SOLID are five basic principles which help to create good software architecture. SOLID is an acronym where:-

* S stands for SRP (Single responsibility principle
* O stands for OCP (Open closed principle)
* L stands for LSP (Liskov substitution principle)
* I stands for ISP ( Interface segregation principle)
* D stands for DIP ( Dependency inversion principle)

class Customer

{

public void Add()

{

try

{

*// Database code goes here*

}

catch (Exception ex)

{

System.IO.File.WriteAllText(@"c:\Error.txt", ex.ToString());

}

}

}

The above customer class is doing things **WHICH HE IS NOT SUPPOSED TO DO**. Customer class should do customer datavalidations, call the customer data access layer etc , but if you see the catch block closely it also doing LOGGING activity. In simple words its over loaded with lot of responsibility. The “Customer” class is now closed for any new modification but it’s open for extensions when new customer types are added to the project.

class FileLogger

{

public void Handle(string error)

{

System.IO.File.WriteAllText(@"c:\Error.txt", error);

}

}

class Customer

{

private FileLogger obj = new FileLogger();

publicvirtual void Add()

{

try

{

*// Database code goes here*

}

catch (Exception ex)

{

obj.Handle(ex.ToString());

}

}

}

Understanding “O” - Open closed principle

Let’s continue with our same customer class example. I have added a simple customer type property to the class. This property decided if this is a “Gold” ora “Silver” customer.

Depending on the same it calculates discount. Have a look at the “getDiscount” function which returns discount accordingly. 1 for Gold customer and 2 for Silver customer.

class Customer

{

private int \_CustType;

public int CustType

{

get { return \_CustType; }

set { \_CustType = value; }

}

public double getDiscount(double TotalSales)

{

if (\_CustType == 1)

{

return TotalSales - 100;

}

else

{

return TotalSales - 50;

}

}

}

class Customer

{

public virtual double getDiscount(double TotalSales)

{

return TotalSales;

}

}

class SilverCustomer : Customer

{

public override double getDiscount(double TotalSales)

{

return base.getDiscount(TotalSales) - 50;

}

}

Hide   Copy Code

class goldCustomer : SilverCustomer

{

public override double getDiscount(double TotalSales)

{

return base.getDiscount(TotalSales) - 100;

}

}

## Understanding “L”- LSP (Liskov substitution principle)

Let’s continue with the same customer. Let’s say our system wants to calculate discounts for Enquiries. Now Enquiries are not actual customer’s they are just leads. Because they are just leads we do not want to save them to database for now.

So we create a new class called as Enquiry which inherits from the “Customer” class. We provide some discounts to the enquiry so that they can be converted to actual customers and we override the “Add’ method with an exception so that no one can add an Enquiry to the database.

class Enquiry : Customer

{

public override double getDiscount(double TotalSales)

{

return base.getDiscount(TotalSales) - 5;

}

public override void Add()

{

throw new Exception("Not allowed");

}

}

List<Customer> Customers = new List<Customer>();

Customers.Add(new SilverCustomer());

Customers.Add(new goldCustomer());

Customers.Add(new Enquiry());

foreach (Customer o in Customers)

{

o.Add();

}

}

But when “Add” method of the “Enquiry” object is invoked it leads to below error because our “Equiry” object does save enquiries to database as they are not actual customers.

interface IDiscount

{

double getDiscount(double TotalSales);

}

interface IDatabase

{

void Add();

}

class Enquiry : IDiscount

{

public double getDiscount(double TotalSales)

{

return TotalSales - 5;

}

}

class Customer : IDiscount, IDatabase

{

private MyException obj = new MyException();

public virtual void Add()

{

try

{

*// Database code goes here*

}

catch (Exception ex)

{

obj.Handle(ex.Message.ToString());

}

}

public virtual double getDiscount(double TotalSales)

{

return TotalSales;

}

}

## Understanding “I” - ISP (Interface Segregation principle)

Now let’s say some new clients come up with a demand saying that we also want a method which will help us to “Read” customer data.

