

Design Patterns in Object Oriented Programming

”
“ *"Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice."* - Christopher Alexander

1.1. Design Patterns: Introduction



Design Patterns are the blueprints to solve commonly occurring problems in software design.

However, design patterns are not to be confused with some reusable piece of code. These are generalized descriptions of Objects and Classes to be customized to solve a design problem for a specific domain. Each pattern focuses on a specific design pattern and provides a solution which in general lays down guidelines about how to structure Objects and Classes, make up the design, their relationships, responsibilities, and collaborations. Patterns are programming language agnostic and concept can be applied across different languages while developing enterprise applications.

1.2. Design Patterns: Mandate or Necessity?

Arguments can be drawn that usage of design patterns is not mandatory while programming and some might even showcase examples of how they have managed to develop and deploy an application without incorporating any patterns.

Argument well taken, however, the only constant to a successful software application is the change in its behavior as per the changing requirements over time to maintain an edge over its competitors. So, we are not just talking about building something and forget it. We are talking about a product which is built to last through turbulence of changing and testing times and requirements. And this is where Design Patterns come handy as necessary tools to maintain the software with efficient design, and if this toolkit is utilized modestly right through the initial design and development phase, then you can do away with some of the post release headaches of refactoring or restructuring the entire codebase.

It is true that most of us don't see the problems incoming at the time of coding, but by utilizing design patterns from the initial design and development phase, we can proactively address potential challenges and ensure the software is built with flexibility, adaptability, and maintainability in mind.

As requirements change and new features need to be added, design patterns allow for easier modification and extension of the existing code without introducing unnecessary complexity or risking the stability of the system. They enable developers to make incremental changes to the codebase, rather than resorting to massive refactoring or restructuring efforts. This saves time, reduces risks, and improves the efficiency of software maintenance.

Let us take a simple use case to understand this situation:

We are developing a payment processing system, for an e-commerce website, that handles various payment methods, such as credit cards, debit cards, and mobile wallets. The system needs to support different payment gateways, each with its own specific implementation and integration requirements. Payment system would be utilized while placing an order on the website.

How would a novice developer attempt to design a solution for this problem?



Let us look at a simple code which can get the job done for us.

```
/**
 * Payment Processor class for processing different payment types
 */
public class PaymentProcessor {
    /**
     * This method processes a payment based on the payment method and amount provided.
     * It uses conditional statements to determine the logic for processing the payment based
     * on the payment method. If the payment method is not one of the three specified options
     * (credit card, debit card, or wallet),
     * additional code can be added to handle those other payment methods.
     * @param paymentMethod
     * @param amount
     */
    public void processPayment(String paymentMethod, double amount) {
        if (paymentMethod.equals("CreditCard")) {
            // Logic for processing credit card payment
        } else if (paymentMethod.equals("DebitCard")) {
            // Logic for processing debit card payment
        } else if (paymentMethod.equals("Wallet")) {
            // Logic for processing wallet payment
        }
        // Additional code for handling other payment methods
    }
}
```

Well, it was pretty simple right? If we have to add additional payment methods, then we will keep extending the if else logic. Do you see anything wrong with this approach?

Let us say, some of the payment methods have additional calculation logic based on some parameters, we will have to keep adding that block of code in specific if-else block of that payment method. With time, the code will get cumbersome and lengthy. May be two years down the line, a new developer joins the team and is handed over the responsibility of the code and told to modify a certain feature.

Imagine the horror on developer's face! Everything in a single place with no segregation at all. If one of the payment methods encounter an error, entire method will throw an exception. Use of conditional statements make the code less readable and harder to maintain. Code duplication is another problem which might introduce unnecessary error.

How to solve this design problem?

In the problem statement described above, we have payment strategy feature which keeps changing with time with respect to Payment Processor class. Main Job of Payment Processor class should be to identify which payment method is to be used, however, it is currently also taking the responsibility of implementation of various payment methods. So, it is doing a lot of heavy lifting. So, the first logical step would be to think of a solution, which would lead the way for segregation of concern in the code structure.

So, you already thinking towards one of the first principles of Object-Oriented Programming, i.e. **Single Responsibility Principle**. We shall be learning about it in the next few pages. Right now, let us focus back to our current example.

