Solid Principles

# Single Responsibility Principle

The Single Responsibility Principle states that every class should have one and only one reason to change. In other words, a class should only handle one responsibility. This is to keep classes focused on a single part of the functionality, making them easier to maintain and less prone to changes due to unrelated reasons.

1. **Why SRP is Subjective**: SRP can be subjective because what constitutes a "responsibility" might vary depending on interpretation. For example, you might think an AuthManager that handles both Login and Logout as handling multiple responsibilities. But if we consider “authentication” as a single responsibility, then these tasks are logically grouped.
2. **One Reason to Change**: If a class has more than one reason to change, it likely violates SRP. For example, an Employee class that calculates salary, produces hours reports, and saves data would need to change for multiple reasons (e.g., if report formatting changes or the way salaries are calculated changes).

### **Example Scenario:**

Let’s say we start with an Employee class that has multiple responsibilities:

public class Employee

{

public string Name { get; set; }

public decimal CalculateMonthlySalary() { /\* calculate salary logic \*/ }

public string ProduceMonthlyHoursReport() { /\* report generation logic \*/ }

public void Save() { /\* data persistence logic \*/ }

}

Here, Employee:

* Calculates salary (finance department changes),
* Generates reports (HR department changes),
* Saves data (technology-related changes).

This class violates SRP because any one of these responsibilities could require changes for different reasons.

### **Solution Using SRP:**

To follow SRP, we break down the responsibilities into separate classes:

public class Employee

{

public string Name { get; set; }

}

public class PaymentService

{

public decimal CalculateMonthlySalary(Employee employee) { /\* salary calculation logic \*/ }

}

public class HoursReportService

{

public string ProduceMonthlyHoursReport(Employee employee) { /\* report generation logic \*/ }

}

public class EmployeeRepository

{

public void Save(Employee employee) { /\* data persistence logic \*/ }

}

* **Employee**: This class now only holds basic employee information.
* **PaymentService**: Responsible solely for salary calculation, so any finance-related changes apply here.
* **HoursReportService**: Handles reporting; any format or content changes for hours reporting can be applied to this class.
* **EmployeeRepository**: Responsible for saving employee data, ensuring changes related to database technology only affect this class.

### **Key Benefits of SRP:**

* **Isolation of Changes**: Each class has one reason to change, making modifications easier and more targeted.
* **Improved Testing**: Smaller classes with single responsibilities are easier to test individually.
* **Better Reusability**: Each class focuses on one task, making it more reusable in other parts of the application.

Following SRP creates a codebase that is modular, maintainable, and adaptable to changes in individual components without affecting the entire system.

# **Open-Closed Principle**

The **Open-Closed Principle (OCP)** asserts that software entities should be *open for extension but closed for modification*. This means that the behavior of a class or module should be extendable (e.g., by adding new functionality or types) without altering its existing source code.

A module is said to be "closed" if it is available for use by other modules

\* It can be stored as a lib for usage....

\* So, no implementation changes should be possible once its released to clients

### **Breaking Down "Open for Extension, Closed for Modification"**

1. **Open for Extension**:
   * This can mean adding new subclasses, using polymorphism (one abstraction with multiple concrete implementations), or adding additional code that doesn’t interfere with existing code.
2. **Closed for Modification**:
   * This doesn’t mean you cannot change the code for bug fixes or refactoring, but once a module is released to clients or integrated as a library, it shouldn’t require changes that impact its core functionality.
   * When modifications are needed (such as adding new employee types in a tax calculator), OCP suggests we add new classes rather than modifying existing ones.

### **Example of OCP Violation**

Consider a SalaryCalculator class that calculates tax based on different employee types. Each time a new employee type is introduced, the SalaryCalculator class must be updated to accommodate the new tax logic, which leads to repeated modification and increased complexity.

public class SalaryCalculator

{

public double CalculateTax(Employee emp)

{

if (emp.Type == EmployeeType.FullTime)

{

// Full-time tax calculation

}

else if (emp.Type == EmployeeType.Freelance)

{

// Freelance tax calculation

}

// New employee types will require more conditions here, violating OCP

}

}

This code violates OCP because the SalaryCalculator class has to be modified every time a new employee type is added.

### **Applying OCP Using Abstraction**

We can introduce an interface, ITaxCalculator, for different tax calculation strategies and create separate classes to handle tax logic for each employee type. This way, new employee types can be added without modifying the original SalaryCalculator class.

