SOLID Principles গুলো Software Design-এর জন্য গুরুত্বপূর্ণ গাইডলাইন, যা Clean, Maintainable এবং Scalable কোড লেখার জন্য ব্যবহৃত হয়।

**S - Single Responsibility Principle (SRP)**

একটি ক্লাসের কেবলমাত্র একটি দায়িত্ব বা কাজ থাকা উচিত।

**উদাহরণ:**

// খারাপ উদাহরণ

public class Report {

public void GenerateReport() { /\* রিপোর্ট তৈরি \*/ }

public void SaveToFile() { /\* ফাইলে সেভ \*/ }

}

// ভালো উদাহরণ

public class ReportGenerator {

public void GenerateReport() { /\* রিপোর্ট তৈরি \*/ }

}

public class ReportSaver {

public void SaveToFile() { /\* ফাইলে সেভ \*/ }

}

**O - Open/Closed Principle (OCP)**

একটি ক্লাস পরিবর্তনের জন্য বন্ধ, কিন্তু সম্প্রসারণের জন্য উন্মুক্ত হওয়া উচিত।

**উদাহরণ:**

// ভালো উদাহরণ: নতুন টাইপের ডিসকাউন্ট যোগ করতে ক্লাস পরিবর্তন করতে হবে না

public abstract class Discount {

public abstract double Calculate(double price);

}

public class RegularDiscount : Discount {

public override double Calculate(double price) => price \* 0.9;

}

public class SeasonalDiscount : Discount {

public override double Calculate(double price) => price \* 0.8;

}

**L - Liskov Substitution Principle (LSP)**

বেজ (base) ক্লাসের যেকোন সাব-ক্লাস সেটিকে রিপ্লেস করতে পারবে, কিন্তু আচরণে পরিবর্তন আনবে না।

**উদাহরণ:**

public class Bird {

public virtual void Fly() { Console.WriteLine("Flying"); }

}

public class Sparrow : Bird { }

public class Ostrich : Bird {

public override void Fly() {

throw new NotImplementedException(); // খারাপ, কারণ Ostrich উড়তে পারে না

}

}

**সমাধান:**

public abstract class Bird { }

public interface IFlyable {

void Fly();

}

public class Sparrow : Bird, IFlyable {

public void Fly() { Console.WriteLine("Flying"); }

}

public class Ostrich : Bird { } // উড়ার প্রয়োজনে IFlyable implement করে না

**I - Interface Segregation Principle (ISP)**

ক্লাসগুলোকে তাদের প্রয়োজন অনুযায়ী ছোট ছোট ইন্টারফেস দিতে হবে, বড় ইন্টারফেস নয়।

**উদাহরণ:**

// খারাপ উদাহরণ

public interface IWorker {

void Work();

void Eat();

}

// ভালো উদাহরণ

public interface IWorkable {

void Work();

}

public interface IFeedable {

void Eat();

}

public class Robot : IWorkable {

public void Work() { Console.WriteLine("Robot working"); }

}

public class Human : IWorkable, IFeedable {

public void Work() { Console.WriteLine("Human working"); }

public void Eat() { Console.WriteLine("Human eating"); }

}

**D - Dependency Inversion Principle (DIP)**

হাই-লেভেল মডিউল (যেটা মূল কাজ করে) লো-লেভেল মডিউলের (যেটা ডিটেইল) উপর নির্ভর করবে না; দুটোই একটি অ্যাবস্ট্রাকশন এর উপর নির্ভর করবে।

**উদাহরণ:**

// খারাপ উদাহরণ

public class EmailSender {

public void SendEmail(string message) { Console.WriteLine("Email sent"); }

}

public class Notification {

private EmailSender sender = new EmailSender();

public void Notify(string message) {

sender.SendEmail(message);

}

}

**ভালো উদাহরণ (DIP অনুযায়ী):**

public interface IMessageSender {

void Send(string message);

}

public class EmailSender : IMessageSender {

public void Send(string message) { Console.WriteLine("Email sent"); }

}

public class Notification {

private readonly IMessageSender sender;

public Notification(IMessageSender sender) {

this.sender = sender;

}

public void Notify(string message) {

sender.Send(message);

}

}

**SOLID** মানে:

* **S**ingle Responsibility – এক ক্লাস, এক কাজ।
* **O**pen/Closed – নতুন ফিচার অ্যাড করো, পুরাতন কোড পরিবর্তন নয়।
* **L**iskov Substitution – বেস ক্লাসের যায়গায় সাব ক্লাস ব্যবহার করো যেন সমস্যা না হয়।
* **I**nterface Segregation – প্রয়োজন অনুযায়ী ছোট ইন্টারফেস।
* **D**ependency Inversion – High-level কোড অ্যাবস্ট্রাকশনের উপর নির্ভর করুক।

