**Coding Against an Interface Rather Than a Class**

When we say **"coding against an interface rather than a class,"** we mean that we should write our code to depend on general definitions of behaviors (the **interface**) instead of relying on specific, concrete examples (the **class**).

Imagine you are designing a system for sending notifications. Instead of directly using a specific class like EmailService to send emails, you would use an **interface** called INotificationService. The interface would define **what** methods should exist (like Send()), but not **how** they are implemented (whether it's email, SMS, etc.).

Why is this useful?

* **Flexibility**: If later you decide to change from sending emails to sending SMS, you can do so without changing the rest of your code. You just implement the INotificationService interface differently, and the code that uses the interface remains the same.
* **Testing**: It makes it easier to test. You can create mock versions of INotificationService to simulate behavior during testing, without needing to rely on a real email or SMS service.

**2. Coding Against Abstraction, Not Concreteness**

**Abstraction** is the concept of hiding the complex details and showing only the essential features. "Coding against abstraction" means you're focusing on the general idea or the **contract** (what the object should do), instead of the specific details (how it does it).

For example, let's say you're building a system to process payments. You don’t care whether the payment is processed through a credit card, PayPal, or bank transfer. You just care about the **action** of processing a payment. So, you create an abstraction like a **PaymentProcessor**, which knows how to **process a payment**. The details (which payment method is used) are hidden away in specific implementations (like CreditCardPayment, PaypalPayment), which all follow the same abstract process of "payment."

Why is this useful?

* **Simplified Design**: By coding against the **abstraction**, you don't need to worry about the specific details of how things work. You only care about the high-level operation (like processing a payment, drawing a shape, etc.).
* **Flexibility for Change**: If you need to change how payments are processed (say, moving from credit cards to crypto payments), you only need to change the concrete implementation. The rest of the code doesn’t need to change because it relies on the abstraction (the general idea of "payment").

**3. Abstraction as a Guideline and How We Can Implement It**

**Abstraction as a guideline** means we should design our systems in a way that hides unnecessary complexity and focuses on the **essence** of what needs to be done. It allows us to deal with the "what" instead of the "how." We should use abstraction to define clear interfaces, classes, or methods that specify **what** a system should do, not **how** exactly it will do it.

For example, in a **payment system**, the abstraction would define actions like ProcessPayment(). The **how** (whether it's using a credit card, PayPal, etc.) is not important at the abstract level. Each specific method (like CreditCardPayment) handles the details.

Implementing this in real-world software:

* **Interfaces**: Create general interfaces like INotificationService or IPaymentProcessor that define the key actions, but don’t dictate how they should be carried out.
* **Encapsulation**: Hide unnecessary details behind abstract methods, leaving the details to be handled by specific classes.

**Why is Abstraction Useful?**

1. **Simplifies the Code**: By focusing on the big picture and abstracting away unnecessary details, your code becomes easier to understand.
2. **Easier to Extend**: You can easily add new features or components by creating new classes that follow the existing abstraction.
3. **Reduces Dependency**: Other parts of the system don’t need to worry about the implementation details of other components, making it easier to change or replace them.

**In Summary:**

* **Coding against interfaces or abstractions** means focusing on the **what** something does, not the **how** it’s done. This makes your code more flexible and easier to maintain.
* **Abstraction** allows you to hide complexity and focus only on the essential parts of your code. You can design systems where components follow general contracts (abstract actions) without being tied to specific implementations.

By following abstraction principles, we build more flexible, maintainable, and scalable systems!