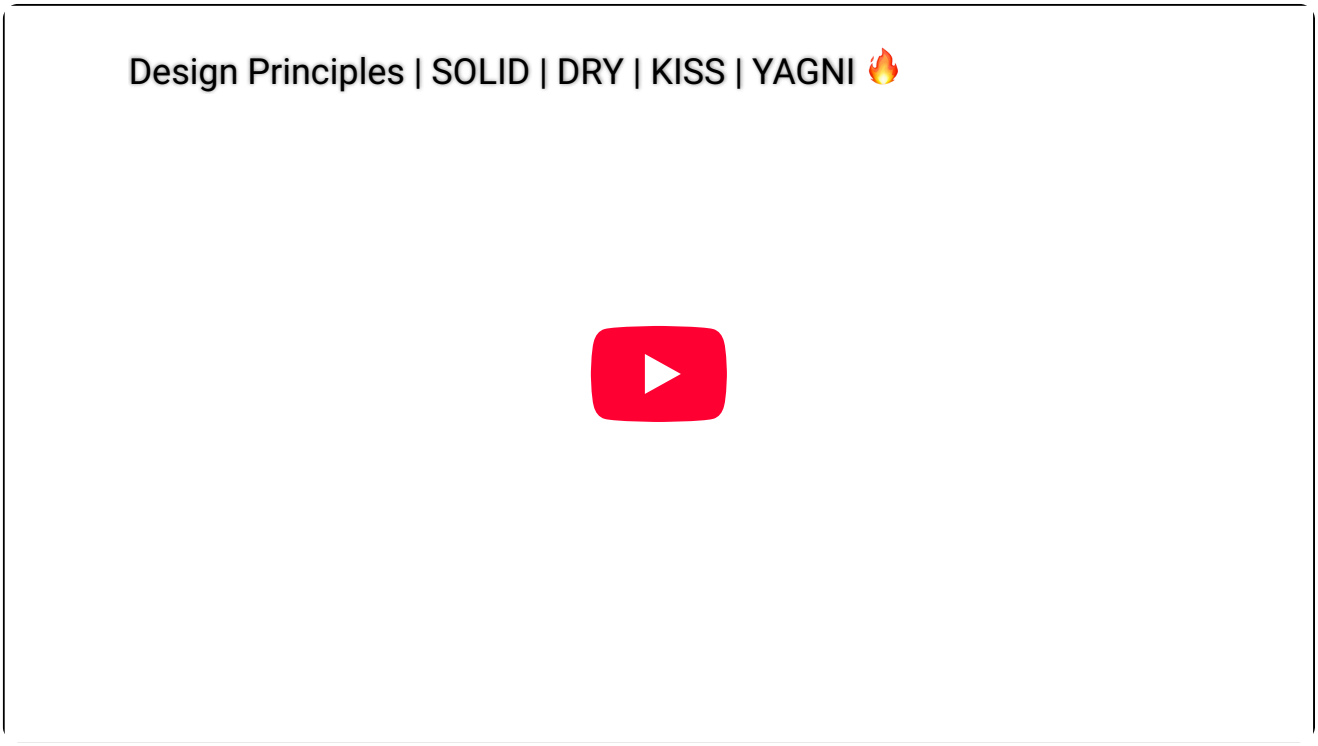
Search articles...

SOLID Principles



Topic Tags:

Design Patterns System Design LLD

🐈‍⬛ Github Codes Link: https://github.com/aryan-0077/CWA-LowLevelDesignCode

Software development is a complex field that requires careful planning and design to ensure that applications are maintainable, scalable, and easy to understand. One of the foundational guidelines for achieving these goals is the set of SOLID principles. 🛠️📊

SOLID principles are five essential guidelines that enhance software design, making code more maintainable and scalable. They include:
1. Single Responsibility Principle 🔑
2. Open/Closed Principle 🔒
3. Liskov Substitution Principle 🔄
4. Interface Segregation Principle 🔪
5. Dependency Inversion Principle 🔗

Need for SOLID Principles in Object-Oriented Design:

Below are some of the main reasons why SOLID principles are important in object-oriented design:

• **Scalability**: Adding new features becomes straightforward. 📈
• **Maintainability**: Changes in one part of the system have minimal impact on others. 🔄
• **Testability**: Decoupled designs make unit testing easier. 🧪
• **Readability**: Clear separation of concerns improves code comprehension. 📚

Adopting SOLID principles is a key step toward mastering clean code and professional software development. 💡 While it takes practice to apply these principles effectively, the long-term benefits for both developers and businesses are undeniable. 🚀

📦 Single Responsibility Principle

This principle states that A class should have only one reason to change which means every class should have a single responsibility or single job or single purpose. In other words, a class should have only one job or purpose within the software system.

