|  |  |
| --- | --- |
| SOILD Principle |  |

| **Principle** | **Name** | **Goal** |
| --- | --- | --- |

|  |  |  |
| --- | --- | --- |
| S | Single Responsibility | One reason to change |

|  |  |  |
| --- | --- | --- |
| O | Open/Closed | Open for extension, closed for modification |

|  |  |  |
| --- | --- | --- |
| L | Liskov Substitution | Subtypes must be replaceable |

|  |  |  |
| --- | --- | --- |
| I | Interface Segregation | No forced implementation |

|  |  |  |
| --- | --- | --- |
| D | Dependency Inversion | Depend on abstractions |

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

**Class should have only one reason to change.**

Bad example

|  |
| --- |
| public class InvoiceService  {  public void GenerateInvoice() { /\* logic \*/ }  public void SaveToDatabase() { /\* logic \*/ }  public void SendEmail() { /\* logic \*/ }  } |

❌ Mixing invoice logic, DB access, and email.

Good Example (SRP Applied):

|  |
| --- |
| public class InvoiceGenerator  {  public void GenerateInvoice() { /\* logic \*/ }  }  public class InvoiceRepository  {  public void Save(Invoice invoice) { /\* logic \*/ }  }  public class EmailService  {  public void Send(Invoice invoice) { /\* logic \*/ }  } |

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

**Software should be open for extension but closed for modification.**

**Bad Example**

|  |
| --- |
| public class DiscountService  {  public decimal GetDiscount(string type)  {  if (type == "Student") return 10;  if (type == "Senior") return 15;  return 0;  }  } |

Every time a new discount type is added, we modify this class.

**Good example**

|  |
| --- |
| public interface IDiscount  {  decimal GetDiscount();  }  public class StudentDiscount : IDiscount  {  public decimal GetDiscount() => 10;  }  public class SeniorDiscount : IDiscount  {  public decimal GetDiscount() => 15;  }  public class DiscountService  {  public decimal Calculate(IDiscount discount) => discount.GetDiscount();  } |

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

**Subtypes must be substitutable for their base types.**

BAD example

|  |
| --- |
| public class Bird  {  public virtual void Fly() {}  }  public class Ostrich : Bird  {  public override void Fly()  {  throw new NotImplementedException(); // Ostriches can't fly!  }  } |

Subclass violates expected behavior of the base

**Good example**

|  |
| --- |
| public abstract class Bird {}  public class FlyingBird : Bird  {  public void Fly() { /\* logic \*/ }  }  public class Ostrich : Bird  {  public void Run() { /\* logic \*/ }  } |

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

Clients should not be forced to depend on methods they do not use.

Bad Example

|  |
| --- |
| **public interface IWorker**  **{**  **void Work();**  **void Eat();**  **}**  **public class Robot : IWorker**  **{**  **public void Work() {}**  **public void Eat() { throw new NotImplementedException(); }**  **}** |

Robot doesn’t eat but is forced to implement Eat()

Good Example

|  |
| --- |
| **public interface IWorkable**  **{**  **void Work();**  **}**  **public interface IEatable**  **{**  **void Eat();**  **}**  **public class Human : IWorkable, IEatable**  **{**  **public void Work() {}**  **public void Eat() {}**  **}**  **public class Robot : IWorkable**  **{**  **public void Work() {}**  **}** |

Only implement what ever is needed

5. DIP Dependency Inversion Principle (DIP)

**High-level modules should not depend on low-level modules. Both should depend on abstractions.**

|  |
| --- |
| public class EmailService  {  public void SendEmail(string to) { /\* ... \*/ }  }  public class Notification  {  private EmailService \_emailService = new EmailService();  public void Send(string to)  {  \_emailService.SendEmail(to);  }  } |
|  |