ISSN: 2754-4915

Journal of Material Sciences & Manufacturing Research

Review Article

Open Access

Utilizing the Factory Method Design Pattern in Practical Manufacturing Scenarios

Nilesh D Kulkarni[1]* and Saurav Bansal[2]

[1]Sr Director, Enterprise Architecture, Fortune Brands Home & Security, USA

[2]Sr Manager, Digital Applications, Fortune Brands Home & Security,USA

ABSTRACT
The paper delves into the utilization of design patterns, specifically focusing on the Factory Method, within the field of software engineering. It underscores the importance of design patterns in addressing recurring problems in a flexible, elegant, and reusable manner. The document explores various categories of design patterns and their practical applications, with special attention to the Factory Method's role in streamlining object creation and enhancing system maintainability. To illustrate the practicality of the Factory Method, a real-world use case in a manufacturing context is presented, showcasing its influence on software architecture and maintainability. The study concludes by underscoring the advantages of design patterns in software development, particularly in terms of their ability to promote reusability and efficiency when dealing with common software design challenges.

***Corresponding authors**
Nilesh D Kulkarni, Sr Director, Enterprise Architecture, Fortune Brands Home & Security, USA.

Received: February 02, 2022; **Accepted:** February 09, 2022; **Published:** February 16, 2022

Keywords: Design Patterns, Factory Method, Object, .NET, Software Maintainability

Introduction

The importance of design experience is widely recognized. How often have you encountered a familiar problem during design, sensing that you've tackled something similar in the past, yet struggling to recall the specifics of where and how it was resolved? If you were able to recall the nuances of that past challenge and the strategy you employed to overcome it, you could leverage that previous experience instead of having to re-explore the solution from scratch.

A design pattern represents a universally recognized solution, widely observed in various cases, that effectively addresses a specific problem in a context that may not be predefined. It offers a highly efficient approach to developing object-oriented software that is not only flexible and elegant but also reusable. The utilization of design patterns facilitates the reuse of successful designs and architectural models. By translating proven technologies and methodologies into design patterns, they become more easily accessible to developers building new systems. Design patterns guide developers in selecting design options that enhance the reusability of a system, while steering clear of choices that could hinder it. Moreover, design patterns can significantly enhance the documentation and maintenance of existing systems by providing a clear and explicit description of class and object interactions, along with their fundamental purposes. In essence, design patterns empower designers to achieve a more effective design more swiftly.

Typically, a design method comprises a set of synthetic notations usually graphical and a set of rules that govern how and when we use each notation. It will also describe problems that occur in a design, how to fix them, and how to evaluate the design. Each pattern describes a problem which occurs over and over again in the environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over without ever doing it the same way twice [1].

Design patterns describe problems that occur repeatedly, and describe the core of the solution to that problem, in such a way that the solution can be used many times in different contexts and applications. A good design should always be independent of the technology and the design should help both experience and the novice designer to recognize situation in which these designs can be used and reused.

Eric gamma at el in their book Design Patterns, discussed total 23 design patterns clarified by two criteria figure 1. The first criterion, called purpose, reflects what a pattern does. Patterns can have either creational, structural, or behavioral purpose. Creational patterns concern the purpose of object creation. Structural pattern deals with the composition of classes or objects. Behavioral pattern characterizes the ways in which classes or objects interact and distribute responsibility [2]. The second criteria called scope, specifies whether the pattern applies primarily to the class or to the object.

Scope	Purpose		
	Creational	Structural	Behavioral
Class	Factory Method	Adapter	Interpreter Template Method
Object	Abstract Factory Builder Prototype Singleton	Adapter Bridge Composite Decorator Façade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

Figure 1: Design Patterns

UML BASICS

The first versions of UML were created by "Three Amigos" -Grady Booch at el defines "The Unified Modeling Language (UML), is a standardized visual language for specifying, constructing, and documenting the artifacts of software systems. It provides a set of diagrams and notations to represent various aspects of software design and architecture, allowing software engineers to communicate, visualize, and model complex systems effectively."

Three Types of Relations between the Classes

- Association Relationship: When classes are connected together conceptually, that connection is called an association. As shown in the Figure 2, let's examine the association between passenger and airplane. A passenger can sit in an airplane or multiple passengers can sit in an airplane.

