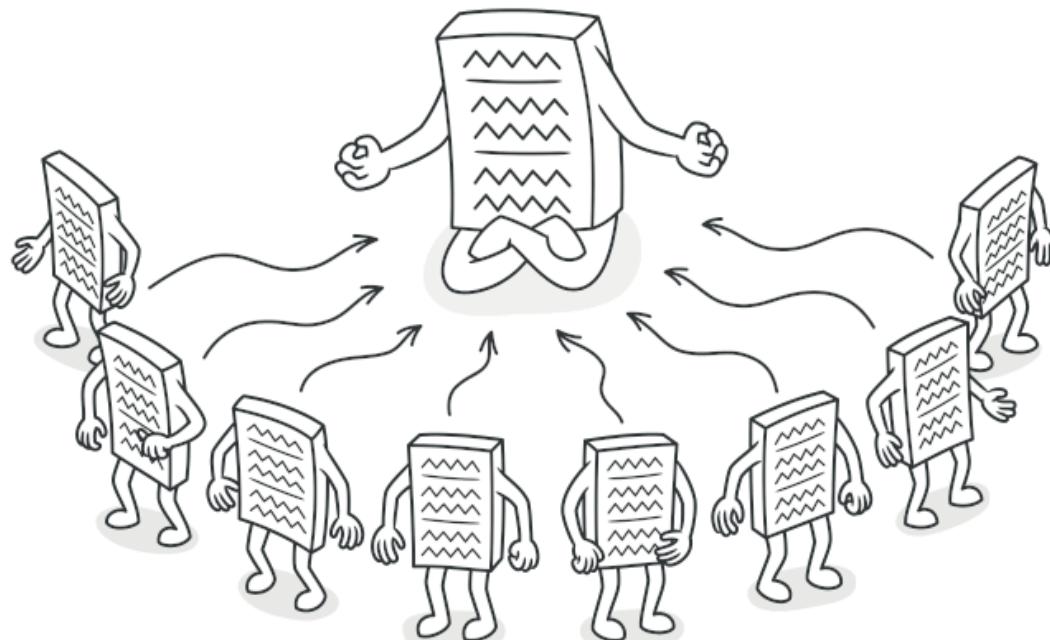


Behavioral

- **Memento:** Allows you to restore an object to its previous state.
- **Observer:** Defines the way one or more classes can be alerted to a change in another class.
- **State:** Allows an object to alter its behavior by delegating to a separate and changeable state object.
- **Strategy:** Enables an algorithm to be encapsulated within a class and switched at run time to alter an object's behavior.
- **Template Method:** Defines the control of flow of an algorithm but allows subclasses to override or implement execution steps.
- **Visitor:** Enables new functionality to be performed on a class without affecting its structure.

Singleton

- Singleton is a creational design pattern that lets you ensure that a class has only one instance, while providing a global access point to this instance.





Singleton

- Key Characteristics of the Singleton Design Pattern in C#
 - Single Instance: This Design Pattern ensures that only one instance of the Singleton class is created throughout the application.
 - Global Access: Provides a global access point to that instance.
 - Lazy Initialization: We can lazily initialize the singleton instance, which means it is created when it is needed for the first time, not when the application starts.
 - Thread Safety: When used in a multi-threaded application, the singleton needs to be thread-safe to prevent multiple instances from being created.



Singleton

- Implementation Guidelines of Singleton Design Pattern in C#:
- Private Parameter less Constructor: We need to create a private and parameter less constructor.
- This is required because it will restrict the class from being instantiated from outside the class; it will only instantiate from within the class.
- Sealed Class: The Singleton class should be declared sealed to ensure it cannot be inherited.
- This will be useful when dealing with nested classes.



Singleton

- Static Variable: We need to create a private static variable that holds the single instance of the class.
- Public Static Method or Property: We also need to create a public static property or method that will return the singleton instance.
- This method or property first checks whether the singleton class instance has been created.
- If the singleton instance is created, it returns that instance; otherwise, it will create an instance and then return it.
- This static method or property provides the global point of access to the singleton instance and ensures that only one instance of the class is created.



Singleton

- Example of Singleton Design Pattern using C#
- We can implement the Singleton Design Pattern in C# in many ways.
- We will discuss all the following methods.
 - No Thread-Safe Singleton Design Pattern in C#
 - Thread-Safety Singleton Implementation using Lock.
 - Implementing Thread-Safety Singleton Design Pattern using Double-Check Locking.
 - Using Eager Loading to Implement Thread-Safety Singleton Design Pattern
 - Using Lazy<T> Generic Class to Implement Lazy Loading in Singleton Design Pattern.



Singleton

- When to Use Singleton Design Pattern in C# Real-time Applications?
- Shared Resources Management: For managing shared resources such as database connection pools or configuration data, where only one instance should manage the resource throughout the system.
- Logging: In scenarios where an application-wide logger instance needs to capture logs from various parts of an application, ensuring that the logging mechanism is consistently used.



Singleton

- Caching: When you need to cache application data so it's accessible globally and maintained within a single object, ensuring consistency.
- Controlled Access and Operations: When you need to perform operations where exactly one instance is needed to coordinate actions across the system, for example, a logging class or managing access to a value shared across various parts of an application.



Singleton

- The advantages of a Singleton Pattern are,
 - Singleton pattern can implement interfaces.
 - Can be lazy-loaded and has Static Initialization.
 - It helps to hide dependencies.
 - It provides a single point of access to a particular instance, so it is easy to maintain.



Singleton

- Disadvantages of Singleton Design Pattern
 - Unit testing is a bit difficult as it introduces a global state into an application
 - Reduces the potential for parallelism within a program by locking.

When to Lazy Loading vs. Eager Loading in Singleton Design Pattern using C#?



- Use lazy loading when:
 - Resource Optimization: If the Singleton instance is heavy and consumes significant resources, lazy loading ensures that these resources are not utilized until necessary.
 - This is useful if there's a chance that the instance might not be needed at all during the application's runtime.
 - Start-up Performance: In applications where start-up time is crucial, lazy loading can help reduce the start-up load by deferring the creation of heavy objects to a later point.

When to Lazy Loading vs. Eager Loading in Singleton Design Pattern using C#?



- This is common in desktop applications where initial responsiveness is critical.
- Conditional Initialization: If your application has multiple execution paths and the Singleton is not required for every path, lazy loading can prevent unnecessary initialization, thus saving resources.

When to Lazy Loading vs. Eager Loading in Singleton Design Pattern using C#?



- Use eager loading when:
 - Predictability: Eager loading contributes to the predictable behavior of the application by initializing the Singleton instance during application startup.
 - Since initialization sequences are consistent, this can simplify debugging and behavior analysis.
 - Concurrency Simplification: In multi-threaded applications, eager loading can simplify the design by avoiding the need for synchronization mechanisms required for safely lazy loading the instance.

When to Lazy Loading vs. Eager Loading in Singleton Design Pattern using C#?



- Once the instance is created during startup, it can be accessed by multiple threads without additional overhead or complexity.
- Performance Critical Situations: If the Singleton is used in performance-critical parts of the application and must be accessed quickly without delay, having it already created and available (eager loading) ensures that there is no delay in accessing the instance when needed.



Realtime Singleton

- Service Proxies
- Database Connection Management
- Load Balancers
- Application Configuration Management
- User Session Management
- Application's Theme Manager
- System Information Gatherer
- Notification Manager
- Task Scheduler
- Service Locator
- Data Sharing
- Application Counter Manager



Factory Method

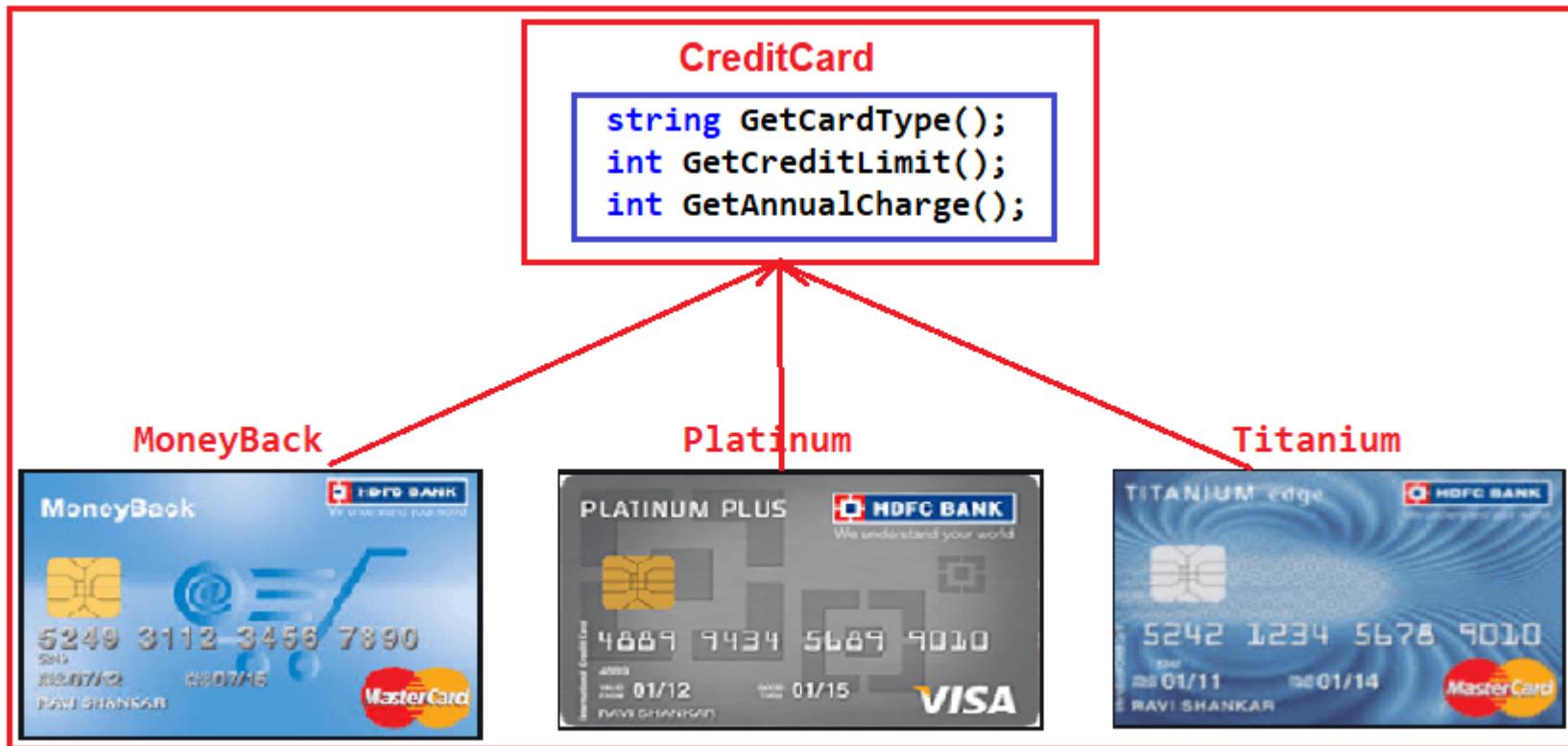
- According to Gang of Four (GoF), a factory is an object used to create other objects.
- In technical terms, a factory is a class with a method.
- That method creates and returns different objects based on the received input parameter.
- In simple words, if we have a superclass and n number of subclasses, and based on the data provided, if we must create and return the object of one of the subclasses, then we need to use the Factory Design Pattern in C#.



Factory Method

- In the Factory Design pattern, we create an object without exposing the Object Creation and Initialization logic to the client, and the client will refer to the newly created object using a common interface.
- The basic principle behind the Factory Design Pattern is that, at run time, we get an object of a similar type based on the parameter we pass.
- So, the client will get the appropriate object and consume the object without knowing how the object is created and initialized.

Real-Time Example to Understand Factory Design Pattern in C#



Real-Time Example to Understand Factory Design Pattern in C#



Main Method

Client

Factory

CreateType(type)

CreditCardFactory

GetCreditCard(string cardType)

Abstract Product

CreditCard

MoneyBack

Product1

Product2

Titanium

Real-Time Examples of the Factory Design Pattern in C#



- Payment Gateway Integration
- Document Conversion System
- Logging System
- A Simple System to Handle Notifications
- Discounts in an E-commerce Application
- Transport Application
- Developing a Graphics Editor
- Designing a System for a Bank
- Report Generation
- Cloud Storage System
- UI Theme System



Abstract Factory Pattern

- The Abstract Factory Design Pattern provides a way to encapsulate a group of factories with a common theme without specifying their concrete classes.
- Abstract means hiding some information, factory means which produces the products, and pattern means a design.
- So, the Abstract Factory Pattern is a software design pattern that provides a way to encapsulate a group of individual factories that have a common theme.



Abstract Factory Pattern

- In simple words, the Abstract Factory is a super factory that creates other factories.
- It is also called the Factory of Factories.
- The Abstract Factory design pattern provides an interface for creating families of related or dependent products but leaves the actual object creation to the concrete factory classes.

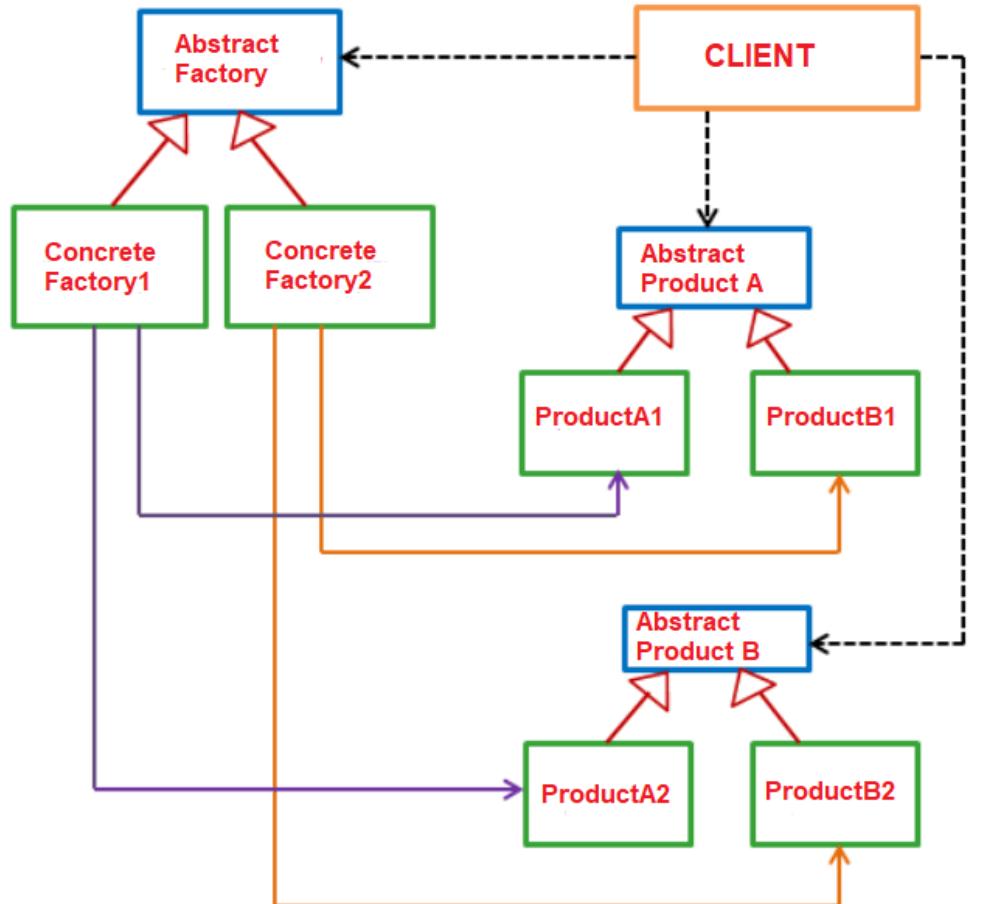
Components of Abstract Factory Design Pattern:

- Abstract Factory: Declares an interface for operations that create abstract products.
- This will be an interface for operations that will create Abstract Product objects.
- Concrete Factory: Implements the operations to create concrete product objects.
- These classes implement the Abstract Factory interface and provide implementations for the interface methods.
- We can use these concrete classes to create concrete product objects.

Components of Abstract Factory Design Pattern:

- Abstract Product: Declares an interface for a type of product object.
- These are going to be interfaces for creating abstract products.
- Here, we need to define the Operations a Product should have.
- Concrete Product: Implements the Abstract Product interface.
- These are the classes that implement the Abstract Product interface.
- Client: Uses interfaces declared by Abstract Factory and Abstract Product classes.
- This class will use our Abstract Factory and Abstract Product interfaces to create a family of products.

Components of Abstract Factory Design Pattern:



Abstract Factory Components in Our Example

Client: Main Method of Program Class

Abstract Product A: IBike.cs

Abstract Product B: ICar.cs

ProductA1: RegularBike.cs

ProductB1: SportsBike.cs

ProductA2: RegularCar.cs

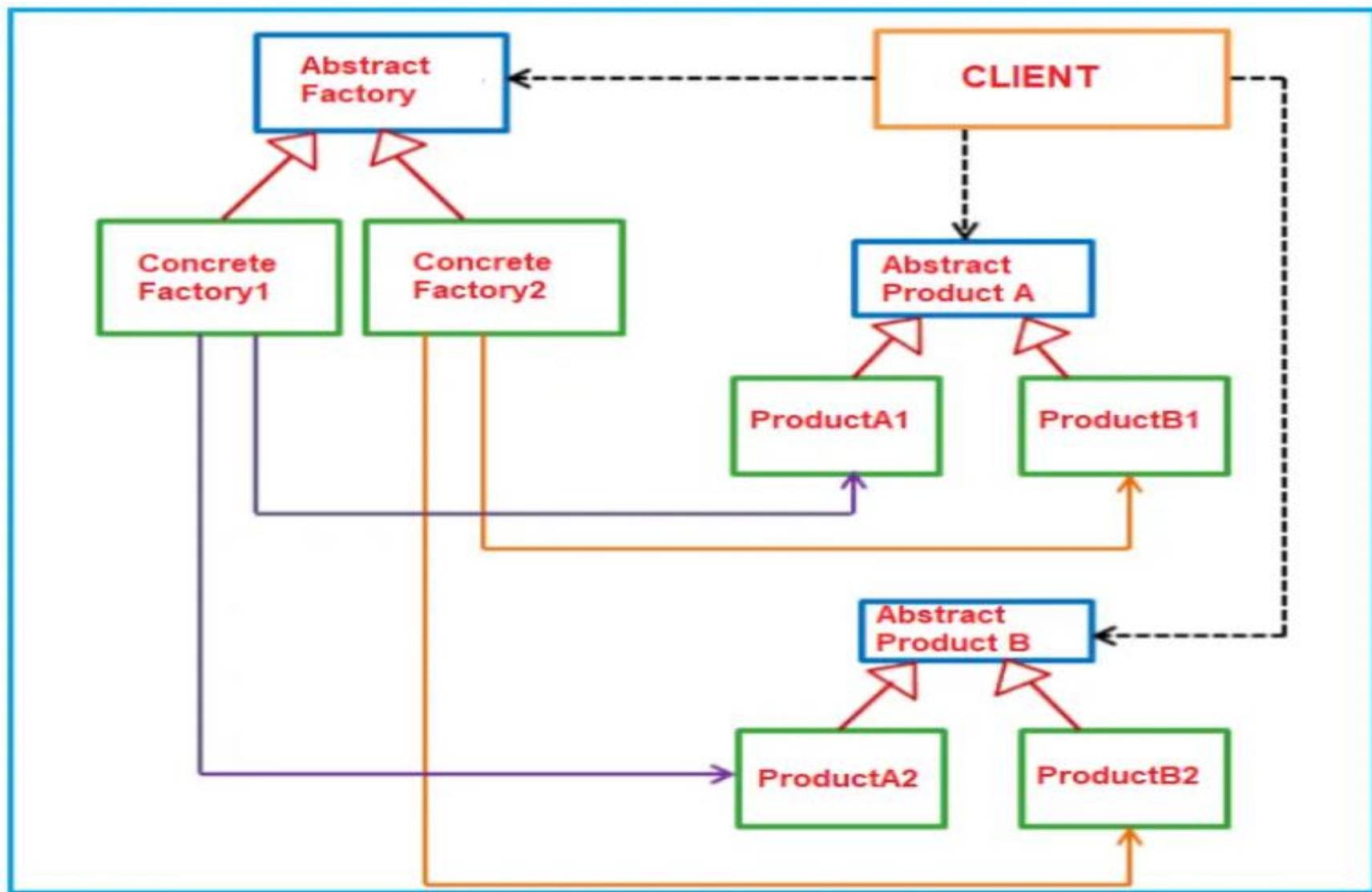
ProductB2: SportsCar.cs

Abstract Factory: IVehicleFactory.cs

Concrete Factory1: RegularVehicleFactory.cs

Concrete Factory2: SportsVehicleFactory.cs

Components of Abstract Factory Design Pattern:



Real-Time Examples of the Abstract Factory Design Pattern in C#



- Payment Gateways in E-commerce
- Cross-Platform UI Development
- Vehicle Manufacturing Company
- Cross-Platform Application Configuration
- Furniture Shop
- Managing Connections to Different Types of Databases
- Multi-Device User Interfaces
- Animal Kingdoms
- Multimedia Software
- Beverages



Builder Pattern

- Builder Design Pattern builds a complex object using many simple objects and a step-by-step approach.
- The Process of constructing the complex object should be so generic that the same construction process can be used to create different representations of the same complex object.
- So, the Builder Design Pattern is about separating the construction process of a complex object from its representation, allowing the same construction process to create different representations.
- When the construction process of the object is very complex, we only need to use the Builder Design Pattern.

Builder Pattern

Laptop Components



Generic Construction Process

1. Plug the memory
2. Plug the Hard Drive
3. Plug the battery
4. Plug the Keyboard
-
-
10. Cover the Laptop with plastic case

Builder Pattern



Add Water



Add Milk



Add Sugar



Add Coffee Powder



Coffee

Builder Pattern



Add Water



Add Milk



Add Sugar



Add Tea Powder



Tea

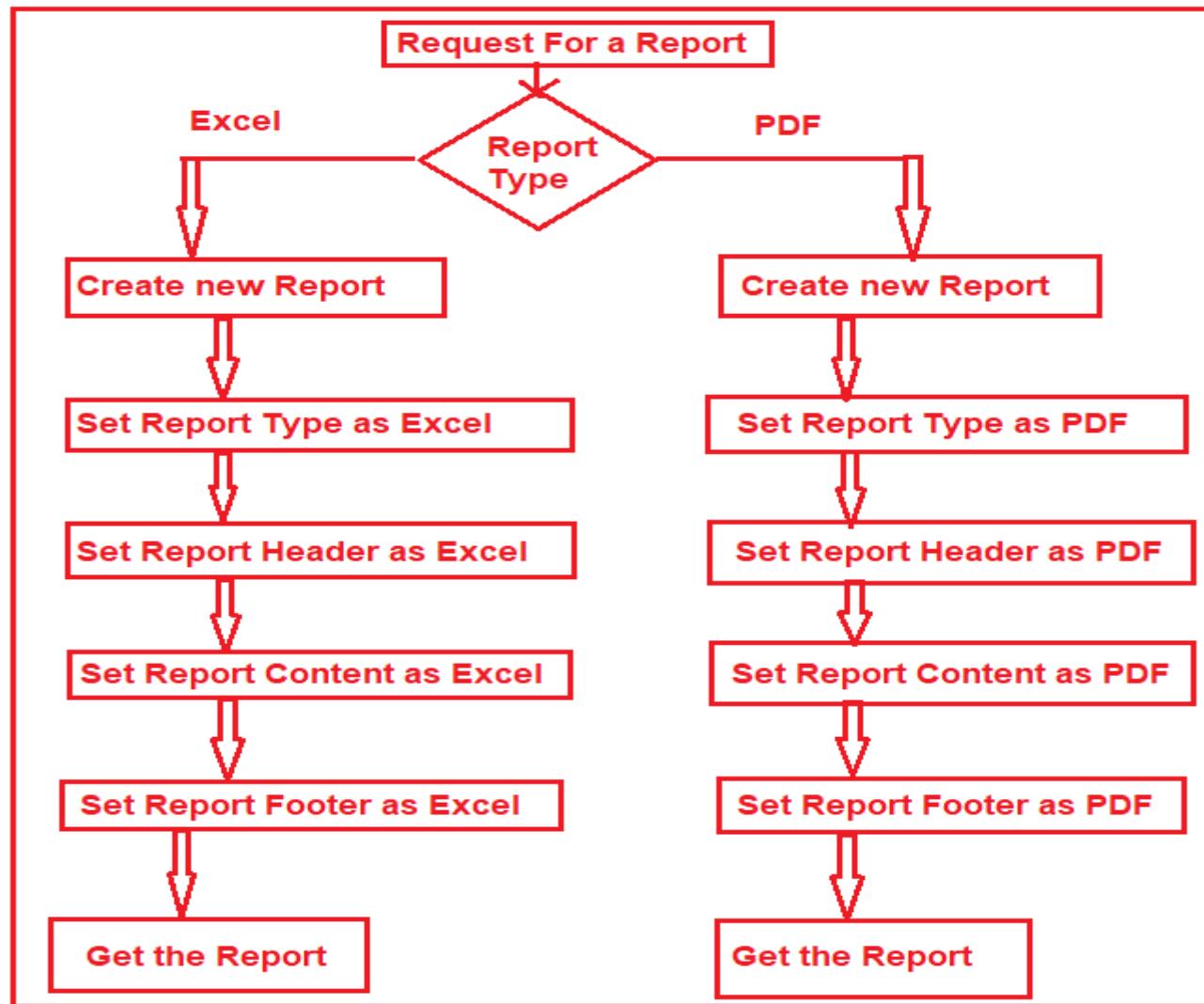


Builder Pattern

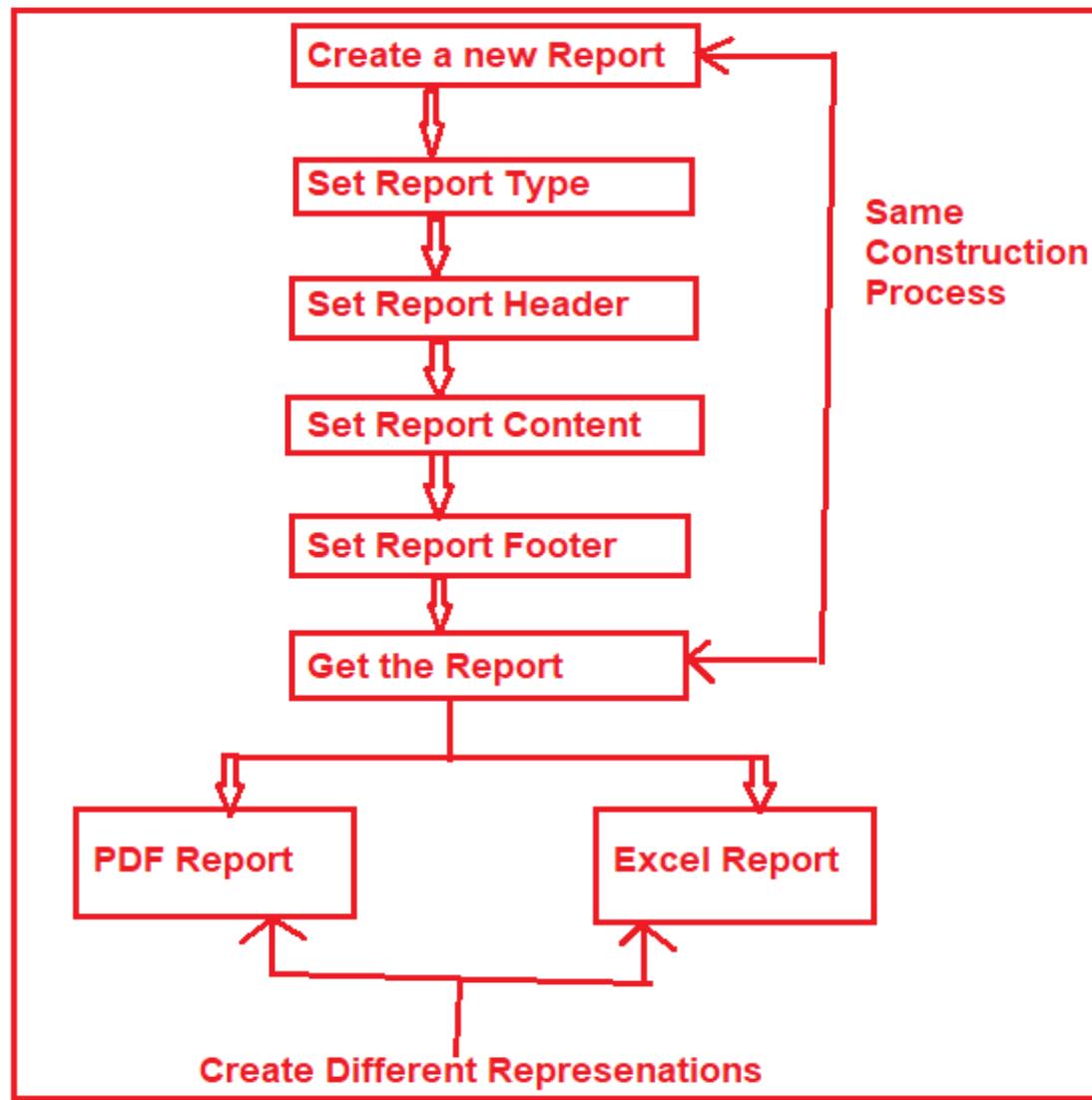
1. Add Water
2. Add Milk
3. Add Sugar
4. Add Powder

Beverage Builder (Generic Process)

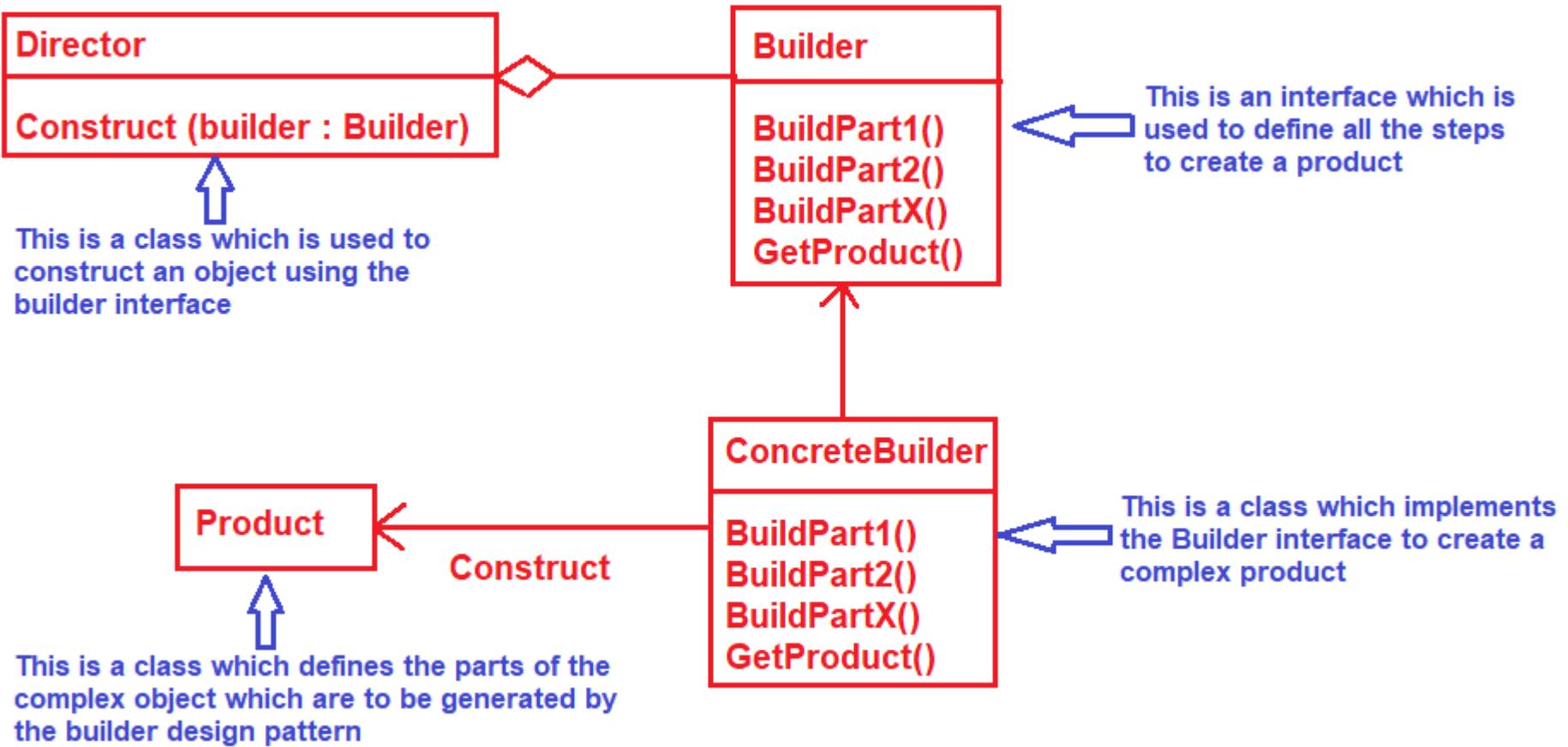
Builder Pattern



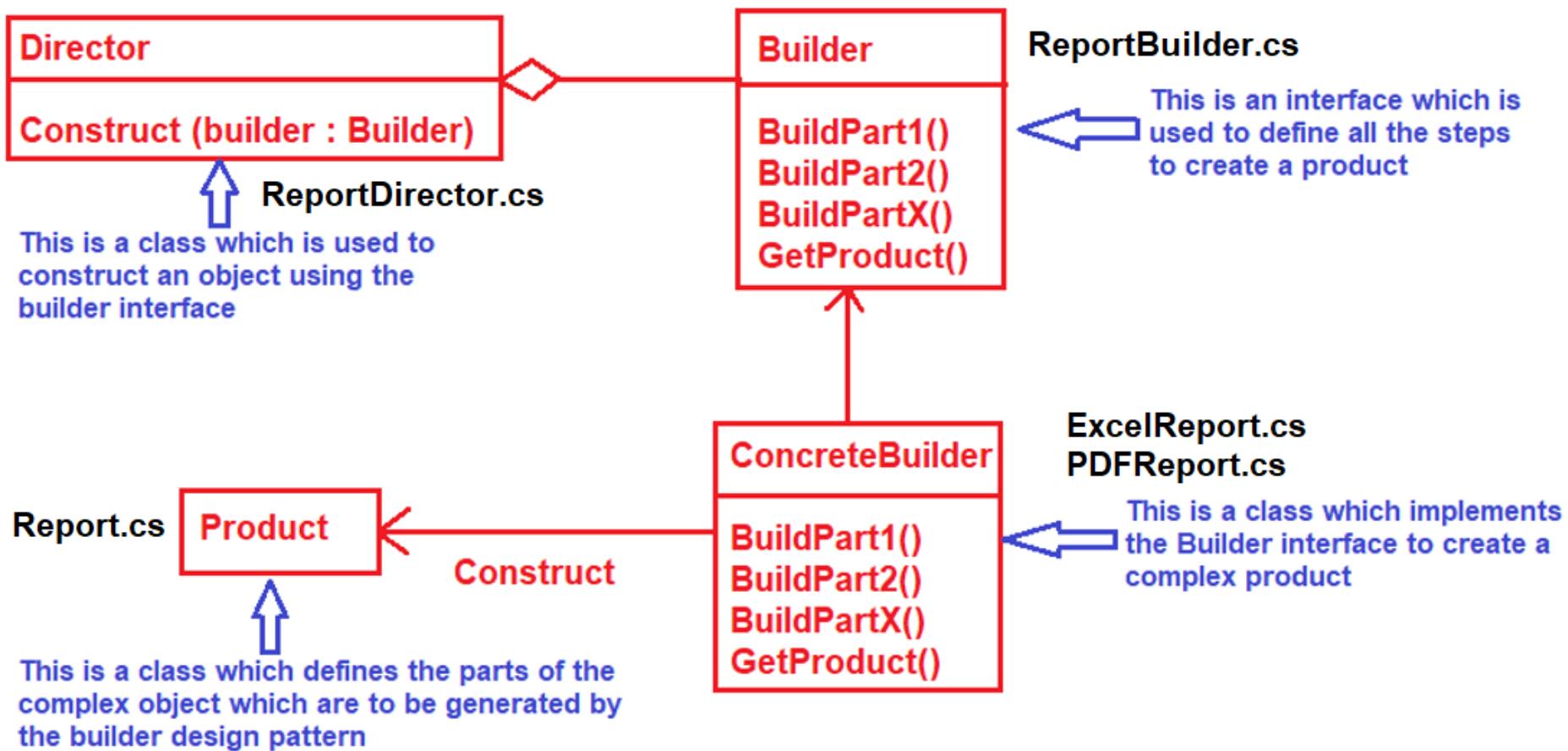
Builder Pattern



Builder Pattern



Builder Pattern



When to use the Builder Design Pattern in C#?



- **Complex Constructors:** When an object requires a complex construction process that involves more than just a few steps or requires a significant setup of various fields and nested objects, the Builder Design Pattern can simplify these tasks by spreading the construction across multiple methods within a builder class.
- **Immutable Objects:** When building an immutable object that does not allow modification after its creation, using a Builder makes it possible to set all of its attributes at creation time and then expose the final object without the ability to alter it.



When to use the Builder Design Pattern in C#?

- Constructing Composite Trees: In scenarios where you are building a complex tree structure, such as a document with various elements (like paragraphs, images, tables), the Builder Design Pattern provides a mechanism to ensure that different nodes or parts of the tree can be constructed in isolation and then composed as needed.
- Multiple Representations of an Object: If your application needs to create different representations of a product (for example, different types of reports that might share the same construction process but have different features or specifications), the Builder Design Pattern allows the construction process to reuse the same building steps while varying the product details.



Builder Pattern

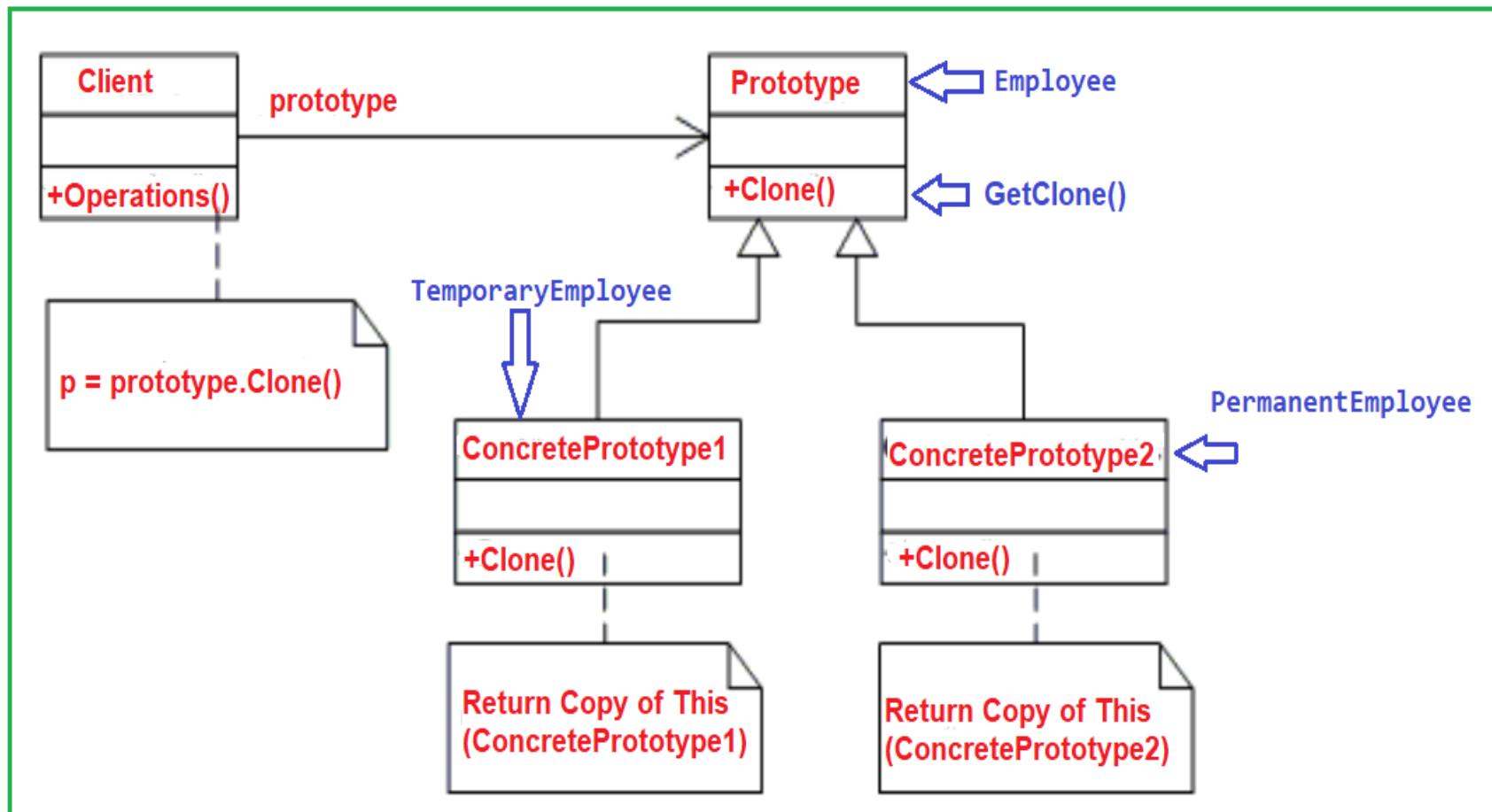
- Real-Life Example
- To complete an order for a computer, different hardware parts are assembled based on customer preferences.
- For example, a customer can opt for a 500GB hard disk with an Intel processor, and another customer can choose a 250GB hard disk with an AMD processor.
- Computer World Example
- You can use this pattern when you want to convert one text format to another text format, such as converting from RTF to ASCII.



Prototype Pattern

- Prototype Design Pattern specifies the kind of objects to create using a prototypical instance and creates new objects by copying this prototype.
- The Prototype Design Pattern creates objects by copying an existing object, known as the prototype.

Prototype Pattern



When to Use Prototype Design Pattern in C#?



- High-Cost Initialization: If initializing an object is resource-intensive, possibly because it requires complex computation, database operations, or reading from external files, using a prototype can be beneficial.
- Once the initial object is created, subsequent objects can be cloned, significantly reducing the cost of creating each new instance.
- Object Pooling: Real-time systems often use object pools (pre-instantiated objects ready to be reused) to manage resources efficiently, especially in high-load scenarios.
- The Prototype pattern can facilitate the management of such pools, allowing for the resetting and reusing of objects by cloning a clean prototype instead of recreating objects from scratch.

When to Use Prototype Design Pattern in C#?



- When Flexibility is Required Over Static Behavior:
Prototypes provide more flexibility in how objects can be constructed compared to other creational patterns like Factory or Builder, which might require new classes to be created for each new object type.
- With prototypes, you can work with object instances that can be cloned to produce new instances, facilitating dynamic behavior.
- Managing Families of Related Objects: If you need to manage families of related objects designed to be used together.
- We need to ensure that the objects you create are always in a particular state or configuration, prototypes can provide a predefined set of objects that are already configured for use.



Prototype Pattern

- Real-Life Example
 - Suppose you have a master copy of a valuable document.
 - You need to incorporate some change into it to analyze the effect of the change.
 - In this case, you can make a photocopy of the original document and edit the changes in the photocopied document.
- Computer World Example
 - Let's assume that you already have an application that is stable.
 - In the future, you may want to modify the application with some small changes.
 - You must start with a copy of your original application, make the changes, and then analyze further.
 - Surely you do not want to start from scratch to merely make a change; this would cost you time and money.



Deep vs Shallow Copy

- Shallow Copy: The `MemberwiseClone` method in C# performs a shallow copy.
- It copies the values of the fields of an object to a new object. If the field is a value type, a bit-by-bit copy of the field is performed.
- For reference types, the reference is copied, not the object itself.
- Deep Copy: If your object has reference-type fields, you might need to implement deep cloning to avoid shared references in your cloned objects.
- Deep cloning involves creating copies of the objects referenced by the fields.