We assign the Payment Processor a single responsibility of identifying the payment method for processing payment and extract the part of payment method implementation.



Let us look at the following code to understand how it can be done effectively.

```

/*
 * This is an interface called IPaymentStrategy which declares a method called processPayment
 * that takes in a double value representing the payment amount.
 * It is meant to be implemented by classes that provide different payment strategies.
 */
public interface IPaymentStrategy {
    void processPayment(double amount);
}

// Concrete implementations of payment strategies

public class CreditCardPaymentStrategy implements IPaymentStrategy {
    @Override
    public void processPayment(double amount) {
        // Logic for processing credit card payment
    }
}

public class DebitCardPaymentStrategy implements IPaymentStrategy {
    @Override
    public void processPayment(double amount) {
        // Logic for processing debit card payment
    }
}

public class WalletPaymentStrategy implements IPaymentStrategy {
    @Override
    public void processPayment(double amount) {
        // Logic for processing wallet payment
    }
}

```



The modified *PaymentProcessor* class would look something like this:

```

/*
 * This is a PaymentProcessor class that has a private instance variable of type
 * IPaymentStrategy.
 */
public class PaymentProcessor {
    private IPaymentStrategy paymentStrategy;

    /**
     * This method sets the payment strategy for the current object.
     * It takes an instance of an IPaymentStrategy as a parameter and
     * assigns it to the paymentStrategy field.
     * This allows the object to use the appropriate payment method when needed.
     * @param paymentStrategy
     */
    public void setPaymentStrategy(IPaymentStrategy paymentStrategy) {
        this.paymentStrategy = paymentStrategy;
    }

    /**
     * This method processes a payment using a selected payment strategy.
     * @param amount
     */
    public void processPayment(double amount) {
        // Delegate the payment processing to the selected strategy
        paymentStrategy.processPayment(amount);
    }
}

```



Now, let us see the use of this code to understand its effectivity.

```

public class Main {
    public static void main(String[] args) {
        // Create an instance of PaymentProcessor
        PaymentProcessor paymentProcessor = new PaymentProcessor();

        // Set the payment strategy based on the selected payment method
        paymentProcessor.setPaymentStrategy(new CreditCardPaymentStrategy());

        // Process the payment
        double amount = 100.0;
        paymentProcessor.processPayment(amount);
    }
}

```



We can observe that now we are able to set the payment strategy at run time. Also, the code has become more extensible and maintainable. If we have to add a new payment method, we do not need to touch the code in Payment Processor section. The separation of concerns is achieved by encapsulating the payment processing logic within each strategy implementation. The *PaymentProcessor* class is modified to use the selected payment strategy and delegate the payment processing to that strategy.

Now, one can argue why did we use Interface of Payment Strategy when we could have leveraged Inheritance by creating a Super Class of Strategy and specific sub-classes extending its functionality.

1.2.1. Limitation of Inheritance

Answer to this question lies in the implementation of Inheritance itself. Following scenarios need to be considered in case of inheritance:

1. We must implement all abstract methods of the parent class even if there is no use of them in child class.
2. We must always make sure that the behavior implemented by the new child class is always in adherence with the functionality of base class or it would result in code breakage. Reason being objects of the subclass may be passed to any code that expects objects of the superclass.
3. There is tight coupling between parent class and child class.
4. When inheritance is used excessively to achieve code reuse, it can lead to the creation of parallel inheritance hierarchies or the explosion of class combinations. This situation occurs when there are multiple dimensions or orthogonal concerns in the system that need to be combined.

Let us elaborate more on the last point.

Suppose we have a base class called "Vehicle" that represents various types of vehicles. Now, let's say we want to introduce another dimension, such as "Color," to represent different colors of vehicles. Here, we have two dimensions: 1. Type of Vehicle 2. Color of Vehicle.

If we approach this problem via inheritance, we will end up create sub classes of multiple combinations.

In case, we want to add more dimensions like size, fuel type, etc. class hierarchy will bloat even further. This approach violates the **Open-Closed Principle (OCP)**, which states that software entities (classes, modules, functions) should be open for extension but closed for modification.