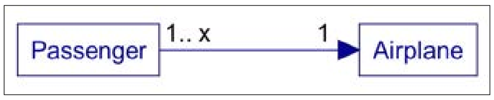

Figure 2: Association Relationship

• **Aggregation Relationship:** This is a special type of relationship, used to model situations where one class (the whole) contains or is composed of other classes or objects (the parts), and the parts have a lifecycle that is independent of the whole. As shown in the Figure 3, next examine the aggregation relationship, an engine (whole) can have many Pistons (parts) similarly an airplane (whole) can have multiple engines (parts) as well as an airplane can have multiple wheels (parts).

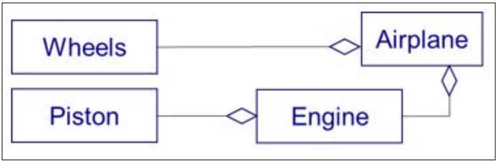

Figure 3: Aggregation Relationship

• **Composition Relationship:** A composition is a strong type of aggregation where each component in the composite can belong to just one whole. As shown in Figure 4, a dog can have a tail, four legs, two ears, and two eyes, but eyes, legs, tail, and ears cannot exist on its own.

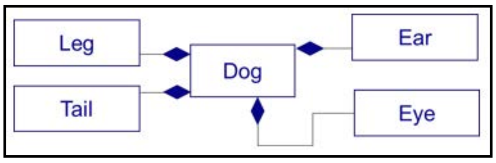

Figure 4: Composition Relationship

Inheritance / Generalization

In this relationship one class (the child class or subclass) can inherit attributes and operations from another (the parent class or superclass). The generalization allows for polymorphism. In generalization, a child is substitutable for parent. That is anywhere the parent appears, the child may appear. The reverse isn't true [3]. As shown in the Figure 5, signifies that "Bus," "Car," and "Truck" inherit from "Vehicle." They are expected to share common characteristics or behaviors that are defined in "Vehicle." For instance, if "Vehicle" has attributes like 'number of wheels' and 'fuel type' and operations like 'start engine ()', then "Bus," "Car," and "Truck" would inherit these operations and attributes.

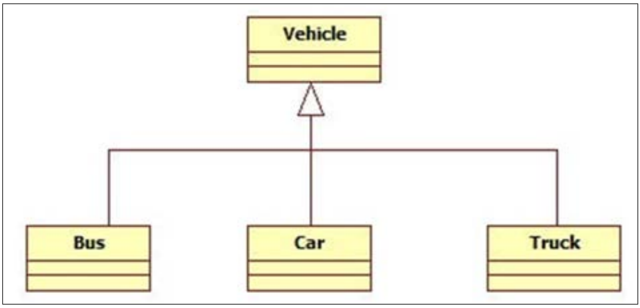

Figure 5: Generization

Interface

An interface is a set of operation that specifies some aspect of classes behavior, and it's set of operation class presents to other classes [3]. As shown in Figure 6, the "Electric System" is considered an interface between the light bulb and the light switch. The "Electric System" serves as a contract between the light bulb and the light switch, stipulating that when the switch is turned on, the bulb should light up. Interfaces are used to decouple the implementation and the abstract design, allowing for changes in implementation without affecting the system that uses the interface. Similarly, the light switch and bulb are decoupled from each other, you could replace either the bulb or the switch without needing to change the other, as long as they both adhere to the same electrical system standards. Interface also allows different classes to be treated through a single interface type, the electric system could work with any device that conforms to its standards, not just a light bulb. This could include a fan, a heater, or any other electric device that can be turned on or off.

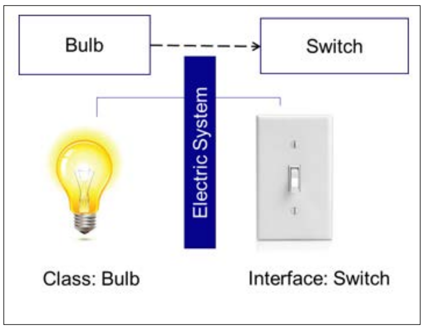

Figure 6: Interface Representation

Programming Technologies
We will using the basic programming tools to show the implementation of the Factory Method design pattern.