Shallow Copy

```
public class Employee
{
    public string Name { get; set; }
    public string Department { get; set; }
    public Address EmpAddress { get; set; }

    public Employee GetClone()
    {
        return (Employee)this.MemberwiseClone();
    }
}

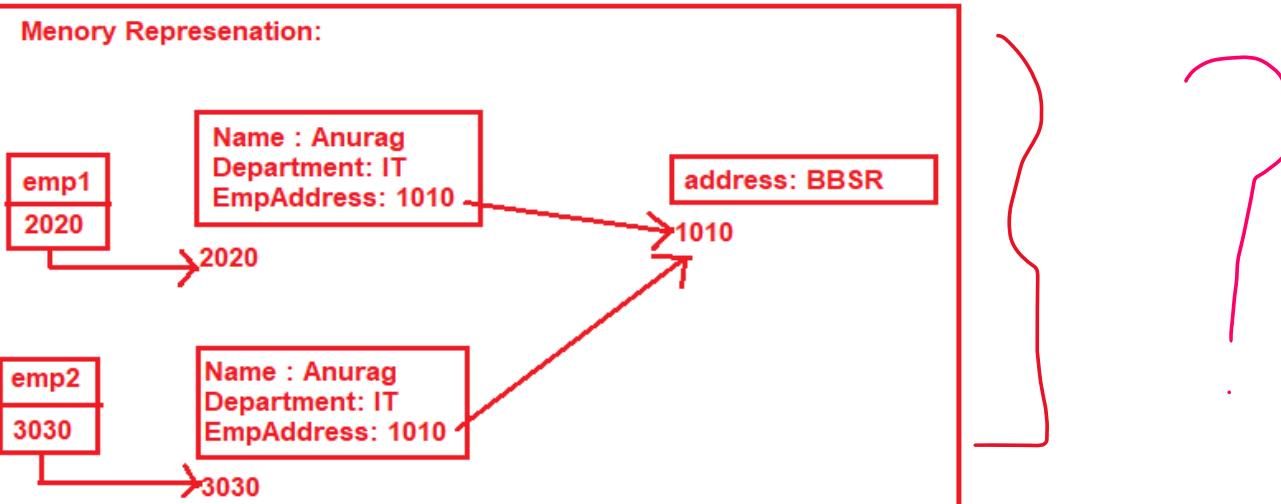
public class Address
{
    public string address { get; set; }
}
```

```
static void Main(string[] args)
{
    Employee emp1 = new Employee();
    emp1.Name = "Anurag";
    emp1.Department = "IT";
    emp1.EmpAddress = new Address() { address = "BBSR" };

    Employee emp2 = emp1.GetClone();
    emp2.Name = "Pranaya";
    emp2.EmpAddress.address = "Mumbai";
}
```

Client Code

Memory Representation:



Deep Copy

```

public class Employee
{
    public string Name { get; set; }
    public string Department { get; set; }
    public Address EmpAddress { get; set; }

    public Employee GetClone()
    {
        Employee employee = (Employee)this.MemberwiseClone();
        employee.EmpAddress = EmpAddress.GetClone();
        return employee;
    }
}

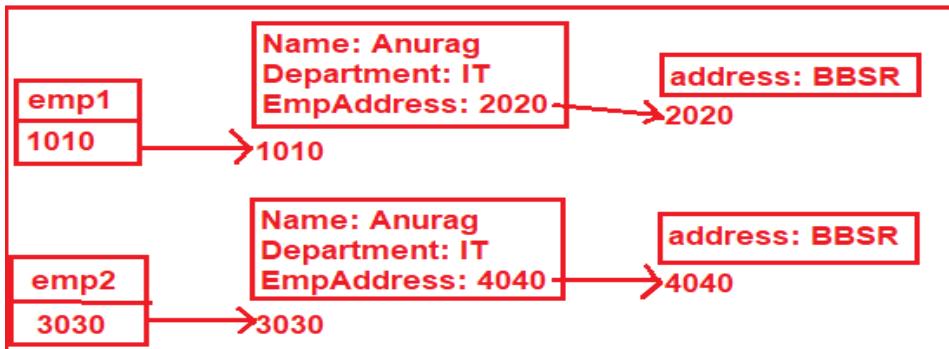
public class Address
{
    public string address { get; set; }
    public Address GetClone()
    {
        return (Address)this.MemberwiseClone();
    }
}
    
```

```

static void Main(string[] args)
{
    Employee emp1 = new Employee();
    emp1.Name = "Anurag";
    emp1.Department = "IT";
    emp1.EmpAddress = new Address()
    {
        address = "BBSR"
    };

    Employee emp2 = emp1.GetClone();
    emp2.Name = "Pranaya";
    emp2.EmpAddress.address = "Mumbai";
}
    
```

Client Code



Memory Representation



Adapter Pattern

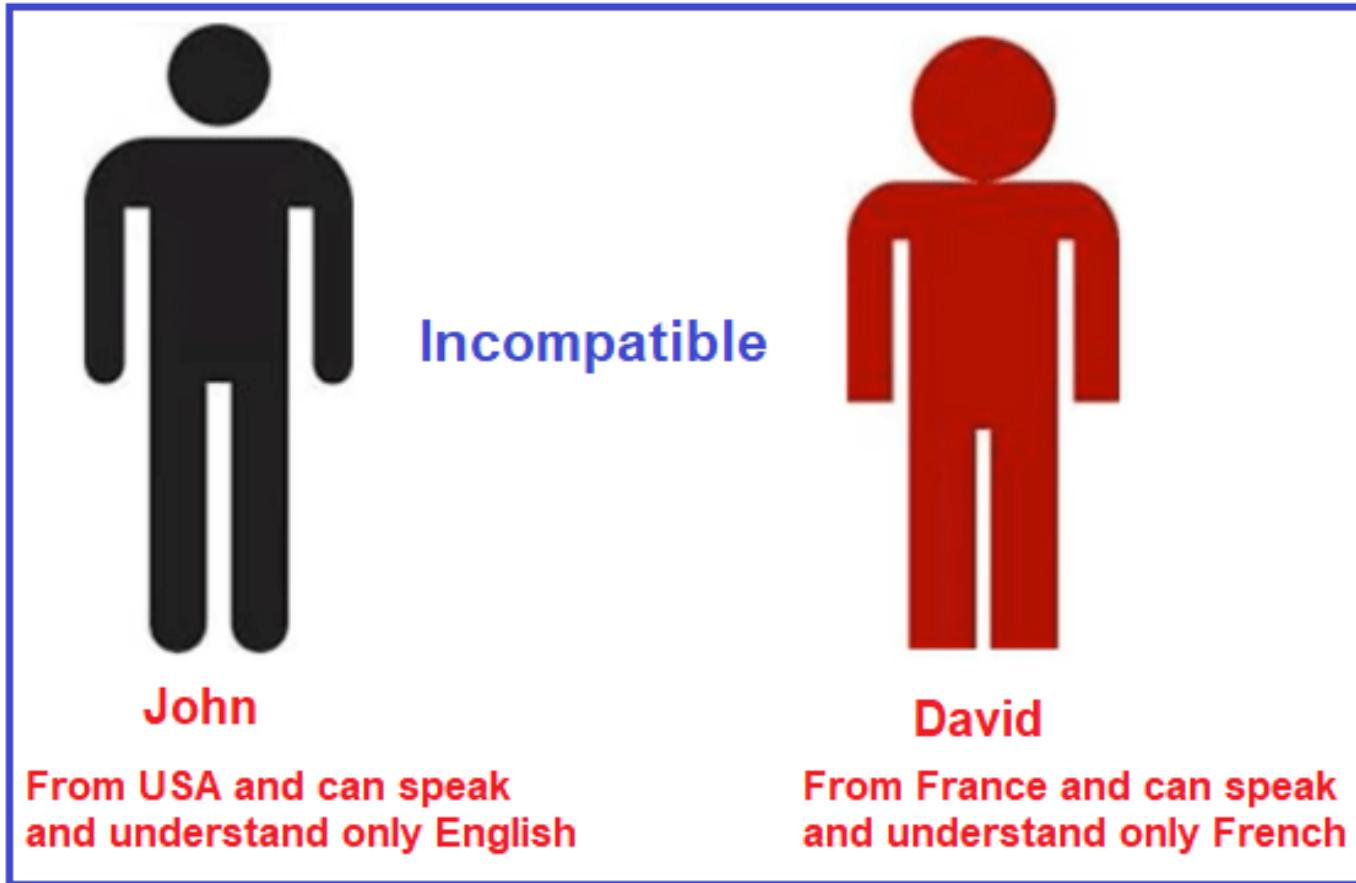
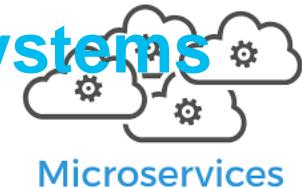
- The Adapter Design Pattern is a structural pattern that allows objects with incompatible interfaces to work together.
- It acts as a bridge between two incompatible interfaces. This pattern is useful when you want to use existing classes, but their interfaces do not match the one you need.
- The Adapter Design Pattern acts as a bridge between two incompatible objects.
- Let's say the first object is A and the second object is B.
- Object A wants to consume some of object B's services.



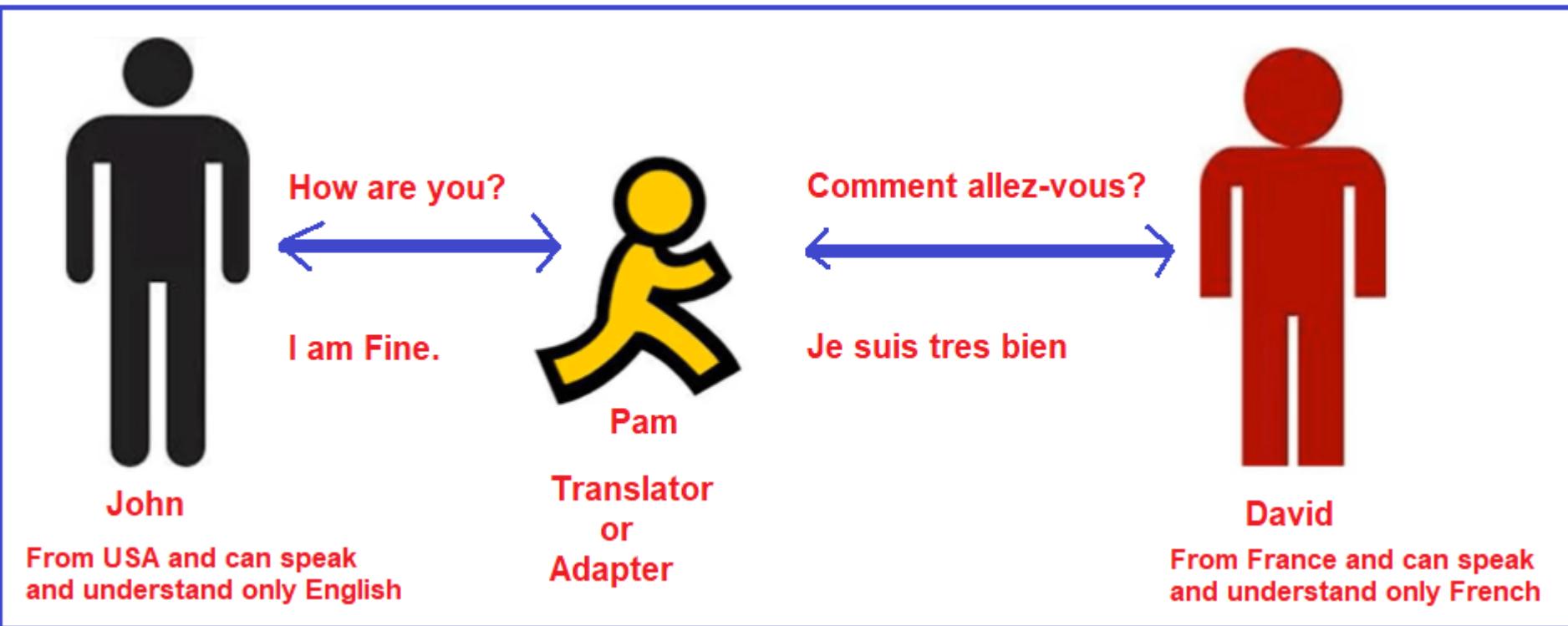
Adapter Pattern

- However, these two objects are incompatible and cannot communicate directly.
- In this case, the Adapter will come into the picture and act as a middleman or bridge between objects A and B.
- Now, object A will call the Adapter, and the Adapter will do the necessary transformations or conversions, and then it will call object B.

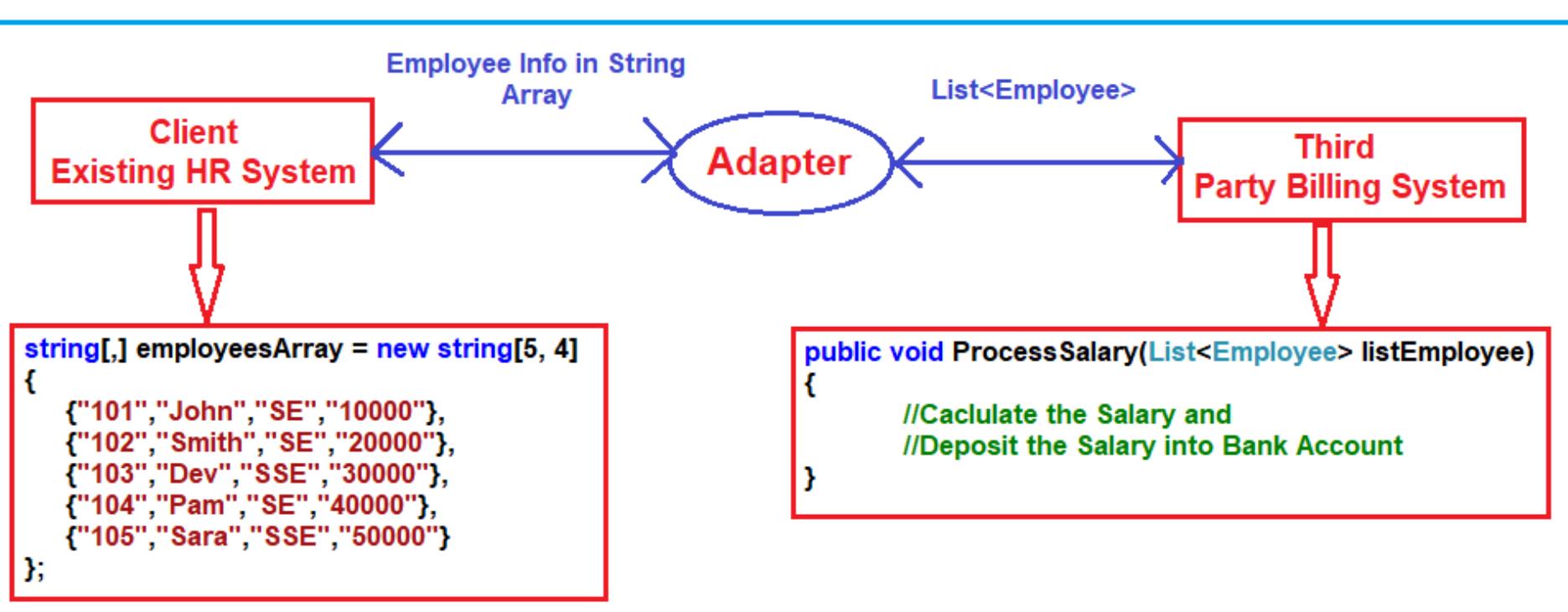
How can we make these two incompatible systems work together?



How can we make these two incompatible systems work together?

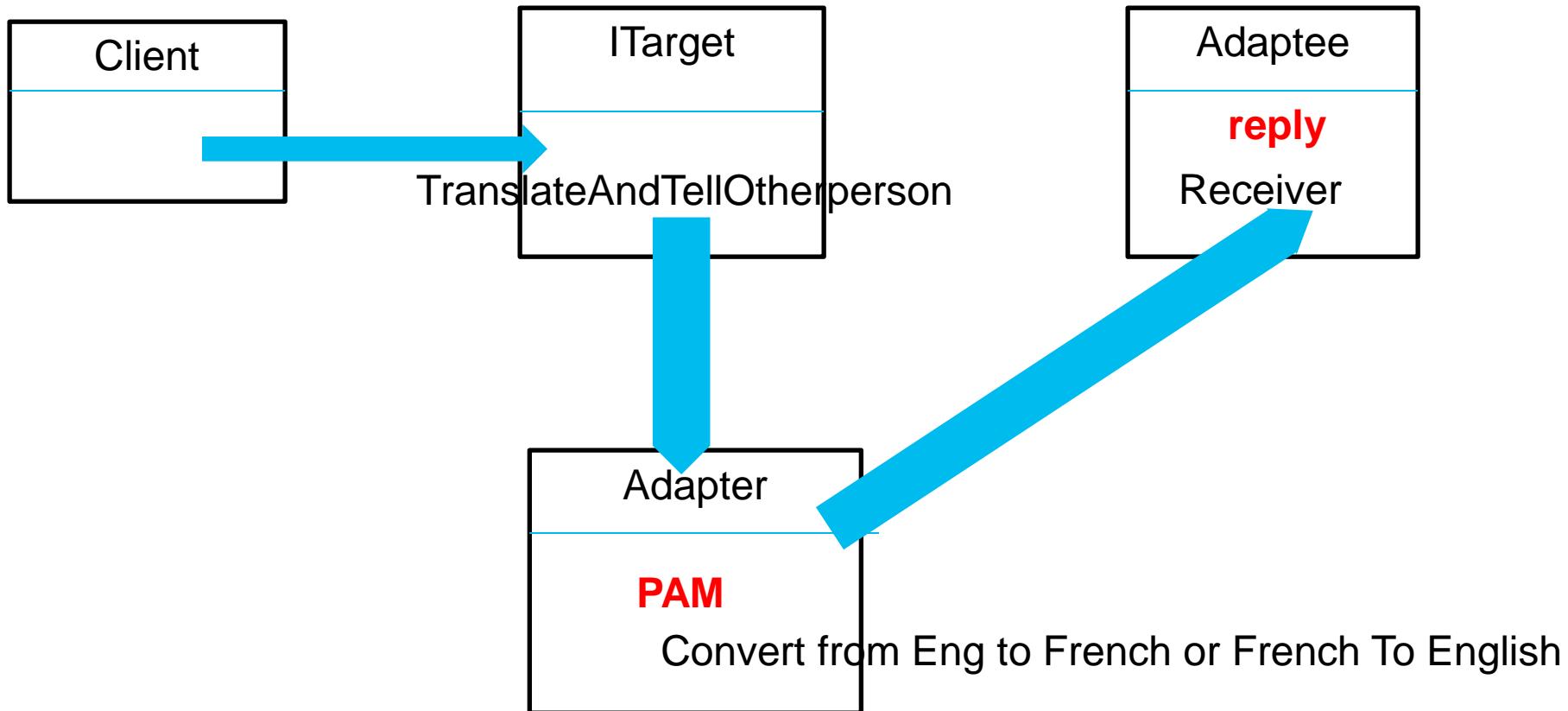


How can we make these two incompatible systems work together?





Adapter Pattern



When to use the Adapter Design Pattern in Real-Time Applications?



- Integration with Third-party or Legacy Systems: When your application needs to interact with an external system or a legacy system, and the interfaces of the external systems are not compatible with your application's interfaces.
- Reusing Existing Code: If you have existing classes with functionality that you need to use, but their interfaces don't match the ones your system currently uses, an adapter can bridge this gap.

When to use the Adapter Design Pattern in Real-Time Applications?



- Creating a Common Interface for Different Classes:
When you have several classes with different interfaces but want to treat them uniformly through a common interface.
- Supporting Multiple Data Sources: When your application needs to handle data from different sources (like databases, file systems, web services) but wants to process them in a uniform manner.

When to use the Adapter Design Pattern in Real-Time Applications?



- Testing and Mocking: Adapters can be used to create stubs or mocks for unit testing, especially when the actual objects are cumbersome to use in a test environment (like database connections or external services).
- Providing Backward Compatibility: When updating an application or library, adapters can be used to maintain backward compatibility with the old versions of APIs or data models.
- Cross-Platform Compatibility: In scenarios where you need to provide support for different platforms or environments while keeping the rest of the application code consistent.



Bridge Pattern

- As per the Gang of Four definitions, the Bridge Design Pattern Decouples an abstraction from its implementation so that the two can vary independently.
- This pattern involves an interface that acts as a bridge between the abstraction class and implementer classes.
- It is useful in scenarios where an abstraction can have several implementations, and you want to separate the implementation details from the abstraction.

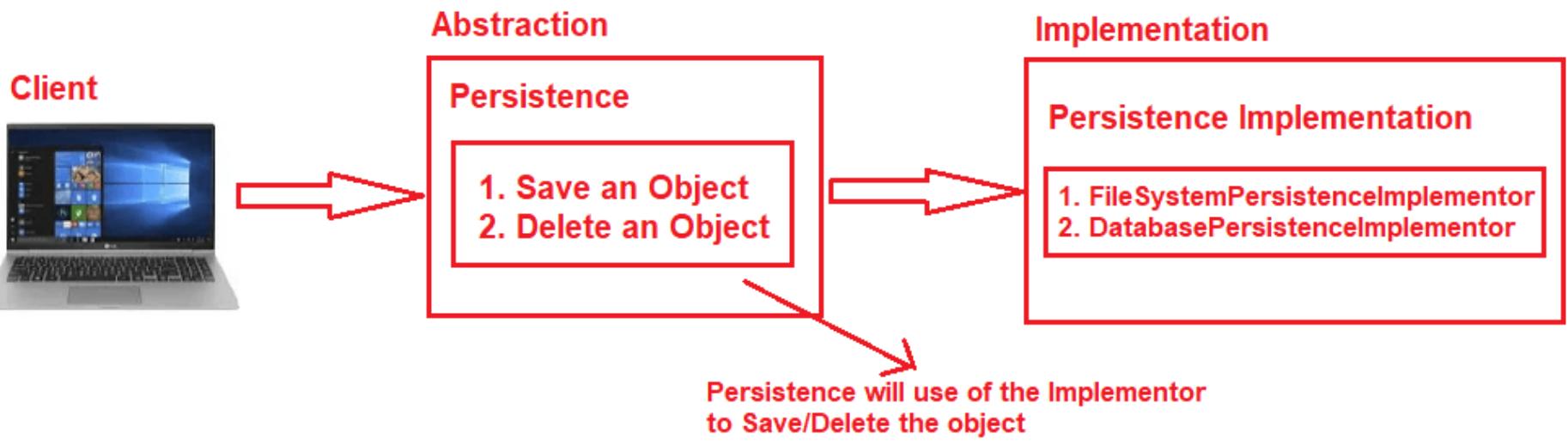


Bridge Pattern

- In the Bridge Design Pattern, there are 2 parts.
 - The first part is the Abstraction, and the second part is the Implementation.
 - The Bridge Design Pattern allows both Abstraction and Implementation to be developed independently, and the client code can only access the Abstraction part without being concerned about the Implementation part.



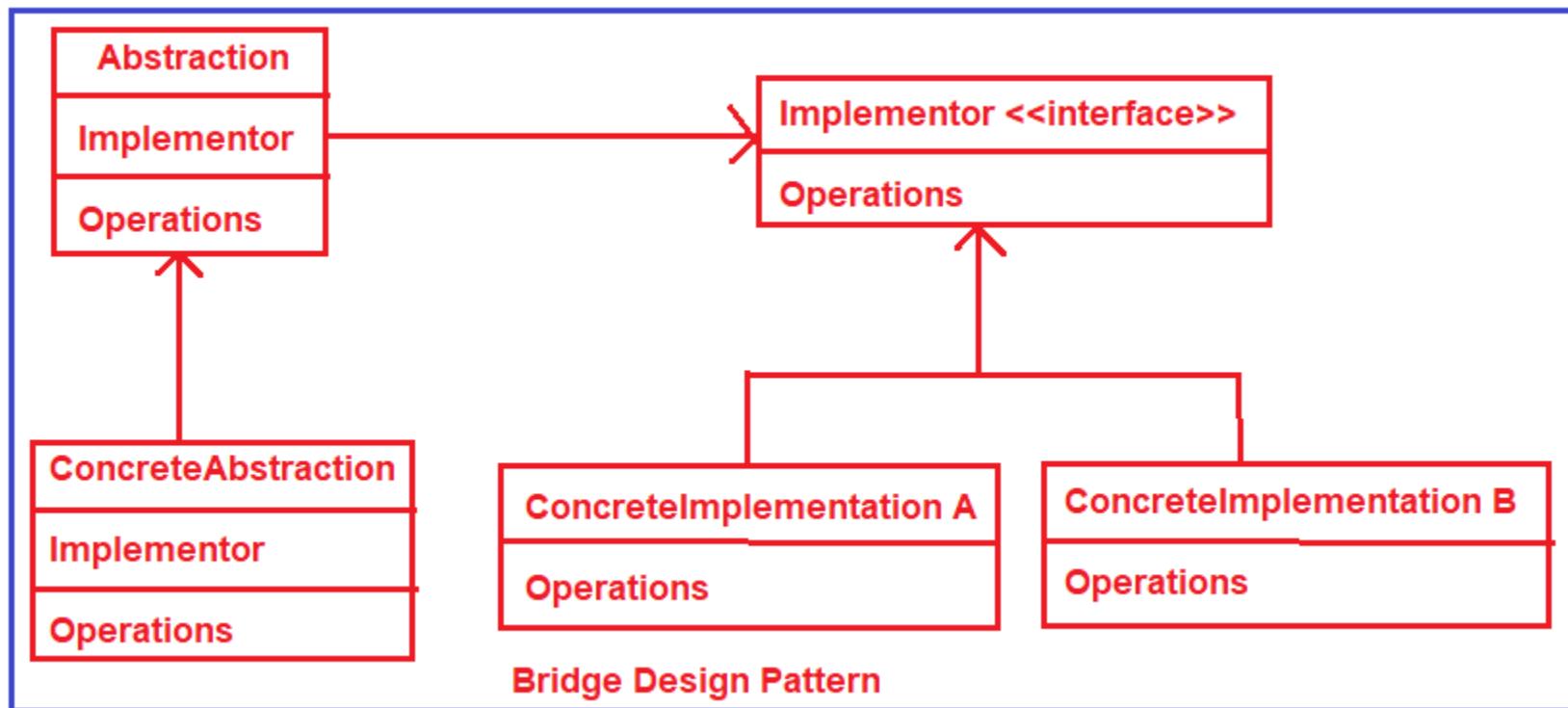
Bridge Pattern



Bridge Pattern



Bridge Pattern





Advantages of Bridge Design Pattern:

- Decoupling: It decouples an abstraction from its implementation, allowing them to vary independently.
- Single Responsibility Principle: It promotes the principle by separating an abstraction from its implementation.
- Flexibility: Increases the flexibility in terms of the framework and its implementation.
- Extensibility: Both the abstractions and implementations can be extended independently.
- Prevents Cartelization: Avoids the ‘cartesian product’ complexity explosion. For example, if you have N abstractions and M implementations, you don’t need N^M classes.

When to Use Bridge Design Pattern in C# Real-Time Applications?



- Abstraction and Implementation Can Vary Independently: When you want to decouple an abstraction from its implementation so that the two can vary independently.
- This is useful in cases where, for instance, the core functionality and the platform-specific details need to be developed and extended separately.
- Changing Implementation at Runtime: If your application needs to switch between different implementations at runtime.
- The Bridge pattern allows you to change the implementation dynamically without altering the abstraction.
- Extending Classes in Separate Dimensions: When you have multiple dimensions in your class hierarchy that need to be extended independently.
- For example, if you have a UI framework, you might want to extend UI controls independently from operating system-specific behaviors.

When to Use Bridge Design Pattern in C# Real-Time Applications?



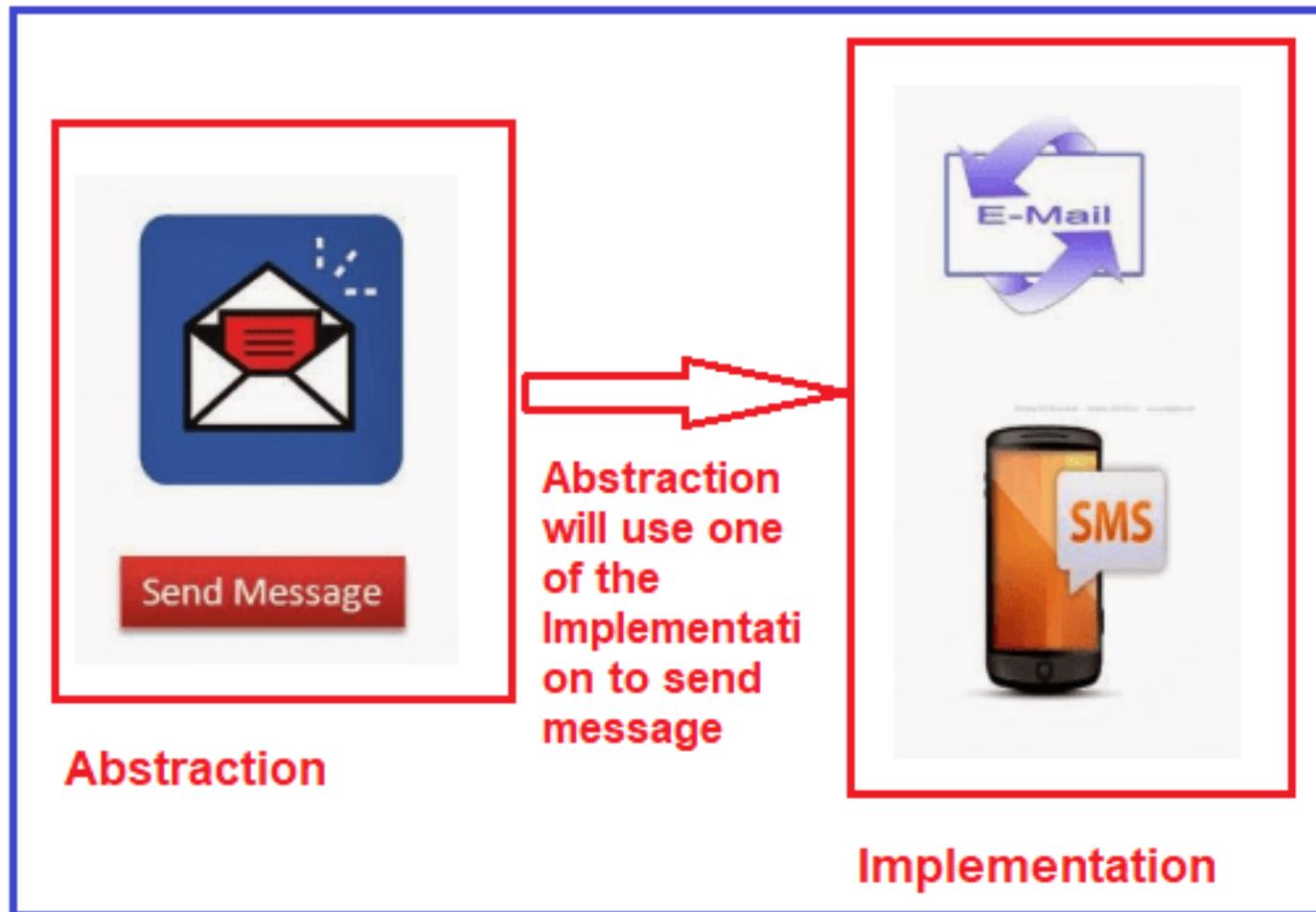
- Avoiding a Permanent Binding to Implementation: In scenarios where a permanent binding between the abstraction and its implementation might limit the flexibility and future scalability of the code.
- Sharing an Implementation Among Multiple Objects: When you need to share an implementation among multiple objects.
- The Bridge pattern allows multiple abstractions to use the same implementation, which can be more efficient.
- Platform Independence: It's particularly useful in cross-platform applications where you want to hide the platform-specific code from the high-level logic.

When to Use Bridge Design Pattern in C# Real-Time Applications?

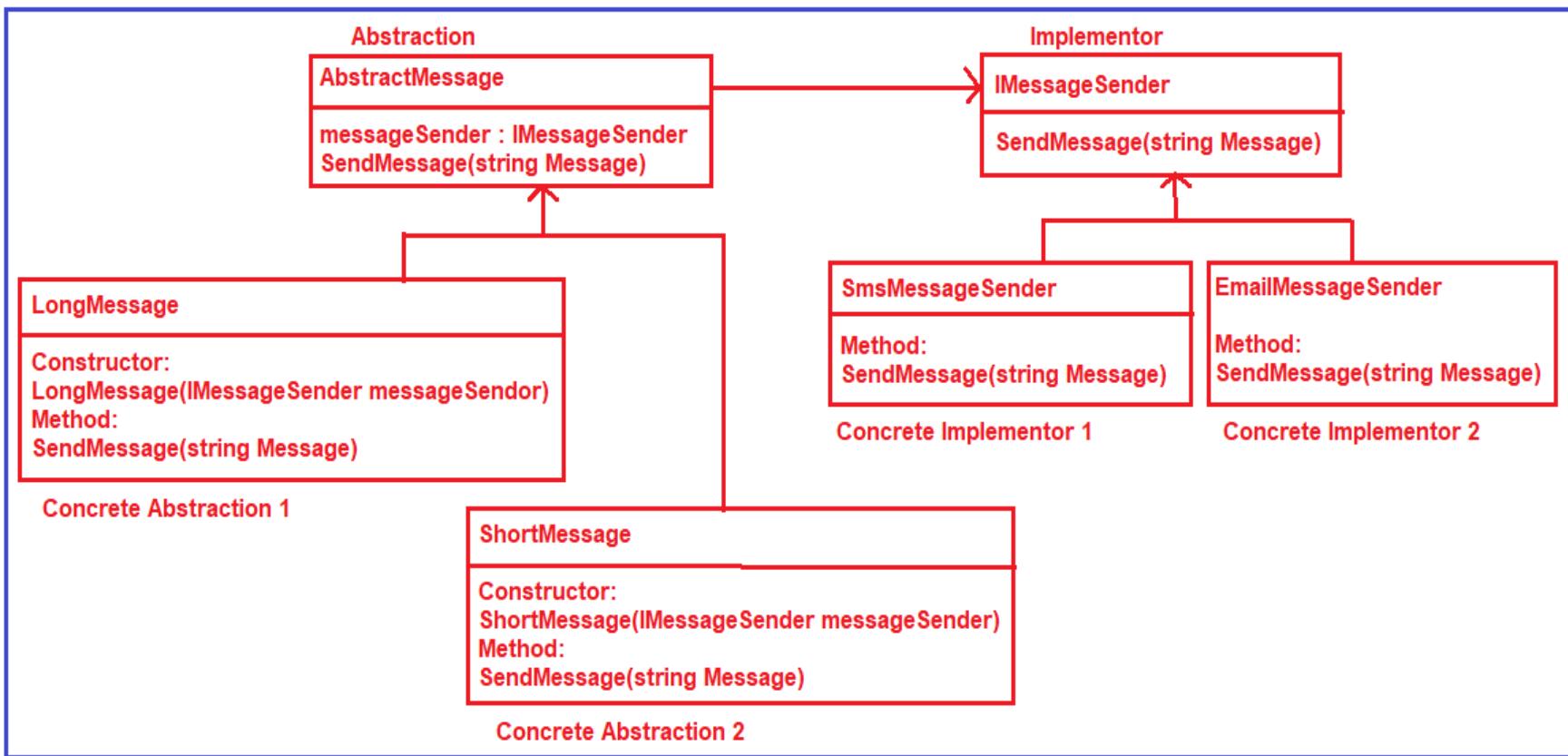


- Preventing Exponential Class Explosion: In cases where a class hierarchy would result in an exponential number of combinations due to the various dimensions that can be extended.
- The Bridge pattern prevents this by separating the hierarchies.
- Long-term Stability of Abstraction and Implementation: When the parts of a system that represent high-level logic (abstraction) and low-level platform details or back-end logic (implementation) are subject to different rates of change or different types of change.

When to Use Bridge Design Pattern in C# Real-Time Applications?



When to Use Bridge Design Pattern in C# Real-Time Applications?

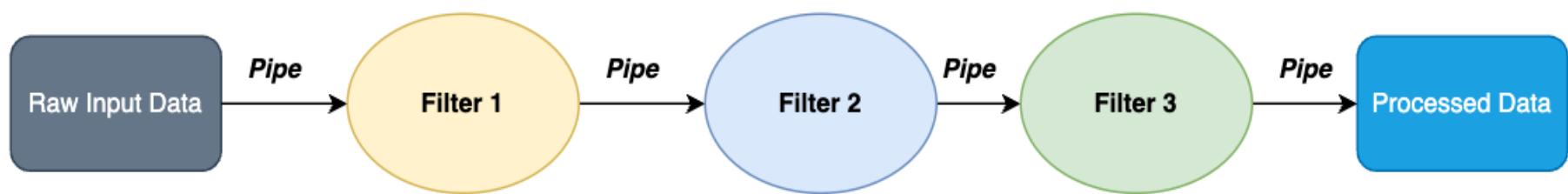




Filter Pattern

- The Pipes and Filters pattern is a component-based architectural design pattern.
- It comprises multiple components known as “Filters”. Each filter is responsible for executing a specific data operation.
- Importantly, these filters intentionally operate in isolation from one another, ensuring that they have no direct knowledge of their position within the pipeline’s sequence.
- These filters are interconnected within the architecture through pipes, which serve as conduits for seamlessly transmitting data from one filter to the next.

Filter Pattern

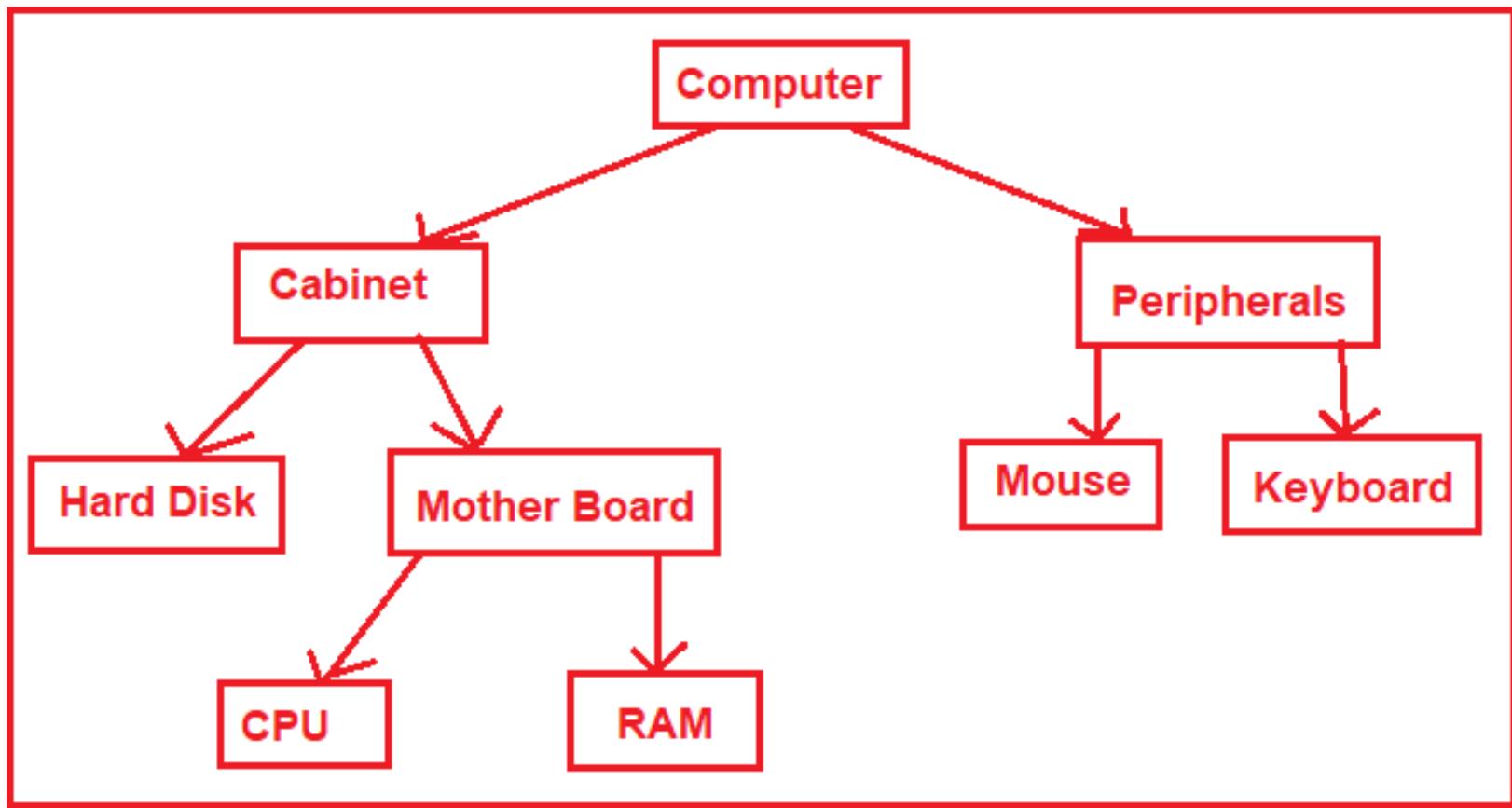




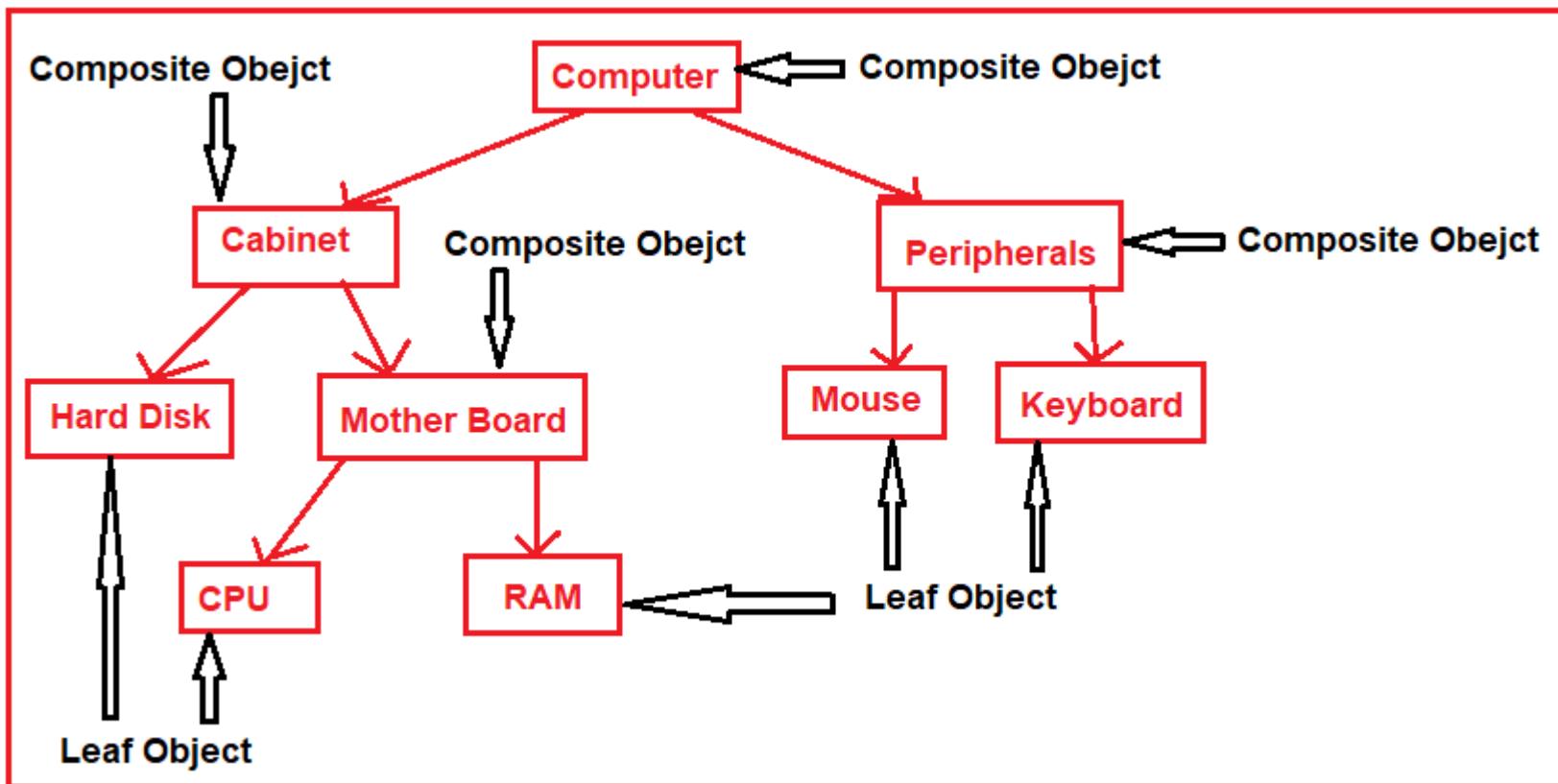
Composite Pattern

- Compose objects into tree structures to represent part-whole hierarchies.
- Composite lets clients treat individual objects and compositions of objects uniformly.
- The Composite Design Pattern is a structural pattern that allows us to compose objects into tree structures to represent part-whole hierarchies.
- This pattern lets clients treat individual objects and compositions of objects uniformly.
- That means the client can access the individual objects or the composition of objects in a uniform manner.
- It's useful for representing hierarchical structures such as file systems, UI components, or organizational structures.

Composite Pattern



Composite Pattern





Advantages of Composite Design Pattern:

- Simplified Client Code: Clients can treat composite structures and individual objects uniformly, simplifying client code.
- Clear Structure: Clearly defines the hierarchy or tree structure of complex objects.
- Ease of Modification: Adding new kinds of components is easy as long as they support the same interface.
- Flexibility in Design: The pattern provides flexibility to compose objects into tree structures to represent part-whole hierarchies.

When to use the Composite Design Pattern in C# Real-Time Applications?



- Hierarchical Tree Structures: When you need to represent a part-whole hierarchy.
- The pattern is ideal for situations where you are dealing with a tree structure with individual objects and compositions of objects treated uniformly.
- Treating Individual and Composite Objects Uniformly: If you want to treat both individual objects and their compositions in the same way.
- This is useful when you want to ignore the difference between compositions of objects and individual objects.
- Simplifying Client Code: It's useful for simplifying client code, as it can treat composite structures and individual objects similarly, simplifying the client's interaction with the structure.

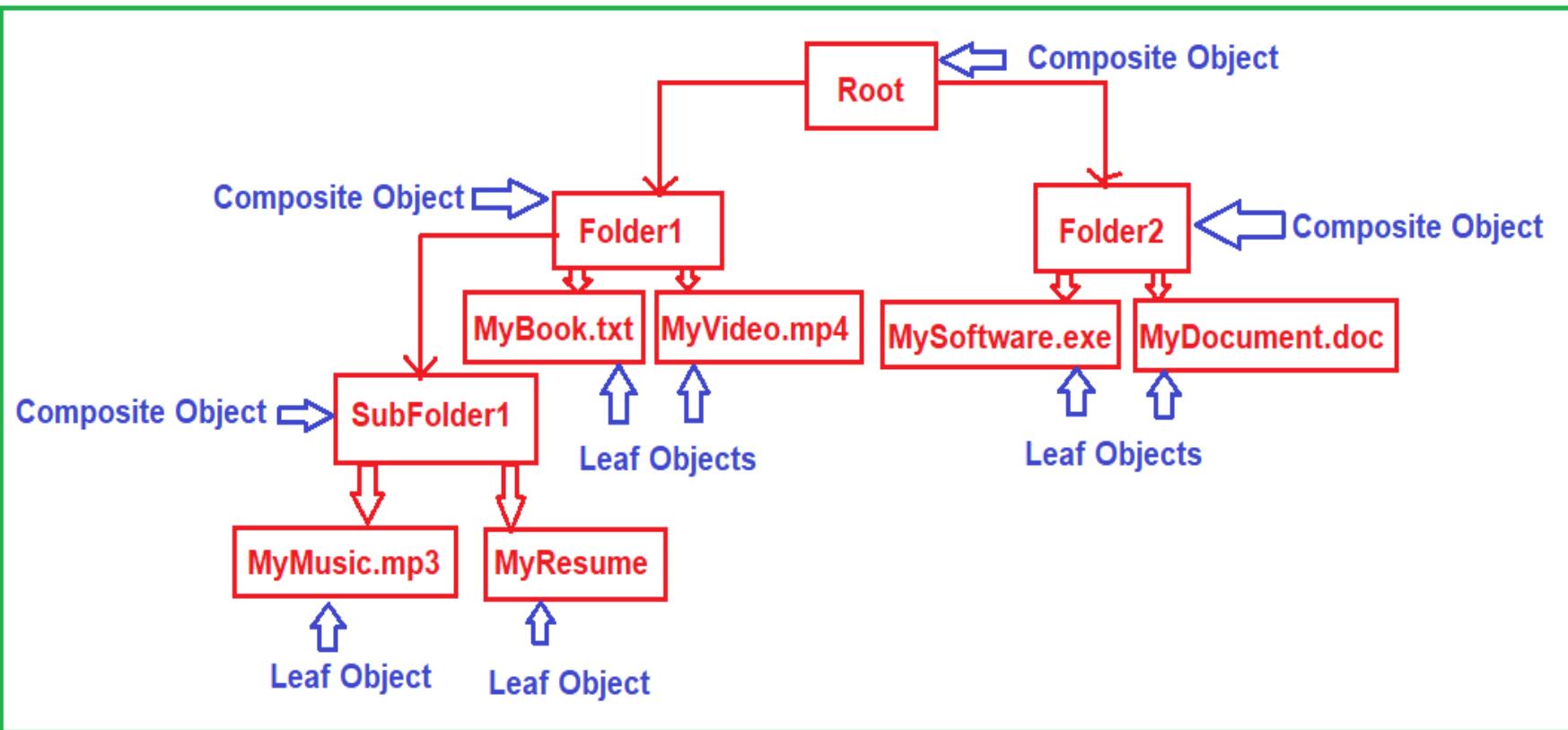
When to use the Composite Design Pattern in C# Real-Time Applications?



