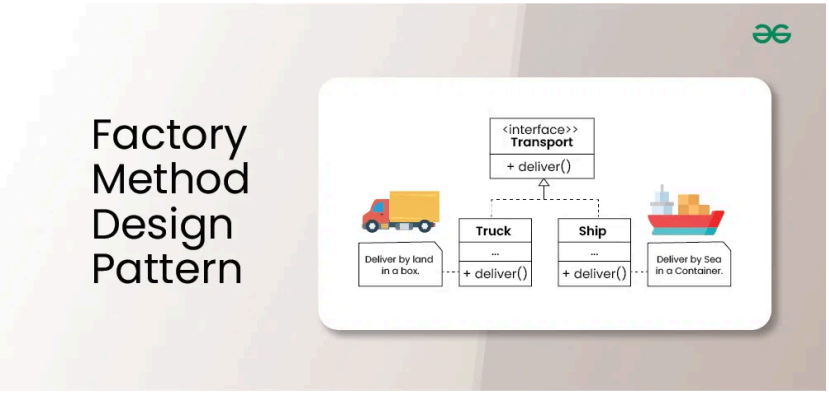
Factory method Design Pattern

Last Updated : 14 Oct, 2024

The Factory Method Design Pattern is a creational design pattern that provides an interface for creating objects in a superclass, allowing subclasses to alter the type of objects that will be created. This pattern is particularly useful when the exact types of objects to be created may vary or need to be determined at runtime, enabling flexibility and extensibility in object creation.



- Components of Factory Method Design Pattern
- Factory Method Design Pattern Example
- Use Cases of the Factory Method Design Pattern
- Advantages of Factory Method Design Pattern
- Disadvantages of Factory Method Design Pattern

What is the Factory Method Design Pattern?

The Factory Method Design Pattern is a creational design pattern used in software development. It provides an interface for creating objects in a superclass while allowing subclasses to specify the types of objects they create.

- This pattern simplifies the object creation process by placing it in a dedicated method, promoting loose coupling between the object creator and the objects themselves.
- This approach enhances flexibility, extensibility, and maintainability, enabling subclasses to implement their own factory methods for creating specific object types.

When to Use the Factory Method Design Pattern

Below is when to use factory method design pattern:

- If your object creation process is complex or varies under different conditions, using a factory method can make your client code simpler and promote reusability.
- The Factory Method Pattern allows you to create objects through an interface or abstract class, hiding the details of concrete implementations. This reduces dependencies and makes it easier to modify or expand the system without affecting existing code.
- If your application needs to create different versions of a product or may introduce new types in the future, the Factory Method Pattern

options to the factory method.

Components of Factory Method Design Pattern

Below are the main components of Factory Design Pattern:

- **Creator**: This is an abstract class or an interface that declares the factory method. The creator typically contains a method that serves as a factory for creating objects. It may also contain other methods that work with the created objects.
- **Concrete Creator**: Concrete Creator classes are subclasses of the Creator that implement the factory method to create specific types of objects. Each Concrete Creator is responsible for creating a particular product.
- **Product**: This is the interface or abstract class for the objects that the factory method creates. The Product defines the common interface for all objects that the factory method can create.
- **Concrete Product**: Concrete Product classes are the actual objects that the factory method creates. Each Concrete Product class implements the Product interface or extends the Product abstract class.

Factory Method Design Pattern Example

Below is the problem statement to understand Factory Method Design Pattern:

Consider a software application that needs to handle the creation of various types of vehicles, such as Two Wheelers, Three Wheelers, and Four Wheelers. Each type of vehicle has its own specific properties and behaviors.