**1. Single Responsibility Principle (SRP)**

**📌 মূল কথা:**

একটি ক্লাস বা মডিউল শুধুমাত্র একটি কাজ বা দায়িত্ব পালন করবে।

**🧠 কেন গুরুত্বপূর্ণ?**

যদি একটি ক্লাস অনেক দায়িত্বে জড়িত থাকে, তাহলে কোড পরিবর্তন কঠিন হয়ে পড়ে। এক কাজ পরিবর্তন করলে অন্য কাজেও সমস্যা হতে পারে।

**❌ ভুল ব্যবহার:**

public class Invoice {

public void CalculateTotal() {

// Total হিসাব

}

public void SaveToDatabase() {

// ডাটাবেসে সংরক্ষণ

}

public void PrintInvoice() {

// ইনভয়েস প্রিন্ট

}

}

এই ক্লাসে তিনটি দায়িত্ব: হিসাব, সংরক্ষণ, এবং প্রিন্ট — যা SRP লঙ্ঘন করে।

**✅ সঠিক ব্যবহার:**

public class InvoiceCalculator {

public void CalculateTotal() {

// Total হিসাব

}

}

public class InvoiceRepository {

public void SaveToDatabase() {

// ডাটাবেসে সংরক্ষণ

}

}

public class InvoicePrinter {

public void PrintInvoice() {

// ইনভয়েস প্রিন্ট

}

}

**2. Open/Closed Principle (OCP)**

**📌 মূল কথা:**

ক্লাস/মডিউল **পরিবর্তনের জন্য বন্ধ** কিন্তু **প্রসারণের জন্য উন্মুক্ত** হওয়া উচিত।

**🧠 কেন গুরুত্বপূর্ণ?**

নতুন ফিচার যোগ করতে চাইলে মূল কোড পরিবর্তন না করে এক্সটেনশন দিয়ে কাজ করাই ভালো।

**❌ ভুল ব্যবহার:**

public class DiscountCalculator {

public double GetDiscount(string customerType, double amount) {

if (customerType == "Regular")

return amount \* 0.1;

else if (customerType == "Premium")

return amount \* 0.2;

else

return 0;

}

}

নতুন কাস্টমার টাইপ এলে কোডে if/else বাড়াতে হবে।

**✅ সঠিক ব্যবহার:**

public abstract class DiscountStrategy {

public abstract double GetDiscount(double amount);

}

public class RegularDiscount : DiscountStrategy {

public override double GetDiscount(double amount) => amount \* 0.1;

}

public class PremiumDiscount : DiscountStrategy {

public override double GetDiscount(double amount) => amount \* 0.2;

}

public class DiscountCalculator {

public double Calculate(DiscountStrategy strategy, double amount) {

return strategy.GetDiscount(amount);

}

}

এভাবে নতুন ডিসকাউন্ট টাইপ সহজেই অ্যাড করা যায়।

**3. Liskov Substitution Principle (LSP)**

**📌 মূল কথা:**

একটি সাবক্লাস অবশ্যই তার বেস ক্লাসের আচরণ বজায় রাখবে। সাবক্লাস ব্যবহারে সিস্টেমের কোনো আচরণ পরিবর্তন হওয়া উচিত নয়।

**❌ ভুল ব্যবহার:**

public class Bird {

public virtual void Fly() {

Console.WriteLine("Flying");

}

}

public class Ostrich : Bird {

public override void Fly() {

throw new NotImplementedException(); // উড়তে পারে না!

}

}

Ostrich আসলে Bird নয় যেহেতু সে উড়তে পারে না।

**✅ সঠিক ব্যবহার:**

public abstract class Bird { }

public interface IFlyingBird {

void Fly();

}

public class Sparrow : Bird, IFlyingBird {

public void Fly() {

Console.WriteLine("Flying");

}

}

public class Ostrich : Bird {

// Fly মেথড নেই

}

এভাবে লিস্কভ রুল বজায় থাকে।

**4. Interface Segregation Principle (ISP)**

**📌 মূল কথা:**

বড় ইন্টারফেস না দিয়ে ছোট ছোট ইন্টারফেস তৈরি করতে হবে যেন ক্লাসগুলো শুধুমাত্র প্রয়োজনীয় মেথডই ইমপ্লিমেন্ট করে।

**❌ ভুল ব্যবহার:**

public interface IWorker {

void Work();

void Eat();

}

public class Robot : IWorker {

public void Work() { }

public void Eat() {

throw new NotImplementedException(); // রোবট খায় না!

