Строитель на Java

Строитель — это порождающий паттерн проектирования, который позволяет создавать объекты пошагово.

В отличие от других порождающих паттернов, Строитель позволяет производить различные продукты, используя один и тот же процесс строительства.

Подробней о паттерне Строитель →

Навигация

- Director
- Demo
- OutputDemo

Сложность: ★★☆

Популярность: ★★★

Применимость: Паттерн можно часто встретить в Java-коде, особенно там, где требуется пошаговое создание продуктов или конфигурация сложных объектов.

Паттерн широко используется в стандартных библиотеках Java:

- **`java.lang.StringBuilder#append()`** (`unsynchronized`)
- **`java.lang.StringBuffer#append()`** (`synchronized`)
- **`java.nio.ByteBuffer#put()`** (также в **`CharBuffer`**, **`ShortBuffer`**, **`IntBuffer`**, **`LongBuffer`**, **`FloatBuffer`** и **`DoubleBuffer`**)
- **`javax.swing.GroupLayout.Group#addComponent()`**
- Все реализации **`java.lang.Appendable`**

Признаки применения паттерна: Строителя можно узнать в классе, который имеет один создающий метод и несколько методов настройки создаваемого продукта. Обычно, методы настройки вызывают для удобства цепочкой (например, `someBuilder.setValueA(1).setValueB(2).create()`).

Пошаговое производство автомобилей

В этом примере Строитель позволяет пошагово конструировать различные конфигурации автомобилей.

Кроме того, здесь показано как с помощью Строителя может создавать другой продукт на основе тех же шагов строительства. Для этого мы имеем два конкретных строителя — `CarBuilder` и `CarManualBuilder`.

Порядок строительства продуктов заключён в Директоре. Он знает какие шаги строителей нужно вызвать, чтобы получить ту или иную конфигурацию продукта. Он работает со строителями только через общий интерфейс, что позволяет взаимозаменять объекты строителей для получения разных продуктов.

builders

builders/Builder.java: Общий интерфейс строителей

```
package refactoring_guru.builder.example.builders;

import refactoring_guru.builder.example.cars.CarType;
import refactoring_guru.builder.example.components.Engine;
import refactoring_guru.builder.example.components.GPSNavigator;
import refactoring_guru.builder.example.components.Transmission;
import refactoring_guru.builder.example.components.TripComputer;

/**
 * Интерфейс Строителя объявляет все возможные этапы и шаги конфигурации
 * продукта.
 */
public interface Builder {
    void setCarType(CarType type);
    void setSeats(int seats);
    void setEngine(Engine engine);
    void setTransmission(Transmission transmission);
    void setTripComputer(TripComputer tripComputer);
    void setGPSNavigator(GPSNavigator gpsNavigator);
}
```

builders/CarBuilder.java: Строитель автомобилей

```
package refactoring_guru.builder.example.builders;

import refactoring_guru.builder.example.cars.Car;
import refactoring_guru.builder.example.cars.CarType;
import refactoring_guru.builder.example.components.Engine;
import refactoring_guru.builder.example.components.GPSNavigator;
import refactoring_guru.builder.example.components.Transmission;
import refactoring_guru.builder.example.components.TripComputer;

/**
 * Конкретные строители реализуют шаги, объявленные в общем интерфейсе.
 */
public class CarBuilder implements Builder {
    private CarType type;
    private int seats;
    private Engine engine;
    private Transmission transmission;
    private TripComputer tripComputer;
    private GPSNavigator gpsNavigator;
```

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

    @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;
    }

    @Override
    public void setTripComputer(TripComputer tripComputer) {
        this.tripComputer = tripComputer;
    }

    @Override
    public void setGPSNavigator(GPSNavigator gpsNavigator) {
        this.gpsNavigator = gpsNavigator;
    }

    public Car getResult() {
        return new Car(type, seats, engine, transmission, tripComputer, gpsNavigator);
    }
}
```

builders/CarManualBuilder.java: Строитель руководств пользователя

```
package refactoring_guru.builder.example.builders;

import refactoring_guru.builder.example.cars.Manual;
import refactoring_guru.builder.example.cars.CarType;
import refactoring_guru.builder.example.components.Engine;
import refactoring_guru.builder.example.components.GPSNavigator;
import refactoring_guru.builder.example.components.Transmission;
import refactoring_guru.builder.example.components.TripComputer;

/**
 * В отличие от других создающих паттернов, Строители могут создавать совершенно
 * разные продукты, не имеющие общего интерфейса.
```

```
 *
 * В данном случае мы производим руководство пользователя автомобиля с помощью
 * тех же шагов, что и сами автомобили. Это устройство позволит создавать
 * руководства под конкретные модели автомобилей, содержащие те или иные фичи.
 */
public class CarManualBuilder implements Builder{
    private CarType type;
    private int seats;
    private Engine engine;
    private Transmission transmission;
    private TripComputer tripComputer;
    private GPSNavigator gpsNavigator;