interface IDatabase

{

void Add(); *// old client are happy with these.*

void Read(); *// Added for new clients.*

}

Now by changing the current interface you are doing an awful thing, disturbing the 1000 satisfied current client’s , even when they are not interested in the “Read” method. You are forcing them to use the “Read” method.

interface IDatabaseV1 : IDatabase *// Gets the Add method*

{

Void Read();

}

class CustomerwithRead : IDatabase, IDatabaseV1

{

public void Add()

{

Customer obj = new Customer();

Obj.Add();

}

Public void Read()

{

*// Implements logic for read*

}

}

IDatabase i = new Customer(); *// 1000 happy old clients not touched*

i.Add();

IDatabaseV1 iv1 = new CustomerWithread(); *// new clients*

Iv1.Read();

## Understanding “D”- Dependency inversion principle

class Customer

{

private FileLogger obj = new FileLogger();

public virtual void Add()

{

try

{

*// Database code goes here*

}

catch (Exception ex)

{

obj.Handle(ex.ToString());

}

}

}

interface ILogger

{

void Handle(string error);

}

class FileLogger : ILogger

{

public void Handle(string error)

{

System.IO.File.WriteAllText(@"c:\Error.txt", error);

}

}

Hide   Copy Code

class EverViewerLogger : ILogger

{

public void Handle(string error)

{

*// log errors to event viewer*

}

}

Hide   Copy Code

class EmailLogger : ILogger

{

public void Handle(string error)

{

*// send errors in email*

}

}

Hide   Copy Code

class Customer : IDiscount, IDatabase

{

private IException obj;

public virtual void Add(int Exhandle)

{

try

{

*// Database code goes here*

}

catch (Exception ex)

{

if (Exhandle == 1)

{

obj = new MyException();

}

else

{

obj = new EmailException();

}

obj.Handle(ex.Message.ToString());

}

}

The above code is again violating SRP but this time the aspect is different ,its about deciding which objects should be created. Now it’s not the work of “Customer” object to decide which instances to be created , he should be concentrating only on Customer class related functionalities.

If you watch closely the biggest problem is the “NEW” keyword. He is taking extra responsibilities of which object needs to be created.

So if we INVERT / DELEGATE this responsibility to someone else rather the customer class doing it that would really solve the problem to a certain extent.

So here’s the modified code with INVERSION implemented. We have opened the constructor mouth and we expect someone else to pass the object rather than the customer class doing it. So now it’s the responsibility of the client who is consuming the customer object to decide which Logger class to inject.

Hide   Copy Code

class Customer : IDiscount, IDatabase

{

private Ilogger obj;

public Customer(ILogger i)

{

obj = i;

}

}

So now the client will inject the Logger object and the customer object is now free from those IF condition which decide which logger class to inject. This is the Last principle in SOLID Dependency Inversion principle.

Customer class has delegated the dependent object creation to client consuming it thus making the customer class concentrate on his work.

Hide   Copy Code

IDatabase i = new Customer(new EmailLogger());

**Unit of Work design** pattern does two important things: first it maintains in-memory updates and second it sends these in-memory updates as one transaction to the database.

So to achieve the above goals it goes through two steps:

* It maintains lists of business objects in-memory which have been changed (inserted, updated, or deleted) during a transaction.
* Once the transaction is completed, all these updates are sent as one **big unit of work** to be persisted physically in a database in one **go**.

1 customer CRUD = 1 unit of work

### **Step 1: Create a generalized interface (IEntity) for business objects**

At the end of the day a unit of work is nothing but a collection which maintains and track changes on the business objects.

So the first step is to create a generalized interface called IEntity which represents a business object in our project.

This IEntity interface will have an ID property and methods (insert, update, delete, and load) which will help us to do the CRUD operation on the business object. The ID property is a unique number which helps us uniquely identify the record in a database.

