Study On Creational Design of Builder Pattern And Abstract Factory Pattern With Challenging Example in C# used for Real Cases

ARABINDA RATH¹, DILLIP KUMAR MISHRA², GAYATRI DEVI³, SOUMYA SEKHAR PRUSTY⁴

ABIT ¹,²,³,⁴, Cuttack

Abstract

The design patterns are the formulas which help to solve software design problems and apply in real-world development. Patterns are about reusable designs and interactions of objects. Patterns are used by developers for their specific design to solve their problems. Pattern choice and usage among various design patterns depends on individual need and their problem. Design patterns are a very powerful tool for software developers. It is important to understand design patterns rather than memorizing their classes, methods and properties. It is also important to learn how to apply patterns to specific problems to get the desired result. This will be the required continuous practice for using and applying design patterns in day to day software development. First identify the software design problem then see how to address these problems using design patterns and determine the best suited design problem to solve the problem. In this paper the comparison of abstract factory patterns and builder pattern with example for more clarity are explained.

Keywords: Builder pattern, Abstract factory pattern, C# language, UML Diagram

1. Introduction

The structure of the current paper is as follows: There are two topics in this paper. Each topic provides motivation behind our work. First, we find the problem factor with an example. Then we introduce the approach and methodology adopted to solve the problem. The UML diagram of the methodology which shows the description of practical example. We use C# Language for testing our approach, methodology which gives better result than [9], [15] and [17]. Secondly, we compare between the abstract factory pattern and builder pattern from creational design pattern.

2. Introduction To Design Pattern

Patterns were invented as an architectural concept by Christopher Alexander in the 1960s and the 1970s. Alexander defines a pattern as ”A recurring solution to a common problem in a given context and system of forces” [10]. Alexander published several books on architectural design that were concerned with creating and using patterns in the architecture domain. Alexander described a variety of patterns in space, human existence, and events, with the aim of improving people’s living quality. In the following decades, the books of Alexander inspired the domain of Computer Science [7]. In 1987, Kent Beck and Ward Cunningham started experimenting with the idea of applying patterns to software programming (specifically pattern languages). In 1988, they [3] developed user interfaces in Smalltalk by using some ideas from Alexander’s architecture patterns. Starting from the 1990s, the work on patterns increased, as described below:

1. In 1991 Jim Coplien developed a set of patterns called idioms in C++ These were a type of low level pattern specific to a programming language After that, Erich Gamma started to concentrate on recurring structures and patterns in his PhD thesis [12].

2. In 1992, numerous professionals in software design including Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides congregated together to discuss patterns at the annual conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA). Later, this group of professionals came to represent the members of the ‘Gang of Four’ [11].
3. In 1993, the first version of a catalogue of design patterns was published. This catalogue became the basis for the milestone textbook, which was published two years later.

4. In 1995, the ‘Gang of Four’ revealed their milestone textbook Design Patterns: Elements of Reusable Object-Oriented Software

**About Gangs of Four**

There are 23 design patterns, also known as Gang of Four (GoF) design patterns. The Gang of Four are the authors of the book, "Design Patterns: Elements of Reusable Object-Oriented Software”. This important book describes various development techniques and pitfalls in addition to providing twenty-three object-oriented programming design patterns. The four authors were Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides [3].

**2.1 Approach and methodology**

C# is a simple, modern, general-purpose, object-oriented programming language developed by Microsoft within its .NET initiative led by Anders Hejlsberg. As an object-oriented language, C# supports the concepts of encapsulation, inheritance, and polymorphism. All variables and methods, including the Main method, the application’s entry point, are encapsulated within class definitions. [1]

**2.2 Unified Modeling Language(UML)**

The Unified Modeling Language (UML) is a general-purpose, developmental, modeling language in the field of software engineering, that is intended to provide a standard way to visualize the design of a system. [2]

The creation of UML was originally motivated by the desire to standardize the disparate notational systems and approaches to software design. It was developed by Grady Booch, Ivar Jacobson and James Rumbaugh at Rational Software in 1994–1995, with further development led by them through 1996. UML is linked with object oriented design and analysis. UML makes the use of elements and forms associations between them to form diagrams[13].