// Interface for tax calculators

public interface ITaxCalculator

{

double CalculateTax(Employee emp);

}

// Tax calculator for full-time employees

public class FullTimeTaxCalculator : ITaxCalculator

{

public double CalculateTax(Employee emp)

{

// Full-time employee tax logic

}

}

// Tax calculator for freelance employees

public class FreelanceTaxCalculator : ITaxCalculator

{

public double CalculateTax(Employee emp)

{

// Freelance employee tax logic

}

}

// Factory to get the appropriate tax calculator based on employee type

public class TaxCalculatorFactory

{

public static ITaxCalculator GetTaxCalculator(EmployeeType empType)

{

return empType switch

{

EmployeeType.FullTime => new FullTimeTaxCalculator(),

EmployeeType.Freelance => new FreelanceTaxCalculator(),

// Add more employee types without modifying existing code

\_ => throw new ArgumentOutOfRangeException(nameof(empType), "Unknown employee type")

};

}

}

// SalaryCalculator uses the factory to get the correct tax calculator

public class SalaryCalculator

{

public double CalculateTax(Employee emp)

{

ITaxCalculator taxCalculator = TaxCalculatorFactory.GetTaxCalculator(emp.Type);

return taxCalculator.CalculateTax(emp);

}

}

### **Benefits of this OCP-Compliant Design**

1. **Easily Extendable**: Adding a new employee type (e.g., Contractor) requires only a new implementation of ITaxCalculator and an update to TaxCalculatorFactory.
2. **Reduces Risk**: Core classes like SalaryCalculator remain stable, minimizing the risk of introducing bugs.
3. **Single Responsibility**: Each tax calculation class has a single responsibility, allowing for focused and clear logic per employee type.
4. **Improved Maintainability**: By separating tax logic, changes are isolated, making the system easier to understand, maintain, and extend.

In summary, OCP helps future-proof your code, ensuring that any additions or changes have a minimal impact on the existing codebase, allowing for easier extensibility and maintainability.

# Liskov Substitution Principle (LSP)

The Liskov Substitution Principle (LSP) is one of the SOLID principles of object-oriented design. It states that if SSS is a subtype of TTT, then objects of type TTT should be replaceable with objects of type SSS without altering any desirable properties of the program (e.g., correctness, tasks performed).

#### **Key Concepts**

* **Substitutability**: You should be able to replace a parent class with a child class without affecting the program’s correctness.
* **Behavior Consistency**: The subclass should behave in a way that is expected by users of the superclass.

LSP Rules to be followed for "proper" subtyping to achieve LSP

\* Method Signature RULE -> Contravariance of args

\* Covariance of result RULE

### **Rectangle-Square Problem in Liskov Substitution Principle (LSP)**

Consider the classes Rectangle and Square. The Rectangle class has properties width and height, which can be independently set. The Square class, however, has equal width and height, so setting one dimension affects the other. Here’s how this works and why it’s problematic:

class Rectangle

{

protected int width;

protected int height;

public virtual void SetWidth(int width) => this.width = width;

public virtual void SetHeight(int height) => this.height = height;

public int GetWidth() => width;

public int GetHeight() => height;

public int Area() => width \* height;

}

class Square : Rectangle

{

public override void SetWidth(int width)

{

this.width = width;

this.height = width; // Ensures width == height

}

public override void SetHeight(int height)

{

this.height = height;

this.width = height; // Ensures width == height

}

}

### **Problem: Inconsistency in Behavior**

* **Incompatible Behavior**: A square cannot maintain the behavior of a rectangle. While a rectangle allows independent setting of width and height, a square forces them to be equal.
* **Unexpected Results**: If you replace a Rectangle with a Square, unexpected behavior can occur in situations where dimensions need to be set independently.

### **Example of Drawing Application**

Imagine a drawing application that expects rectangles, and you want to adjust the width and height independently to shape your design.

#### **Rectangle Class: Independent Dimensions**

A Rectangle in this application has independent dimensions:

Rectangle rect = new Rectangle();

rect.SetWidth(5);

rect.SetHeight(10);

// Width and height are set independently.

In this case, the rectangle now has a width of 5 and a height of 10, resulting in a rectangular shape.

#### 

#### **Square Class: Dependent Dimensions**

Now, suppose we try to use a Square as a substitute for a Rectangle in the drawing program. Since a square has equal width and height, whenever you set one dimension, the other must change to match.

Square sq = new Square();

sq.SetWidth(5); // Sets both width and height to 5 because it's a square.

sq.SetHeight(10); // Also sets both width and height to 10.

