

## Design Patterns

### Part 4

1. Mediator: When you have two classes that are in need of each other, you might create a strong coupling between them. In other words, a simple change in one class can affect the other. This is not a good thing since one of the core concepts in object oriented programming is to reduce the coupling between classes. Doing so will reduce the amount of changes to the application when one of the classes does change. Having a mediator between the two classes gives us the solution we are looking for. I can ask the mediator to send some information to another class. The mediator takes care of the details. That means I do not have to know much about the other class. My only interface is the mediator. To illustrate this concept clearly, let's create an example where the application class is sending a message to the database class. This is an easy concept to digest since we all know that there is at least one layer between the application layer and the database layer. We start first by creating the Application class and the Database class. Both of those classes inherit from the Layer class as see here:

```
abstract class Layer
{
    protected Mediator mediator;

    public Layer(Mediator m)
    {
        mediator = m;
    }

    public abstract void Send(string message);
    public abstract void Notify(string message);
}

class Application : Layer
{
    public Application(Mediator mediator)
        : base(mediator)
    { }

    public override void Send(string message)
    {
        mediator.SendMessage(message, this);
    }

    public override void Notify(string message)
    {
        Console.WriteLine("Application gets the message: {0}",
```

```

        message);
    }
}

class Database : Layer
{
    public Database(Mediator mediator)
        : base(mediator)
    { }

    public override void Send(string message)
    {
        mediator.SendMessage(message, this);
    }

    public override void Notify(string message)
    {
        Console.WriteLine("Database gets the message: {0}",
            message);
    }
}

```

We have not created the Mediator class yet, but at least we know that the Layer class has two methods to send message and notify the sender. These abstract methods are implemented in the Application and Database classes. Notice how sending the message is not happening directly, but instead it is using the mediator to do so. This is the key in this pattern. We are using the 'this' keyword as a second parameter to know which class is sending the message.

Now, let's look at the Mediator and ConcreteMediator classes:

```

abstract class Mediator
{
    public abstract void SendMessage(
        string message, Layer layer);
}

class ConcreteMediator : Mediator
{
    private Application application;
    private Database database;

    public Application Application
    {
        set { application = value; }
    }

    public Database Database
    {
        set { database = value; }
    }

    public override void SendMessage(string message, Layer layer)

```

```

    {
        if (layer is Application)
            database.Notify(message);
        else if (layer is Database)
            application.Notify(message);
    }
}

```

Since the mediator needs to know which classes it is mediating, we need to have instances for the Application class and the Database class. Then inside the SendMessage() method, we need to determine the type of layer to be able to send the correct notification.

Now that all the classes have been created, we can look at the main() method to see how the calls are taking place:

```

static void Main(string[] args)
{
    ConcreteMediator concreteMediator = new ConcreteMediator();

    Application application = new Application(concreteMediator);
    Database database = new Database(concreteMediator);

    concreteMediator.Application = application;
    concreteMediator.Database = database;

    application.Send("Update the Employee table");
    database.Send("Data changed; Update your window");

    Console.ReadLine();
}

```

We create instances of the three classes. We pass the mediator to the Application and the Database. We also used properties to set the application and database for the mediator. Finally, we send the messages. Here is the output:

```

Database gets the message: Update the Employee table
Application gets the message: Data changed; Update your window

```

2. Observer: When developing applications, you most likely ran into cases when a change in the data store (database, xml file...) needs to be reflected in the UI. Say you have added a new account type and you need to update the account type drop down list. Or, when you are writing a game application, all the soldiers on the screen need to be aware of an enemy appearing from behind a tank. These examples have something in

common: dependency. When one object changes state, other objects need to be notified to take action. This is the purpose of the observer pattern. As the name implies, an observer is an object "watching" other objects. As soon as these objects change somehow, the observer alerts the objects in its list about the change. The example we are going to tackle is about a list of employees that need to be notified when a meeting changes. So, let's start by creating the Employee class and its interface:

```
interface IEmployee
{
    void Update(Meeting meeting);
}

class Employee : IEmployee
{
    private string name;
    private Meeting meeting;

    public Employee(string name)
    {
        this.name = name;
    }

    public Meeting Meeting
    {
        get { return meeting; }
        set { meeting = value; }
    }

    public void Update(Meeting meeting)
    {
        Console.WriteLine("{0} has been notified about the meeting\n {1}", name, meeting);
    }
}
```

As you can see, the Employee class has a Meeting class as a member (this will be declared soon). The Update method will be called every time there is an update.

