Design Principles Session

Fundamental/Blueprint of the building before construction

Before implementation also, we have design

Cost of dev ⬇️ less

Set of guidelines

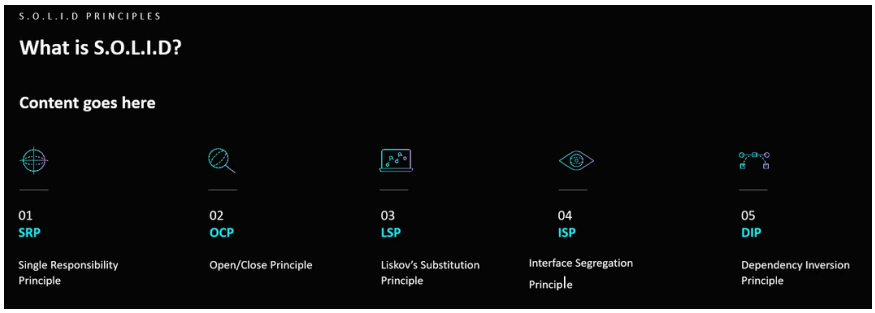
SOLID Principles (Subset of design)

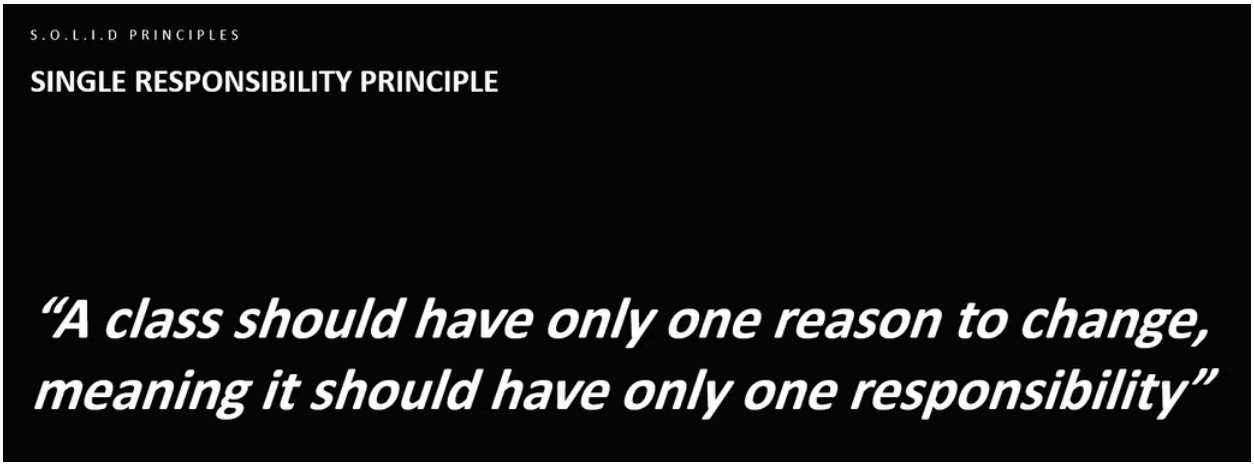
Introduction

SOLID principles and broader design principles form the backbone of modern software engineering, guiding us toward more maintainable, flexible, and efficient code.

- **SOLID** (introduced by Robert C. Martin) provides five interconnected guidelines that create robust object-oriented designs—like a well-engineered building that can adapt to changing requirements without compromising structural integrity.
- **Design Principles** complement SOLID by offering a wider framework of best practices for creating elegant software architectures—think of them as the blueprint for crafting systems that are not only functional but also sustainable, scalable, and comprehensible.

Together, these principles transform how we approach software development, shifting our focus from merely "making it work" to "making it work well for years to come." They represent the collective wisdom of our industry, distilled into practical guidance for building systems that thrive in ever-changing environments.





Scenario:

Imagine you are processing orders in an e-commerce application. In an e-commerce application we need to

1. Manage order details (products, quantities, prices).
2. Process payments (handle transactions).
3. Send invoices (generate and email invoices)

A bad design violates SRP by handling all these responsibilities in a single Order class. A good design follows SRP by separating these concerns into dedicated classes.

S.O.L.I.D PRINCIPLES

Single Responsibility Principle

CODE VIOLATING SRP:

Problems in the Above Code

1. Multiple Responsibilities:
- processPayment() handles **payment logic**.
- generateInvoice() handles **invoice logic**.
- sendInvoiceEmail() handles **email sending**.

The Order class should only store order data!

2. Difficult Maintenance:
- If we change the payment gateway, we modify Order.
- If we modify invoice generation, we modify Order.
- If we switch email services, we modify Order.

