Java Abstract Factory Pattern - Mẫu thiết kế Abstract Factory trong Java

Abstract Factory cung cấp một interface cho việc tạo lập các đối tượng (có liên hệ với nhau) mà không cần qui định lớp khi hay xác định lớp cụ thể (concrete) tạo mỗi đối tượng.

1. Tại sao cần có Abstract Factory Pattern

Hãy tưởng tượng, Abstract factory như là một nhà máy lớn chứa nhiều nhà máy nhỏ, trong các nhà máy đó có những xưởng sản xuất, các xưởng đó tạo ra những sản phẩm khác nhau.

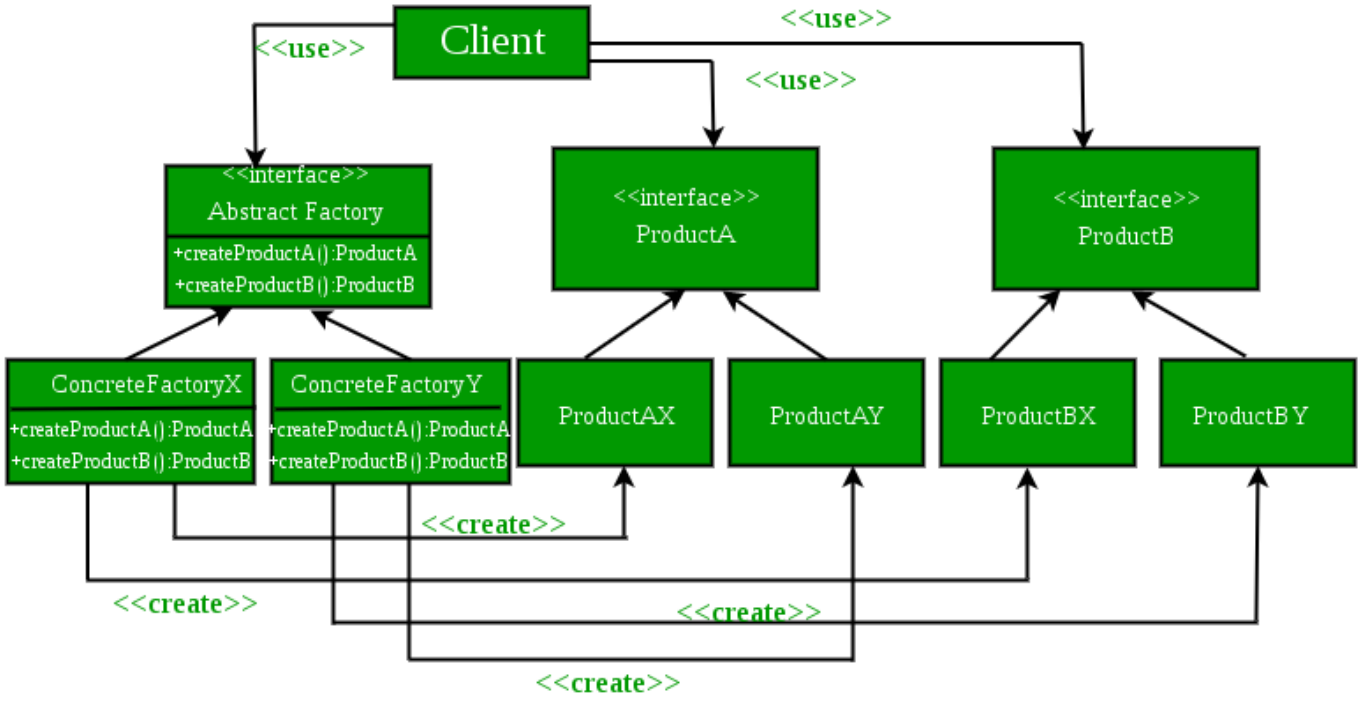
- Cung cấp hướng tiếp cận với Interface thay thì các implement, che giấu sự phức tạp của việc khởi tạo các đối tượng với người dùng (client), độc lập giữa việc khởi tạo đối tượng và hệ thống sử dụng, ...
- Giúp tránh được việc sử dụng điều kiện logic bên trong **Factory Pattern**. Khi một **Factory Method** lớn (có quá nhiều sử lý if-else hay switch-case), chúng ta nên sử dụng theo mô hình **Abstract Factory** để dễ quản lý hơn (cách phân chia có thể là gom nhóm các sub-class cùng loại vào một Factory).
- Abstract Factory Pattern là factory của các factory, có thể dễ dạng mở rộng để chứa thêm các factory và các sub-class khác.
- Dễ dàng xây dựng một hệ thống đóng gói (encapsulate): sử dụng được với nhiều nhóm đối tượng (factory) và tạo nhiều product khác nhau.

