

Agile Software Development

By- Md. Mojammel Haque

CSM, CSPO, CSD, CSP (Scrum Alliance)

Software Architect- Raven Systems Ltd.

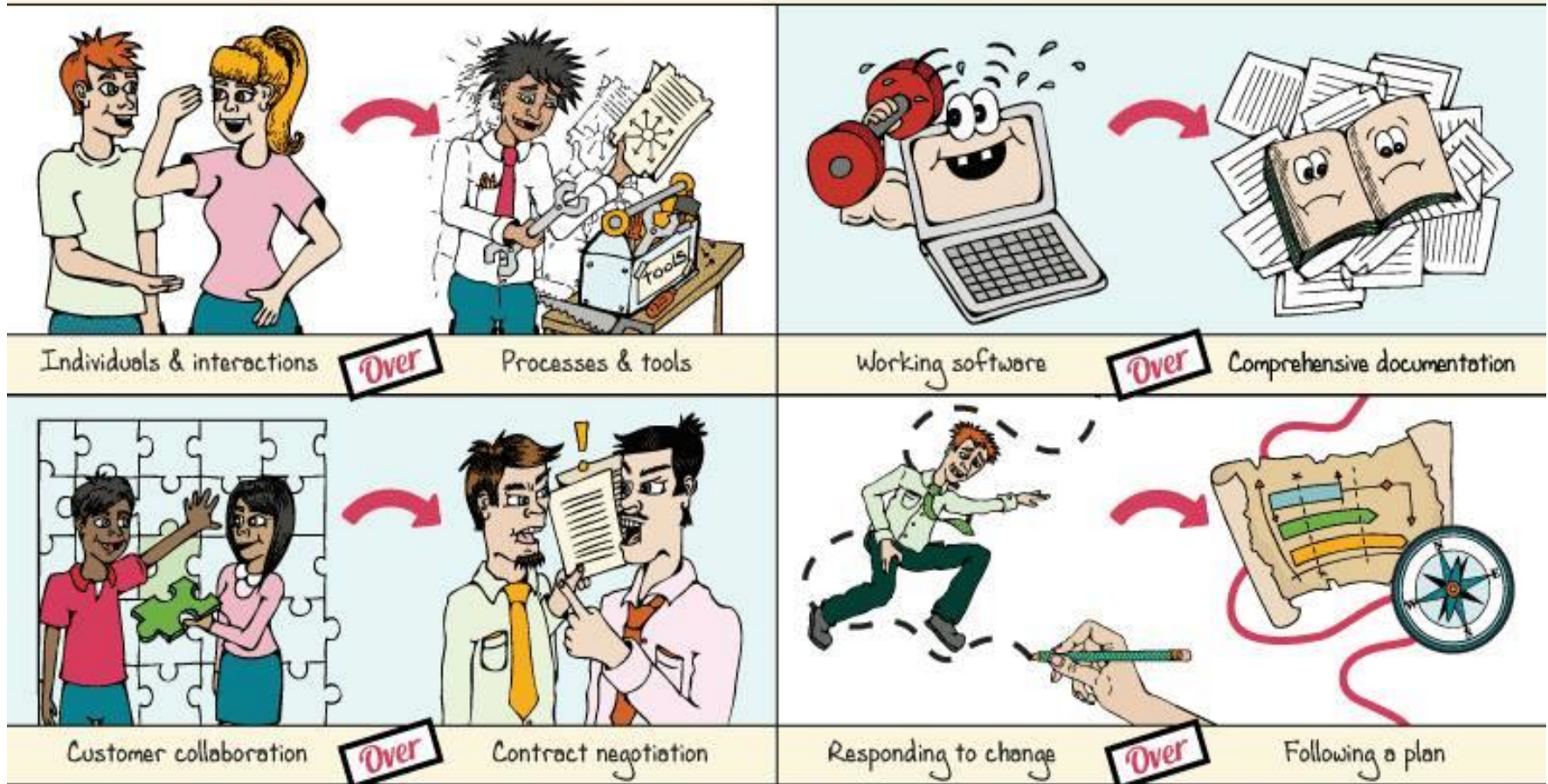
mojamcpds@gmail.com

<http://codermojam.com>

Cell- 01795-231350

Manifesto for Agile Software Development*

"We are uncovering better ways of developing software by doing it and helping others do it.
Through this work we have come to value:



That is, while there is value in the items on the right, we value the items on the left more."

Agile Principles

- *Satisfy Customer by through early and contentious delivery of Valuable Software*
- *Welcome changing requirements even late in Development*
- *Deliver working Software frequently from a couple of weeks to couple of months*
- *Business People & Developers must work together daily*

Agile Principles

- *Build projects around motivated individuals who should be trusted*
- *Face to face communication is the best form of Conversation even in co-location*
- *Working software is the principal measure of progress*
- *Sustainable development, able to maintain a constant pace*

Agile Principles

- *Continuous attention to technical excellence and good design*
- *Simplicity—the art of maximizing the amount of work not done—is essential*
- *Self-organizing teams*
- *Regular adaptation to changing circumstance*

Agile Software Fundamentals

- **Principles**
- **Patterns**
- **Practices**

Spaghetti Code

11 references | 0/6 passing

```
public IList<IEmployee> GetAllEmployees()
{
    IList<IEmployee> results = new List<IEmployee>();
    SqlConnection sqlConnection = new SqlConnection("data source=(local);Integrated Security=SSPI;Initial Catalog=DesignPatterns");
    DataTable rawData = new DataTable();
    using (sqlConnection)
    {
        SqlCommand sqlCommand = sqlConnection.CreateCommand();
        sqlCommand.CommandText = "SELECT * FROM Employees";
        sqlCommand.CommandType = System.Data.CommandType.Text;

        sqlConnection.Open();

        SqlDataReader sqlDataReader = sqlCommand.ExecuteReader();
        using (sqlDataReader)
        {
            rawData.Load(sqlDataReader);
        }
    }

    foreach (DataRow dr in rawData.Rows)
    {
        results.Add(new Employee(
            dr["FirstName"].ToString(),
            dr["LastName"].ToString(),
            Convert.ToDateTime(dr["BirthDate"])
        ));
    }

    return results;
}
```