3. Tightly Coupled Code:
- Payment, invoice, and email logic **depend on the Order class**.
- Testing one feature requires dealing with **unrelated functionality**.

CODE SNIPPET PART-1:

```
class Order {
    private String orderId;
    private double amount;
    public Order(String orderId, double amount) {
        this.orderId = orderId;
        this.amount = amount;
    }
    public String getOrderId() {
        return orderId;
    }
    public double getAmount() {
        return amount;
    }
    // Business Logic: Payment processing
    public void processPayment(String paymentMethod) {
        System.out.println("Processing payment of $" + amount + " using " + paymentMethod);
        // Payment gateway logic here...
    }

    // Business Logic: Invoice generation
    public void generateInvoice() {
        System.out.println("Generating invoice for Order ID: " + orderId);
        // Invoice generation logic here...
    }

    // Sending invoice via email
    public void sendInvoiceEmail() {
        System.out.println("Sending invoice email for Order ID: " + orderId);
        // Email logic here...
    }
}
```

<epam>

EPAM Proprietary & Confidential. 10

When you change payment logic/invoice Order class is modified : Bad Practice (Tech. Debt)

Refactor the code

S.O.L.I.D PRINCIPLES

Single Responsibility Principle

REFACTORED CODE THAT FOLLOWS SRP:

We refactor the code into **three separate classes**, each with a **single responsibility**:

1.Each class has a single responsibility:

- Order stores order details.
- PaymentProcessor handles payment processing.
- InvoiceService manages invoices and emails.

2. **Better Maintainability:**

- If we change the payment method, we only modify PaymentProcessor.
- If we add new invoice features, we only update InvoiceService.

3. **More Testable Code:**

- We can test payment processing, invoices, and order logic independently.

CODE SNIPPET PART-1:

```java
// 1. This class only holds order data
class Order {
    private String orderId;
    private double amount;
    public Order(String orderId, double amount) {
        this.orderId = orderId;
        this.amount = amount;
    }

    public String getOrderId() {
        return orderId;
    }
    public double getAmount() {
        return amount;
    }
}
```

```java
// 2. This class is responsible for payment processing
class PaymentProcessor {
    public void processPayment(Order order, String paymentMethod) {
        System.out.println("Processing payment of $" + order.getAmount() + " using " + paymentMethod);
        // Payment gateway logic here...
    }
}
// 3. This class is responsible for invoice generation and email sending
class InvoiceService {
    public void generateInvoice(Order order) {
        System.out.println("Generating invoice for Order ID: " + order.getOrderId());
        // Invoice generation logic here...
    }

    public void sendInvoiceEmail(Order order) {
        System.out.println("Sending invoice email for Order ID: " + order.getOrderId());
        // Email logic here...
    }
}
// Testing the SRP-compliant code
public class Orderg {
    Run | Debug
    public static void main(String[] args) {
        Order order = new Order(orderId:"ORD12345", amount:250.75);
        PaymentProcessor paymentProcessor = new PaymentProcessor();
        InvoiceService invoiceService = new InvoiceService();

        paymentProcessor.processPayment(order, paymentMethod:"Credit Card");
        invoiceService.generateInvoice(order);
        invoiceService.sendInvoiceEmail(order);
    }
}
```

Methods with same logic can be clubbed together

Now modification of invoice/payment doesn't have to deal with order class → More maintainable/testable

SOLID PRINCIPLES

OPEN/CLOSE PRINCIPLE

"A class should be open for extension but closed for modification."

Whenever we have class, derived class should not modify behaviour of the parent class

Say different discounts (%

S.O.L.I.D PRINCIPLES

Open/Close Principle

CODE VIOLATING OCP:

Problems in the Above Code

1. Violates OCP:

- Every time a new discount type is introduced, we must modify the DiscountCalculator class, leading to high maintenance overhead.

2. Hard to Scale:

- Adding a "Loyalty Discount" or "Student Discount" requires editing the same method and increasing the complexity.

3. Poor Testability & Maintainability:

- Unit testing is harder since changes to calculateDiscount() affect multiple discount types.

CODE SNIPPET:

```
class DiscountCalculator {
    public double calculateDiscount(String discountType, double price) {
        if (discountType.equals(anObject:"PERCENTAGE")) {
            return price * 0.10; // 10% discount
        } else if (discountType.equals(anObject:"FIXED")) {
            return 20; // Fixed $20 discount
        } else if (discountType.equals(anObject:"SEASONAL")) {
            return price * 0.15; // 15% seasonal discount
        } else {
            return 0; // No discount
        }
    }
}
```

Unit test is harder,

S.O.L.I.D PRINCIPLES

Open/Close Principle

REFACTORED CODE FOLLOWING OCP:

Why is this a good example

1. Follows OCP:

- The OrderProcessor class does not change when a new discount type is added.
- Instead, new classes (strategies) are created, adhering to open for extension but closed for modification.