🌟 Why it matters:

• Improves maintainability: Changes in one class do not affect unrelated parts.
• Enhances readability and reduces complexity.

💭 Example:

Imagine a baker who is responsible for baking bread. The baker's role is to focus on the task of baking bread, ensuring that the bread is of high quality, properly baked, and meets the bakery's standards.
However, if the baker is also responsible for managing the inventory, ordering supplies, serving customers, and cleaning the bakery, this would violate the SRP. Each of these tasks represents a separate responsibility, and by combining them, the baker's focus and effectiveness in baking bread could be compromised.

Java

```java
// Class with multiple responsibilities
class BreadBaker {
 public
  void bakeBread() { System.out.println("Baking high-quality bread.

 public
  void manageInventory() { System.out.println("Managing inventory..

 public
  void orderSupplies() { System.out.println("Ordering supplies...")

 public
  void serveCustomer() { System.out.println("Serving customers...")

```

```
15  public
16   void cleanBakery() { System.out.println("Cleaning the bakery...")
17
18  public
19   static void main(String[] args) {
20     BreadBaker baker = new BreadBaker();
21     baker.bakeBread();
22     baker.manageInventory();
23     baker.orderSupplies();
24     baker.serveCustomer();
25     baker.cleanBakery();
26   }
27 }
```

In the above code example, the BreadBaker class has multiple responsibilities: baking bread, managing inventory, ordering supplies, serving customers, and cleaning the bakery. This violates the Single Responsibility Principle (SRP) because the class has more than one reason to change. If any of these responsibilities need to be modified, the BreadBaker class would need to be changed, making the code harder to maintain and understand. For instance, if there is a change in the way inventory is managed, the BreadBaker class would need to be updated, even though the change is unrelated to baking bread. This coupling of unrelated responsibilities increases the risk of introducing bugs and makes the codebase more difficult to maintain.

Moreover, having multiple responsibilities in a single class reduces readability and increases complexity. It becomes challenging for developers to understand the purpose of the class and to locate specific functionality within the code. By combining different responsibilities, the class becomes a monolithic block of code that is harder to test and debug. This complexity can lead to longer development times and increased chances of errors. Adhering to the SRP by separating these responsibilities into distinct classes improves code maintainability, readability, and reduces the likelihood of unintended side effects when making changes.

To adhere to the SRP, the bakery could assign different roles to different individuals or teams. For example, there could be a separate person or team responsible for managing the inventory, another for ordering supplies, another for serving customers, and another for cleaning the bakery.

Java

```
1  // Class for baking bread
2  class BreadBaker {
3   public
4    void bakeBread() { System.out.println("Baking high-quality bread.
```

```
}

// Class for managing inventory
class InventoryManager {
 public
  void manageInventory() { System.out.println("Managing inventory..
}

// Class for ordering supplies
class SupplyOrder {
 public
  void orderSupplies() { System.out.println("Ordering supplies...")
}

// Class for serving customers
class CustomerService {
 public
  void serveCustomer() { System.out.println("Serving customers...")
}

// Class for cleaning the bakery
class BakeryCleaner {
 public
  void cleanBakery() { System.out.println("Cleaning the bakery...")
}

public class Bakery {
 public
  static void main(String[] args) {
    BreadBaker baker = new BreadBaker();
    InventoryManager inventoryManager = new InventoryManager();
    SupplyOrder supplyOrder = new SupplyOrder();
    CustomerService customerService = new CustomerService();
    BakeryCleaner cleaner = new BakeryCleaner();
    // Each class focuses on its specific responsibility
    baker.bakeBread();
    inventoryManager.manageInventory();
    supplyOrder.orderSupplies();
    customerService.serveCustomer();
    cleaner.cleanBakery();
  }
}
```

In the above example:
• **Bread Baker Class**:
Responsible solely for baking bread. This class focuses on ensuring the quality and standards of the bread without being burdened by other tasks.

• **Inventory Manager Class**:
Handles inventory management, ensuring that the bakery has the right ingredients and supplies available.

• **Supply Order Class**:
Manages ordering supplies, ensuring that the bakery is stocked with necessary items.