Simplify Client Access to complex functionality- Façade

Problem with Above Code

- **Violation of Single Responsibility Principle**
 - Establish & Open Connection
 - Loading Data Table
 - Taking the Result & Mapping with Domain Object
- **Violation of Don't Repeat Yourself Principle**
 - Each time we create new method lot of stuff need to repeat e.g.- Define Connection, Provide Connection String, Open Connection, Create Command etc.

Refactoring To Pattern- Factory

```
public interface IDbConnectionFactory
{
    2 references
    IDbConnection Create();
}

public class DbConnectionFactory: IDbConnectionFactory
{
    2 references
    public IDbConnection Create()
    {
        SqlConnection sqlConnection = new SqlConnection("data source=(local);Integrated Security=SSPI;Initial Catalog=DesignPatterns");
        return sqlConnection;
    }
}
```

- Only use to establish the connection with Database- SRP
- Can reuse this code in whole project- DRY
- Create As much instance as required- Factory Pattern

Use Factory pattern to Façade

```
public IList<IEmployee> GetAllEmployees()
{
    IList<IEmployee> results = new List<IEmployee>();

    DataTable rawData = new DataTable();
    using (IDbConnection sqlConnection = new DbConnectionFactory().Create())
    {
        IDbCommand sqlCommand = sqlConnection.CreateCommand();
        sqlCommand.CommandText = "SELECT * FROM Employees";
        sqlCommand.CommandType = System.Data.CommandType.Text;

        sqlConnection.Open();

        IDataReader sqlDataReader = sqlCommand.ExecuteReader();
        using (sqlDataReader)
        {
            rawData.Load(sqlDataReader);
        }
    }

    foreach (DataRow dr in rawData.Rows)
    {
        results.Add(new Employee(
            dr["FirstName"].ToString(),
            dr["LastName"].ToString(),
            Convert.ToDateTime(dr["BirthDate"])
        ));
    }

    return results;
}
```

Is our code Refactored enough?

Yes! - “Our Code is not tightly coupled with any specific connection. Instead it relies on Abstraction i.e. IDbConnection.”

If user wants to use “Oracle” or any other database instead of “SQL Server”?

“Create another Connection Factory for that database.”

OracleDbProviderFactory

```
public class OracleProviderFactory: IDbProviderFactory
{
    3 references
    public IDbConnection CreateConnection()
    {
        return new OracleConnection();
    }

    2 references
    public IDbCommand CreateCommand()
    {
        return new OracleCommand();
    }
}
```

Call from Façade

```
public IList<IEmployee> GetAllEmployees()
{
    IList<IEmployee> results = new List<IEmployee>();
    DataTable rawData = new DataTable();

    using (IDbConnection sqlConnection = new OracleProviderFactory().CreateConnection())
    {
        IDbCommand sqlCommand = sqlConnection.CreateCommand();
        sqlCommand.CommandText = "SELECT * FROM Employees";
        sqlCommand.CommandType = System.Data.CommandType.Text;

        sqlConnection.Open();

        IDataReader sqlDataReader = sqlCommand.ExecuteReader();
        using (sqlDataReader)
        {
            rawData.Load(sqlDataReader);
        }
    }

    foreach (DataRow dr in rawData.Rows)
    {
        results.Add(new Employee(
            dr["FirstName"].ToString(),
            dr["LastName"].ToString(),
            Convert.ToDateTime(dr["BirthDate"])
        ));
    }

    return results;
}
```

Problem with Above Code

- Violate the rule of **Abstract Factory** Pattern-
“User will not be aware about what is being created and how is being created.”
- Each time we call-
 - Either **SQLServerDbProviderFactory**
 - Or **OracleDbProviderFactory**

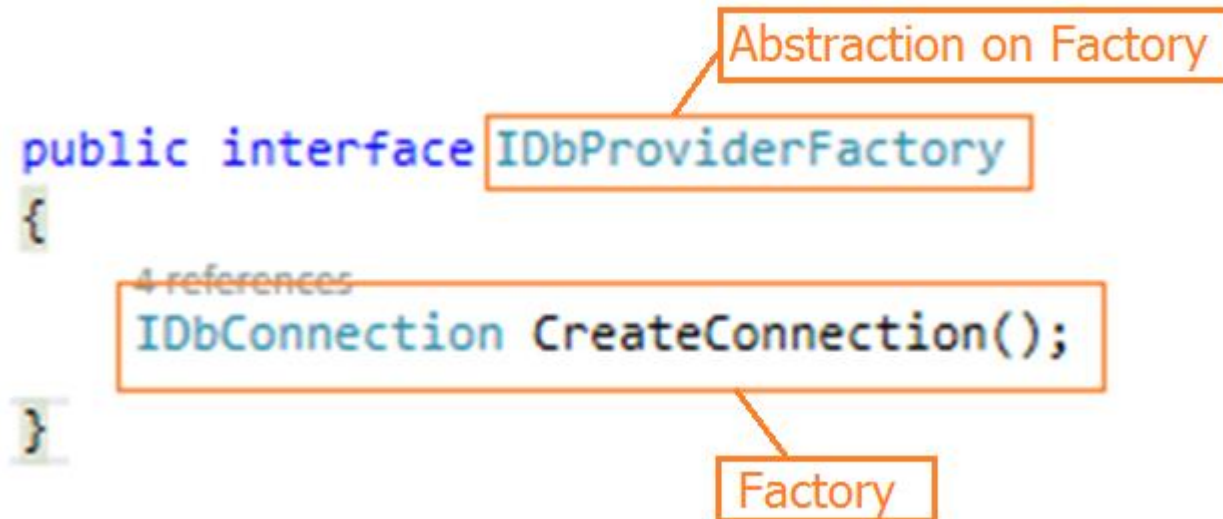
Our user knows it- which one is calling

```
using (IDbConnection sqlConnection = new OracleProviderFactory().CreateConnection())
{
    collapsed the detail
}

using (IDbConnection sqlConnection = new SqlProviderFactory().CreateConnection())
{
    Collapsed the detail
}
```

Abstract Factory?

