

# Порождающие паттерны

## *Фабричный метод, строитель, прототип*

Подготовили:

Шилов Павел

Назарова Анастасия

# Что такое порождающие паттерны?

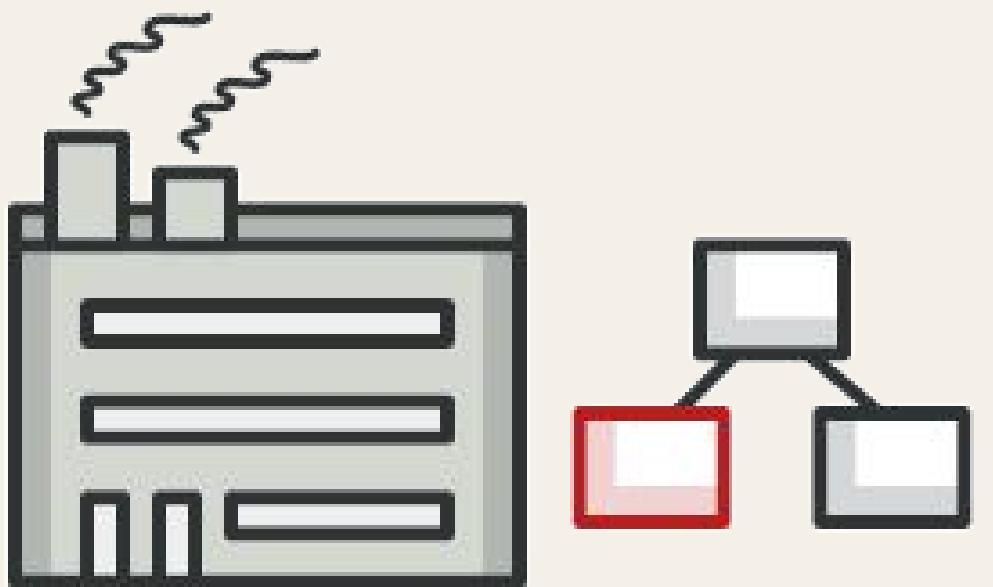
- Инкапсуляция процесса создания объектов
- Уменьшение связанности кода
- Повышение гибкости и переиспользуемости
- Упрощение добавления новых типов объектов

# Зачем нужны порождающие паттерны?

- Жёсткая привязка к конкретным классам
- Сложность создания объектов с множеством параметров
- Необходимость копирования существующих объектов
- Трудности в расширении и поддержке кода

## Фабричный метод

Определяет общий интерфейс для создания объектов, позволяя подклассам изменять тип создаваемых объектов