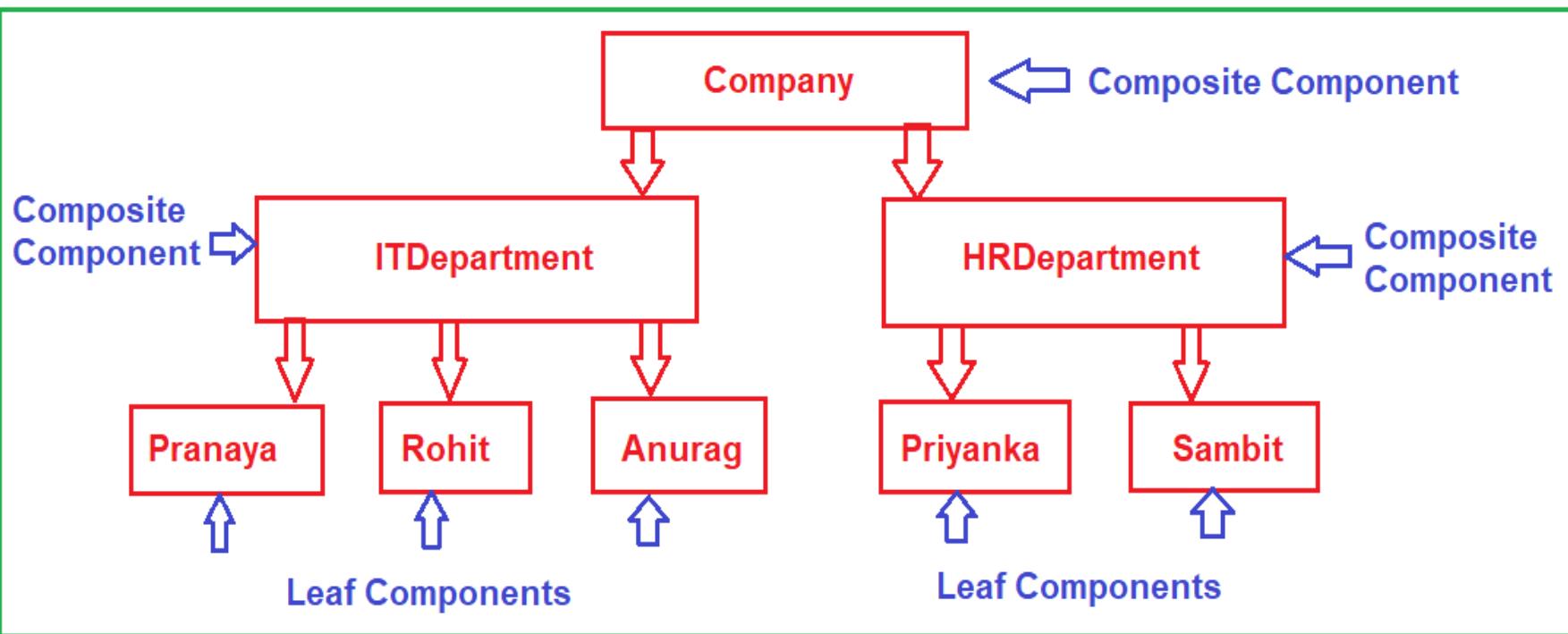
- Dynamic Configuration: When the configuration of the object structure can change at runtime, the Composite pattern allows for the dynamic addition or removal of components in the tree structure.
- Graphic User Interfaces: In GUI development, you might have complex widgets composed of simpler components but want to treat them all as part of a uniform interface.
- File System Representations: Representing file and directory structures, where directories can contain files and other directories, and you want to treat them all as a single file system entity.



Real Time Composite Design Pattern in C#



Real Time Composite Design Pattern in C#





Decorator Pattern

- The Decorator Design Pattern in C# allows us to dynamically add new functionalities to an existing object without altering or modifying its structure, and this design pattern acts as a wrapper to the existing class.
- That means the Decorator Design Pattern dynamically changes the functionality of an object at runtime without impacting the existing functionality of the object.
- In short, this design pattern adds additional functionalities to the object by wrapping it.
- A decorator is an object that adds features to another object.

Decorator Pattern



Car without Engine



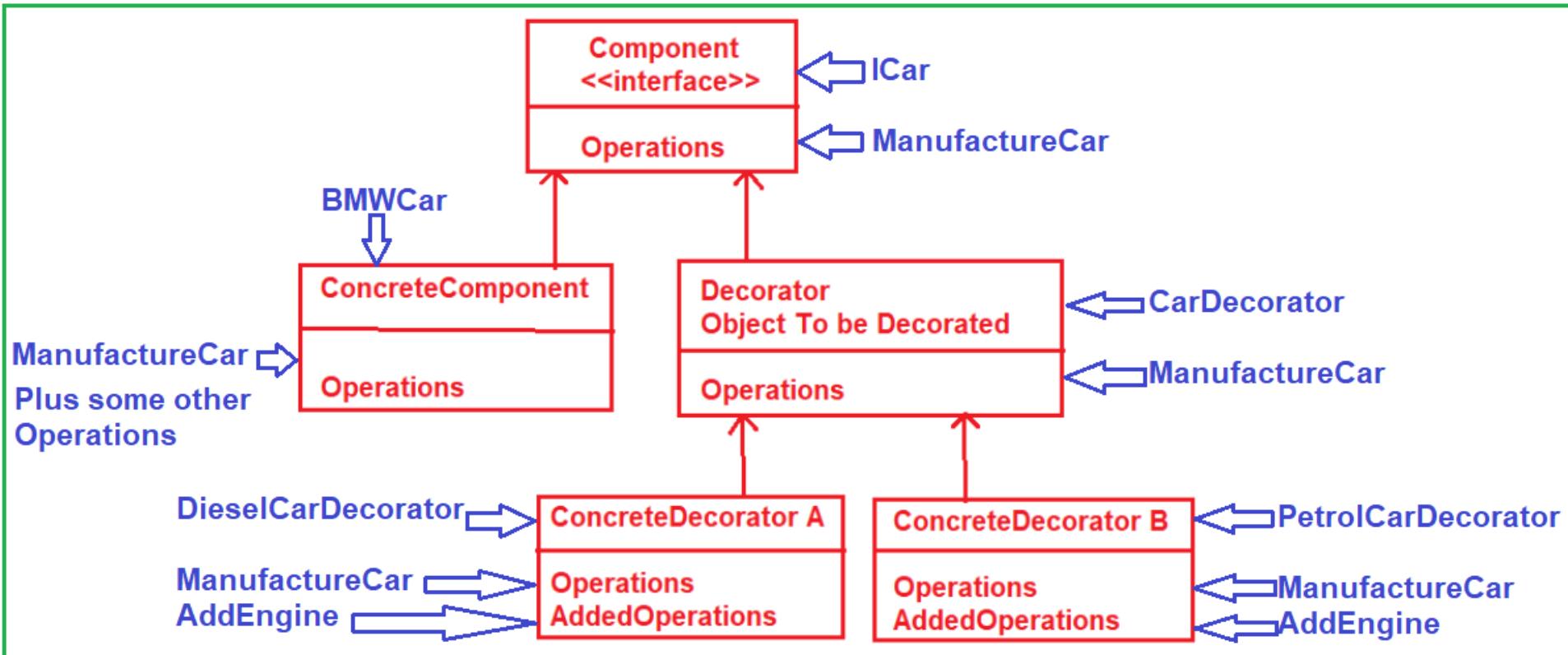
Car with Petrol Engine



Car with Diesel Engine



Decorator Pattern





Advantages of Decorator Design Pattern:

- Flexibility: Provides a more flexible way to add object responsibilities than static inheritance.
- Dynamic Responsibilities: You can add or remove responsibilities from an object at runtime.
- Functionality Layering: Allows for the layering of functionalities.
- Each decorator adds its behavior before and/or after delegating the task to the base component.
- Single Responsibility Principle: Promotes smaller, cohesive classes that each handle one responsibility.

When to Use Decorator Design Pattern in C#?



- Adding Responsibilities to Objects Dynamically: When you need to add additional responsibilities or functionalities to an object at runtime without altering its structure.
- This is particularly useful in situations where subclassing would exponentially increase the number of classes.
- Extending Functionality of Classes in a Scalable Way: If you have a requirement to extend the functionality of classes in a scalable manner, using inheritance would be impractical due to the large number of subclasses it would create.

When to Use Decorator Design Pattern in C#?



- Modifying Specific Objects: When you want to modify class instances without affecting other instances of the same class.
- Decorators provide a flexible alternative to subclassing for extending functionality.

When to Use Decorator Design Pattern in C#?



- Combining Behaviors: In scenarios where behaviors need to be combined, you want to allow an easy way to mix and match these behaviors.
- This pattern allows for creating several different combinations of behaviors at runtime.
- Maintaining Open/Closed Principle: In cases where you need to adhere to the Open/Closed Principle, where classes are open for extension but closed for modification.
- Decorators provide a way to extend the behavior of a class without modifying the existing code.

When to Use Decorator Design Pattern in C#?



- **Avoiding Complex Hierarchies:** If you're dealing with a situation where a class hierarchy would become too complex or unwieldy with subclasses.
- **Decorators can provide functionality extension without the need for a deep inheritance hierarchy.**
- **Runtime Flexibility:** When the capabilities of objects need to be determined at runtime rather than compile-time.
- **Decorators can dynamically add or remove responsibilities from an object, offering more flexibility than static inheritance.**

When to Use Decorator Design Pattern in C#?



Plain Pizza



Pizza Decorator

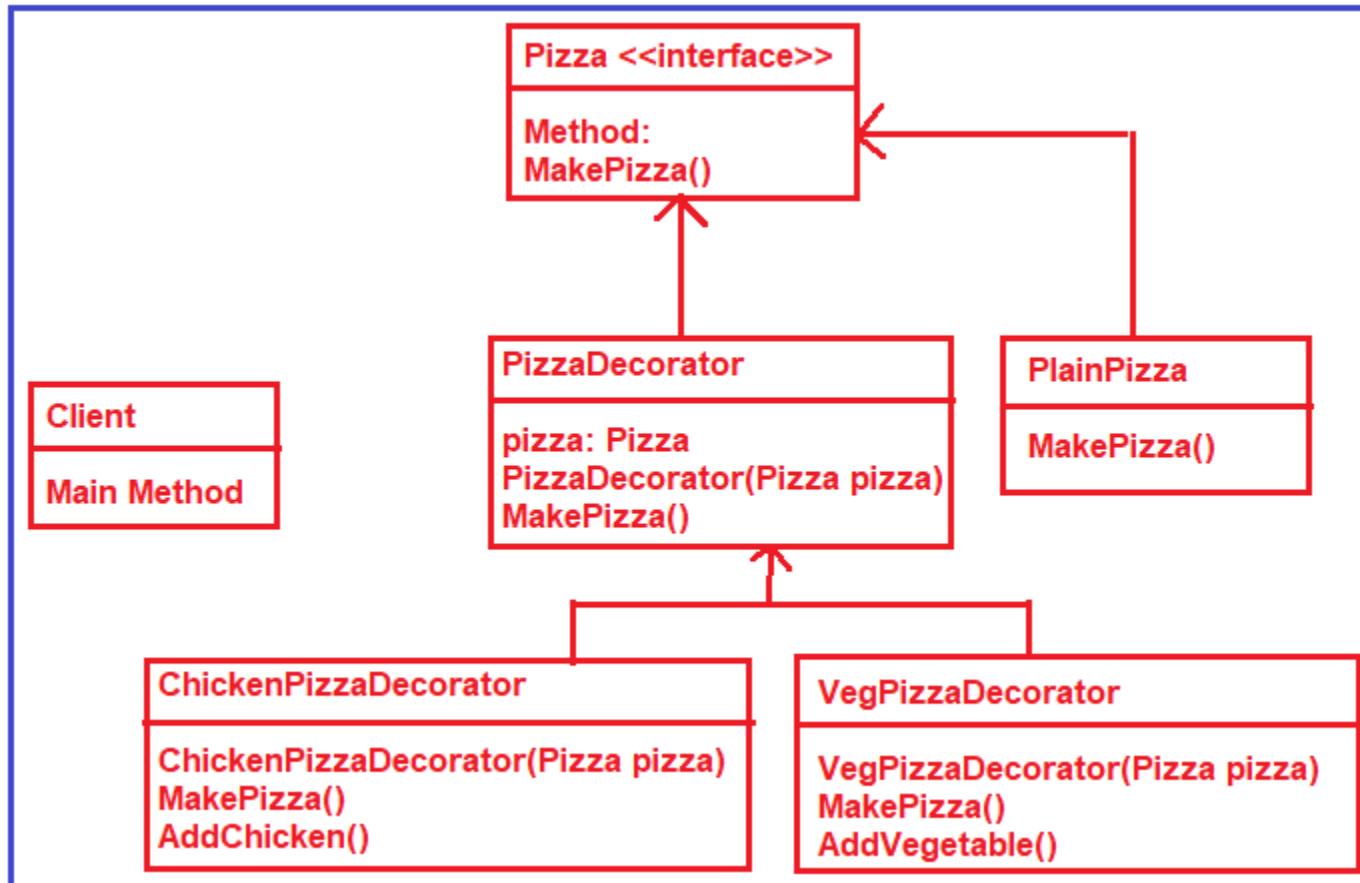


Vegetable Pizza



Chicken Pizza

When to Use Decorator Design Pattern in C#?





Façade Pattern

- Facade Design Pattern states that you need to provide a unified interface to a set of interfaces in a subsystem.
- The Facade Design Pattern defines a higher-level interface that makes the subsystem easier to use.
- The Facade Design Pattern is a structural pattern that provides a simplified interface to a complex system of classes, libraries, or frameworks.



Façade Pattern

- The primary goal of the Facade pattern is to present a clear, simplified, and minimized interface to the external clients while delegating all the complex underlying operations to the appropriate classes within the system.
- The Facade (usually a wrapper) class sits on the top of a group of subsystems and allows them to communicate in a unified manner.



Façade Pattern

- Façade means the Face of the Building. Suppose you created one building.
- The people walking outside the building can only see the walls and glass of the Building.
- The People do not know anything about the wiring, the pipes, the interiors, and other complexities inside the building.
- That means the Façade hides all the complexities of the building and displays a friendly face to people walking outside the building.

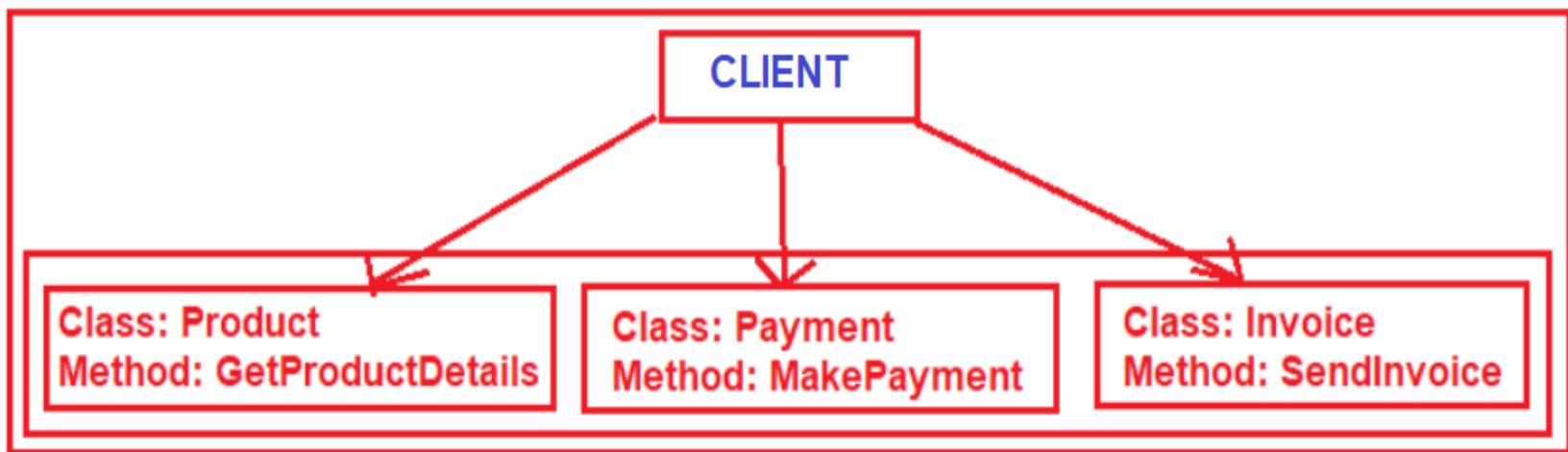
Understanding Facade Design Pattern in C# with one Real-Time Example:



- Identify Complex Subsystems: First, identify the complex parts of your system that need simplification.
- These could be complex libraries or systems with multiple interacting classes.
- Create a Facade Class: Design a facade class that provides a simple interface to the complex subsystems.
- Delegate Calls to Subsystems: The facade should delegate the client requests to the appropriate objects within the subsystem.
- The facade should handle all the intricacies and dependencies of the subsystems.
- Client Code Interaction: The client interacts with the system through the facade, simplifying its use of the complex subsystems.

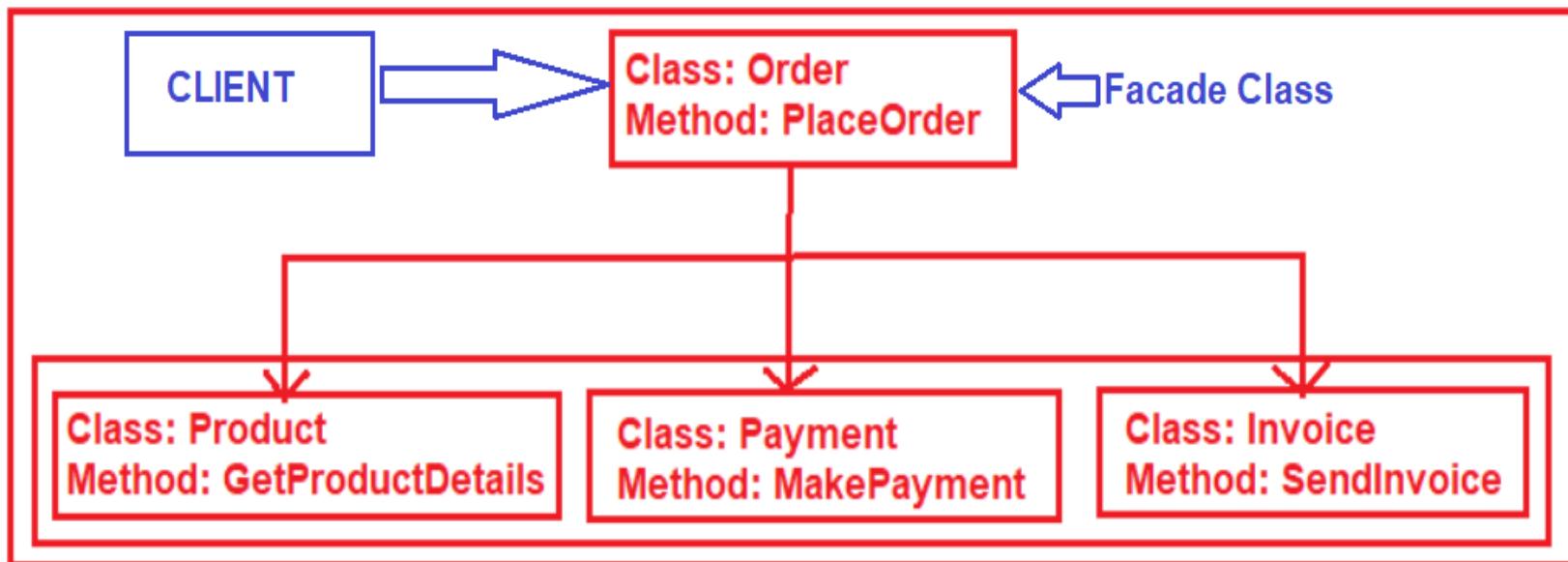


System without Facade



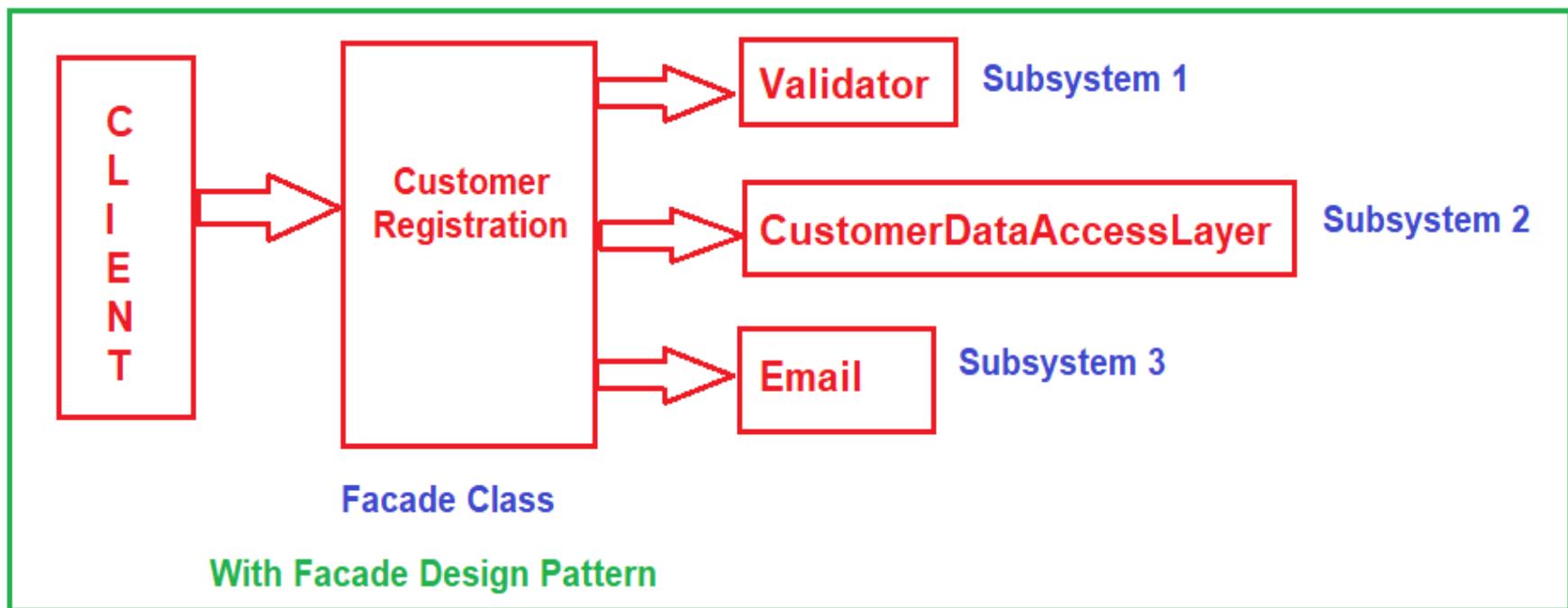


System with Facade





System with Facade





Flyweight Design Pattern in C#

- The Flyweight Design Pattern reduces the number of objects created, decreases memory footprint, and increases performance.
- It's especially useful when many objects share some common properties.
- That means the Flyweight Design Pattern is used when there is a need to create many objects of almost similar nature.
- Many objects means it consumes a large amount of memory, and the Flyweight Design Pattern provides a solution for reducing the load on memory by sharing objects.



Flyweight Design Pattern in C#

- For example, you have one image, and you want thousands of copies of that image.
- There are two ways to achieve that.
 - In the first approach, we can get the printouts 1000 times that image.
 - In the second approach, we can get a printout of that image, then we can use that printout, and then we can take 999 xeroxes of that image.

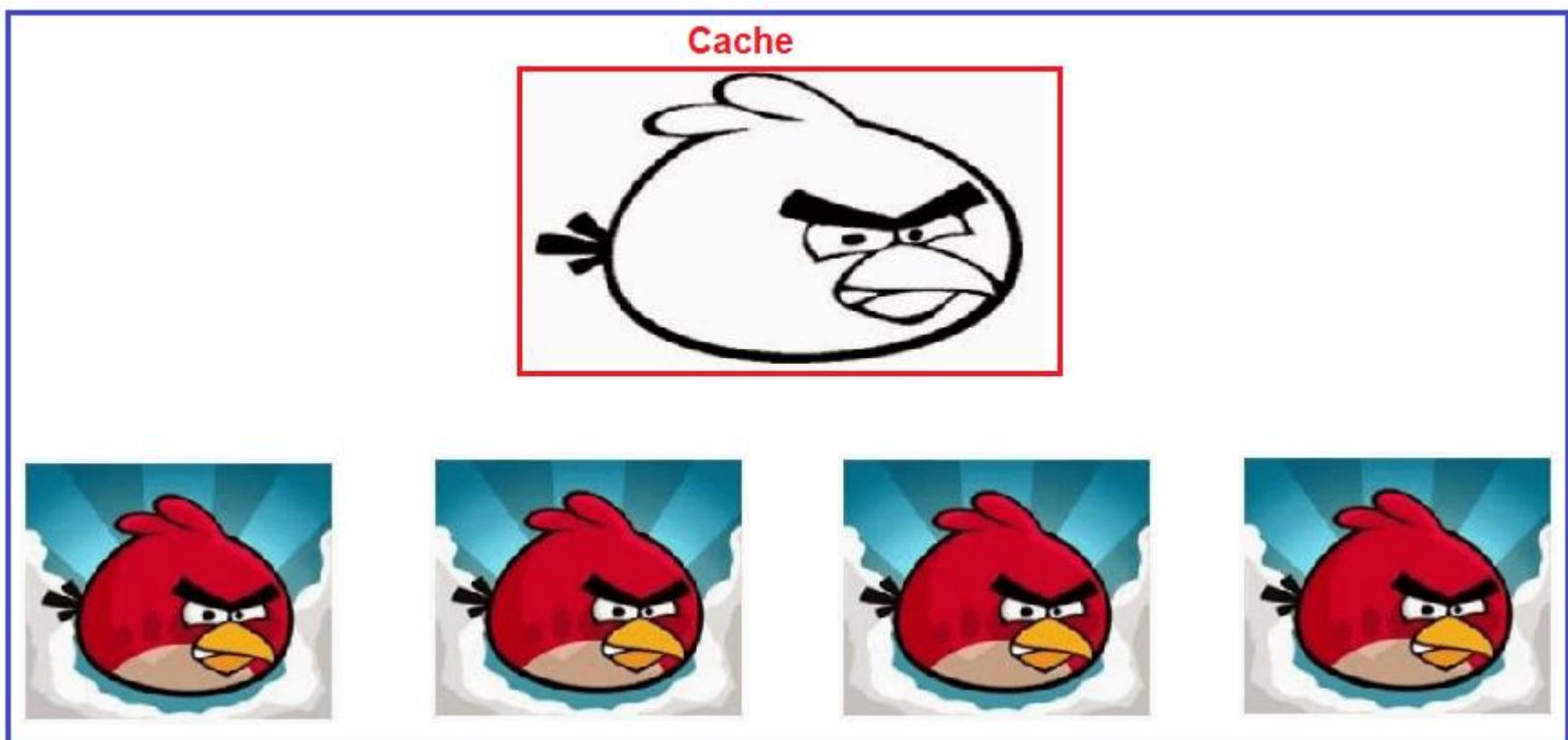


Flyweight Design Pattern in C#

- Suppose the printout for one image is 2 USD.
- Then the total amount required is $1000 * 2 = 2000$ USD.
- If the Xerox price is 1 USD, then the total amount required is $999 * 1 = 999$ USD, and one printout is 2 USD, so a total of 1001 USD.
- So, we can save much amount if we follow the second approach.
- This is also the same in programming. We can achieve this by using the Flyweight Design Pattern in C#.



Flyweight Design Pattern in C#



When to use the Flyweight Design Pattern in Real-Time application?



- Efficient Resource Management: When your application needs to manage many similar objects, keeping an instance of each object is resource-intensive.
- The Flyweight pattern helps in sharing objects to reduce the memory footprint.
- Shared State Objects: In situations where objects can have shared states (intrinsic state) that can be factored out and shared among multiple objects.
- Flyweight allows for sharing common parts to reduce the system's memory usage.
- High Memory Usage Due to Object Quantity: If your application risks using a large amount of memory due to the sheer volume of objects, the Flyweight pattern can reduce memory consumption by sharing objects instead of creating new ones.

When to use the Flyweight Design Pattern in Real-Time application?



- Performance-Critical Applications: In performance-critical applications, the overhead of object creation and garbage collection needs to be minimized, especially in constrained environments such as games or mobile applications.
- Immutable Object Requirements: When dealing with immutable objects, their state cannot change after they are constructed.
- Flyweight makes it easier to manage and share these immutable objects.
- Graphical Applications: Commonly used in graphical applications, like graphic editors or games, where many objects (like trees, bullets, or characters) look similar but might differ in some attributes (like position or size).

When to use the Flyweight Design Pattern in Real-Time application?



- Text Processing: In text processing applications where you need to represent each character in a document, using a flyweight can significantly reduce memory usage, as each character type can be represented as a single object.
- UI Controls: When dealing with user interface elements, similar objects (like buttons, icons, or menu items) are used repeatedly across the application.

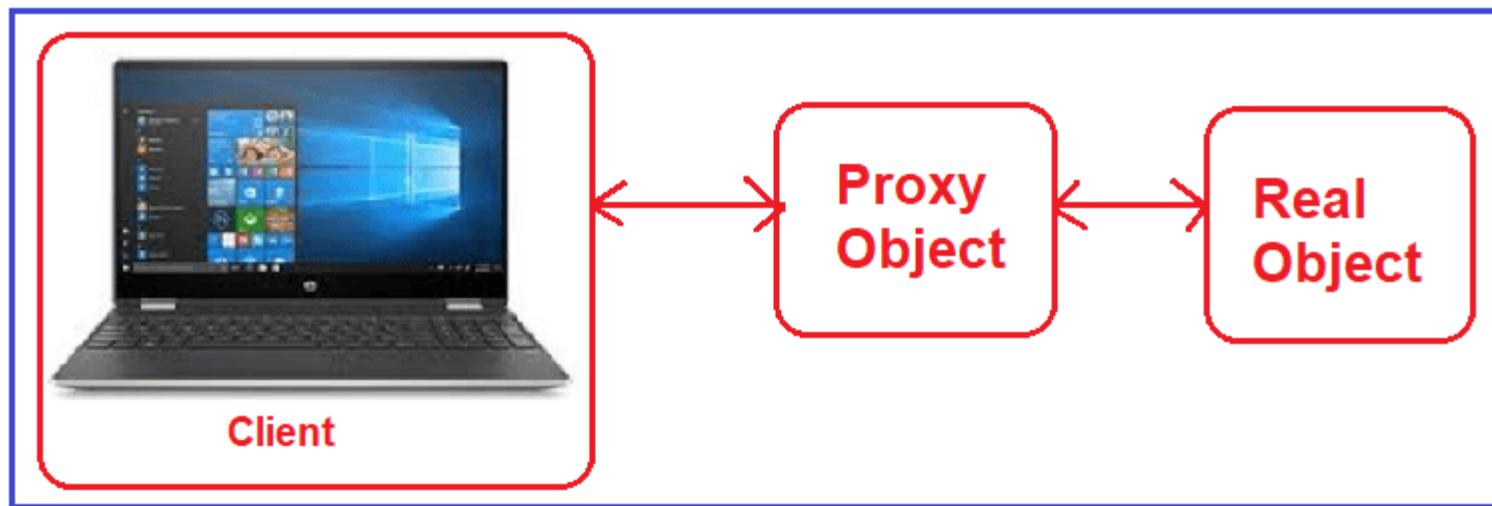


Proxy Pattern

- Proxy Design Pattern provides a surrogate (act on behalf of another) or placeholder for another object to control access to it.
- The Proxy Design Pattern allows us to create a class that represents the functionality of other classes.
- The proxy could interface with anything, such as a network connection, a large object in memory, a file, or other resources that are expensive or impossible to duplicate.
- We can also say that the Proxy is the object the client calls to access the real object behind the scene.
- Proxy means in place of or on behalf of. That means, In the Proxy Design Pattern, a class represents the functionality of another class.



Proxy Pattern





Types of Proxy Pattern

- Virtual Proxy: A virtual proxy is a placeholder for “expensive to create” objects. The real object is only created when a client first requests or accesses the object.
- Remote Proxy: A remote proxy provides local representation for an object that resides in a different address space.
- Protection Proxy: A protection proxy controls access to a sensitive master object. The surrogate object checks that the caller has the access permissions required before forwarding the request.

Types of Proxy Pattern

Anurag



Real Object





Types of Proxy Pattern



When to use the Proxy Design Pattern in C# Real-Time Applications?



- Adding security access to an existing object. The proxy will determine if the client can access the object or not.
- Simplifying the API of a complex object. The proxy can provide a simple API so that the client code does not have to deal with the complexity of the object of interest.
- Providing interfaces for remote resources such as web service or REST resources.
- Coordinating expensive operations on remote resources by asking the remote resources to start the operation as soon as possible before accessing the resources.
- Adding a thread-safe feature to an existing class without changing the existing class code.

Advantages of Virtual Proxy Design Pattern in C#

- Lazy Initialization: One of the main advantages of using a virtual proxy is that it can create objects on demand.
- This can improve system performance if creating the object is costly and the object is not always needed.
- Separation of Concerns: The proxy pattern helps keep the code related to access control or other intermediary tasks separated from the actual implementation of the class.
- Memory Efficiency: If you have heavyweight objects that consume a lot of resources, virtual proxies can be useful to delay their instantiation until they are truly necessary.

Advantages of Virtual Proxy Design Pattern in C#

- Flexibility: Proxies can introduce additional functionality like logging, caching, etc., without changing the real object's code.
- Protection: Proxies can control access to the real object, acting as a protective barrier, ensuring that the real object operates within certain constraints.

Disadvantages of Virtual Proxy Design Pattern in C#

- Overhead: Introducing proxies can add an additional layer of complexity and may incur some overhead, especially if not really needed.
- Maintenance: If the interface of the real subject changes, the proxy needs to be updated as well, leading to increased maintenance effort.
- Complexity: Introducing another level of indirection can make the system more complicated and harder to understand for developers unfamiliar with the pattern.

Disadvantages of Virtual Proxy Design Pattern in C#

- Latency: While the lazy initialization can benefit, it can also lead to unexpected latency when the real object is finally accessed.
- This behavior might be undesirable when consistent response times are needed.
- Misuse: It can be overengineering if used without a valid reason.
- Not every object requires a proxy, and weighing the benefits against the complexity introduced is important.

Chain of Responsibility Design Pattern in C#



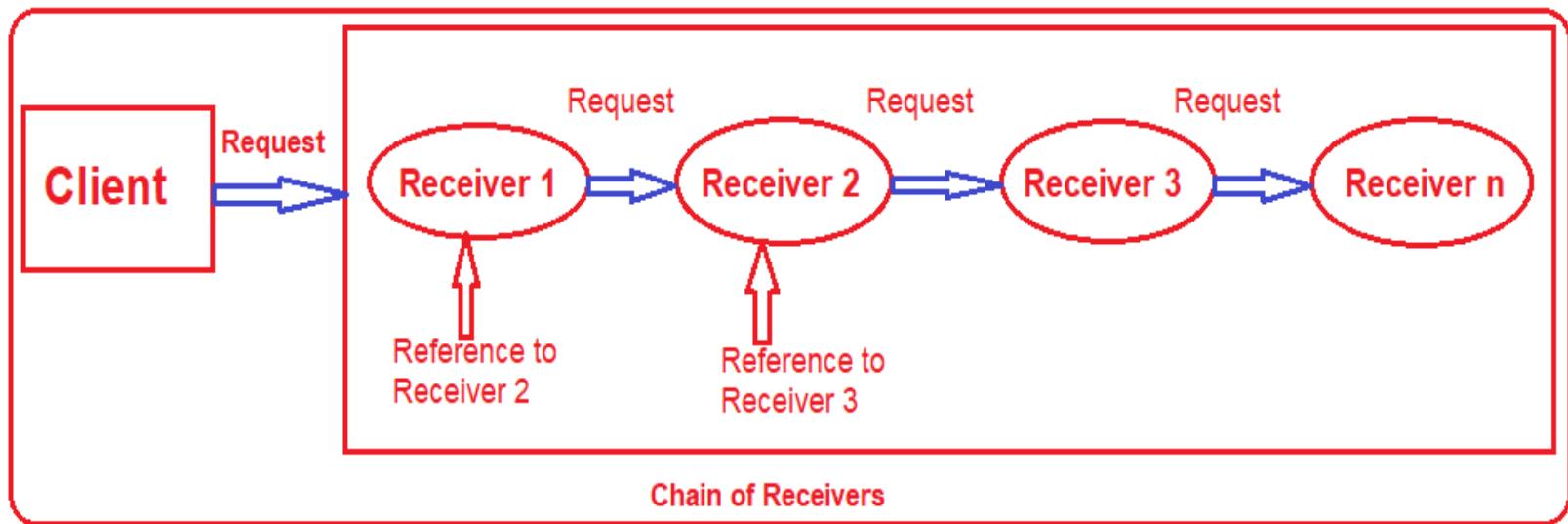
- According to the Gang of Four Definitions, the Chain of Responsibility Design Pattern states, “Avoid coupling the sender of a request to its receiver by giving more than one receiver object a chance to handle the request.”
- Chain the receiving objects and pass the request along until an object handles it”.

Chain of Responsibility Design Pattern in C#



- The Chain of Responsibility Design Pattern is a Behavioral Design Pattern that allows passing requests along a chain of handlers.
- Instead of sending a request directly to a specific receiver, a chain of potential receivers is formed.
- Each handler either processes the request or passes it to the next handler in the chain.
- This pattern allows multiple objects to handle the request without coupling the sender class to the concrete classes of the receivers.

Chain of Responsibility Design Pattern in C#



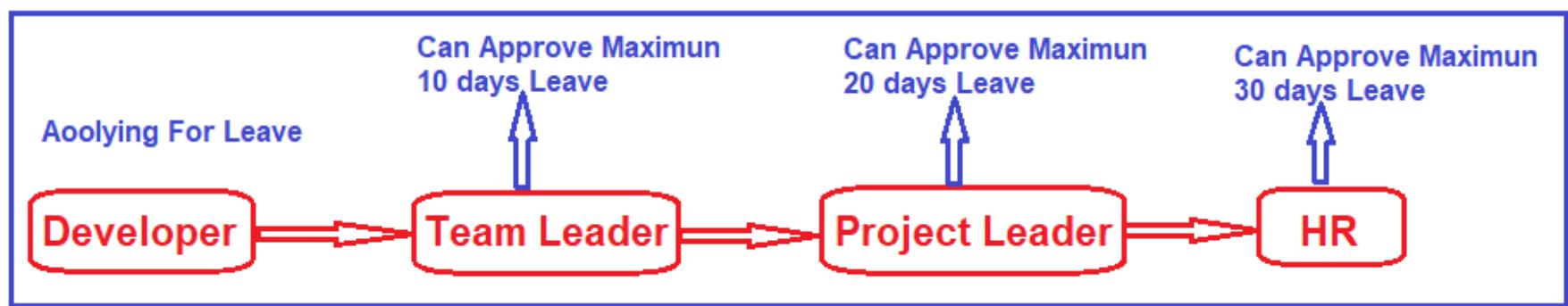


Chain of Responsibility Design Pattern in C#





Chain of Responsibility Design Pattern in C#





Use Cases of Responsibility Design Pattern:

- Multiple Handlers for a Request: When multiple objects can handle a request, but the specific handler isn't known in advance, the request can be passed along a chain of handlers until one handles it.
- Decoupling Request Senders and Receivers: If you want to decouple the sender of a request from its receivers.
- The sender doesn't need to know which part of the chain will handle the request, promoting loose coupling in the system.
- Conditional Handling of Requests: In scenarios where the handling of a request depends on a set of conditions or criteria that only the handlers can evaluate.
- Each handler in the chain can decide whether to process the request or pass it along



Use Cases of Responsibility Design Pattern:

- Dynamic Handling: If the handling logic needs to be changed or reconfigured dynamically at runtime, the Chain of Responsibility allows for adding or removing handlers from the chain.
- Grouping Related Handlers Together: When you want to group several related handlers together.
- For instance, different handlers can perform different validations in a sequence.
- Preventing Direct Coupling: To avoid direct coupling between the sender of a request and its receivers, thereby allowing an object chain to operate independently.



Use Cases of Responsibility Design Pattern:

- Creating an Audit Trail: In some cases, especially for logging or auditing purposes, you might want multiple objects to process the same request, with each object performing its own operation.
- Complex Validation or Authorization Processes: When a request requires a series of validation or authorization steps before it can be processed.
- Each step in this process can be encapsulated in a handler.



Command Pattern

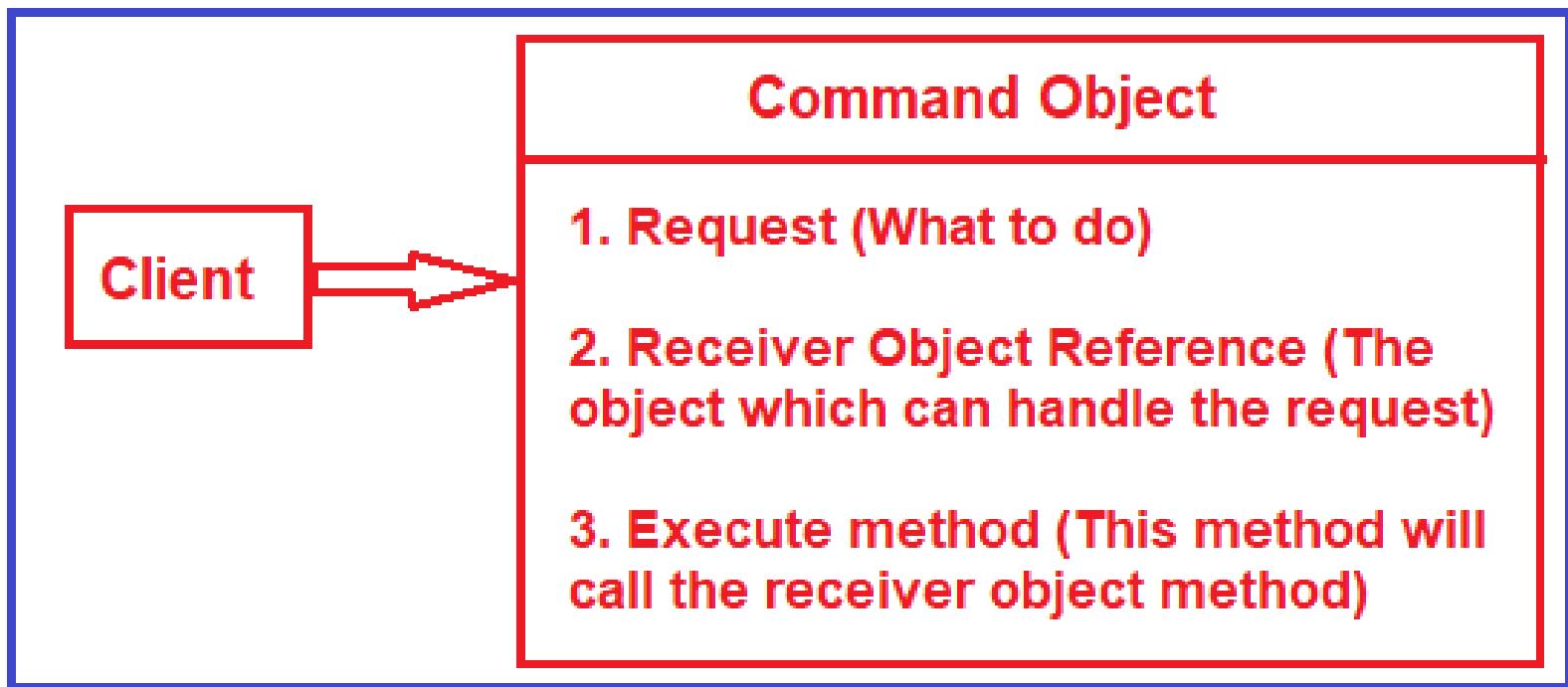
- Command Design Pattern is used to encapsulate a request as an object (i.e., a command) and pass it to an invoker.
- Wherein the invoker does not know how to serve the request but uses the encapsulated command to perform an action.



Command Pattern

- The Command Design Pattern is a Behavioral Design pattern that turns a request into a stand-alone object that contains all information about the request.
- This transformation allows you to parameterize methods with different requests, delay or queue a request's execution, and support undoable operations.
- It's useful in scenarios where you need to issue requests without knowing anything about the operation being requested or the receiver of the request.