Following [picture](#) depicts the class structure:

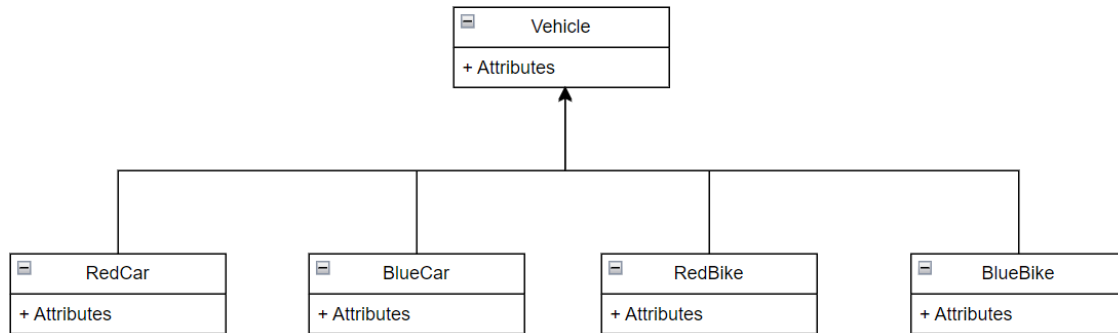


Figure 3-0-1

Hence, to overcome such scenarios, an alternative concept of **Composition** is introduced.

1.2.2. Composition



Composition in object-oriented programming refers to the practice of creating complex objects by combining or composing simpler objects.

It is a design principle that emphasizes building objects by assembling or "composing" them from other objects, rather than inheriting their behavior. In other words, while Inheritance represents IS-A relationship between classes, Composition represents HAS-A relationship between classes.

In our example above, we have used a HAS-A relationship between *PaymentProcessor* and *IPaymentStrategy* instance.

1.3. SOLID Principles

The SOLID Principles are five basic principles of Object-Oriented architecture, first introduced by Robert Martin. They are a set of rules and best practices to follow while designing a class structure.

SOLID is a mnemonic for following five design principles:

1. **S**ingle Responsibility Principle
2. **O**pen-Closed Principle
3. **L**iskov Substitution Principle
4. **I**nterface Segregation Principle
5. **D**ependency Inversion Principle

We already had an introduction with first two principles while discussing limitations of inheritance, nevertheless, we shall be categorically introduced in the following sections.

1.3.1. The Single Responsibility Principle



A class should have only one reason to change

In the context of this principle, responsibility is the reason for change and "If we can think of more than one motive for changing a class, then that class has more than one responsibility."

1.3.1.1. Example

Let us consider the following scenario in the following picture.

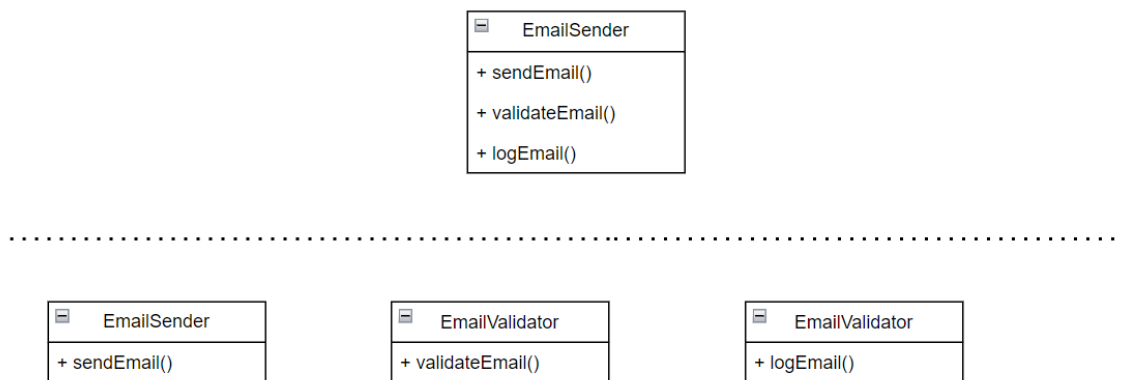


Figure 3-0-1

The *EmailSender* class is responsible for sending emails, validating email addresses, and logging sent emails. However, this violates the SRP because the class has multiple responsibilities. Hence, we have separated the responsibilities into three distinct classes:

1. *EmailSender* is now solely responsible for sending emails.
2. *EmailValidator* is responsible for validating email addresses.
3. *EmailLogger* is responsible for logging sent emails.