# Суть паттерна



# Проблема:

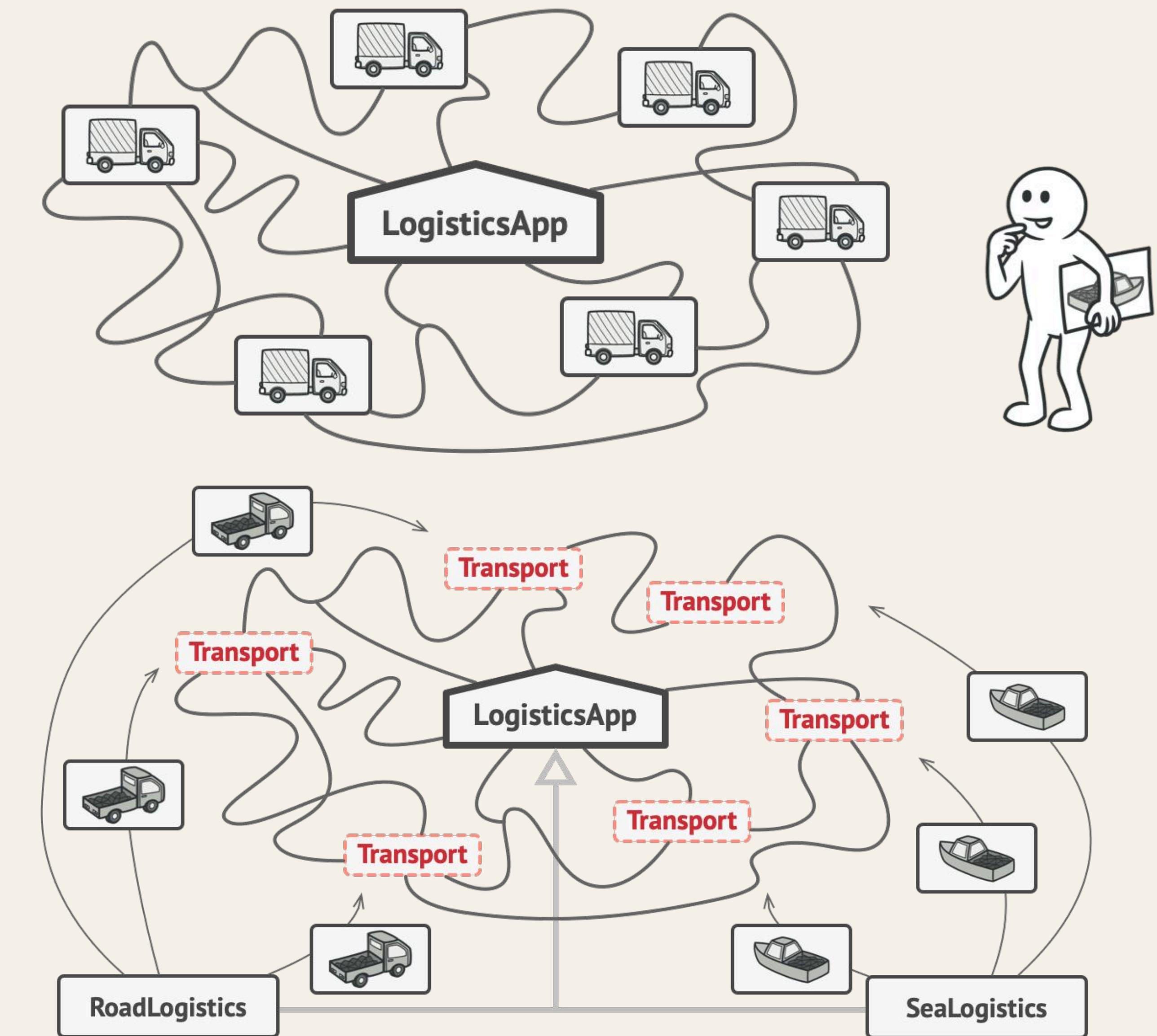
- Изначально только грузовики
- Потом добавились корабли
- Затем самолёты

```

1 if transport_type == "truck":
2     vehicle = Truck()
3 elif transport_type == "ship":
4     vehicle = Ship()
5 # Добавление нового типа
# требует изменения кода
    
```