Command Pattern



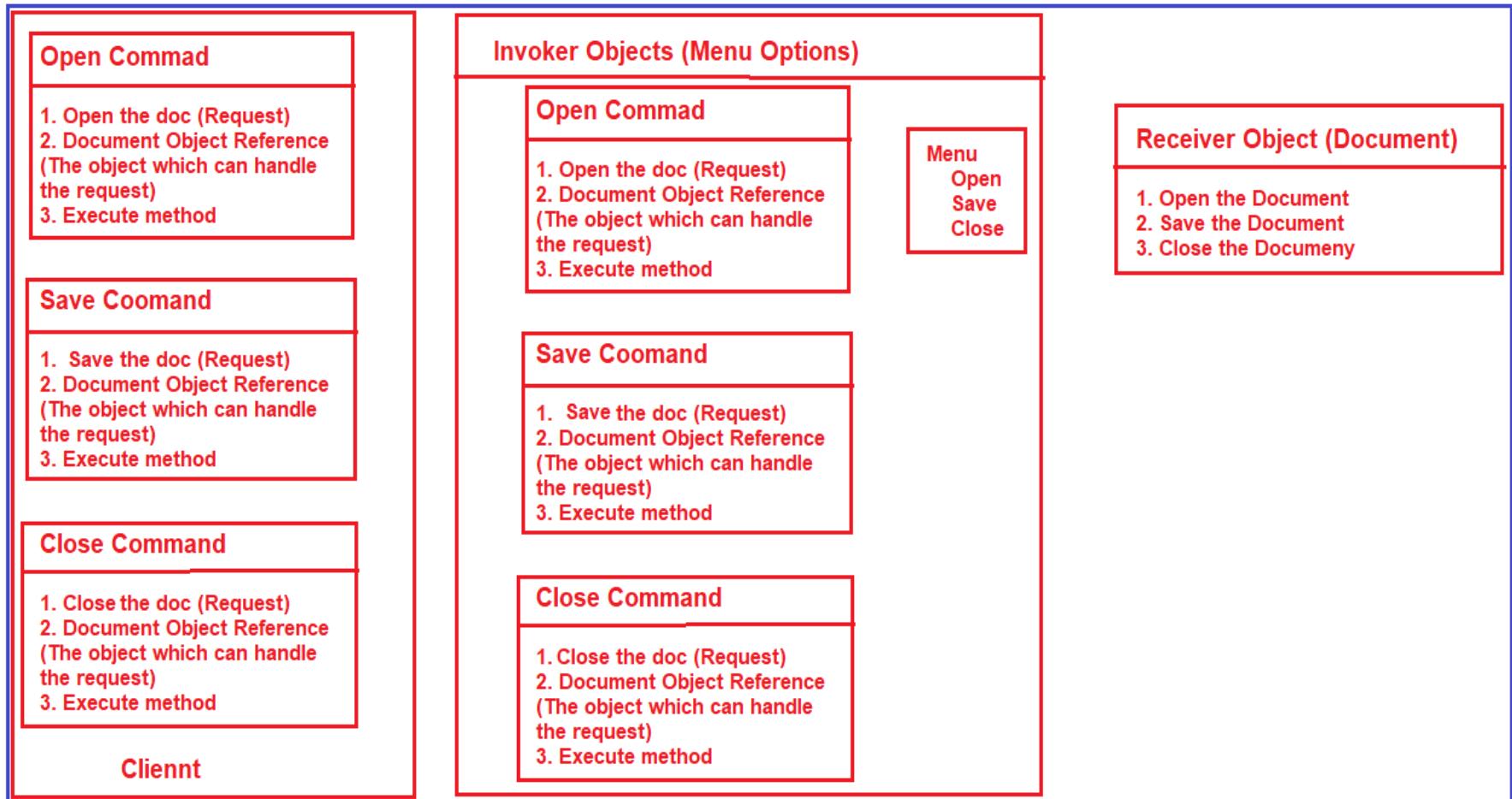
Real-Time Example of Command Design Pattern:

- In a Restaurant, the Waiter will be there. What the Waiter will do is, the Waiter will take an order from the Customer.
- The Customer will tell the Waiter what kind of food he/she wants.
- The Waiter will note it down on a checklist. Then, the Waiter passes the checklist to the cook.
- The Cook is the one who will prepare the food and give it back to the Waiter, and the Waiter will give it back to the Customer.

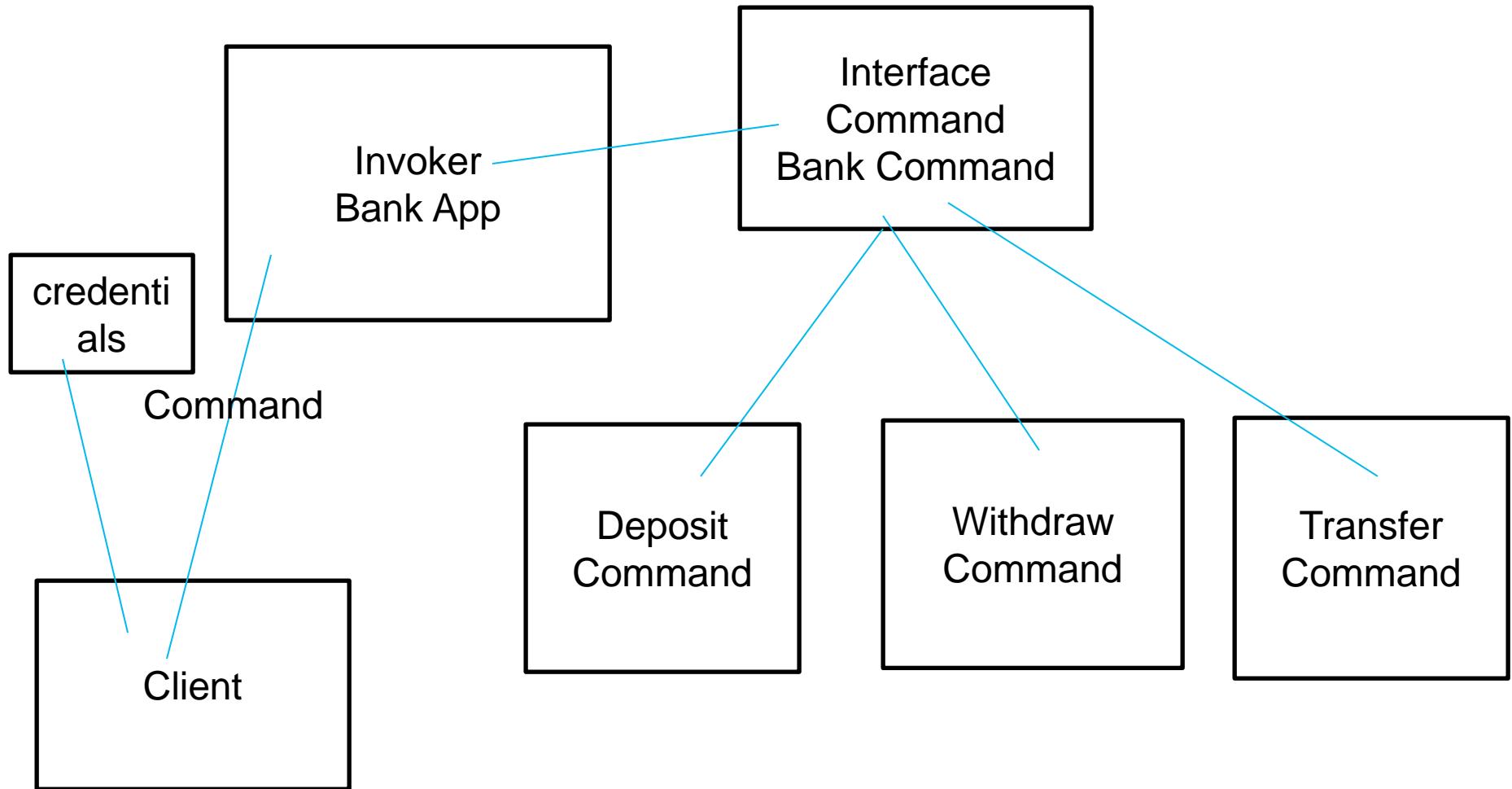
Real-Time Example of Command Design Pattern:

- In this example, the Customer is the Client. Order, i.e., what kind of food the customer wants, is the Command.
- The Waiter is the Invoker.
- The Waiter does not know how to cook the food.
- So, the Waiter passes the request to the Receiver, i.e., the Cook, who will prepare the food and give it back to the Waiter, and the Waiter gives it back to the customer.

Real-Time Example of Command Design Pattern:



Command Design Pattern UML



Real-Time Examples of Command Design Patterns in C#



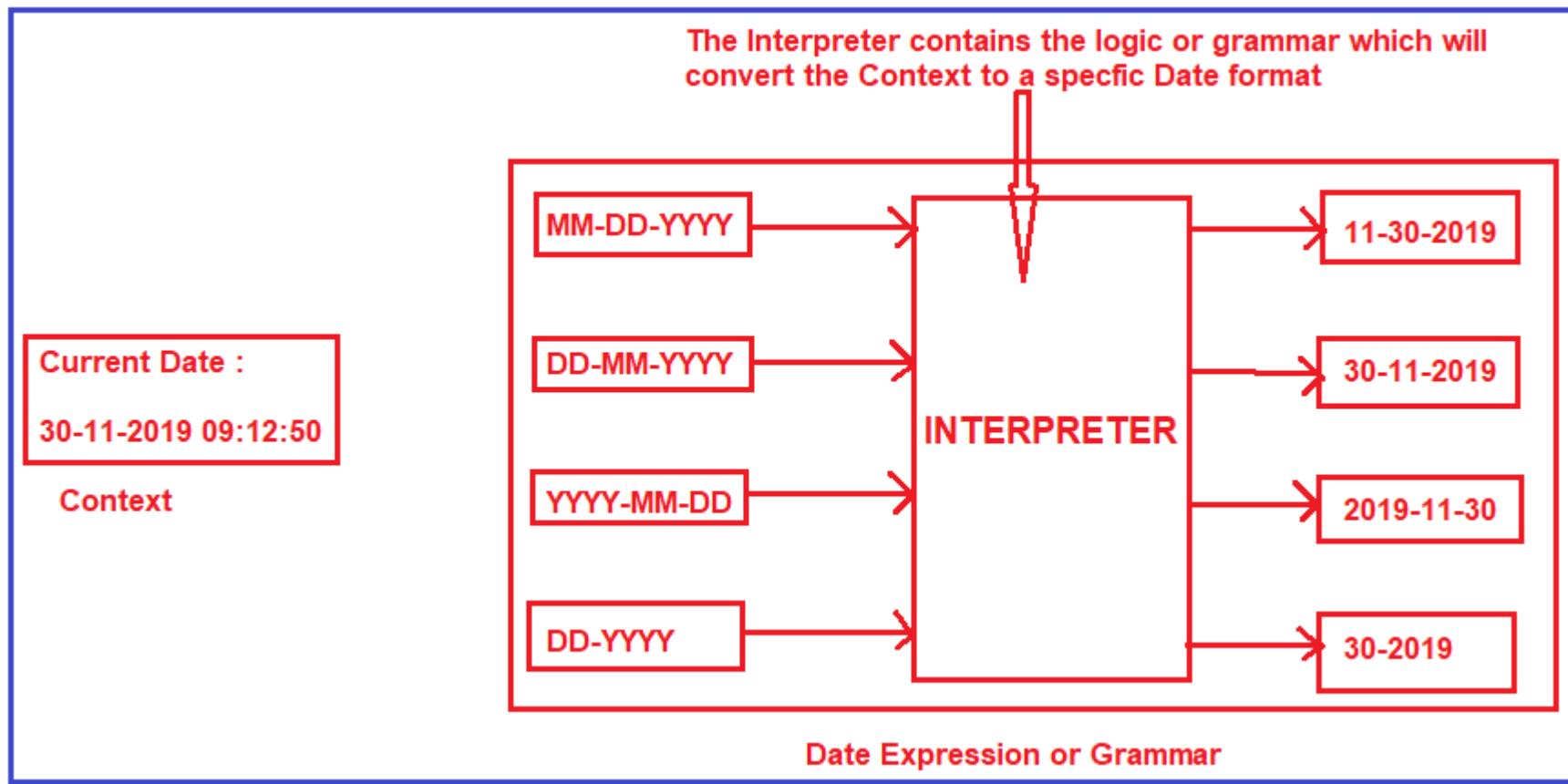
- Remote Control for a Television
- Restaurant Order System
- Smart Home System
- Simple Graphics Editor
- Banking System
- Text Editor
- Music Player Application
- E-Commerce Shopping Cart System



Interpreter Pattern

- The Interpreter Design Pattern is a Behavioral Design Pattern that defines a grammatical representation of a language and provides an interpreter to deal with this grammar.
- The main idea is to define a domain language (a small language specific to your application) and interpret expressions in that language.
- It is useful when interpreting sentences in a language according to a defined grammar.
- That means the Interpreter Design Pattern Provides a way to evaluate language grammar or expression.
- This pattern is used in SQL Parsing, Symbol Processing Engines, etc.

Example to Understand Interpreter Design Pattern





Iterator Pattern

- The Iterator Design Pattern is a Behavioral Design Pattern that allows sequential access to the elements of an aggregate object (i.e., collection) without exposing its underlying representation.
- That means using the Iterator Design Pattern, we can access the elements of a collection sequentially without knowing its internal representations.
- This pattern provides a uniform interface for traversing different data structures.



Iterator Pattern

- The collections in C#, like List, ArrayList, Array, etc., are containers containing many objects.
- In object-oriented programming, the iterator pattern is a design pattern in which an iterator is used to traverse a container and access the elements of the container.

Real-Time Examples of Iterator Design Pattern in C#

- Browsing Music Playlist
- Bookshelf
- Radio Station
- Navigating a Collection of TV Channels
- University Course Registration System
- Social Media Feed
- Restaurant Menu



Mediator Pattern

- Define an object that encapsulates how a set of objects interact with each other.
- Mediator promotes loose coupling by keeping objects from explicitly referring to each other and letting you vary their interaction independently.
- The Mediator Design Pattern reduces the communication complexity between multiple objects.
- This design pattern provides a mediator object, which will be responsible for handling all the communication complexities between different objects.



Mediator Pattern

- The Mediator Design Pattern restricts direct communications between the objects and forces them to collaborate only via a mediator object.
- This pattern is used to centralize complex communications and control between related objects in a system.
- The Mediator object acts as the communication center for all objects.

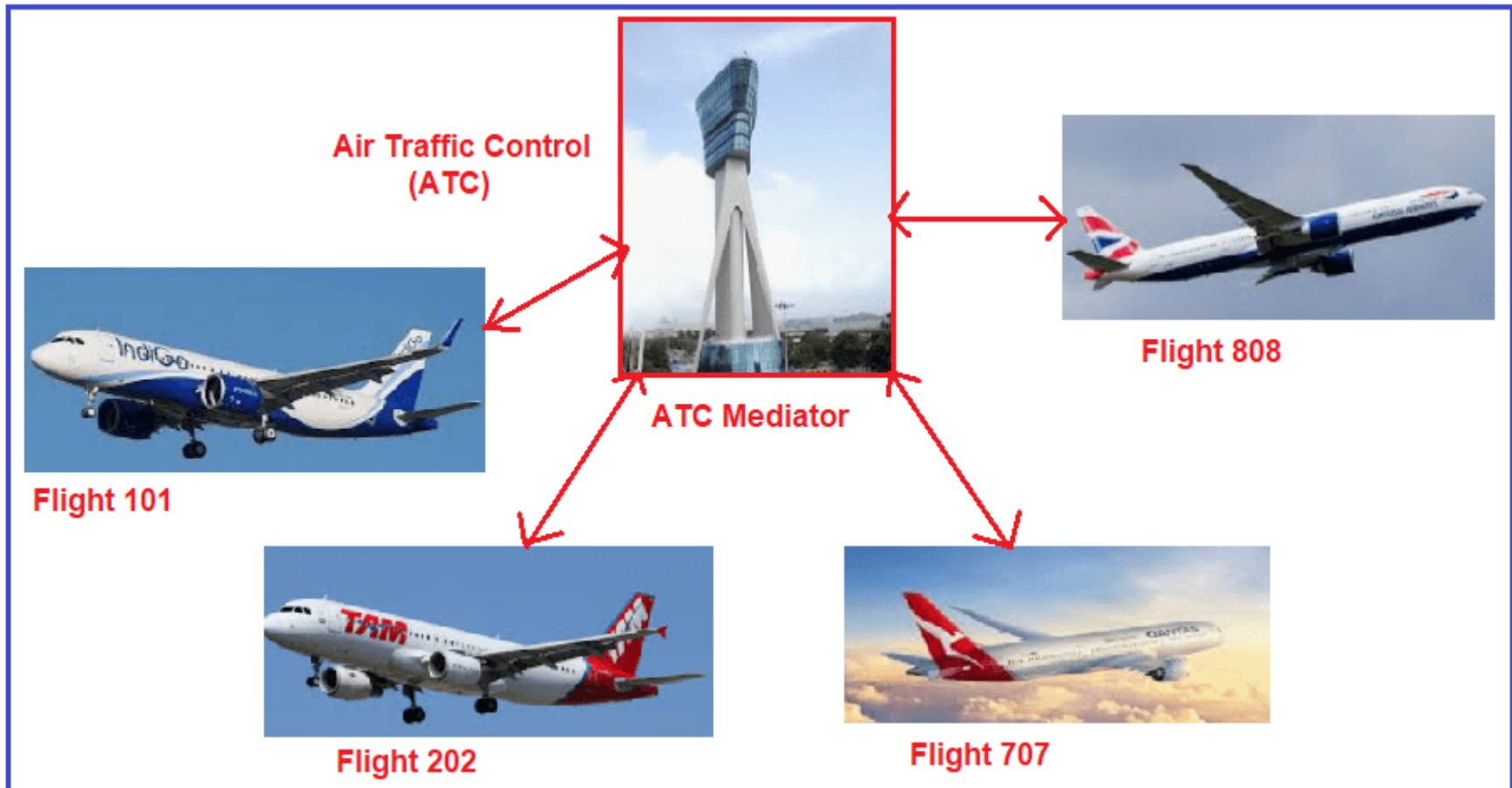


Mediator Pattern





Mediator Pattern





Memento Design Pattern

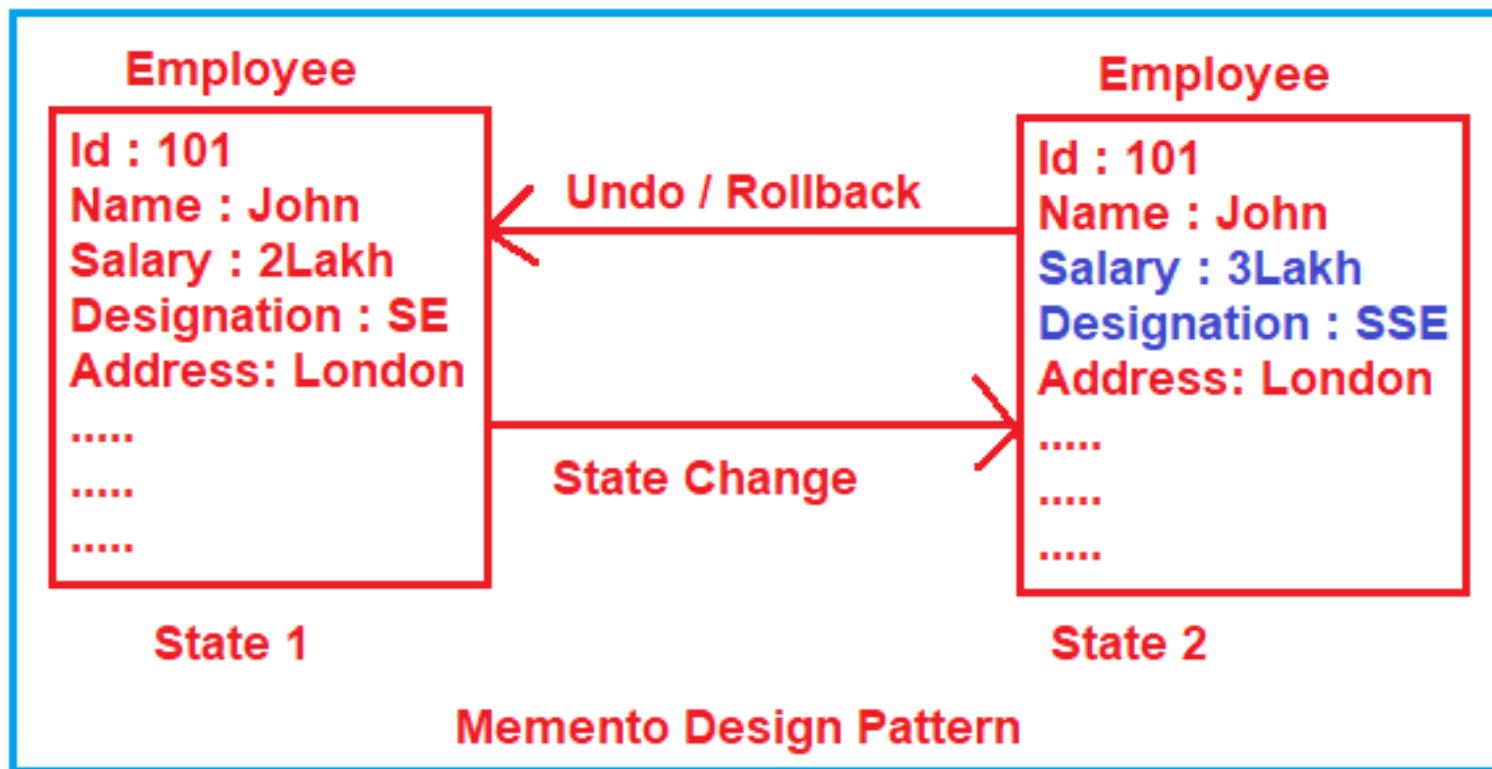
- The Memento Design Pattern is a Behavioral Design Pattern that can restore an object to its previous state.
- This pattern is useful for scenarios where you need to perform an undo or rollback operation in your application.
- The Memento pattern captures an object's internal state so that the object can be restored to this state later.
- It is especially useful when implementing undo functionality in an application.



Memento Design Pattern

- Let us understand the Memento Design Pattern in C# with an example.
- As per the image, on the left-hand side, we have an employee with Id =101, Name =John, Salary = 2Lakhs, Designation = Software Engineer, Address = London, and many more attributes.
- Later, we changed some of the properties, i.e., Salary to 3Lakhs and designation to Senior Software Engineer;
- we also changed some other employee attributes, shown on the right-hand side of the image below.
- That means we change the object state from State 1 to State 2.

Memento Design Pattern



Understanding the Class or UML Diagram of Memento Design Pattern:



UML Diagram of Memento Design Pattern

Understanding the Class or UML Diagram of Memento Design Pattern:



- Originator: The Originator is a class that creates a memento object containing a snapshot of the Originator's current state.
 - It also restores the Originator to one of its previous states.
 - The Originator class has two methods.
 - One is CreateMemento, and the other one is SetMemento.
 - The CreateMemento Method will Create a snapshot of the current state of the Originator and return that Memento.
 - We can store in the Caretaker for later use, i.e., for restoring purposes.
 - The SetMemento method accepts the memento object, and this method changes the Internal State of the Originator to one of its Previous States.

Understanding the Class or UML Diagram of Memento Design Pattern:



- Caretaker: The Caretaker class will hold the Memento objects for later use.
- This class acts as a store only. It never Checks or Modifies the contents of the Memento object.
- This class will have two methods, i.e., AddMemento and GetMemento.
- The AddMemento Method will add the memento, i.e., the internal state of the Originator, into the Caretaker.
- The GetMemento Method returns one of the Previous Originator Internal States, saved in the Caretaker.

Understanding the Class or UML Diagram of Memento Design Pattern:



- Memento: The Memento class holds information about the Originator's saved state.
- That means it sets the internal state and gets the internal state of the Originator object.
- This class has one method called GetState, which will return the Internal State of the Originator.
- This class also has one parameterized constructor, which you can set the internal state of the originator.



Use Cases of Memento Design Pattern:

- When you need to provide an undo mechanism in applications like text editors, graphic editors, or more complex transactional systems.
- In scenarios where you want to capture an object's state without exposing its implementation details.



When to Use Memento Design Pattern in C#?

- Undo Functionality: When you need to implement undo functionality in an application.
- The Memento pattern allows an object to save its state to be restored to this state later, effectively allowing operations to be reversed.
- Snapshot of an Object's State: In situations where you need to take a snapshot of an object's state at a particular point in time.
- This is useful in scenarios where you might need to roll back to a previous state if certain operations fail or if specific conditions are met.
- Preserving Encapsulation: If you need to preserve the encapsulation boundaries of an object while still capturing its internal state.
- The Memento pattern allows for capturing the internal state of an object without exposing its internal structure.

When to Use Memento Design Pattern in C#?



- State Save and Restore: In applications where it's necessary to save the state of an object so that it can be restored later, such as in save-and-restore features in games or application settings.
- Transactional Operations: When implementing transactional operations where changes to objects can be committed or rolled back based on the success or failure of the transaction.
- Point-in-time Recovery: For point-in-time recovery in applications, especially where the state of objects may change frequently, and there is a need to revert to a specific state under certain conditions.
- Journaling or Auditing: In cases where you need to maintain a history of object states for auditing or journaling purposes, allowing you to track changes over time.

Real-Time Examples of Memento Design Pattern in C#



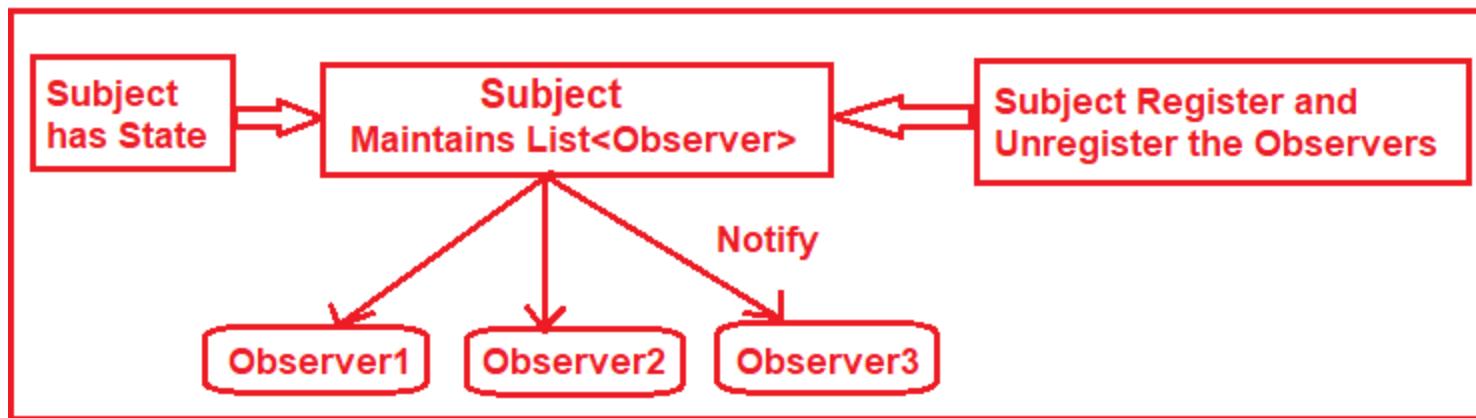
- Content Management System (CMS)
- Settings in a Software Application
- Banking System
- Game Character's State Management
- Drawing Board
- Text Editors
- Shopping Cart System
- Game Save Mechanism
- Graphic Design Tool
- Thermostat Temperature Control



Observer Pattern

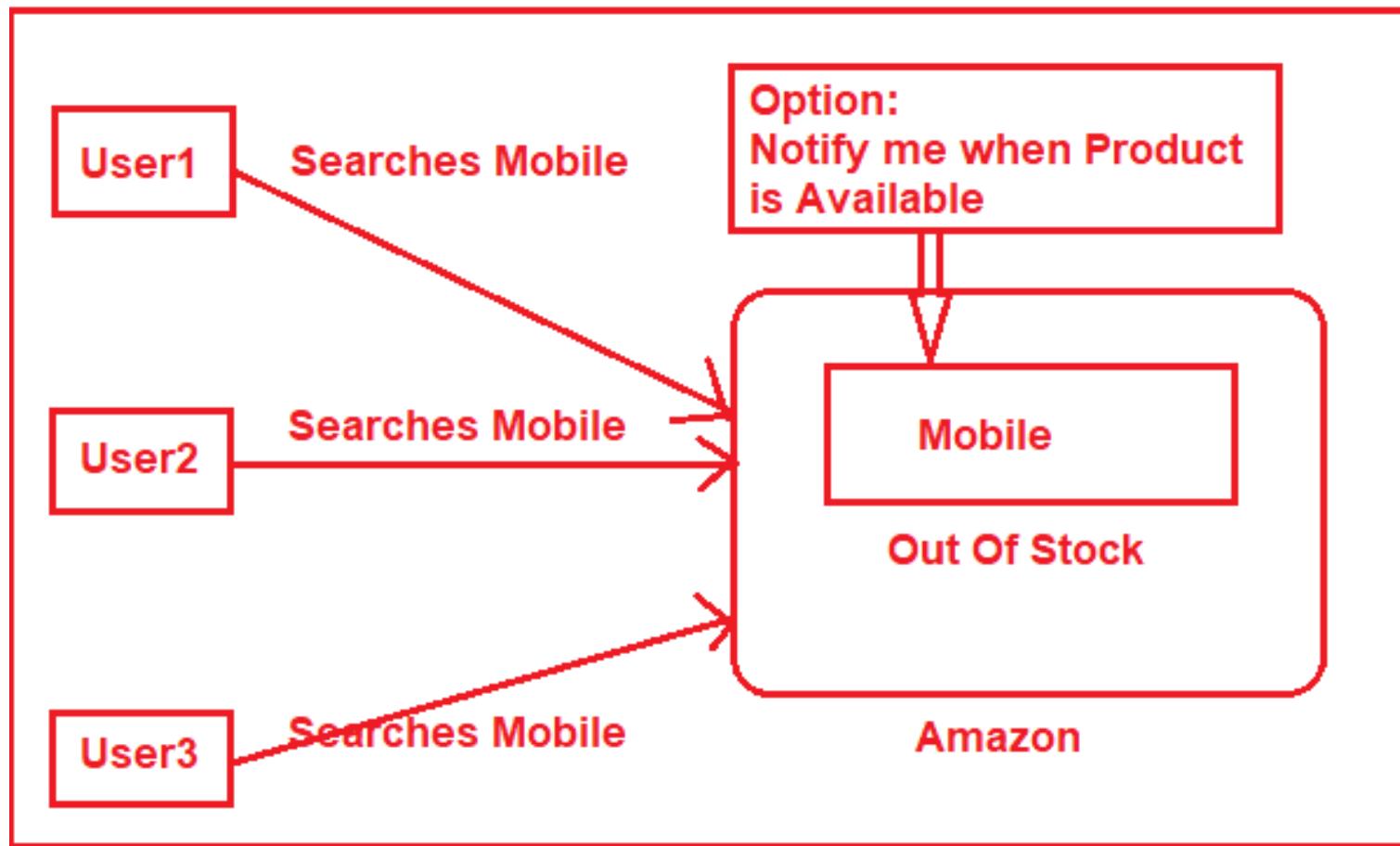
- Observer Design Pattern Defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
- This Design Pattern is widely used for implementing distributed event-handling systems where an object needs to notify other objects about its state changes without knowing who these objects are.
- In the Observer Design Pattern, an object (called a Subject) maintains a list of its dependents (called Observers).
- It notifies them automatically whenever any state changes by calling one of their methods.
- The Other names of this pattern are Producer/Consumer and Publish/Subscribe.

How Does the Observer Design Pattern Work?

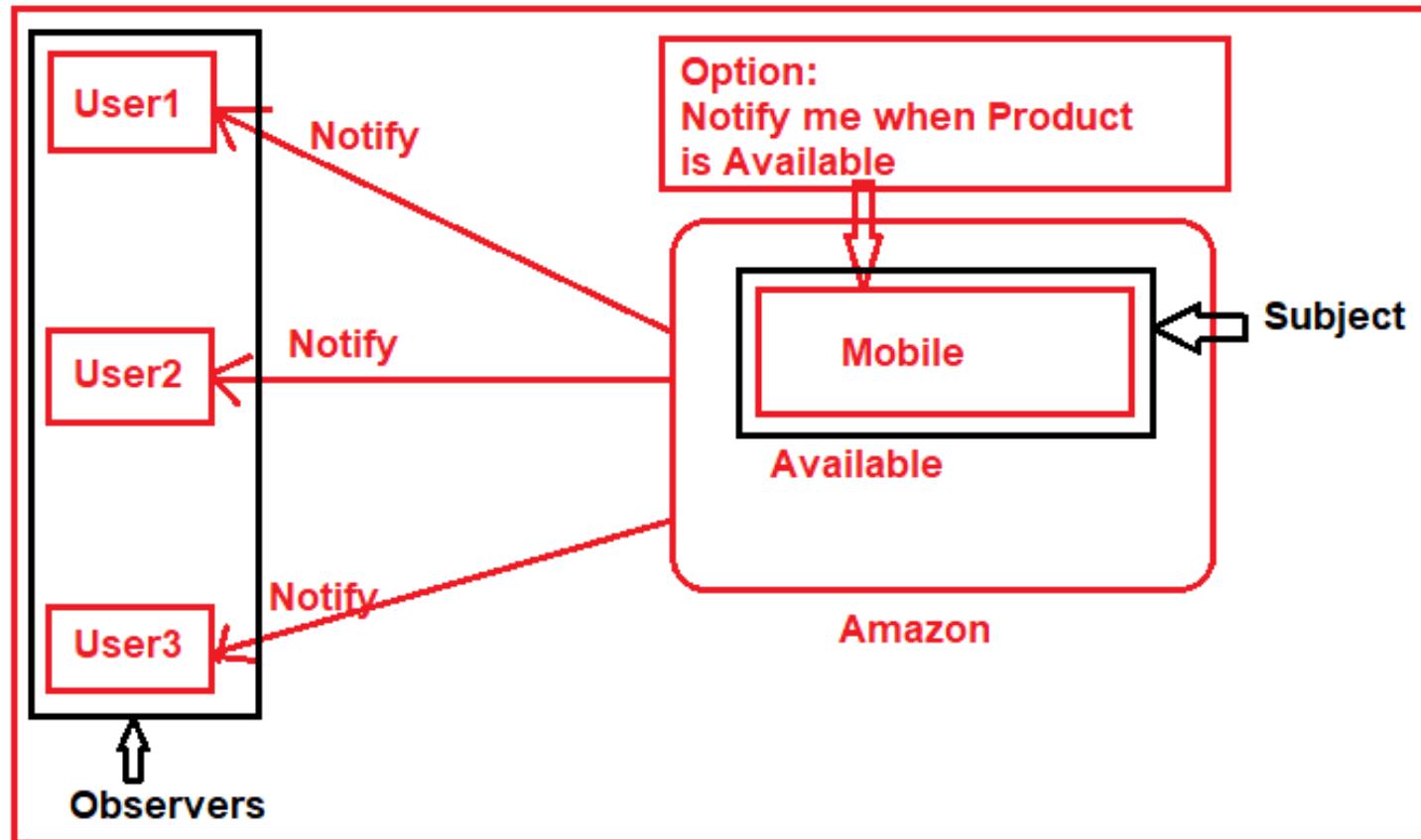




How Does the Observer Design Pattern Work?

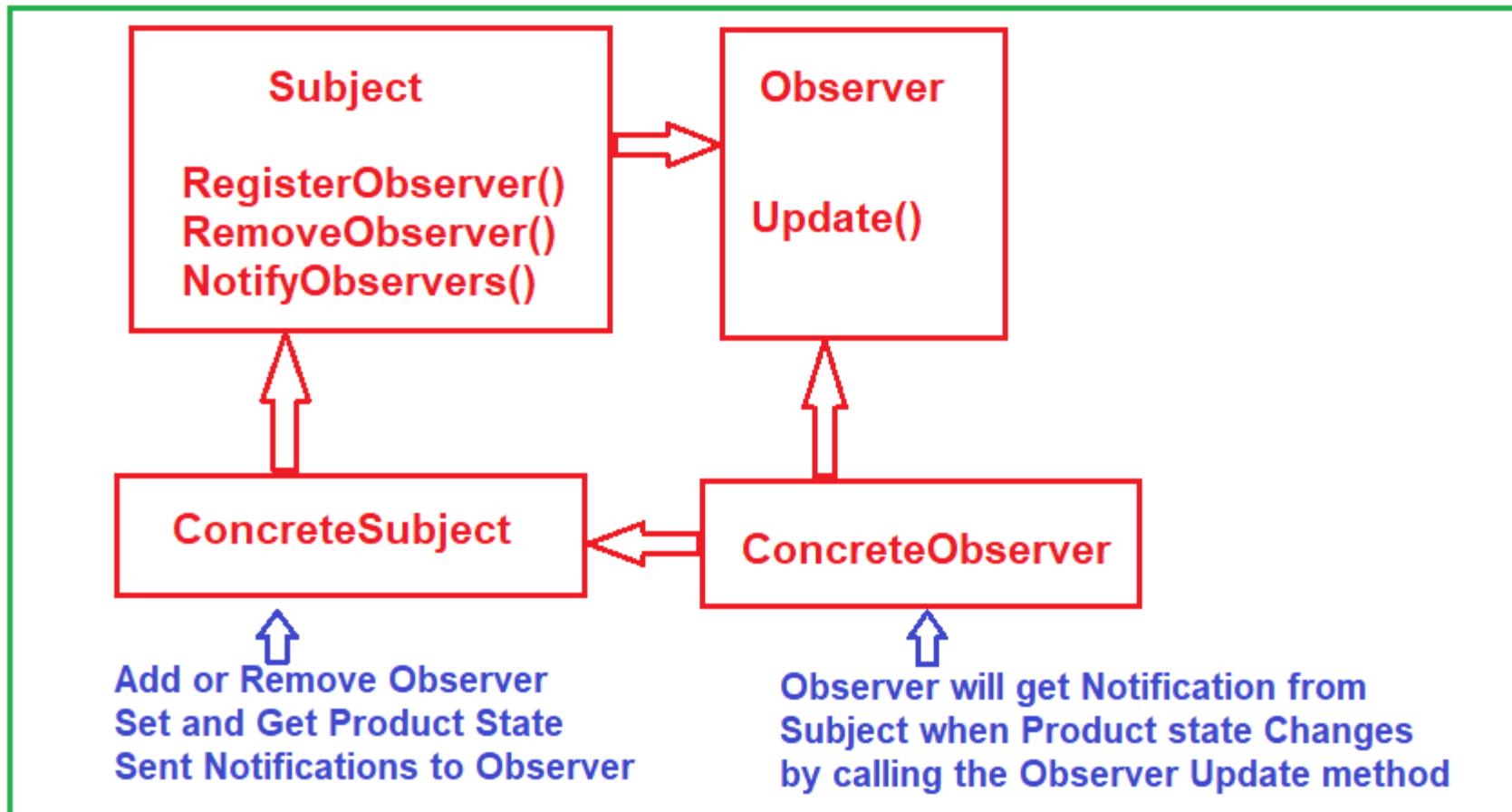


How Does the Observer Design Pattern Work?





Observer Pattern UML Diagram





Advantages of Observer Design Pattern

- Loose Coupling: The subject and observers are loosely coupled.
- The subject knows nothing about the observers except that they implement the observer interface.
- Dynamic Relationships: You can add and remove observers dynamically at runtime.
- Broadcast Communication: The subject broadcasts notifications to all interested observers without knowing their details.



Use Cases Observer Design Pattern:

- When a change to one object requires changing others, 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.
- For example, a mailing list where every time an event happens (i.e., a new product, a gathering, etc.), a message needs to be sent to the users who subscribed to the list.
- The Company needs to notify all its employees of any decision they make.
- Here, the Company is the Subject, and the Employees are the Observers.
- Any change in the company's policy and the company needs to notify all its employees or, you can say, Observers.

Real-Time Examples of Observer Design Pattern in C#



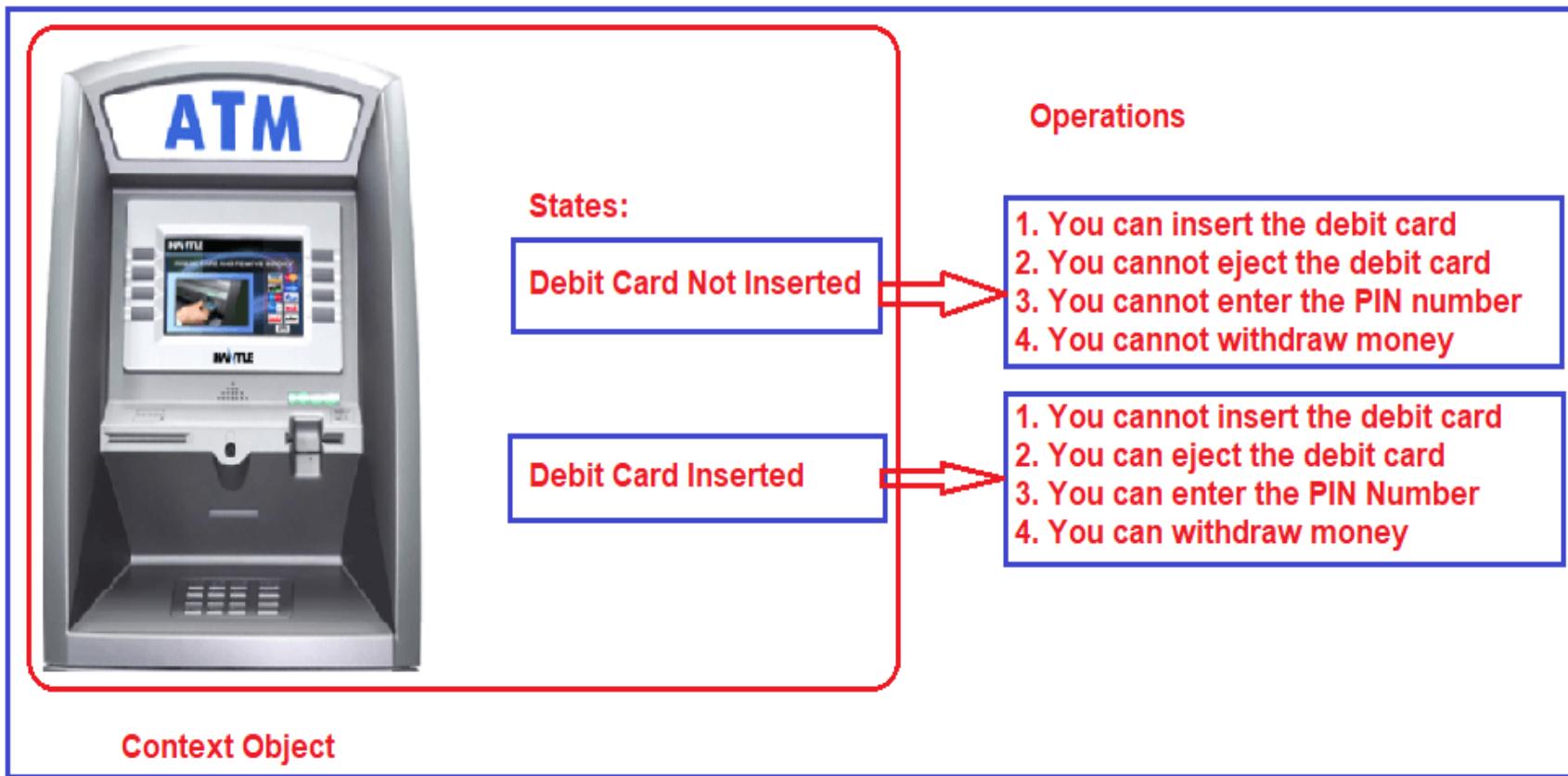
- JOB Portal System
- Server Monitoring System
- Weather Monitoring System
- Social Media Notification System
- Stock Market Tracking System
- E-Commerce System for Product Price Updates
- A News Publishing System



State Pattern

- State Design Pattern allows an object to alter its behavior when its internal state changes.
- State Design Pattern allows an object to change its behavior depending on its current internal state.
- The State Design Pattern encapsulates varying behavior for the same object based on its internal state.
- This pattern is useful when an object needs to go through several states, and its behavior differs for each state.
- Instead of having conditional statements throughout a class to handle state-specific behaviors, the State Design Pattern delegates this responsibility to individual state classes.

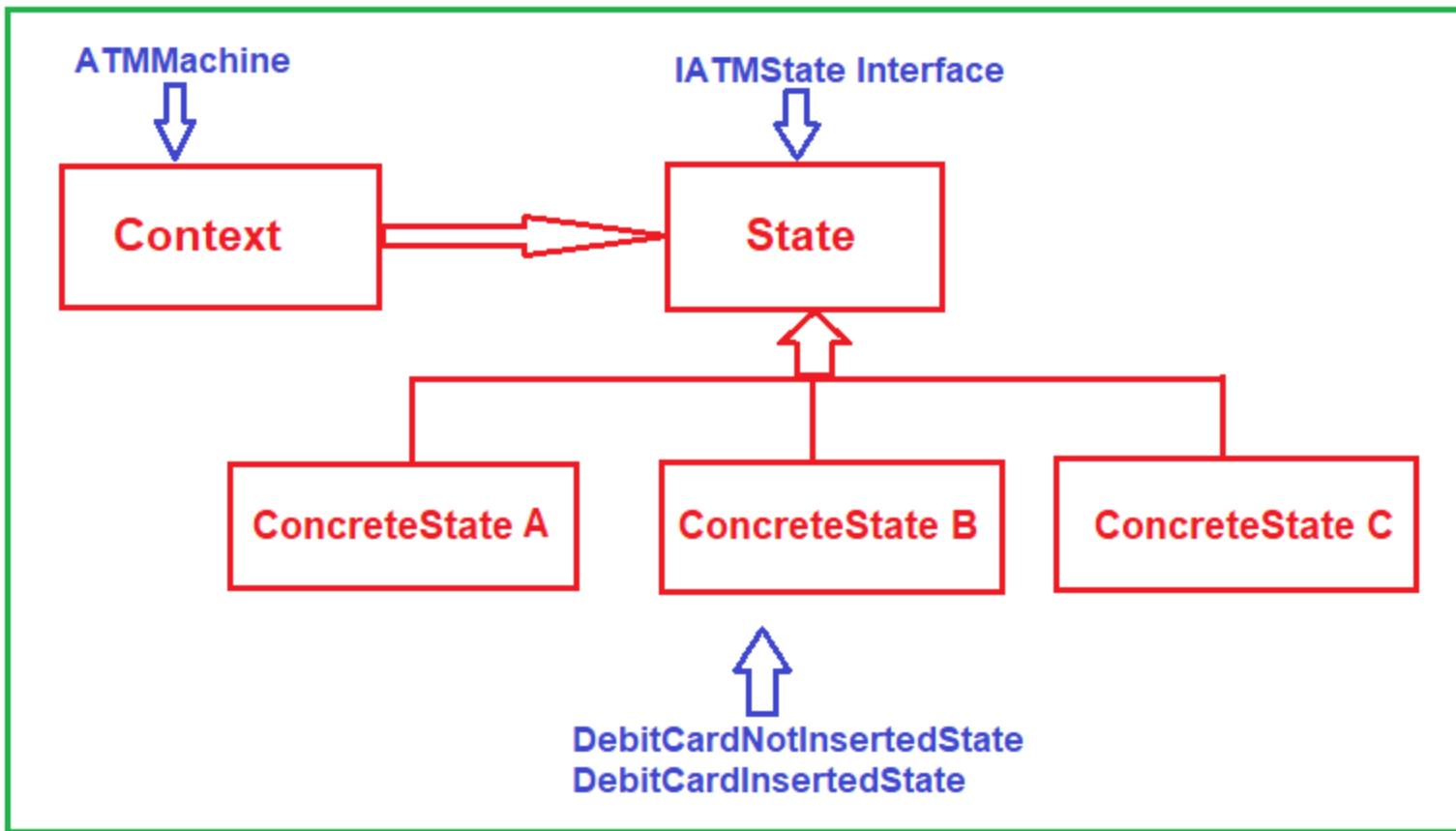
Understanding State Design Pattern With Real-Time Example:



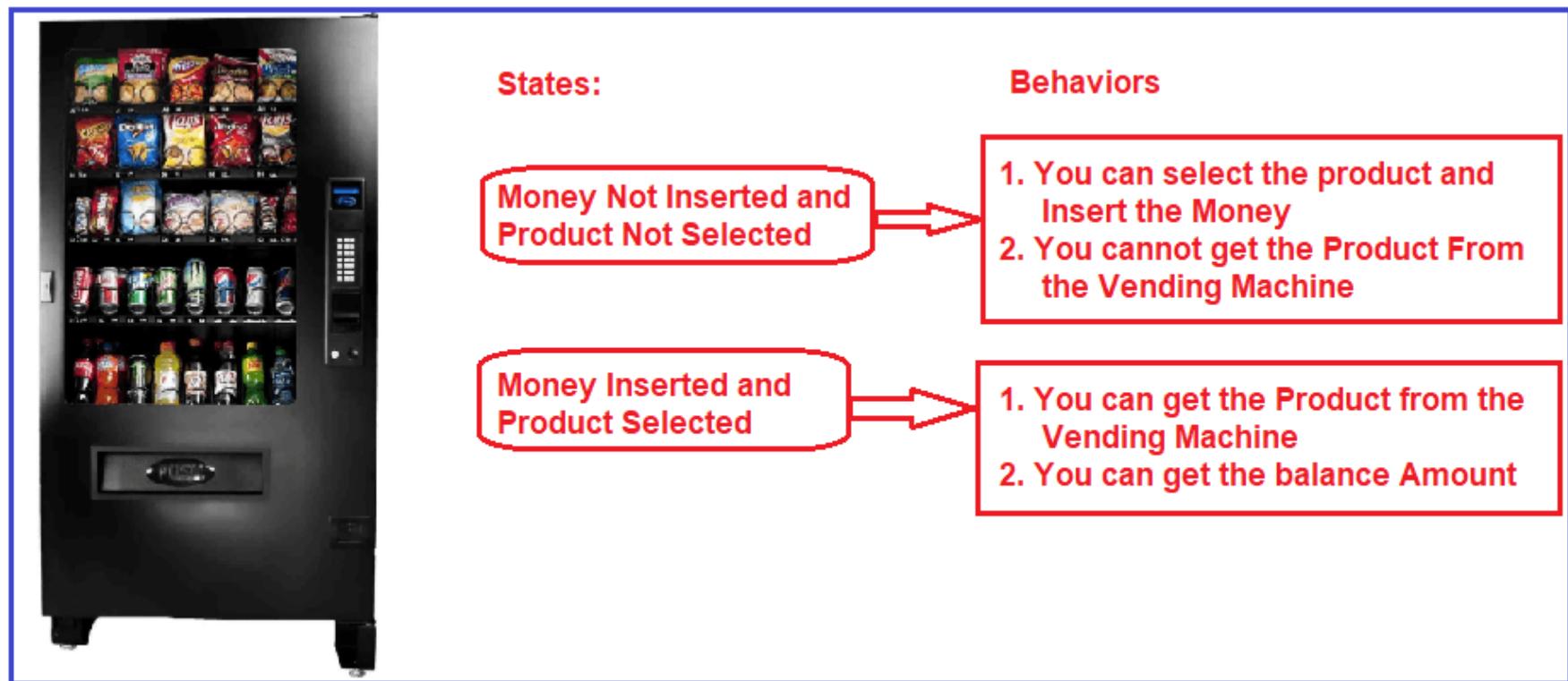
- State Interface: This defines an interface for encapsulating the behavior associated with a particular context state.
- Concrete State Classes: These classes implement the State interface and provide the actual behavior specific to a state.
- Context: This class maintains an instance of a Concrete State subclass that defines the current state. The context delegates state-specific behavior to the current State object.
- Client Code: Modifies the state of the context.