**2.3 Abstract Factory**

This pattern supports the creation of products that exist in families and are designed to be produced together. The abstract factory can be refined to concrete factories, each of which can create different products of different types and in different combinations. The pattern isolates the product definitions and their class names from the client so that the only way to get one of them is through a factory. For this reason, product families can easily be interchanged or updated without upsetting the structure of the client [6].

**2.3.1 Problem Description and Objective**

Imagine that you are creating a simulator of a mobile phone shop. Your code consists of:

2. Several variants of this brand. For example, products Samsung+ Nokia+ Karbonn available in these variants:Nokia 6, Galexy S8, K9, Nokia 3310, Guru 1200, K2.

You need a way to create individual phone model objects so that they match other objects of the same brand. Customers get quite frustrated when receive non-matching mobile phone.

Also, you do not want to change existing code when adding new products or brand of products to the program [14]. Mobile phone vendors update their catalogs very often, and you do not want to change the core code each time it happens.

**2.3.2 Solution**

The first thing that Abstract Factory pattern suggests is to go over all distinct products and force their variants to follow common interfaces. For example, all Tuch Screen variants must follow the ITuchScreen interface; all Push Button mobiles must implement the IPushButton interface, etc [8].

So, Abstract Factory = Factory 1 + Factory 2 + ---------Factory n

The second step is to create the AbstractFactory, a base interface that declares methods for creating all products that make a product family (i.e.GetPushButton,GetTuchScreen). The important thing here is to make these methods to return abstract product types represented by interfaces we extracted previously: ITuchScreen, IPushButton.
The third step is to implement concrete factories. Factories are classes that return products of a particular kind. For example, IMobilePhoneFactory, will only return NokiaFactory, SamsungFactory and KarbonnFactory objects. All factories must follow the AbstractFactory interface while creating the same variety of products.

Client code has to work with factories and products only through their abstract interfaces. This way you can alter the type of products used in client code by passing it a different factory object.

So, when client code asks a factory to show a Nokia mobile variant, it must not be aware of the factory's concrete class. It must not be aware of the concrete class of Nokia it will get either. Whether it will be a modern Touch Screen model or a Push Button, it must work with all Nokia Mobiles in the same way, using the ITouchScreen and IPushButton interface. The thing that the client code will know is that the resulting ITouchScreen implements the ModelName method, declared in the interface. It also knows that whichever touch screen mobile will be returned, it will match the type of push button mobile, produced by the same factory.

Okay, but who creates the actual factory objects? Usually, the program creates a concrete factory object at initialization stage, and the factory type is picked depending on the configuration or environment. [9]

### 2.3.3 UML Class Diagram

The Abstract Factory pattern has a lot of players, as illustrated in the following UML diagram (Figure 1.1).

**Figure 1.1** shows the UML diagram for abstract factory method.
2.3.4 The players in the pattern

AbstractFactory(IMobilePhoneFactories)
An interface with Create operations for each of the abstract products.

NokiaFactory, SamsungFactory, KarbonnFactory
Implementations of all the AbstractFactory creation operations.

AbstractProduct(ITouchScreen,IPushButton)
An interface for a kind of product with its own operations

Nokia6, Galaxy, K9, Nokia3310, Guru1200, K2
Classes that implement the AbstractProduct interface and define product objects to be created by the corresponding factories.

Client
A class that accesses only the AbstractFactory and AbstractProduct interfaces [3]

3. Implementation and Example: Variety Mobile Phone Store
Now let us focus on the problem at hand. We need to create the appropriate object containing the information about cell phone based on the user request of 1. Type of phone 2. Phone manufacturer. For the sake of simplicity, let's assume we have 3 manufacturers:

1. Nokia
2. Samsung
3. Karbonn

and there could be two types of phones:
1. Touchpad
2. Push Button

So with this information, we can safely say that we need three concrete factories (one for each manufacturer) and two sets of related products (one for touch screen and one for push button).