By adhering to the SRP, each class has a single responsibility, and if any changes or modifications are needed, they can be made independently without affecting other classes.

1.3.2. Open Close Principle



Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification.

In the vehicle [example](#) in paragraphs above, we discussed how adding new behaviour to classes leads to modification of existing behaviours. The main idea of this principle is to keep existing code from breaking when new behaviours are introduced in the system. A system where a single change triggers a cascade of changes in dependent modules, is called a rigid system. We want our software entities to be loosely coupled while designing the architecture.

When we say, “Open for extension”, it means we can extend a class, produce a sub-class, and do whatever you want with it—add new methods or fields, override base behaviour, etc.

And when we say, “Closed for Modification”, it means whenever we want to extend the behaviour of the class, we shall never be changing its source code.



Confused?? How can we change the behaviour of an entity without ever changing its source code?




Abstraction is the answer to this problem.

Abstract base classes provide a level of abstraction that defines the common interface, behaviour, and attributes shared by a group of related classes. They can serve as a blueprint or contract for derived classes to follow, while allowing each derived class to provide its own implementation details.

By defining abstract base classes, we can create a fixed set of methods and properties that all derived classes must implement. However, the actual behaviour and functionality can vary among the derived classes, effectively representing an unbounded group of possible behaviours.


Modifying existing code, especially in production environments, should be approached with caution and careful consideration. By creating a subclass and overriding specific parts of the original class, we can achieve the desired behaviour without directly modifying the original class. This approach provides a way to extend the functionality of a class without impacting existing clients or breaking their code.

 Getting back to the vehicle [example](#), let us write the code in accordance with OCP principle and understand the outcomes.

```
public class Vehicle {
    // Common vehicle attributes and behavior
}
public class Car implements Vehicle {
    // Car-specific behavior and properties
}
public class Bike implements Vehicle {
    // Bike-specific behavior and properties
}
public class Color {
    // Color-related behavior and properties
}
public class Red implements Color {
    // Implementation for red color
}
public class Blue implements Color {
    // Implementation for blue color
}
public class ColoredVehicle {
    private Vehicle vehicle;
    private Color color;
    public ColoredVehicle(Vehicle vehicle, Color color) {
        this.vehicle = vehicle;
        this.color = color;
    }
    public void setColor(Color color) {
        this.color = color;
    }
    // Additional behavior and methods for the colored vehicle
}
Vehicle car = new Car();
Vehicle bike = new Bike();

Color red = new Red();
Color blue = new Blue();

ColoredVehicle redCar = new ColoredVehicle(car, red);
redCar.setColor(blue); // Changing color of the vehicle dynamically
ColoredVehicle blueBike = new ColoredVehicle(bike, blue);
```

 With this approach, you can dynamically combine different vehicle types with different colours without the need for a massive hierarchy of subclasses. By separating the concerns of vehicle type and color into distinct classes and using composition, we achieve greater flexibility and maintainability. We can easily introduce new colors, vehicle types, or other dimensions without modifying existing code. The combination of dimensions is handled dynamically at runtime, allowing for a more scalable and adaptable solution.

This approach adheres to the principles of composition over inheritance.

1.3.3. Liskov Substitution Principle



Subtypes must be substitutable for their base types.

This principle states that the subclass should remain compatible with the behavior of the superclass. In other words, objects of a superclass should be replaceable with objects of its subclasses without affecting the correctness of the program.

If a class B is a subtype of class A, then instances of class B should be able to be used wherever instances of class A are expected, without causing any unexpected behavior or violating the contract of the superclass.

In layman terms, a class's contract tells its clients what to expect. If a subclass extends or overrides the behavior of the superclass in unintended ways, it will break the contract. There can be various ways via which this contract can be violated.

Let us look at the checkpoints, if not adhered, might result in contract violation by sub-class.

1.3.3.1. Sub-Class Checkpoints

- The parameter types in a subclass method should either match exactly or be more general (abstract) than the parameter types in the corresponding superclass method.

Let us understand this via a use case.

