HowToDoInJava

Abstract Factory Pattern Explained

Last Updated: August 31, 2021 By: Lokesh Gupta Creational Patterns Design Patterns

Abstract factory pattern is yet another creational design pattern and is considered as another layer of abstraction over factory pattern. In this tutorial, we will expand the scope of car factory problem discussed in factory pattern. We will learn when to use factory pattern by expanding scope of car factory and then how abstract factory pattern solves the expanded scope.

Table of Contents

1. Design global car factory using abstract factory pattern

In "factory design pattern", we discussed how to abstract the car making process for various car model types and their additional logic included in car making process. Now, imagine if our car maker decides to go global.

To **support global operations**, we will require to enhance the system to support different car making styles for different countries. For example, in some countries we see steering wheel on left side, and in some countries it is on right side. There can be many more such differences in different part of cars and their making processes.

To describable the abstract factory pattern, we will consider 3 kind of makes – the *USA*, *Asia*, and the *default* for all other countries. Supporting multiple locations will need critical design changes.

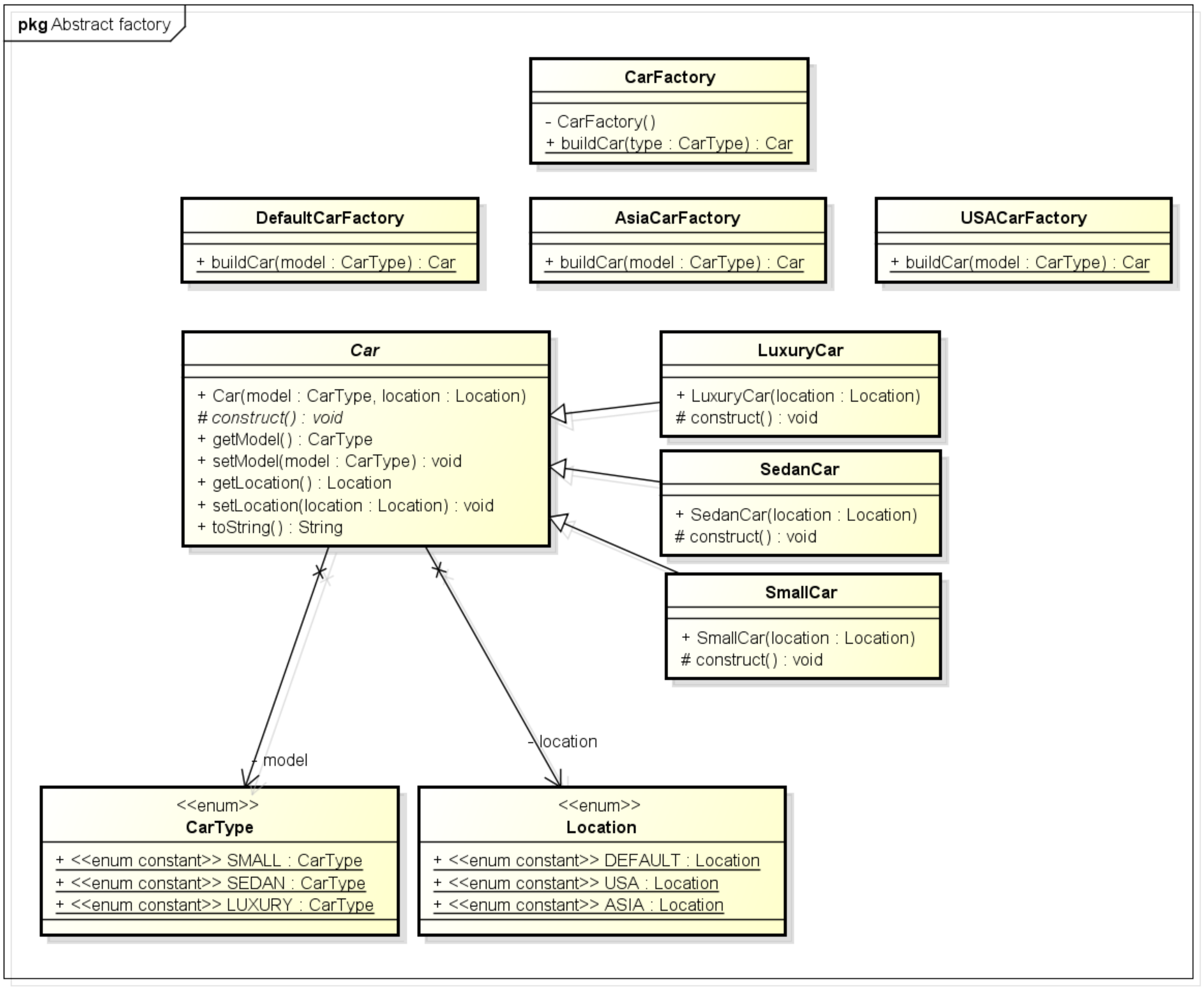
First of all, we need car factories in each **location** specified in the problem statement. i.e. *USACarFactory*, *AsiaCarFactory* and *DefaultCarFactory*. Now, our application should be smart enough to identify the location where is being used, so we should be able to use appropriate car factory without even knowing which car factory implementation will be used internally. This also saves us from someone calling the wrong factory for a particular location.

