|  |  |
| --- | --- |
|  | LAB | SOLID Design Patterns |
|  | WORKSHOP | Design Patterns |
|  | Crystal Tenn  crtenn@microsoft.com |

# SETUP LAB ENVRIOMENT

## Prerequisites

To perform the tasks in this lab you need following:

* Visual Studio 2015 or 2017, any version

## Objectives

In this lab you will complete following tasks:

* Create a simple Console calculator application with dependency injection using Mef

### Estimated Completion Time: 30 minutes.

## Task: Create a Console Application

In this task you will create a new Console application.

1. Open Visual Studio.
2. Choose New C# Console Project.
3. Name it SolidPrinciples. Hit OK.
4. Add 5 folders to your SolidPrinciples project: SingleResponsibility, OpenClose, Liskov, InterfaceSegregation, Dependency Injection.

Note: **Single Responsibility Principle** states that each class should have responsibility over a single part of the functionality provided by the software, and that responsibility should be entirely encapsulated by the class. A class or module should have one, and only one, reason to be changed (i.e. rewritten). As an example, consider a module that compiles and prints a report. Imagine such a module can be changed for two reasons. First, the content of the report could change. Second, the format of the report could change. These two things change for very different causes; one substantive, and one cosmetic.

The single responsibility principle says that these two aspects of the problem are really two separate responsibilities and should therefore be in separate classes or modules. It would be a bad design to couple two things that change for different reasons at different times.

1. Add a new class called Employee to your SingleResponsibility folder.

public class Employee

{

public int Employee\_Id { get; set; }

public string Employee\_Name { get; set; }

/// <summary>

/// This method used to insert into employee table

/// </summary>

/// <param name="em">Employee object</param>

/// <returns>Successfully inserted or not</returns>

public bool InsertIntoEmployeeTable(Employee em)

{

// Insert into employee table.

return true;

}

/// <summary>

/// Method to generate report

/// </summary>

/// <param name="em"></param>

public void GenerateReport(Employee em)

{

// Report generation with employee data using crystal report.

}

}

Note: ‘Employee’ class is taking 2 responsibilities, dealing with an insert to the database and managing the generation of a report. Employee class should not be responsible for the report generation as this class could need to be changed. According to the Single Responsibility Principle, one class should take one responsibility so we should write one different class for report generation, so that any change in report generation should not affect the ‘Employee’ class.

1. Create a new class called ReportGeneration.cs in the SingleResponsibility folder and move the GenerateReport method into that class and remove it from your Employee class. Your ReportGeneration.cs class should look like this:

/// <summary>

/// Method to generate report

/// </summary>

/// <param name="em"></param>

public void GenerateReport(Employee em)

{

// Report generation with employee data using crystal report.

}

1. Let’s say that the GenerateReport method needs to change now. We have followed the Single Responsibility Principle and therefore any changes will not affect the Employee class. Each class now has one responsibility. Change the GenerateReport method to the following code:

public class ReportGeneration

{

/// <summary>

/// Report type

/// </summary>

public string ReportType { get; set; }

/// <summary>

/// Method to generate report

/// </summary>

/// <param name="em"></param>

public void GenerateReport(Employee em)

{

if (ReportType == "XLS")

{

// Report generation with employee data in Crystal Report.

}

if (ReportType == "PDF")

{

// Report generation with employee data in PDF.

}

}

}

Note: Open Closed Principle states "software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification"; that is, such an entity can allow its behavior to be extended without modifying its source code.

1. View our ReportGeneration.cs class. Notice we are using if statements, so we need to add more if statements in the future to add more types of reports. This is not the best practice as we will continually need to keep modifying this class. How can we make it open for extension and closed from modification?
2. In the OpenClose folder create a new interface called IReportGeneration:

interface IReportGeneration

{

/// <summary>

/// Method to generate report

/// </summary>

/// <param name="em"></param>

void GenerateReport(Employee em);

}

1. In the OpenClose folder, create a new class called ExcelReportGeneration.cs, this will depreciate part of the GenerateReport.cs class that we used earlier.