2. Scalable and Maintainable:

- Adding a LoyaltyDiscount or BulkPurchaseDiscount only requires creating a new class implementing DiscountStrategy, without modifying OrderProcessor.

3. Better Testability:

- Each discount strategy can be tested independently.

CODE SNIPPET PART-1:

```
interface DiscountStrategy {
    double applyDiscount(double price);
}

// Step 2: Implement different discount strategies

class PercentageDiscount implements DiscountStrategy {
    private double percentage;

    public PercentageDiscount(double percentage) {
        this.percentage = percentage;
    }

    @Override
    public double applyDiscount(double price) {
        return price * (percentage / 100);
    }
}

class FixedDiscount implements DiscountStrategy {
    private double amount;

    public FixedDiscount(double amount) {
        this.amount = amount;
    }

    @Override
    public double applyDiscount(double price) {
        return amount;
    }
}
```

SOLID PRINCIPLES

LISKOV'S SUBSTITUTION PRINCIPLE

"Subtypes must be substitutable for their base types without affecting the correctness of the program."

Say inside a class we have several methods, new subclasses should be able to have those methods without creating any disturbances in main class

Scenario:

In an online payment system,where users can pay using different payment methods:

1. Credit Card- Requires Authorization
2. Paypal- Requires Authorization
3. Cryptocurrency(Bitcoin, Ethereum etc.,)- Doesn't require authorization

If we **incorrectly design** the system by enforcing authentication on all payments, **CryptoPayment** will have to override or ignore authentication-related methods, breaking Liskov Substitution. A **proper design** ensures that all subclasses can be used interchangeably without breaking functionality.

Say auth on all payments, but in crypto we don’t need

S.O.L.I.D PRINCIPLES

Liskov Substitution Principle (LSP)

CODE VIOLATING LSP:

Problems in the above code:

1. LSP Violation:

- The base class (Payment) enforces authentication, which is not required for CryptoPayment.
- CryptoPayment breaks the expected behavior by throwing an exception in authenticateUser().

2. Incorrect Abstraction:

- Every payment method should not be forced to implement authentication.
- The CryptoPayment subclass cannot be used as a valid substitute for the Payment superclass.

CODE SNIPPET PART-1:

```
abstract class Payment {
    abstract void authenticateUser(); // Enforces authentication on all payments
    abstract void processPayment(double amount);
}

class PayPalPayment extends Payment {
    @Override
    void authenticateUser() {
        System.out.println(x:"Authenticating PayPal user...");
    }

    @Override
    void processPayment(double amount) {
        System.out.println("Processing PayPal payment of $" + amount);
    }
}

class CreditCardPayment extends Payment {
    @Override
    void authenticateUser() {
        System.out.println(x:"Authenticating Credit Card user...");
    }

    @Override
    void processPayment(double amount) {
        System.out.println("Processing Credit Card payment of $" + amount);
    }
}
```

S.O.L.I.D PRINCIPLES

Liskov’s Substitution Principle

REFACTORED CODE THAT FOLLOWS LSP:

Why is this a Good Example?

1. Follows Liskov Substitution Principle

- (LSP)CryptoPayment, PayPalPayment, and CreditCardPayment can be used interchangeably without breaking the system.
- CryptoPayment does not need to override an unnecessary authentication method.

2. Correct Abstraction

- Authentication is only required for some payments, so it is moved into a separate interface (AuthenticatedPayment).

3. Scalable & Extendable

- Adding new payment methods (e.g., ApplePay or BankTransfer) without authentication is easy.

CODE SNIPPET PART-2:

```
// Step 4: Implement payments that do NOT require authentication
class CryptoPayment implements Payment {
    @Override
    public void processPayment(double amount) {
        System.out.println("Processing Cryptocurrency payment of $" + amount);
    }
}

// Step 5: Payment Processor that correctly handles different payment types
public class PaymentProcessorg {
    public static void process(AuthenticatedPayment payment, double amount) {
        payment.authenticateUser();
        payment.processPayment(amount);
    }

    public static void process(Payment payment, double amount) {
        payment.processPayment(amount);
    }

    Run | Debug
    public static void main(String[] args) {
        AuthenticatedPayment paypal = new PayPalPayment();
        AuthenticatedPayment creditCard = new CreditCardPayment();
        Payment crypto = new CryptoPayment();

        // Process payments correctly
        process(paypal, amount:100);      // ✅ Authenticates and processes PayPal
        process(creditCard, amount:200);  // ✅ Authenticates and processes Credit Card
        process(crypto, amount:500);      // ✅ Directly processes Crypto without authentication
    }
}
```

S.O.L.I.D PRINCIPLES

INTERFACE SEGREGATION PRINCIPLE

"A Class should not be forced to implement methods it doesn't need."