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

    @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;
    }

    @Override
    public void setTripComputer(TripComputer tripComputer) {
        this.tripComputer = tripComputer;
    }

    @Override
    public void setGPSNavigator(GPSNavigator gpsNavigator) {
        this.gpsNavigator = gpsNavigator;
    }

    public Manual getResult() {
        return new Manual(type, seats, engine, transmission, tripComputer, gpsNavigator);
    }
}
```

cars

cars/Car.java: Продукт-автомобиль

```
package refactoring_guru.builder.example.cars;

import refactoring_guru.builder.example.components.Engine;
import refactoring_guru.builder.example.components.GPSNavigator;
import refactoring_guru.builder.example.components.Transmission;
import refactoring_guru.builder.example.components.TripComputer;

/**
 * Автомобиль — это класс продукта.
 */
public class Car {
    private final CarType carType;
    private final int seats;
    private final Engine engine;
    private final Transmission transmission;
    private final TripComputer tripComputer;
    private final GPSNavigator gpsNavigator;
    private double fuel = 0;

    public Car(CarType carType, int seats, Engine engine, Transmission transmission,
               TripComputer tripComputer, GPSNavigator gpsNavigator) {
        this.carType = carType;
        this.seats = seats;
        this.engine = engine;
        this.transmission = transmission;
        this.tripComputer = tripComputer;
        if (this.tripComputer != null) {
            this.tripComputer.setCar(this);
        }
        this.gpsNavigator = gpsNavigator;
    }

    public CarType getCarType() {
        return carType;
    }

    public double getFuel() {
        return fuel;
    }

    public void setFuel(double fuel) {
        this.fuel = fuel;
    }

    public int getSeats() {
        return seats;
    }

    public Engine getEngine() {
```

```
        return engine;
    }

    public Transmission getTransmission() {
        return transmission;
    }

    public TripComputer getTripComputer() {
        return tripComputer;
    }

    public GPSNavigator getGpsNavigator() {
        return gpsNavigator;
    }
}
```

cars/Manual.java: Продукт-руководство

```
package refactoring_guru.builder.example.cars;

import refactoring_guru.builder.example.components.Engine;
import refactoring_guru.builder.example.components.GPSNavigator;
import refactoring_guru.builder.example.components.Transmission;
import refactoring_guru.builder.example.components.TripComputer;

/**
 * Руководство автомобиля — это второй продукт. Заметьте, что руководство и сам
 * автомобиль не имеют общего родительского класса. По сути, они независимы.
 */
public class Manual {
    private final CarType carType;
    private final int seats;
    private final Engine engine;
    private final Transmission transmission;
    private final TripComputer tripComputer;
    private final GPSNavigator gpsNavigator;

    public Manual(CarType carType, int seats, Engine engine, Transmission transmission,
                  TripComputer tripComputer, GPSNavigator gpsNavigator) {
        this.carType = carType;
        this.seats = seats;
        this.engine = engine;
        this.transmission = transmission;
        this.tripComputer = tripComputer;
        this.gpsNavigator = gpsNavigator;
    }

    public String print() {
```

```
        String info = "";
        info += "Type of car: " + carType + "\n";
        info += "Count of seats: " + seats + "\n";
        info += "Engine: volume - " + engine.getVolume() + "; mileage - " + engine.getMil
        info += "Transmission: " + transmission + "\n";
        if (this.tripComputer != null) {
            info += "Trip Computer: Functional" + "\n";
        } else {
            info += "Trip Computer: N/A" + "\n";
        }
        if (this.gpsNavigator != null) {
            info += "GPS Navigator: Functional" + "\n";
        } else {
            info += "GPS Navigator: N/A" + "\n";
        }
        return info;
    }
}
```

cars/CarType.java

```
package refactoring_guru.builder.example.cars;

public enum CarType {
    CITY_CAR, SPORTS_CAR, SUV
}
```

components

components/Engine.java: Разные фичи продуктов

```
package refactoring_guru.builder.example.components;

/**
 * Одна из фишек автомобиля.
 */
public class Engine {
    private final double volume;
    private double mileage;
    private boolean started;

    public Engine(double volume, double mileage) {
        this.volume = volume;
        this.mileage = mileage;
```

```
    }

    public void on() {
        started = true;
    }

    public void off() {
        started = false;
    }

    public boolean isStarted() {
        return started;
    }

    public void go(double mileage) {
        if (started) {
            this.mileage += mileage;
        } else {
            System.err.println("Cannot go(), you must start engine first!");
        }
    }

    public double getVolume() {
        return volume;
    }

    public double getMileage() {
        return mileage;
    }
}
```

components/GPSNavigator.java: Разные фичи продуктов

```
package refactoring_guru.builder.example.components;