3.1 Creating the Abstract Products
In our case, we need two abstract products ITouchScreen and IPushButton.

```csharp
namespace AbstractFactoryMethod
{
    /// <summary>
    /// Abstract Factory Pattern
    /// Author : Soumya Sekhar Prusty
    /// </summary>

    #region Create Abstract Phone Model

    public interface ITouchScreen
    {
        string ModelName();
    }

    public interface IPushButton
    {
        string ModelName();
    }

    #endregion
```
3.2 Creating the Concrete Products

Now let us go ahead and create some concrete products for ITouchScreen:

    #region Concrete Phone Model ForITouchScreen

    public class Nokia6 : ITouchScreen
    {
        public string ModelName()
        {
            return "Nokia 6";
        }
    }

    public class Galaxy : ITouchScreen
    {
        public string ModelName()
        {
            return "Galaxy S8";
        }
    }

    public class K9 : ITouchScreen
    {
        public string ModelName()
        {
            return "K9";
        }
    }

    #endregion

Let's do the same for IPushButton:

    #region Concrete Phone Model ForIPushButton

    public class Nokia3310 : IPushButton
    {
        public string ModelName()
        {
            return "Nokia 3310";
        }
    }

    public class Guru1200 : IPushButton
    {
        public string ModelName()
        {
            return "Guru 1200";
        }
    }

    public class k2 : IPushButton
    {
        public string ModelName()
        {
            return "K2";
        }
    }

    #endregion
So we have all the concrete classes ready for all the touch screen Phones and push button phones irrespective of their manufacturers.

3.3 Creating the Abstract Factory
Now the way we associate these Concrete products with their manufacturers is using the Concrete factories. But before having the concrete factories, we need to have an Abstract Factory.

    #region Create Abstract Factories
    public interface IMobilePhoneFactories
    {
        IPushButtonGetPushButton();
        ITouchScreenGetTouchScreen();
    }
    #endregion

3.4 Creating the Concrete Factories
Now we can create our Concrete Factories for each manufacturer:

    #region Create Concrete Factories
    public class NokiaFactory :IMobilePhoneFactories
    {
        public IPushButtonGetPushButton()
        {
            return new Nokia3310();
        }
        public ITouchScreenGetTouchScreen()
        {
            return new Nokia6();
        }
    }

    public class SamsungFactory :IMobilePhoneFactories
    {
        public IPushButtonGetPushButton()
        {
            return new Guru1200();
        }
        public ITouchScreenGetTouchScreen()
        {
            return new Galaxy();
        }
    }

    public class KarbonnFactroy :IMobilePhoneFactories
    {
        public IPushButtonGetPushButton()
        {
            return new k2();
        }
        public ITouchScreenGetTouchScreen()
        {
            return new K9();
        }
    }
    #endregion
3.5 Creating the Client