S.O.L.I.D PRINCIPLES

INTERFACE SEGREGATION PRINCIPLE

Scenario:

In a food delivery app like Uber Eats or Swiggy, there are three different user roles:

Customers: Place food orders.

Restaurant Owners: Prepare and manage food orders.

Delivery Partners: Deliver the food.

A bad design would force all these roles to implement all functionalities, even if they don't need them. For example, a Customer should not have to prepare or deliver food, and a Delivery Partner should not place orders.

S.O.L.I.D PRINCIPLES

Interface Segregation Principle (ISP)

CODE VIOLATING ISP:

Problems in the above code:

1. Violates Interface Segregation Principle (ISP)

- Every class must implement all methods, even if they do not apply.
- Results in empty implementations or exceptions, which is a bad practice.

2. Difficult to Maintain

- If a new role is added (e.g., a Kitchen Staff role), the interface will have to be modified, breaking existing code.

CODE SNIPPET PART-1:

```
interface FoodDeliveryApp {
    void placeOrder();
    void prepareOrder();
    void deliverOrder();
}

class Customer implements FoodDeliveryApp {
    @Override
    public void placeOrder() {
        System.out.println(x:"Customer places a food order.");
    }

    @Override
    public void prepareOrder() {
        throw new UnsupportedOperationException(message:"Customers cannot prepare food.");
    }

    @Override
    public void deliverOrder() {
        throw new UnsupportedOperationException(message:"Customers cannot deliver food.");
    }
}
```

CODE SNIPPET PART-2

```
class RestaurantOwner implements FoodDeliveryApp {
    @Override
    public void placeOrder() {
        throw new UnsupportedOperationException(message:"Restaurant owners do not place orders.");
    }

    @Override
    public void prepareOrder() {
        System.out.println(x:"Restaurant prepares the food order.");
    }

    @Override
    public void deliverOrder() {
        throw new UnsupportedOperationException(message:"Restaurant owners do not deliver food.");
    }
}

class DeliveryPartner implements FoodDeliveryApp {
    @Override
    public void placeOrder() {
        throw new UnsupportedOperationException(message:"Delivery partners do not place orders.");
    }

    @Override
    public void prepareOrder() {
        throw new UnsupportedOperationException(message:"Delivery partners do not prepare food.");
    }

    @Override
    public void deliverOrder() {
        System.out.println(x:"Delivery partner delivers the food.");
    }
}
```

every class is implementing all the methods which it doesn’t require → Refactor

S.O.L.I.D PRINCIPLES

Interface Segregation Principle

REFACTORED CODE THAT FOLLOWS ISP:

Why is this a Good Example?

1. Follows Interface Segregation Principle

- (ISP)Each class implements only the interface relevant to its role.
- No unnecessary methods.

2. More Maintainable & Scalable

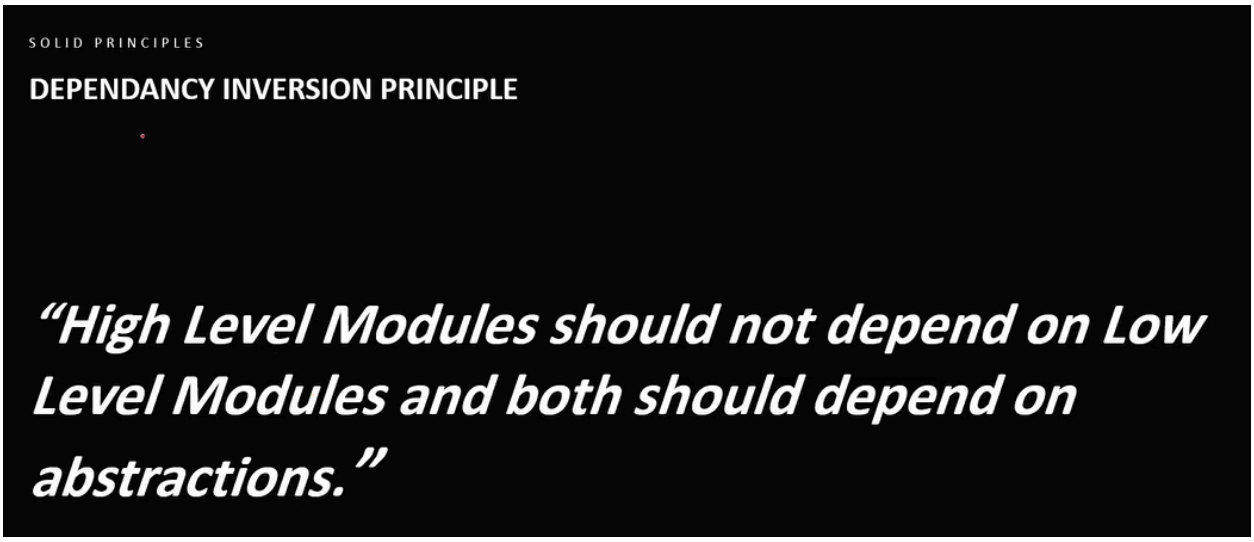
- If we introduce a new role (e.g., Kitchen Staff), we can add a new interface without modifying existing classes.