Hide   Copy Code

public interface IEntity

{

int Id { set; get; }

void Insert();

void Update();

List<IEntity> Load();

}

### **Step 2: Implement the IEntity interface**

The next step is to implement the IEntity in all our business objects

public class Customer : IEntity

{

private int \_CustomerCode = 0;

public int Id

{

get { return \_CustomerCode; }

set { \_CustomerCode = value; }

}

private string \_CustomerName = "";

public string CustomerName

{

get { return \_CustomerName; }

set { \_CustomerName = value; }

}

public void Insert()

{

DataAccess obj = new DataAccess();

obj.InsertCustomer(\_CustomerCode, CustomerName);

}

public List<IEntity> Load()

{

DataAccess obj = new DataAccess();

Customer o = new Customer();

SqlDataReader ds = obj.GetCustomer(Id);

while (ds.Read())

{

o.CustomerName = ds["CustomerName"].ToString();

}

List<IEntity> Li = (new List<Customer>()).ToList<IEntity>();

Li.Add((IEntity) o);

return Li;

}

public void Update()

{

DataAccess obj = new DataAccess();

obj.UpdateCustomer(\_CustomerCode, CustomerName);

}

}

### **Step 3: Create the unit of work collection**

The next step is to create the unit of work collection class.

public class SimpleExampleUOW

{

private List<IEntity> Changed = new List<IEntity>();

private List<IEntity> New = new List<IEntity>();

public void Add(IEntity obj)

{

New.Add(obj);

}

public void Committ()

{

using (TransactionScope scope = new TransactionScope())

{

foreach (IEntity o in Changed)

{

o.Update();

}

foreach (IEntity o in New)

{

o.Insert();

}

scope.Complete();

}

}

public void Load(IEntity o)

{

Changed = o.Load() as List<IEntity>;

}

}

### **Step 4: See it working**

On the client side we can create the Customer object, add business objects in memory, and finally all these changes are sent in an atomic manner to the physical database by calling the Commit method.

Hide   Copy Code

Customer Customerobj = new Customer();*// record 1 Customer*

Customerobj.Id = 1000;

Customerobj.CustomerName = "shiv";

Supplier SupplierObj = new Supplier(); *// Record 2 Supplier*

Supplierobj.Id = 2000;

Supplierobj.SupplierName = "xxxx";

SimpleExampleUOW UowObj = new SimpleExampleUOW();

UowObj.Add(Customerobj); *// record 1 added to inmemory*

UowObj.Add(Supplierobj); *// record 1 added to inmemory*

UowObj.Committ(); *// The full inmemory collection is sent for final committ*

Observer Pattern

Consider an online electronics store which has a huge inventory and they keep on updating it. **The store wants to update all its users/customers whenever any product arrives in the store**. **The online electronic store is going to be the subject. Whenever the subject would have any addition in its inventory, the observers (customers/users) who have subscribed to store notifications would be notified through email.**

interface ISubject

{

void Subscribe(Observer observer);

void Unsubscribe(Observer observer);

void Notify();

}

interface IObserver

{

void Update();

}

public class Subject:ISubject

{

private List<Observer> observers = new List<Observer>();

private int \_int;

public int Inventory

{

get

{

return \_int;

}

set

{

*// Just to make sure that if there is an increase in inventory then only we are notifying*

the observers.

if (value > \_int)

Notify();

\_int = value;

}

}

public void Subscribe(Observer observer)

{

observers.Add(observer);

}

public void Unsubscribe(Observer observer)

{

observers.Remove(observer);

}

public void Notify()

{

observers.ForEach(x => x.Update());

}

}

public class Observer:IObserver

{

public string ObserverName { get;private set; }

public Observer(string name)

{

this.ObserverName = name;

}

public void Update()

{

Console.WriteLine("{0}: A new product has arrived at the

store",this.ObserverName);

}

}

Dispose Pattern

When we want to clear resource after uses of any object, we can implement this pattern. In finalize dispose pattern we have to implement one interface called IDisposable in class where we want to implement.