}

}

**✅ সঠিক ব্যবহার:**

public interface IWorkable {

void Work();

}

public interface IFeedable {

void Eat();

}

public class Human : IWorkable, IFeedable {

public void Work() { }

public void Eat() { }

}

public class Robot : IWorkable {

public void Work() { }

}

এভাবে শুধুমাত্র প্রয়োজনীয় দায়িত্ব দেওয়া যায়।

**5. Dependency Inversion Principle (DIP)**

**📌 মূল কথা:**

উচ্চ-স্তরের ক্লাস বা মডিউল (যা মূল লজিক কাজ করে) নিম্ন-স্তরের ক্লাসের উপর নির্ভর না করে **interface বা abstraction-এর উপর নির্ভর করবে**।

**❌ ভুল ব্যবহার:**

public class EmailService {

public void SendEmail(string message) {

Console.WriteLine("Sending Email...");

}

}

public class Notification {

private EmailService \_email = new EmailService();

public void NotifyUser(string message) {

\_email.SendEmail(message);

}

}

Notification ক্লাস EmailService-এর উপর নির্ভরশীল। অন্য কিছু ব্যবহার করা কঠিন।

**✅ সঠিক ব্যবহার:**

public interface IMessageService {

void Send(string message);

}

public class EmailService : IMessageService {

public void Send(string message) {

Console.WriteLine("Sending Email...");

}

}

public class SMSService : IMessageService {

public void Send(string message) {

Console.WriteLine("Sending SMS...");

}

}

public class Notification {

private readonly IMessageService \_messageService;

public Notification(IMessageService messageService) {

\_messageService = messageService;

}

public void NotifyUser(string message) {

\_messageService.Send(message);

}

}

এভাবে যেকোনো সার্ভিস (Email/SMS/Push) ব্যবহার করা যায়, কোড পরিবর্তন ছাড়াই।

| **Principle** | **বাংলা ব্যাখ্যা** | **উদাহরণ** |
| --- | --- | --- |
| SRP | এক ক্লাসে এক দায়িত্ব | রিপোর্ট তৈরি ≠ রিপোর্ট সংরক্ষণ |
| OCP | কোড পরিবর্তন না করে এক্সটেন্ড করো | নতুন ডিসকাউন্ট ক্লাস |
| LSP | সাবক্লাস বেস ক্লাসের মতো আচরণ করবে | Fly() না থাকা শ্রেয় |
| ISP | বড় ইন্টারফেস ভেঙে ছোট করো | IWorkable, IFeedable আলাদা |
| DIP | Interface বা abstraction এর উপর নির্ভর করো | IMessageService interface |

**🎯 প্রকল্পের ধরন: ইনভয়েস ম্যানেজমেন্ট সিস্টেম**

ফিচারসমূহ:

* ইনভয়েস তৈরি
* ডিসকাউন্ট প্রয়োগ
* ইনভয়েস প্রিন্ট
* ইনভয়েস ইমেইল করা

**✅ SOLID অনুযায়ী প্রজেক্ট ডিজাইন:**

**1. Single Responsibility Principle (SRP)**

**✅ সমস্যা:**

একটি InvoiceService ক্লাসে নিচের কাজগুলো একসাথে আছে:

* ইনভয়েস তৈরি
* ডিসকাউন্ট হিসাব
* ডাটাবেসে সংরক্ষণ
* প্রিন্ট করা
* ইমেইল পাঠানো

**❌ ভুল কোড (সব দায়িত্ব এক জায়গায়):**

public class InvoiceService {

public void CreateInvoice() { }

public void CalculateDiscount() { }

public void SaveToDb() { }

public void PrintInvoice() { }

public void SendEmail() { }

}

**✅ সঠিক কোড (SRP অনুযায়ী ভাগ করা):**

public class InvoiceCreator {

public Invoice Create() {

// ইনভয়েস তৈরি

}

}

public class DiscountCalculator {

public decimal Calculate(Invoice invoice) {

// ডিসকাউন্ট হিসাব

}

}

public class InvoiceRepository {

public void Save(Invoice invoice) {

// ডাটাবেসে সংরক্ষণ

}

}

public class InvoicePrinter {

public void Print(Invoice invoice) {

// প্রিন্ট

}

}

public class EmailSender {

public void Send(Invoice invoice) {

// ইমেইল পাঠানো

}

}

**2. Open/Closed Principle (OCP)**

**✅ সমস্যা:**

আপনি নতুন ডিসকাউন্ট টাইপ যোগ করতে চান, কিন্তু if-else দিয়ে কোড পরিবর্তন করতে হচ্ছে।

