Software Design Patterns in IoC Perspective

Michał Warkocz

08 March 2017

Plan

- Introduction
- Chain of Responsibility
- Object Pool
- Factory
- Strategy
- Adapter
- Proxy

What are Software Design Patterns?

- **General reusable solution** to a commonly occurring problem within a given context in software design.
- **Formalized best practices** that the programmer can use to solve common problems when designing an application or system.
- **Descriptions or templates** for how to solve a problem that can be used in many different situations.

A design pattern is **not** a finished design that can be transformed directly into source or machine code.

Software Design Patterns vs S.O.L.I.D.

S.O.L.I.D. stands for:

- **S**ingle responsibility principle
- **O**pen/closed principle
- **L**iskov substitution principle
- **I**nterface segregation principle
- **D**ependency inversion principle

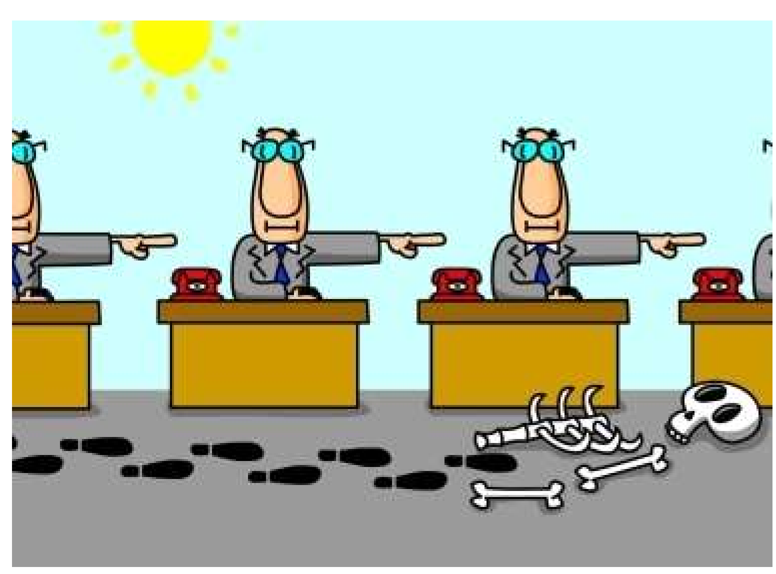
Chain of Responsibility

Problem:

„There is a potentially variable number of "handler" or "processing element" or "node" objects, and a stream of requests that must be handled.”

Examples:

- Workflow steps,
- Exeptions,
- Request routing,



Chain of Responsibility

```
class LoanRequest
{
    public string Customer { get; set; }
    public decimal Amount { get; set; }
}

abstract class RequestHandler
{
    public string Name { get; set; }
    public RequestHandler Successor { get; set; }

    public abstract void HandleRequest(LoanRequest req);
}
```

Chain of Responsibility

```
// Concrete Request Handler - Cachier
// Cachier can approve requests upto 1.000$$
class Cashier : RequestHandler
{
    public override void HandleRequest(LoanRequest req)
    {
        Console.WriteLine("{0} $$ Loan Requested by {1}",
            req.Amount, req.Customer);

        if (req.Amount < 1000)
        {
            Console.WriteLine("{0}$ Loan approved for {1}, by {2}",
                req.Amount, req.Customer, this.Name);
        }
        else
        {
            this.TrySuccessor(req);
        }
    }
}
```

Chain of Responsibility

```
// Concrete Request Handler - Manager
// Manager can approve requests up to 10.000$
class Manager : RequestHandler
{
    public override void HandleRequest(LoanRequest req)
    {
        if (req.Amount < 10000)
        {
            Console.WriteLine("{0}$ Loan approved for {1}, by {2}",
                req.Amount, req.Customer, Name);
        }
        else
        {
            this.TrySuccessor(req);
        }
    }
}
```

Chain of Responsibility - Example

```
//Just an extension method for the passing the request
static class RequestHandlerExtension
{
    public static void TrySuccessor(this RequestHandler current, LoanRequest req)
    {
        if (current.Successor != null)
        {
            Console.WriteLine("{0} Can't approve - Passing request to {1}",
                current.Name, current.Successor.Name);
            current.Successor.HandleRequest(req);
        }
        else
        {
            Console.WriteLine("Amount invaid, no approval given");
        }
    }
}
```