Here is the declaration for the Meeting class:

```
class Meeting
{
    private string subject;
    private DateTime when;
    private List<IEmployee> employees = new List<IEmployee>();

    public Meeting(string subject, DateTime when)
```

```

    {
        this.subject = subject;
        this.when = when;
    }

    public DateTime When
    {
        get { return when; }
        set
        {
            if (when != value)
            {
                when = value;
                Notify();
            }
        }
    }

    public void Attach(IEmployee employee)
    {
        employees.Add(employee);
    }

    public void Detach(IEmployee employee)
    {
        employees.Remove(employee);
    }

    public void Notify()
    {
        foreach (IEmployee employee in employees)
            employee.Update(this);
        Console.WriteLine();
    }

    public override string ToString()
    {
        return string.Format("Subject: {0}, When: {1}", subject, when);
    }
}

class BudgetMeeting : Meeting
{
    public BudgetMeeting(string subject, DateTime when)
        : base(subject, when)
    { }
}

```

The key point here is that the Meeting class has a list of employees that can be notified of the change. The When property, upon a change (set), will notify this list. We can add and remove employees to be notified using the methods Attach() and Detach(). The method Notify() will loop over the list and call the Update method. The BudgetMeeting class is nothing more than a concrete meeting.

Let's see how we can use these classes:

```
static void Main(string[] args)
{
    BudgetMeeting meeting =
        new BudgetMeeting("Budget Discussion", DateTime.Now);
    meeting.Attach(new Employee("James"));
    meeting.Attach(new Employee("Lisa"));

    meeting.When = DateTime.Now.AddDays(2);
    meeting.When = DateTime.Now.AddDays(7);
    meeting.When = DateTime.Now.AddDays(12);

    Console.ReadLine();
}
```

You notice that we create a meeting, attach some employees, and change the time using When. Here is how the output will look like:

```
James has been notified about the meeting
Subject: Budget Discussion, When: 12/16/2010 5:23:34 PM
Lisa has been notified about the meeting
Subject: Budget Discussion, When: 12/16/2010 5:23:34 PM

James has been notified about the meeting
Subject: Budget Discussion, When: 12/21/2010 5:23:34 PM
Lisa has been notified about the meeting
Subject: Budget Discussion, When: 12/21/2010 5:23:34 PM

James has been notified about the meeting
Subject: Budget Discussion, When: 12/26/2010 5:23:34 PM
Lisa has been notified about the meeting
Subject: Budget Discussion, When: 12/26/2010 5:23:34 PM
```

3. Strategy: In this design pattern, you are looking at many algorithms to solve a certain problem. Any of these algorithms is a standalone solution that can be easily pluggable to get the job done. We can exchange any of these algorithms with minimal or no impact on the client that uses it. Let's take, for example, the case of displaying students' grades. We have different strategies on how these grades will be displayed (we can display them in ascending order, descending order, letters or numeric, and so on...). We will implement different strategies for that. Then, we will use these strategies in different contexts to get the ultimate job of displaying the grades.

