به نام خدا

جواب سوال اول :

اصل SRP (Single Responsibility Principle) یا «اصل تک‌وظیفگی» می‌گوید که هر کلاس باید تنها یک مسئولیت یا وظیفه مشخص داشته باشد که باعث کاهش پیچیدگی، قابلیت تغییرپذیری بهتر ،

افزایش قابلیت استفاده مجدد و تسهیل تست‌نویسی می شود.

مثال از نقض SRP :

فرض کنید کلاسی به اسم Report داریم:

public class Report {

public String reportContent;

public void generateReport() {

// منطق تولید گزارش

}

public void printReport() {

// منطق چاپ گزارش

}

public void saveToFile() { // منطق ذخیره گزارش در فایل

}

}

اگر بخواهیم نحوه‌ی ذخیره در فایل را تغییر دهیم، این کلاس تغییر می‌کند. اگر روش چاپ تغییر کند، باز همین کلاس تغییر می‌کند. این نقض اصل SRP است.

اصلاح با رعایت SRP :

کلاس‌ها را بر اساس وظیفه جدا می‌کنیم:

public class Report {

public String content;

public void generate() {

// منطق تولید گزارش

}

}

public class ReportPrinter {

public void print(Report report) {

// منطق چاپ گزارش

}

}

public class ReportSaver {

public void saveToFile(Report report) {

// منطق ذخیره گزارش

}

}

در نتیجه تغییر در نحوه چاپ فقط ReportPrinter و تغییر در نحوه ذخیره فقط ReportSaver را تغییر می‌دهد و کلاس Report فقط مسئول تولید محتوای گزارش است.

جواب سوال دوم :

اصل OCP (Open/Closed Principle) می‌گوید کلاس‌ها و ماژول‌ها باید برای توسعه باز، و برای تغییر بسته باشند.

یعنی باید بتوانیم بدون تغییر در کد موجود، رفتار جدیدی به آن اضافه کنیم اگر مجبور باشیم برای افزودن ویژگی جدید، کد موجود را تغییر دهیم، ممکن است باگ‌های جدید ایجاد شود.

نقض ocp :

برای اضافه کردن شکل جدید (مثلاً مثلث)، باید کد همین کلاس تغییر کند.

public class ShapeDrawer

{

public void Draw(string shape)

{

if (shape == "Circle")

{

Console.WriteLine("Drawing Circle");

}

else if (shape == "Square")

{

Console.WriteLine("Drawing Square");

}

}

{

رعایت OCP :

حالا اگر بخواهیم Triangle اضافه کنیم، فقط کافیست کلاس جدیدش را بسازیم، بدون دستکاری در ShapeDrawer.

public class Triangle : IShape

{

public void Draw()

{

Console.WriteLine("Drawing Triangle");

}

{

جواب سوال سوم :

اصل LSP (Liskov Substitution Principle) هر جایی که از کلاس والد استفاده می‌کنیم، باید بتوانیم بدون مشکل از کلاس فرزند هم استفاده کنیم.

یعنی کلاس‌های فرزند باید رفتار کلاس والد را نقض نکنند و جایگزینی‌شان باعث تغییر غیرمنتظره در عملکرد نشود.

مثال نقض LSP :

public class Bird

{

public virtual void Fly()

{

Console.WriteLine("Bird is flying");

}

}

public class Ostrich : Bird

{

public override void Fly()

{

throw new NotSupportedException("Ostrich can't fly!");

}

}

Ostrich از Bird ارث‌بری کرده، اما نمی‌تواند پرواز کند.

اصلاح LSP :

راه‌حل اینه که کلاس‌ها رو طوری طراحی کنیم که رفتار نامعتبر تحمیل نشه.

public abstract class Bird

{

public abstract void Move();

}

public class Sparrow : Bird

{

public override void Move()

{

Console.WriteLine("Sparrow is flying");

}

}

public class Ostrich : Bird

{

public override void Move()

{

Console.WriteLine("Ostrich is running");

}

}

جواب تمرین چهارم :

اصلISP : کلاس‌ها نباید مجبور شوند متدهایی را پیاده‌سازی کنند که به آن‌ها نیازی ندارند.

در مثال زیر پرینتر ساده فقط چاپ می‌کنه، ولی مجبور شده متد Scan() رو هم پیاده‌سازی کنه.

public interface IPrinter

{

void Print();

void Scan();

}

public class SimplePrinter : IPrinter

{

public void Print()

{

Console.WriteLine("Printing...");

}

public void Scan()

{

throw new NotImplementedException("This printer can't scan!");

}

}

رعایت ISP :

public interface IPrinter

{

void Print();

}

public interface IScanner

{

void Scan();

}

public class SimplePrinter : IPrinter

{

public void Print()

{

Console.WriteLine("Printing...");

}

}

جواب تمرین پنجم :

اصل DIP (Dependency Inversion Principle) : ماژول‌های سطح بالا نباید به ماژول‌های سطح پایین وابسته باشند؛ هر دو باید به انتزاع (Interface/Abstract) وابسته باشند.

همچنین انتزاع نباید به جزئیات وابسته باشد، بلکه جزئیات باید به انتزاع وابسته باشند.

نقض DIP:

public class Lamp

{

public void On() => Console.WriteLine("Lamp On");

}

public class Switch

{

private Lamp lamp = new Lamp();

public void Press() => lamp.On();

}

مشکل این است که Switch فقط با Lamp کار می‌کنه.

رعایت: DIP

public interface IDevice

{

void On();

}

public class Lamp : IDevice

{

public void On() => Console.WriteLine("Lamp On");

}

public class Switch

{

private IDevice device;

public Switch(IDevice d) => device = d;

public void Press() => device.On();

}

حالا Switch می‌تونه با هر چیزی که IDevice باشه کار کنه، نه فقط Lamp.

جواب سوال ششم :

کدام یک از گزینه های زیر اصل باز-بسته را نقض میکند؟

A)افزودن یک کلاس جدید که از اینترفیس موجود ارثبری میکند

B)تغییر در کد کلاس برای افزودن قابلیت جدید

C)ایجاد یک متد جدید در یک کالس انتزاعی

D)پیادهسازی چندین اینترفیس کوچک در یک کلاس

جواب گزینه 2 ، علت طبق اصل OCP اگر بخواهیم ویژگی جدید اضافه کنیم، نباید مجبور باشیم کد کلاس موجود را تغییر دهیم.

باید بتوانیم قابلیت جدید را با افزودن کلاس یا متد جدید پیاده‌سازی کنیم، بدون دستکاری کد قدیمی.