*Suppose we have a super **class A** with a method **drive** to drive the cars: **drive(Car c)**. Now we create a sub **class B** and it overrides the existing **drive** functionality to drive all vehicles: **drive(Vehicle v)**. If we pass the object of newly created subclass to the existing clients of the super class, things will work as expected. Reason: Earlier, it was expected to drive the cars however even with the new modification, it can drive all the vehicles which means it can still drive the car.*

*Say, we create another sub **class C** and override **drive** method to drive only Lamborghini: **drive(Lamborghini l)**. Now, we have a problem if we pass the object of sub **class C** to the clients of superclass, it will break the contract as it is expected to drive all the cars not just Lamborghini.*

- Returning an object that's compatible with the object returned by the superclass method.

*We can think of an inverse of the example above. If method **getName(): Car** of Super **class A** is returning an object of type **car** and the sub **class B** overrides this method which returns object of **Lamborghini**, it won't cause any contract violation. However, if sub **class C** overrides this method and returns an object of **Vehicle** then it will break the client contract, as it gets back a generic vehicle which it is not programmed to entertain.*

- A method in a child class shouldn't throw types of exceptions which the parent class method isn't expected to throw.

This point is in line with the first two points. Modern day programming compilers also take care of this code smell.

- A child class should not change the semantics or introduce side effects that are not part of the superclass's contract.

This point reflects about not introducing a side effect. A side effect could be anything, even a print or log statement in sub class method which is not present in parent class.

- A sub class method should not change the pre-conditions or post conditions or else the contract expected by the client of super class will be violated.

For example, if a method of super class expects Numerical values in the argument but a sub class method applies a check to only accept positive values, it would lead to contract violation.

1.3.3.2. Importance of Liskov Substitution Principle

When client code cannot freely substitute a superclass reference with a subclass object, it often leads to the introduction of conditional code that checks the type of the object and performs special handling for specific subclasses. This conditional code tends to be scattered across the codebase, resulting in code that is difficult to maintain.

1.3.4. Interface Segregation Principle



Clients should not be forced to depend upon interfaces that they don't use.

Interface is a set of abstractions which the class implementing the interface must follow. The Interface Segregation Principles states that the client should not be exposed to the methods it is not using.

Say, we have a big fat interface with 5 methods abstracted, however, the client, which is implementing this interface, makes usage of only 3 methods. Thus, the program violated Interface Segregation principle. This principle advises that instead of one fat interface many small interfaces should be preferred based on groups of methods, each one serving one submodule.

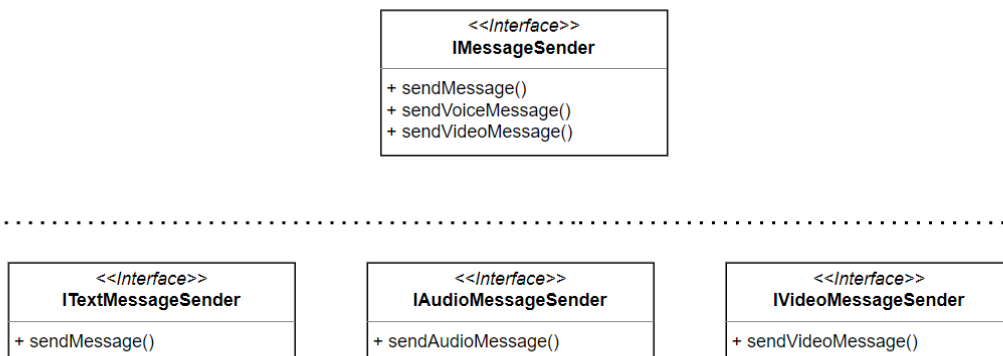


Figure 3-0-2

In the above [diagram](#), we can observe that the fat interface Message Sender has been broken down to three interfaces, each specific to their functionality. So, if a client needs only to implement Audio and Text message sending functionality, it does not longer need to worry about sending video message code, which was earlier abstracted.

1.3.5. The Dependency-Inversion Principle



High-level modules should not depend on low-level modules. Both should depend on abstractions. Abstractions should not depend on details. Details should depend on abstractions.

Let us first distinguish between High-Level Modules and Low-Level Modules.

High-level modules: They contain the important policy decisions and business models of an application. They contain complex business logics which directs low level modules to do tasks.

Low-Level modules: They deal with basic operations or tasks. We can also say that low level modules are created to manage unit behaviours separately, such as data transfer, database connection, etc.