Now we have all the Abstract product classes ready, all the Concrete Product classes ready. Our Abstract Factory is ready and all the Concrete Factories are ready. Now we can write client that will use this hierarchy of related products to create the products.

```csharp
#region Client Interaction

class PhoneTypeChecker
{
    IMobilePhoneFactories factory;
    public void CheckProduct()
    {
        Console.Write("PLEASE PROVIDE MOBILE COMPANY NAME: ");
        string phone = Console.ReadLine();
        Console.WriteLine();
        switch (phone)
        {
            case "nokia":
                factory = new NokiaFactory();
                break;
            case "samsung":
                factory = new SamsungFactory();
                break;
            case "karbonn":
                factory = new KarbonnFactroy();
                break;
            default:
                Console.WriteLine("No Matches Found According to the Input");
                break;
        }
        if (factory != null)
        {
            Console.WriteLine("Phone Models For " + phone + " push button is: " + factory.GetPushButton().ModelName() + " and TouchScreen is: " + factory.GetTouchScreen().ModelName());
            Console.WriteLine();
        }
    }
}

class DPAbstractFactoryMethod
{
    static void Main(string[] args)
    {
        while (true) {
            PhoneTypeChecker phType = new PhoneTypeChecker();
            phType.CheckProduct();
        }
    }
}
#endregion

3.6 Simulation:

We can see here if a customer search for a mobile according to a manufacturer or a brand then he/she can get the models according to the mobile types.
Figure 2.2 shows the output of the above example

4. Builder Pattern
Builder pattern separate the construction of a complex object from its representation so that the same construction process can create different representations. Builder pattern was introduced to solve some of the problems with Factory and Abstract Factory design patterns when the Object contains a lot of attributes. Also builder pattern helps to hide the complexity of making instances from the client [6].

4.1 Problem Description and Objective
There are three major issues with Factory and Abstract Factory design patterns when the Object contains a lot of attributes [16].

1. Too Many arguments to pass from client program to the Factory class that can be error prone because most of the time, the type of arguments are same and from client side it is hard to maintain the order of the argument[4].
2. Some of the parameters might be optional but in Factory pattern, we are forced to send all the parameters and optional parameters need to send as NULL.
3. If the object is heavy and its creation is complex, then all that complexity will be part of Factory classes that is confusing.

4.2 Solution
We can solve the issues with large number of parameters by providing a constructor with required parameters and then different setter methods to set the optional parameters. The problem with this approach is that the Object state will be inconsistent until unless all the attributes are set explicitly[5].

Builder pattern solves the issue with large number of optional parameters and inconsistent state by providing a way to build the object step-by-step and provide a method that will actually return the final Object [16].

Developer should ask two question before choosing builder pattern

- Is the construction process of object is complex
- Multiple representation of same construction process

4.3 UML Class Diagram
Builder pattern players are illustrated in the following class diagram[14].
4.4 Players in the pattern

**Builder**
This is an interface which is used to define all the steps to create a product.

**Concrete Builder**
This is a class which implements the Builder interface to create complex product.

**Director**
This is a class which is used to construct an object using the Builder interface.

**Product**
This is a class which defines the parts of the complex object which are to be generated by the builder pattern.

4.5 Implementation and Example: Construction of Different Types of Mobile Phone

Let’s consider about a mobile phone factory which constructs different types of mobile phone. Here we are going to take phone models and the construction process. There are two mobile phones which are touch screen type and push button type. The two phones are made off of different materials, So that the method its used for the construction of the object is different. The example is given below.
4.5.1 Creating the Builder

This is an abstract interface for creating parts of mobile phone type. Here IPhoneBuilder is the builder.

```csharp
/// <summary>
/// Builder Interface
/// Author: Soumya Sekhar Prusty
/// </summary>
public interface IPhoneBuilder
{
    void SetModelNumber();
    void SetScreenType();
    void SetScreenSize();
    void SetKeyboardType();
    void SetMotherBoard();
    void SetRam();
    void SetProcessor();
    PhoneProduct GetPhoneDetails();
}
```

4.5.2 Concrete Builder

Constructs and assembles parts of the product by implementing the builder interface. Here there are two concrete builder. I.e. PushButtonBuilder and TouchScreenBuilder.

These concrete builder shows the construct and assemble of two type of mobile phones. These concrete builder patterns are using the methods of IPhoneBuilder.

```csharp
/// <summary>
/// Concrete Touch Screen Phone Type Builder Class
/// Author: Soumya Sekhar Prusty
/// </summary>
public class TouchScreenBuilder : IPhoneBuilder
{
    PhoneProduct phone = new PhoneProduct();
    public void SetModelNumber()
    {
        phone.ModelNumber = "Touch001";
    }
    public void SetScreenType()
    {
        phone.ScreenType = "Touch Screen";
    }
    public void SetScreenSize()
    {
        phone.ScreenSize = 5.0;
    }
    public void SetKeyboardType()
    {
        //Keyboard is not used in touch screen type phone
    }
    public void SetMotherBoard()
    {
        phone.MotherBoard = "TTYM001";
    }
    public void SetRam()
    {
    }
}
```
```csharp
{
    phone.Ram = "4 Gb";
    