3. Prevents Empty Implementations

- No need for UnsupportedOperationException, which was present in the bad example.

CODE SNIPPET PART-1:

```
interface OrderPlacer {
    void placeOrder();
}

interface OrderPreparer {
    void prepareOrder();
}

interface OrderDeliverer {
    void deliverOrder();
}

// Implement only relevant interfaces

class Customer implements OrderPlacer {
    @Override
    public void placeOrder() {
        System.out.println(x:"Customer places a food order.");
    }
}

class RestaurantOwner implements OrderPreparer {
    @Override
    public void prepareOrder() {
        System.out.println(x:"Restaurant prepares the food order.");
    }
}
```



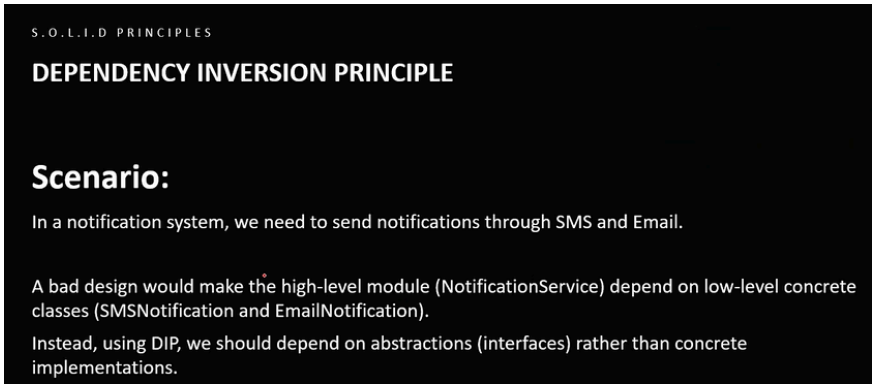
Classes should depend on abstractions (Interfaces/Abstract Classes)

Bakery → Burger Class and Pizza Class used to create burgers and pizza

Chef depends on Burger Class and Pizza Class

Burger Class and Pizza → Low Level Module

Chef should depend on interface



S.O.L.I.D PRINCIPLES

Dependecy Inversion Principle (DIP)

CODE VIOLATING DIP:

1. Violates Dependency Inversion Principle (DIP):

- High-level module (NotificationService) is tightly coupled to low-level modules (SMSNotification, EmailNotification).
- If a new notification method (e.g., Push Notification) is added, NotificationService must be modified, violating Open-Closed Principle as well.

2. Difficult to Extend

- Adding a new notification type requires modifying existing code, which can introduce bugs.

3. Difficult to Unit Test

- Since NotificationService depends on concrete classes (SMSNotification, EmailNotification), we cannot easily mock them for testing.

<epam>

CODE SNIPPET PART-1:

```
class SMSNotification {
    void sendSMS(String message) {
        System.out.println("Sending SMS: " + message);
    }
}

class EmailNotification {
    void sendEmail(String message) {
        System.out.println("Sending Email: " + message);
    }
}

// High-Level module directly depends on low-level classes
class NotificationService {
    private SMSNotification smsNotification;
    private EmailNotification emailNotification;

    public NotificationService() {
        this.smsNotification = new SMSNotification(); // Tight coupling
        this.emailNotification = new EmailNotification(); // Tight coupling
    }

    void sendNotification(String type, String message) {
        if (type.equals(anObject:"SMS")) {
            smsNotification.sendSMS(message);
        } else if (type.equals(anObject:"EMAIL")) {
            emailNotification.sendEmail(message);
        }
    }
}
```


S.O.L.I.D PRINCIPLES

Dependency Inversion Principle

REFACTORED CODE THAT FOLLOWS DIP:

Why is this a Good Example?

1. Follows Dependency Inversion Principle (DIP)

- High-level module (NotificationService) does not depend on low-level modules (SMSNotification, EmailNotification).
- It depends on the abstraction (NotificationSender) instead.

2. Easier to Extend

- If we need to add a new notification type (e.g., Push Notification), we can create a new class without modifying NotificationService.

3. Improves Maintainability & Flexibility

- The dependency is injected at runtime, making it more flexible.