In our Connection code we have two level of Abstraction-



- + **IDbConnection** : Create the factory of **SQL Connection** or **Oracle Connection**
- + **IDbProviderFactory**: Create the instance of **SQL Provider** or **Oracle Provider Factory**.

So, we have abstraction on Factory– Abstract Factory.

Refactor to Next Level

```
public class DbProviderFactory : IDbProviderFactory
{
    private System.Data.Common.DbProviderFactory frameworkProviderFactory;
    private ConnectionStringSettings settings;
    1 reference
    public DbProviderFactory() : this(ConfigurationManager.ConnectionStrings[1])
    {
    }
}
1 reference
public DbProviderFactory(ConnectionStringSettings settings)
{
    frameworkProviderFactory = DbProviderFactories.GetFactory(settings.ProviderName);
    this.settings = settings;
}
5 references
public IDbConnection CreateConnection()
{
    IDbConnection connection = frameworkProviderFactory.CreateConnection();
    connection.ConnectionString = settings.ConnectionString;
    return connection;
}
}
<add name="SqlDatabaseConnection"
    connectionString="data source=(local);Integrated Security=SSPI;Initial Catalog=DesignPatterns"
    providerName="System.Data.SqlClient"
/>
```

+. Get the Provider Name from Configuration file no matter weather it is SQL client or Oracle client. All other code are same. So we don't need to create individual factory class for each database provider

Consume the code from Façade

```
public class EmployeeService : IEmployeeService
{
    private IDbProviderFactory providerFactory;
    1 reference | 1/1 passing
    public EmployeeService()
    {
        this.providerFactory = new DbProviderFactory();
    }
    13 references | 1/7 passing
    public IList<IEmployee> GetAllEmployees()
    {
        IList<IEmployee> results = new List<IEmployee>();
        DataTable rawData = new DataTable();
        using (IDbConnection sqlConnection = providerFactory.CreateConnection())
        {
            IDbCommand sqlCommand = sqlConnection.CreateCommand();
            sqlCommand.CommandText = "SELECT * FROM Employees";
            sqlCommand.CommandType = System.Data.CommandType.Text;
            sqlConnection.Open();
            IDataReader sqlDataReader = sqlCommand.ExecuteReader();
            using (sqlDataReader)
            {
                rawData.Load(sqlDataReader);
            }
        }
        foreach (DataRow dr in rawData.Rows)
        {
            results.Add(new Employee(
                dr["FirstName"].ToString(),
                dr["LastName"].ToString(),
                Convert.ToDateTime(dr["BirthDate"])
            ));
        }
        return results;
    }
}
```

+. Created the instance of **ProviderFactory** and used it to **CreateConnection**. So end user doesn't know about which provider we are being using.

But.... Problem Here

- + Each time we call **EmployeeService** it creates an instance of **DbProviderFactory** class which is wastage.
- + So we have to Avoid this Wastage.

Introducing Singleton instance of Factory

```
public class DbProviderFactory: IDbProviderFactory
{
    public static readonly IDbProviderFactory Instance = new DbProviderFactory(ConfigurationManager.ConnectionStrings[1]);
    private System.Data.Common.DbProviderFactory frameworkProviderFactory;
    private ConnectionStringSettings settings;

    1 reference
    public DbProviderFactory(ConnectionStringSettings settings)
    {
        frameworkProviderFactory = DbProviderFactories.GetFactory(settings.ProviderName);
        this.settings = settings;
    }

    3 references
    public IDbConnection CreateConnection()
    {
        IDbConnection connection = frameworkProviderFactory.CreateConnection();
        connection.ConnectionString = settings.ConnectionString;
        return connection;
    }
}
```

Create the **Singleton Instance** of **DbProviderFactory** class. So only one instance of **DbProviderFactory** will use instead of creating multiple instances.

Note: We are not using one connection for the application. We are actually using one factory for each application

Call from Façade – Dependency Injection

```
public class V4EmployeeService:IEmployeeService
```

```
{
```

```
    private IDbProviderFactory providerFactory;
```

```
    1 reference | 0/1 passing
```

```
    public EmployeeService() :this( DbProviderFactory.Instance ) { }
```

```
    2 references | 0/1 passing
```

```
    public EmployeeService( IDbProviderFactory providerFactory)
```

```
    {
```

```
        this.providerFactory = providerFactory;
```

```
    }
```

```
    13 references | 1/7 passing
```

```
    public IList<IEmployee> GetAllEmployees()
```

```
    {
```

```
        IList<IEmployee> results = new List<IEmployee>();
```

```
        DataTable rawData = new DataTable();
```

```
        using (IDbConnection sqlConnection = providerFactory.CreateConnection())
```

```
        {
```

```
            IDbCommand sqlCommand = sqlConnection.CreateCommand();
```

```
            sqlCommand.CommandText = "SELECT * FROM Employees";
```

```
            sqlCommand.CommandType = System.Data.CommandType.Text;
```

```
            sqlConnection.Open();
```

```
            IDataReader sqlDataReader = sqlCommand.ExecuteReader();
```

```
            using (sqlDataReader)
```

```
            {
```

```
                rawData.Load(sqlDataReader);
```

```
            }
```

```
        }
```

```
        foreach (DataRow dr in rawData.Rows)
```

```
        {
```

```
            results.Add(new Employee(
```

```
                dr["FirstName"].ToString(),
```

```
                dr["LastName"].ToString(),
```

```
                Convert.ToDateTime(dr["BirthDate"])
```

```
            ));
```

```
        }
```

```
        return results;
```

```
    }
```

Handled dependency injection by passing the Singleton Instance of DbProviderFactory in poor man style.