/// <summary>

/// Class to generate Excel report

/// </summary>

public class ExcelReportGeneration : IReportGeneration

{

public void GenerateReport(Employee em)

{

// Generate excel report.

}

}

1. Also in the OpenClose folder, create a new class called PdfReportGeneration.

/// <summary>

/// Class to generate PDF report

/// </summary>

public class PDFReportGeneration : IReportGeneration

{

public void GenerateReport(Employee em)

{

// Generate PDF report.

}

}

Note: Now, if you want to introduce a new report type, then just implement IReportGeneration, which is open for extension but closed for modification.

1. In the Liskov folder, create a new class Employee.cs with the following code:

public abstract class Employee

{

public virtual string GetProjectDetails(int employeeId)

{

return "Base Project";

}

public virtual string GetEmployeeDetails(int employeeId)

{

return "Base Employee";

}

}

Note: Liskov's notion of a behavioral subtype defines a notion of substitutability for objects; that is, if S is a subtype of T, then objects of type T in a program may be replaced with objects of type S without altering any of the desirable properties of that program (e.g. correctness).

In mathematics, a Square is a Rectangle. Indeed, it is a specialization of a rectangle. The "is a" makes you want to model this with inheritance. However, if in code you made Square derive from Rectangle, then a Square should be usable anywhere you expect a Rectangle. This makes for some strange behavior.

Imagine you had SetWidth and SetHeight methods on your Rectangle base class; this seems perfectly logical. However, if your Rectangle reference pointed to a Square, then SetWidth and SetHeight doesn't make sense because setting one would change the other to match it. In this case Square fails the Liskov Substitution Test with Rectangle and the abstraction of having Square inherit from Rectangle is a bad one.

1. Add a class called FullTimeExployee.cs with this code:

public class FullTimeEmployee : Employee

{

public override string GetProjectDetails(int employeeId)

{

return "Restaurant Project";

}

// May be for contractual employee we do not need to store the details into database.

public override string GetEmployeeBenefits(int employeeId)

{

return "Free Medical care for all";

}

}

1. Add a class called Contractor.cs with this code:

public class Contractor : Employee

{

public override string GetProjectDetails(int employeeId)

{

return "Marketing Project";

}

// May be for contractual employee we do not need to store the details into database.

public override string GetEmployeeBenefits(int employeeId)

{

throw new **NotImplementedException**();

}

}

1. Let’s pretend we will implement this code, based on the code already in the project what is the issue?

List<Employee> employeeList = new List<Employee>();

employeeList.Add(new ContractualEmployee());

employeeList.Add(new CasualEmployee());

foreach (Employee e in employeeList)

{

e.GetEmployeeBenefits(1245);

}

1. Contractors do not have medical benefits typically, so the GetEmployeeBenefits method is not going to be implemented in a Contractor class. Does it make sense to have both FullTimeEmployees and Contractors using the same Employee base class? No. So let’s get rid of the Employee class and make two interfaces instead in the Liskov folder.
2. Create the IEmployee.cs interface:

public interface IEmployee

{

string GetEmployeeBenefits(int employeeId);

}

1. Create the IProject.cs interface:

public interface IProject

{

string GetProjectDetails(int employeeId);

}

1. Now edit your FullTimeEmployee class to implement both needed interfaces:

public class FullTimeEmployee : IEmployee, IProject

{

public string GetProjectDetails(int employeeId)

{

return "Restaurant Project";

}

// May be for contractual employee we do not need to store the details into database.

public string GetEmployeeBenefits(int employeeId)

{

return "Child Employee";

}

}

1. Now, edit your Contractor class so it implements only IProject, and now you can get rid of the extra non-implemented method.

public class Contractor : IProject

{

public string GetProjectDetails(int employeeId)

{

return " Free Medical care for all";

}

}

1. Go to the interface segregation folder and add an interface called IEmployeeDatabase with this code:

public interface IEmployeeDatabase

{

bool AddEmployeeDetails();

bool ShowEmployeeBenefits(int employeeId);

}

1. Think about the FullTimeEmployee and the Contractor classes we had before. Contractors did not have Benefits, therefore it would be a bad fit to force both FullTime and Contractors to use this IEmployeeDatabase interface in order to do operations on the database. Instead of forcing one interface, we should give the Benefits responsibility to a new interface. Remove the IEmployeeDatabase interface.