Even though you intended only to change the height, Square forces both dimensions to be equal. This prevents you from creating rectangular shapes because every change affects both dimensions. In a drawing application, this behavior is limiting and breaks the user’s expectations of how a rectangle should work.

### **Why This Violates Liskov Substitution**

If your application expects to use rectangles (with independent width and height), substituting a Square causes incorrect behavior. The program can no longer handle changes to width and height separately, as it expects from a Rectangle. Thus, a Square cannot fully replace a Rectangle without disrupting the expected functionality, violating the Liskov Substitution Principle.

### **Conclusion**

Simply subclassing Square from Rectangle does not imply a true subtype relation because Square fails to fully satisfy the behavior contract of Rectangle. Instead, it is a misuse of inheritance.

### 

### 

### 

### 

### 

### **Fixing the Rectangle-Square Example**

#### **Invariant Rule**

* **Invariant Rule for Rectangle**: A rectangle maintains the rule that the methods SetWidth and SetHeight must be independent. This means you can set the width and height separately without affecting each other.
* **Mutual Exclusivity for Square**: For a square to be a proper subtype of a rectangle, we need to change how we think about setting dimensions.

#### **Composition Without Inheritance**

Instead of using inheritance, you could redesign the Square class to contain a Rectangle object, leveraging composition:

namespace N3

{

public class Rectangle

{

protected int width;

protected int height;

public virtual void SetWidth(int width) => this.width = width;

public virtual void SetHeight(int height) => this.height = height;

public int Area() => width \* height; // Calculates area

}

// Square class using composition

public class Square

{

private Rectangle rectangle;

public Square(int sideLength)

{

rectangle = new Rectangle();

rectangle.SetWidth(sideLength);

rectangle.SetHeight(sideLength);

}

public void SetSideLength(int length)

{

rectangle.SetWidth(length);

rectangle.SetHeight(length);

}

public int Area() => rectangle.Area(); // Delegates to rectangle's area calculation

}

public class Startup

{

public static void Main(string[] args)

{

Square sq = new Square(5);

Console.WriteLine($"Square area: {sq.Area()}"); // Outputs: 25

sq.SetSideLength(10); // Changes both width and height to 10

Console.WriteLine($"Square area after changing side length: {sq.Area()}"); // Outputs: 100

}

}

}

### **Characteristics**

* **Composition**: Square does not inherit from Rectangle but contains an instance of Rectangle. This means it is a separate entity, maintaining its identity.
* **Mutual Exclusivity Maintained**: The Square class has a method SetSideLength, which ensures that both dimensions are set to the same value, but does so in a way that doesn’t disrupt the expected behavior of Rectangle.
* **Flexibility and Clarity**: Using composition allows the Square to fulfill its purpose without violating the principles of inheritance, leading to clearer code that adheres to LSP and respects invariants.

### 

### 

### **Summary of Differences**

* **Inheritance vs. Composition**: The first code uses inheritance, while the second uses composition.
* **Behavior**: The first code can lead to violations of LSP due to mutual dependency on width and height, whereas the second adheres to the principles of independent dimensions.
* **Design Clarity**: The second approach results in clearer and more maintainable code by ensuring that each class has a defined responsibility without inappropriate relationships.

Using composition rather than inheritance in this scenario provides a cleaner and more robust design that avoids the pitfalls of the rectangle-square relationship, aligning with solid design principles.

### **Final Summary**

* **Invariant Rule**: Maintain independent dimensions for rectangles and enforce mutual exclusivity for squares.
* **Composition**: Instead of subclassing, use composition to create a Square that contains a Rectangle. This way, Square can maintain its specific behavior without violating the principles of LSP and without creating an inappropriate subtype relationship.

This design leads to clearer and more maintainable code while respecting the relationships between rectangles and squares.

# Interface Segregation Principle (ISP)

The Interface Segregation Principle (ISP) states that "clients should not be forced to depend on methods they do not use." In simpler terms, it means **segregating larger interfaces into smaller, more specific ones** so that clients only implement the functionality they need.

#### **Why Follow ISP?**

* **Avoid Unused Methods**: Large interfaces with many methods may require classes to implement methods that are irrelevant to their role.
* **Maintainability**: Small, focused interfaces make code easier to maintain and update.
* **Flexibility**: Clients using smaller interfaces can change their implementations with minimal impact on the codebase.

### **Example**

Imagine an interface representing actions of a multi-functional printer. Not all devices will support every function (e.g., some may only print, others may print and scan, etc.).