So basically, we need another layer of abstraction which will identify the location and internally use correct car factory implementation without even giving a single hint to the user. This is exactly the problem, which abstract factory pattern is used to solve.

2. Abstract factory pattern based solution

2.1. Package Diagram

Class diagram for participating classes in design of global car factory using abstract factory pattern.


pkg Abstract factory
CarFactory
- CarFactory()
+ buildCar(type : CarType) : Car
DefaultCarFactory
+ buildCar(model : CarType) : Car
AsiaCarFactory
+ buildCar(model : CarType) : Car
USACarFactory
+ buildCar(model : CarType) : Car
Car
+ Car(model : CarType, location : Location)
# construct() : void
+ getModel() : CarType
+ setModel(model : CarType) : void
+ getLocation() : Location
+ setLocation(location : Location) : void
+ toString() : String
LuxuryCar
+ LuxuryCar(location : Location)
# construct() : void
SedanCar
+ SedanCar(location : Location)
# construct() : void
SmallCar
+ SmallCar(location : Location)
# construct() : void
- model
- location
<<enum>>
CarType
+ <<enum constant>> SMALL : CarType
+ <<enum constant>> SEDAN : CarType
+ <<enum constant>> LUXURY : CarType
<<enum>>
Location
+ <<enum constant>> DEFAULT : Location
+ <<enum constant>> USA : Location
+ <<enum constant>> ASIA : Location

2.2. Sequence Diagram

This diagram shows the interaction between classes and abstraction behind `CarFactory` factory class.

> Please note that I have designed the solution to completely hide the location detail with end user. So, I have not exposed any location specific factory directly.
>
> In alternate solution, we can first get the location specific factory based on location argument and then use it's `buildCar()` method on abstract reference to build the actual car instance.

3. Abstract factory pattern implementation

Java classes implementing abstract factory pattern for global car factory.

First, wee have to write all separate car factories for different locations. To support, location specific features, begin with modifying our `Car.java` class with another attribute – `location`.

```
Car.java

public abstract class Car {
    public Car(CarType model, Location location){
      this.model = model;
      this.location = location;
    }

    protected abstract void construct();

    private CarType model = null;
    private Location location = null;

    //getters and setters

    @Override
    public String toString() {
      return "Model- "+model + " built in "+location;
    }
  }
```

This adds extra work of creating another enum for storing different locations.

```
Location.java

public enum Location {
  DEFAULT, USA, ASIA
}
```

All car types will also have additional `location` property. We are writing only for the luxury car. Same follows for small and sedan also.

```
LuxuryCar.java

public class LuxuryCar extends Car
{
  public LuxuryCar(Location location)
  {
    super(CarType.LUXURY, location);
    construct();
  }

  @Override
  protected void construct() {
    System.out.println("Building luxury car");
    //add accessories
  }
}
```