In the first set of code, we will look at the different strategies, then the context, and finally, how to put the whole thing together in the main method. Here are the different strategies:

```
abstract class GradeStrategy
{
    public abstract void DisplayGrades(List<double> grades);
}

class LowestGradeFirstStrategy : GradeStrategy
{
    public override void DisplayGrades(List<double> grades)
    {
        grades.Sort();
        grades.ForEach(g => Console.WriteLine(g + " "));
        Console.WriteLine("\n");
    }
}
```

As you can see, we start with the abstract `GradeStrategy` that defines the skeleton. Then we move into the strategy that displays the grades in the ascending order. We passed a lambda expression to the `ForEach` loop. For every `g` (a grade that is) we print its value on the console. We do not have to define what `g` is. It is understood from the context. You could simply use a `for` loop and print the values without using lambda expressions, of course. Then, we move into defining the other strategies:

```
class HighestGradeFirstStrategy : GradeStrategy
{
    public override void DisplayGrades(List<double> grades)
    {
        grades.Sort((x, y) => y.CompareTo(x));
        grades.ForEach(g => Console.WriteLine(g + " "));
        Console.WriteLine("\n");
    }
}

class LetterGradeStrategy : GradeStrategy
{
    public override void DisplayGrades(List<double> grades)
    {
        List<string> letterGrades = grades.ConvertAll(
            new Converter<double, string>(IntToGrade));
        letterGrades.Sort();
        letterGrades.ForEach(g => Console.WriteLine(g + " "));
    }
}
```

```

        Console.WriteLine("\n");
    }

    static string IntToGrade(double x)
    {
        if (x >= 90) return "A";
        else if (x >= 80) return "B";
        else if (x >= 70) return "C";
        else if (x >= 60) return "D";
        else return "F";
    }
}

class DiffFromAverageStrategy : GradeStrategy
{
    public override void DisplayGrades(List<double> grades)
    {
        double sum = grades.Sum();
        double average = sum / grades.Count;
        Console.WriteLine("Average is: {0}", average);
        grades.ForEach(g => Console.WriteLine((g - average) + " "));
        Console.WriteLine("\n");
    }
}

```

The other strategies follow the same pattern. But notice how I am using the `ConvertAll()` method that takes a method that accepts number grades and convert them to letter grades.

After defining the strategies, we need to create a context that we can use to switch between the strategies depending on our needs. Here is how this class will look like:

```

class Context
{
    private List<double> grades = new List<double>();
    private GradeStrategy strategy;

    public void SetStrategy(GradeStrategy strategy)
    {
        this.strategy = strategy;
    }

    public void Add(double grade)
    {
        grades.Add(grade);
    }

    public void DisplayGrades()
    {
        strategy.DisplayGrades(grades);
    }
}

```

```
}  
}
```

This class is simple. It contains a GradeStrategy that we can set, and a list of grades that we can populate. Calling DisplayGrades() will call the method of the appropriate strategy that we already set. Now, the main method will make use of the context and the strategies as follows:

```
static void Main(string[] args)  
{  
    Context context = new Context();  
  
    context.Add(80.5);  
    context.Add(92);  
    context.Add(78.5);  
    context.Add(75);  
    context.Add(98.75);  
    context.Add(45.25);  
    context.Add(50);  
    context.Add(64);  
    context.Add(88);  
    context.Add(30);  
    context.Add(79);  
  
    context.SetStrategy(new LowestGradeFirstStrategy());  
    context.DisplayGrades();  
    context.SetStrategy(new HighestGradeFirstStrategy());  
    context.DisplayGrades();  
    context.SetStrategy(new LetterGradeStrategy());  
    context.DisplayGrades();  
    context.SetStrategy(new DiffFromAverageStrategy());  
    context.DisplayGrades();  
  
    Console.ReadLine();  
}
```

From the main method, you can notice that core concept behind this pattern: see how we can easily switch between strategies to get the job done. Finally, the output will be as such:

```
30 45.25 50 64 75 78.5 79 80.5 88 92 98.75  
98.75 92 88 80.5 79 78.5 75 64 50 45.25 30  
A A B B C C C D F F F  
Average is: 71  
27.75 21 17 9.5 8 7.5 4 -7 -21 -25.75 -41
```

4. Visitor: In this pattern, we have an operation that needs to be performed on the elements of a certain object. The terminology here is that the first object will “accept” a visitor, and the second object “visits” the first and performs some operation. The key point to keep in mind here is that you do not need the two objects to be tightly coupled. To illustrate, we look at the example of having a list of books. This list will accept a reviewer and a publisher. So, we have a book (with different variations like programming and database and such), a collection of these books, a reviewer visitor and a publisher visitor.