Chain of Responsibility - Example

```
class Program
{
    static void Main(string[] args)
    {
        var request1 = new LoanRequest() { Amount = 800, Customer = "Jimmy" };
        var request2 = new LoanRequest() { Amount = 5000, Customer = "Ben" };
        var request3 = new LoanRequest() { Amount = 200000, Customer = "Harry" };

        var manager = new Manager(){ Name = "Tom, Manager" };
        var cashier = new Cashier(){ Name = "Job, Cachier", Successor = manager };

        cashier.HandleRequest(request1);
        Console.WriteLine();
        cashier.HandleRequest(request2);
        Console.WriteLine();
        cashier.HandleRequest(request3);
        Console.ReadLine();
    }
}
```

Chain of Responsibility - Example

file:///F:/Projects/Tests/ConsoleApplication1/ConsoleApplication1/bin/Debug/ConsoleApplication1.EXE

```
800 $$ Loan Requested by Jimmy
800$ Loan approved for Jimmy, by Job, Cachier

5000 $$ Loan Requested by Ben
Job, Cachier Can't approve - Passing request to Tom, Manager
5000$ Loan approved for Ben, by Tom, Manager

200000 $$ Loan Requested by Harry
Job, Cachier Can't approve - Passing request to Tom, Manager
Amount invaid, no approval given
```

Chain of Responsibility

Pros

- Reduces coupling between sender and receiver.
- Flexibility in responsibilities.

Cons

- As handlers make decision about their action – there is no guarantee of being handled.

Object Pool

Problem:

„System uses many instances of the same class over and over causing performance issues."

Examples:

- Threads,
- Database connections,
- Socket connections,
- Large graphical objects,

Object Pool

Actual pattern

```
public class ObjectPool<T>
{
    private readonly ConcurrentBag<T> _items = new ConcurrentBag<T>();
    private readonly Func<T> _generator;

    public ObjectPool(Func<T> generator)
    {
        _generator = generator;
    }

    public T Get()
    {
        T item;
        return _items.TryTake(out item) ? item : _generator();
    }

    public void Put(T item)
    {
        _items.Add(item);
    }
}
```

Object Pool

Problems

- You have to **reset pooled object state manually**. They are not hard to implement but every one carries some drawbacks. There are several ways to automatize it:
  - use common interface for every pooled type that will enforce implementing Reset method.
  - pass reset method to Object Pool that will perform operation on each pooled object
- **Inadequate resetting** object may lead to **information leak** or random and **unreproducible exceptions**.
- Pool **may waste memory** on unneeded objects because in most case it does not reduce number of stored objects.
- **Code** becomes **more complicated**.
- Some **resource** may **expire** and still be artificially **kept alive** by the pool.
- **Limited number of items** in pool. If the pool has a number limit – some threads may be forced to wait for their items. On the other hand if pool has unlimited size – it may lead to allocation enormous amounts of memory without releasing them later.

Object Pool

Conclusion

Object pool can bring great benefits or be a disaster in you project. It all comes down to individual situation. If you think of using it please take two things into account:

- Check if Object Pool will actually significantly increase performance of your solution. Because if the performance increase is not very high – it is better not to use this pattern.
- Check if there already is library/framework that you can reuse in your application. Because someone has probably spend significant amount of time doing it the right way.