public class Garbage : IDisposable  
    {  
        public String name = String.Empty;  
        public SqlConnection con = null;  
        public Garbage()  
        {  
            name = "This is managable resource";  
            con = new SqlConnection();  
        }  
        ~Garbage()  
        {  
            Dispose(false);

}  
        protected virtual void Dispose(bool disposing)  
        {  
            if (disposing)  
            {  
                //Free managed resource here  
                name = null;  
            }  
            //Free unmanaged resource here  
            con = null;  
            Console.WriteLine("Object has disposed");  
        }  
   
        public void Dispose()  
        {  
            Dispose(true);  
            GC.SuppressFinalize(this);  
        }  
    }  
      
    class Program  
    {  
        static void Main(string[] args)  
        {  
            Garbage g = new Garbage();  
            Console.WriteLine("String created" + g.name);  
            g.Dispose();  //Dispose the g object  
              
            Console.ReadLine();  
        }  
    }

Singleton

singleton is a class which only allows a single instance of itself to be created, and usually gives simple access to that instance.

public sealed class Singleton  
{  
    private static Singleton instance = null;  
    private static readonly object padlock = new object();  
  
    private Singleton()  
    {  
    }  
  
    public static Singleton Instance  
    {  
        get  
        {  
            if (instance == null)  
            {  
                lock (padlock)  
                {  
                    if (instance == null)  
                    {  
                        instance = new Singleton();  
                    }  
                }  
            }  
            return instance;  
        }  
    }  
}

* A static class can not be a top level class and can not implement interfaces where a singleton class can.
* All members of a static class are static but for a Singleton class it is not a requirement.
* A static class get initialized when it is loaded so it can not be lazily loaded where a singleton class can be lazily loaded.
* A static class object is stored in stack whereas singlton class object is stored in heap memory space.

# **Factory Method**

Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

1. using System;
2. using System.Collections.Generic;
4. namespace DoFactory.GangOfFour.Factory.RealWorld
5. {
6. /// <summary>
7. /// MainApp startup class for Real-World
8. /// Factory Method Design Pattern.
9. /// </summary>
10. class MainApp
11. {
12. /// <summary>
13. /// Entry point into console application.
14. /// </summary>
15. static void Main()
16. {
17. // Note: constructors call Factory Method
18. Document[] documents = new Document[2];
20. documents[0] = new Resume();
21. documents[1] = new Report();
23. // Display document pages
24. foreach (Document document in documents)
25. {
26. Console.WriteLine("\n" + document.GetType().Name + "--");
27. foreach (Page page in document.Pages)
28. {
29. Console.WriteLine(" " + page.GetType().Name);
30. }
31. }
33. // Wait for user
34. Console.ReadKey();
35. }
36. }
38. /// <summary>
39. /// The 'Product' abstract class
40. /// </summary>
41. abstract class Page
42. {
43. }
45. /// <summary>
46. /// A 'ConcreteProduct' class
47. /// </summary>
48. class SkillsPage : Page
49. {
50. }
52. /// <summary>
53. /// A 'ConcreteProduct' class
54. /// </summary>
55. class EducationPage : Page
56. {
57. }
59. /// <summary>
60. /// A 'ConcreteProduct' class
61. /// </summary>
62. class ExperiencePage : Page
63. {
64. }
66. /// <summary>
67. /// A 'ConcreteProduct' class
68. /// </summary>
69. class IntroductionPage : Page
70. {
71. }
73. /// <summary>
74. /// A 'ConcreteProduct' class
75. /// </summary>
76. class ResultsPage : Page
77. {
78. }
80. /// <summary>
81. /// A 'ConcreteProduct' class
82. /// </summary>
83. class ConclusionPage : Page
84. {
85. }
87. /// <summary>
88. /// A 'ConcreteProduct' class
89. /// </summary>
90. class SummaryPage : Page
91. {
92. }
94. /// <summary>
95. /// A 'ConcreteProduct' class
96. /// </summary>
97. class BibliographyPage : Page
98. {
99. }
101. /// <summary>
102. /// The 'Creator' abstract class
103. /// </summary>
104. abstract class Document
105. {
106. private List<Page> \_pages = new List<Page>();
108. // Constructor calls abstract Factory method
109. public Document()
110. {
111. this.CreatePages();
112. }
114. public List<Page> Pages
115. {
116. get { return \_pages; }
117. }
119. // Factory Method
120. public abstract void CreatePages();
121. }
123. /// <summary>
124. /// A 'ConcreteCreator' class
125. /// </summary>
126. class Resume : Document
127. {
128. // Factory Method implementation
129. public override void CreatePages()
130. {
131. Pages.Add(new SkillsPage());
132. Pages.Add(new EducationPage());
133. Pages.Add(new ExperiencePage());
134. }
135. }
137. /// <summary>
138. /// A 'ConcreteCreator' class
139. /// </summary>
140. class Report : Document
141. {
142. // Factory Method implementation
143. public override void CreatePages()
144. {
145. Pages.Add(new IntroductionPage());
146. Pages.Add(new ResultsPage());
147. Pages.Add(new ConclusionPage());
148. Pages.Add(new SummaryPage());
149. Pages.Add(new BibliographyPage());
150. }
151. }
152. }