#### **Violating ISP: Single Large Interface**

public interface IMultiFunctionDevice

{

void Print(Document document);

void Scan(Document document);

void Fax(Document document);

}

public class Printer : IMultiFunctionDevice

{

public void Print(Document document) => Console.WriteLine("Printing document...");

public void Scan(Document document)

{

throw new NotImplementedException("Printer cannot scan");

}

public void Fax(Document document)

{

throw new NotImplementedException("Printer cannot fax");

}

}

Here, **Printer** is forced to implement Scan and Fax methods, even though it does not support these functionalities. This violates ISP as the Printer class depends on methods it does not use.

#### **Following ISP: Segregated Interfaces**

Instead, segregate the functionality into multiple smaller interfaces:

public interface IPrinter

{

void Print(Document document);

}

public interface IScanner

{

void Scan(Document document);

}

public interface IFax

{

void Fax(Document document);

}

public class Printer : IPrinter

{

public void Print(Document document) => Console.WriteLine("Printing document...");

}

public class Scanner : IScanner

{

public void Scan(Document document) => Console.WriteLine("Scanning document...");

}

public class MultiFunctionPrinter : IPrinter, IScanner, IFax

{

public void Print(Document document) => Console.WriteLine("Printing document...");

public void Scan(Document document) => Console.WriteLine("Scanning document...");

public void Fax(Document document) => Console.WriteLine("Faxing document...");

}

### **Explanation**

* **Printer** implements only IPrinter, as it can only print.
* **Scanner** implements only IScanner, as it can only scan.
* **MultiFunctionPrinter** implements all three interfaces, as it can print, scan, and fax.

### **Summary**

The Interface Segregation Principle ensures that:

* Each class only depends on methods it needs.
* Interfaces are specialized, clear, and maintainable.

This leads to more flexible, adaptable code that respects the needs of individual clients without imposing unnecessary requirements.

# Dependency Inversion Principle (DIP)

The Dependency Inversion Principle (DIP) states:

1. **High-level modules should not depend on low-level modules**; both should depend on abstractions.
2. **Abstractions should not depend on details**, and details should depend on abstractions.

#### **Why DIP?**

* **Reduce Coupling**: High-level modules (core functionalities) are shielded from changes in low-level modules (implementation details).
* **Enhance Flexibility**: Code becomes more adaptable, as changes in low-level details do not affect higher-level functionalities.

DIP **inverts the dependency** relationship by making low-level modules depend on high-level abstractions, rather than directly implementing them.

#### **Achieving DIP with Dependency Injection**

Dependency Injection is commonly used to inject dependencies (low-level modules) into high-level modules, ensuring they rely on abstractions rather than specific implementations.

#### **Example**

Suppose we have a ReportGenerator (high-level module) that generates reports and depends on a FileStorage class (low-level module) to save them. Without DIP, ReportGenerator directly depends on FileStorage, leading to tightly coupled code.

#### **Without DIP: Tightly Coupled Code**

public class FileStorage

{

public void Save(string data) => Console.WriteLine("Saving data to file storage...");

}

public class ReportGenerator

{

private FileStorage \_fileStorage = new FileStorage();

public void GenerateReport(string data)

{

// Generate report logic

\_fileStorage.Save(data); // Direct dependency on FileStorage

}

}

Here, ReportGenerator is directly tied to FileStorage, making it hard to change the storage mechanism.

#### **With DIP: Depend on Abstractions**

By introducing an abstraction (IStorage interface), both ReportGenerator and FileStorage now depend on IStorage. This design adheres to DIP.

public interface IStorage

{

void Save(string data);

}

public class FileStorage : IStorage

{

public void Save(string data) => Console.WriteLine("Saving data to file storage...");

}

public class DatabaseStorage : IStorage

{

public void Save(string data) => Console.WriteLine("Saving data to database...");

}

public class ReportGenerator

{

private readonly IStorage \_storage;

public ReportGenerator(IStorage storage) => \_storage = storage;

public void GenerateReport(string data)

{

// Generate report logic

\_storage.Save(data); // Uses abstraction, not specific storage

}

}

Now, ReportGenerator only relies on IStorage and can work with any storage mechanism that implements IStorage. This flexibility lets us inject FileStorage, DatabaseStorage, or any other storage class without changing ReportGenerator.

#### **Benefits of DIP**

1. **Independence**: High-level modules can proceed independently of low-level module details.
2. **Testability**: Mock or fake implementations can be injected for testing, making the code easier to test.
3. **Scalability**: New features can be added by implementing additional abstractions without modifying existing code.