1. .NET Framework
The .NET Framework, is a software development framework designed and supported by Microsoft. It provides a controlled environment for developing and running applications on Windows. Few features listed below

- **Windows-Specific:** The .NET Framework is designed to work on Windows operating systems.
- **Base Class Library (BCL):** It includes a large class library known as the Framework Class Library (FCL), providing user interface, data access, database connectivity, cryptography, web application development, numeric algorithms, and network communications.
- **Common Language Runtime (CLR):** Programs written for the .NET Framework execute in a software environment named the Common Language Runtime, which provides services such as security, memory management, and exception handling.
- **Languages:** The .NET Framework supports multiple programming languages, such as C#, VB.NET, and F#.
- **CLI:** Console programming refers to the process of writing software applications that interact with the user through a text-based interface. These applications run in a console or a command-line interface (CLI), where the user inputs text commands and the program provide output in text form.

2. Visual Studio Code (VS Code) for .NET Development
Visual Studio Code is a lightweight, open-source, and cross-platform code editor developed by Microsoft. It's not specific to any one programming language or framework. With the help of extensions, it can support a wide variety of languages and frameworks, including those of the .NET ecosystem. Few features listed below

- **Cross-Platform:** VS Code runs on Windows, Linux, and macOS.
- **Extensions:** The C# extension by Omni Sharp adds support for .NET development, including features like IntelliSense, debugging, project file navigation, and run tasks.
- **Lightweight Editor:** VS Code is designed to be a fast and lightweight editor, with a smaller footprint than a full IDE like Visual Studio.
- **Integrated Terminal:** Developers can use the integrated terminal to execute .NET CLI commands, enabling them to create, build, run, and test .NET applications.
- **Git Integration:** VS Code has built-in Git support, which is essential for modern software development workflows.
- **Language Features:** VS Code with the C# extension supports advanced language features like code refactoring, unit testing, and code snippets for .NET.

Creational Pattern – Factory Method
Creational design patterns provide mechanisms for creating objects while hiding the complexities of how objects are instantiated or composed. These patterns help ensure that the system remains modular, easy to maintain, and less dependent on the concrete classes being used. Factory Method is a creational design pattern that provides an interface for creating objects but allows subclasses to alter the type of objects that will be created.

Key components and characteristics of the Factory Method pattern are

- **Creator:** This is an abstract class or interface that declares a factory method, which returns an object of a product type. The Creator may also provide default implementation code that uses the factory method to create the product.
- **Concrete Creator:** Concrete classes that implement the Creator interface and override the factory method to provide their own implementation for creating specific products.
- **Product:** The product is the object that the factory method creates. It's an abstract class or interface shared by all product types. Concrete products are derived from this base product class.
- **Concrete Products:** These are concrete classes that implement the Product interface. Each concrete product represents a specific type of object created by the factory method.

Manufacturing Application - Use Case
ABC Manufacturing, a prominent printer manufacturer operating in the United States, not only designs and manufactures printers but also handles their product shipments through an in-house logistics department. Currently, their operations are primarily within the USA, and they efficiently utilize trucks and road logistics for the crucial last-mile delivery of their products shown in Figure 7.

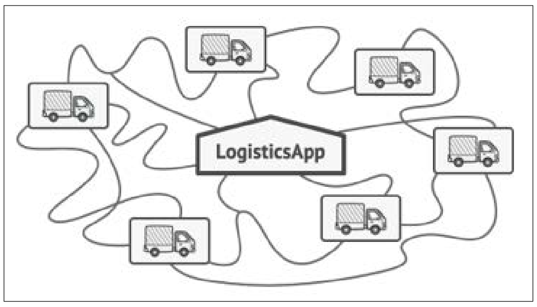

Figure 7: Logistics Application

However, there's an exciting opportunity on the horizon. The company's CEO has recognized a growing international demand for their printers and is eager to expand their market globally. This expansion ambition is indeed promising for the business, but it comes with a significant logistical challenge.

The current logistics approach, heavily reliant on trucks and road transportation, won't be viable for international deliveries. To address this challenge and meet the CEO's vision of international expansion, ABC Manufacturing needs to introduce a completely new logistics method "Sea" transportation.

While this opens up new horizons for business growth, it also presents a software challenge. The existing logistics planning and routing software was initially developed exclusively for domestic operations in the USA. Most of the code is tightly coupled with the "Truck" class, which means that introducing a "Sea" delivery system into the software requires a substantial code overhaul.

