



# Abstract Factory Pattern

Last Updated : 03 Jan, 2025

The Abstract Factory Pattern is one of the [creational design patterns](#) that provides an interface for creating families of related or dependent objects without specifying their concrete classes and implementation, in simpler terms the Abstract Factory Pattern is a way of organizing how you create groups of things that are related to each other.



## Table of Content

- [What is the Abstract Factory Pattern?](#)
- [Components of Abstract Factory Pattern](#)
- [Abstract Factory example](#)
- [Advantages of using Abstract Factory Pattern](#)
- [Disadvantages of using Abstract Factory Pattern](#)
- [When to use Abstract Factory Pattern](#)
- [When not to use Abstract Factory Pattern](#)

## What is the Abstract Factory Pattern?

The Abstract Factory Pattern is a way of organizing how you create groups of things that are related to each other. It provides a set of rules or instructions

that let you create different types of things without knowing exactly what those things are. This helps you keep everything organized and lets you switch between different types easily.

- Abstract Factory pattern is almost same as [Factory Pattern](#) and is considered as another layer of abstraction over factory pattern.
- Abstract Factory patterns work around a super-factory which creates other factories.
- At runtime, the abstract factory is coupled with any desired concrete factory which can create objects of the desired type.

## Components of Abstract Factory Pattern

To understand abstract factory pattern, we have to understand the components of it and relationships between them.

- **Abstract Factory:**
  - Abstract Factory provides a high-level blueprint that defines rules for creating families of related object without specifying their concrete classes.
  - It provides a way such that concrete factories follow a common interface, providing consistent way to produce related set of objects.
- **Concrete Factories:**
  - Concrete Factories implement the rules specified by the abstract factory. It contain the logic for creating specific instances of objects within a family.
  - Also multiple concrete factories can exist, each produce a distinct family of related objects.
- **Abstract Products:**
  - Abstract Products represents a family of related objects by defining a set of common methods or properties.
  - It acts as an abstract or interface type that all concrete products within a family must follow to and provides a unified way for concrete products to be used interchangeably.
- **Concrete Products:**

- They are the actual instances of objects created by concrete factories.
- They implement the methods declared in the abstract products, ensuring consistency within a family and belong to a specific category or family of related objects.
- **Client:**
  - Client utilizes the abstract factory to create families of objects without specifying their concrete types and interacts with objects through abstract interfaces provided by abstract products.

## Example of Abstract Factory Design Pattern

Let's understand Abstract Factory Design Pattern using an example:

*Imagine you're managing a global car manufacturing company*

- *You want to design a system to create cars with specific configurations for different regions, such as North America and Europe.*
- *Each region may have unique requirements and regulations, and you want to ensure that cars produced for each region meet those standards.*

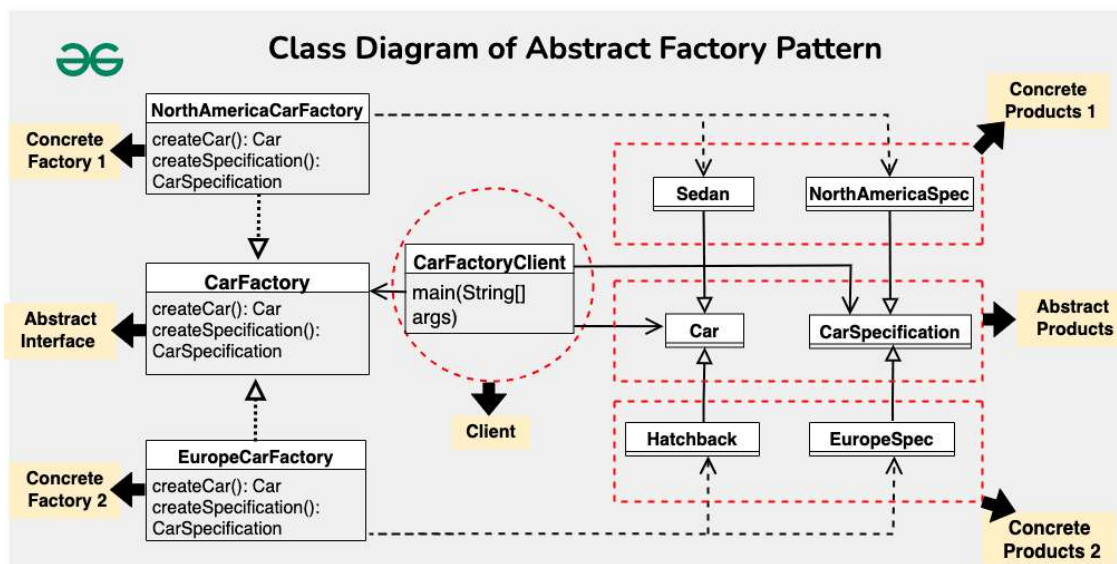
### What can be the challenges while implementing this system?