Abstract Factory

Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

1. using System;
3. namespace DoFactory.GangOfFour.Abstract.RealWorld
4. {
5. /// <summary>
6. /// MainApp startup class for Real-World
7. /// Abstract Factory Design Pattern.
8. /// </summary>
9. class MainApp
10. {
11. /// <summary>
12. /// Entry point into console application.
13. /// </summary>
14. public static void Main()
15. {
16. // Create and run the African animal world
17. ContinentFactory africa = new AfricaFactory();
18. AnimalWorld world = new AnimalWorld(africa);
19. world.RunFoodChain();
21. // Create and run the American animal world
22. ContinentFactory america = new AmericaFactory();
23. world = new AnimalWorld(america);
24. world.RunFoodChain();
26. // Wait for user input
27. Console.ReadKey();
28. }
29. }

32. /// <summary>
33. /// The 'AbstractFactory' abstract class
34. /// </summary>
35. abstract class ContinentFactory
36. {
37. public abstract Herbivore CreateHerbivore();
38. public abstract Carnivore CreateCarnivore();
39. }
41. /// <summary>
42. /// The 'ConcreteFactory1' class
43. /// </summary>
44. class AfricaFactory : ContinentFactory
45. {
46. public override Herbivore CreateHerbivore()
47. {
48. return new Wildebeest();
49. }
50. public override Carnivore CreateCarnivore()
51. {
52. return new Lion();
53. }
54. }
56. /// <summary>
57. /// The 'ConcreteFactory2' class
58. /// </summary>
59. class AmericaFactory : ContinentFactory
60. {
61. public override Herbivore CreateHerbivore()
62. {
63. return new Bison();
64. }
65. public override Carnivore CreateCarnivore()
66. {
67. return new Wolf();
68. }
69. }
71. /// <summary>
72. /// The 'AbstractProductA' abstract class
73. /// </summary>
74. abstract class Herbivore
75. {
76. }
78. /// <summary>
79. /// The 'AbstractProductB' abstract class
80. /// </summary>
81. abstract class Carnivore
82. {
83. public abstract void Eat(Herbivore h);
84. }
86. /// <summary>
87. /// The 'ProductA1' class
88. /// </summary>
89. class Wildebeest : Herbivore
90. {
91. }
93. /// <summary>
94. /// The 'ProductB1' class
95. /// </summary>
96. class Lion : Carnivore
97. {
98. public override void Eat(Herbivore h)
99. {
100. // Eat Wildebeest
101. Console.WriteLine(this.GetType().Name +
102. " eats " + h.GetType().Name);
103. }
104. }
106. /// <summary>
107. /// The 'ProductA2' class
108. /// </summary>
109. class Bison : Herbivore
110. {
111. }
113. /// <summary>
114. /// The 'ProductB2' class
115. /// </summary>
116. class Wolf : Carnivore
117. {
118. public override void Eat(Herbivore h)
119. {
120. // Eat Bison
121. Console.WriteLine(this.GetType().Name +
122. " eats " + h.GetType().Name);
123. }
124. }
126. /// <summary>
127. /// The 'Client' class
128. /// </summary>
129. class AnimalWorld
130. {
131. private Herbivore \_herbivore;
132. private Carnivore \_carnivore;
134. // Constructor
135. public AnimalWorld(ContinentFactory factory)
136. {
137. \_carnivore = factory.CreateCarnivore();
138. \_herbivore = factory.CreateHerbivore();
139. }
141. public void RunFoodChain()
142. {
143. \_carnivore.Eat(\_herbivore);
144. }
145. }
146. }