1. Without Factory Method Design Pattern

```
/*package whatever //do not write package name here */

import java.io.*;

// Library classes
abstract class Vehicle {
    public abstract void printVehicle();
}

class TwoWheeler extends Vehicle {
    public void printVehicle() {
        System.out.println("I am two wheeler");
    }
}

class FourWheeler extends Vehicle {
    public void printVehicle() {
        System.out.println("I am four wheeler");
    }
}

// Client (or user) class
class Client {
    private Vehicle pVehicle;
```

```
29            } else if (type == 2) {
30                pVehicle = new FourWheeler();
31            } else {
32                pVehicle = null;
33            }
34        }
35
36        public void cleanup() {
37            if (pVehicle != null) {
38                pVehicle = null;
39            }
40        }
41
42        public Vehicle getVehicle() {
43            return pVehicle;
44        }
45    }
46
47    // Driver program
48    public class GFG {
49        public static void main(String[] args) {
50            Client pClient = new Client(1);
51            Vehicle pVehicle =
      pClient.getVehicle();
52            if (pVehicle != null) {
53                pVehicle.printVehicle();
54            }
55            pClient.cleanup();
56        }
57    }
```

```
1    I am two wheeler
```

Issues with the Current Design

- The `Client` class creates `TwoWheeler` and `FourWheeler` objects directly based on input. This strong dependency makes the code hard to maintain or update.
- The `Client` class not only decides which vehicle to create but also handles its lifecycle. This mixes responsibilities, which goes against the principle that a class should only have one reason to change.
- To add a new vehicle type, you must modify the `Client` class, which makes it difficult to scale the design. This conflicts with the idea that classes should be open for extension but closed for modification.

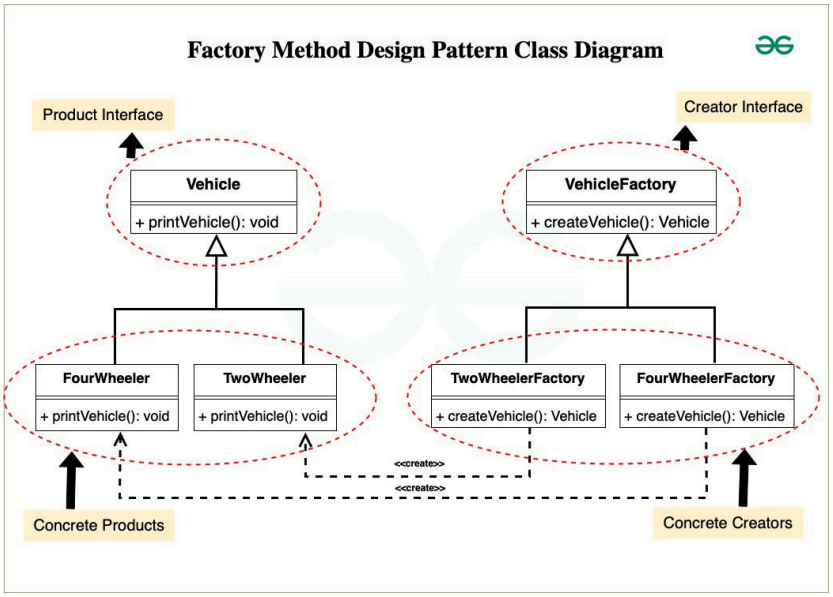
Solutions to the Problems

- **Define a Factory Interface**: Create an interface, `VehicleFactory`, with a method to produce vehicles.

- **Revise the Client Class**: Change the `Client` class to use a `VehicleFactory` instance instead of creating vehicles directly. This way, it can request vehicles without using conditional logic.
- **Enhance Flexibility**: This structure allows for easy addition of new vehicle types by simply creating new factory classes, without needing to alter existing `Client` code.