- Different regions have different cars with different features, so designing this can be challenging.
- The other main challenge is to ensure consistency in the production of cars and their specifications within each region.
- There can be updation in having new cars in different regions so adapting the system to changes in regulations or introducing new features for a specific region becomes challenging.
- So, Modifications would need to be made in multiple places, increasing the chances of introducing bugs and making the system more prone to errors.

### How Abstract Factory Pattern help to solve above challenges?

Below is how abstract factory pattern help to solve the above challenges. After using this pattern:

- Different regions has their own factory to create cars for local needs.
- This helps to keeps the design and features the same for vehicles in each region.
- You can change one region without affecting others (e.g., updating North America doesn't impact Europe).
- To add a new region, just create a new factory, no need to change existing code.
- The pattern keeps car creation separate from how they are used.



Below is the code of above problem statement using Abstract Factory Pattern:

### 1. Abstract Factory Interface (CarFactory)

“**CarFactory**” is a Abstract Factory Interface that defines methods for creating cars and their specifications.

```

1  // Abstract Factory Interface
2  interface CarFactory {
3      Car createCar();
4      CarSpecification createSpecification();

```

```
5    }
```

## 2. Concrete Factories (NorthAmericaCarFactory and EuropeCarFactory)

“NorthAmericaCarFactory” and “EuropeCarFactory” are concrete factories that implement the abstract factory interface “CarFactory” to create cars and specifications specific to North America, Europe.

```
1  // Concrete Factory for North America Cars
2  class NorthAmericaCarFactory implements CarFactory {
3      public Car createCar() {
4          return new Sedan();
5      }
6
7      public CarSpecification createSpecification() {
8          return new NorthAmericaSpecification();
9      }
10 }
11
12 // Concrete Factory for Europe Cars
13 class EuropeCarFactory implements CarFactory {
14     public Car createCar() {
15         return new Hatchback();
16     }
17
18     public CarSpecification createSpecification() {
19         return new EuropeSpecification();
20     }
21 }
22
23 }
```

## 3. Abstract Products (Car and CarSpecification interfaces)

Define interfaces for cars and specifications to ensure a common structure.

```
1 // Abstract Product Interface for Cars
2 interface Car {
3     void assemble();
4 }
5
6 // Abstract Product Interface for Car Specifications
7 interface CarSpecification {
8     void display();
9 }
```

#### 4. Concrete Products (Sedan, Hatchback, NorthAmericaSpecification, EuropeSpecification)



“Sedan”, “Hatchback”, “NorthAmericaSpecification”, “EuropeSpecification” are concrete products that implement the interfaces to create specific instances of cars and specifications.

```
1 // Concrete Product for Sedan Car
2 class Sedan implements Car {
3     public void assemble() {
4         System.out.println("Assembling Sedan car.");
5     }
6 }
7
8 // Concrete Product for Hatchback Car
9 class Hatchback implements Car {
10     public void assemble() {
11         System.out.println("Assembling Hatchback car.");
12     }
13 }
14
15 // Concrete Product for North America Car Specification
16 class NorthAmericaSpecification implements
17 CarSpecification {
18     public void display() {
19         System.out.println("North America Car
20 Specification: Safety features compliant with local
regulations.");
21     }
22 }
```

```
21
22 // Concrete Product for Europe Car Specification
23 class EuropeSpecification implements CarSpecification {
24     public void display() {
25         System.out.println("Europe Car Specification: Fuel
26         efficiency and emissions compliant with EU standards.");
27     }
28 }
```

