TMPS - Laboratory work 1

Topic: Electric Cars Manufacturer
Author: FAF-191, Boico Alexandr

Objectives:

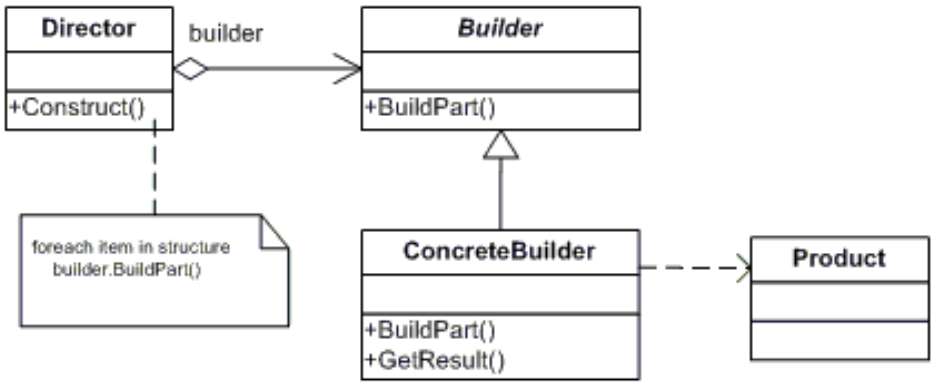
- Get familiar with the Creational DPs;
- Choose a specific domain;
- Implement at least 3 CDPs for the specific domain;

Used Design Patterns:

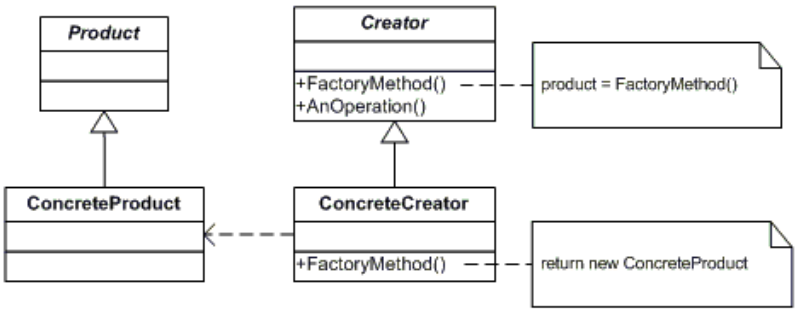
- Builder
- Factory method
- Abstract Factory method

Theory

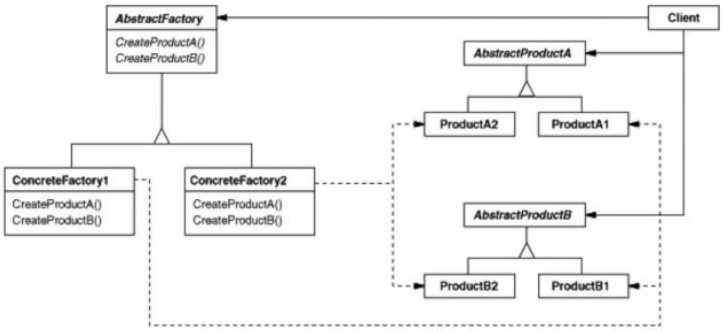
Builder is a creational design pattern that lets you construct complex objects step by step. The pattern allows you to produce different types and representations of an object using the same construction code.



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



Abstract Factory is a creational design pattern that lets you produce families of related objects without specifying their concrete classes.



Implementation

Builder Design Pattern Representation

A car is a complex object that can be constructed in a hundred different ways. Instead of bloating the `Car` class with a huge constructor, we extracted the car assembly code into a separate car builder class. This class has a set of methods for configuring various parts of a car.

```
public class CarBuilder implements Builder {
    private CarType type;
    private int seats;
    private Complectation complectation;
    private Engine engine;
    private Transmission transmission;

    public void setCarType(CarType type) { this.type = type; }

    @Override
    public void setComplectation(Complectation complectation) { this.complectation = complectation; }

    @Override
    public void setSeats(int seats) { this.seats = seats; }

    @Override
    public void setEngine(Engine engine) { this.engine = engine; }

    @Override
    public void setTransmission(Transmission transmission) { this.transmission = transmission; }
```

If the client code needs to assemble a special, fine-tuned model of a car, it can work with the builder directly.

```
public class CarManualConstructor {
    private final CarType carType;
    private final Complectation complectation;
    private final Engine engine;
    private final Transmission transmission;
    private final int seats;

    1 related problem
    public CarManualConstructor(CarType carType, Complectation complectation, Engine engine,
                                Transmission transmission, int seats) {
        this.carType = carType;
        this.complectation = complectation;
        this.engine = engine;
        this.transmission = transmission;
        this.seats = seats;
    }
```