• **Customer Service Class:**
Takes care of serving customers, providing a focused approach to customer interactions.

• **Bakery Cleaner Class:**
Responsible for cleaning the bakery, ensuring a hygienic environment.

By adhering to the SRP, the code becomes more maintainable and easier to understand. Changes in one class do not affect unrelated parts, enhancing readability and reducing complexity. Each class has a clear and focused responsibility, making the system more modular and easier to manage.

🚪 Open/Closed Principle

This principle states that Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification which means you should be able to extend a class behavior, without modifying it.

🌟 Why it matters:

• Prevents breaking existing code.
• Encourages reusable components.

💭 Example:

Suppose we have a Shape class that calculates the area of different shapes. Initially, it supports only circles and rectangles. Adding a new shape, like a triangle, would require modifying the existing code.
The Open/Closed Principle states that software entities should be open for extension but closed for modification. Here since we are modifying the existing code and not extending it, that is why the current approach is problematic:

Java

```
// Incorrect approach
class Shape {
 private
  String type;
 public
  double calculateArea() {
    if (type.equals("circle")) {
      // Circle area calculation
    } else if (type.equals("rectangle")) {
      // Rectangle area calculation
    }
    // Adding a triangle requires modifying this method
  }
}
```

In the above code example, the Shape class has multiple responsibilities: calculating the area for different shapes. This violates the Open/Closed Principle because adding a new shape, like a triangle, requires modifying the existing code. This can lead to potential bugs and makes the code harder to maintain. When a new shape is introduced, the calculateArea method must be updated to handle the new shape, which increases the risk of errors and makes the code less flexible. This approach tightly couples the Shape class to the specific shapes it supports, reducing its extensibility.

Moreover, this design makes it difficult to add new shapes without altering the existing codebase. Each time a new shape is added, the Shape class must be modified, which can introduce unintended side effects and disrupt the functionality of the existing shapes. This lack of modularity and extensibility makes the system harder to maintain and evolve over time. By adhering to the Open/Closed Principle, we can create a more robust and maintainable system where new shapes can be added without modifying the existing code, thus reducing the risk of bugs and improving the overall design.

Instead, we can implement the Open/Closed principle correctly by creating and abstract class / Interface of Shape and all the other classes will extend/inherit the methods of the Shape class efficiently following the Open/Closed Principle.

Java

```
// Better approach following Open/Closed Principle
abstract class Shape {
  abstract double calculateArea();
  // We can also use an interface instead of an abstract class
}

```

```
class Circle extends Shape {
 private
  double radius;
  @Override
public double calculateArea() {
    return Math.PI * radius * radius;
  }
}

class Rectangle extends Shape {
 private
  double width;
 private
  double height;
  @Override
public double calculateArea() {
    return width * height;
  }
}

// Adding a new shape without modifying existing code
class Triangle extends Shape {
 private
  double base;
 private
  double height;
  @Override
public double calculateArea() {
    return 0.5 * base * height;
  }
}
```

📖 Explanation:

In the corrected code example, each shape class has a single responsibility:
• Base Class (Shape): This is an abstract base class with an abstract function calculateArea(). It defines a common interface for all shapes.
• Circle Class: This class implements the calculateArea() logic for circles.
• Rectangle Class: This class implements the calculateArea() logic for rectangles.
• Triangle Class: This class implements the calculateArea() logic for triangles.

By adhering to the Open/Closed Principle, the code becomes more maintainable and easier to understand. Adding a new shape, like a triangle, does not require modifying the existing

code. Instead, we extend the Shape class by creating a new class for the triangle. This approach prevents breaking existing code and encourages the creation of reusable components.

🔄 Liskov Substitution Principle

The principle was introduced by Barbara Liskov in 1987 and according to this principle Derived or child classes must be substitutable for their base or parent classes. This principle ensures that any class that is the child of a parent class should be usable in place of its parent without any unexpected behavior.

🌟 Why it matters:

• Ensures reliability when using polymorphism.
• Avoids unexpected behaviors in subclass implementations.

💭 Example:

Consider a Vehicle class and a Car subclass. If the Vehicle class has a startEngine method, a subclass like Car would work fine but if a subclass Bicycle is created, then it will violate LSP because bicycles do not have engines.