## Complete code for the above example

Below is the complete code for the above example:



```
1 // Abstract Factory Interface
2 interface CarFactory {
3     Car createCar();
4     CarSpecification createSpecification();
5 }
6
7 // Concrete Factory for North America Cars
8 class NorthAmericaCarFactory implements CarFactory {
9     public Car createCar() {
10         return new Sedan();
11     }
12
13     public CarSpecification createSpecification() {
14         return new NorthAmericaSpecification();
15     }
16 }
17
18 // Concrete Factory for Europe Cars
19 class EuropeCarFactory implements CarFactory {
20     public Car createCar() {
21         return new Hatchback();
22     }
23
24     public CarSpecification createSpecification() {
25         return new EuropeSpecification();
26     }
27 }
```

```
28
29 // Abstract Product Interface for Cars
30 interface Car {
31     void assemble();
32 }
33
34 // Abstract Product Interface for Car Specifications
35 interface CarSpecification {
36     void display();
37 }
38
39 // Concrete Product for Sedan Car
40 class Sedan implements Car {
41     public void assemble() {
42         System.out.println("Assembling Sedan car.");
43     }
44 }
45
46 // Concrete Product for Hatchback Car
47 class Hatchback implements Car {
48     public void assemble() {
49         System.out.println("Assembling Hatchback car.");
50     }
51 }
52
53 // Concrete Product for North America Car Specification
54 class NorthAmericaSpecification implements
CarSpecification {
55     public void display() {
56         System.out.println("North America Car
Specification: Safety features compliant with local
regulations.");
57     }
58 }
59
60 // Concrete Product for Europe Car Specification
61 class EuropeSpecification implements CarSpecification {
62     public void display() {
63         System.out.println("Europe Car Specification: Fuel
efficiency and emissions compliant with EU standards.");
64     }
65 }
66
67
```



```
68 // Client Code
69 public class CarFactoryClient {
70     public static void main(String[] args) {
71         // Creating cars for North America
72         CarFactory northAmericaFactory = new
NorthAmericaCarFactory();
73         Car northAmericaCar =
northAmericaFactory.createCar();
74         CarSpecification northAmericaSpec =
northAmericaFactory.createSpecification();
75
76         northAmericaCar.assemble();
77         northAmericaSpec.display();
78
79         // Creating cars for Europe
80         CarFactory europeFactory = new EuropeCarFactory();
81         Car europeCar = europeFactory.createCar();
82         CarSpecification europeSpec =
europeFactory.createSpecification();
83
84         europeCar.assemble();
85         europeSpec.display();
86     }
87 }
```

## Output

Assembling Sedan car.

North America Car Specification: Safety features compliant with local regulations.

Assembling Hatchback car.

Europe Car Specification: Fuel efficiency and emissions compliant wit...

## Benefits of using Abstract Factory Pattern

Below are the main benefits of abstract factory pattern:

- The Abstract Factory pattern separates the creation of objects, so clients don't need to know specific classes.
- Clients interact with objects through abstract interfaces, keeping class names hidden from client code.

- Changing the factory allows for different product configurations, as all related products change together.
- The pattern ensures that an application uses objects from only one family at a time for better compatibility.

## Challenges of using Abstract Factory Pattern

Below are the main challenges of using abstract factory pattern:

- The Abstract Factory pattern can add unnecessary complexity to simpler projects with multiple factories and interfaces.
- Adding new product types may require changes to both concrete factories and the abstract factory interface, impacting existing code.
- Introducing more factories and product families can quickly increase the number of classes, making code management difficult in smaller projects.
- It may violate the Dependency Inversion Principle if client code depends directly on concrete factories rather than abstract interfaces.

## When to use Abstract Factory Pattern

Choose using abstract factory pattern when:

- When your system requires multiple families of related products and you want to ensure compatibility between them.
- When you need flexibility and extensibility, allowing for new product variants to be added without changing existing client code.
- When you want to encapsulate the creation logic, making it easier to modify or extend the object creation process without affecting the client.
- When you aim to maintain consistency across different product families, ensuring a uniform interface for the products.