The principle suggests that both high-level and low-level modules should depend on abstractions rather than concrete implementations. Abstractions provide a generalized interface or contract that can be implemented by various concrete classes. By depending on abstractions, modules become decoupled from specific implementations, making them more flexible and easier to modify or replace.

The principle also emphasizes that abstractions should not depend on implementation details. This means that the definition and behavior of abstractions should remain independent of specific implementation choices. By keeping abstractions free from implementation details, we ensure that they can be reused and the changes in implementation details do not affect the overall system design.

Instead it suggests that the implementation details should be depending on abstractions. This implies that low-level modules, which handle specific implementation details, should be designed in a way that they can be easily substituted or modified to conform to the abstractions defined by higher-level modules.

Let us consider an example of a messaging system that sends notifications through different channels such as email, SMS, and push notifications.

In a traditional approach, the high-level module responsible for sending notifications might directly depend on the low-level modules that handle the specific implementations of email, SMS, and push notification sending.

```
package DIP;

public class MessageSender {
    private EmailSender emailSender;
    private SMSSender smsSender;
    private PushNotificationSender pushNotificationSender;

    public MessageSender() {
        this.emailSender = new EmailSender();
        this.smsSender = new SMSSender();
        this.pushNotificationSender = new PushNotificationSender();
    }

    /**
     * Sends notification via various channels
     * @param recipient
     * @param message
     */
    public void sendNotification(String recipient, String message) {
        // Sending email notification
        if (isValidEmail(recipient)) {
            emailSender.sendNotification(recipient, message);
        }

        // Sending SMS notification
        if (isValidPhoneNumber(recipient)) {
            smsSender.sendNotification(recipient, message);
        }

        // Sending push notification
        pushNotificationSender.sendPushNotification(recipient, message);
    }
}

package DIP;
```

```

public class EmailSender {
    public void sendEmail(String recipient, String message) {
        // Implementation for sending an email notification
    }
}

package DIP;

public class MessageSender {
    private EmailSender emailSender;
    private SMSSender smsSender;
    private PushNotificationSender pushNotificationSender;

    public MessageSender() {
        this.emailSender = new EmailSender();
        this.smsSender = new SMSSender();
        this.pushNotificationSender = new PushNotificationSender();
    }

    /**
     * Sends notification via various channels
     * @param recipient
     * @param message
     */
    public void sendNotification(String recipient, String message) {
        // Sending email notification
        emailSender.sendEmail(recipient, message);

        // Sending SMS notification
        smsSender.sendSMS(recipient, message);

        // Sending push notification
        pushNotificationSender.sendPushNotification(recipient, message);
    }
}

package DIP;
public class PushNotificationSender {
    public void sendPushNotification(String recipient, String message) {
        // Implementation for sending a push notification
    }
}

```



We can observe the tight coupling in the send method of high-level module: *MessageSender*. We are performing checks before sending to appropriate channels. If in future, new channels are introduced, we will have to modify existing code, which will violate Open Close Principle.

If any changes are required in the messaging system, such as adding a new notification channel or modifying the behavior of existing channels, the high-level module: *MessageSender* would need to be modified. This violates the principle of separation of concerns and makes the code less maintainable and extensible.



Now, let us modify our code in adherence with DIP principle.

```

package DIP;

public interface INotifier {
    void sendNotification(String recipient, String message);
}

package DIP;

public class MessageSender {
    private INotifier notifier;
}

```

```

public MessageSender(INotifier notifier) {
    this.notifier = notifier;
}

/**
 * Sends notification via various channels
 * @param recipient
 * @param message
 */
public void sendNotification(String recipient, String message) {
    // Send notification
    notifier.sendNotification(recipient, message);
}
}

package DIP;

public class EmailSender implements INotifier {
    @Override
    public void sendNotification(String recipient, String message) {
        // Implementation for sending an email notification
    }
}

package DIP;

public class SMSSender implements INotifier{

    @Override
    public void sendNotification(String recipient, String message) {
        // Implementation for sending an SMS notification
    }
}

package DIP;

public class PushNotificationSender implements INotifier{

    @Override
    public void sendNotification(String recipient, String message) {
        // Implementation for sending a push notification
    }
}

```



Here, we have introduced an abstraction which represents a notifier. Message Sender will have a HAS-A relationship with this notifier. Also, the implementation of low-level modules for sending Emails, SMS or Push notification depend on the abstraction we created. This allows for flexibility and extensibility in the system. Say, if a new notification channel like a chatbot is added, a new class implementing the *Notifier* interface can be easily introduced without modifying the high-level module. There is no tight coupling between high-level modules and low-level modules.