State Design Pattern UML Diagram



State Design Pattern UML Diagram



Real-Time Examples of State Design Patterns in C#

- Order Processing System
- Mobile Phone Ringer
- ATM Machine
- Traffic Light System
- Document Management
- Audio Player Context
- Product Lifecycle in an E-commerce Platform



Null Object Pattern

- The Null Object Design Pattern is a behavioral design pattern that is used to provide a consistent way of handling null or non-existing objects.
- It is particularly useful in situations where you want to avoid explicit null checks and provide a default behavior for objects that may not exist.



Null Object Pattern

- The Null object pattern is a design pattern that simplifies the use of dependencies that can be undefined.
- This is achieved by using instances of a concrete class that implements a known interface, instead of null references.
- We create an abstract class specifying various operations to be done, concrete classes extending this class.
- Null object class providing do-nothing implementation of this class which will be used seamlessly where we need to check the null value.

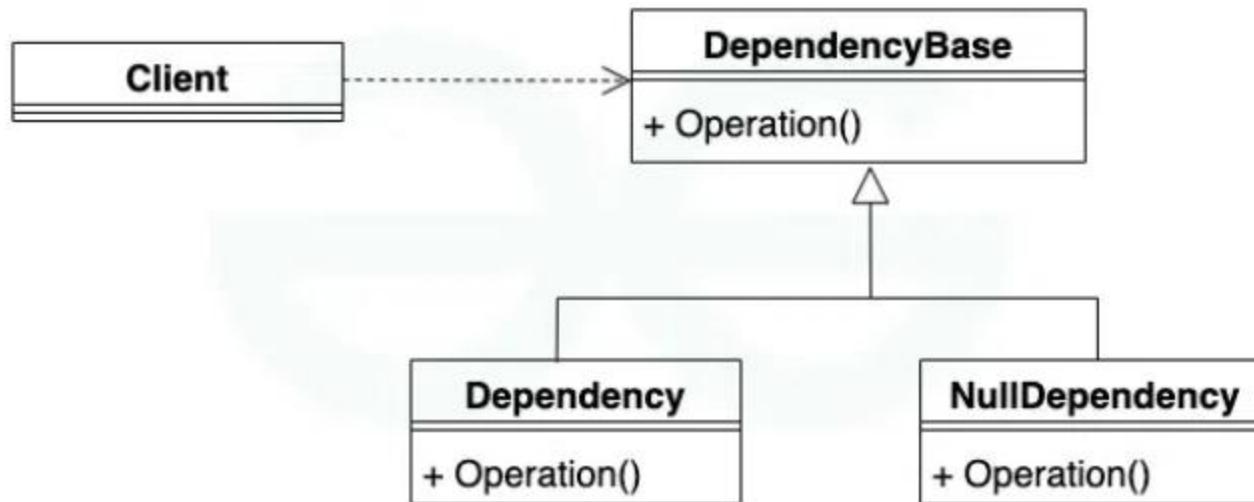


The Essence of the Null Object Pattern

- It eliminates the need for repetitive null checks throughout the codebase.
- It allows methods to safely call on objects without risking a NullReferenceException.
- It simplifies code by ensuring that an object always adheres to a known interface, even when it's effectively "empty."



Components of Null Object Class





Components of Null Object Class

- 1. Client
 - The client is the code that depends on an object that implements a Dependency interface or extends a Dependency Base abstract class.
 - The client uses this object to perform some operation.
 - The client should be able to treat both real and null objects uniformly, without needing to know which type of object it is dealing with.



Components of Null Object Class

- 2. DependencyBase (or Abstract Dependency)
 - DependencyBase is an abstract class or interface that declares the methods that all concrete dependencies, including the null object, must implement.
 - This class defines the contract that all dependencies must adhere to.



Components of Null Object Class

- 3. Dependency(or Real Dependency)
 - This class is a functional dependency that may be used by the Client.
 - The client interacts with Dependency objects without needing to know whether they are real or null objects.

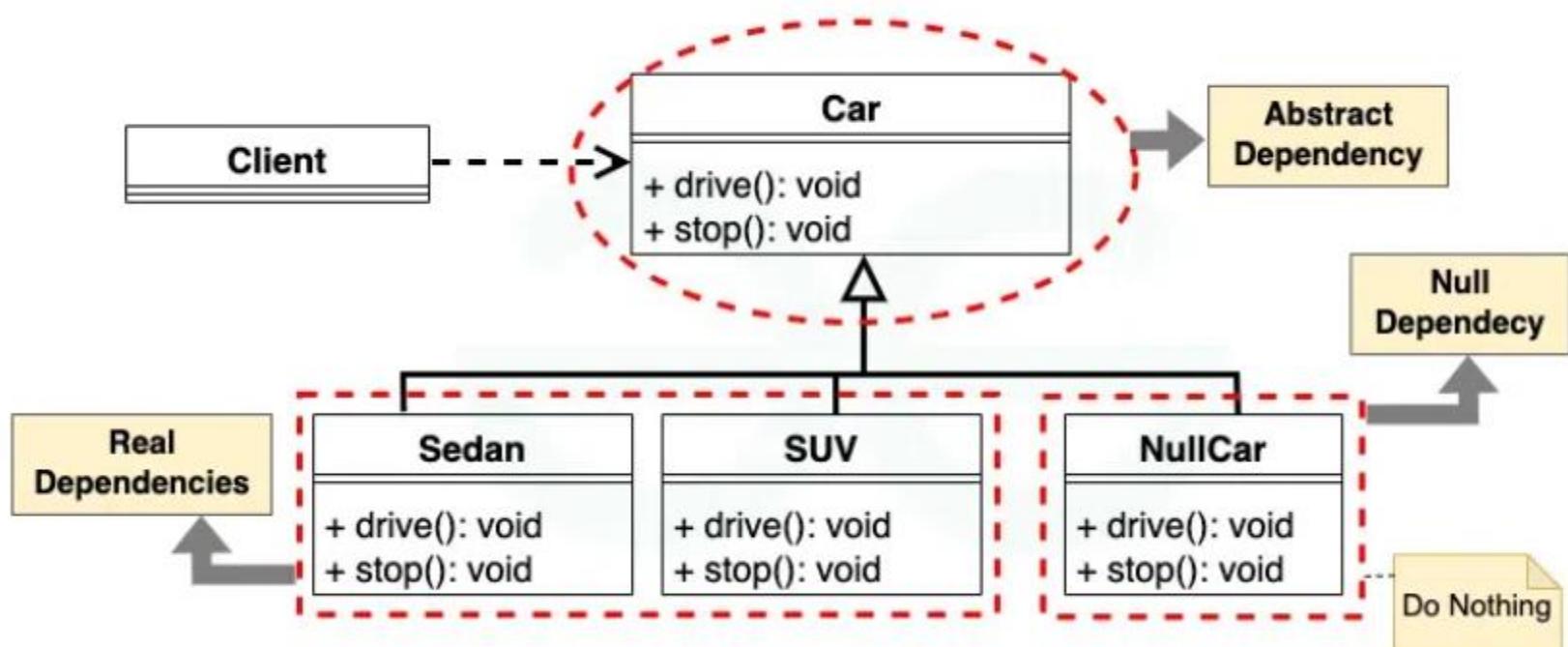


Components of Null Object Class

- 4. Null Object(or Null Dependency)
 - This is the null object class that can be used as a dependency by the Client.
 - It contains no functionality but implements all of the members defined by the Dependency Base abstract class.
 - Null Object represents a null or non-existing dependency in the system.
 - The client can safely call methods on a Null Object without causing errors or needing to perform null checks.



Components of Null Object Class

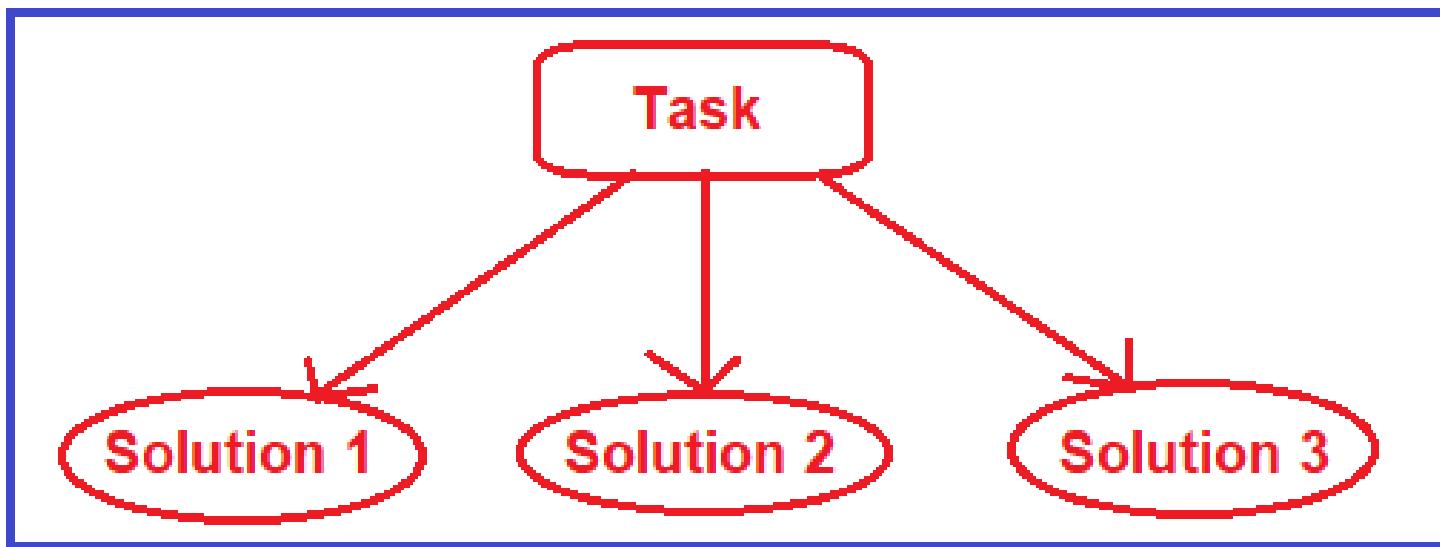




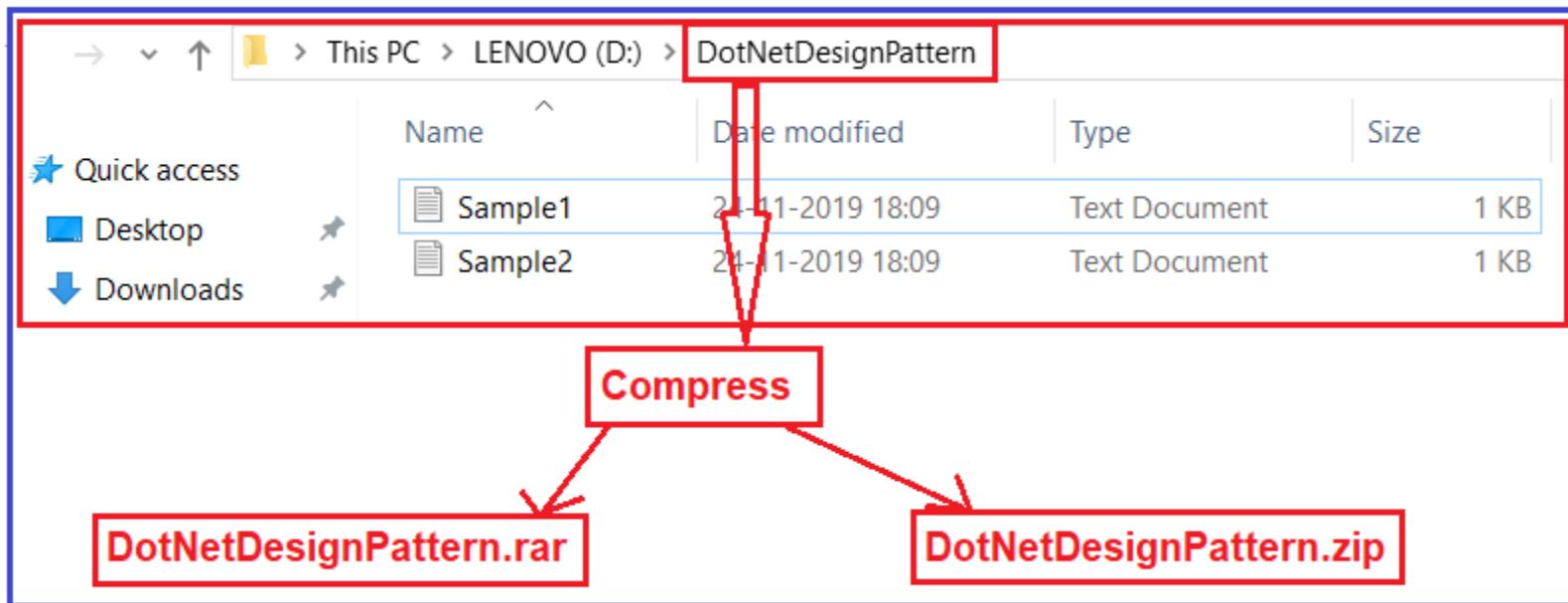
Strategy Pattern

- Define a family of algorithms, encapsulate each one, and make them interchangeable.
- Strategy lets the algorithm vary independently from clients that use it.
- The Strategy Design Pattern is a Behavioral Design Pattern that enables selecting an algorithm's behavior at runtime.
- Instead of implementing a single algorithm directly, runtime instructions specify which of a family of algorithms to use.
- This pattern is ideal when you need to switch between different algorithms or actions in an object dynamically.
- That means the Strategy Design Pattern is used when we have multiple algorithms (solutions) for a specific task, and the client decides which algorithm to use at runtime.

Example to Understand Strategy Design Pattern



Understanding the Strategy Design Pattern with Real-time Example



Solution 1

Solution 2

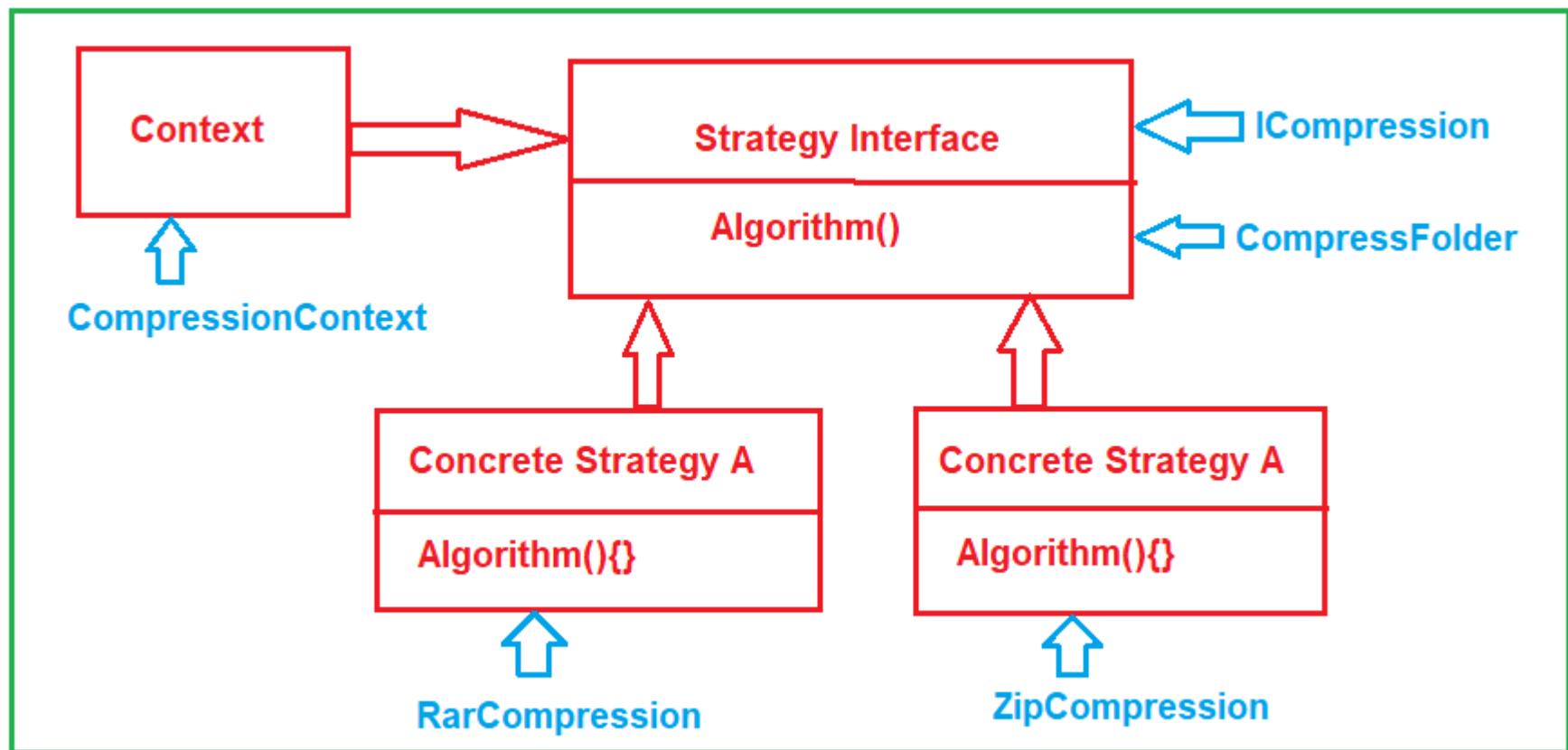
How to Implement the Strategy Design Pattern in C#?



- **Strategy (Interface or Abstract Class):** This defines an interface common to all supported algorithms.
- **Context** uses this interface to call the algorithm defined by a **Concrete Strategy**.
- **Concrete Strategy:** This implements the algorithm using the **Strategy** interface.
- **Context:** This maintains a reference to a **Strategy** object and may define an interface that lets **Strategy** access its data.



Strategy Design Pattern UML Diagram





When to Use Strategy Design Pattern in C#?

- Dynamic Behavior Change: When you need to dynamically change the behavior of an object based on some context or state.
- This allows the algorithm to be selected at runtime rather than being statically defined.
- Encapsulating Algorithms: If you have several different algorithms or behaviors for a specific task, and you want to encapsulate each one in its own class.
- The Strategy pattern allows you to switch between these algorithms without changing the clients that use them.
- Avoiding Conditional Statements: In situations where you might otherwise use multiple conditional statements (like if/else or switch/case) to select desired behaviors.
- The Strategy pattern helps to avoid this conditional complexity.

When to Use Strategy Design Pattern in C#?



- Support for Open/Closed Principle: If you anticipate that new algorithms or behaviors will be added in the future, the Strategy pattern allows for extending the capabilities without modifying the existing code, thus supporting the Open/Closed Principle.
- Isolating the Implementation Details: When you want to isolate the implementation details of an algorithm from the code that uses it.
- This can help in making the code easier to maintain and understand.
- Unit Testing and Mocking: The pattern makes it easier to unit test the different behaviors of a class independently, as well as to mock out these behaviors if needed.
- Implementing Different Business Rules: If different clients or systems require different business rules or algorithms, the Strategy pattern allows for configuring a class with one of many behaviors based on the client's needs.

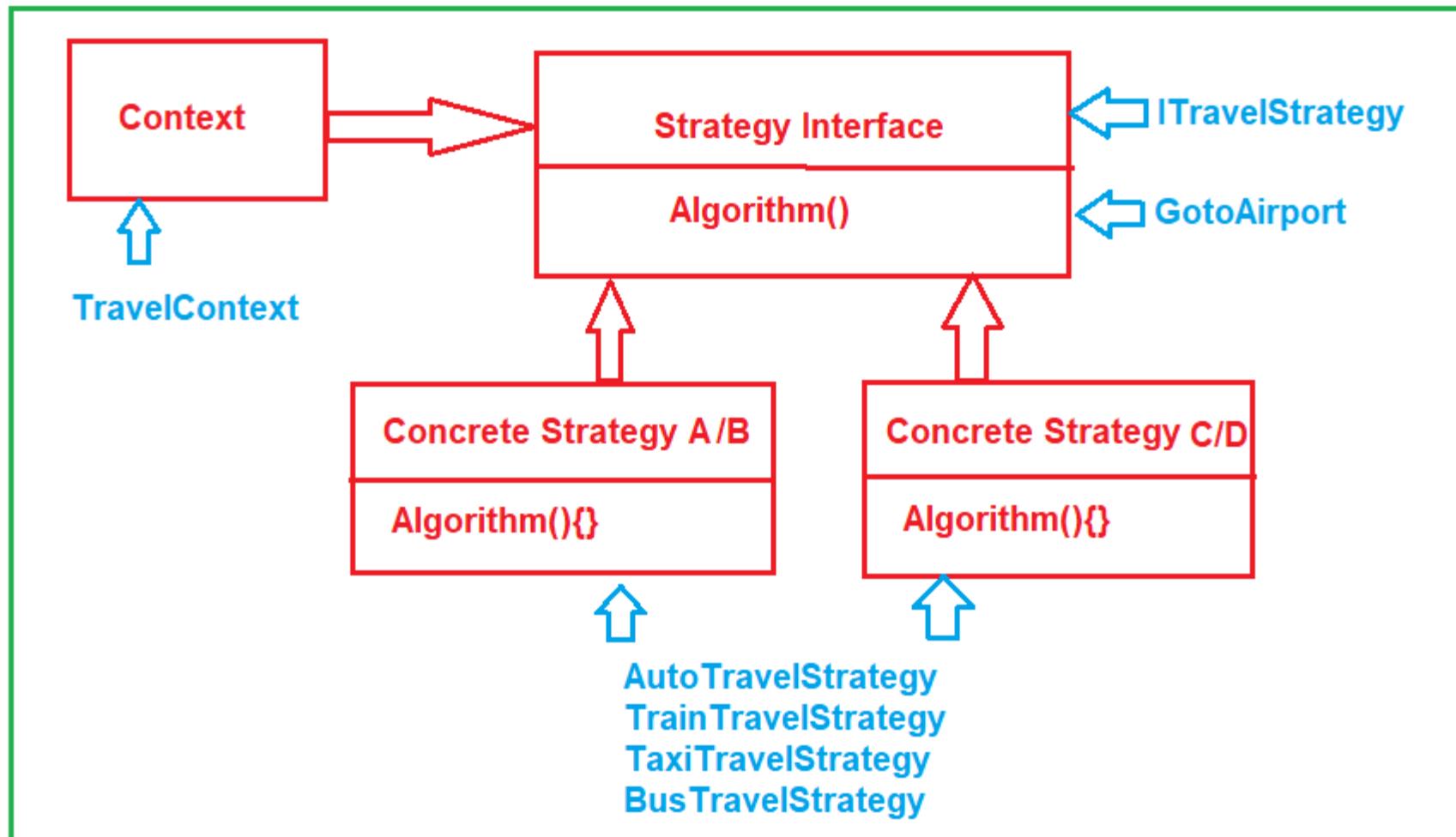


Strategy Pattern Real Time



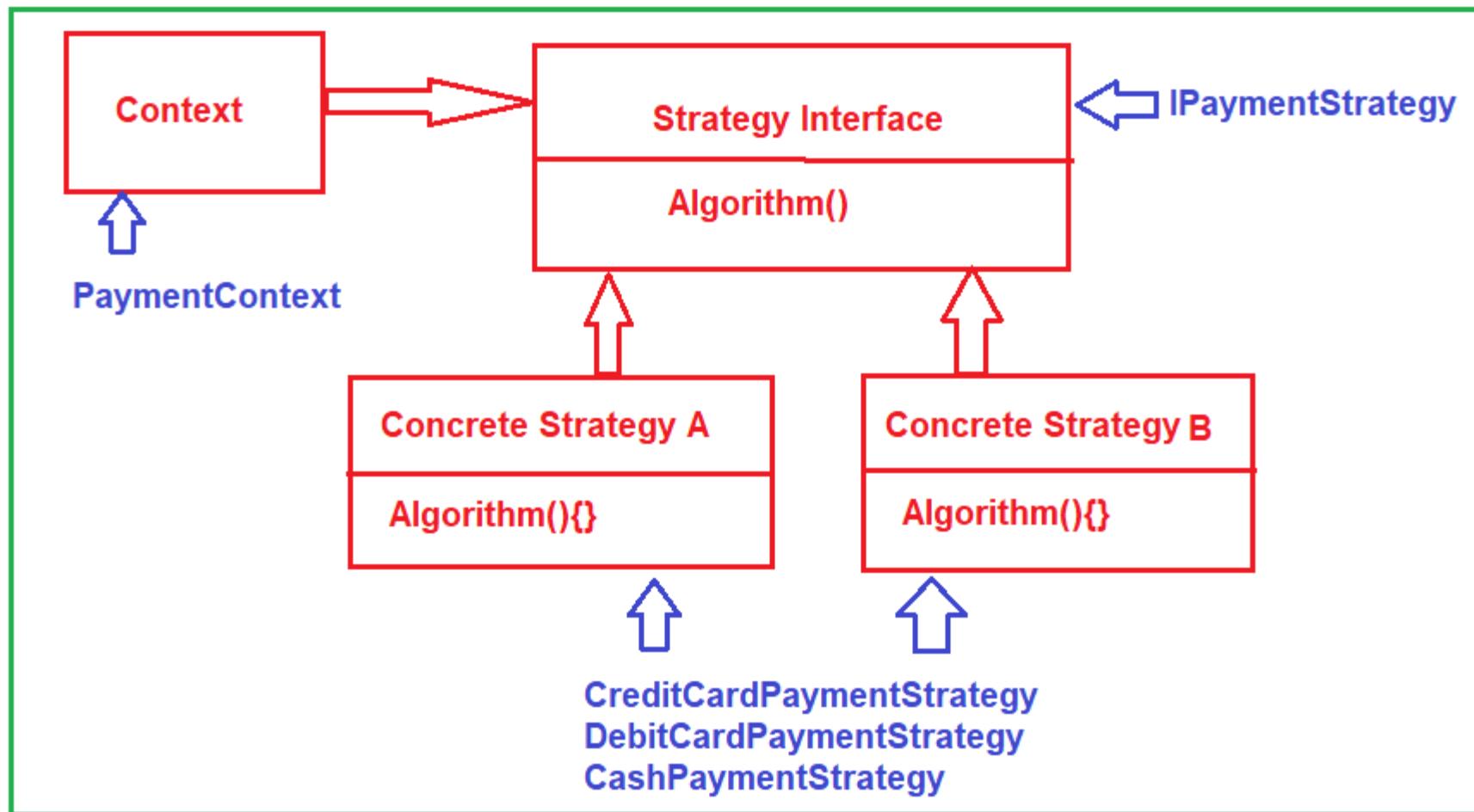


Strategy Pattern Real Time





Strategy Pattern Real Time





Template Pattern

- Template Method Design Pattern defines a sequence of steps of an algorithm and allows the subclasses to override the steps but is not allowed to change the sequence.

Template Pattern



Foundation



Pillars



Walls



Windows



Concrete House

Template Pattern



Foundation



Pillars



Walls

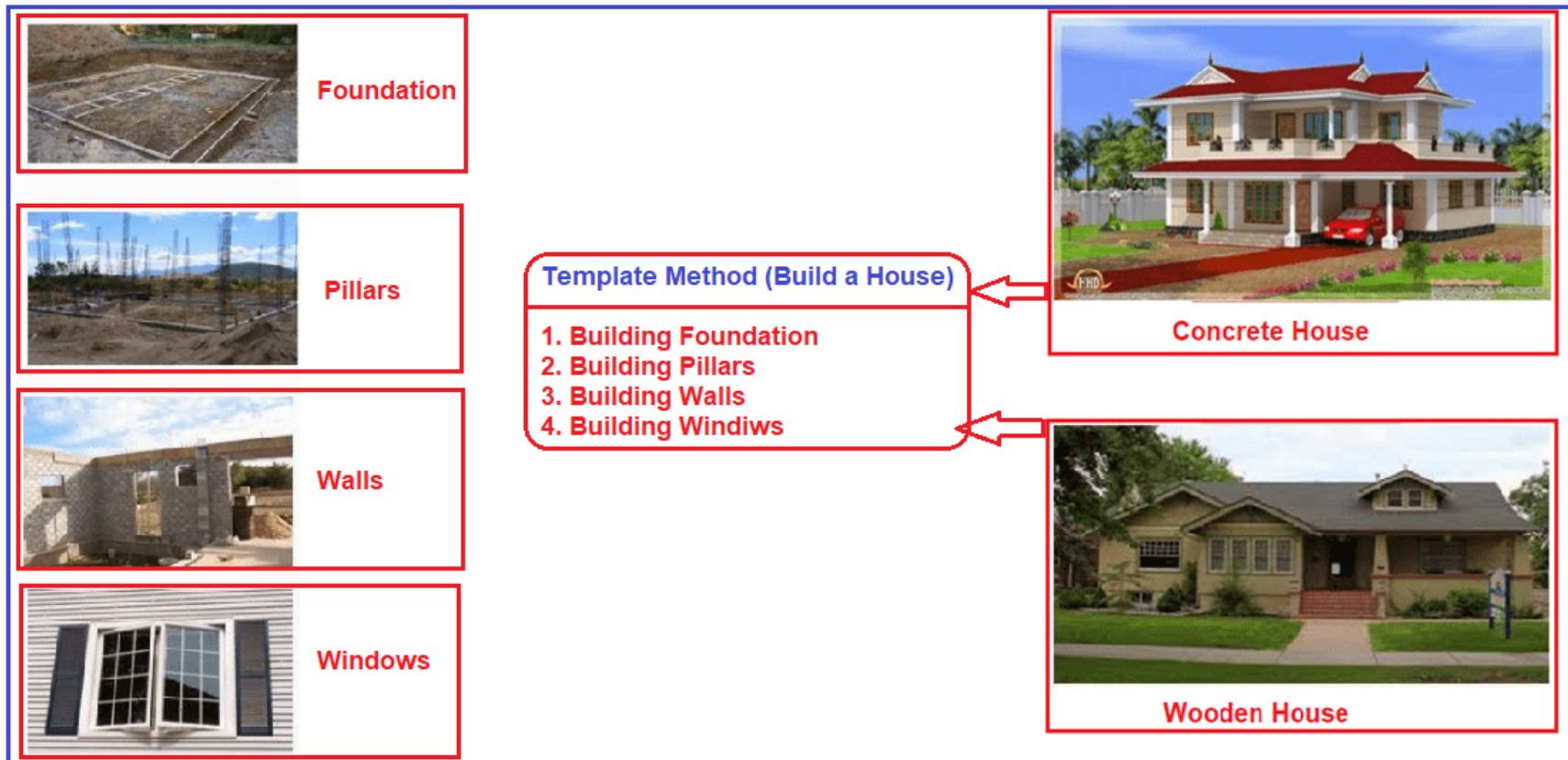


Windows



Wooden House

Template Pattern



Real-Time Examples of Template Method Design Pattern in C#



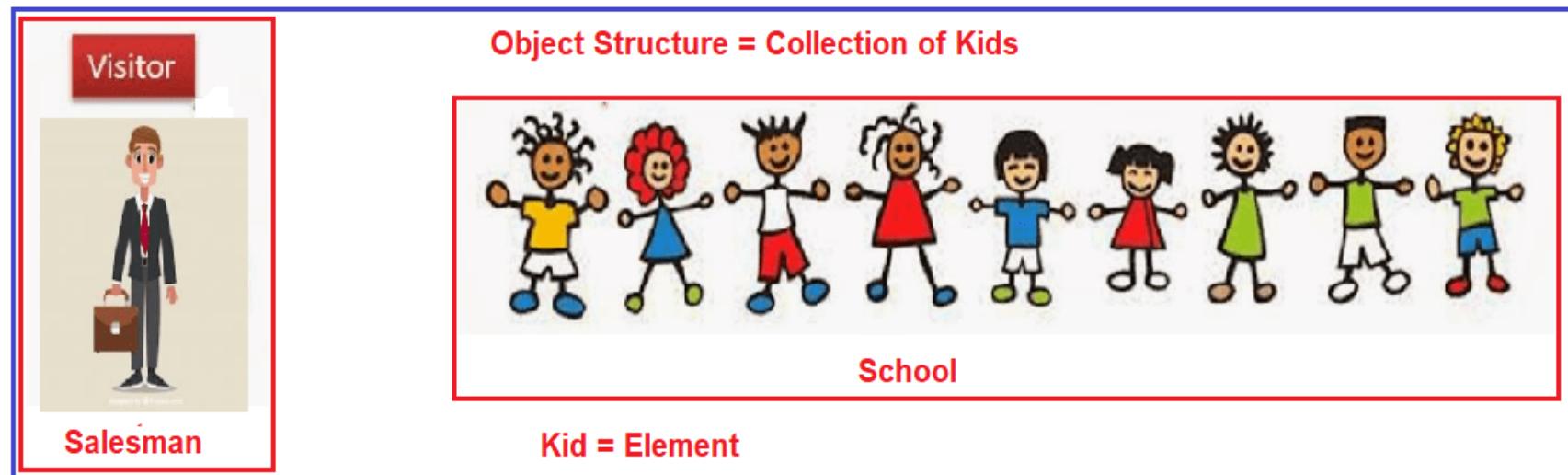
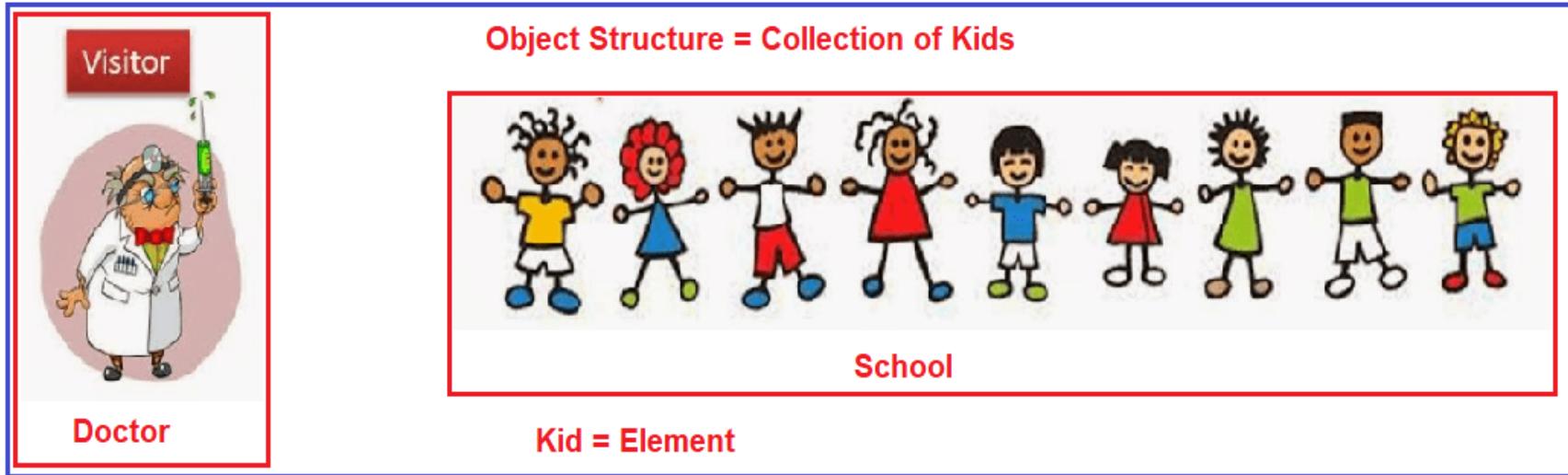
- Data Synchronization Between Source and Destination
- Online Shopping System
- Document Converter Tool
- Test-Taking Platforms (Online Exam)
- Online Food Recipes
- Fitness Workouts
- CI/CD Pipeline Execution
- Bank Account Operations
- Planning a Trip



Visitor Pattern

- In the Visitor Design Pattern, we use a Visitor object that changes an element object's executing algorithm.
- In this way, when the visitor varies, the execution algorithm of the element object can also vary.
- As per the Visitor Design Pattern, the element object must accept the visitor object so that the visitor object handles the operation on the element object.
- The Visitor Design Pattern should be used when you have distinct and unrelated operations to perform across a structure of objects (element objects).
- That means the Visitor Design is used to create and perform new operations on a set of objects without changing the object structure or classes.

Visitor Pattern



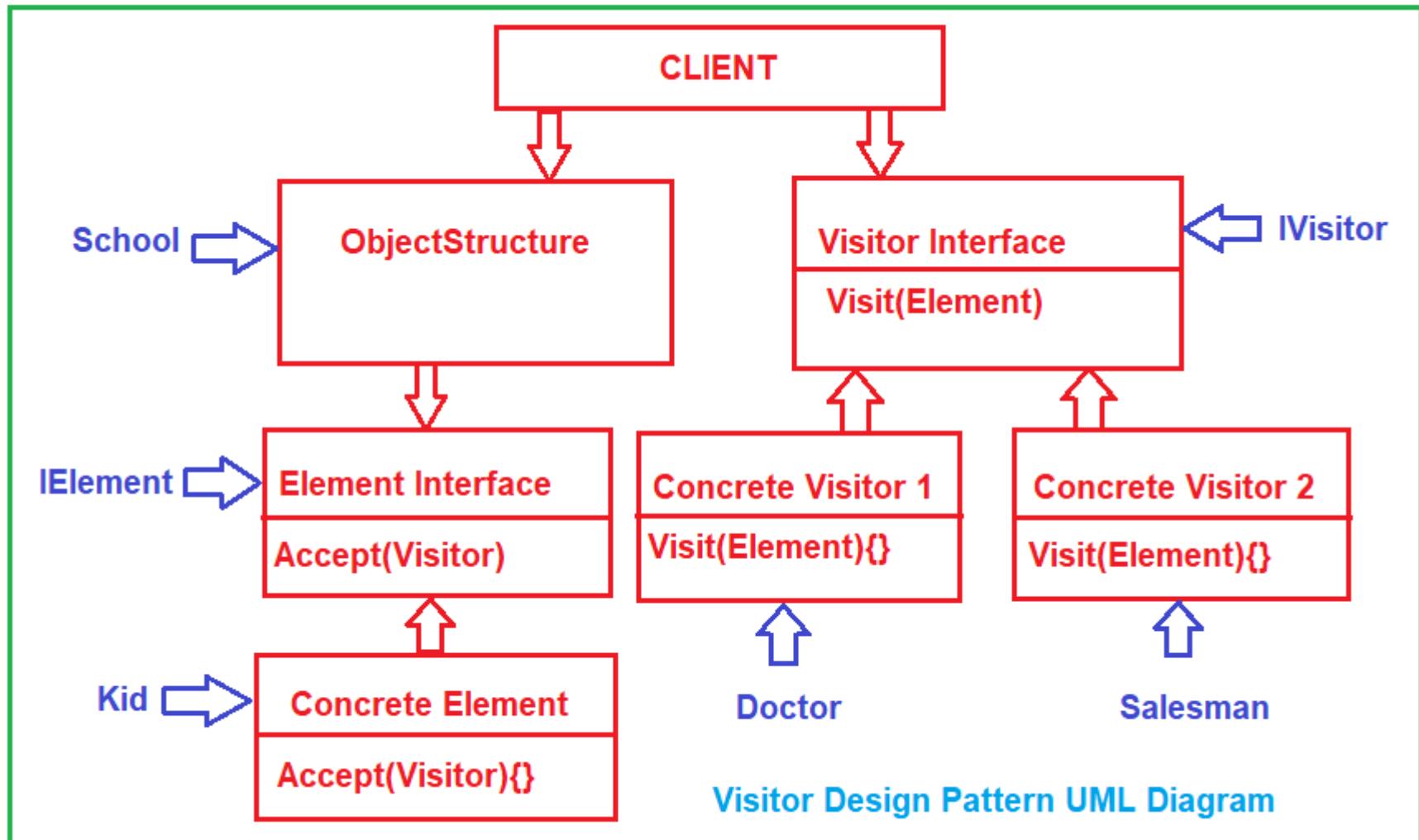
How to Implement Visitor Design Pattern in C#?



- Element: This represents an element of an object structure. It provides an Accept method that takes a visitor as an argument.
- ConcreteElement: This is a concrete class that implements the Element interface.
- Visitor: This interface declares a visit operation for each class of ConcreteElement.
- ConcreteVisitor: This is a concrete class that implements the Visitor interface.



Visitor Design Pattern UML Diagram



Real-Time Examples of Visitor Design Patterns in C#



- Computer Parts Store
- Postal Service System
- Online Shopping System
- Zoo Simulation
- Computer System Component Hierarchy
- Tax Calculation System
- Ticket Pricing System
- Post and Package Delivery System



When to use Visitor Design Pattern in C#?

- Operations across Diverse Classes: If you have several different classes and need to perform operations that are specific to each of those classes, but you don't want to add these operations directly to the classes.
- This is particularly useful when those operations are not tightly related to the primary responsibility of those classes.
- Separation of Concerns: If you want to separate an algorithm or operation from the classes on which it operates, the Visitor pattern can be useful.
- This can lead to a cleaner code by segregating responsibilities.
- Adding Operations without Modifying Classes: When you need to add new operations without modifying the classes, especially if they are part of a closed library for modification.
- This way, the Visitor pattern allows you to maintain the open/closed principle.



When to use Visitor Design Pattern in C#?

- Accumulating State: If you need to accumulate state as you visit each element in a structure.
- Visitors can maintain their state during the traversal process, allowing them to gather and accumulate data as they visit each element.
- Unified Access across Composite Structures: In conjunction with the Composite pattern, the Visitor pattern can provide a way to perform operations over structures composed of disparate objects, allowing unified access to the composite elements.
- When you need to break dependency cycles: Visitors can be used to break dependency cycles by decoupling operations from the data structures they operate upon.



Data Access Object or Repository Pattern

- A repository is a class defined for an entity with all the possible database operations.
- For example, a repository for an Employee entity will have the basic CRUD operations and any other possible operations related to the Employee entity.



Data Access Object or Repository Pattern

- One of the most important aspects of this strategy is the separation between the actual database, queries, and other data access logic from the rest of the application.
- In our example, we must separate the data access logic from the Employee Controller.
- The Repository Design Pattern is one of the most popular design patterns to achieve such separation between the actual database, queries, and other data access logic from the rest of the application.



Data Access Object or Repository Pattern

```
public ActionResult Index()  
{  
  
public ActionResult AddEmployee()  
{  
  
public ActionResult UpdateEmployee()  
{  
  
public ActionResult DeleteEmployee()  
{
```

EmployeeController

Entity Framework
Data Context
And Entities

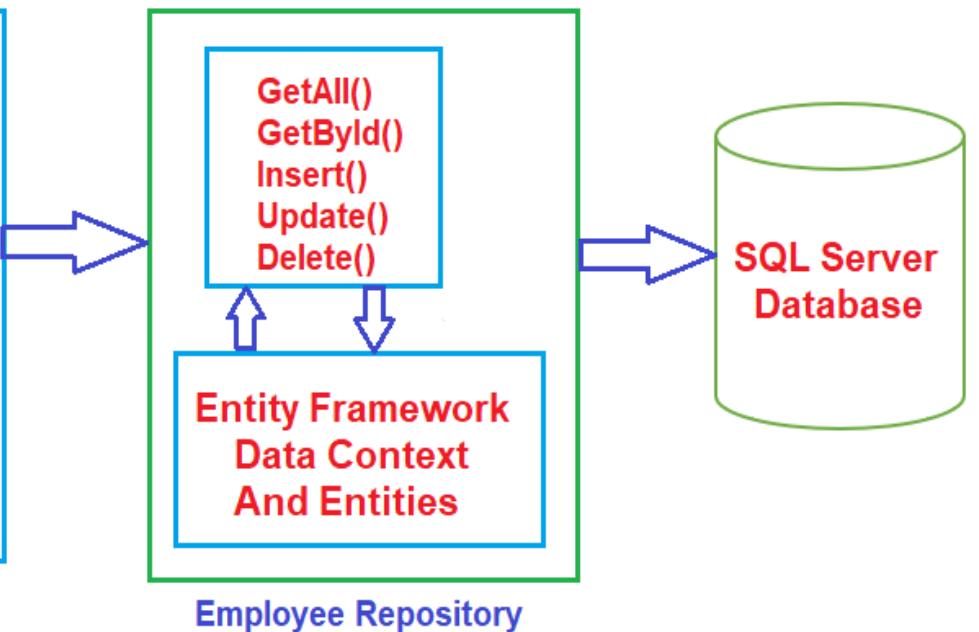
SQL Server
Database





Data Access Object or Repository Pattern

```
public ActionResult Index()  
{  
  
public ActionResult AddEmployee()  
{  
  
public ActionResult UpdateEmployee()  
{  
  
public ActionResult DeleteEmployee()  
{
```





Intercepting Filter Pattern

- In software development, cross-cutting concerns, such as logging, authentication, and data validation, often permeate throughout an application.
- Managing these concerns can become complex and lead to code duplication.
- The Intercepting Filter design pattern provides a systematic approach to address cross-cutting concerns by introducing filters that intercept and process requests and responses.