So far we have created basic classes. Now let's have different car factories which is the core idea behind abstract factory pattern.

```
AsiaCarFactory.java

public class AsiaCarFactory
{
  public static Car buildCar(CarType model)
  {
    Car car = null;
    switch (model)
    {
      case SMALL:
      car = new SmallCar(Location.ASIA);
      break;

      case SEDAN:
      car = new SedanCar(Location.ASIA);
      break;

      case LUXURY:
      car = new LuxuryCar(Location.ASIA);
      break;

      default:
      //throw some exception
      break;
    }
    return car;
  }
}
```

```
DefaultCarFactory.java

public class DefaultCarFactory
{
  public static Car buildCar(CarType model)
  {
    Car car = null;
    switch (model)
    {
      case SMALL:
      car = new SmallCar(Location.DEFAULT);
      break;

      case SEDAN:
      car = new SedanCar(Location.DEFAULT);
      break;

      case LUXURY:
      car = new LuxuryCar(Location.DEFAULT);
      break;

      default:
      //throw some exception
      break;
    }
    return car;
  }
}
```

```
USACarFactory.java

public class USACarFactory
{
  public static Car buildCar(CarType model)
  {
    Car car = null;
    switch (model)
    {
      case SMALL:
      car = new SmallCar(Location.USA);
      break;

      case SEDAN:
      car = new SedanCar(Location.USA);
      break;

      case LUXURY:
      car = new LuxuryCar(Location.USA);
      break;

      default:
      //throw some exception
      break;
    }
  return car;
  }
}
```

Well, now we have all 3 different Car factories. Now, we have to abstract the way these factories are accessed.

```
CarFactory.java

public class CarFactory
{
  private CarFactory() {
    //Prevent instantiation
  }

  public static Car buildCar(CarType type)
  {
    Car car = null;
    Location location = Location.ASIA; //Read location property somewhere from co
    //Use location specific car factory
    switch(location)
    {
      case USA:
        car = USACarFactory.buildCar(type);
        break;
      case ASIA:
        car = AsiaCarFactory.buildCar(type);
        break;
      default:
        car = DefaultCarFactory.buildCar(type);
    }
  return car;
  }
}
```

We are done with writing code. Now, let's test the factories and cars.

```
TestFactoryPattern.java

public class TestFactoryPattern
{
  public static void main(String[] args)
  {
    System.out.println(CarFactory.buildCar(CarType.SMALL));
    System.out.println(CarFactory.buildCar(CarType.SEDAN));
    System.out.println(CarFactory.buildCar(CarType.LUXURY));
  }
}
```

Program output:

```
Console
```

```
Output: (Default location is Asia)

Building small car
Model- SMALL built in ASIA

Building sedan car
Model- SEDAN built in ASIA

Building luxury car
Model- LUXURY built in ASIA
```

4. Summary

We already have seen the use case scenarios of Factory pattern so whenever you need **another level of abstraction over a group of factories**, you should consider using the *abstract factory pattern*. It is probably only **difference between factory pattern vs abstract factory pattern**.

You can already look deeper into different *real time examples of abstract factory* in JDK distribution:

- DocumentBuilderFactory#newInstance()
- TransformerFactory#newInstance()

There are other similar examples but the need is to have the feel of the **abstract factory design pattern**, which you must have got till now.

Happy Learning!!

References:

Abstract factory – Wikipedia

Was this post helpful?

Let us know if you liked the post. That's the only way we can improve.

Yes

No

Recommended Reading:

1. Java Factory Pattern Explained
2. Java abstract keyword – abstract classes and methods
3. Java Singleton Pattern Explained
4. Interface vs Abstract Class in Java
5. TestNG @Factory
6. TestNG – @Factory vs @DataProvider
7. Spring static factory-method example
8. Immutable Collections with Factory Methods in Java 9
9. Prototype design pattern in Java
0. Builder Design Pattern

Subscribe

** We do not spam !!*

18 thoughts on “Abstract Factory Pattern Explained”

Saheb

September 5, 2017 at 12:53 am

As per Abstract Factory pattern , It provides an interface for creating families of related or dependent objects without specifying their concrete class. but here the factory is returning only Car type. I think this a Factory method pattern with multiple concrete factory.
Reference: https://en.wikipedia.org/wiki/Abstract_factory_pattern

Could anyone please explain which one is correct implementation and why?

Reply

Chauhan

July 8, 2015 at 2:43 pm

Hi Lokesh,

I just want to understand whether location should be part of "Car"?
Car has nothing to do with location then why did you put location as part of Car?