Inject dependency through Constructor

- In Above Code we are used Dependency Injection Pattern by injecting **IDbProviderFactory** through the Constructor of **EmployeeService** Class.
- Though we are not using any **loc container** here so we handled the dependency in **Poor man Style**

There are some problem of using Singleton in above style

- Our above Singleton is not Thread Safe
- Also the Singleton instance is Immutable

So we have to Refactor our above code-

Refactor to Monostate Pattern

```
public class MonostateDbProviderFactory : IDbProviderFactory
{
    private static System.Data.Common.DbProviderFactory frameworkProviderFactory;

    0 references
    static MonostateDbProviderFactory()
    {
        frameworkProviderFactory = DbProviderFactories.GetFactory("System.Data.SqlClient");
    }

    2 references
    public IDbCommand CreateCommand()
    {
        return frameworkProviderFactory.CreateCommand();
    }

    3 references
    public IDbConnection CreateConnection()
    {
        IDbConnection connection = frameworkProviderFactory.CreateConnection();
        connection.ConnectionString = ConfigurationManager.ConnectionStrings[1].ToString();
        return connection;
    }
}
```

- In Monostate Pattern we have **Private Static** instance of DbProviderFactory instead of **Public Static** instance. We also have a **static Constructor**.
- The only difference between **Monostate** and **Singleton** is in **Monostate** client doesn't realize that they are working with a Single Instance

Refactor to Monostate Pattern

- In Monostate Pattern we have **Private Static** instance of **DbProviderFactory** instead of **Public Static** instance. We also have a **static Constructor**.
- Monostate Design Pattern allows us to create instance as much as required. But though **DbProviderFactory** instance static so it get initialized only once.
- The Difference between **Singleton** and **Monostate** is
 - The Singleton is the single instance
 - Monostate has multiple instance but internal instance is limited to only one.
- In **Monostate** client doesn't realize that they are working with a Single Instance.
- In **MonostateProviderFactory** all methods are mutable. So they will return completely brand new instance to the client and they can be used very safely in Multi-Threaded environment.

Strategy Design Pattern

- Clients needs-
 - Employee list needs to shown in sorted by their birth date

So we write our code for this client requirement-

```
public IList<IEmployee> EmployeeSortByBirthDate()  
{  
    List<IEmployee> employees = this.GetAllEmployees().ToList();  
    return employees.OrderBy(x => x.BirthDate).ToList();  
}
```

Our Unit Test code is

```
[TestMethod]  
[0 references]  
public void GetAllEmployees_SortByBirthDate()  
{  
    IEmpService empService = new VSEmployeeService();  
    IList<IEmployee> employees = empService.EmployeeSortByBirthDate();  
    foreach (var emp in employees)  
    {  
        System.Diagnostics.Debug.WriteLine(emp.FirstName + ", " + emp.LastName + ", " + emp.BirthDate.ToString());  
    }  
}
```

Wow! Our unit test passed and our code works properly-

Strategy Design Pattern- Contd.

After few days Client has another requirements which is-

“In another place Employee List should be seen sorted by their First Name”

Now what will we do?

- Create another method for Sorting by their First Name.
- Or introduce a if condition for sorting by First Name

```
//Create individual method for each sorting logic
2 references | 1/1 passing
public IList<IEmployee> EmployeeSortByBirthDate()
{
    List<IEmployee> employees = this.GetAllEmployees().ToList();
    return employees.OrderBy(x => x.BirthDate).ToList();
}

0 references
public IList<IEmployee> SortByFirstName()
{
    List<IEmployee> employees = this.GetAllEmployees().ToList();
    return employees.OrderBy(x => x.FirstName).ToList();
}
```

```
//Place separate if condition for each individual sorting logic
1 reference
public IList<IEmployee> SortingBy(string sortby)
{
    List<IEmployee> employees = this.GetAllEmployees().ToList();
    if (sortby.CompareTo("FirstName") == 0)
    {
        return employees.OrderBy(x => x.FirstName).ToList();
    }
    else
    {
        return employees.OrderBy(x => x.BirthDate).ToList();
    }
}
```

Strategy Design Pattern- Contd.

Few days later user has another requirement to show
“Employee List by sorting their Last Name”

We have to follow one of above two processes i.e. -

- So Create another method for Sorting by their Last Name.
- Or introduce a if condition for sorting by Last Name

+ As day by day client requirements increases and our code also increases. So after few days our class will become bulky and completely loss it's maintainability.