Java

```java
// Problematic approach that violates LSP
class Vehicle {
 public
  void startEngine() {
    // Engine starting logic
  }
}

class Car extends Vehicle {
  @Override public void startEngine() {
    // Car-specific engine starting logic
  }
}

class Bicycle extends Vehicle {
  @Override public void startEngine() {
    // Problem: Bicycles don't have engines!
    throw new UnsupportedOperationException("Bicycles don't have en
  }
}

```

```
public class Main {
 public
  static void main(String[] args) {
    // Creating objects of different subclasses
    Vehicle car = new Car();
    Vehicle bicycle = new Bicycle();
    // Using polymorphism
    System.out.println("Car:");
    car.startEngine();  // Output: Car engine started.
    System.out.println("nBicycle:");
    try {
      bicycle.startEngine();  // Throws UnsupportedOperationExcepti
    } catch (UnsupportedOperationException e) {
      System.out.println("Error: " + e.getMessage());
    }
  }
}
```

In this problematic approach, the Vehicle class has a startEngine() method, which is appropriate for Car but not for Bicycle. This violates the Liskov Substitution Principle (LSP) because the Bicycle class cannot be substituted / used in place of the Vehicle class without causing unexpected behavior (i.e., throwing an exception). When a subclass cannot fulfill the contract of its parent class, it leads to a breakdown in polymorphism, making the code less reliable and predictable. The Bicycle class, when forced to implement the startEngine() method, must either provide a meaningless implementation or throw an exception, both of which are undesirable outcomes.

This design flaw also makes the code less flexible and harder to extend. If new vehicle types are added that do not have engines, they too would be forced to implement the startEngine() method, leading to further violations of the LSP. This tight coupling between the Vehicle class and its subclasses reduces the modularity and reusability of the code. By adhering to the LSP, we can create a more robust and maintainable system where subclasses can be used interchangeably with their parent classes without causing unexpected behavior, ensuring that the code remains flexible and easy to extend.

To properly implement the Liskov Substitution Principle, we can restructure the vehicle hierarchy through a more refined abstraction. The base Vehicle class serves as the foundation, with two specialized abstract classes: EngineVehicle and NonEngineVehicle. This segregation creates a clear distinction between motorized and non-motorized transport.

Java

```
// Better approach following LSP
abstract class Vehicle {
```

```
  // Common vehicle behaviors
 public
  void move() {
    // Movement logic
  }
}

abstract class EngineVehicle extends Vehicle {
 public
  void startEngine() {
    // Engine starting logic
  }
}

abstract class NonEngineVehicle extends Vehicle {
  // No engine-related methods
}

class Car extends EngineVehicle {
  @Override public void startEngine() {
    // Car-specific engine starting logic
  }
}

class Bicycle extends NonEngineVehicle {
  // Bicycle-specific methods
  // No need to implement engine-related methods
}

public class Main {
 public
  static void main(String[] args) {
    // Using EngineVehicle
    EngineVehicle car = new Car();
    car.startEngine();  // Output: Car-specific engine starting log
    car.move();         // Output: Movement logic


    // Using NonEngineVehicle
    NonEngineVehicle bicycle = new Bicycle();
    bicycle.move();  // Output: Movement logic
  }
}
```

📖 Explanation:

In the corrected code example, the vehicle hierarchy is refined to distinguish between motorized and non-motorized vehicles:
• **Base Class (Vehicle):** This is an abstract base class with common vehicle behaviors.
• **EngineVehicle Class:** This abstract class extends Vehicle and includes engine-related methods.
• **NonEngineVehicle Class:** This abstract class extends Vehicle and excludes engine-related methods.
• **Car Class:** This class extends EngineVehicle and implements the startEngine method.
• **Bicycle Class:** This class extends NonEngineVehicle and does not need to implement engine-related methods.

💥 This design ensures that:

1. Each subtype fully satisfies the behavioral contract of its parent type
2. Client code can interact with either vehicle type without unexpected behavior

The inheritance hierarchy accurately models the real-world domain

🎛️ Interface Segregation Principle

This principle is the first principle that applies to Interfaces instead of classes in SOLID and it is similar to the Single Responsibility principle. It states that do not force any client to implement an interface which is irrelevant to them. Here your main goal is to focus on avoiding fat interface and give preference to many small client-specific interfaces. You should prefer many client interfaces rather than one general interface and each interface should have a specific responsibility.

🌟 Why it matters:

• Reduces unnecessary dependencies.
• Simplifies implementation for specific use cases.

💭 Example :

Consider a software system modeling various office equipment through a Machine interface that encompasses multiple functionalities: printing, scanning, and faxing. This design presents a violation of the Interface Segregation Principle when implementing a basic printer device.