    public void SetProcessor()
    {
    
    phone.Processor = "Snapdragon";
    
    public PhoneProductGetPhoneDetails()
    {
        return phone;
    }
    }

    /// <summary>
    /// Concrete Push Button Type Builder class
    /// Soumya Sekhar Prusty
    /// </summary>
    public class PushButtonBuilder :IPhoneBuilder
    {
        PhoneProduct phone = new PhoneProduct();

        public void SetModelNumber()
        {
            phone.ModelNumber = "Button001";
            
            public void SetScreenType()
            {
                phone.ScreenType = "Non Touch Screen";
                
            public void SetScreenSize()
            {
                phone.ScreenSize = 3.0;
                
            public void SetKeyBoardType()
            {
                phone.KeyPressType = "QWERTY Keypad";
                
            public void SetMotherBoard()
            {
                phone.MotherBoard = "PBTYM001";
                
            public void SetRam()
            {
                //No ram memory required for this type of phone
                
            public void SetProcessor()
            {
                //No processor required in this type of phone
                
            public PhoneProductGetPhoneDetails()
            {
                return phone;
            }
        }

4.5.3 Director

Constructs an object using builder interface. Here **PhoneManufactorDirector** is the director class
/// <summary>
/// Director class
/// Author: Soumya Sekhar Prusty
/// </summary>
public class PhoneManufacturerDirector
{
    public void BuildPhone(IPhoneBuilder buildPhone)
    {
        buildPhone.SetModelNumber();
        buildPhone.SetScreenType();
        buildPhone.SetScreenSize();
        buildPhone.SetKeyBoardType();
        buildPhone.SetMotherBoard();
        buildPhone.SetRam();
        buildPhone.SetProcessor();
        buildPhone.GetPhoneDetails();
    }
}

4.5.4 Product

Represents the complex object under construction. Concrete Builder builds the products’s internal representation and defines the process by which it’s assembled. Here PhoneProduct is the product class. This defines the constituent parts of the mobile phone product.

/// <summary>
/// Product Class
/// Author: Soumya Sekhar Prusty
/// </summary>
public class PhoneProduct
{
    public string ModelNumber { get; set; }
    public string ScreenType { get; set; }
    public double ScreenSize { get; set; }
    public string KeyPressType { get; set; }
    public string MotherBoard { get; set; }
    public string Ram { get; set; }
    public string Processor { get; set; }

    public void DisplayDetails(string phoneType)
    {
        Console.WriteLine("******************** \"{0}\" Phone Specification ********************\n", phoneType);
        if (!string.IsNullOrWhiteSpace(ModelNumber))
            Console.WriteLine("Model Number: \"{0}\"\n", ModelNumber);
        if (!string.IsNullOrWhiteSpace(ScreenType))
            Console.WriteLine("Screen Type: \"{0}\"\n", ScreenType);
        if (ScreenSize != 0)
            Console.WriteLine("Screen Size: \"{0}\" inch\n", ScreenSize);
        if (!string.IsNullOrWhiteSpace(KeyPressType))
            Console.WriteLine("Button Press Style: \"{0}\"\n", KeyPressType);
        if (!string.IsNullOrWhiteSpace(MotherBoard))
            Console.WriteLine("Mother board used: \"{0}\"\n", MotherBoard);
        if (!string.IsNullOrWhiteSpace(Ram))
            Console.WriteLine("RAM used in this phone: \"{0}\"\n", Ram);
        if (!string.IsNullOrWhiteSpace(Processor))
            Console.WriteLine("Processor used in this phone: \"{0}\"\n", Processor);
    }
}
4.5.5 Client