**❌ ভুল কোড:**

public class DiscountCalculator {

public decimal Calculate(string customerType, decimal amount) {

if (customerType == "Regular") return amount \* 0.1m;

if (customerType == "Premium") return amount \* 0.2m;

return 0;

}

}

**✅ সঠিক কোড (OCP অনুযায়ী):**

public interface IDiscountStrategy {

decimal GetDiscount(decimal amount);

}

public class RegularCustomerDiscount : IDiscountStrategy {

public decimal GetDiscount(decimal amount) => amount \* 0.1m;

}

public class PremiumCustomerDiscount : IDiscountStrategy {

public decimal GetDiscount(decimal amount) => amount \* 0.2m;

}

public class DiscountCalculator {

private readonly IDiscountStrategy \_strategy;

public DiscountCalculator(IDiscountStrategy strategy) {

\_strategy = strategy;

}

public decimal Calculate(decimal amount) {

return \_strategy.GetDiscount(amount);

}

}

নতুন ডিসকাউন্ট ক্লাস যোগ করলেই হবে, পুরাতন ক্লাসে কিছু পরিবর্তনের দরকার নেই।

**3. Liskov Substitution Principle (LSP)**

**✅ সমস্যা:**

সাবক্লাস এমন আচরণ করছে যা বেস ক্লাসের নিয়ম ভেঙে দিচ্ছে।

**❌ ভুল উদাহরণ:**

public class NotificationService {

public virtual void Send() {

Console.WriteLine("Sending Notification");

}

}

public class SMSNotification : NotificationService {

public override void Send() {

throw new Exception("SMS not supported"); // ভুল আচরণ

}

}

**✅ সঠিক উদাহরণ:**

public interface INotification {

void Send();

}

public class EmailNotification : INotification {

public void Send() {

Console.WriteLine("Sending Email");

}

}

public class SMSNotification : INotification {

public void Send() {

Console.WriteLine("Sending SMS");

}

}

প্রত্যেক ক্লাস তার নিজস্ব দায়িত্বে আচরণ করছে।

**4. Interface Segregation Principle (ISP)**

**✅ সমস্যা:**

সব ক্লায়েন্টকে বড় ইন্টারফেস ইমপ্লিমেন্ট করতে বাধ্য করা হচ্ছে।

**❌ ভুল উদাহরণ:**

public interface IInvoiceService {

void Create();

void Print();

void Email();

void Save();

}

সব ক্লাসে সব মেথড ইমপ্লিমেন্ট করতে বাধ্য হতে হয়।

**✅ সঠিক উদাহরণ:**

public interface IInvoiceCreator {

void Create();

}

public interface IInvoicePrinter {

void Print();

}

public interface IInvoiceEmailer {

void Email();

}

public interface IInvoiceSaver {

void Save();

}

প্রত্যেক ক্লাস শুধু তার দরকারি ইন্টারফেস ইমপ্লিমেন্ট করে।

**5. Dependency Inversion Principle (DIP)**

**✅ সমস্যা:**

হাই-লেভেল মডিউল EmailService ক্লাসের উপর সরাসরি নির্ভর করছে।

**❌ ভুল কোড:**

public class NotificationManager {

private EmailService \_emailService = new EmailService();

public void Notify(string message) {

\_emailService.Send(message);

}

}

**✅ সঠিক কোড:**

public interface IMessageSender {

void Send(string message);

}

public class EmailService : IMessageSender {

public void Send(string message) {

Console.WriteLine("Email sent: " + message);

}

}

public class NotificationManager {

private readonly IMessageSender \_messageSender;

public NotificationManager(IMessageSender sender) {

\_messageSender = sender;

}

public void Notify(string message) {

\_messageSender.Send(message);

}

}

এভাবে ইন্টারফেসের উপর নির্ভর করে, যেকোনো টাইপ সার্ভিস ব্যবহার করা যায় (SMS, Push, etc)।

**🔚 উপসংহার (Full Structure):**

/InvoiceSystem

├── Models/

│ └── Invoice.cs

├── Interfaces/

│ ├── IDiscountStrategy.cs

│ ├── IMessageSender.cs

│ ├── IInvoiceCreator.cs

│ └── ...

├── Services/

│ ├── InvoiceCreator.cs

│ ├── EmailSender.cs

│ ├── DiscountCalculator.cs

│ └── NotificationManager.cs

└── Program.cs

এই স্ট্রাকচার অনুযায়ী আপনার প্রজেক্ট:

* SRP অনুসরণ করে আলাদা ক্লাস
* OCP অনুযায়ী extensible
* LSP অনুযায়ী interchangeable
* ISP অনুযায়ী ছোট ইন্টারফেস
* DIP অনুযায়ী loosely coupled