Intercepting Filter Pattern

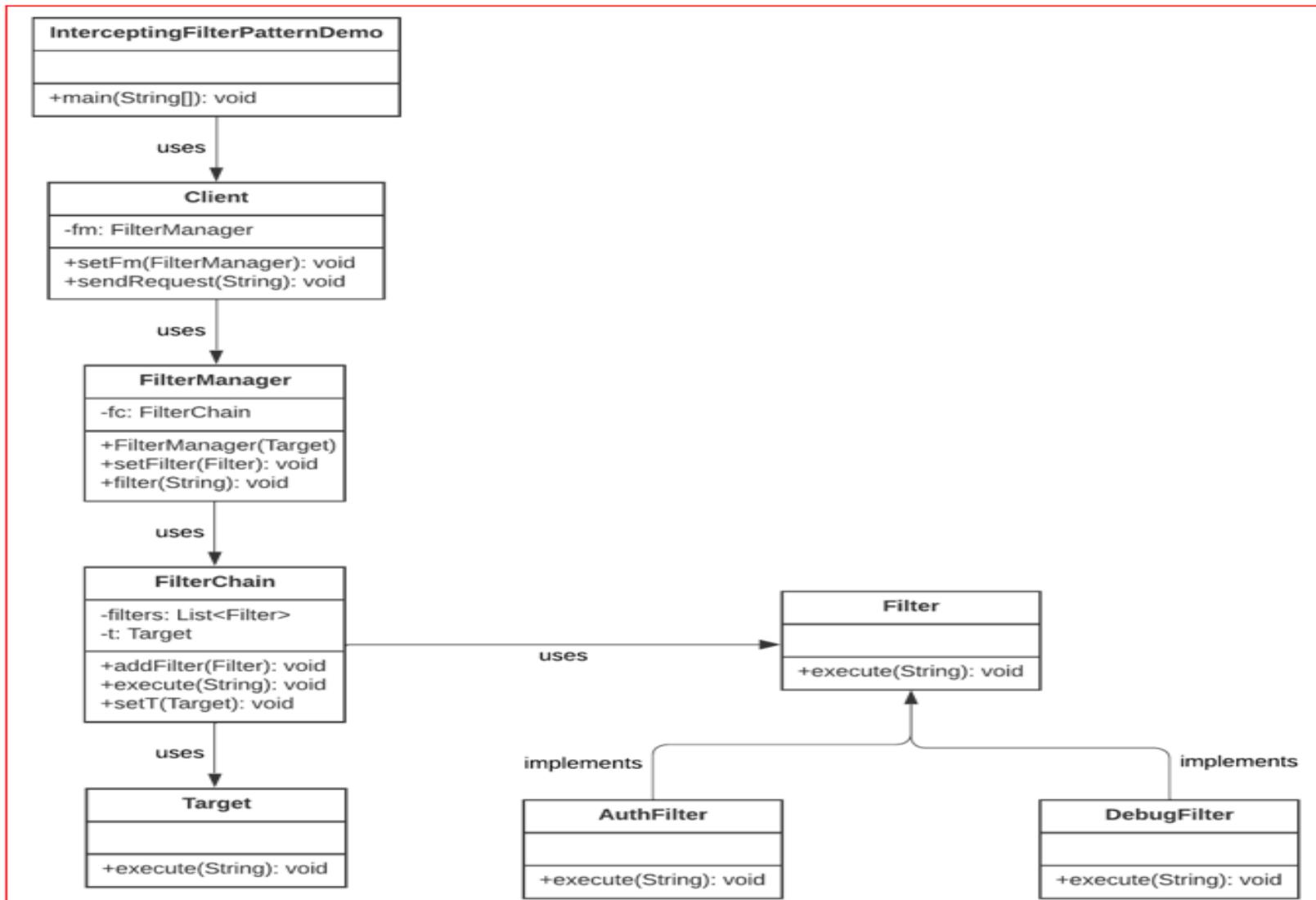
- The Intercepting Filter design pattern focuses on intercepting requests and responses in a system to apply common processing or manipulations.
- It introduces filters that intercept incoming requests before they reach the core processing logic.
- These filters can modify or enhance the requests and responses, adding functionality such as authentication, logging, or input validation.

Components of Intercepting Filter Design Pattern

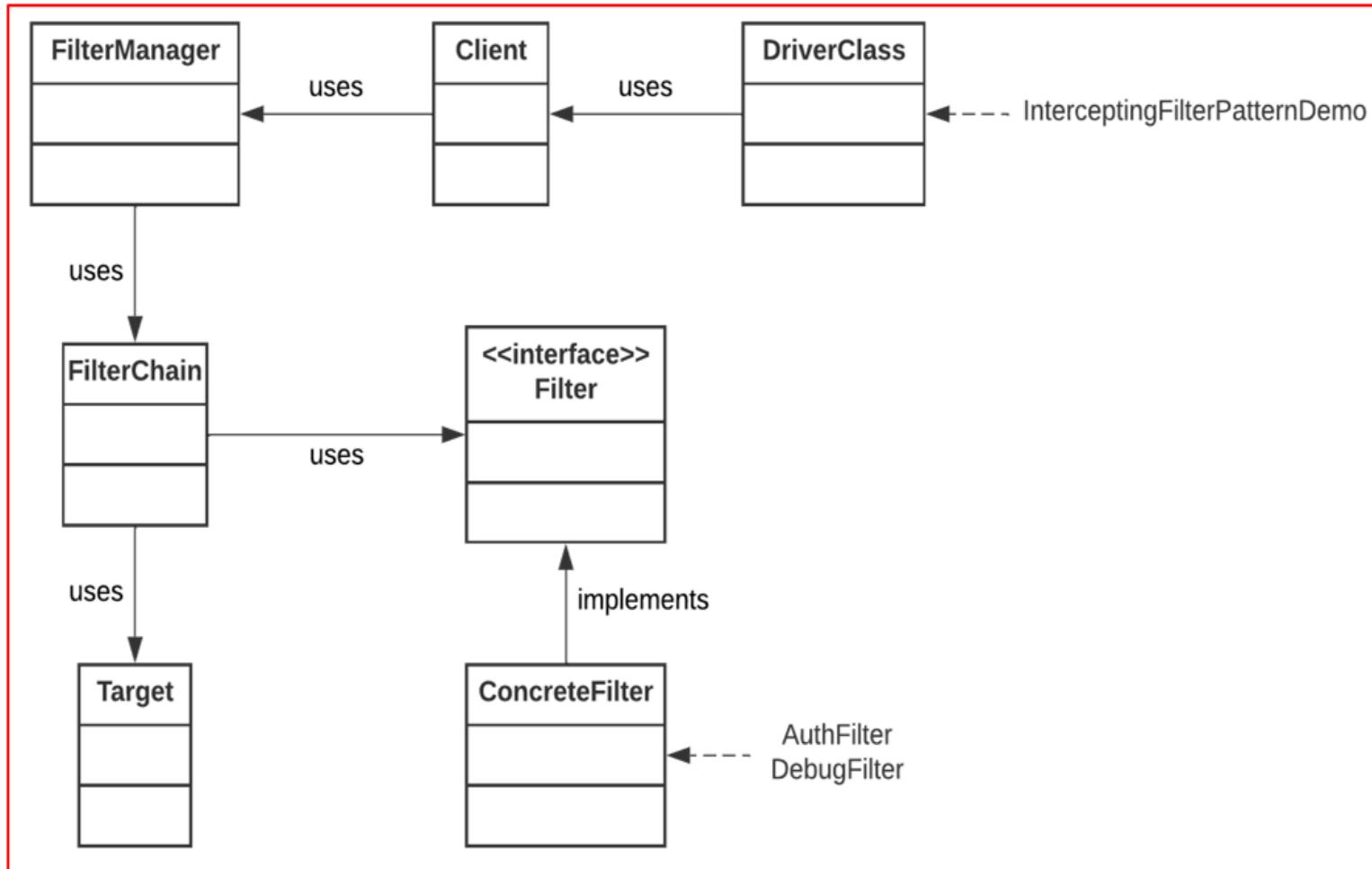
- Filter: Filters encapsulate specific cross-cutting concerns and intercept requests and responses.
- They perform processing or manipulation tasks and can be chained together to create a pipeline of filters.
- Filter Chain: The Filter Chain represents the collection of filters that are applied sequentially to intercept requests and responses.
- Each filter in the chain can modify or enrich the request or response data before passing it to the next filter or the core processing logic.
- Target: The Target represents the core processing logic or the component that the request is ultimately forwarded to after passing through the filter chain.



Intercepting Filter Design Pattern



Intercepting Filter UML Diagram



Service Locator/ Dependency Injection Pattern



- The service locator pattern is a design pattern used in software development to encapsulate the processes involved in obtaining a service with a strong abstraction layer.
- This pattern uses a central registry known as the “service locator” which on request returns the information necessary to perform a certain task.
- The Service Locator is responsible for returning instances of services when they are requested for by the service consumers or the service clients.



Transfer Object Pattern

- A Data Transfer Object (DTO) is a design pattern used to encapsulate and transfer data between different layers or components of a software application.
- DTOs are simple, lightweight, and contain only the necessary data fields required for a specific operation or communication between different parts of an application.
- They serve as a container for data, allowing us to transport information between various parts of our application without exposing the underlying implementation details.



Why Use DTOs?

- Reduced Overhead: DTOs allow us to send only the data needed for a particular task, minimizing the amount of data transmitted over the network or passed between application layers.
- This reduces the overhead associated with unnecessary data transfer.
- Data Transformation: DTOs provide a structured way to transform data from one format to another, which is particularly useful when dealing with data received from external sources or APIs.
- Enhanced Security: By limiting the data exposed to external components or layers, DTOs can help improve security by preventing sensitive information from being inadvertently exposed.

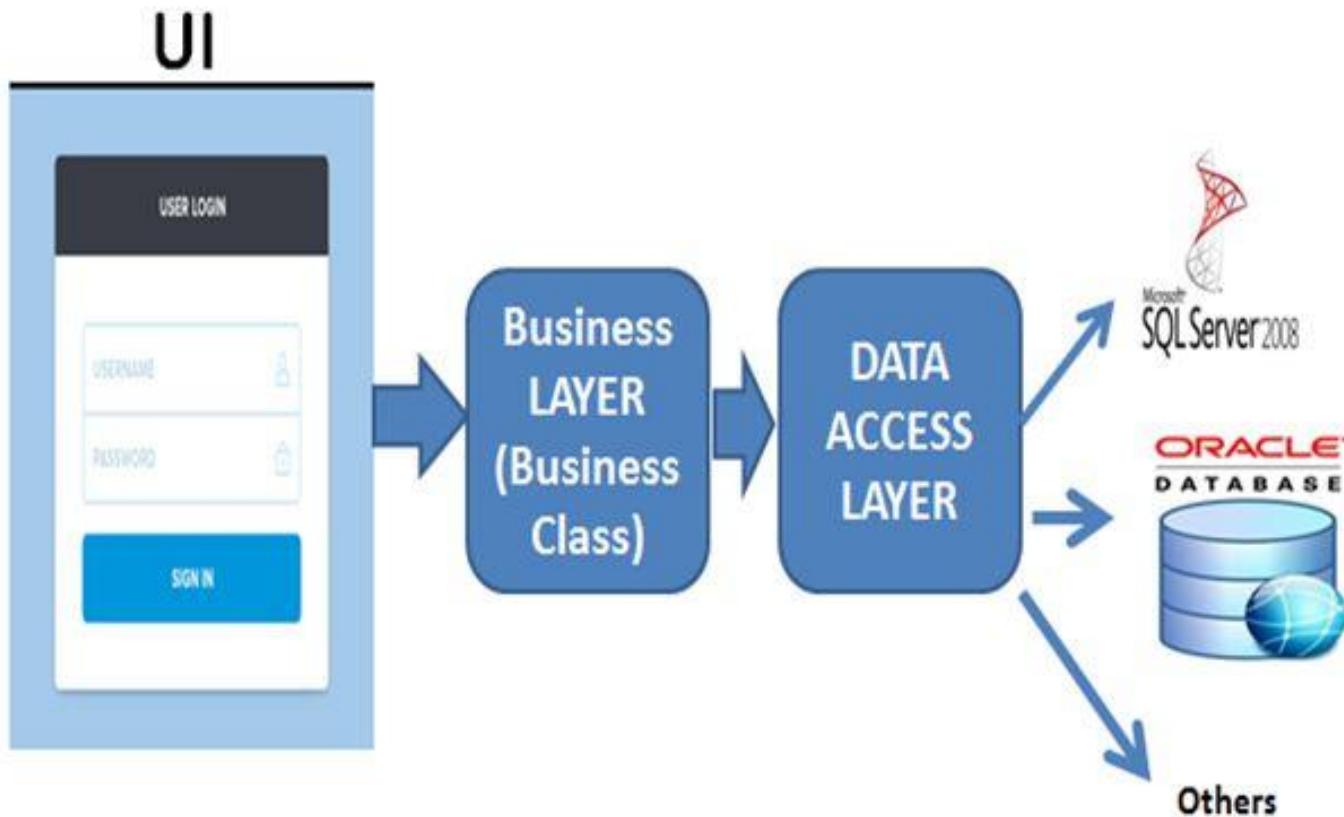


Why Use DTOs?

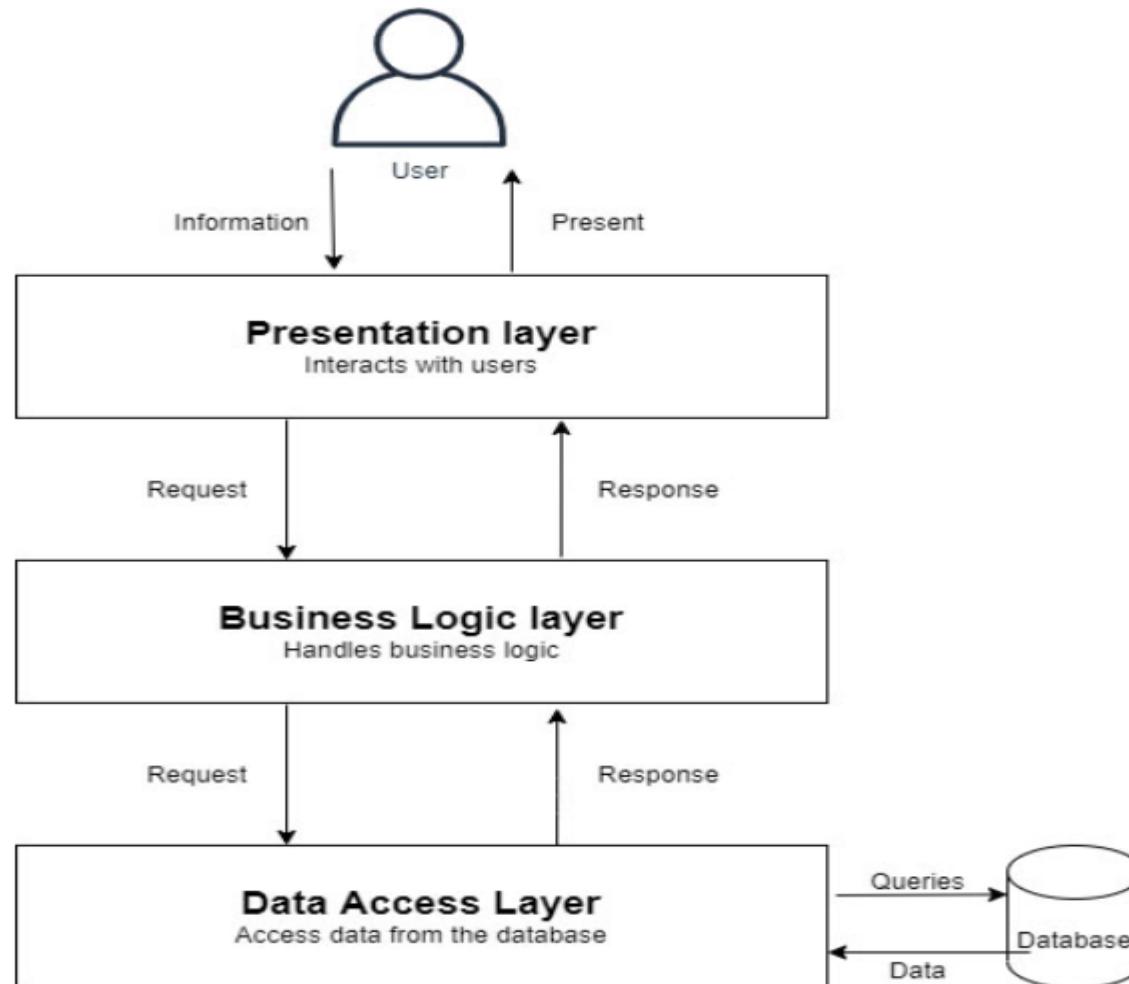
- Versioning and Compatibility: DTOs can be versioned independently of the underlying data model, making it easier to handle changes and updates to the data structure without affecting the entire application.
- Performance Optimization: When working with large datasets, DTOs allow us to select and transmit only the necessary data fields, resulting in improved performance and reduced latency.



Layered Pattern

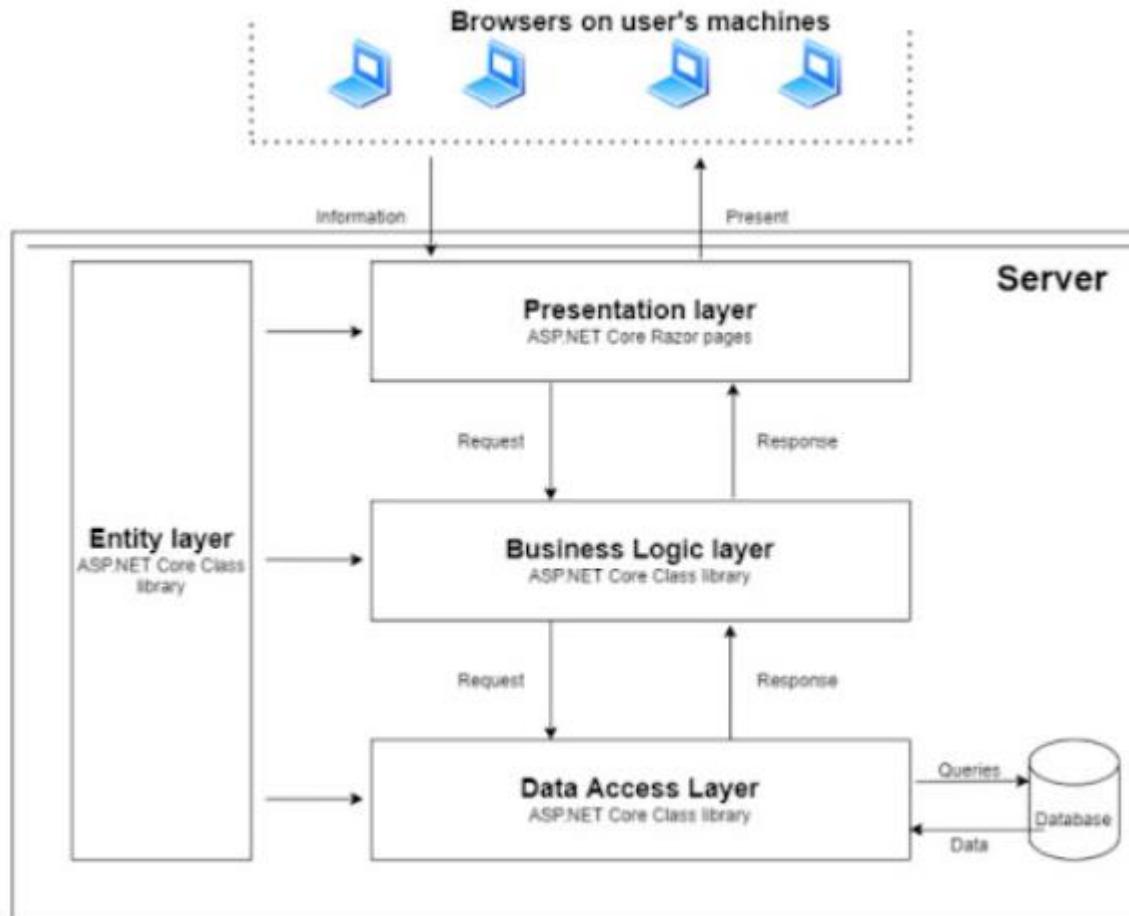


Layered Pattern

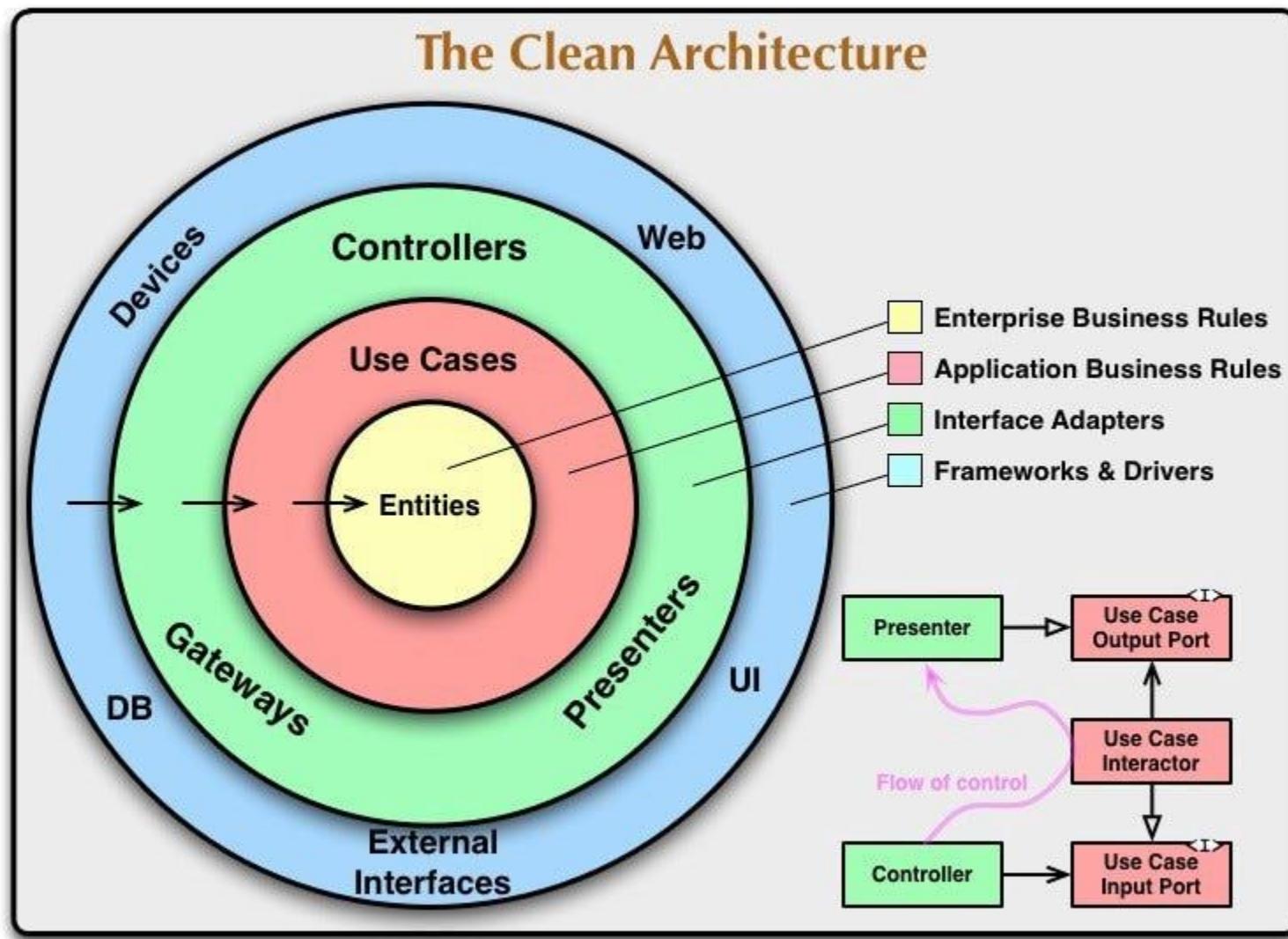




Layered Pattern



Layered Pattern





Categorization of EAA

- Domain-logic patterns
- Data-source architectural patterns
- Object relational behavioural patterns
- Object relational structural patterns
- Object relational metadata-mapping patterns
- Web presentation patterns
- Distribution patterns
- Offline concurrency patterns
- Session-state patterns
- Base patterns

Refactoring

Refactoring is a systematic process of improving code without creating new functionality that can transform a mess into clean code and simple design.

Dirty Code

While dirty code is often result of laziness and incompetence, it can also pile-up due to shortcuts taken during development process.

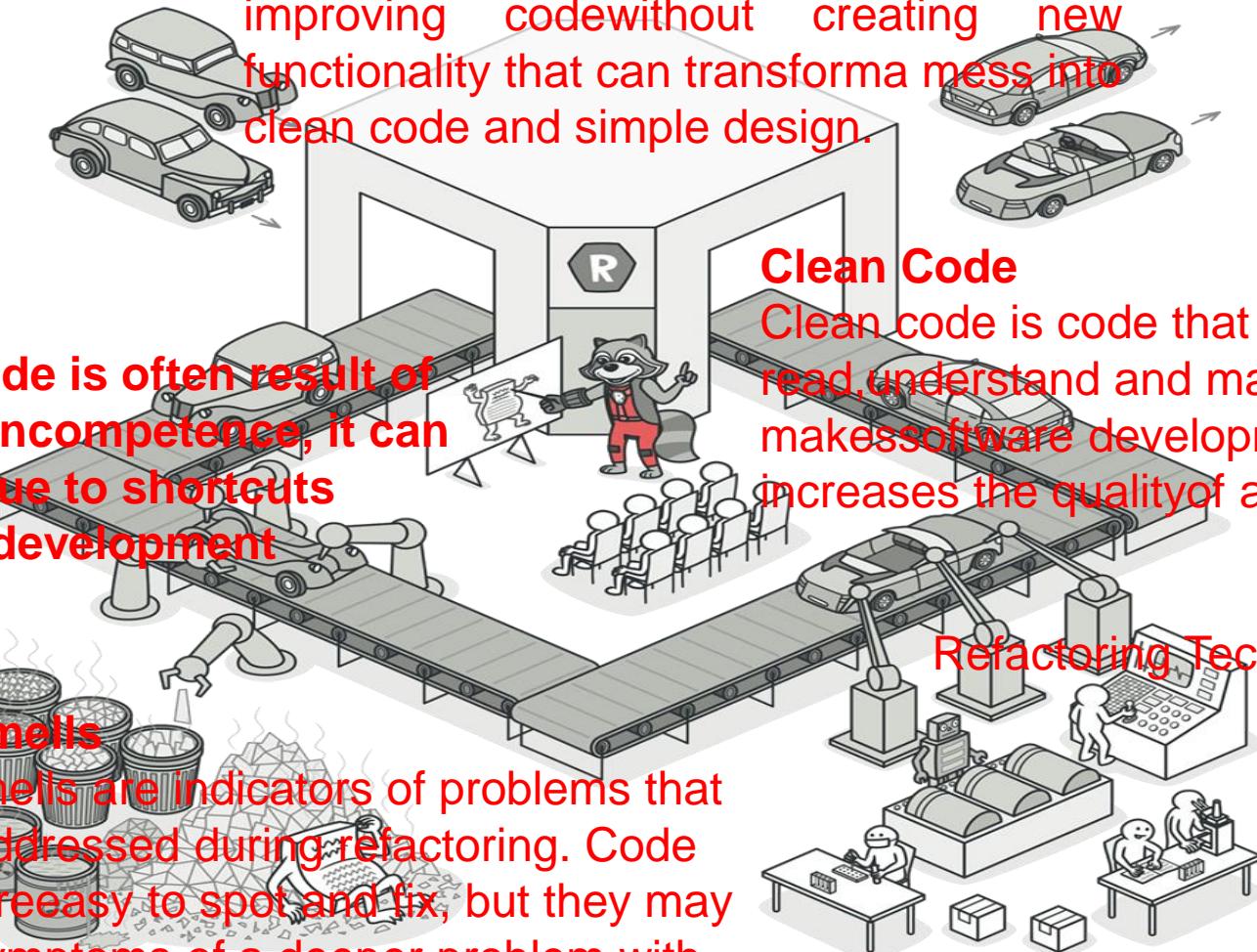
Code Smells

Code smells are indicators of problems that can be addressed during refactoring. Code smells are easy to spot and fix, but they may be just symptoms of a deeper problem with code.

Clean Code

Clean code is code that is easy to read, understand and maintain. Clean code makes software development predictable and increases the quality of a resulting product.

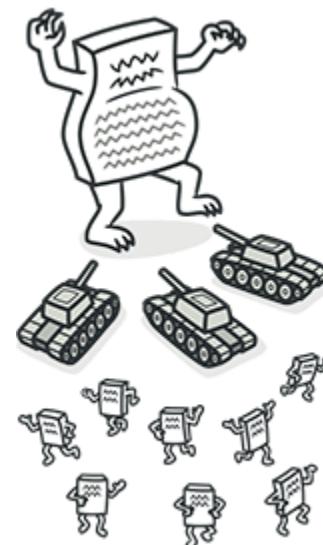
Refactoring Techniques



Code Smells

Bloaters

- Bloaters are code, methods and classes that have increased to such gargantuan proportions that they are hard to work with. Usually these smells do not crop up right away, rather they accumulate over time as the program evolves (and especially when nobody makes an effort to eradicate them).
- Long Method
- Large Class
- Primitive Obsession
- Long Parameter List



Code Smells



Object-Orientation Abusers

All these smells are incomplete or incorrect application of object-oriented programming principles.

Alternative Classes with Different Interfaces

Refused Request

Switch Statements



Code Smells

Change Preventers

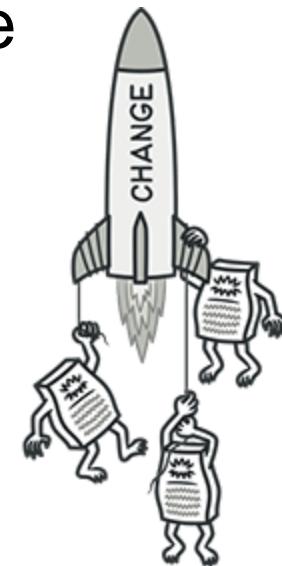
These smells mean that if you need to change something in one place in your code, you have to make many changes in other places too.

Program development becomes much more complicated and expensive as a result.

Divergent Change

Parallel Inheritance Hierarchies

Shotgun Surgery



Code Smells

Dispensables

A dispensable is something pointless and unneeded whose absence would make the code cleaner, more efficient and easier to understand.

Comments

Duplicate Code

Data Class

Dead Code

Lazy Class

Speculative Generality



Code Smells

Couplers

All the smells in this group contribute to excessive coupling between classes or show what happens if coupling is replaced by excessive delegation.

Feature Envy

Inappropriate Intimacy

Incomplete Library Class

Message Chains



Refactoring Techniques

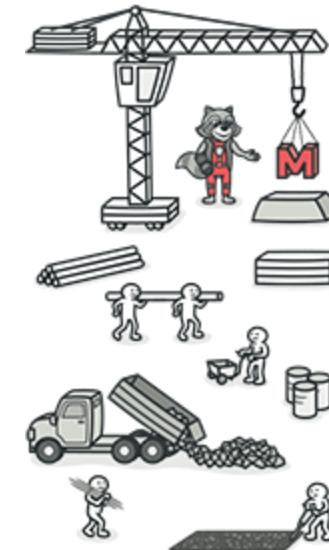
Composing Methods

- Much of refactoring is devoted to correctly composing methods. In most cases, excessively long methods are the root of all evil. The vagaries of code inside these methods conceal the execution logic and make the method extremely hard to understand – and even harder to change.
- The refactoring techniques in this group streamline methods, remove code duplication, and pave the way for future improvements.
- Extract Method
- Inline Method
- Extract Variable
- Inline Temp



Refactoring Techniques

- Replace Temp with Query
- Split Temporary Variable
- Remove Assignments to Parameters
- Replace Method with Method Object
- Substitute Algorithm



Refactoring Techniques

Moving Features between Objects

- Even if you have distributed functionality among different classes in a less-than-perfect way, there is still hope.
- These refactoring techniques show how to safely move functionality between classes, create new classes, and hide implementation details from public access.

- Move Method
- Move Field
- Extract Class
- Inline Class



Refactoring Techniques

- Hide Delegate
- Remove Middle Man
- Introduce Foreign Method
- Introduce Local Extension



Refactoring Techniques

Organizing Data

- These refactoring techniques help with data handling, replacing primitives with rich class functionality. Another important result is untangling of class associations, which makes classes more portable and reusable.
- **Change Value to Reference**
- **Change Reference to Value**
- **Duplicate Observed Data**
- **Self Encapsulate Field**
- **Replace Data Value with Object**



Refactoring Techniques

- Replace Array with Object
- Change Unidirectional Association to Bidirectional
- Change Bidirectional Association to Unidirectional
- Encapsulate Field
- Encapsulate Collection
- Replace Magic Number with Symbolic Constant
- Replace Type Code with Class
- Replace Type Code with Subclasses
- Replace Type Code with State/Strategy
- Replace Subclass with Fields



Refactoring Techniques

Simplifying Conditional Expressions

- Conditionals tend to get more and more complicated in their logic over time, and there are yet more techniques to combat this as well.
- **Consolidate Conditional Expression**
- **Consolidate Duplicate Conditional Fragments**
- **Decompose Conditional**
- **Replace Conditional with Polymorphism**
- **Remove Control Flag**
- **Replace Nested Conditional with Guard Clauses**



Refactoring Techniques



Simplifying Method Calls

These techniques make method calls simpler and easier to understand. This, in turn, simplifies the interfaces for interaction between classes.

Add Parameter

Remove Parameter

Rename Method

Separate Query from Modifier

Parameterize Method

Introduce Parameter Object

Preserve Whole Object



Refactoring Techniques

Remove Setting Method

Replace Parameter with Explicit Methods

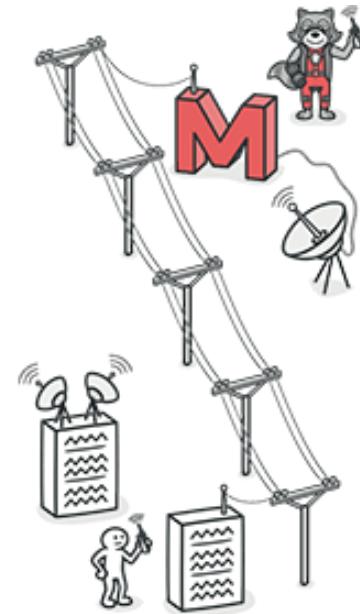
Replace Parameter with Method Call

Hide Method

Replace Constructor with Factory Method

Replace Error Code with Exception

Replace Exception with Test



Refactoring Techniques

Dealing with Generalization

Abstraction has its own group of refactoring techniques, primarily associated with moving functionality along the class inheritance hierarchy, creating new classes and interfaces, and replacing inheritance with delegation and vice versa.

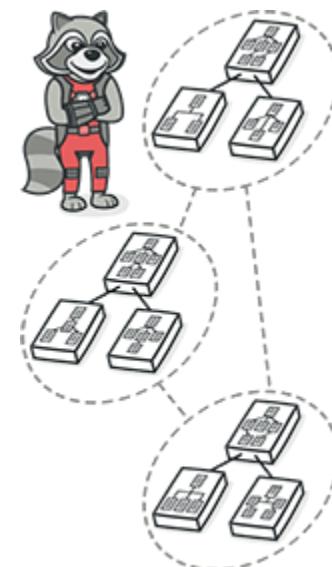
Pull Up Field

Pull Up Method

Pull Up Constructor Body

Push Down Field

Push Down Method



Refactoring Techniques

Extract Subclass

Extract Superclass

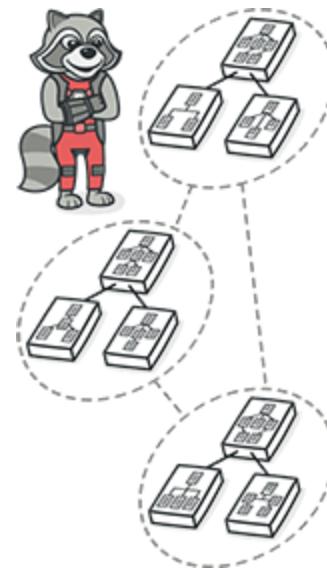
Extract Interface

Collapse Hierarchy

Form Template Method

Replace Inheritance with Delegation

Replace Delegation with Inheritance



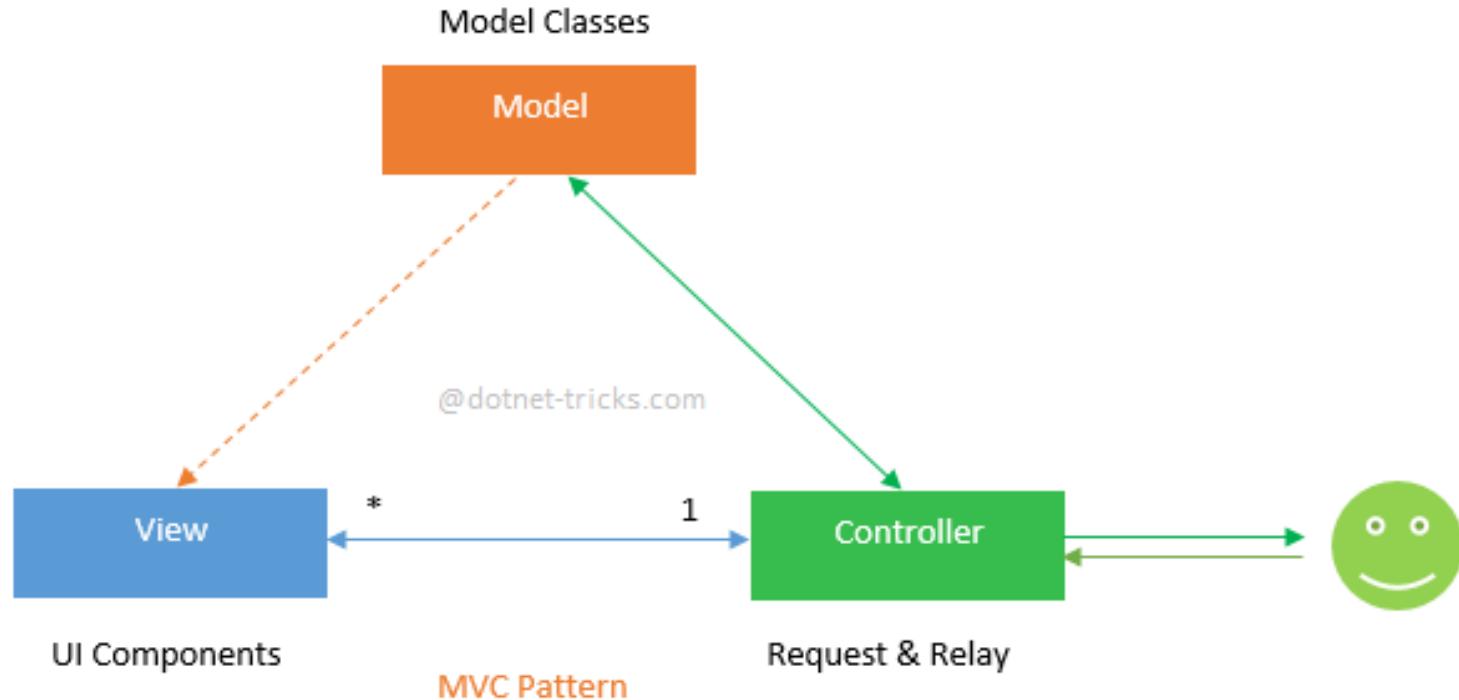
MVC Pattern

- MVC stands for Model-View-Controller. It is a software design pattern which was introduced in 1970s.
- Also, MVC pattern forces a separation of concerns, it means domain model and controller logic are decoupled from user interface (view).
- As a result maintenance and testing of the application become simpler and easier.
- MVC design pattern splits an application into three main aspects: Model, View and Controller

MVC, MVP and MVVM



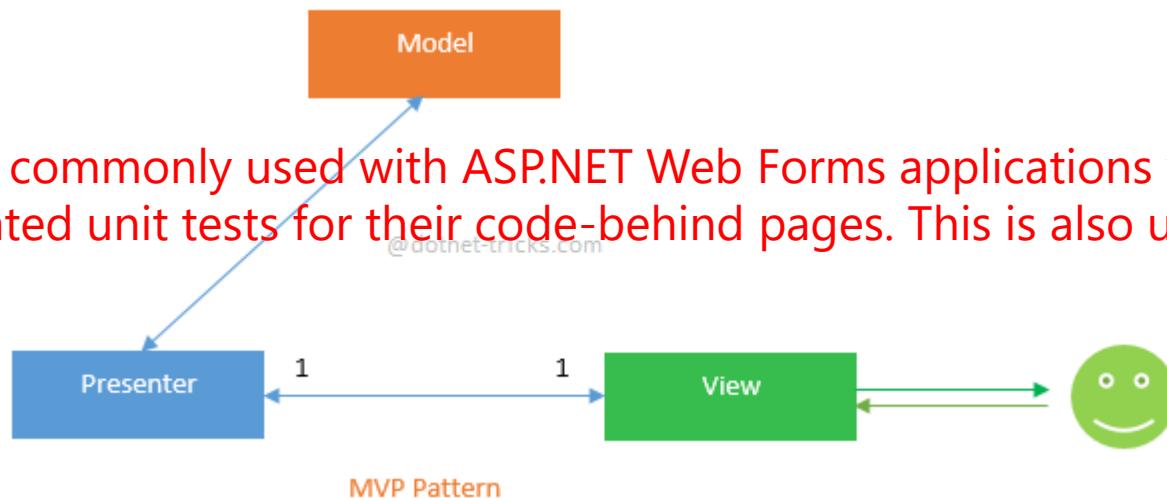
Today, this pattern is used by many popular framework like as Ruby on Rails, Spring Framework, Apple iOS Development and ASP.NET MVC.



MVP pattern

- This pattern is similar to MVC pattern in which controller has been replaced by the presenter. This design pattern splits an application into three main aspects: Model, View and Presenter.

This pattern is commonly used with ASP.NET Web Forms applications which require to create automated unit tests for their code-behind pages. This is also used with windows forms.



Presenter

- The Presenter is responsible for handling all UI events on behalf of the view.
- This receive input from users via the View, then process the user's data with the help of Model and passing the results back to the View.
- Unlike view and controller, view and presenter are completely decoupled from each other's and communicate to each other's by an interface.
- Also, presenter does not manage the incoming request traffic as controller.



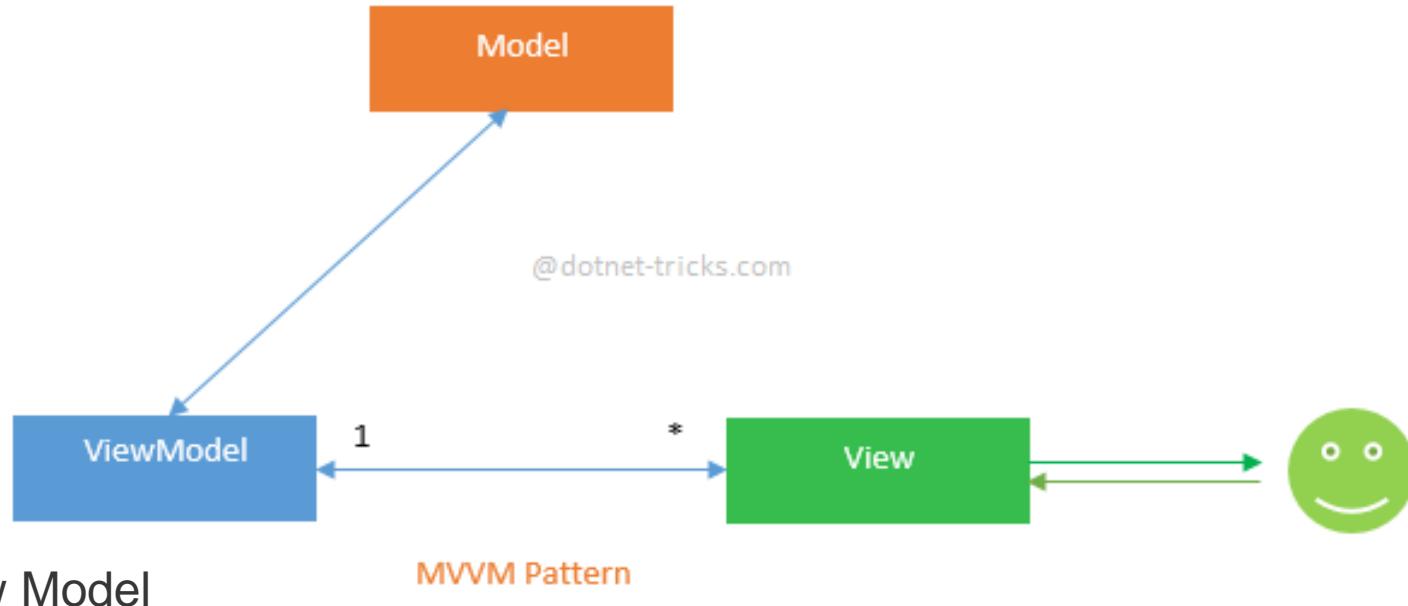
Key Points about MVP Pattern:

- User interacts with the View.
- There is one-to-one relationship between View and Presenter means one View is mapped to only one Presenter.
- View has a reference to Presenter but View has no reference to Model.
- Provides two way communication between View and Presenter.

MVVM pattern

- MVVM stands for Model-View-View Model. This pattern supports two-way data binding between view and View model.
- This enables automatic propagation of changes, within the state of view model to the View.
- Typically, the view model uses the observer pattern to notify changes in the view model to model.