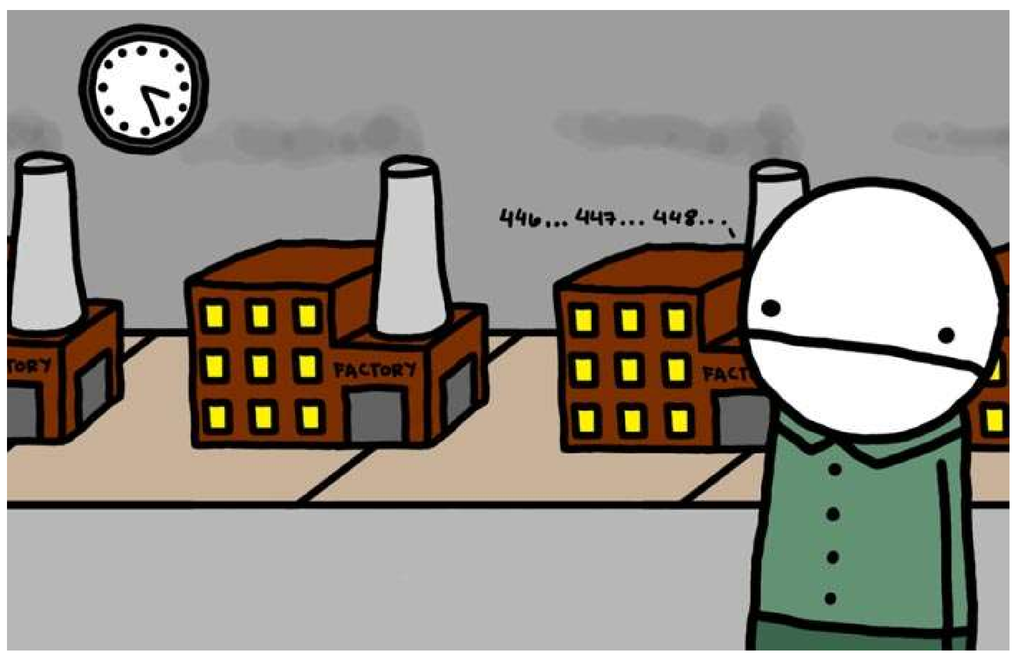
Strategy + Factory

Problem:

„Application structure becomes not redable due to multiple similar, but not identicall entities. (objects, processes, algorithms)”

Examples:

- Workflow steps,
- Price calculation,
- Processing algorithms,



POCOs

```
public class Product
{
    public string Name { get; set; }
    public decimal Price { get; set; }
}


public enum PriceAlgorithm
{
    Standard,
    Second50Off,
    ThirdForOne
}
```

Usual code

```
public class PriceHelper
{
    public decimal GetPrice(IEnumerable<Product> basket, PriceAlgorithm promo)
    {
        switch (promo)
        {
            case PriceAlgorithm.Standard:
                return basket.Sum(p => p.Price);
            case PriceAlgorithm.Second50Off:
                var sortedProducts = basket.OrderByDescending(p => p.Price);
                int productCount = sortedProducts.Count();
                int half = (productCount % 2) == 0 ? (productCount / 2) : (productCount / 2) + 1;
                decimal fullPriceSum = sortedProducts.Take(half).Sum(p => p.Price);
                decimal halfPriceSum = sortedProducts.Skip(half).Sum(p => p.Price / 2);
                return fullPriceSum + halfPriceSum;
            case PriceAlgorithm.ThirdForOne:
                var sortedProducts1 = basket.OrderByDescending(p => p.Price);
                int productCount1 = sortedProducts1.Count();
                int third = productCount1 / 3;
                int full = productCount1 - third;
                decimal fullPriceSum1 = sortedProducts1.Take(full).Sum(p => p.Price);
                return fullPriceSum1 + third;
            default:
                throw new ArgumentException("Inavalid Price Algorithm");
        }
    }
}
```

1. Define Interface

```
public interface IPriceCalculator
{
    decimal GetPrice(IEnumerable<Product> products);
}
```

2. Define concrete classes

```
public class StandardPriceCalculator : IPriceCalculator
{
    public decimal GetPrice(IEnumerable<Product> products)
    {
        return products.Sum(p => p.Price);
    }
}
```

2. Define concrete classes

```
public class Second50OffPriceCalculator : IPriceCalculator
{
    public decimal GetPrice(IEnumerable<Product> products)
    {
        var sortedProducts = products.OrderByDescending(p => p.Price);
        int productCount = sortedProducts.Count();
        int halfIndex = (productCount % 2) == 0
            ? productCount / 2
            : (productCount / 2) + 1;

        decimal fullPriceSum = sortedProducts.Take(halfIndex).Sum(p => p.Price);
        decimal halfPriceSum = sortedProducts.Skip(halfIndex).Sum(p => p.Price / 2);

        return fullPriceSum + halfPriceSum;
    }
}
```

2. Define concrete classes

```
public class ThirdForOnePriceCalculator : IPriceCalculator
{
    public decimal GetPrice(IEnumerable<Product> products)
    {
        var sortedProducts = products.OrderByDescending(p => p.Price);

        int fullPriceProductsCount = FullPriceProductsCount(sortedProducts);
        int promoItemsCount = products.Count() - fullPriceProductsCount;

        decimal fullPriceSum = sortedProducts.Take(fullPriceProductsCount)
                                            .Sum(p => p.Price);
        return fullPriceSum + promoItemsCount;
    }

    private static int FullPriceProductsCount(IEnumerable<Product> sortedProducts)
    {
        int productCount = sortedProducts.Count();

        int thirdProductsCount = productCount/3;
        int fullPriceProductsCount = productCount - thirdProductsCount;
        return fullPriceProductsCount;
    }
}
```

3. Define factory interface

```
public interface IPriceCalculatorFactory
{
    IPriceCalculator GetCalculator(PriceAlgorithm algorithm);
}
```