## When not to use Abstract Factory Pattern

Avoid using abstract factory pattern when:

- When the product families are unlikely to change, as it may add unnecessary complexity.
- When your application only requires single, independent objects and isn't concerned with families of related products.

- When overhead of maintaining multiple factories outweighs the benefits, particularly in smaller applications.
- When simpler solutions, like the Factory Method or Builder pattern, if they meet your needs without adding the complexity of the Abstract Factory pattern.

[Comment](#)[More info](#)[Advertise with us](#)

## Next Article

Singleton Method Design Pattern in  
JavaScript

## 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 understand the concep...

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 and refined over tim...

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 again in our...

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 represented. Creational...

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. This pattern is...

8 min read

### Abstract Factory Pattern

The Abstract Factory Pattern is one of the creational design patterns that provides an interface for creating families of related or dependent objects without specifying their concrete classes and implementation, in...

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 of...

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. It's ideal for scenarios requiring centralized control, like managing database connections or...

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 the client. The...

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 construction process ca...

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 between...

7 min read

---

## Types of Structural Patterns

---

### 3. Behavioural Design Patterns

## Behavioral Design Patterns

Behavioral design patterns are a category of design patterns that focus on the interactions and communication between objects. They help define how objects collaborate and distribute responsibility among them, makin...

5 min read

---

## 3. Types of Behavioural Patterns

---

## Top Design Patterns Interview Questions [2024]

A design pattern is basically a reusable and generalized solution to a common problem that arises during software design and development. Design patterns are not specific to a particular programming language or...

9 min read

---

## Software Design Pattern in Different Programming Languages

## Java Design Patterns Tutorial

Design patterns in Java refer to structured approaches involving objects and classes that aim to solve recurring design issues within specific contexts. These patterns offer reusable, general solutions to common problems...

8 min read

---

## Python Design Patterns Tutorial

Design patterns in Python are communicating objects and classes that are customized to solve a general design problem in a particular context. Software design patterns are general, reusable solutions to common...

8 min read

---

## Modern C++ Design Patterns Tutorial

Design patterns in C++ help developers create maintainable, flexible, and understandable code. They encapsulate the expertise and experience of seasoned software architects and developers, making it easier f...

7 min read

---

## JavaScript Design Patterns Tutorial

Design patterns in Javascript are communicating objects and classes that are customized to solve a general design problem in a particular context. Software design patterns are general, reusable solutions to common...

8 min read

---

## Software Design Pattern Books

---

## Software Design Pattern in Development

---

## Some other Popular Design Patterns

---

## Design Patterns in Different Languages

---



### 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



Advertise with us

### Company

[About Us](#)  
[Legal](#)  
[Privacy Policy](#)  
[In Media](#)  
[Contact Us](#)  
[Advertise with us](#)  
[GFG Corporate Solution](#)

### Languages

[Python](#)  
[Java](#)  
[C++](#)  
[PHP](#)  
[GoLang](#)  
[SQL](#)  
[R Language](#)

Placement Training Program  
GeeksforGeeks Community

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

## Web Technologies

HTML  
CSS  
JavaScript  
TypeScript  
ReactJS  
NextJS  
Bootstrap  
Web Design

## Computer Science

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

## System Design

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

## School Subjects

Mathematics  
Physics  
Chemistry  
Biology

## Data Science & ML

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

## Python Tutorial

Python Programming Examples  
Python Projects  
Python Tkinter  
Web Scraping  
OpenCV Tutorial  
Python Interview Question  
Django

## DevOps

Git  
Linux  
AWS  
Docker  
Kubernetes  
Azure  
GCP  
DevOps Roadmap

## Interview Preparation

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

## GeeksforGeeks Videos

DSA  
Python  
Java  
C++

Social Science  
English Grammar  
Commerce  
World GK

Web Development  
Data Science  
CS Subjects

---

@GeeksforGeeks, Sanchhaya Education Private Limited, All rights reserved