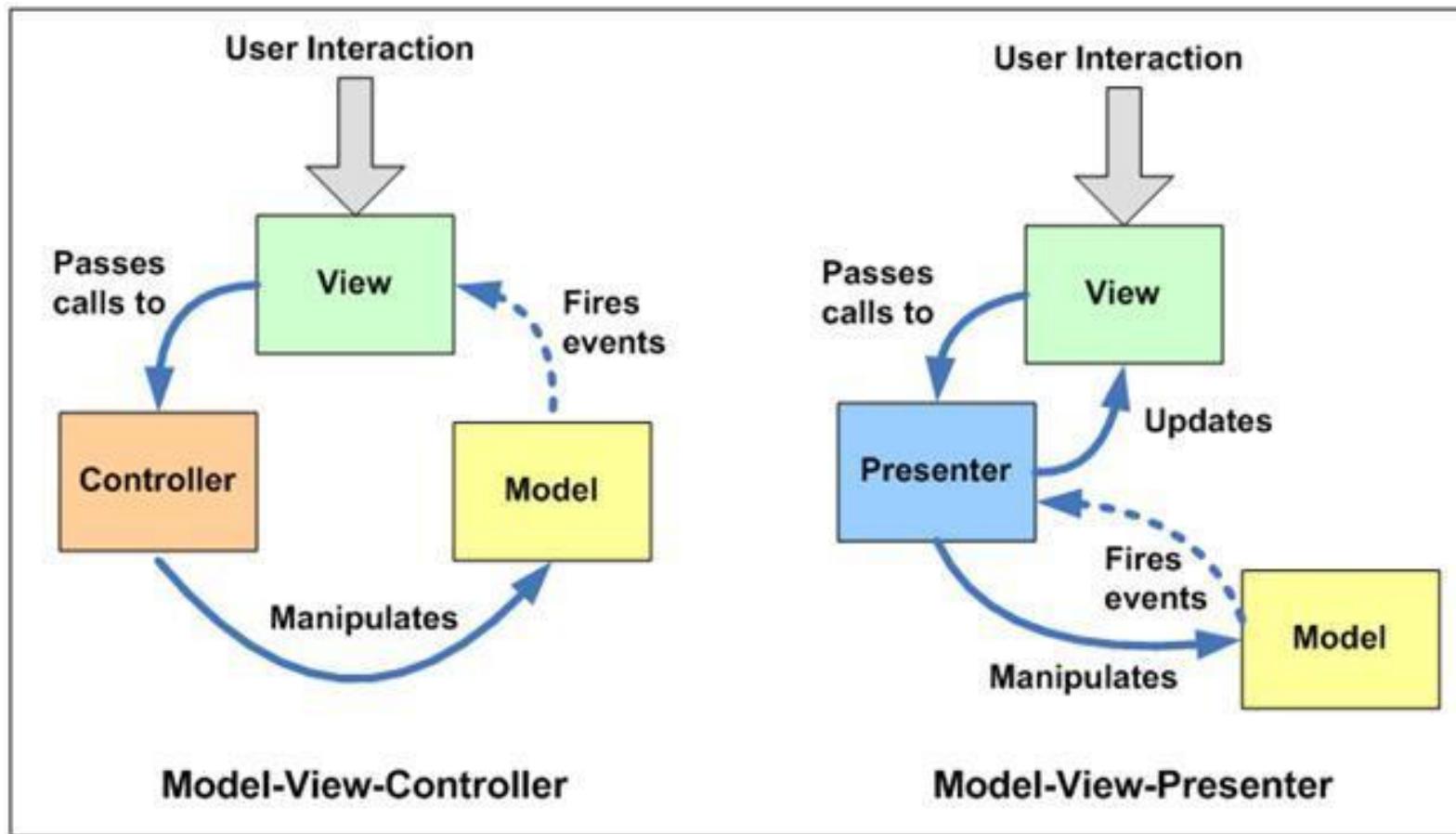
MVC, MVP and MVVM



The View Model is responsible for exposing methods, commands, and other properties that helps to maintain the state of the view, manipulate the model as the result of actions on the view, and trigger events in the view itself

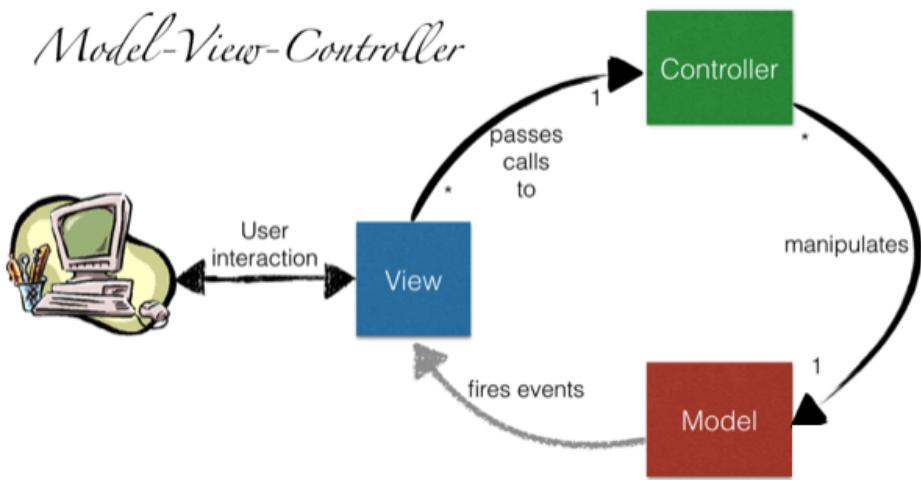


- Key Points about MVVM Pattern:
- User interacts with the View.
- There is many-to-one relationship between View and ViewModel means many View can be mapped to one ViewModel.
- View has a reference to ViewModel but View Model has no information about the View.
- Supports two-way data binding between View and ViewModel.

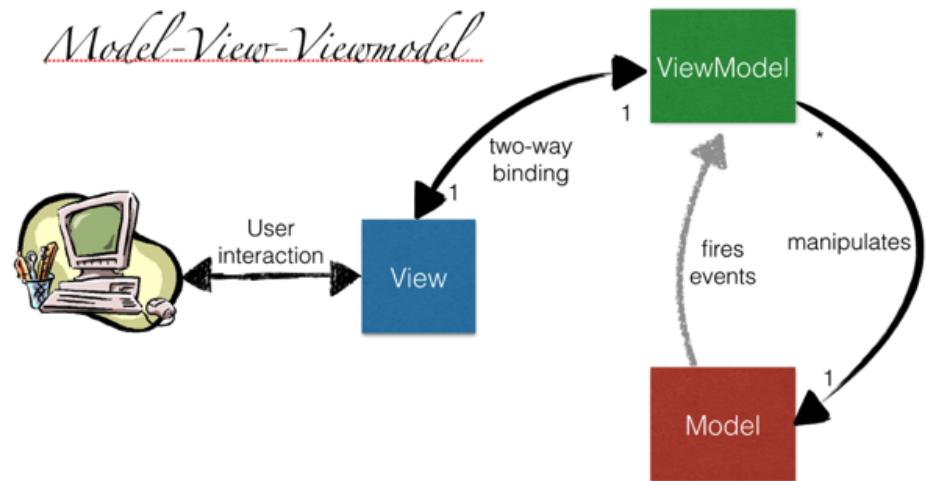




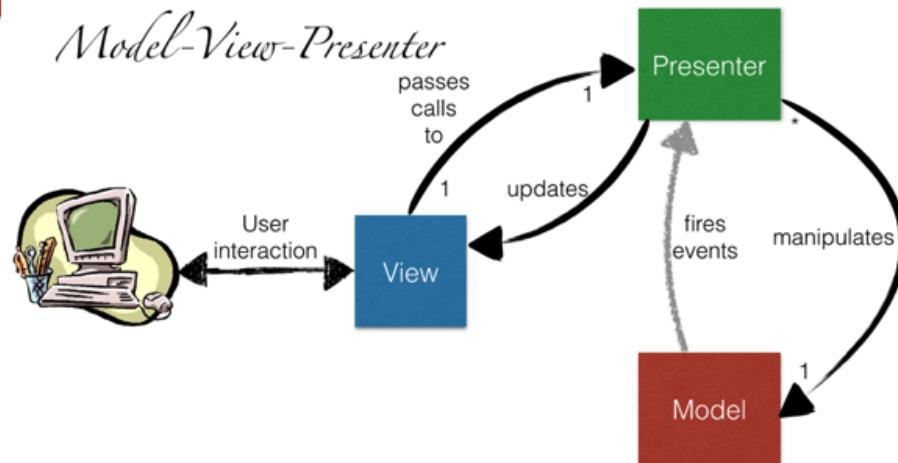
Model-View-Controller



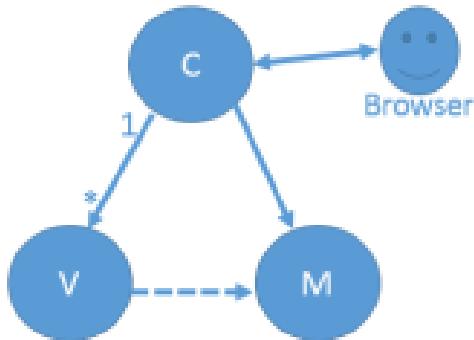
Model-View-Viewmodel



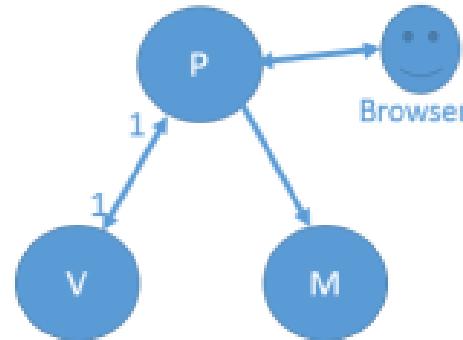
Model-View-Presenter



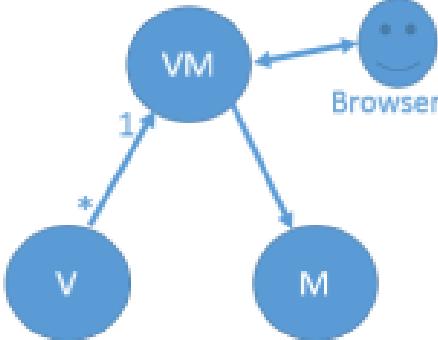
MVC – MVP - MVVM



- Controller is the entry point to the application
- One to Many relationship between Controller and View
- View does not have reference to the Controller
- View is very well aware of the Model
- Smalltalk, ASP.Net MVC

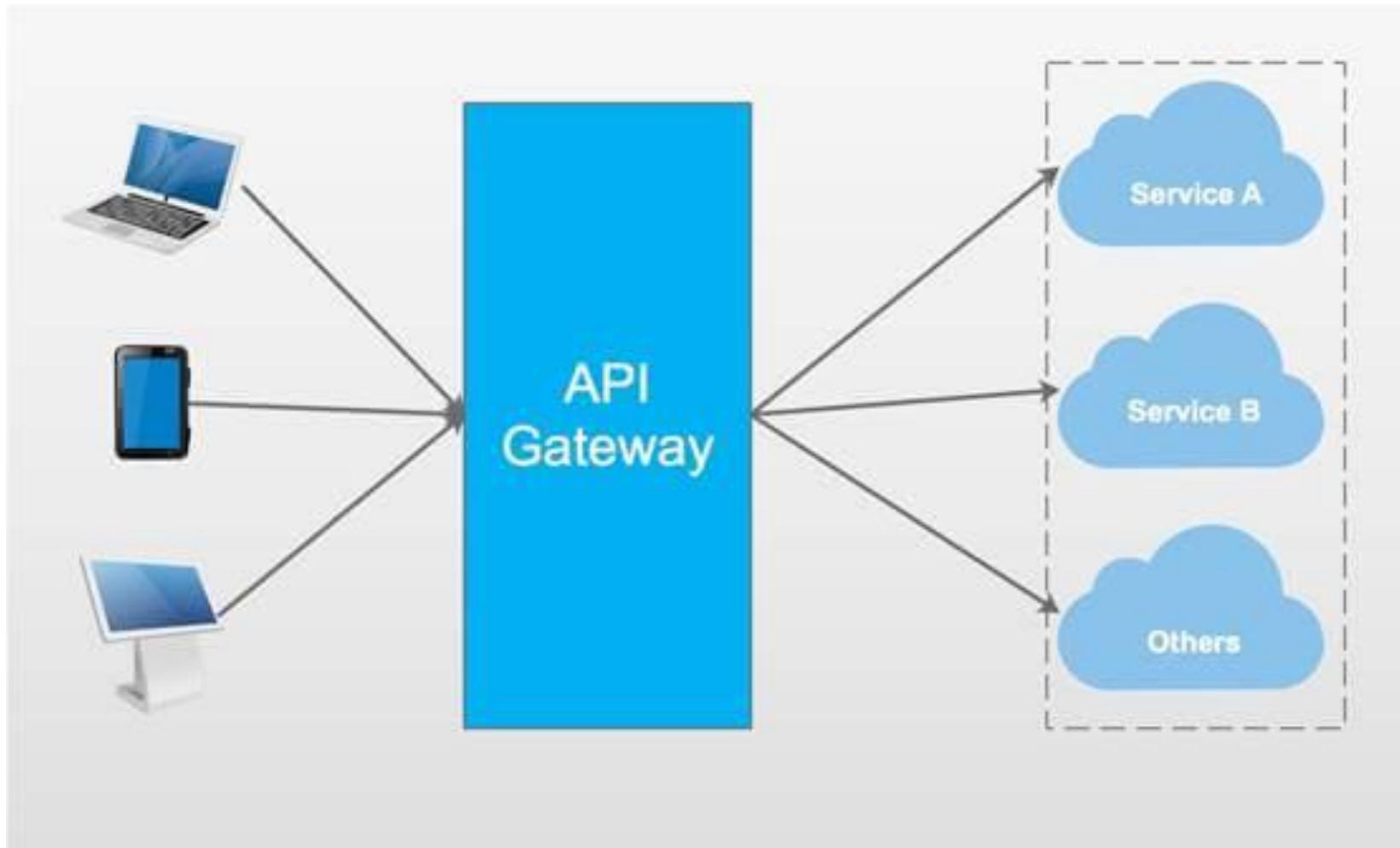


- View is the entry point to the application
- One to One mapping between View and Presenter
- View have the reference to the Presenter
- View is not aware of the Model
- Windows forms



- View is the entry point to the application
- One to Many relationship between View and View Model
- View have the reference to the View Model
- View is not aware of the Model
- Silverlight, WPF, HTML5 with Knockout/AngularJS

Ocelot API Gateway

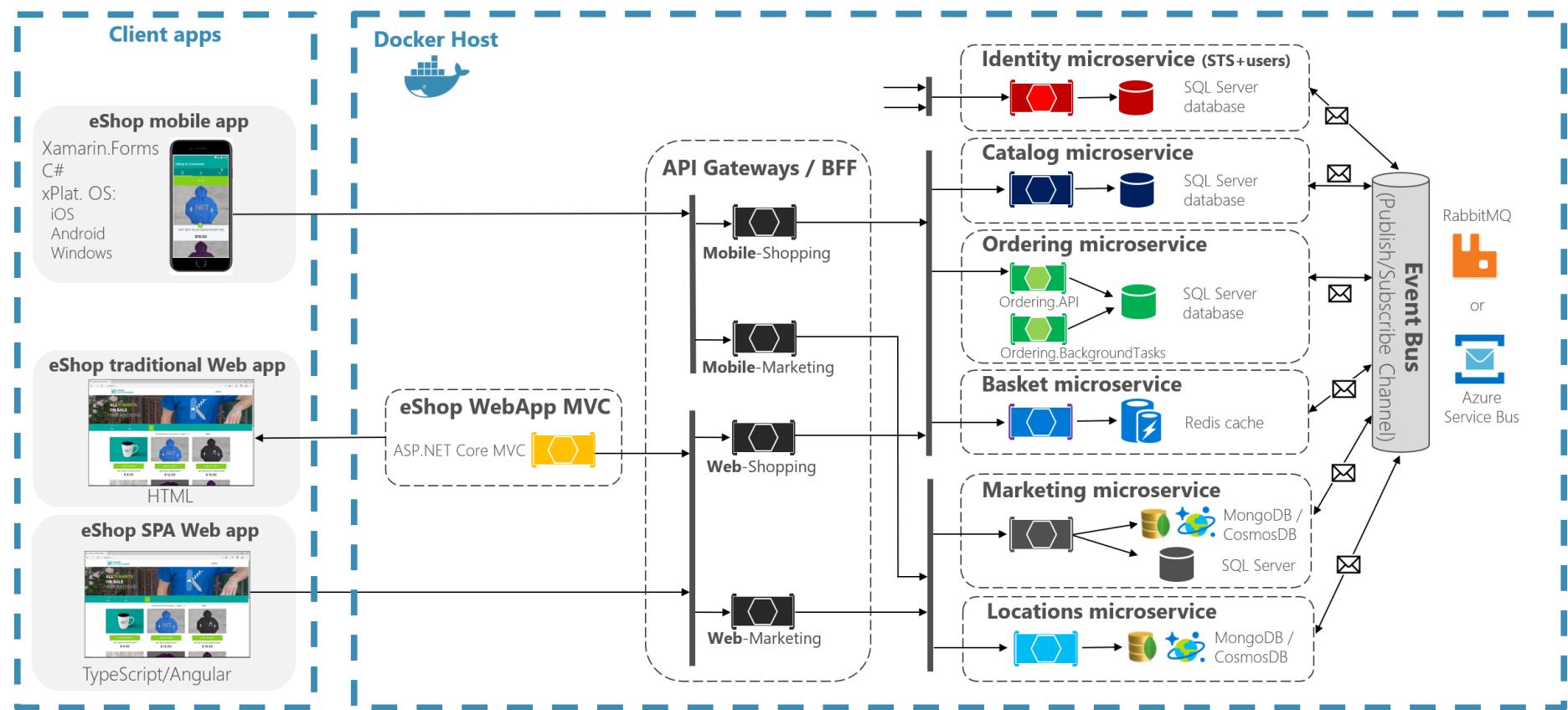




Microservice

eShopOnContainers reference application

(Development environment architecture)

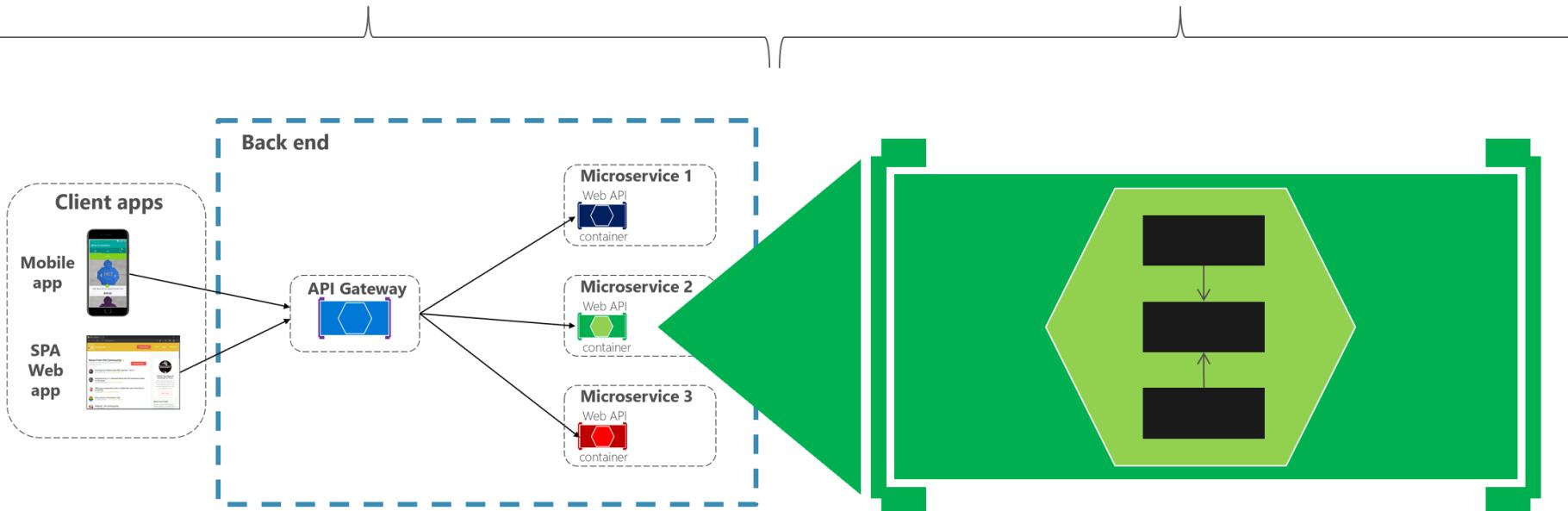




Microservice

External architecture
per application

Internal architecture
per microservice



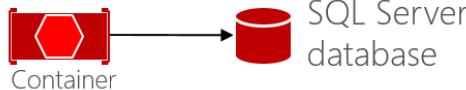
- External microservice patterns
- API Gateway
- Resilient communication
- Pub/Sub and event driven

- Simple data driven/CRUD design versus advanced design patterns, DDD, etc.
- Single or multiple libraries
- Dependency Injection, IoC, and SOLID



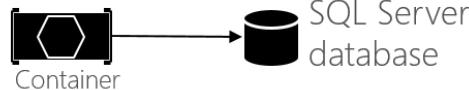
The Multi-Architectural-Patterns and polyglot microservices world

Microservice 1



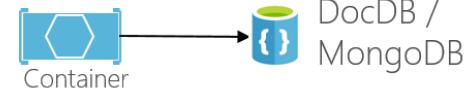
- **ASP.NET Core**
- Simple CRUD Design
- Entity Framework Core

Microservice 2



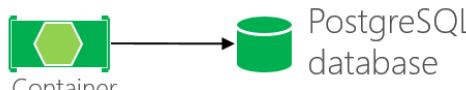
- **ASP.NET Core**
- DDD & CQRS patterns
- EF Core + Dapper

Microservice 3



- **ASP.NET Core**
- Queries projection
- DocDB/MongoDB API

Microservice 4



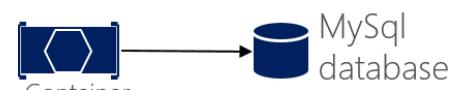
- **NancyFX (.NET Core)**
- Simple CRUD Design
- Massive

Microservice 5



- **ASP.NET Core**
- Simple CRUD Design
- Redis API

Microservice 6



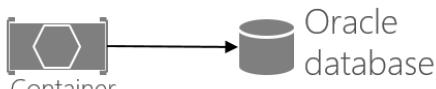
- **Node.js**
- Simple CRUD Design

Microservice 7



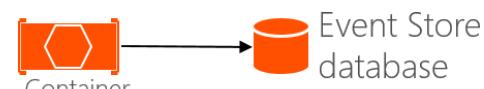
- **Python**
- Simple CRUD Design

Microservice 8



- **Java**
- DDD patterns

Microservice 9



- **ASP.NET Core**
- Event Sourcing patterns
- Event Store API

Microservice 10



- **SignalR (.NET Core 2)**
- Hub for Real Time comm.

Microservice 11



- **F# .NET Core**
- i.e. Calculus focused

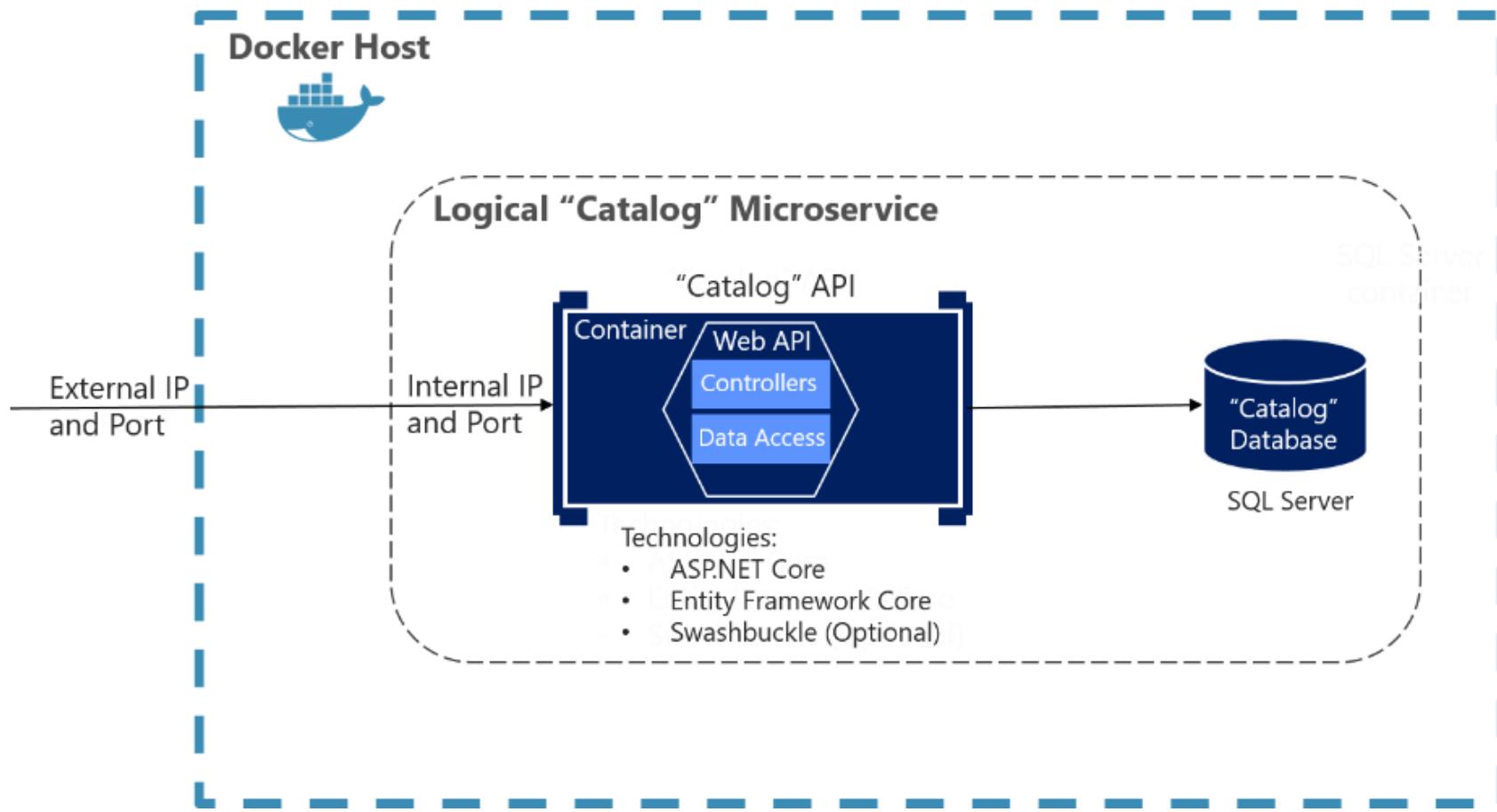
Microservice 10



- **GoLang**
- Stateless process



Data-Driven/CRUD microservice container





Dockerfile	docker-compose.override.yml
<p>This file is the entry point for running any Docker application.</p> <p>It is used to build an image of the application's published code in "obj/Docker/publish."</p>	<p>This file is used when running an application using Visual Studio.</p>
<pre>compose.override.yml ✘ X Dockerfile ✘ X CoreAppWithDocker FROM microsoft/aspnetcore:2.0 ARG source WORKDIR /app EXPOSE 80 COPY \${source:-obj/Docker/publish} . ENTRYPOINT ["dotnet", "CoreAppWithDocker.dll"]</pre>	<pre>docker-compose.override.yml ✘ X Dockerfile CoreAppWithDocker 1 version: '3' 2 3 services: 4 coreappwithdocker: 5 environment: 6 - ASPNETCORE_ENVIRONMENT=Development 7 ports: 8 - "80"</pre>



Dockerfile	docker-compose.override.yml
<pre>pose.override.yml ✘ X Dockerfile ✘ X CoreAppWithDocker FROM microsoft/aspnetcore:2.0 ARG source WORKDIR /app ENV ASPNETCORE_URLS http://+:83 EXPOSE 83 COPY \${source:-obj/Docker/publish} . ENTRYPOINT ["dotnet", "CoreAppWithDocker.dll"]</pre>	<pre>docker-compose.override.yml ✘ X Dockerfile CoreAppWithDock 1 version: '3' 2 3 services: 4 coreappwithdocker: 5 environment: 6 - ASPNETCORE_ENVIRONMENT=Development 7 ports: 8 - "83"</pre>



Docker ps commands

- Docker ps -a -q
 - Docker rm \$(docker ps -a -q)
-
- Dotnet restore
 - dotnet publish -o obj/Docker/publish
 - docker build -t imagename .
 - docker run -d -p 8001:83 –name core1 coreappimage

DSL

- A domain-specific language (DSL) is designed to express statements in a particular problem space, or domain.
- Well-known DSLs include regular expressions and SQL.
- Each DSL is much better than a general-purpose language for describing operations on text strings or a database, but much worse for describing ideas that are outside its own scope.
- Individual industries also have their own DSLs.
- For example, in the telecommunications industry, call description languages are widely used to specify the sequence of states in a telephone call, and in the air travel industry a standard DSL is used to describe flight bookings.



DSL Applications

- Plan of navigation paths in a website.
- Wiring diagrams for electronic components.
- Networks of conveyor belts and baggage handling equipment for an airport.



DSL benefits

- Contains constructs that exactly fit the problem space.
 - An insurance policy application must include elements for policies and claims.
 - A domain-specific language makes it easier to design the application, and find and correct errors of logic.
- Lets non-developers and people who do not know the domain understand the overall design.
 - By using a graphical domain-specific language, you can create a visual representation of the domain so that non-developers can easily understand the design of the application.



DSL benefits

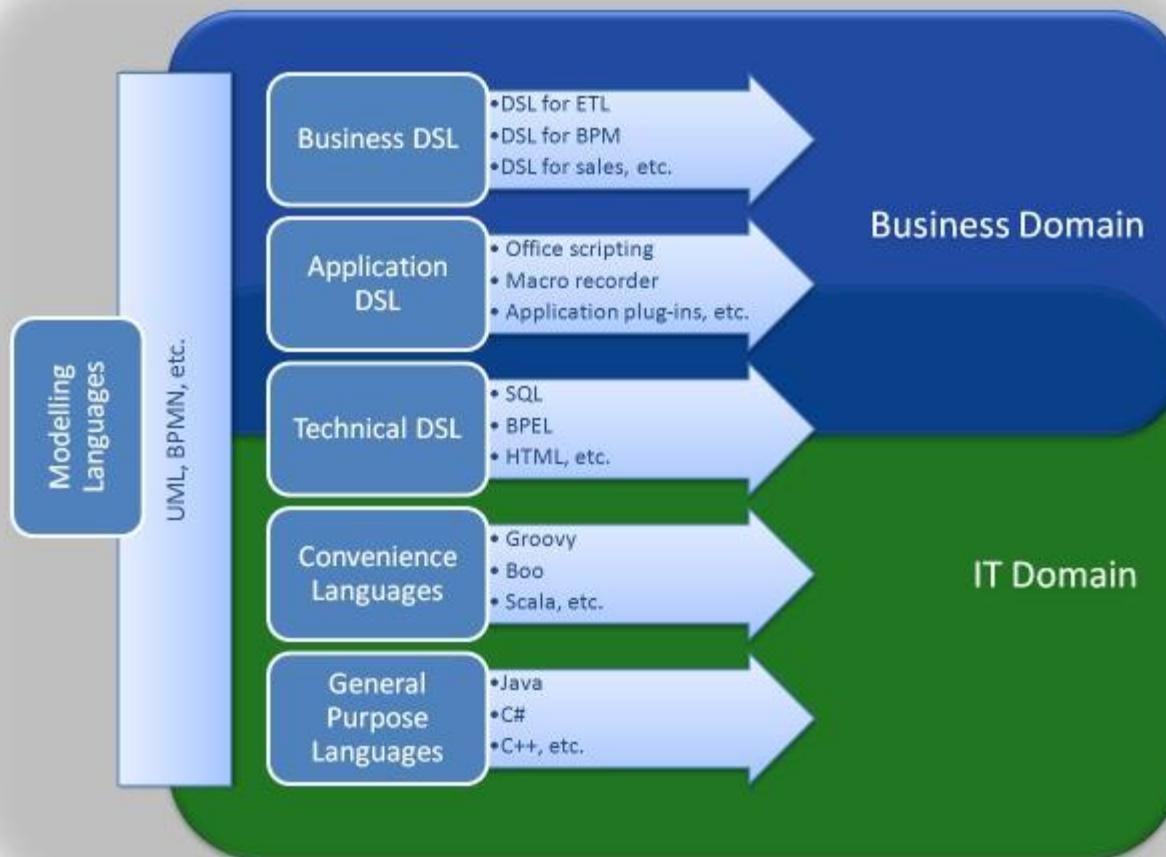
- Makes it easier to create a prototype of the final application.
 - Developers can use the code that their model generates to create a prototype application that they can show to clients.

The DSL Tools Solution



- Task Flow
- Class Diagrams
- Minimal Language
- Component Models
- Minimal WPF
- Minimal Windows.Forms
- DSL Library

Implementing a Domain Specific Language on .NET





Creational Pattern

- **Singleton Pattern**
- GoF Definition
 - Ensure a class has only one instance, and provide a global point of access to it.
- Concept
 - A particular class should have only one instance. You can use this instance whenever you need it and therefore avoid creating unnecessary objects.



Creational Pattern

Real-Life Example

- Suppose you are a member of a sports team and your team is participating in a tournament.
- When your team plays against another team, as per the rules of the game, the captains of the two sides must have a coin toss.
- If your team does not have a captain, you need to elect someone to be the captain first.
- Your team must have one and only one captain.



Creational Pattern

Computer World Example

- In some software systems, you may decide to maintain only one file system so that you can use it for the centralized management of resources.



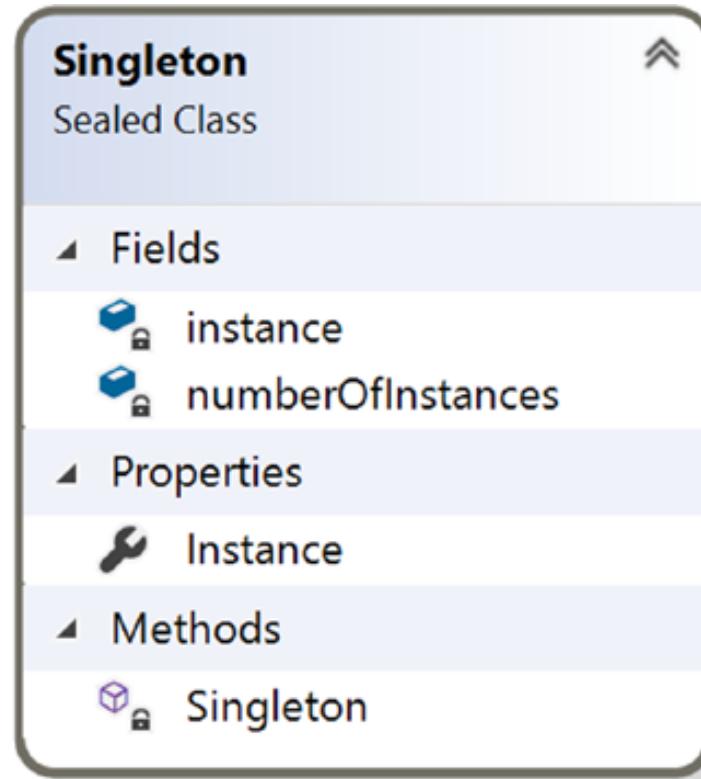
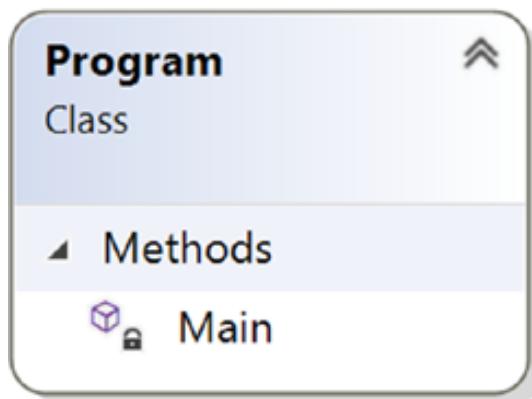
Creational Pattern

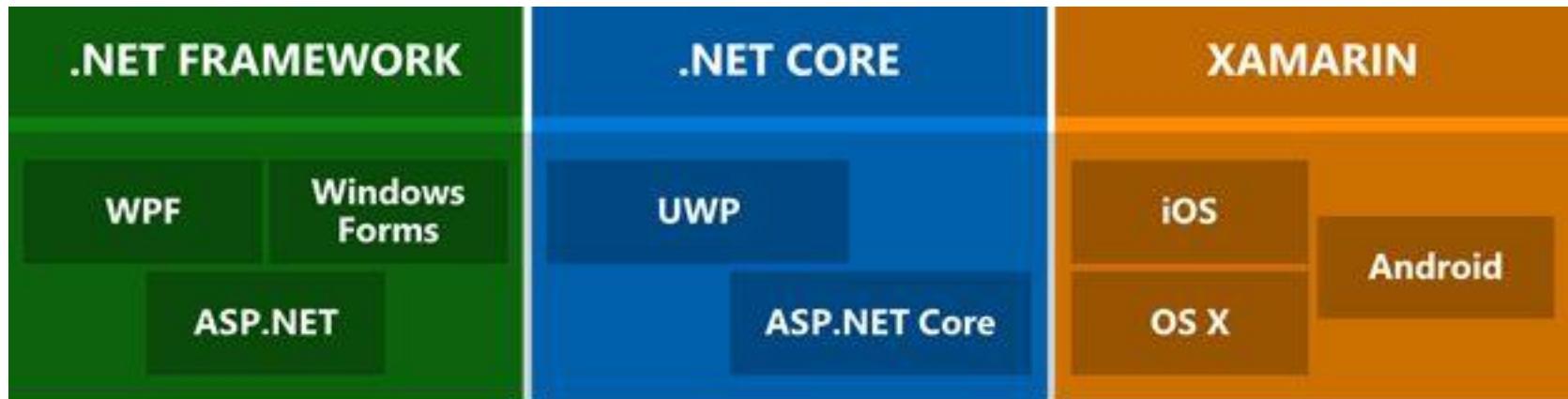
Illustration

- These are the key characteristics in the following implementation:
- The constructor is private in this example. So, you cannot instantiate in a normal fashion (using new).
- Before you attempt to create an instance of a class, you check whether you already have an available copy.
- If you do not have any such copy, you create it; otherwise, you simply reuse the existing copy.



Singleton Design Pattern





Build Tools

Languages

Runtime components



Decision

.NET Framework is a better choice if you:

- Do not have time to learn a new technology.
- Need a stable environment to work in.
- Have nearer release schedules.
- Are already working on an existing app and extending its functionality.
- Already have an existing team with .NET expertise and building production ready software.
- Do not want to deal with continuous upgrades and changes.
- Building Windows client applications using Windows Forms or WPF



Decision

- .NET Core is a better choice if you:
 - Want to target your apps on Windows, Linux, and Mac operating systems.
 - Are not afraid of learning new things.
 - Are not afraid of breaking and fixing things since .NET Core is not fully matured yet.
 - A student who is just learning .NET.
 - Love open source.



Decision

High-performance and scalable system without UI	.NET Core is much faster.
Docker containers support	Both. But .NET Core is born to live in a container.
Heavily reply on command line	.NET Core has a better support.
Cross-platform needs	.NET Core
Using Microservices	Both but .NET Core is designed to keep in mind today's needs.
User interface centric Web applications	.NET Framework is better now until .NET Core catches up.
Windows client applications using Windows Forms and WPF	.NET Framework
Already have a pre-configured environment and systems	.NET Framework is better.
Stable version for immediate need to build and deploy	.NET Framework has been around since 2001. .NET Core is just a baby.
Have existing experienced .NET team	.NET Core has a learning curve.
Time is not a problem. Experiments are acceptable. No rush to deployment.	.NET Core is the future of .NET.



Alternative Approach for single threaded model

```
public static Singleton Instance
{
    get
    {
        if (instance == null)
        {
            instance = new Singleton();
        }
        return instance;
    }
}
```



Alternative Approach for single threaded model

- //Double checked locking
 - using System;
 - public sealed class Singleton
- ```
{
//We are using volatile to ensure that
//assignment to the instance variable finishes before it's
//access.

private static volatile Singleton instance;
private static object lockObject = new Object();
private Singleton() { }
}
```



# Alternative Approach for single threaded model

```
public static Singleton Instance
{
 get
 {
 if (instance == null)
 {
 lock (lockObject)
 {
 if (instance == null)
 instance = new Singleton();
 }
 }
 return instance;
 }
}
```

# The middleman or event bus



- How do you achieve anonymity between publisher and subscriber? An easy way is let a middleman take care of all the communication. An event bus is one such middleman.
- An event bus is typically composed of two parts:
- The abstraction or interface.
- One or more implementations.



# Event bus (Publish/Subscribe channel)

## Event bus abstractions/interface

## Event bus implementations

RabbitMQ

Azure  
Service  
Bus

Other  
message /  
event broker



# Iterator Pattern

---

- GoF Definition
- Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.



# Iterator Pattern

---

- Concept
- Iterators are generally used to traverse a container (which is basically an object) to access its elements, but you do not need to deal with the element's internal details.
- You will frequently use the concepts of iterators when you want to traverse different kinds of collection objects in a standard and uniform way.
- This concept is used frequently to traverse the nodes of a tree-like structure. So, in many scenarios, you may notice the use of the Iterator pattern with the Composite pattern.
- C# has its own iterators that were introduced in Visual Studio 2005.
- The foreach statement is frequently used in this context..



# Iterator Pattern

---

- Real-Life Example
- Suppose there are two companies, Company A and Company B. Company A stores its employee records (say each employee's name, address, salary details, and so on) in a linked list data structure, and Company B stores its employee data in an array.
- One day the two companies decide to merge. The Iterator pattern becomes handy in such a situation because you do not need to write any code from scratch.
- In a situation like this, you can have a common interface through which you can access the data for both companies. So, you can simply call those methods without rewriting the code.

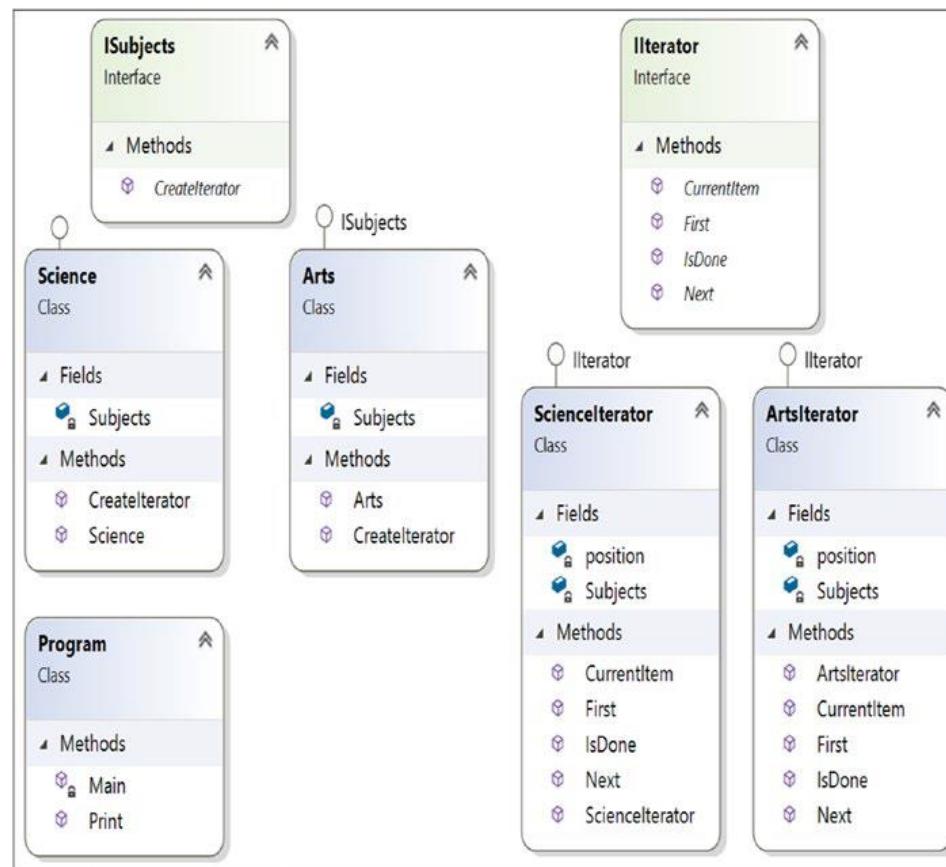


# Iterator Pattern

---

- Computer World Example
- Similarly suppose, in a college, the arts department is using an array data structure to maintain its student records, and the science department is using a linked list data structure to keep its student records.
- The administrative department does not care about the different data structures.
- It is simply interested in getting the data from each department and wants to access the data in a uniform way.