CODE SNIPPET PART-1:

```
interface NotificationSender {
    void send(String message);
}

// Low-Level modules implement the interface
class SMSNotification implements NotificationSender {
    @Override
    public void send(String message) {
        System.out.println("Sending SMS: " + message);
    }
}

class EmailNotification implements NotificationSender {
    @Override
    public void send(String message) {
        System.out.println("Sending Email: " + message);
    }
}
```

S.O.L.I.D PRINCIPLES

Dependency Inversion Principle

REFACTORED CODE THAT FOLLOWS DIP:

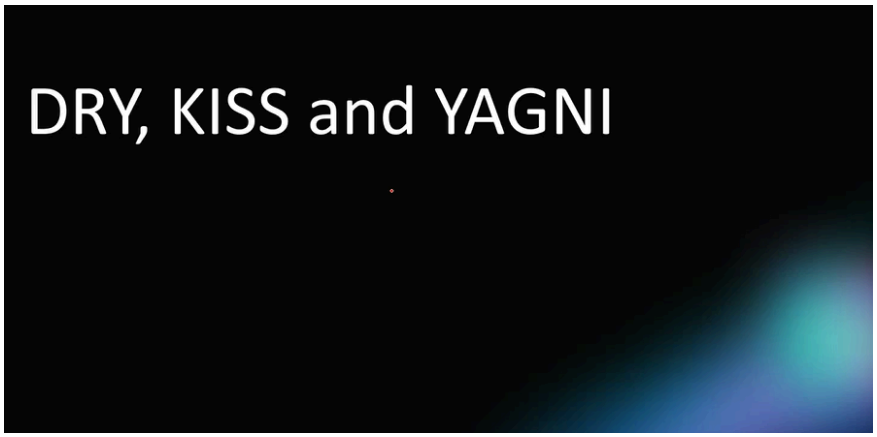
4. Easier Unit Testing:

- We can mock NotificationSender to test NotificationService without using real SMS or Email services.

CODE SNIPPET PART-2:

```
// High-level module depends on the abstraction
class NotificationService {
    private final NotificationSender notificationSender;

    // Dependency Injection via Constructor
    public NotificationService(NotificationSender notificationSender) {
        this.notificationSender = notificationSender;
    }

    void sendNotification(String message) {
        notificationSender.send(message);
    }
}

public class NotificationApp {
    Run | Debug
    public static void main(String[] args) {
        NotificationSender smsSender = new SMSNotification();
        NotificationSender emailSender = new EmailNotification();

        NotificationService smsService = new NotificationService(smsSender);
        NotificationService emailService = new NotificationService(emailSender);

        smsService.sendNotification(message:"Your order has been shipped.");
        emailService.sendNotification(message:"Your invoice is attached.");
    }
}
```

Dependency Injection → Passing Notification sender (interface) directly in notification service



Introduction

The DRY (Don't Repeat Yourself), KISS (Keep It Simple, Stupid), and YAGNI (You Ain't Gonna Need It) principles help developers create scalable and optimized solutions by reducing redundancy, keeping code simple, and avoiding unnecessary complexity.

Today we will be covering the following Design principles:

1. DRY(Do not repeat yourself)
2. KISS(Keep It Simple, Stupid)
3. YAGNI(You ain't gonna need it).

DESIGN PRINCIPLES

DRY- Do Not Repeat Yourself

"Avoid Code Duplication by Abstraction and Modularity"

Scenario:

Consider a notification system where both customers and restaurant owners receive order status notification via email and SMS, where the logic for sending notification through SMS and Email is similar.

DESIGN PRINCIPLES

Do Not Repeat Yourself (DRY)

CODE VIOLATING DRY:

1. Code duplication:
   - Both methods format messages the same way.

2. Hard to maintain:

   If we change the message format, we must update multiple places.

3. Violates DRY:
   - The logic should be centralized.

CODE SNIPPET:

```
class NotificationService {
    public void sendEmailNotification(String recipientEmail, String message) {
        String formattedMessage = "[Food Delivery] " + message;
        System.out.println("Sending Email to " + recipientEmail + ": " + formattedMessage);
    }

    public void sendSmsNotification(String recipientPhone, String message) {
        String formattedMessage = "[Food Delivery] " + message;
        System.out.println("Sending SMS to " + recipientPhone + ": " + formattedMessage);
    }
}
```

Change → Multiple places

DESIGN PRINCIPLES

Do Not Repeat Yourself (DRY)

REFRACTORED CODE TO SOLVE THE PROBLEM:

Why the refactored code is better

1. Avoids duplication:
   - centralizes message formatting to avoid duplication
2. Easier maintenance:
   - If we change the message format, it's updated in one place.
3. More readable and scalable:
   - The notification methods now focus only on sending messages.