+ Also it violates the rules of –

1. Open Closed Principle (OCP):

*+>. Either modify our class (introducing a brand new method for each logic i.e. **SortByDate**, **SortByName** etc.)*

*+>. Or modify existing method by adding new condition e.g.- condition for **sortByDate**, **sortByName** etc.*

2. Don't Repeat Yourself (DRY) : Same logic repeat again and again

Refactoring to Strategy Pattern

```
4 references
public interface IEmpService
{
    4 references | 0/1 passing
    IList<IEmployee> GetAllEmployees();
    3 references | 1/1 passing
    IList<IEmployee> EmployeeSortUsing(IComparer<IEmployee> sortStrategy);
}

2 references | 1/1 passing
public IList<IEmployee> EmployeeSortUsing(IComparer<IEmployee> sortStrategy)
{
    List<IEmployee> employees = this.GetAllEmployees().ToList();
    employees.Sort(sortStrategy);
    return new ReadOnlyCollection<IEmployee>(employees);
}
```

Create a method that will get
IComparer<IEmployee> as Parameter

Now create a class that will Compare two employee objects by their birth date and consume it from client-

```
public class EmployeeBirthDateComparror : IComparer<IEmployee>
{
    5 references
    public int Compare(IEmployee emp1, IEmployee emp2)
    {
        return emp1.BirthDate.CompareTo(emp2.BirthDate);
    }
}

[TestMethod]
0 references
public void GetAllEmployees_ShouldGetAllEmployeeFromDB_InSorted_Order()
{
    IEmpService empService = new V5EmployeeService();
    IComparer<IEmployee> sortByDate = new RefactoringToPatterns.Service.V5.EmployeeBirthDateComparror();
    IList<IEmployee> sortedEmployee = empService.EmployeeSortUsing(sortByDate);
    foreach (var emp in sortedEmployee)
    {
        System.Diagnostics.Debug.WriteLine(emp.FirstName + ", " + emp.LastName + ", " + emp.BirthDate.ToString());
    }
}
```

Refactoring to Strategy Pattern

Now for First Name sorting we will create another class

```
public class EmployeeFirstNameComparator : IComparer<IEmployee>
{
    5 references
    public int Compare(IEmployee emp1, IEmployee emp2)
    {
        return emp1.FirstName.CompareTo(emp2.FirstName);
    }
}
```

For Last Name Sorting we will follow the same process i.e. create another class

```
public class EmployeeLastNameComparator : IComparer<IEmployee>
{
    5 references
    public int Compare(IEmployee emp1, IEmployee emp2)
    {
        return emp1.LastName.CompareTo(emp2.LastName);
    }
}
```

Refactoring to Strategy Pattern

No modification will require for our Sorting method. Also don't need to introduce new method for each sorting logic-

Our end user will just consume the method by passing individual **Icomparer<Iemployee>** as parameter value and return the output according to sorting logic as follows-

```
[TestMethod]
0 references
public void GetAllEmployees_ShouldGetAllEmployeeFromDB_InSorted_Order()
{
    IEmpService empService = new VSEmployeeService();
    IComparer<IEmployee> sortByDate = new RefactoringToPatterns.Service.V5.EmployeeBirthDateComparror();

    IList<IEmployee> sortedEmployee = empService.EmployeeSortUsing(sortByDate);

    foreach (var emp in sortedEmployee)
    {
        System.Diagnostics.Debug.WriteLine(emp.FirstName + ", " + emp.LastName + ", " + emp.BirthDate.ToString());
    }
}
```

Same above process should follows for sortByName

+. This pattern is known as “**Strategy Pattern**” because it interchanges the **Compare** method of different class based on client need.

+. This is also known as **Replace Conditionals with Polymorphism (RCP/RIP)** pattern because instead of using if conditions we just used polymorphism logic to override the compare method in different classes.

Gateway Design Pattern

- Till now we write all database access logic into Service Layer.

```
public IList<IEmployee> GetAllEmployees()
{
    IList<IEmployee> results = new List<IEmployee>();

    DataTable rawData = new DataTable();

    using (IDbConnection sqlConnection = providerFactory.CreateConnection())
    {
        IDbCommand sqlCommand = sqlConnection.CreateCommand();
        sqlCommand.CommandText = "SELECT * FROM Employees";
        sqlCommand.CommandType = System.Data.CommandType.Text;

        sqlConnection.Open();

        IDataReader sqlDataReader = sqlCommand.ExecuteReader();
        using (sqlDataReader)
        {
            rawData.Load(sqlDataReader);
        }

        foreach (DataRow dr in rawData.Rows)
        {
            results.Add(new Employee(
                dr["FirstName"].ToString(),
                dr["LastName"].ToString(),
                Convert.ToDateTime(dr["BirthDate"])
            ));
        }

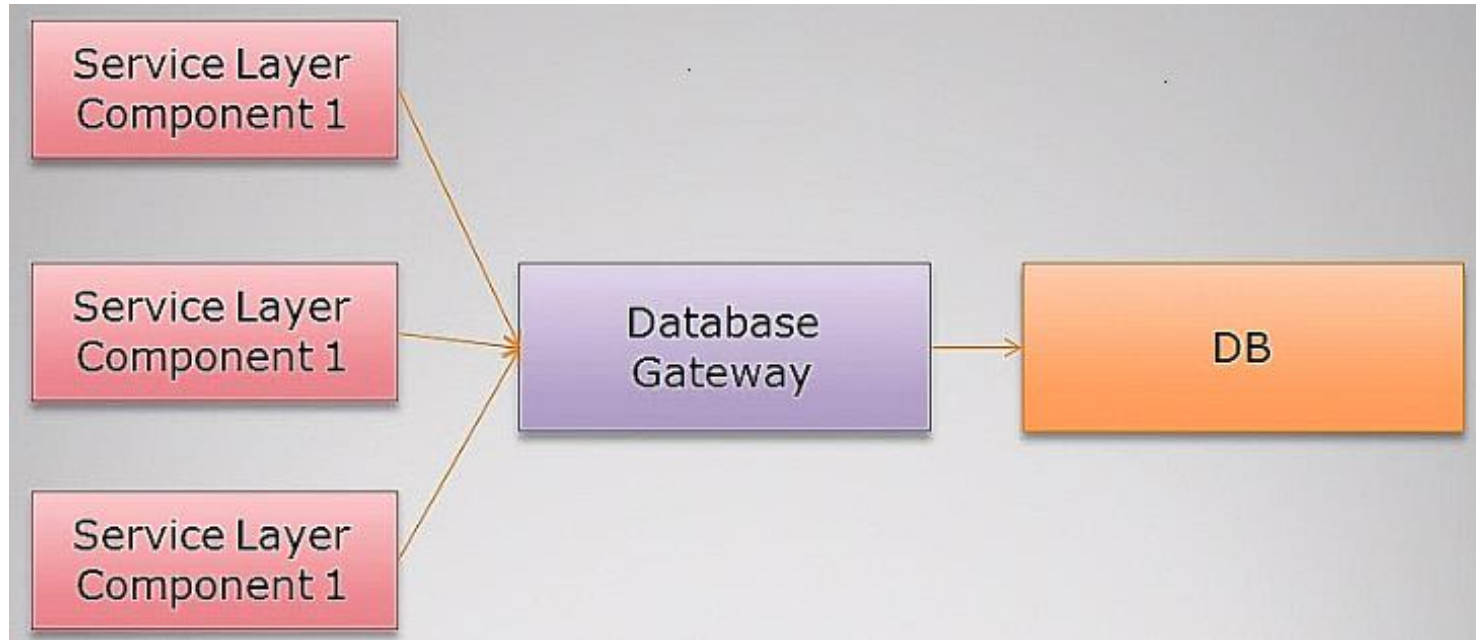
        return results;
    }
}
```