Consider an **online shopping application** where customers can place orders, and the application sends notifications through various channels (e.g., email, SMS, or push notifications).

#### **Problem: Direct Dependency on Notification Service**

The order processing module (high-level module) depends on specific notification services (low-level modules) to send notifications. If we directly use these services, the code becomes tightly coupled, making it challenging to switch or add new notification channels.

### **Without DIP: Tightly Coupled Code**

Here, the OrderProcessor class directly depends on EmailNotification to send notifications.

public class EmailNotification

{

public void Send(string message) => Console.WriteLine("Sending email notification: " + message);

}

public class OrderProcessor

{

private readonly EmailNotification \_notification = new EmailNotification();

public void ProcessOrder(string orderDetails)

{

Console.WriteLine("Processing order...");

\_notification.Send("Order processed: " + orderDetails); // Direct dependency on EmailNotification

}

}

In this setup, if we need to change the notification service (e.g., to SMS), we must modify the OrderProcessor class itself, breaking the **Open-Closed Principle** and making the code harder to maintain.

### **With DIP: Using Abstractions**

Let’s apply DIP by creating a common interface INotification for notifications and have OrderProcessor depend on INotification instead of a specific notification service.

#### **Step 1: Define an Abstraction**

Create an interface INotification that defines a Send method.

public interface INotification

{

void Send(string message);

}

#### **Step 2: Implement the Interface for Different Notification Services**

Now, we implement INotification for each specific notification service.

public class EmailNotification : INotification

{

public void Send(string message) => Console.WriteLine("Sending email notification: " + message);

}

public class SmsNotification : INotification

{

public void Send(string message) => Console.WriteLine("Sending SMS notification: " + message);

}

public class PushNotification : INotification

{

public void Send(string message) => Console.WriteLine("Sending push notification: " + message);

}

#### 

#### **Step 3: Modify the High-Level Module to Use Abstractions**

Now, OrderProcessor depends on the INotification abstraction instead of a specific implementation.

public class OrderProcessor

{

private readonly INotification \_notification;

public OrderProcessor(INotification notification)

{

\_notification = notification;

}

public void ProcessOrder(string orderDetails)

{

Console.WriteLine("Processing order...");

\_notification.Send("Order processed: " + orderDetails);

}

}

#### **Step 4: Inject the Dependency**

At runtime, we can now inject any notification service we want, achieving flexibility and scalability.

public class Startup

{

public static void Main(string[] args)

{

// Dependency Injection: Injecting an Email Notification

INotification emailNotification = new EmailNotification();

OrderProcessor orderProcessor = new OrderProcessor(emailNotification);

orderProcessor.ProcessOrder("Order #123");

// Later, we can switch to SMS without changing OrderProcessor

INotification smsNotification = new SmsNotification();

orderProcessor = new OrderProcessor(smsNotification);

orderProcessor.ProcessOrder("Order #124");

}

}

### **Benefits of DIP in the Real-Life Scenario**

1. **Flexibility**: Switching from email to SMS or push notifications is straightforward without altering OrderProcessor code.
2. **Testability**: We can easily mock INotification in tests, isolating OrderProcessor from specific notification implementations.
3. **Future-Proof**: Adding new notification channels (like a new messaging platform) requires only a new class implementing INotification—no need to alter OrderProcessor, aligning with the Open-Closed Principle.

# Restaurant Order System

Let's build a **Restaurant Ordering System** example, which incorporates all the SOLID principles. In this system, a customer can order food, and we need to manage different types of orders, calculate bills based on the order type, and send notifications.

### **Key Requirements:**

1. **Single Responsibility Principle (SRP)**: Separate responsibilities across classes (e.g., order processing, billing, notification).
2. **Open-Closed Principle (OCP)**: Allow adding new order types without changing the core order processing logic.
3. **Liskov Substitution Principle (LSP)**: Ensure that new order types can replace the existing ones without altering the program's behavior.
4. **Interface Segregation Principle (ISP)**: Use small, focused interfaces that meet the specific needs of each client.
5. **Dependency Inversion Principle (DIP)**: The high-level module (order processing) should depend on abstractions, not specific low-level implementations.

Here’s a detailed code example that meets each principle.