Or, the client can delegate the assembly to the director class, which knows how to use a builder to construct several of the most popular models of cars.

```
public class Director {

    public void constructSedan(Builder builder) {
        builder.setCarType(CarType.SEDAN);
        builder.setComplectation(Complectation.COMFORT);
        builder.setEngine(new Engine( volume: 2.0,  power: 160, Fuel.diesel));
        builder.setTransmission(Transmission.ROBOT);
        builder.setSeats(5);
    }

    public void constructHatchback(Builder builder) {
        builder.setCarType(CarType.HATCHBACK);
        builder.setComplectation(Complectation.STANDARD);
        builder.setEngine(new Engine( volume: 1.2,  power: 90, Fuel.gasoline));
        builder.setTransmission(Transmission.CVT);
        builder.setSeats(5);
    }

    public void constructCoupe(Builder builder) {
        builder.setCarType(CarType.CUOPE);
        builder.setComplectation(Complectation.LUXURY);
        builder.setEngine(new Engine( volume: 5.2,  power: 450, Fuel.gasoline));
        builder.setTransmission(Transmission.MANUAL);
        builder.setSeats(4);
    }
}
```

Client:

```
public class Main {
    public static void main(String[] args) {
        Director director = new Director();

        CarBuilder builder = new CarBuilder();
        director.constructSedan(builder);

        Cars car = builder.getResult();
        System.out.println("Car built:\n" + car.getCarType());
        car.print();

        CarConstructor manualBuilder = new CarConstructor();

        director.constructCoupe(manualBuilder);
        CarManualConstructor carManual = manualBuilder.getResult();
        System.out.println("\nCar manual built:\n" + carManual.print());
    }
}
```

RESULTS:

```
Car built:
SEDAN

Car manual built:
Type of car: CUOPE
Complectation: LUXURY
Engine: volume - 5.2; power - 450; fuel - gasoline
Transmission: MANUAL
Count of seats: 4
```

Factory method

in base we have 2 interfaces “Car” and “CarFactory”

```java
package creational.FactoryDP;

public interface Car {

    void produceCar();
}
```

```java
public interface CarFactory {
    Car createCarFactory();
}
```

and a car creator that initialises a car factory depending on car type.

```java
public static CarFactory createCarByType(String carType){
    if (carType.equalsIgnoreCase( anotherString: "sedan"))
        return new SedanFactory();
        else if (carType.equalsIgnoreCase( anotherString: "hatchback"))
            return  new HatchbackFactory();
        else  throw new RuntimeException(carType + "is unknown");

}
```

Client:

```java
public static void main(String[] args) {
CarFactory carFactory = createCarByType( carType: "sedan");
Car createCar = carFactory.createCarFactory();
createCar.produceCar();
}
```

Results:

```
Sedan factory produced one sedan
```

Abstract Factory:

First, we have to write all separate car factories for different complectation sets. To support, location specific features, begin with modifying our “Car” class with another attribute “Complectation”

```
public abstract class Car {

        public Car(CarType type, Complectation complectation){
            this.type = type;
            this.complectation = complectation;
        }


        protected CarType type;
        protected Complectation complectation;
        //getters and setters

        public String construct() {
            return "Model - "+ type + "; Complectation - " + complectation;
        }
}
```

For each Complectation we have several factory:

```
public class StandardCarFactory extends Car{

    public StandardCarFactory(CarType type){
        super(type, Complectation.STANDARD);
    }

    @Override
    public String construct() {
        return "Car produced: \n Model - "+ type + "; Complectation - " + complectation;
    }
}
```

And “main factory” that gives orders to particular

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

    public static void buildCar(CarType type, Complectation complectation)
    {
        Car car;

        switch(complectation)
        {
            case STANDARD:
                car = new StandardCarFactory(type);
                System.out.println(car.construct());
                break;
            case COMFORT:
                car = new ComfortCarFactory(type);
                System.out.println(car.construct());
                break;
            case LUXURY:
                car = new LuxuryCarFactory(type);
                System.out.println(car.construct());
        }
    }
}
```

Client:

```
public class Main {

    public static void main(String[] args) {
        CarFactory.buildCar(CarType.HATCHBACK, Complectation.COMFORT);
        CarFactory.buildCar(CarType.SEDAN, Complectation.STANDARD);
        CarFactory.buildCar(CarType.CUOPE, Complectation.LUXURY);
    }
}
```

Results:

```
Car produced:
 Model - HATCHBACK; Complectation - COMFORT
Car produced:
 Model - SEDAN; Complectation - STANDARD
Car produced:
 Model - CUOPE; Complectation - LUXURY
```

Conclusion:

- Many designs start by using **Factory Method** (less complicated and more customizable via subclasses) and evolve toward **Abstract Factory**, **Prototype**, or **Builder** (more flexible, but more complicated).
- **Builder** focuses on constructing complex objects step by step. **Abstract Factory** specializes in creating families of related objects. *Abstract Factory* returns the product immediately, whereas *Builder* lets you run some additional construction steps before fetching the product.
- **Abstract Factory** classes are often based on a set of **Factory Methods**