- If we have multiple service class then we have to write all the database access logic all those Service classes.
- It's not any good practice. The good practices is-

“We’ll create a single access point to database so that all service layer components can access database by using that access point.”

Refactor to Gateway Design Pattern

This access point is our Gateway



Now our Service Layer will contact with database via Gateway.

Refactor to Gateway Design Pattern

Source code of our access point-

```
public interface IDbGateway
{
    4 references
    int ExecuteCommand(string sqlCommandString);
    2 references
    DataTable ExecuteQuery(string selectStatement);
    3 references
    IDbTransaction DbTransaction { get; }
}
```

```
all 10 tests passed 0 failures
public class DbGateway : IDbGateway
{
    private readonly IDbProviderFactory dbProviderFactory;
    private readonly IDbConnection dbConnection;
    private readonly IDbTransaction dbTransaction;
    4 references | 3/4 passing
    public DbGateway(IDbProviderFactory dbProviderFactory)
    {
        this.dbProviderFactory = dbProviderFactory;
        this.dbConnection = dbProviderFactory.CreateConnection();
        this.dbTransaction = dbConnection.BeginTransaction();
    }
    2 references
    private IDbCommand CreateCommand()
    {
        IDbCommand dbCommand = dbConnection.CreateCommand();
        dbCommand.Transaction = DbTransaction;
        return dbCommand;
    }
    4 references
    public IDbTransaction DbTransaction
    {
        get
        {
            return dbTransaction;
        }
    }
    5 references
    public int ExecuteCommand(string sqlCommandString)
    {
        // IDbCommand command = dbConnection.CreateCommand();
        // command.Transaction = this.DbTransaction;

        IDbCommand command = CreateCommand();
        command.CommandType = CommandType.Text;
        command.CommandText = sqlCommandString;

        return command.ExecuteNonQuery();
    }
    3 references
    public DataTable ExecuteQuery(string selectStatement)
    {
        DataTable rawData = new DataTable();
        IDbCommand dbCommand = CreateCommand();
        dbCommand.CommandText = selectStatement;
        dbCommand.CommandType = CommandType.Text;
        using (IDataReader reader = dbCommand.ExecuteReader())
        {
            rawData.Load(reader);
        }
        return rawData;
    }
}
```


Refactor to Gateway Design Pattern

Access Gateway from Service Layer

```
public interface IEmployeeGateWayService
{
    2 references | 1/1 passing
    void SaveEmployee(Employee employee);
    2 references | 1/1 passing
    DataTable GetAllEmployee();
}
```

3 references

```
public class EmployeeGateWayService:IEmployeeGateWayService
```

```
{
    protected readonly IDbGateway dbGateWay;
    2 references | 2/2 passing
    public EmployeeGateWayService(IDbGateway dbGateWay)
    {
        this.dbGateWay = dbGateWay;
    }

    2 references | 1/1 passing
    public void SaveEmployee(Employee employee)
    {
        string sqlString = "INSERT INTO Employees (LastName, FirstName, BirthDate) "+
            "VALUES ('" + employee.LastName + "','" + employee.FirstName + "','" + employee.BirthDate + "')";
        dbGateWay.ExecuteCommand(sqlString);
        dbGateWay.DbTransaction.Commit();
    }

    2 references | 1/1 passing
    public DataTable GetAllEmployee()
    {
        return dbGateWay.ExecuteQuery("SELECT * FROM EMPLOYEES");
    }
}
```

Repository & Unit Of Work Design Pattern

According to Eric Evans-

“A repository will connect Factories with Gateways.”

[“Factory is the convenient way to create objects”]

- + . Here Factory means instance of Domain Model
- + . Gateway is the access point that we discussed earlier

So we will introduce Repository pattern as the connector between Domain Model and Gateway via Service Layer.