/**
 * Одна из фишек автомобиля.
 */
public class GPSNavigator {
    private String route;

    public GPSNavigator() {
        this.route = "221b, Baker Street, London  to Scotland Yard, 8-10 Broadway, Londor
    }

    public GPSNavigator(String manualRoute) {
        this.route = manualRoute;
    }
```

```
    public String getRoute() {
        return route;
    }
}
```

components/Transmission.java: Разные фичи продуктов

```
package refactoring_guru.builder.example.components;

/**
 * Одна из фишек автомобиля.
 */
public enum Transmission {
    SINGLE_SPEED, MANUAL, AUTOMATIC, SEMI_AUTOMATIC
}
```

components/TripComputer.java: Разные фичи продуктов

```
package refactoring_guru.builder.example.components;

import refactoring_guru.builder.example.cars.Car;

/**
 * Одна из фишек автомобиля.
 */
public class TripComputer {

    private Car car;

    public void setCar(Car car) {
        this.car = car;
    }

    public void showFuelLevel() {
        System.out.println("Fuel level: " + car.getFuel());
    }

    public void showStatus() {
        if (this.car.getEngine().isStarted()) {
            System.out.println("Car is started");
        } else {
            System.out.println("Car isn't started");
        }
```

```
        }
    }
```

director

director/Director.java: Директор управляет строителями

```
package refactoring_guru.builder.example.director;

import refactoring_guru.builder.example.builders.Builder;
import refactoring_guru.builder.example.cars.CarType;
import refactoring_guru.builder.example.components.Engine;
import refactoring_guru.builder.example.components.GPSNavigator;
import refactoring_guru.builder.example.components.Transmission;
import refactoring_guru.builder.example.components.TripComputer;

/**
 * Директор знает в какой последовательности заставлять работать строителя. Он
 * работает с ним через общий интерфейс Строителя. Из-за этого, он может не
 * знать какой конкретно продукт сейчас строится.
 */
public class Director {

    public void constructSportsCar(Builder builder) {
        builder.setCarType(CarType.SPORTS_CAR);
        builder.setSeats(2);
        builder.setEngine(new Engine(3.0, 0));
        builder.setTransmission(Transmission.SEMI_AUTOMATIC);
        builder.setTripComputer(new TripComputer());
        builder.setGPSNavigator(new GPSNavigator());
    }

    public void constructCityCar(Builder builder) {
        builder.setCarType(CarType.CITY_CAR);
        builder.setSeats(2);
        builder.setEngine(new Engine(1.2, 0));
        builder.setTransmission(Transmission.AUTOMATIC);
        builder.setTripComputer(new TripComputer());
        builder.setGPSNavigator(new GPSNavigator());
    }

    public void constructSUV(Builder builder) {
        builder.setCarType(CarType.SUV);
        builder.setSeats(4);
        builder.setEngine(new Engine(2.5, 0));
        builder.setTransmission(Transmission.MANUAL);
        builder.setGPSNavigator(new GPSNavigator());
```

```
    }
}
```

Demo.java: Клиентский код

```
package refactoring_guru.builder.example;

import refactoring_guru.builder.example.builders.CarBuilder;
import refactoring_guru.builder.example.builders.CarManualBuilder;
import refactoring_guru.builder.example.cars.Car;
import refactoring_guru.builder.example.cars.Manual;
import refactoring_guru.builder.example.director.Director;

/**
 * Демо-класс. Здесь всё сводится воедино.
 */
public class Demo {

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

        // Директор получает объект конкретного строителя от клиента
        // (приложения). Приложение само знает какой строитель использовать,
        // чтобы получить нужный продукт.
        CarBuilder builder = new CarBuilder();
        director.constructSportsCar(builder);

        // Готовый продукт возвращает строитель, так как Директор чаще всего не
        // знает и не зависит от конкретных классов строителей и продуктов.
        Car car = builder.getResult();
        System.out.println("Car built:\n" + car.getCarType());


        CarManualBuilder manualBuilder = new CarManualBuilder();

        // Директор может знать больше одного рецепта строительства.
        director.constructSportsCar(manualBuilder);
        Manual carManual = manualBuilder.getResult();
        System.out.println("\nCar manual built:\n" + carManual.print());
    }

}
```

OutputDemo.txt: Результаты выполнения

```
Car built:
SPORTS_CAR

Car manual built:
Type of car: SPORTS_CAR
Count of seats: 2
Engine: volume - 3.0; mileage - 0.0
Transmission: SEMI_AUTOMATIC
Trip Computer: Functional
GPS Navigator: Functional
```

ЧИТАЕМ ДАЛЬШЕ

Фабричный метод на Java →

ВЕРНУТЬСЯ НАЗАД

← Абстрактная фабрика на Java

Главная
Рефакторинг
Паттерны
Премиум контент
Форум
Связаться

Хмельницкое шоссе 19 / 27, Каменец-Подольский, Украина, 32305
Email: support@refactoring.guru

Условия использования
Политика конфиденциальности
Использование контента