2. With Factory Method Design Pattern

Let's breakdown the code into component wise code:



1. Product Interface

```
// Product interface representing a vehicle
public abstract class Vehicle {
    public abstract void printVehicle();
}
```

```
// Concrete product classes representing
different types of vehicles
public class TwoWheeler extends Vehicle {
    public void printVehicle() {
        System.out.println("I am two wheeler");
    }
}

public class FourWheeler extends Vehicle {
    public void printVehicle() {
        System.out.println("I am four wheeler");
    }
}
```

3. Creator Interface (Factory Interface)

```
// Factory interface defining the factory method
public interface VehicleFactory {
    Vehicle createVehicle();
}
```

```
// Concrete factory class for TwoWheeler
public class TwoWheelerFactory implements VehicleFactory {
    public Vehicle createVehicle() {
        return new TwoWheeler();
    }
}

// Concrete factory class for FourWheeler
public class FourWheelerFactory implements VehicleFactory {
    public Vehicle createVehicle() {
        return new FourWheeler();
    }
}
```

Complete Code of this example:

```
// Library classes
abstract class Vehicle {
    public abstract void printVehicle();
}

class TwoWheeler extends Vehicle {
```

```
11
12  class FourWheeler extends Vehicle {
13      public void printVehicle() {
14          System.out.println("I am four
    wheeler");
15      }
16  }
17
18  // Factory Interface
19  interface VehicleFactory {
20      Vehicle createVehicle();
21  }
22
23  // Concrete Factory for TwoWheeler
24  class TwoWheelerFactory implements
    VehicleFactory {
25      public Vehicle createVehicle() {
26          return new TwoWheeler();
27      }
28  }
29
30  // Concrete Factory for FourWheeler
31  class FourWheelerFactory implements
    VehicleFactory {
32      public Vehicle createVehicle() {
33          return new FourWheeler();
34      }
35  }
36
37  // Client class
38  class Client {
39      private Vehicle pVehicle;
40
41      public Client(VehicleFactory factory) {
42          pVehicle = factory.createVehicle();
43      }
44
```

```
49
50  // Driver program
51  public class GFG {
52      public static void main(String[] args) {
53          VehicleFactory twoWheelerFactory = new
    TwoWheelerFactory();
54          Client twoWheelerClient = new
    Client(twoWheelerFactory);
55          Vehicle twoWheeler =
    twoWheelerClient.getVehicle();
56          twoWheeler.printVehicle();
57
58          VehicleFactory fourWheelerFactory = new
    FourWheelerFactory();
59          Client fourWheelerClient = new
    Client(fourWheelerFactory);
60          Vehicle fourWheeler =
    fourWheelerClient.getVehicle();
61          fourWheeler.printVehicle();
62      }
63  }
```

```
I am two wheeler
I am four wheeler
```

In the above code:

- `Vehicle` serves as the Product interface, defining the common method `printVehicle()` that all concrete products must implement.
- `TwoWheeler` and `FourWheeler` are concrete product classes representing different types of vehicles, implementing the `printVehicle()` method.
- `VehicleFactory` acts as the Creator interface (Factory Interface) with a method `createVehicle()` representing the factory method.
- `TwoWheelerFactory` and `FourWheelerFactory` are concrete creator classes (Concrete Factories) implementing the `VehicleFactory` interface to create instances of specific types of vehicles.

Use Cases of the Factory Method

Below are the main use cases of factory method design pattern:

- Libraries like Swing and JavaFX use factories to create flexible UI components.
- Tools like Log4j rely on factories to create configurable loggers.
- Factories help create objects from serialized data, supporting various formats.

Advantages of the Factory Method

Below are the main advantages of factory method design pattern:

- Separates creation logic from client code, improving flexibility.
- New product types can be added easily.
- Simplifies unit testing by allowing mock product creation.
- Centralizes object creation logic across the application.
- Hides specific product classes from clients, reducing dependency.

Disadvantages of the Factory Method

Below are the main advantages of factory method design pattern:

- Adds more classes and interfaces, which can complicate maintenance.
- Slight performance impacts due to polymorphism.
- Concrete creators are linked to their products.
- Clients need knowledge of specific subclasses.
- May lead to unnecessary complexity if applied too broadly.
- Factory logic can be harder to test.

Comment More info

Next Article

Similar Reads

Software Design Patterns Tutorial

Software design patterns are important tools developers, providing proven solutions to common problems encountered during software development. This article will act as tutorial to help you...