In practical terms, this overhaul entails making extensive changes to the entire codebase to accommodate the new "Sea" logistics method. It's a complex task that involves modifying the software architecture, integrating sea routes, and ensuring seamless coordination between both land and sea logistics. Despite the challenges, successfully implementing this change is essential for ABC Manufacturing's international expansion strategy to reach its full potential shown in Figure 8.

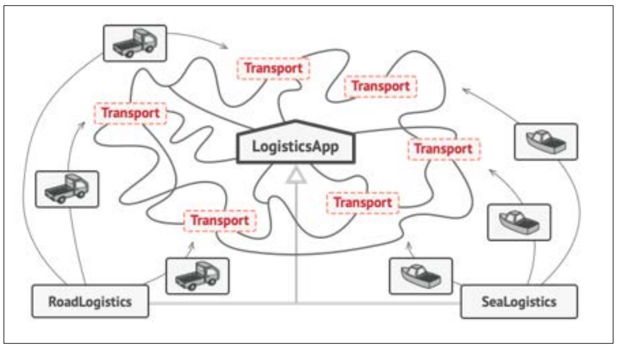

Figure 8: Potential Expansion of Code

Design Pattern Application

Design pattern that provides an interface (1) for creating objects in a superclass (3) but allows subclasses (4) to alter the type of objects (2) that will be created – see Figure 9.

- The Product declares the interface, which is common to all objects that can be produced by the creator and its subclasses
- Concrete Products are different implementations of the product interface
- The Creator class declares the factory method that returns new product objects. It's important that the return type of this method matches the product interface
- Concrete Creators override the base factory method so it returns a different type of product

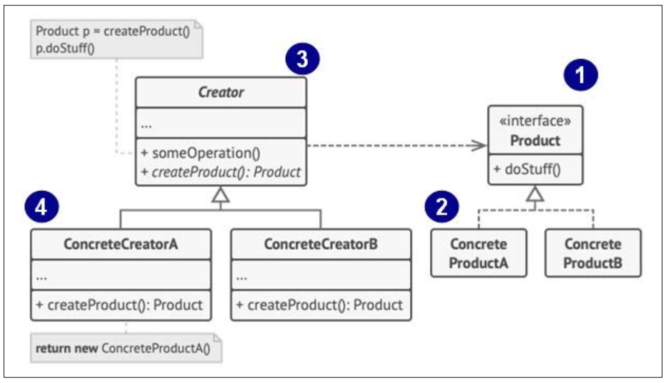

Figure 9: Factory Method Design Pattern

Factory Method Construction

The Figure 10 shows the class design to solve the use case for ABC Manufacturing, the abstract class Logistics acting as a creator class, this class has two abstract methods planDelivery() and createTransport. The Concrete Creator classes RoadLogistics and SeaLogistics implements the abstract methods. The Transport interface has a definition of deliver() method, which is then implemented by the Truck and Ship class with the specific mode, the construction of the Factory Method explained below

- **Logistics:** This is the 'Creator' class in the pattern. It defines an abstract method createTransport(). This method is what subclasses will override to create specific types of Transport objects. The planDelivery() method in this class uses the Transport object created by createTransport(). This method doesn't need to know the specifics of the transport type; it just knows that it can deliver.
- **Road Logistics and Sea Logistics:** These are the 'Concrete Creator' classes. These classes implement or override the createTransport() method inherited from Logistics. Road Logistics creates a Truck object, while Sea Logistics creates a Ship object. Each of these classes knows what type of Transport they should create and return.
- **Transport:** This is the 'Product' interface in the pattern. It defines the operations that all concrete products must implement, in this case, a deliver() method.
- **Truck and Ship:** These are the 'Concrete Product' classes. They provide implementations of the Transport interface. Specifically, they each provide their own version of the deliver() method, which will be different depending on whether it's delivering by road or by sea.