The fundamental issue arises when a BasicPrinter class, which is designed solely for printing operations, must implement the complete Machine interface. This forces the class to provide implementations for scan() and fax() methods, despite these capabilities being outside its core functionality.

Java

```java
// Problematic approach that violates ISP
interface Machine {
  void print();
  void scan();
  void fax();
}

class AllInOnePrinter implements Machine {
  @Override public void print() {
    // Printing functionality
  }
  @Override public void scan() {
    // Scanning functionality
  }
  @Override public void fax() {
    // Fax functionality
  }
}

class BasicPrinter implements Machine {
  @Override public void print() {
    // Printing functionality
  }
  @Override public void scan() {
    // Problem: Basic printer can't scan!
    throw new UnsupportedOperationException("Cannot scan");
  }
  @Override public void fax() {
    // Problem: Basic printer can't fax!
    throw new UnsupportedOperationException("Cannot fax");
  }
}
```

In the above example, the BasicPrinter class is forced to implement methods for scanning and faxing, which it cannot support. This violates the Interface Segregation Principle because the class is burdened with irrelevant methods, leading to unnecessary complexity and potential runtime errors. When a class is required to implement methods that it does not need, it results in a bloated interface, making the class more difficult to understand and maintain. Additionally, the presence of unsupported methods can lead to runtime exceptions,

as seen with the UnsupportedOperationException thrown by the scan() and fax() methods in the BasicPrinter class.

This design flaw also makes the code less flexible and harder to extend. If new functionalities are added to the Machine interface, all implementing classes, including those that do not require the new functionalities, must be updated. This creates unnecessary dependencies and increases the risk of introducing bugs. By forcing classes to implement irrelevant methods, the code becomes tightly coupled, reducing its modularity and reusability. Adhering to the Interface Segregation Principle by creating smaller, more focused interfaces ensures that classes only implement the methods they need, leading to a more robust and maintainable system.

To adhere to the Interface Segregation Principle, we can split the interface into smaller, more focused interfaces:

Java

```java
// Better approach following ISP
interface Printer {
  void print();
}

interface Scanner {
  void scan();
}

interface FaxMachine {
  void fax();
}

class BasicPrinter implements Printer {
  @Override
  public void print() {
    // Printing functionality
  }
}

class AllInOnePrinter implements Printer, Scanner, FaxMachine {
  @Override
  public void print() {
    // Printing functionality
  }
  @Override
  public void scan() {
    // Scanning functionality
```

```
29    }
30    @Override
31    public void fax() {
32      // Fax functionality
33    }
34  }
```

Explanation:

In the corrected code example, the interfaces are segregated into smaller, more focused interfaces:
• **Printer Interface:** Defines the print() method.
• **Scanner Interface:** Defines the scan() method.
• **FaxMachine Interface:** Defines the fax() method.

By segregating the interfaces, we ensure that classes only implement methods that are relevant to their functionality. The BasicPrinter class now only implements the Printer interface, avoiding unnecessary methods. The AllInOnePrinter class implements all three interfaces, providing the full range of functionalities.

💥This design ensures that:

• Allows classes to implement only the interfaces they need: Classes are not burdened with irrelevant methods.
• Prevents classes from being forced to provide dummy implementations: Avoids unnecessary complexity and potential runtime errors.
• Makes the system more flexible and maintainable: Smaller, focused interfaces lead to a more modular and maintainable codebase.
• Follows the principle of 'interface cohesion': Each interface has a specific responsibility, leading to better cohesion and separation of concerns.

By segregating interfaces, we ensure that classes only implement methods that are relevant to their functionality, leading to a more robust and maintainable system.

🔌Dependency Inversion Principle

The Dependency Inversion Principle (DIP) is a principle in object-oriented design that states that High-level modules should not depend on low-level modules. Both should depend on abstractions. Additionally, abstractions should not depend on details. Details should depend on abstractions.

In simpler terms, the DIP suggests that classes should rely on abstractions (e.g., interfaces or abstract classes) rather than concrete implementations. This allows for more flexible and decoupled code, making it easier to change implementations without affecting other parts of the codebase.

🌟 Why it matters:

• Promotes decoupled architecture.
• Facilitates testing and maintainability.

