1. **From the SOLID principles, What is the Open/Closed principle? Write an example.**

კლასი უნდა იყოს დახურული ცვლილებებისთვის და გახსნილი გაფართოებისთვის. ეს ნიშნავს, რომ არსებულ კოდს არ ვცვლით. ახალ შესაძლებლობებს ვამატებთ ახალი კლასებით.

მაგალითი: თუ მაქვს PaymentProcessor, რომელიც იღებს გადახდებს, ახალ გადახდის მეთოდს არ ვამატებ ძველი კლასის შეცვლით. ვქმნი ახალ კლასს, მაგალითად CardPayment ან PaypalPayment, და ინტერფეისის მეშვეობით ვაერთიანებ. არსებული PaymentProcessor არ იცვლება. ახალი გადახდის ტიპი ემატება ახალი კლასით.

```
public interface IPayment
{
   void Pay();
}

public class CardPayment : IPayment
{
   public void Pay()
   {
      Console.WriteLine("Card payment");
   }
}

public class PaypalPayment : IPayment
{
   public void Pay()
   {
      Console.WriteLine("Paypal payment");
   }
}

public class PaymentProcessor
{
   public void Process(IPayment payment)
   {
      payment.Pay();
   }
}
```

2. **From the SOLID principles, What is the Single Resposibility principle? Write an example.**

კლასს უნდა ჰქონდეს მხოლოდ ერთი პასუხისმგებლობა. ერთი მიზეზი ცვლილებისთვის.

მაგალითი: UserManager არ უნდა ინახავდეს ლოგებს. ლოგირება უნდა იყოს ცალკე Logger კლასში. UserManager არ აკეთებს ლოგირებას. Logger არის ცალკე.

```
public class UserManager
{
   public void CreateUser(string name)
   {
      Console.WriteLine("User created");
   }
}

public class Logger
{
   public void Log(string message)
   {
      Console.WriteLine(message);
   }
}
```

3. **What is Chain of Responsibility Design pattern? Write an example.**

ეს არის დიზაინის პატერნი, სადაც მოთხოვნა გადადის დამამუშავებლების ჯაჭვში. თითოეული დამამუშავებელი წყვეტს, დაამუშაოს თუ გადაამისამართოს მომდევნო ობიექტზე.

მაგალითი: ავტორიზაციის პროცესში Request გადის RoleCheckHandler, შემდეგ PermissionCheckHandler, შემდეგ AuthenticationHandler. თუ ერთმა ვერ დაამუშავა, გადასცემს შემდეგს. მოთხოვნა ჯერ RoleHandler-ში შედის. თუ ვერ დაამუშავა, მიდის PermissionHandler-ში.

```
public abstract class Handler
{
   protected Handler next;

   public void SetNext(Handler nextHandler)
   {
      next = nextHandler;
   }

   public abstract void Handle(string request);
}

public class RoleHandler : Handler
{
   public override void Handle(string request)
   {
      if (request == "RoleOK")
         Console.WriteLine("Role checked");
      else if (next != null)
         next.Handle(request);
   }
}

public class PermissionHandler : Handler
{
   public override void Handle(string request)
   {
      if (request == "PermOK")
         Console.WriteLine("Permission checked");
      else if (next != null)
         next.Handle(request);
   }
```

```
}
```

4. **What is Adapter Design pattern? Write an example.**

Adapter გამოიყენება მაშინ, როცა ორი შეუთავსებელი კლასი უნდა დავაკავშიროთ ისე, რომ კოდი არ გადავწეროთ. Adapter იცვლის ინტერფეისს ჩვენთვის შესაფერის ფორმატში.

მაგალითი: თუ მაქვს MediaPlayer, რომელიც უკრავს mp3-ს, და მაქვს AdvancedPlayer, რომელიც უკრავს mp4-ს, ვქმნი Mp4Adapter-ს, რომელიც mp4-ს mp3-ის სტანდარტზე მოარგებს. Adapter აკავშირებს mp4 დამკვრელს mp3-ის სტანდარტთან.

```
public interface IMediaPlayer
{
   void Play();
}

public class Mp3Player : IMediaPlayer
{
   public void Play()
   {
      Console.WriteLine("Play mp3");
   }
}

public class AdvancedMp4Player
{
   public void PlayMp4()
   {
      Console.WriteLine("Play mp4");
   }
}

public class Mp4Adapter : IMediaPlayer
{
   private AdvancedMp4Player mp4;

   public Mp4Adapter()
   {
      mp4 = new AdvancedMp4Player();
   }

   public void Play()
```

```
    {
        mp4.PlayMp4();
    }
}
```

5. **What is Decorator Design pattern? Write an example.**

Decorator გვაძლევს საშუალებას, ობიექტს დამატებითი ფუნქციონალი დავუმატოთ მისი სტრუქტურის ცვლილების გარეშე. გაფართოება ხდება wrapper კლასებით.

მაგალითი: Coffee ობიექტს დამატებით ვაძლევ შოკოლადს ან რძეს ისე, რომ Coffee კლასს არ ვიყენებ ახალ ვერსიებად. ვქმნი MilkDecorator ან ChocolateDecorator wrapper-ებს, რომლებიც ბაზის Coffee-ს ფუნქციას ავსებენ. BasicCoffee რჩება უცვლელად. MilkDecorator დამატებით ფუნქციას ამატებს.

```
public interface ICoffee
{
    string GetInfo();
}

public class BasicCoffee : ICoffee
{
    public string GetInfo()
    {
        return "Coffee";
    }
}

public class MilkDecorator : ICoffee
{
    private ICoffee coffee;

    public MilkDecorator(ICoffee c)
    {
        coffee = c;
    }

    public string GetInfo()
    {
        return coffee.GetInfo() + " + Milk";
    }
}
```