### **Step 1: Define Abstractions for Different Responsibilities**

using System;

// 1. Define abstractions for Order and Billing

public interface IOrder

{

void PlaceOrder(); // Handles order placing

}

public interface IBilling

{

decimal CalculateTotal(); // Calculates the total bill

}

// 2. Define abstractions for notifications

public interface INotification

{

void Send(string message);

}

### **Step 2: Implement Various Order Types Using SRP, LSP, and OCP**

// Implement order types for DineIn and TakeAway

public class DineInOrder : IOrder

{

public void PlaceOrder()

{

Console.WriteLine("Placing a Dine-In order.");

// Specific logic for dine-in orders

}

}

public class TakeAwayOrder : IOrder

{

public void PlaceOrder()

{

Console.WriteLine("Placing a Take-Away order.");

// Specific logic for take-away orders

}

}

These classes follow **SRP** by focusing only on placing orders and **LSP** because DineInOrder and TakeAwayOrder can replace IOrder in any context.

### **Step 3: Implement Billing Calculations Using OCP**

public class DineInBilling : IBilling

{

public decimal CalculateTotal()

{

decimal total = 50.00m; // Example logic

Console.WriteLine("Calculating total for Dine-In order.");

return total;

}

}

public class TakeAwayBilling : IBilling

{

public decimal CalculateTotal()

{

decimal total = 45.00m; // Example logic with take-away discount

Console.WriteLine("Calculating total for Take-Away order.");

return total;

}

}

With **OCP** in mind, adding a new billing type requires only a new IBilling implementation.

### **Step 4: Implement Notifications Following ISP**

public class EmailNotification : INotification

{

public void Send(string message)

{

Console.WriteLine("Sending email: " + message);

}

}

public class SmsNotification : INotification

{

public void Send(string message)

{

Console.WriteLine("Sending SMS: " + message);

}

}

Here, **ISP** is achieved because each notification class has only one specific method, Send, which it needs.

### **Step 5: Implement Order Processing Using DIP and Dependency Injection**

public class OrderProcessor

{

private readonly IOrder \_order;

private readonly IBilling \_billing;

private readonly INotification \_notification;

// Dependencies are injected, following DIP

public OrderProcessor(IOrder order, IBilling billing, INotification notification)

{

\_order = order;

\_billing = billing;

\_notification = notification;

}

public void ProcessOrder()

{

\_order.PlaceOrder();

decimal total = \_billing.CalculateTotal();

\_notification.Send($"Order processed with total: ${total}");

}

}

The OrderProcessor depends on abstractions (IOrder, IBilling, and INotification), following the **DIP**.

### 

### 

### 

### 

### 

### **Step 6: Use Dependency Injection to Assemble and Execute**

public class Program

{

public static void Main()

{

// Dine-In order with Email Notification

IOrder dineInOrder = new DineInOrder();

IBilling dineInBilling = new DineInBilling();

INotification emailNotification = new EmailNotification();

OrderProcessor dineInOrderProcessor = new OrderProcessor(dineInOrder, dineInBilling, emailNotification);

dineInOrderProcessor.ProcessOrder(); // Output for Dine-In order

Console.WriteLine("\n-----\n");

// Take-Away order with SMS Notification

IOrder takeAwayOrder = new TakeAwayOrder();

IBilling takeAwayBilling = new TakeAwayBilling();

INotification smsNotification = new SmsNotification();

OrderProcessor takeAwayOrderProcessor = new OrderProcessor(takeAwayOrder, takeAwayBilling, smsNotification);

takeAwayOrderProcessor.ProcessOrder(); // Output for Take-Away order

}

}

### 

### 

### 

### **Expected Output**

Placing a Dine-In order.

Calculating total for Dine-In order.

Sending email: Order processed with total: $50.00

-----

Placing a Take-Away order.

Calculating total for Take-Away order.

Sending SMS: Order processed with total: $45.00

### **How SOLID Principles Are Applied**

1. **Single Responsibility Principle (SRP)**:
   * Each class has a single responsibility (placing orders, billing, notifications).
2. **Open-Closed Principle (OCP)**:
   * New order types or billing strategies can be added without modifying existing classes.
3. **Liskov Substitution Principle (LSP)**:
   * All IOrder implementations (e.g., DineInOrder, TakeAwayOrder) can replace each other without altering OrderProcessor's behavior.
4. **Interface Segregation Principle (ISP)**:
   * Each interface (IOrder, IBilling, INotification) is tailored to specific client needs, preventing clients from depending on unnecessary methods.
5. **Dependency Inversion Principle (DIP)**:
   * OrderProcessor depends on abstractions (IOrder, IBilling, INotification), not specific implementations, allowing easy substitution through dependency injection.

This design aligns with SOLID principles, enabling flexibility, testability, and maintainability.