//Bhupesh

Reply

Lokesh Gupta

July 8, 2015 at 2:51 pm

Given that each car instance in this application is going to be strictly location specific, how would you determine the location attribute of a random car instance in runtime of application if it's not tied to car instance itself.

Reply

Fanquilen

July 6, 2015 at 2:47 pm

Shouldn't class CarFactory be abstract instead of setting the constructor to private?

Reply

rAceR

April 15, 2015 at 1:02 pm

CarFactory method needs to be modified every time for switch case , when a location gets added. How could we make it extensible without modification.

Reply

Supun Gajaweera

March 4, 2015 at 5:47 pm

Hello Lokesh,
Thank you for the great tutorial!
but I have a question. Can we implement an interfaces instead of Abstract Car and Abstract Factory classes?

Supun

Reply

Lokesh Gupta

March 5, 2015 at 1:54 am

Hi Supun, In my view interfaces should be used for adding the behavior to classes only. Here, we are creating instances of Car only in different ways. I think that abstract class make more sense here.
BUT, there is no such restriction from design pattern side, you are free to implement using interfaces as well if you find it more easy. It's matter of choice only.

Reply

Supun

March 6, 2015 at 1:08 pm

Thank you for clearing that for me lokesh. I see your point. Keep up with the great work!

Cheers
Supun

Reply

deepak

November 4, 2014 at 3:01 pm

Nice article about abstract factory design pattern.

Reply

Alps

October 4, 2014 at 11:11 am

Thanks Lokesh.Very nice explanation in simple words.

Reply

Nilesh Patil

July 2, 2014 at 4:07 am

Hi Lokesh, Thank you for such nice and simple explanation. I have one question, for example I have two different factories one for fruit, one for Vegetable, question is how can I use abstract factory pattern here.

Reply

Lokesh Gupta

July 2, 2014 at 8:50 am

Talking about myself, I do not see any value addition in making abstract factory for your particular case. Fruits and Vegetables are two different entities and there are many major differences between them. So making two separate factories; one for each, makes more sense to me.

Also, factory (or abstract factory) pattern should be applied on objects which you can build in different ways; in other words; there can be different representations of a object which belong a common parent.

Your question is very good but to me, it's really doesn't fit into context.

Reply

abc

June 5, 2014 at 10:50 am

how to get Location from a configuration file in CarFactory

Reply

Lokesh Gupta

June 5, 2014 at 11:03 am

Take help of this article: https://howtodoinjava.com/java/library/auto-reload-configuration-when-any-change-happen-part-2/

Reply

narayana

December 9, 2013 at 12:22 pm

It is surprised to me to see no abstract classes except CAR in your example ..

Reply

Lokesh Gupta

December 9, 2013 at 9:41 pm

Abstract factory does not mandate abstract class usage. It's pattern. Implementation can be of your choice.

Reply

Gaurav

November 19, 2013 at 12:28 pm

Thanks for the article.Its explained well using the example of car.Now I need a real time scenario where this pattern is used(like factory is used in SessionFactory in hibernate)

Reply

Lokesh Gupta

November 19, 2013 at 10:54 pm

Refer here [best link]: https://stackoverflow.com/questions/1673841/examples-of-gof-design-patterns-in-javas-core-libraries

Reply

Leave a Comment

Name *

Email *

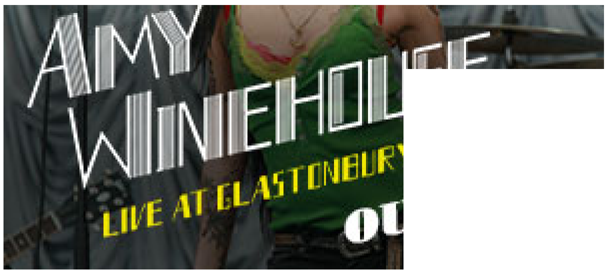
Website

☐ Add me to your newsletter and keep me updated whenever you publish new blog posts

Post Comment

Search …

Modernize
your archive
mimecast


AMY WINEHOUSE
LIVE AT GLASTONBURY
OU

Guest Posts

Blogs

REST API Tutorial