2. Thực thi Abstract Factory Pattern

Một Abstract Factory Pattern bao gồm các thành phần cơ bản sau:

- **AbstractFactory**: Khai báo dạng interface hoặc abstract class chứa các phương thức để tạo ra các đối tượng abstract.
- **ConcreteFactory**: Xây dựng, cài đặt các phương thức tạo các đối tượng cụ thể.
- **AbstractProduct**: Khai báo dạng interface hoặc abstract class để định nghĩa đối tượng abstract.
- **Product**: Cài đặt của các đối tượng cụ thể, cài đặt các phương thức được quy định tại AbstractProduct.
- **Client**: là đối tượng sử dụng AbstractFactory và các AbstractProduct.

Ví du: Chúng ta có các nhà máy sản xuất Ô tô được đặt tại IndiaCarFactory, USACarFactory và DefaultCarFactory. Và chương trình của chúng ta cần đủ thông minh để xác định được vị trí mà nhà máy được đặt, vì vậy chúng ta sẽ có thể sử dụng nhà máy ô tô phù hợp mà không cần biết việc thực hiện nhà máy ô tô nào sẽ được sử dụng trong nội bộ. Điều này cũng tiết kiệm cho chúng ta từ một người gọi nhầm nhà máy cho một vị trí cụ thể.



`CarType.class`

```
// Java Program to demonstrate the working of Abstract Factory Pattern
enum CarType {
    MICRO,
    MINI,
    LUXURY
}
```

`Car.class`

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

    abstract void construct();

    CarType model = null;
```

```
    Location location = null;

    CarType getModel() {
        return model;
    }

    void setModel(CarType model) {
        this.model = model;
    }

    Location getLocation() {
        return location;
    }

    void setLocation(Location location) {
        this.location = location;
    }

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

`LuxuryCar.class`

```
class LuxuryCar extends Car {
    LuxuryCar(Location location) {
        super(CarType.LUXURY, location);
        construct();
    }
    @Override
    protected void construct() {
        System.out.println("Connecting to luxury car");
    }
}
```

`MicroCar.class`

```
class MicroCar extends Car {
    MicroCar(Location location) {
```

```
        super(CarType.MICRO, location);
        construct();
    }
    @Override
    protected void construct() {
        System.out.println("Connecting to Micro Car ");
    }
}
```

`Minicar.class`

```
class MiniCar extends Car {
    MiniCar(Location location) {
        super(CarType.MINI, location);
        construct();
    }

    @Override
    void construct() {
        System.out.println("Connecting to Mini car");
    }
}
```

`Location.class`

```
enum Location {
    DEFAULT,
    USA,
    INDIA
}
```

`INDIACarFactory.class`

```
class INDIACarFactory {
    static Car buildCar(CarType model) {
        Car car = null;
        switch (model) {
            case MICRO:
                car = new MicroCar(Location.INDIA);
                break;
```

```
            case MINI:
                car = new MiniCar(Location.INDIA);
                break;

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

            default:
                break;

        }
        return car;
    }
}
```

`DefaultCarFactory.class`

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

            case MINI:
                car = new MiniCar(Location.DEFAULT);
                break;

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

            default:
                break;

        }
        return car;
    }
```

```
}
```

`USACarFactory.class`

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

            case MINI:
                car = new MiniCar(Location.USA);
                break;

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

            default:
                break;

        }
        return car;
    }
}
```

`CarFactory.class`

```
class CarFactory {
    private CarFactory() {

    }
    public static Car buildCar(CarType type) {
        Car car = null;
        // We can add any GPS Function here which
        // read location property somewhere from configuration
        // and use location specific car factory
        // Currently I'm just using INDIA as Location
        Location location = Location.INDIA;
```

```
        switch (location) {
            case USA:
                car = USACarFactory.buildCar(type);
                break;

            case INDIA:
                car = INDIACarFactory.buildCar(type);
                break;

            default:
                car = DefaultCarFactory.buildCar(type);

        }

        return car;

    }
}
```

`AbstractDesign.class`

```
class AbstractDesign {
    public static void main(String[] args) {
        System.out.println(CarFactory.buildCar(CarType.MICRO));
        System.out.println(CarFactory.buildCar(CarType.MINI));
        System.out.println(CarFactory.buildCar(CarType.LUXURY));
    }
}
```

Output:

```
Connecting to Micro Car
CarModel - MICRO located in INDIA
Connecting to Mini car
CarModel - MINI located in INDIA
Connecting to luxury car
CarModel - LUXURY located in INDIA
```

Revision #3
Created 2 November 2019 14:46:06 by Laptrinh.vn
Updated 12 April 2020 14:52:27 by Laptrinh.vn