Repository & Unit Of Work Design Pattern

```
public interface IRepository<T> where T:class
{
    3 references
    void Save(string saveSqlString);
    1 reference
    void Update(string updateSqlString);
    2 references
    void Delete(string deleteSqlString);
    2 references
    DataTable Get(string sqlStatement);
}
```

```
public class BaseRepository<T> : IRepository<T> where T:class, new()
{
    private readonly IDbGateway dbGateway;
    2 references | 1/2 passing
    public BaseRepository(IDbGateway dbGateway)
    {
        this.dbGateway = dbGateway;
    }

    3 references
    public void Save(string saveSqlString)
    {
        dbGateway.ExecuteCommand(saveSqlString);
    }

    1 reference
    public void Update(string updateSqlString)
    {
        dbGateway.ExecuteCommand(updateSqlString);
    }

    2 references
    public void Delete(string deleteSqlString)
    {
        dbGateway.ExecuteCommand(deleteSqlString);
    }

    2 references
    public DataTable Get(string sqlStatement)
    {
        return dbGateway.ExecuteQuery(sqlStatement);
    }
}
```

Repository & Unit Of Work Design Pattern

Unit of Work uses to Commit or Rollback one or more operations as a Single Transaction.

```
public interface IUnitOfWork
{
    4 references
    void Commit();
    4 references
    void Rollback();
}
```

```
public class UnitOfWork : IUnitOfWork
{
    private IDbGateway dbGateway;
    private IDbTransaction dbTransaction;
    2 references | 1/2 passing
    public UnitOfWork(IDbGateway dbGateway)
    {
        this.dbGateway = dbGateway;
        dbTransaction = dbGateway.DbTransaction;
    }

    4 references
    public void Commit()
    {
        dbTransaction.Commit();
        dbTransaction = null;
    }

    4 references
    public void Rollback()
    {
        dbTransaction.Rollback();
        dbTransaction = null;
    }
}
```

Repository & Unit Of Work Design Pattern

Use Repository and & Unit of work in Service Layer-

```
public class EmployeeService : IEmpService
{
    protected readonly IRepository<Employee> employeeRepository;
    protected readonly IUnitOfWork unitOfWork;
    2 references | 1/2 passing
    public EmployeeService(IRepository<Employee> employeeRepository, IUnitOfWork unitOfWork)
    {
        this.employeeRepository = employeeRepository;
        this.unitOfWork = unitOfWork;
    }
    1 reference
    public void DeleteEmployee(int id) ...
    2 references | 1/1 passing
    public DataTable GetEmployee() ...
    1 reference
    public void SaveEmployee(IEmployee employee) ...
    2 references | 0/1 passing
    public void SaveEmployees(List<Employee> employees) ...
    1 reference
    public void UpdateEmployee(IEmployee employee)
    {
        throw new NotImplementedException();
    }
}
```

Repository & Unit Of Work Design Pattern

Use Repository and & Unit of work in Service Layer-

```
public void SaveEmployees(List<Employee> employees)
{
    try
    {
        foreach (var emp in employees)
        {
            string sqlString = "INSERT INTO Employees (LastName, FirstName, BirthDate) "+
                               "VALUES ('" + emp.LastName + "','" + emp.FirstName + "','" + emp.BirthDate + "')";
            employeeRepository.Save(sqlString);
        }
        unitOfWork.Commit();
        System.Diagnostics.Debug.WriteLine("Saved Successfully");
    }
    catch (Exception ex)
    {
        unitOfWork.Rollback();
        //throw new Exception("Failed To Save Data");
        throw ex;
    }
}

public void SaveEmployee(IEmployee employee)
{
    try
    {
        string sqlString = "INSERT INTO Employees (LastName, FirstName, BirthDate) "+
                           "VALUES ('" + employee.LastName + "','" + employee.FirstName + "','" + employee.BirthDate + "')";
        employeeRepository.Save(sqlString);
        unitOfWork.Commit();
        System.Diagnostics.Debug.WriteLine("Saved Successfully");
    }
    catch (Exception ex)
    {
        unitOfWork.Rollback();
        //throw new Exception("Failed To Save Data");
        throw ex;
    }
}
```

Test Read Statement

✓ Test Passed - Employee_Should_Be_Shown_From_Database

Elapsed time: 279 ms

Run | Debug

TestMethod]

0 references

```
public void Employee_Should_Be_Shown_From_Database()
```

```
{  
    IDbProviderFactory dbProviderFactory = new MonostateDbProviderFactory();  
    IDbGateway dbGateway = new DbGateway(dbProviderFactory);  
    IRepository<Employee> employeeRepository = new BaseRepository<Employee>(dbGateway);  
    IUnitOfWork unitOfWork = new UnitOfWork(dbGateway);  
    IEmpService service = new EmployeeService(employeeRepository, unitOfWork);  
  
    var employess = service.GetEmployee();  
    Assert.IsTrue(employess.Rows.Count > 0);  
}
```

Test Single Save Statement

✓ Test Passed - Single_Employee_Should_Insert_Into_Database

Elapsed time: 20 ms

Output

Run | Debug

estmethod]

✓ 0 references

public void Single_Employee_Should_Insert_Into_Database()

```
{  
    IDbProviderFactory dbProviderFactory = new MonostateDbProviderFactory();  
    IDbGateway dbGateway = new DbGateway(dbProviderFactory);  
    IRepository<Employee> employeeRepository = new BaseRepository<Employee>(dbGateway);  
    IUnitOfWork unitOfWork = new UnitOfWork(dbGateway);  
    IEmpService service = new EmployeeService(employeeRepository, unitOfWork);  
  
    IEmployee emp = new Employee("First Name", "Last Name", new DateTime(1990, 08, 08));  
  
    service.SaveEmployee(emp);  
}
```