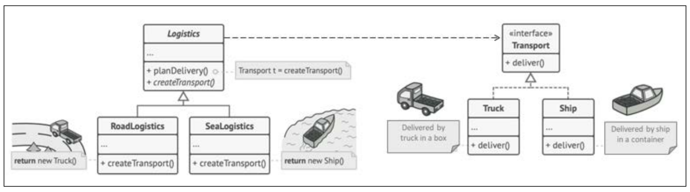

Figure 10: Applying Factory Method

Code Construction

The representation of the code using C# , Visual Studio Code and .Net Framework shown below-

```
using System;
// 'Product' interface
public interface ITransport
{
   void Deliver();
}

// 'ConcreteProduct' classes
public class Truck : ITransport
{
   public void Deliver()
   {
      Console.WriteLine("Delivered by truck in a box.");
   }
}

public class Ship : ITransport
{
   public void Deliver()
   {
      Console.WriteLine("Delivered by ship in a container.");
   }
}

// 'Creator' abstract class
public abstract class Logistics
{
   // The factory method
   public abstract ITransport CreateTransport();

   public void PlanDelivery()
   {
      // Call the factory method to create a transport object.
      var transport = CreateTransport();
      // Now use the product
      transport.Deliver();
   }
}

// 'ConcreteCreator' classes
public class RoadLogistics : Logistics
{
   public override ITransport CreateTransport()
   {
        return new Truck();
    }
}

public class SeaLogistics : Logistics
{
    public override ITransport CreateTransport()
    {
        return new Ship();
    }
}

public class Program
{
    public static void Main(string[] args)
    {
        // Create an instance of RoadLogistics
        Logistics roadLogistics = new RoadLogistics();
        roadLogistics.PlanDelivery();

        // Create an instance of SeaLogistics
        Logistics seaLogistics = new SeaLogistics();
        seaLogistics.PlanDelivery();

        Console.WriteLine("Press any key to exit...");
        Console.ReadKey();
    }
}
```

Design Pattern and Software Maintainability

The original study to evaluate the impact of design patterns on software maintenance was applied by Prechelt et al [4]. They conducted an experiment call PatMain by comparing the maintainability of two implementations of an application, one using a design pattern and the other using a simple alternative. They used four different subject systems in the same programming language. They addressed five patterns - decorator, composite, abstract factory, observer and visitor. The researchers measure the time and correctness of the given maintenance task for professional participants. They found that it was useful to use a design pattern but in case where simple solution is preferred, it is good to follow the software engineer common sense about whether to use a pattern or not, and in case of uncertainty it is better to use a pattern as a default approach.

Conclusion

A design pattern is a generalized reusable solution two commonly occurring problem in a software design. It can be defined as a description or template for how to solve a problem that can be used in many different situations [5]. In this paper, we aim to demonstrate the practical application of the Factory Method design pattern in a specific use case. Design patterns serve as invaluable communication tools and expedite the design process. They empower solution providers to focus on solving the business problem while promoting reusability in the design. Reusability extends not only to individual components but also to the entire design process, from problem-solving to the final solution. The ability to apply patterns that offer repeatable solutions is well worth the time invested in learning them. There are promising results indicating that the utilization of design patterns enhances quality and contributes to maintainability. The proportion of source code lines involved in design patterns within a system shows a strong correlation with maintainability. However, it's important to note that these findings represent just a small step in the empirical analysis of software quality concerning design patterns. Design patterns should facilitate the reuse of software architecture across different application domains and promote the reuse of flexible components.

References

1. Alexander C, Ishikawa S, Silverstein M, Jacobson M, Fiksdahl-King I, et al. (1977) A Pattern Language. https://arl.human.cornell.edu/linked%20docs/Alexander_A_Pattern_Language.pdf.
2. Gamma H (1995) Design Patterns Elements of Reusable Object-Oriented Software. https://www.javier8a.com/itc/bd1/articulo.pdf.
3. Schmuller J (1999) Sams Teach Yourself Uml in 24 Hours. https://nibmehub.com/opac-service/pdf/read/Sams%20teach%20yourself%20UML%20in%2024%20hours%20by%20Joseph%20Schmuller%20-A.pdf.
4. Prechelt L, Unger B, Tichy WF, Brossler P, Votta LG (2001) A controlled experiment in maintenance: comparing design patterns to simpler solutions. IEEE Transactions on Software Engineering 27: 1134-1144.
5. Zhang C, Budgen D (2012) What Do We Know about the Effectiveness of Software Design Patterns? IEEE Transactions on Software Engineering 38: 1213- 1231.