Solid Principles

Saturday, February 8, 2025 8:51 AM

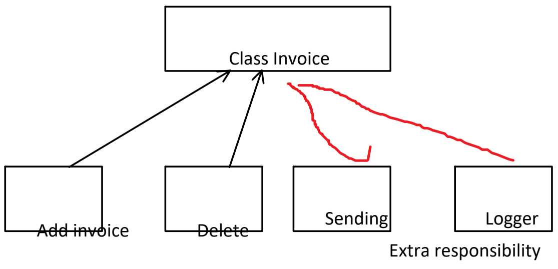
1. S -- Single Responsibility Principle
2. O-- Open Closed Principle
3. L -- Liskov Substitution Principle
4. I -- Interface Segregation Principle
5. D -- Dependency Inversion Principle

Single Responsibility Principle : When one class should have only one responsibility and the methods or members we create it should be the part of that class only or resemble to that class .

```
Class Login
{

login(){

}

Register(){

}

}
```

```
Public Class CommunicationService
{

Void Notification(){
Third party  service  EMAIL
}
Void Sms(){

}

}
```

```
Class Invoice

{



}
```



2. O -- Open closed Principle (Open for extension but closed for modification)
You can use that class to extend but not to modify that class

3. I -- Interface Segregation --> Do not combine all the functionalities in an interface . Interface should be different for different implementation

4. Liskov Substitution Principle :you can create an object a child class by taking a reference of base class but it should not change the behaviour of base class.

```
Class Apple

{
 void GetColor()
{
Return "Red";
}


}
```

```
Class Orange: Apple
{

 void GetColor()
{
Return "Orange";

}


}
```

```
Apple a = new Orange();


a.getColor();
```

```
Abstract Shape
{
Void Area()

}


Shape r = new Rectangle();
Shape s = new Square();
```

Rectangle r = new Square();

r.changedimension(r,5,6)

}

IBankType t = new currentAccount();

IBankType t = new SavingAccount();

5. Dependency Inversion Principle : High -Level Modules/ classes should not depend on Low-level classes

HLM and LLM should depend on the abstractions

Let's say:

```
//Entity class -- attributes, methods , properties , toString , constructor
EmployeeEntity.cs
{
Id ,name,

}
```

```
DataFactoryClass.cs
{

Public static EmployeeDataFetchingLayer getEmployeeObject()
{

 return new EmployeeDataFetchingLayer();
}


}
```

```
// DAO  -- Data base
EmployeeDataFetchingLayer.cs

{
Public Employee getEmployeeDetails(int id)
{
EmployeeEntity e = new EmployeeEntity()
{
Id = id,
Name = " Niti"
}
return e;

}

}
```

High Level Module : It is a module that always depends on other modules .. Tightly coupled

```
Class EmployeeBusinessLogic.cs

{
EmployeeDataFetchingLayer _DataAccess ;

Public EmployeeBusinessLogic()
{

  _DataAccess =  DataFactoryClass.getEmployeeObject();

}

Public Employee getEmployeeDetails(int id)
{
```

```
Return _DataAccess.getEmployeeDetails(id);


}

}
```

```
Interface IEmployeeDataFetchingLayer
{

// method declaration

Employee GetEmployeeDetails(int id);



}
```

```
Public class EmployeeDataAccessLogic :
IEmployeeDataFetchingLayer

{


Public Employee GetEmployeeDetails(int id)
{
//EmployeeEntity e = new EmployeeEntity()
{
Id = id,
Name = " Niti"
}
return e;}}}
```

```
Public class DataAccessFactory
{

Public static IEmployeeDataFetching getEmployeeObj()
{

Return new EmployeeDataAccessLogic;
}


}
```

```
Public class EmployeeBusinessLogic
{
IEmployeeDataFetchingLayer _Ifetching;

Public EmployeeBusinessLogic()
{
 _Ifetching = DataAccessFactory.getEmployeeObj()

}
Public Employee getEmployeeDetails(int id)
{
Return _Ifetching.GetEmployeeDetails(id);

}


}
```

HLM and LLM depend on abstraction
(IEmployeeDataFetchingLayer)

```
Program.cs

{
Main()
{

 EmployeeBusinessLogic el = new EmployeeBusinessLogic(
Employee emp = new
EmployeeBusinessLogic.getEmployeeDetails(23);
Console.Write("It will print all the details");

}


}
```

Dependency Inversion : Reduced dependencies , Easy to maintain the implementation details , testing will also b

Payment processing system