9 min read

Complete Guide to Design Patterns

Design patterns help in addressing the recurring issues in software design and provide a shared vocabulary for developers to communicate and collaborate effectively. They have been documented...

11 min read

Types of Software Design Patterns

Designing object-oriented software is hard, and designing reusable object-oriented software is even harder. Christopher Alexander says, "Each pattern describes a problem which occurs over and over...

9 min read

---

1. Creational Design Patterns

Creational Design Patterns

Creational Design Patterns focus on the process of object creation or problems related to object creation. They help in making a system independent of how its objects are created, composed, and...

4 min read

Types of Creational Patterns

Factory method Design Pattern

The Factory Method Design Pattern is a creational design pattern that provides an interface for creating objects in a superclass, allowing subclasses to alter the type of objects that will be created....

8 min read

Singleton Method Design Pattern in JavaScript

Singleton Method or Singleton Design Pattern is a part of the Gang of Four design pattern and it is categorized under creational design patterns. It is one of the most simple design patterns in terms o...

10 min read

Singleton Method Design Pattern

The Singleton Method Design Pattern ensures a class has only one instance and provides a global access point to it. Itâ€™s ideal for scenarios requiring centralized control, like managing database...

11 min read

Prototype Design Pattern

The Prototype Design Pattern is a creational pattern that enables the creation of new objects by copying an existing object. Prototype allows us to hide the complexity of making new instances from...

8 min read

Builder Design Pattern

The Builder Design Pattern is a creational pattern used in software design to construct a complex object step by step. It allows the construction of a product in a step-by-step manner, where the...

7 min read

2. Structural Design Patterns

Structural Design Patterns

Structural Design Patterns are solutions in software design that focus on how classes and objects are organized to form larger, functional structures. These patterns help developers simplify relationships...

7 min read

Types of Structural Patterns

3. Behvioural Design Patterns

Corporate & Communications Address:
A-143, 7th Floor, Sovereign Corporate Tower, Sector- 136, Noida, Uttar Pradesh (201305)

Registered Address:
K 061, Tower K, Gulshan Vivante Apartment, Sector 137, Noida, Gautam Buddh Nagar, Uttar Pradesh, 201305

Company
About Us
Legal
Privacy Policy
In Media
Contact Us
Advertise with us
GFG Corporate Solution
Placement Training Program
GeeksforGeeks Community

Languages
Python
Java
C++
PHP
GoLang
SQL
R Language
Android Tutorial
Tutorials Archive

DSA
Data Structures
Algorithms
DSA for Beginners
Basic DSA Problems
DSA Roadmap
Top 100 DSA Interview Problems
DSA Roadmap by Sandeep Jain
All Cheat Sheets

Data Science & ML
Data Science With Python
Data Science For Beginner
Machine Learning
ML Maths
Data Visualisation
Pandas
NumPy
NLP

JavaScript
TypeScript
ReactJS
NextJS
Bootstrap
Web Design

Python Tkinter
Web Scraping
OpenCV Tutorial
Python Interview Question
Django

Computer Science

Operating Systems
Computer Network
Database Management System
Software Engineering
Digital Logic Design
Engineering Maths
Software Development
Software Testing

DevOps

Git
Linux
AWS
Docker
Kubernetes
Azure
GCP
DevOps Roadmap

System Design

High Level Design
Low Level Design
UML Diagrams
Interview Guide
Design Patterns
OOAD
System Design Bootcamp
Interview Questions

Inteview Preparation

Competitive Programming
Top DS or Algo for CP
Company-Wise Recruitment Process
Company-Wise Preparation
Aptitude Preparation
Puzzles

School Subjects

Mathematics
Physics
Chemistry
Biology
Social Science
English Grammar
Commerce
World GK

GeeksforGeeks Videos

DSA
Python
Java
C++
Web Development
Data Science
CS Subjects