	EmployeeID	LastName	FirstName	Title	TitleOfCourtesy	BirthDate
1	1	Davolio	Nancy	Sales Representative	Ms.	1948-12-08 00:00:00.000
2	2	Fuller	Andrew	Vice President, Sales	Dr.	1952-02-19 00:00:00.000
3	3	Leverling	Janet	Sales Representative	Ms.	1963-08-30 00:00:00.000
4	4	Peacock	Margaret	Sales Representative	Mrs.	1937-09-19 00:00:00.000
5	5	Buchanan	Steven	Sales Manager	Mr.	1955-03-04 00:00:00.000
6	6	Suyama	Michael	Sales Representative	Mr.	1963-07-02 00:00:00.000
7	7	King	Robert	Sales Representative	Mr.	1960-05-29 00:00:00.000
8	8	Callahan	Laura	Inside Sales Coordinator	Ms.	1958-01-09 00:00:00.000
9	9	Dodsworth	Anne	Sales Representative	Ms.	1966-01-27 00:00:00.000
10	11	Last Name	First Name	NULL	NULL	1990-08-08 00:00:00.000

New row inserted to Database

Test Multiple Save Statement- Failed Case

[TestMethod]

✖ | 0 references

```
public void Group_of_Employee_Should_Insert_Into_Database()
{
    IDbProviderFactory dbProviderFactory = new MonostateDbProviderFactory();
    IDbGateway dbGateway = new DbGateway(dbProviderFactory);
    IRepository<Employee> employeeRepository = new BaseRepository<Employee>(dbGateway);
    IUnitOfWork unitOfWork = new UnitOfWork(dbGateway);
    IEmpService service = new EmployeeService(employeeRepository, unitOfWork);

    List<Employee> rightEmployees = this.EmployeesRightData();
    List<Employee> wrongEmployees = this.EmployeesWrongData();

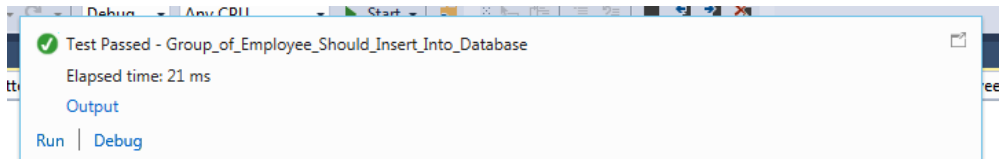
    service.SaveEmployees(wrongEmployees);
}
```

1 reference | ✖ 0/1 passing

```
private List<Employee> EmployeesWrongData()
{
    List<Employee> employees = new List<Employee>();
    employees.Add(new Employee("Name1", "Name1", new DateTime(1990, 8, 6)));
    employees.Add(new Employee("Name2", "Name2", new DateTime(1980, 8, 7)));
    employees.Add(new Employee("Name3", "Name3", new DateTime(1970, 8, 8)));
    employees.Add(new Employee("Name4", "Name4", new DateTime(1960, 8, 9)));
    employees.Add(new Employee("Name5", "Name5", new DateTime(1950, 8, 10)));
    employees.Add(new Employee("Name6", "Name6", new DateTime(1940, 8, 11)));
    employees.Add(new Employee("Name7", "Name7", new DateTime(1930, 8, 12)));
    employees.Add(new Employee("Name8", "Name8", new DateTime(1920, 32, 33)));
    return employees;
}
```

We have a wrong datetime value in a emp object. All others are correct but none save as Unit Of Work treat all statements as a Single Transaction

Test Multiple Save Statement- Passed Case



Test Passed - Group_of_Employee_Should_Insert_Into_Database
Elapsed time: 21 ms
Output
Run | Debug

```
TestMethod]
0 references
public void Group_of_Employee_Should_Insert_Into_Database()
{
    IDbProviderFactory dbProviderFactory = new MonostateDbProviderFactory();
    IDbGateway dbGateway = new DbGateway(dbProviderFactory);
    IRepository<Employee> employeeRepository = new BaseRepository<Employee>(dbGateway);
    IUnitOfWork unitOfWork = new UnitOfWork(dbGateway);
    IEmpService service = new EmployeeService(employeeRepository, unitOfWork);

    List<Employee> rightEmployees = this.EmployeesRightData();
    //List<Employee> wrongEmployees = this.EmployeesWrongData();

    service.SaveEmployees(rightEmployees);
}

1 reference | 1/1 passing
private List<Employee> EmployeesRightData()
{
    List<Employee> employees = new List<Employee>();
    employees.Add(new Employee("Name1", "Name1", new DateTime(1990, 8, 6)));
    employees.Add(new Employee("Name2", "Name2", new DateTime(1980, 8, 7)));
    employees.Add(new Employee("Name3", "Name3", new DateTime(1970, 8, 8)));
    employees.Add(new Employee("Name4", "Name4", new DateTime(1960, 8, 9)));
    employees.Add(new Employee("Name5", "Name5", new DateTime(1950, 8, 10)));
    employees.Add(new Employee("Name6", "Name6", new DateTime(1940, 8, 11)));
    employees.Add(new Employee("Name7", "Name7", new DateTime(1930, 8, 12)));
    employees.Add(new Employee("Name8", "Name8", new DateTime(1920, 8, 12)));
    return employees;
}
```

EmployeeID	LastName	FirstName	Title	TitleOfCourtesy	BirthDate
12	Name1	Name1	NULL	NULL	1990-06-08 00:00:00.000
13	Name2	Name2	NULL	NULL	1980-07-08 00:00:00.000
14	Name3	Name3	NULL	NULL	1970-08-08 00:00:00.000
15	Name4	Name4	NULL	NULL	1960-09-08 00:00:00.000
16	Name5	Name5	NULL	NULL	1950-10-08 00:00:00.000
17	Name6	Name6	NULL	NULL	1940-11-08 00:00:00.000
18	Name7	Name7	NULL	NULL	1930-12-08 00:00:00.000
19	Name8	Name8	NULL	NULL	1920-12-08 00:00:00.000