1.4. Cataloguing Design Patterns

In the book “*Design patterns: Elements of reusable Object-Oriented Software*”, authors (historically named as *Gang of Four*) classified design patterns based on two criteria: **Purpose & Scope**.

Purpose reflects on what a pattern does. There are three purposes identified with respect to Objects.

1. **Creational:** deals with process of object creation.
2. **Structural:** deals with composition of classes or objects.
3. **Behavioral:** deals with interaction and responsibility distribution of classes or objects.

Scope defines whether patterns apply to classes or objects.



Design Patterns being followed in software development, are hugely inspired from how humans manage organizations effectively in the real world. These are not new inventions, but a blueprint prepared from learning the segregation behavior effectively employed by us in our daily scenarios for smooth functioning of an organisation or any system, while ensuring the integrity and security constraints.

As we learn about different design patterns in coming chapter, we can start creating a mental model and relate how we have been using or have experienced the usage of some of these patterns in our life already.



Abstraction, Encapsulation, Composition, or Inheritance are concepts, which are not bound to just software world but have been in practice since humans started designing effective organisations like court rooms, military bases, offices, schools, banks, etc. Likewise, the design patterns, rooted in above mentioned concepts, have emerged from decades of software development experience, and they provide proven solutions to common design problems. By applying these patterns, developers can create more maintainable, scalable, and secure software systems. The overlap with organizational behavior is a testament to how effective human problem-solving strategies can inspire robust software development practices.

Let us now discuss the broad categorisation of Design Patterns and then we shall start studying them in detail.

1.4.1. Creational Patterns

Class Patterns: In class-based creational patterns, the responsibility of creating objects is delegated to subclasses. The base class defines the interface or abstract methods for creating objects, while the subclasses provide the specific implementation. Examples of class-based creational patterns include the **Factory Method** pattern.

Object Patterns: In object-based creational patterns, the responsibility of object creation is delegated to another object. Instead of using inheritance, these patterns use composition, where one object holds a reference to another object responsible for creating the desired objects. Examples of object-based creational patterns include the **Abstract Factory**, **Prototype**, **Singleton** and **Builder** patterns.

1.4.2. Structural Patterns

Class Patterns: In class-based structural patterns, inheritance is used to compose classes. The patterns focus on class relationships and how they can be structured to form larger, more complex structures. Examples of class-based structural patterns include the **Adapter** pattern.

Object Patterns: In object-based structural patterns, objects are assembled or composed to create more complex structures. The focus is on object composition rather than inheritance. These patterns describe how objects can be connected and work together to achieve specific functionalities. Examples of object-based structural patterns include the **Bridge**, **Composite**, **Façade**, **Decorator** and **Proxy** patterns.

1.4.3. Behavioral Patterns

Class Patterns: In class-based behavioral patterns, inheritance is used to describe algorithms and flow of control. The patterns focus on defining common interfaces or base classes that describe the behavior and allow subclasses to override or implement specific algorithms. Examples of class-based behavioral patterns include the **Template Method** and **Interpreter** patterns.

Object Patterns: In object-based behavioral patterns, a group of objects cooperates to perform a task that no single object can accomplish alone. These patterns focus on the interactions and communication between objects to achieve a specific behavior. Examples of object-based behavioral patterns include

the **Observer**, **Command**, **Mediator**, **Memento**, **Flyweight**, **State**, **Visitor**, **Chain of Responsibility**, and **Strategy** patterns.



It's important to note that these categorizations are not mutually exclusive, and some patterns may fall into multiple categories or have variations in their implementation. Some of the patterns are also used as an alternate to each other. Some of the patterns have lot of similarities in code implementation even though the motive might be different. The intent of each pattern should be considered based on its specific use case and the problem it addresses.

1.5. Summary

In this chapter, we introduced basic design principles of software development in object-oriented programming and catalogued design patterns based on their purpose and scope. In the next few chapters, we shall be looking at some of the key design patterns in details and observe how they can be used.