We start first by creating the books:

```
abstract class Publication
{
    public abstract void Accept(IVisitor visitor);
}

class Book : Publication
{
    private string title;
    private string author;
    private decimal price;

    public Book(string title, string author, decimal price)
    {
        this.title = title;
        this.author = author;
        this.price = price;
    }

    public string Title
    {
        get { return title; }
        set { title = value; }
    }

    public string Author
    {
        get { return author; }
        set { author = value; }
    }

    public decimal Price
    {
        get { return price; }
        set { price = value; }
    }
}
```

```

    }

    public override string ToString()
    {
        return string.Format("Title: {0}, Author: {1}, Price: {2}",
            title, author, price);
    }

    public override void Accept(IVisitor visitor)
    {
        visitor.Visit(this);
    }
}

class Programming : Book
{
    public Programming() :
        base("C#", "King", 65.95m)
    {}
}

class Database : Book
{
    public Database() :
        base("SQL Server", "Johnson", 49.99m)
    {}
}

class Design : Book
{
    public Design() :
        base("Software Design", "Smith", 55.45m)
    {}
}

```

You notice that we create a Book, Programming, Database, and Design. The Book inherits from Publication, and accepts a Visitor (that we have not created yet). We passed 'this' into the Visit() method because the visitor is visiting the current class we are implementing. Everything else in the code is straightforward.

Before we look at the visitor classes, we need to create the book collection:

```

class Books
{
    private List<Book> books = new List<Book>();

    public void Attach(Book b)
    {
        books.Add(b);
    }
}

```

```

public void Detach(Book b)
{
    books.Remove(b);
}

public void Accept(IVisitor visitor)
{
    foreach (Book b in books)
    {
        b.Accept(visitor);
    }
    Console.WriteLine();
}
}

```

Here, we have added the flexibility of attaching and detaching books, and also adding the Accept() method so the visitor can visit the books in the collection one at a time.

Next, let's look at the visitor classes:

```

interface IVisitor
{
    void Visit(Publication publication);
}

class ReviewerVisitor : IVisitor
{
    public void Visit(Publication publication)
    {
        Book b = publication as Book;
        Console.WriteLine("The book {0} has been reviewed", b);
    }
}

class PublisherVisitor : IVisitor
{
    public void Visit(Publication publication)
    {
        Book b = publication as Book;
        Console.WriteLine("The book {0} has been published", b);
    }
}

```

The reviewer visits the book and simulates the review process, and the publisher visits the book and simulates the publishing process.

The main method puts them together nicely:

```
static void Main(string[] args)
{
    Books books = new Books();

    books.Attach(new Programming());
    books.Attach(new Database());
    books.Attach(new Design());

    IVisitor reviewer = new ReviewerVisitor();
    books.Accept(reviewer);
    IVisitor publisher = new PublisherVisitor();
    books.Accept(publisher);

    Console.ReadLine();
}
```

Now you can see how we attached a list of books to the books collection, and how the books collection then accepts two visitors (a reviewer and a publisher).

Conclusion: As the applications are getting more and more complex, the importance for design patterns is becoming more and more apparent. Every design pattern we experimented with in this article has its use in software development. As developers, we try to keep an eye on our applications and requirements to see where one of those design patterns would fit. Having said that, it is important to understand that just because these patterns are handy and well structured, it does not mean that we have to plug them all over the place. Selecting the appropriate design pattern for a specific scenario requires knowledge and experience. But when the correct pattern is used, you will definitely realize that your application is well structured and flexible to changes in user requirements to say the least.