# Iterator Pattern





## Services

File Action View Help



## Services (Local)

## RabbitMQ

[Start the service](#)

Description:  
Multi-protocol open source  
messaging broker

```
D:\Program Files\RabbitMQ Server\rabbitmq_server-3.6.10\sbin>rabbitmq-plugins.bat enable rabbitmq_management --offline
The following plugins have been enabled:
 amqp_client
 cowlib
 cowboy
 rabbitmq_web_dispatch
 rabbitmq_management_agent
 rabbitmq_management
Offline change; changes will take effect at broker restart.

D:\Program Files\RabbitMQ Server\rabbitmq_server-3.6.10\sbin>rabbitmq-server.bat start

 RabbitMQ 3.6.10. Copyright (C) 2007-2017 Pivotal Software, Inc.
Licensed under the MPL. See http://www.rabbitmq.com/
##
Logs: C:/Users/BALASU~1/AppData/Roaming/RabbitMQ/log/RABBIT~1.LOG
C:/Users/BALASU~1/AppData/Roaming/RabbitMQ/log/RABBIT~2.LOG
#####
Starting broker...
completed with 6 plugins.
```

Extended / Standard /



Type here to search



8:27 12/12/2018 ENG



# Containerization

```
Administrator: RabbitMQ Command Prompt (sbin dir) - docker build -t dockerimageapp.

Microsoft Windows [Version 10.0.17134.471]
(c) 2018 Microsoft Corporation. All rights reserved.

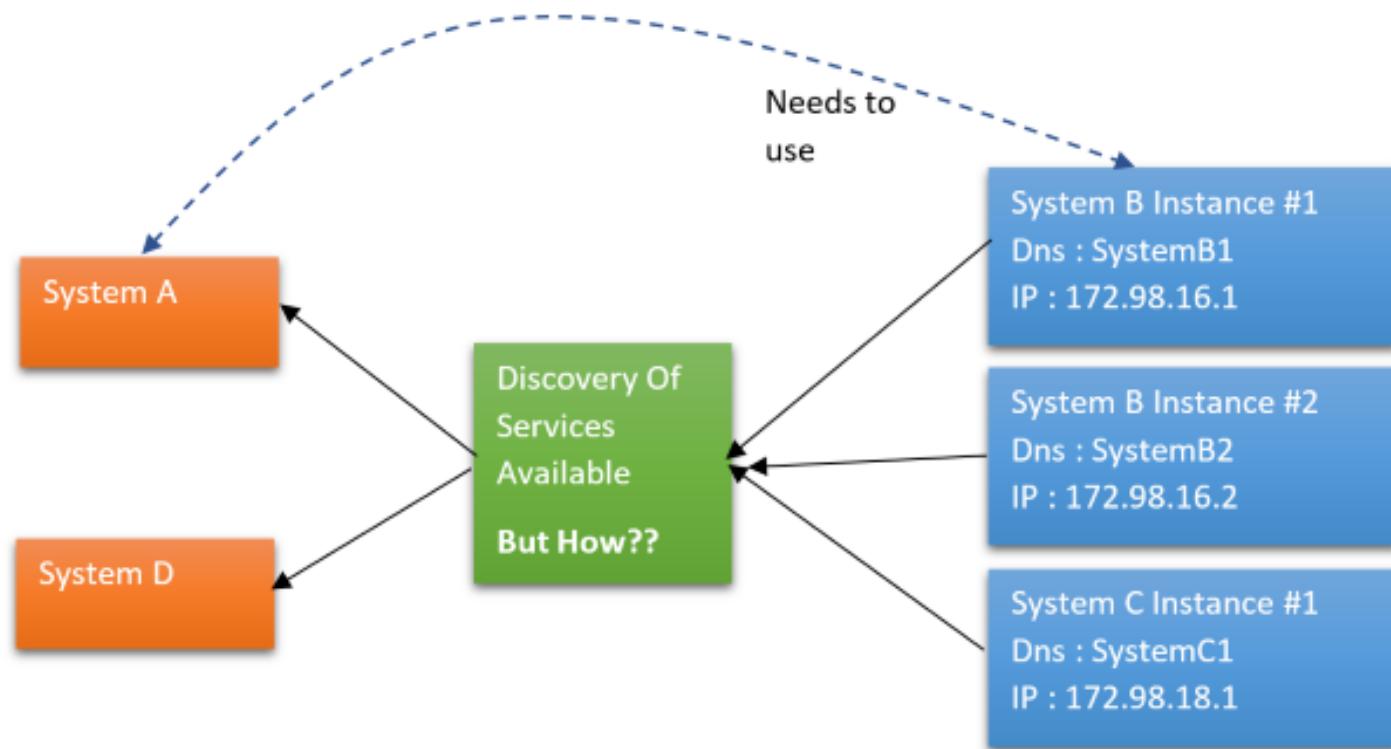
C:\WINDOWS\system32>cd G:\Local disk\dotnet_design_patterns\DockerContainerDemo\DockerContainerDemo

C:\WINDOWS\system32>g:

G:\Local disk\dotnet_design_patterns\DockerContainerDemo\DockerContainerDemo>docker build -t dockerimageapp .
Sending build context to Docker daemon 921.1kB
Step 1/7 : FROM microsoft/dotnet:onbuild
onbuild: Pulling from microsoft/dotnet
43c265008fae: Pull complete
af36d2c7a148: Pull complete
143e9d501644: Pull complete
882c59fec304: Pull complete
e8fd1a99b43e: Downloading 26.22MB/51.85MB
e4f74ce87b0c: Downloading 7.493MB/82.12MB
ee0b3487643d: Download complete
[...]
```



# Discovery Pattern



# Existing solutions



- Why do developers choose Zookeeper?
- High performance ,easy to generate node specific config
- Kafka support
- Java
- Supports extensive distributed IPC
- Spring Boot Support
- Supports DC/OS
- Embeddable In Java Service
- Used in Hadoop

# Existing solutions



## Consul

- Infrastructure
- Health checking
- Distributed key-value store
- Monitoring
- High-availability
- Web-UI
- Token-based acls
- Gossip clustering
- Dns server
- Docker integration

# Existing solutions



- Redis/DataStore
- You could use Redis cache to be used by services to store meta data, and then a consumer could query the cache.
- However to make this resilient you really need some form of clustering, and some consensus/gossip to achieve consistency.
- This is fairly hard so most people just cut corners and make this a singleton, which is obviously a single point of failure.
- .

# Existing solutions



- Kubernetes
- Kubernetes does a good job of "Discovery" by way of Services/DNS addon/pods all of which can easily be load balanced.
- This is a great solution to run in containers (preferably in a cloud environment)
- Consul
- Consul (to my mind) is the only tool/framework that tackles "discovery" head on, and actually provides a rich tool that does that this job, and does it well, with little effort from the developer.

# Configuring Consul in Local workstation



- Download from Consul portal. Choose particular package based on the operating System. Once downloaded the zip, we need to unzip it to desired place.
- Start Consul Agent in local workstation – The Zip file that we have unzipped, has only one exe file called `consul.exe`. We will start a command prompt here and use below command to start the agent.
- IPV4 is binding address.
- `consul agent -server -bootstrap-expect=1 -data-dir=consul-data -ui -bind=192.168.99.1`

# Configuring Consul in Local workstation



```
C:\Windows\system32\cmd.exe - consul agent -server -bootstrap-expect=1 -data-dir=consul-data -ui -bind=192.168.6.1

F:\Study\installations\consul_0.9.0_windows_amd64>consul agent -server -bootstrap-expect=1 -data-dir=consul-data -ui -bind=192.168.6.1
==> WARNING: BootstrapExpect Mode is specified as 1; this is the same as Bootstrap mode.
==> WARNING: Bootstrap mode enabled! Do not enable unless necessary
==> Starting Consul agent...
==> Consul agent running!
 Version: 'v0.9.0'
 Node ID: 'f8db8d09-ac67-e997-9306-aeb20e4ecd3e'
 Node name: 'Sajal-HP'
 Datacenter: 'dc1'
 Server: true <Bootstrap: true>
 Client Addr: 127.0.0.1 <HTTP: 8500, HTTPS: -1, DNS: 8600>
 Cluster Addr: 192.168.6.1 <LAN: 8301, WAN: 8302>
 Gossip encrypt: false, RPC-TLS: false, TLS-Incoming: false

==> Log data will now stream in as it occurs:

2017/07/20 13:31:42 [INFO] raft: Initial configuration <index=1>: [{Suffrage:Voter ID:192.168.6.1:8300 Address:192.168.6.1:8300}]
2017/07/20 13:31:42 [INFO] raft: Node at 192.168.6.1:8300 [Follower] entering Follower state <Leader: "">
2017/07/20 13:31:42 [INFO] serf: EventMemberJoin: Sajal-HP.dc1 192.168.6.1
2017/07/20 13:31:42 [WARN] serf: Failed to re-join any previously known node
2017/07/20 13:31:42 [INFO] serf: EventMemberJoin: Sajal-HP 192.168.6.1
2017/07/20 13:31:42 [INFO] consul: Handled member-join event for server "Sajal-HP.dc1" in area "wan"
2017/07/20 13:31:42 [INFO] consul: Adding LAN server Sajal-HP <Addr: tcp/192.168.6.1:8300> <DC: dc1>
2017/07/20 13:31:42 [WARN] serf: Failed to re-join any previously known node
2017/07/20 13:31:42 [INFO] agent: Started DNS server 127.0.0.1:8600 <udp>
2017/07/20 13:31:42 [INFO] agent: Started DNS server 127.0.0.1:8600 <tcp>
2017/07/20 13:31:42 [INFO] agent: Started HTTP server on 127.0.0.1:8500
2017/07/20 13:31:49 [ERR] agent: failed to sync remote state: No cluster leader
2017/07/20 13:31:49 [ERR] http: Request GET /v1/catalog/services?wait=2s&index=169, error: No cluster leader from=127.0.0.1:53785
2017/07/20 13:31:50 [ERR] http: Request GET /v1/catalog/services, error: No cluster leader from=127.0.0.1:53789
2017/07/20 13:31:51 [ERR] http: Request GET /v1/catalog/services, error: No cluster leader from=127.0.0.1:53792
2017/07/20 13:31:52 [WARN] raft: Heartbeat timeout from "" reached, starting election
2017/07/20 13:31:52 [INFO] raft: Node at 192.168.6.1:8300 [Candidate] entering Candidate state in term 66
2017/07/20 13:31:52 [INFO] raft: Election won. Tally: 1
2017/07/20 13:31:52 [INFO] raft: Node at 192.168.6.1:8300 [Leader] entering Leader state
2017/07/20 13:31:52 [INFO] consul: cluster leadership acquired
2017/07/20 13:31:52 [INFO] consul: New leader elected: Sajal-HP
2017/07/20 13:31:52 [WARN] agent: Check 'service:student-service-9099' is now critical
2017/07/20 13:31:52 [WARN] agent: Check 'service:student-service-9097' is now critical
2017/07/20 13:31:54 [INFO] agent: Synced check 'service:student-service-9097'
2017/07/20 13:31:54 [INFO] agent: Synced check 'service:school-service-8098'
2017/07/20 13:31:54 [INFO] agent: Synced check 'service:student-service-9099'
2017/07/20 13:31:54 [INFO] agent: Synced check 'service:student-service-9098'
```

# Configuring Consul in Local workstation



- **Test whether Consul Server is running** – Consul runs on default port and once agent started successfully, browse <http://localhost:8500/ui> and you should see a console screen like –

A screenshot of a web browser window displaying the Consul UI. The address bar shows the URL <http://localhost:8500/ui/#/dc1/services>. The browser's toolbar includes icons for Apps, Bookmarks, Suggested Sites, and various links like myemail.accenture.co, Stocks, Imported From IE, Study, Gmail, YouTube, Stock/Share Market, The Economic Times, and social media links for Facebook and Google+. The main interface has a navigation bar with tabs: SERVICES (highlighted in pink), NODES, KEY/VALUE, ACL, DC1 (highlighted in green with a dropdown arrow), and a settings gear icon. Below the tabs are filters: 'Filter by name' (text input), 'any status' (dropdown menu), and 'EXPAND' (button). A search bar contains the text 'consul'. To the right, a summary box shows '1 passing' services. The overall theme is light-colored with a modern design.



## Consul Web

---

- Registered services
- The cluster nodes
- The Key/Values for the KeyValueStore
- Any ACL (Access Control List) values you have setup



# Consul

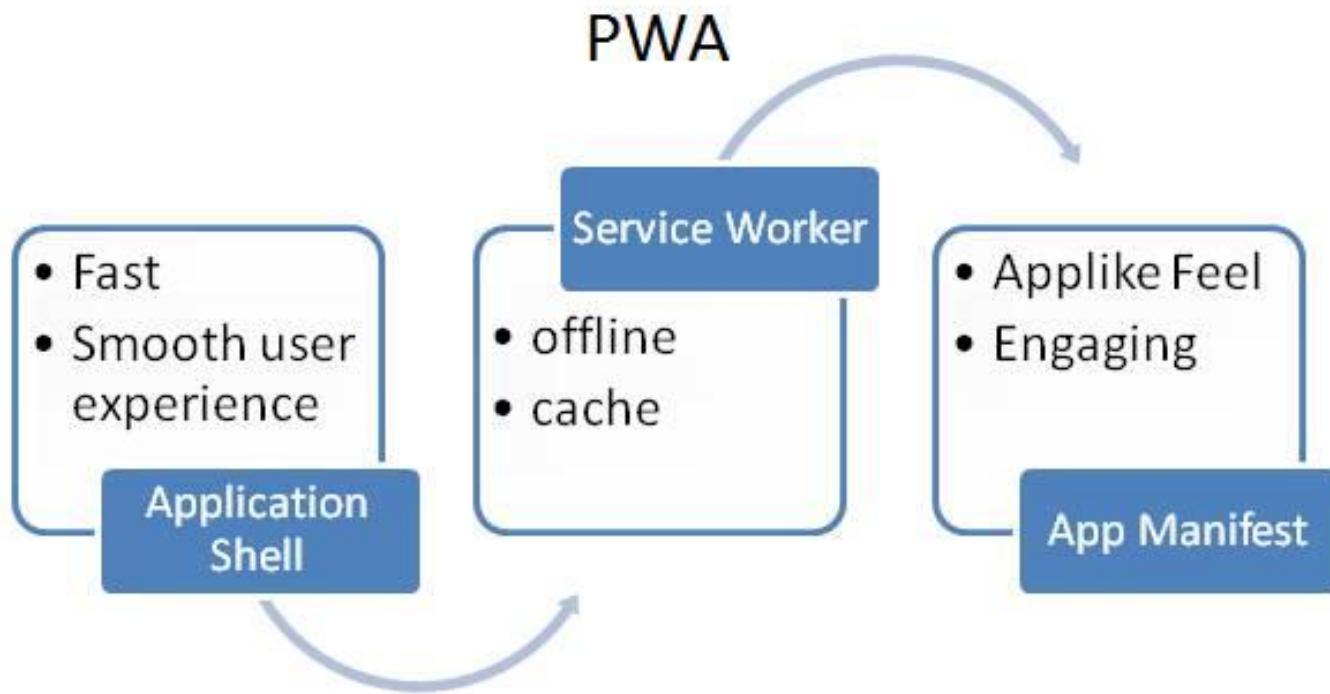
---

- Install-Package consul

- PWAs are just websites, enhanced using modern design and web technologies (Responsive Design, Touch Friendly, Service Workers, Fetch networking, Cache API, Push notifications, Web App Manifest) to provide a more native app-like experience.

- **Manifest, Service Worker, Secure:**
- To create a PWA the website will need: a manifest, service workers, and be on Https:
- Standard manifest file: The site requires a manifest file to define the features and behavior of the PWA. The file follows the W3C standard. This includes everything from images, to language, or the start page of your web app.
- Service workers: The Progressive Web App should have a mechanism (e.g. through a service worker) to help control traffic when the network isn't there or isn't reliable. The app should be able to work independent of network.
- Secure: A secure connection (HTTPS) over your site makes sure all traffic is as safe as a native app. A secure endpoint also allows the service worker to securely take action on the behalf of your app.
-

- **Manifest, Service Worker, Secure:**
- The manifest defines some details for your site if it's downloaded, which includes custom splash-screens, icon-standards and more.
- It also prompts the user to install it as a PWA, via an install-banner, if they visit the site frequently on a phone.
- This is one of the core properties of a PWA (Progressive Web App) since it gives the immersive App part.



Progressive Web Apps are applications that use modern web capabilities to provide a mobile application-like experience to users via the web.



# Progressive Web Apps are

---

- *Reliable*  
Service Workers enable PWA to load instantly, regardless of the network state. It never shows that awful dinosaur screen, even if there is no network.
- *Fast*  
Application Shell makes it faster and provides a smooth user experience.
- *Engaging*  
Manifest makes more engaging web apps. Manifest file makes PWA installable and live on the user's home screen. We can even make the users re-engaged with the help of push notifications.
- *Progressive*  
Works for every user, regardless of browser.
- *Safe*  
PWA Served via HTTPS to prevent snooping and ensure the security of content.
- *Linkable*  
It can be easily shared via URL.



# Proxy Pattern

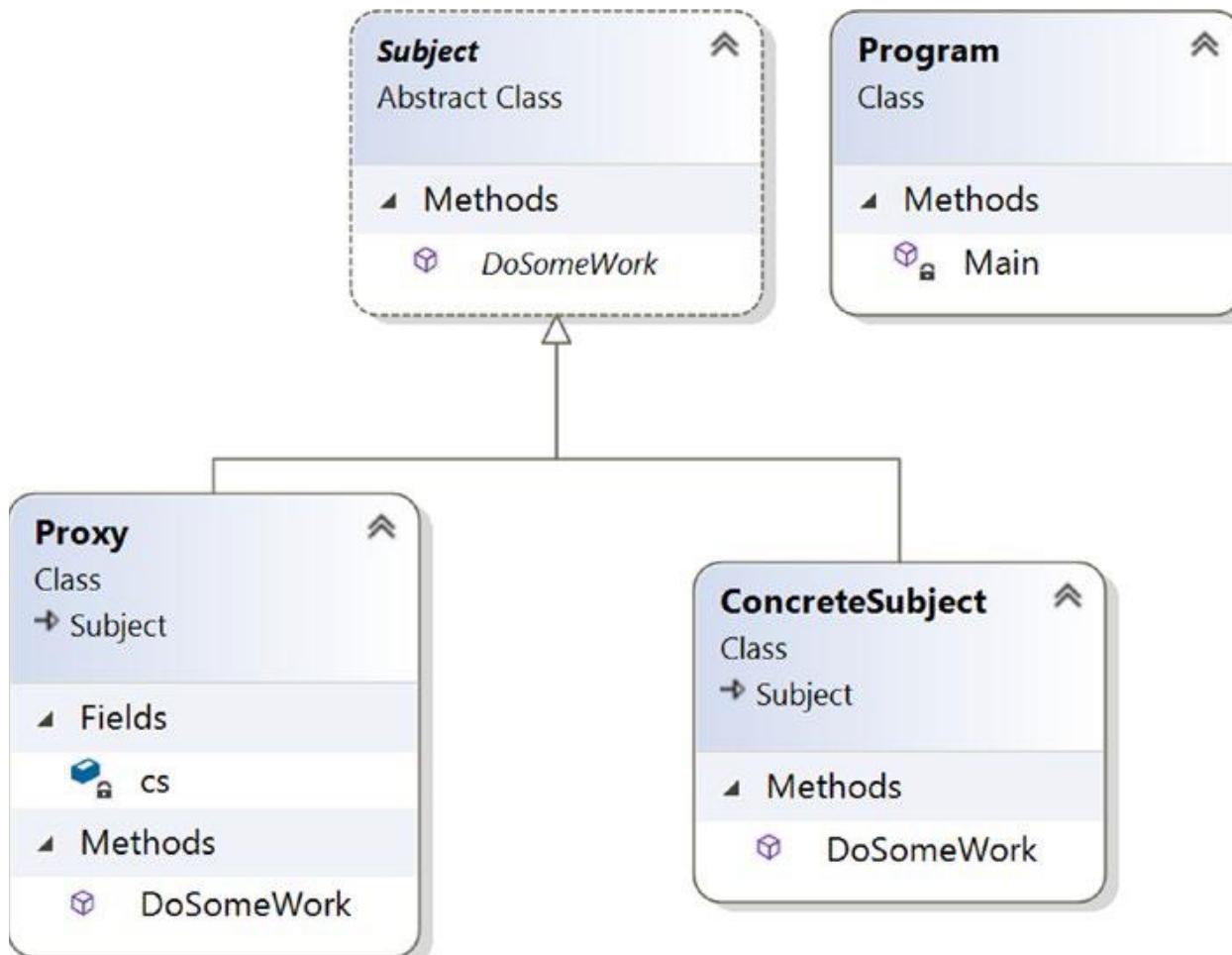
- Provide a surrogate or placeholder for another object to control access to it.
- Concept
- A proxy is basically a substitute for an intended object.
- When a client deals with a proxy object, it thinks that it is dealing with the actual object.
- You need to support this kind of design because dealing with an original object is not always possible.
- This is because of many factors such as security issues, for example.
- So, in this pattern, you may want to use a class that can perform as an interface to something else.



# Proxy Pattern

---

- An ATM implementation will hold proxy objects for bank information that exists on a remote server.
- In the real programming world, creating multiple instances of a complex object (a heavy object) is costly in general.
- So, whenever you can, you should create multiple proxy objects that can point to the original object.
- This mechanism can also help you to save the computer/system memory.





# Strategy (Policy) Pattern

---

- Define a family of algorithms, encapsulate each one, and make them interchangeable.
- Strategy lets the algorithm vary independently from clients that use it.
- Concept
- You can select the behavior of an algorithm dynamically at runtime.



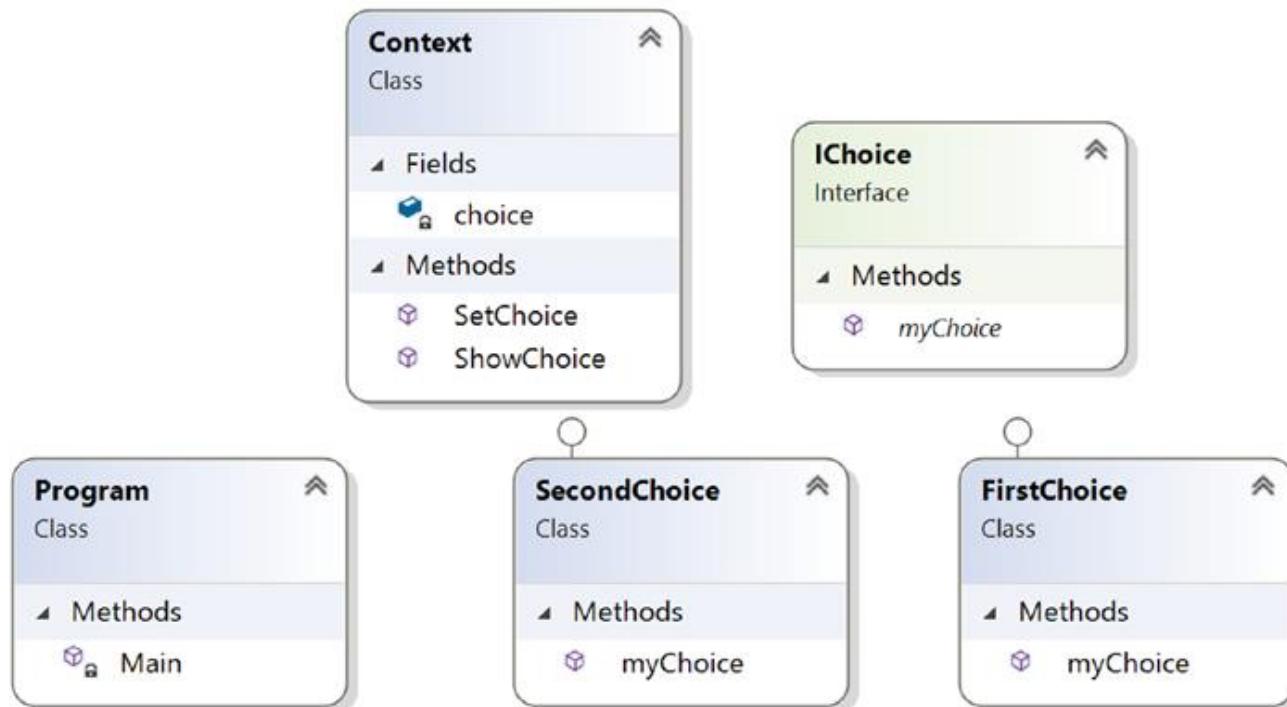
# Strategy (Policy) Pattern

---

- Suppose you have a backup memory slot.
- If your primary memory is full and you need to store more data, you can store it in the backup memory slot.
- If you do not have this backup memory slot and you try to store additional data into your primary memory (when it is full), then that data will be discarded, you will receive exceptions, or you will encounter some peculiar behavior (based on the architecture of the program).
- So, a runtime check is necessary before storing the data, and then you can proceed



# Moq Framework



# Switch to classic ui



A screenshot of a Microsoft Edge browser window showing the Okta Admin Console. The title bar shows the URL <https://dev-979282-admin.oktapreview.com/admin/apps/add-app>. The browser toolbar includes various pinned tabs and icons. A blue arrow points from the top right towards the 'Classic UI' dropdown menu in the top right corner of the Okta interface.

The Okta interface has a dark blue header with the 'okta' logo and navigation links: Dashboard, Directory, Applications, Security, Reports, Settings, My Applications, and Upgrade. The 'Applications' link is highlighted.

The main content area is titled 'Add Application'. It features a search bar and a grid of application cards. The grid includes:

- TELADOC (Okta Verified) - Add button
- &frankly (Okta Verified, ✓ SAML) - Add button
- 10000ft (Okta Verified) - Add button
- 101domains.com (Okta Verified) - Add button
- 123RF (Okta Verified) - Add button
- 15Five (Okta Verified, ✓ SAML, ✓ Provisioning) - Add button

On the left, there's a sidebar with sections for 'Can't find an app?' (Create New App button), 'Apps you created (0)', 'INTEGRATION PROPERTIES' (Any selected, Supports SAML, Supports Provisioning), and 'CATEGORIES' (API Management, All selected, 6071 results, Application Delivery Controllers).

The bottom of the screen shows the Windows taskbar with the Start button, a search bar, pinned app icons (Word, Excel, File Explorer, etc.), and system status indicators.



Freelancer-dev-979282 - Application Setting up a SAML application in | +

<https://dev-979282-admin.oktapreview.com/admin/apps/add-app>

Apps Insert title here Empire New Tab How to use Assertion Browser Automation node.js - How can I fi Freelancer-dev-81048 Courses New Tab

P. Bala - Freelancer-dev-979282 Help and Support Sign out

okta Dashboard Directory Applications Security Reports Settings My Applications [Upgrade](#)

Create a New Application Integration

← Back to Applications

Add App

Search for a platform

Platform: Web

Sign on method:

- Secure Web Authentication (SWA)  
Uses credentials to sign in. This integration works with most apps.
- SAML 2.0  
Uses the SAML protocol to log users into the app. This is a better option than SWA, if the app supports it.
- OpenID Connect  
Uses the OpenID Connect protocol to log users into an app you've built.

Create Cancel

INTEGRATION PROVIDERS

Any

Supports SAML

Supports Provisioning

CATEGORIES

API Management 3

All 6071

Application Delivery Controllers 2

123RF Okta Verified

15Five Okta Verified ✓ SAML ✓ Provisioning

Type here to search

23:44 13/12/2018



# Preview SAML

---

- <?xml version="1.0" encoding="UTF-8"?>
- <saml2:Assertion>
- xmlns:saml2="urn:oasis:names:tc:SAML:2.0:assertion" ID="id8452218445246736637109369" IssueInstant="2018-12-13T18:19:26.791Z" Version="2.0">
- <saml2:Issuer Format="urn:oasis:names:tc:SAML:2.0:nameid-format:entity"><http://www.okta.com/Issuer></saml2:Issuer>
- <saml2:Subject>
- <saml2:NameID Format="urn:oasis:names:tc:SAML:1.1:nameid-format:unspecified">userName</saml2:NameID>
- <saml2:SubjectConfirmation Method="urn:oasis:names:tc:SAML:2.0:cm:bearer">
- <saml2:SubjectConfirmationData NotOnOrAfter="2018-12-13T18:24:27.004Z" Recipient="http://example.com/saml/sso/example-okta-com"/>
- </saml2:SubjectConfirmation>
- </saml2:Subject>
- <saml2:Conditions NotBefore="2018-12-13T18:14:27.004Z" NotOnOrAfter="2018-12-13T18:24:27.004Z">
- <saml2:AudienceRestriction>
- <saml2:Audience><http://example.com/saml/sso/example-okta-com></saml2:Audience>
- </saml2:AudienceRestriction>
- </saml2:Conditions>
- <saml2:AuthnStatement AuthnInstant="2018-12-13T18:19:26.791Z">
- <saml2:AuthnContext>
- <saml2:AuthnContextClassRef>urn:oasis:names:tc:SAML:2.0:ac:classes:PasswordProtectedTransport</saml2:AuthnContextClassRef>
- </saml2:AuthnContext>
- </saml2:AuthnStatement>
- </saml2:Assertion>



# SAML Authentication Steps

---

[https://developer.okta.com/standards/SAML/setting\\_up\\_a\\_saml\\_application\\_in\\_okta](https://developer.okta.com/standards/SAML/setting_up_a_saml_application_in_okta)

Client Id 0oaibarqtmkLjQy0o0h7

Client Secret KM48VeTKtPW0W2\_Vh5iOy95Ej6aHMuDWvjtXBGVK



# Template method pattern

---

- GoF Definition
- Define the skeleton of an algorithm in an operation, deferring some steps to subclasses.
- The Template Method pattern lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure



# Template method pattern

---

- With the Template Method pattern, you define the minimum or essential structure of an algorithm.
- Then you defer some responsibility to the subclasses. The key idea is that you can redefine certain steps of an algorithm, but those changes will not impact the core architecture.



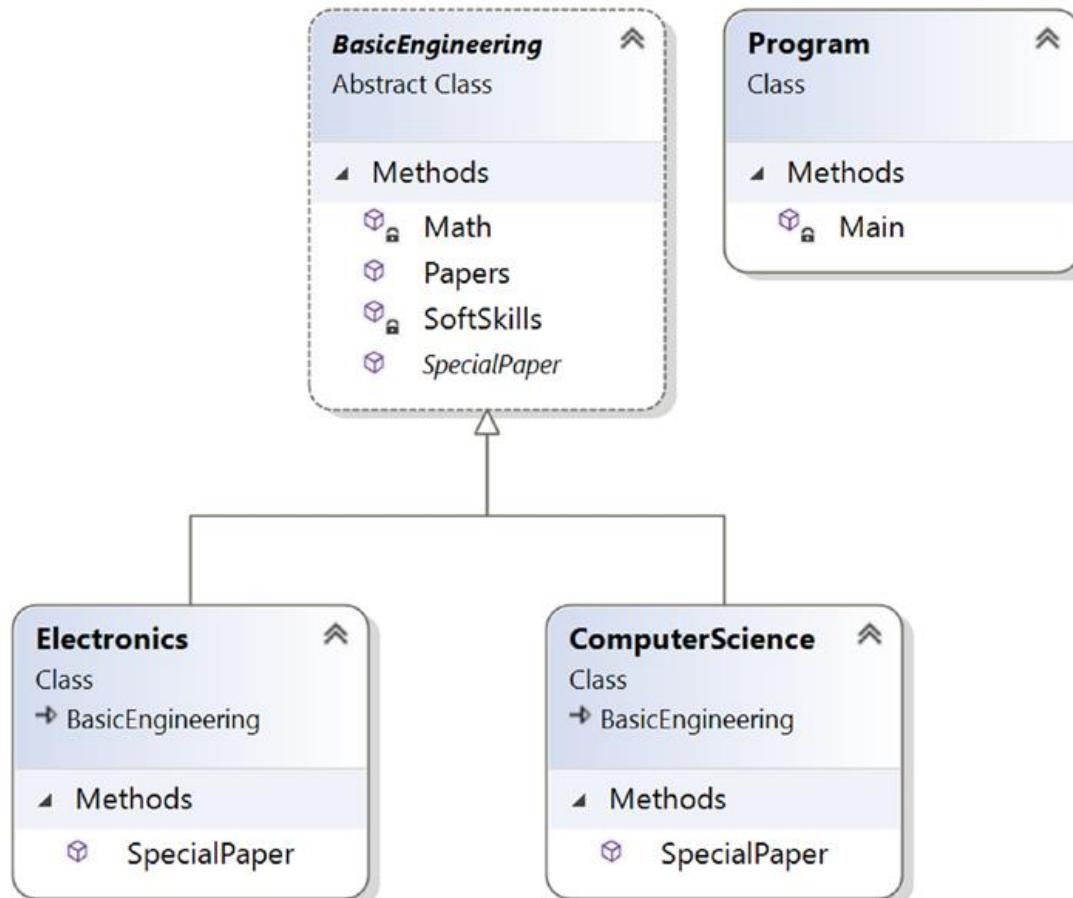
## Template method pattern

---

- The Template Method pattern makes sense when you want to avoid duplicate code in your application but allow subclasses to change some specific details of the base class workflow to bring varying behavior to the application.



# Template Pattern





# Decorator

---

- GoF Definition
- Attach additional responsibilities to an object dynamically.
- Decorators provide a flexible alternative to subclassing for extending functionality.



# Decorator

---

- This pattern promotes the concept that your class should be closed for modification but open for extension.
- In other words, you can add a functionality without disturbing the existing functionalities.
- The concept is useful when you want to add some special functionality to a specific object instead of the whole class.
- This pattern prefers object composition over inheritance.
- Once you master this technique, you can add new responsibilities to an object without affecting the underlying classes



## Decorator

---

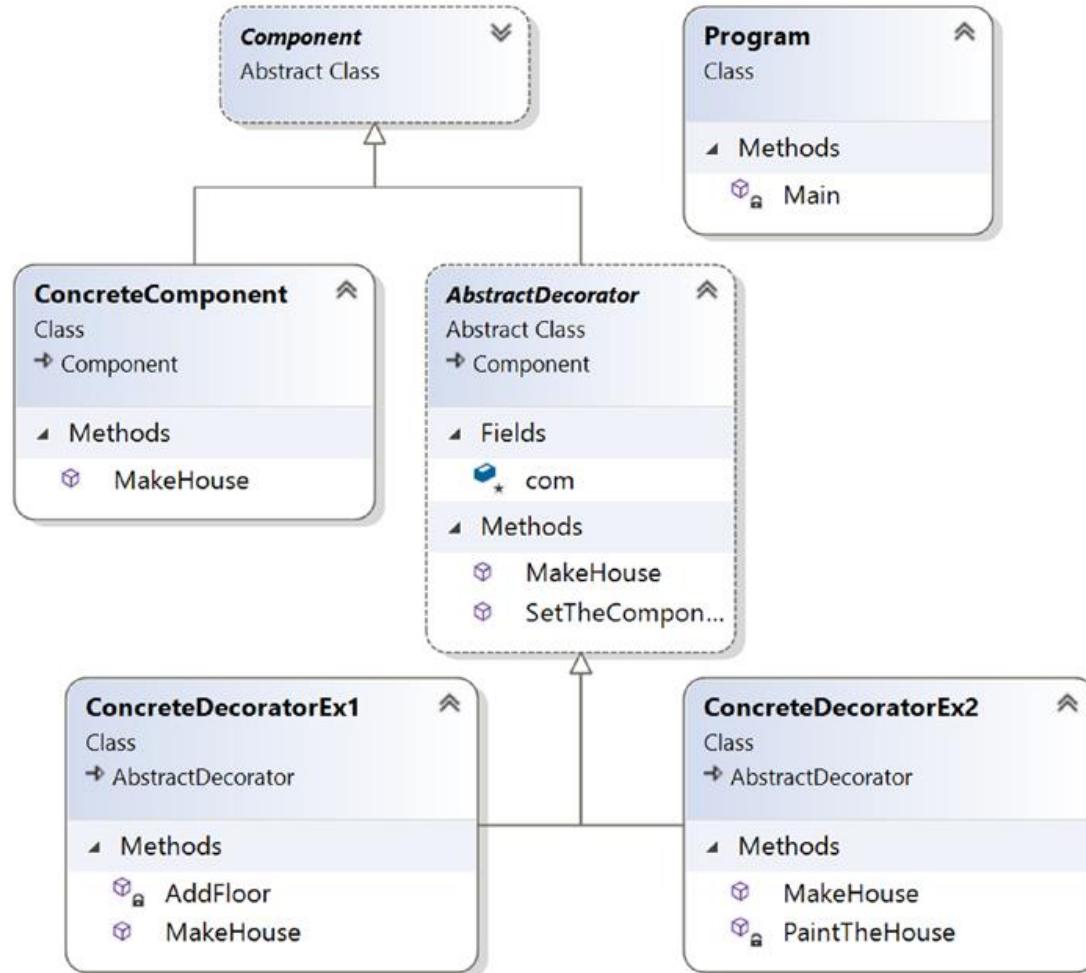
- Suppose you own a single-story house and you decide to build a second floor on top of it.
- Obviously, you may not want to change the architecture of the ground floor.
- But you may want to change the design of the architecture for the newly added floor without affecting the existing architecture.



# Decorator

- Suppose in a GUI-based toolkit you want to add some border properties.
- You could do this with inheritance, but that cannot be treated as an ultimate solution because you do not have absolute control over everything from the beginning.
- So, this technique is static in nature.
- Decorators offer a flexible approach. They promote the concept of dynamic choices.
- For example, you can surround the component in another object. The enclosing object is termed a *decorator*, and it must conform to the interface of the component that it decorates.
- It will forward the requests to the component, and it can perform additional operations before or after those requests. In fact, you can add an unlimited number of responsibilities with this concept.

# Decorator





# State Pattern

- Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.
- The concept is best described by the following examples.
- Consider the scenario of a network connection, say a TCP connection.
- An object can be in various states; for example, a connection might already be established, a connection might be closed, or the object has already started listening through the connection.
- When this connection receives a request from other objects, it responds as per its present state.

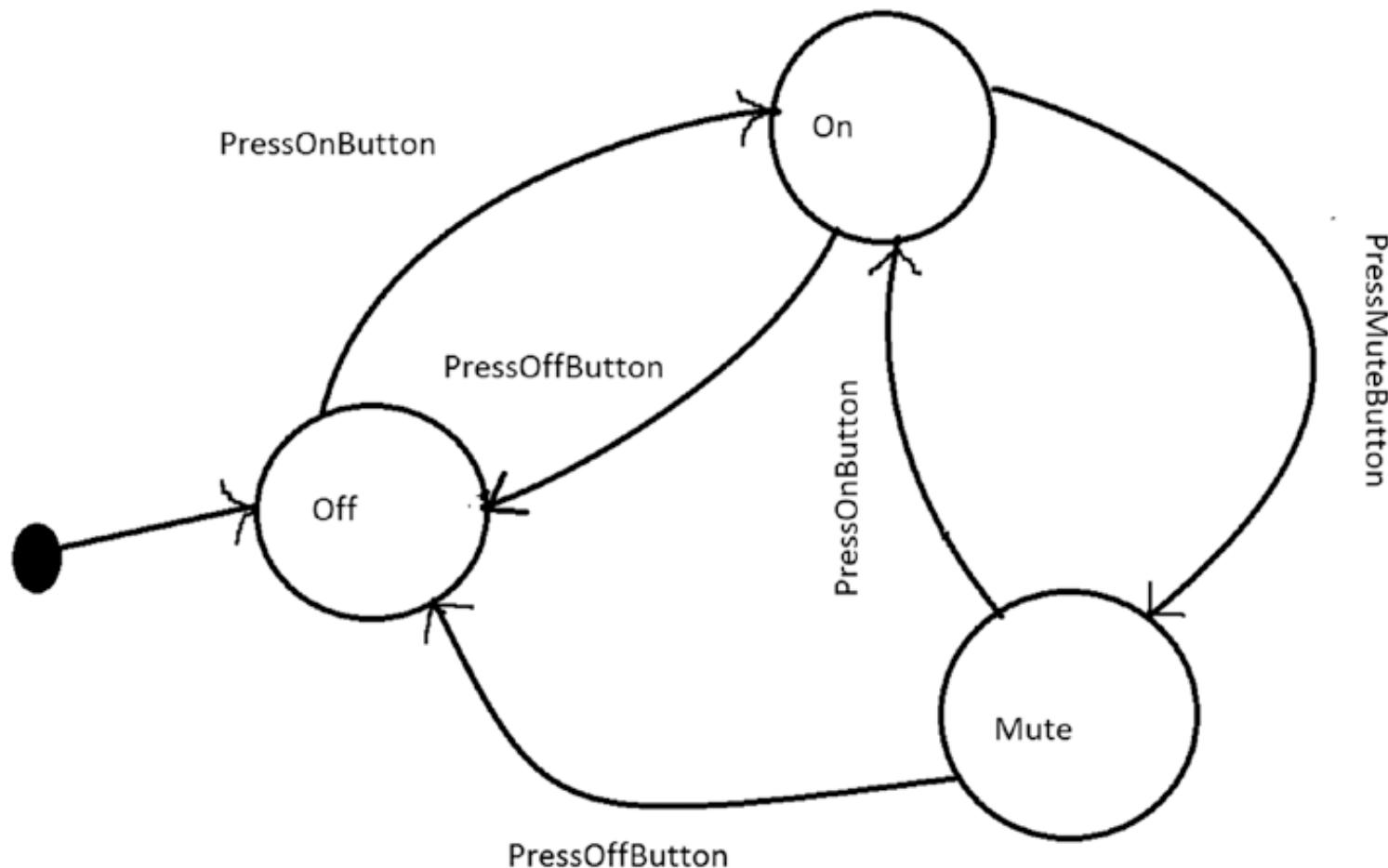


# State Pattern

---

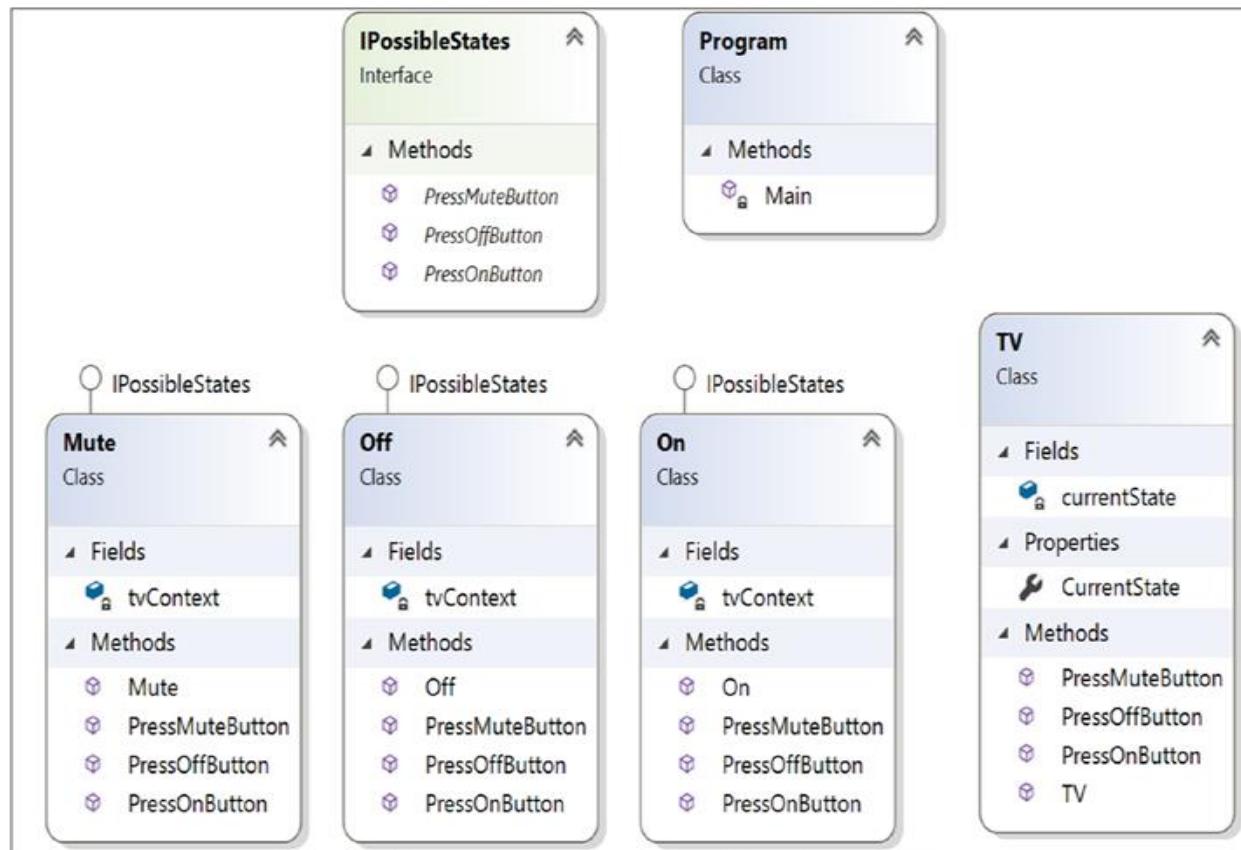
- Suppose you have a job-processing system that can process a certain number of jobs at a time.
- When a new job appears, either the system processes the job or it signals that the system is busy with the maximum number of jobs that it can process at a time.
- In other words, the system will send a busy signal if it gets another job-processing request when its total number of job-processing capabilities has been reached.

# State Pattern





# State Pattern





# Visitor Pattern

---

- GoF Definition
- 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.



# Visitor Pattern

---

- Concept
- With this pattern, you separate an algorithm from an object structure.
- So, you can add new operations without modifying the existing architecture.
- This pattern supports the open/close principle (which says extension is allowed but modification is disallowed for entities such as class, function, modules, and so on)..



## Visitor Pattern

---

- Think of a taxi-booking scenario.
- When the taxi arrives at your door and you enter the vehicle, the visiting taxi takes control of the transportation.
- It can take you to your destination through a new route that you are not familiar with, and in the worst case, it can alter the destination.



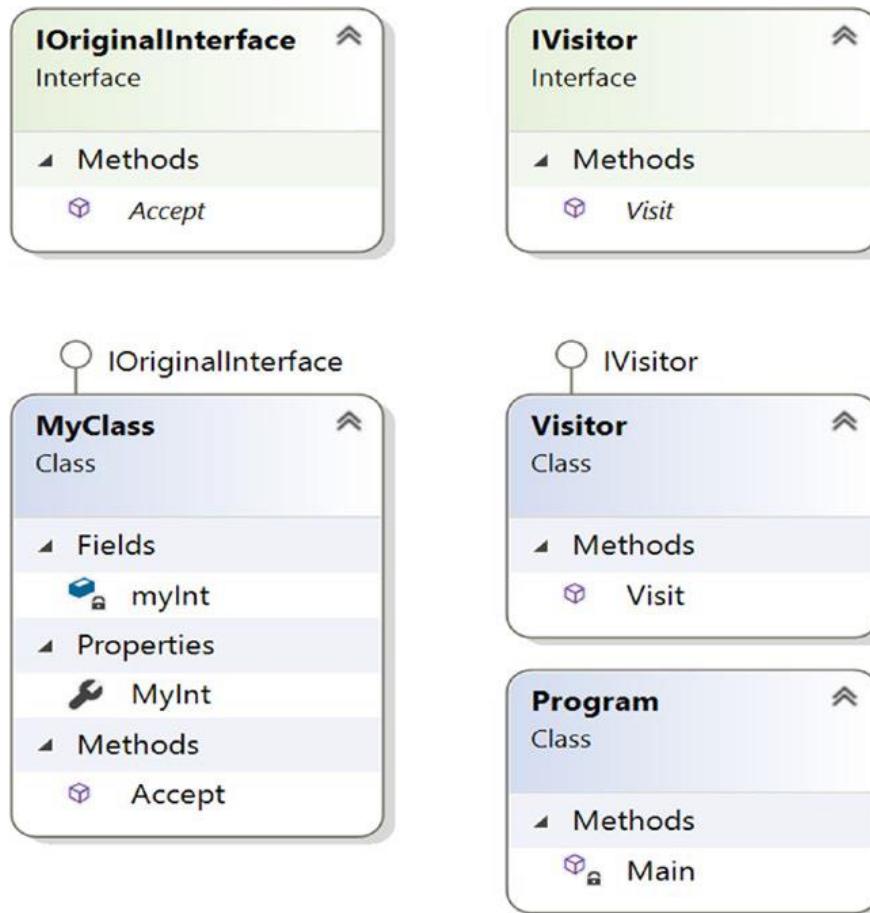
# Visitor Pattern

---

- This pattern is useful when public APIs need to support *plug-in* operations.
- Clients can then perform their intended operations on a class (with the visiting class) without modifying the source.



# Visitor Pattern



# Questions



# Module Summary

---

- Microservices using docker and Kubernets.
- Message, Channel and Adapter
- Understood the different Component Integration
- Understood the Event-Driven Architecture