Note: The interface-segregation principle (ISP) states that no client should be forced to depend on methods it does not use. ISP splits interfaces that are very large into smaller and more specific ones so that clients will only have to know about the methods that are of interest to them. Such shrunken interfaces are also called role interfaces. ISP is intended to keep a system decoupled and thus easier to refactor, change, and redeploy.

1. Implement an interface IAddOperation:

public interface IAddOperation

{

bool AddEmployeeDetails();

}

1. Implement an interface IGetsOperation:

public interface IGetOperation

{

bool ShowEmployeeDetails(int employeeId);

}

1. Now you can implement the interfaces as needed separately. The interface concerns are separated.

Note: The dependency inversion principle refers to a specific form of decoupling software modules. When following this principle, the conventional dependency relationships established from high-level, policy-setting modules to low-level, dependency modules are reversed, thus rendering high-level modules independent of the low-level module implementation details. The principle states:

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

B. Abstractions should not depend on details. Details should depend on abstractions.

By dictating that both high-level and low-level objects must depend on the same abstraction this design principle inverts the way some people may think about object-oriented programming.

The idea behind points A and B of this principle is that when designing the interaction between a high-level module and a low-level one, the interaction should be thought of as an abstract interaction between them. This not only has implications on the design of the high-level module, but also on the low-level one: the low-level one should be designed with the interaction in mind and it may be necessary to change its usage interface.

In many cases, thinking about the interaction in itself as an abstract concept allows the coupling of the components to be reduced without introducing additional coding patterns, allowing only a lighter and less implementation dependent interaction schema.

When the discovered abstract interaction schema(s) between two modules is/are generic and generalization makes sense, this design principle also leads to the following dependency inversion coding pattern.

1. In the Dependency inversion folder, create a new class Email.cs. Add the following code:

public class Email

{

public void SendEmail()

{

// code to send mail

}

}

1. Add a Notification.cs class:

public class Notification

{

private Email \_email;

public Notification()

{

\_email = new Email();

}

public void PromotionalNotification()

{

\_email.SendEmail();

}

}

Note: Notice how Notification class totally depends on Email class, because it only sends one type of notification. If we want to introduce any other like SMS then? We need to change the notification system also. And this is called tightly coupled. What can we do to make it loosely coupled?

1. Create an interface IMessenger:

public interface IMessenger

{

void SendMessage();

}

1. Change the Email class to:

public class Email : IMessenger

{

public void SendMessage()

{

throw new **NotImplementedException**();

}

}

1. Add a SMS.cs class that looks like:

public class SMS : IMessenger

{

public void SendMessage()

{

throw new **NotImplementedException**();

}

}

1. Change the Notification class to look like this:

public class Notification

{

private IMessenger \_iMessenger;

public Notification()

{

\_iMessenger = new Email();

}

public void DoNotify()

{

\_iMessenger.SendMessage();

}

}

Note: There are three types of dependency injection you could do to this Notification class:

// Constructor Injection

public class Notification

{

private IMessenger \_iMessenger;

public Notification(IMessenger pMessenger)

{

\_iMessenger = pMessenger;

}

public void DoNotify()

{

\_iMessenger.SendMessage();

}

}

// Property Injection

public class Notification

{

private IMessenger \_iMessenger;

public Notification()

{

}

public IMessenger MessageService

{

private get;

set

{

\_iMessenger = value;

}

}

public void DoNotify()

{

\_iMessenger.SendMessage();

}

}

// Method Injection

public class Notification

{

public void DoNotify(IMessenger pMessenger)

{

pMessenger.SendMessage();

}

}