CODE SNIPPET :

```
class NotificationService {
    private String formatMessage(String message) {
        return "[Food Delivery] " + message;
    }

    public void sendEmailNotification(String recipientEmail, String message) {
        System.out.println("Sending Email to " + recipientEmail + ": " + formatMessage(message));
    }

    public void sendSmsNotification(String recipientPhone, String message) {
        System.out.println("Sending SMS to " + recipientPhone + ": " + formatMessage(message));
    }
}
```

DESIGN PRINCIPLES

KISS- Keep It Simple, Stupid

"Simplicity in design leads to easier maintenance"

easy to understand, unit test cases, find bugs rather than complicated code

Scenario:

Consider a Parking System that calculates the parking fee based on the number of hours a vehicle is parked, We need to keep the calculation logic for parking fee as simple as possible

DESIGN PRINCIPLES

Keep It Simple (KISS)

CODE VIOLATING KISS:

1. Poor Readability:
   - Too many nested conditions make it hard to read.
2. Code duplication:
   - Similar logic is repeated for different vehicle types.
3. Difficult to extend:
   - Adding a new vehicle type requires modifying multiple conditions.

CODE SNIPPET:

```
class ParkingSystem {
    public double calculateFee(int hours, String vehicleType) {
        double fee = 0;

        if (vehicleType.equalsIgnoreCase(anotherString:"car")) {
            if (hours <= 2) {
                fee = 5;
            } else if (hours > 2 && hours <= 5) {
                fee = 10;
            } else {
                fee = 10 + (hours - 5) * 2;
            }
        } else if (vehicleType.equalsIgnoreCase(anotherString:"bike")) {
            if (hours <= 2) {
                fee = 3;
            } else if (hours > 2 && hours <= 5) {
                fee = 6;
            } else {
                fee = 6 + (hours - 5) * 1.5;
            }
        } else if (vehicleType.equalsIgnoreCase(anotherString:"truck")) {
            if (hours <= 2) {
                fee = 10;
            } else if (hours > 2 && hours <= 5) {
                fee = 15;
            } else {
                fee = 15 + (hours - 5) * 3;
            }
        }

        return fee;
    }
}
```

DESIGN PRINCIPLES

Keep It Simple, Stupid (KISS)

REFRACTORED CODE TO SOLVE THE PROBLEM:

Why the refactored code is better

Reduces Redundancy:
- Eliminates redundant conditions by storing values in a map.

Easy to extend:
- Adding a new vehicle type only requires updating the maps.

More readable and maintainable:
- The logic is straightforward.

CODE SNIPPET :

```
import java.util.Map;

class ParkingSystem {
    private static final Map<String, Double> BASE_FEES = Map.of(
        k1:"car", v1:5.0,
        k2:"bike", v2:3.0,
        k3:"truck", v3:10.0
    );

    private static final Map<String, Double> ADDITIONAL_FEES = Map.of(
        k1:"car", v1:2.0,
        k2:"bike", v2:1.5,
        k3:"truck", v3:3.0
    );

    public double calculateFee(int hours, String vehicleType) {
        double baseFee = BASE_FEES.getOrDefault(vehicleType.toLowerCase(), defaultValue:0.0);
        double additionalFee = ADDITIONAL_FEES.getOrDefault(vehicleType.toLowerCase(), defaultValue:0.0);

        return hours <= 5 ? baseFee : baseFee + (hours - 5) * additionalFee;
    }
}
```

DESIGN PRINCIPLES

YAGNI- You A'int Gonna Need It.

"Don't Implement Functionality Until It's Necessary"

Focus on sprint/shouldn't spend time on other extra deatures

Scenario:

Consider a library management system where users can borrow and return books. The initial requirement is to implement book borrowing and returning functionality. However, a developer prematurely adds a "book recommendation system" that is not required at the moment.

DESIGN PRINCIPLES

You Ain't Gonna Need It (YAGNI)

CODE VIOLATING YAGNI:

1. Added unnecessary functionality that was not requested.
2. Increases code complexity without adding immediate value.
3. Wastes development time that could have been spent on core features.

CODE SNIPPET PART -1:

```
class Book {
    String title;
    boolean isBorrowed;

    public Book(String title) {
        this.title = title;
        this.isBorrowed = false;
    }
}

class Library {
    private List<Book> books = new ArrayList<>();

    public void addBook(String title) {
        books.add(new Book(title));
    }

    public void borrowBook(String title) {
        for (Book book : books) {
            if (book.title.equalsIgnoreCase(title) && !book.isBorrowed) {
                book.isBorrowed = true;
                System.out.println(title + " has been borrowed.");
                return;
            }
        }
        System.out.println(title + " is not available.");
    }
```

DESIGN PRINCIPLES

You Ain't Gonna Need It (YAGNI)

CODE VIOLATING YAGNI:

1. Added unnecessary functionality that was not requested.
2. Increases code complexity without adding immediate value.
3. Wastes development time that could have been spent on core features.