4. Define concrete factory

```
public class PriceCalculatorFactory : IPriceCalculatorFactory
{
    public IPriceCalculator GetCalculator(PriceAlgorithm algorithm)
    {
        switch (algorithm)
        {
            case PriceAlgorithm.Standard:
                return new StandardPriceCalculator();
            case PriceAlgorithm.Second50Off:
                return new Second50OffPriceCalculator();
            case PriceAlgorithm.ThirdForOne:
                return new ThirdForOnePriceCalculator();
            default:
                throw new Exception("Invalid Algorithm");
        }
    }
}
```

5. Refactor existing code

```
public class PriceHelper
{
    private IPriceCalculatorFactory _factory;

    public PriceHelper(IPriceCalculatorFactory factory)
    {
        _factory = factory;
    }

    public decimal GetPrice(IEnumerable<Product> basket, PriceAlgorithm promo)
    {
        IPriceCalculator calculator = _factory.GetCalculator(promo);
        return calculator.GetPrice(basket);
    }
}

//----------------------------------------------------------------------

container.Register(Component
    .For<IPriceCalculatorFactory>()
    .ImplementedBy<PriceCalculatorFactory>());
```

Strategy + Factory

Pros

- The algorithms and behaviours can be reused
- The algorithms are loosely coupled with the context entity. They can be changed/replaced without changing the context entity
- It allows you to easily manipulate creation process of object
- It allows you to easily test the seam of an application

Cons

- It increases the number of objects in the application.
- It makes code more difficult to read as all of your code is behind an abstraction that may in turn hide abstractions.

Adapter

Problem:

„There is a need to treat objects with incompatible interfaces in common fasion."

Examples:

- Link legacy system with new one without modificating any of them.
- Use objects/methods from different libraries interchangeably.

Adapter

```
class FileReader
{
    public string[] GetDataFromFile()
    {
        return new string[] { "1", "2" };
    }
}

class DbRepository
{
    public List<string> LoadData()
    {
        return new List<string> { "3", "4" };
    }
}
```

Adapter

```
interface IDataCollector
{
    IEnumerable<string> GetData();
}

class FileReaderAdapter : IDataCollector
{
    public IEnumerable<string> GetData()
    {
        return new FileReader().GetDataFromFile();
    }
}

class DbRepositoryAdapter : IDataCollector
{
    public IEnumerable<string> GetData()
    {
        return new DbRepository().LoadData();
    }
}
```

Adapter

```
static void Main()
{
    var container = new WindsorContainer();
    container.Register(Component.For<IDataCollector>()
        .ImplementedBy<FileReaderAdapter>());
    container.Register(Component.For<IDataCollector>()
        .ImplementedBy<DbRepositoryAdapter>());

    var dataSources = container.ResolveAll<IDataCollector>();
    var results = new List<string>();

    foreach (var dataSource in dataSources)
    {
        results.AddRange(dataSource.GetData());
    }

    foreach (var result in results)
    {
        Console.WriteLine(result);
    }

    Console.ReadKey();
}
```

Console output:

```
file:///F:/Projects/Tests/
1
2
3
4
```

Adapter

Pros

- Only one new object – no additional indirection
- It can override Adaptee's behaviour

Cons

- Adds additional layer of abstraction

Proxy

Problem:

„There is a need to perform action(s) before using class that is not modifiable."

Examples:

- Adding security access to an existing object,
- Providing interface for remote resources,
- Adding a thread-safe feature to an existing class without changing the existing class's code,

Proxy

```
interface ICar
{
    void DriveCar();
}

public class Car : ICar
{
    public void DriveCar()
    {
        Console.WriteLine("Car has been driven!");
    }
}
```

Proxy

```
public class ProxyCar : ICar
{
    public Driver Driver { get; set; }
    private ICar realCar = new Car();

    public void DriveCar()
    {
        if (Driver.Age <= 16)
        {
            Console.WriteLine("Sorry, the driver is too young to drive.");
        }
        else
        {
            realCar.DriveCar();
        }
    }
}

public class Driver
{
    public int Age { get; set; }
}
```

Proxy

```
static void Main()
{
    ICar proxy1 = new ProxyCar
    {
        Driver = new Driver { Age = 15 }
    };
    proxy1.DriveCar();

    ICar proxy2 = new ProxyCar
    {
        Driver = new Driver { Age = 26 }
    };
    proxy2.DriveCar();

    Console.ReadKey();
}
```

```
Sorry, the driver is too young to drive.
Car has been driven!
```

Proxy

Pros

- Single point of control
- Protection of underlying object

Cons

- If existing class exposes public variables, all would need to be refactored into getter/setter methods

Thank you

michal.warkocz@goyello.com