Facade

Provide a unified interface to a set of interfaces in a subsystem. Façade defines a higher-level interface that makes the subsystem easier to use.

1. using System;
3. namespace DoFactory.GangOfFour.Facade.RealWorld
4. {
5. /// <summary>
6. /// MainApp startup class for Real-World
7. /// Facade Design Pattern.
8. /// </summary>
9. class MainApp
10. {
11. /// <summary>
12. /// Entry point into console application.
13. /// </summary>
14. static void Main()
15. {
16. // Facade
17. Mortgage mortgage = new Mortgage();
19. // Evaluate mortgage eligibility for customer
20. Customer customer = new Customer("Ann McKinsey");
21. bool eligible = mortgage.IsEligible(customer, 125000);
23. Console.WriteLine("\n" + customer.Name +
24. " has been " + (eligible ? "Approved" : "Rejected"));
26. // Wait for user
27. Console.ReadKey();
28. }
29. }
31. /// <summary>
32. /// The 'Subsystem ClassA' class
33. /// </summary>
34. class Bank
35. {
36. public bool HasSufficientSavings(Customer c, int amount)
37. {
38. Console.WriteLine("Check bank for " + c.Name);
39. return true;
40. }
41. }
43. /// <summary>
44. /// The 'Subsystem ClassB' class
45. /// </summary>
46. class Credit
47. {
48. public bool HasGoodCredit(Customer c)
49. {
50. Console.WriteLine("Check credit for " + c.Name);
51. return true;
52. }
53. }
55. /// <summary>
56. /// The 'Subsystem ClassC' class
57. /// </summary>
58. class Loan
59. {
60. public bool HasNoBadLoans(Customer c)
61. {
62. Console.WriteLine("Check loans for " + c.Name);
63. return true;
64. }
65. }
67. /// <summary>
68. /// Customer class
69. /// </summary>
70. class Customer
71. {
72. private string \_name;
74. // Constructor
75. public Customer(string name)
76. {
77. this.\_name = name;
78. }
80. // Gets the name
81. public string Name
82. {
83. get { return \_name; }
84. }
85. }
87. /// <summary>
88. /// The 'Facade' class
89. /// </summary>
90. class Mortgage
91. {
92. private Bank \_bank = new Bank();
93. private Loan \_loan = new Loan();
94. private Credit \_credit = new Credit();
96. public bool IsEligible(Customer cust, int amount)
97. {
98. Console.WriteLine("{0} applies for {1:C} loan\n",
99. cust.Name, amount);
101. bool eligible = true;
103. // Check creditworthyness of applicant
104. if (!\_bank.HasSufficientSavings(cust, amount))
105. {
106. eligible = false;
107. }
108. else if (!\_loan.HasNoBadLoans(cust))
109. {
110. eligible = false;
111. }
112. else if (!\_credit.HasGoodCredit(cust))
113. {
114. eligible = false;
115. }
117. return eligible;
118. }
119. }
120. }