The client creates the director object and configures it with the desired builder object.

```csharp
/// <summary>
/// Client Class
/// Author: Soumya Sekhar Prusty
/// </summary>
public class Client
{
    static void Main(string[] args)
    {
        //Creating object of Director
        PhoneManafactorDirector phoneMaker = new PhoneManafactorDirector();

        //*************** Touch Screen Type Object Build ***************
        //Creating Builder type object of touchscreen type builder class
        IPhoneBuilder touchScreenBuilder = new TouchScreenBuilder();
        //Passing touchscreen type builder to the Director to construct object
        phoneMaker.BuildPhone(touchScreenBuilder);
        //Arranges the properties of the object
        PhoneProduct touchScreenPhone = touchScreenBuilder.GetPhoneDetails();
        //Shows the constructed object
        touchScreenPhone.DisplayDetails("Touch Screen");

        //*************** Push Button Type Object Build ***************
        //Creating Builder type object of pushbutton type builder class
        IPhoneBuilder pushButtonBuilder = new PushButtonBuilder();
        //Passing pushbutton type builder to the Director to construct object
        phoneMaker.BuildPhone(pushButtonBuilder);
        //Arranges the properties of the object
        PhoneProduct pushButtonPhone = pushButtonBuilder.GetPhoneDetails();
        //Shows the constructed object
        pushButtonPhone.DisplayDetails("Push Button");
        Console.ReadKey();
    }
}
```
4.5.6 Simulation:

![Simulation Output](image)

Figure 3.2 shows the output of the program.

4.5.7 Related Patterns as Described in GOF:

Abstract Factory is similar to builder in that it too may construct complex objects. The primary difference is that the builder pattern focuses on constructing a complex object step by step. Abstract factory’s emphasis is on families of product objects (either simple or complex)[3].

### 5. Comparison Study Between Abstract Factory and Builder Pattern

<table>
<thead>
<tr>
<th>Abstract Factory</th>
<th>Builder Pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>Emphasizes a family of product objects (either simple or complex)</td>
<td>Focuses on constructing a complex object step by step</td>
</tr>
<tr>
<td>Focus on <em>what</em> is made</td>
<td>Focus on <em>how</em> it is made</td>
</tr>
<tr>
<td>Focus on defining many different types of <em>factories</em> to build many <em>products</em>, and it is not a one builder for just one product</td>
<td>Focus on building a one complex but one single <em>product</em></td>
</tr>
<tr>
<td>Defers the choice of what concrete type of object to make until run time</td>
<td>Hide the logic/operation of how to compile that complex object</td>
</tr>
<tr>
<td><em>Every</em> method call creates and returns different objects</td>
<td>Only the <em>last</em> method call returns the object, while other calls partially build the object</td>
</tr>
</tbody>
</table>

Conclusion:

In future, the software will be equipped with deep learning and machine learning. In this category of software, the design pattern of software are so complicated that there is a justification of using C# language for large library products with their implementations.
REFERENCES

[5.] https://refactoring.guru/design-patterns/factory-method
[6.] Judith Bishop “C# 3.0 Design Patterns” copyright 2008
[8.] https://refactoring.guru/design-patterns/abstract-factory
[14.] Benneth Christiansson"Gof design patterns with examples using java and UML2".
[17.] Rathet.al. “ Comparison study between factory pattern and abstract factory pattern from creational design patterns with example in C#”, IJAIEM v.7, No.7, July2018 page 25-36.

AUTHOR

Arabinda Rath :B-Tech , M-tech in Computer Science and Engineering from BPUT , Odisha and FM university respectively. Now he is working as phdscholar ,his specialization in communication , NeuralNetwork.

Dillip Kumar Mishra :Received the B-tech from RTMNU, Nagpur university And M-tech from BPUT , Raurkela , with Computer science and engineering . Now doing his patent work on smart communication with ion saturation . His specialization on WSN, ANN, NeuroFuzzy , Deep Learning , Quantum computation.

Dr Gayatri Devi:PhD in 1998 from Utkal University . Her specialization OR, NN, Software Engg, PCB design , communication . Recently working for patent “ new technology for communication with ion saturation” .

Soumya Sekhar Prusty: M-tech in CSE from BPUT , Odisha. His specialization in Software development and design. He is working on several real-time projects with realtime simulations .