CODE SNIPPET PART -2:

```
    public void returnBook(String title) {
        for (Book book : books) {
            if (book.title.equalsIgnoreCase(title) && book.isBorrowed) {
                book.isBorrowed = false;
                System.out.println(title + " has been returned.");
                return;
            }
        }
        System.out.println(title + " was not borrowed.");
    }
    // **Unnecessary feature:** A recommendation system that is not needed yet
    public List<String> recommendBooks() {
        List<String> recommendedBooks = new ArrayList<>();
        for (Book book : books) {
            if (!book.isBorrowed) {
                recommendedBooks.add(book.title);
            }
        }
        System.out.println("Recommended books: " + recommendedBooks);
        return recommendedBooks;
    }
}
public class libraryb {
    Run|Debug
    public static void main(String[] args) {
        Library library = new Library();
        library.addBook(title:"The Alchemist");
        library.addBook(title:"1984");
        library.borrowBook(title:"The Alchemist");
        library.returnBook(title:"The Alchemist");
        // **Unnecessary call:** This feature was never requested
        library.recommendBooks();
    }
}
```

DESIGN PRINCIPLES

You Ain't Gonna Need It (YAGNI)

REFRACTORED CODE TO SOLVE THE PROBLEM:

Why the refactored code is better

1. Does not include unnecessary features (like book recommendations).
2. Simpler and more maintainable: Only implements what is needed.
3. Faster development and deployment by focusing on essential features.

CODE SNIPPET PART -1 :

```
class Book {
    String title;
    boolean isBorrowed;

    public Book(String title) {
        this.title = title;
        this.isBorrowed = false;
    }
}

class Library {
    private List<Book> books = new ArrayList<>();

    public void addBook(String title) {
        books.add(new Book(title));
    }

    public void borrowBook(String title) {
        for (Book book : books) {
            if (book.title.equalsIgnoreCase(title) && !book.isBorrowed) {
                book.isBorrowed = true;
                System.out.println(title + " has been borrowed.");
                return;
            }
        }
        System.out.println(title + " is not available.");
    }
```

DESIGN PRINCIPLES

You Ain't Gonna Need It (YAGNI)

REFRACTORED CODE TO SOLVE THE PROBLEM:

Why the refactored code is better

1. Does not include unnecessary features (like book recommendations).
2. Simpler and more maintainable: Only implements what is needed.
3. Faster development and deployment by focusing on essential features.

Key Takeaways

- YAGNI prevents adding features that are not needed now.
- Focus only on the requirements at hand.
- Avoid over-engineering to keep the code clean and manageable.

CODE SNIPPET PART -2 :

```
    public void returnBook(String title) {
        for (Book book : books) {
            if (book.title.equalsIgnoreCase(title) && book.isBorrowed) {
                book.isBorrowed = false;
                System.out.println(title + " has been returned.");
                return;
            }
        }
        System.out.println(title + " was not borrowed.");
    }
}

public class libraryg {
    Run | Debug
    public static void main(String[] args) {
        Library library = new Library();
        library.addBook(title:"The Alchemist");
        library.addBook(title:"1984");

        library.borrowBook(title:"The Alchemist");
        library.returnBook(title:"The Alchemist");
    }
}
```

S.O.L.I.D PRINCIPLES AND DESIGN PRINCIPLES

WHAT NEXT?

Now that you understand **SOLID, KISS, DRY, and YAGNI**, the next step is applying these principles effectively.

Explore:

- **Design Patterns** – Reusable solutions to common problems (Factory, Singleton, Strategy, etc.).
- **Architectural Patterns** – Structuring applications efficiently (MVC, Microservices, Event-Driven).

S.O.L.I.D PRINCIPLES AND DESIGN PRINCIPLES

CONCLUSION

Scenario:

In software development, following **SOLID principles** ensures our code is **scalable, maintainable, and flexible**, making it easier to adapt to future changes. Meanwhile, **design principles like DRY, KISS, and YAGNI** help us write **clean, efficient, and focused** code by eliminating redundancy, keeping solutions simple, and avoiding unnecessary complexity.

By applying these principles, we create software that is **robust, easy to maintain, and adaptable to evolving requirements**—ultimately leading to **better productivity and fewer bugs** in the long run.

“ Write Less Think More and Code Smart”

VANKA ESHWAR PRABHAS

TURLAPATI SAI KRISHNA

eshwarprabhas_vanka@epam.com

turlapatisai_krishna@epam.com