💭 Example :

Consider an enterprise e-commerce system where order processing requires various types of notifications to be sent to customers, administrators, and inventory systems

Java

```java
// Problematic approach that violates DIP
class EmailNotifier {
  public void sendEmail(String message) {
    // Configure SMTP
    // Set up email templates
    // Send email implementation
  }
}

class OrderService {
  private EmailNotifier emailNotifier;
  private DatabaseLogger logger;
  private InventorySystem inventory;
  public OrderService() {
    // Direct dependencies on concrete implementations
    this.emailNotifier = new EmailNotifier();
    this.logger = new DatabaseLogger();
    this.inventory = new InventorySystem();
  }
  public void placeOrder(Order order) {
    // Process order
    inventory.updateStock(order);
    emailNotifier.sendEmail("Order #" + order.getId() + " placed su
    logger.logTransaction("Order placed: " + order.getId());
  }
}
```

📖 Explanation :

The OrderService class has direct dependencies on concrete implementations like EmailNotifier, DatabaseLogger, and InventorySystem. This design creates several

challenges:

1. Tight coupling to specific implementations: The OrderService class is tightly coupled to the specific implementations, making it difficult to change or replace them.

2. Difficulty in unit testing due to concrete dependencies: Testing the OrderService class becomes challenging because it relies on concrete implementations.

3. Inflexibility when business requirements change: Any change in the notification method or logging mechanism requires modifying the OrderService class.

4. Challenges in maintaining and modifying the system: The code becomes harder to maintain and modify due to the tight coupling.

5. Difficulty in implementing different notification strategies for different markets or customer segments: The system lacks flexibility to accommodate different notification methods.

Here's the improved design following DIP:
To adhere to the Dependency Inversion Principle, we can introduce abstractions and use dependency injection to decouple the OrderService class from specific implementations.

Java

```java
// Better approach following DIP
interface NotificationService {
  void sendNotification(String message);
}

interface LoggingService {
  void logMessage(String message);
  void logError(String error);
}

interface InventoryService {
  void updateStock(Order order);
  boolean checkAvailability(Product product);
}

class EmailNotifier implements NotificationService {
  @Override
  public void sendNotification(String message) {
    // Email specific implementation
  }
}

```

```java
class SMSNotifier implements NotificationService {
  @Override
  public void sendNotification(String message) {
    // SMS specific implementation
  }
}

class PushNotifier implements NotificationService {
  @Override
  public void sendNotification(String message) {
    // Push notification specific implementation
  }
}

class DatabaseLogger implements LoggingService {
  @Override
  public void logMessage(String message) {
    // Database logging implementation
  }
  @Override
  public void logError(String error) {
    // Error logging implementation
  }
}

class OrderService {
  private final NotificationService notificationService;
  private final LoggingService loggingService;
  private final InventoryService inventoryService;
  // Constructor injection of dependencies
  public OrderService(NotificationService notificationService,
      LoggingService loggingService, InventoryService inventoryServ
    this.notificationService = notificationService;
    this.loggingService = loggingService;
    this.inventoryService = inventoryService;
  }

  public void placeOrder(Order order) {
    try {
      // Check inventory
      if (inventoryService.checkAvailability(order.getProduct())) {
        // Process order
        inventoryService.updateStock(order);
        // Send notification
```

```
        notificationService.sendNotification(
            "Order #" + order.getId() + " placed successfully");
        // Log success
        loggingService.logMessage(
            "Order processed successfully: " + order.getId());
      }
    } catch (Exception e) {
      loggingService.logError(
          "Error processing order: " + order.getId() + " - " + e.ge
      throw e;
    }
  }
}

// Usage with dependency injection
NotificationService emailNotifier = new EmailNotifier();
LoggingService logger = new DatabaseLogger();
InventoryService inventory = new WarehouseInventoryService();
OrderService orderService = new OrderService(emailNotifier, logger,
```

📖 Explanation:

In the improved design, we introduce interfaces for the notification, logging, and inventory services:
1. NotificationService Interface: Defines the sendNotification() method.
2. LoggingService Interface: Defines the logMessage() and logError() methods.
3. InventoryService Interface: Defines the updateStock() and checkAvailability() methods.

By using these interfaces, the OrderService class depends on abstractions rather than concrete implementations.

💥 This design offers several benefits:

1. **Flexibility:** The system can easily accommodate new notification methods, logging mechanisms, or inventory systems without modifying existing code.

2. **Testability:** Dependencies can be easily mocked for unit testing.

3. **Maintainability:** Each component has a single responsibility and can be modified independently.

4. **Scalability:** New implementations can be added without affecting existing code.

5. **Configuration Flexibility:** Different implementations can be injected based on runtime conditions or configuration.