# Решение:

- Новые типы транспорта добавляются без изменения существующего кода



# Ключевые компоненты

Продукт — общий интерфейс для всех

создаваемых объектов

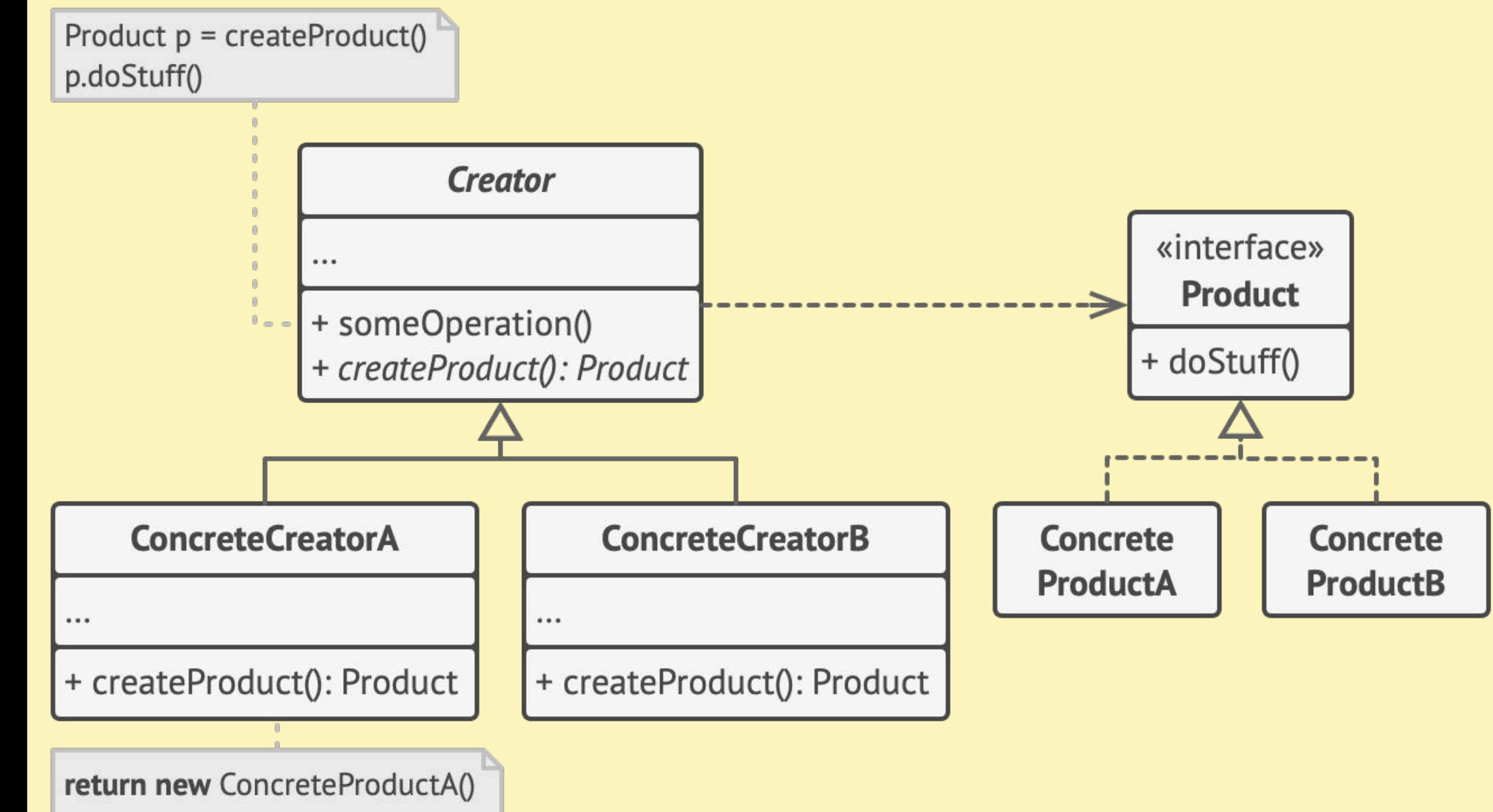
Конкретные продукты — реализации

интерфейса

Создатель — объявляет фабричный метод

Конкретные создатели — переопределяют

фабричный метод



```

1  public class FactoryMethodExample {
2      interface Transport {
3          void deliver(String cargo);}
4      static class Truck implements Transport {
5          @Override
6          public void deliver(String cargo) {
7              System.out.println("Грузовик доставляет '" + cargo + "' по дороге");}
8      static class Ship implements Transport {
9          @Override
10         public void deliver(String cargo) {
11             System.out.println("Корабль доставляет '" + cargo + "' по морю");}
12     static abstract class Logistics {
13         public abstract Transport createTransport();
14         public void planDelivery(String cargo) {
15             Transport transport = createTransport(); // Вызов фабричного метода
16             transport.deliver(cargo);}
17     static class RoadLogistics extends Logistics {
18         @Override
19         public Transport createTransport() {
20             return new Truck();}
21     static class SeaLogistics extends Logistics {
22         @Override
23         public Transport createTransport() {
24             return new Ship();}
25     public static void main(String[] args) {
26         Logistics roadCompany = new RoadLogistics();
27         Logistics seaCompany = new SeaLogistics();
28         System.out.println("Дорожная логистика:");
29         roadCompany.planDelivery("компьютеры");
30         System.out.println("\nМорская логистика:");
31         seaCompany.planDelivery("нефть");
32
33
34
35

```

## Применимость:

1. Когда заранее неизвестны типы объектов
2. Когда нужно дать возможность расширять части системы
3. Когда нужно экономить ресурсы, переиспользуя объекты
4. Когда нужно отделить создание объектов от их использования

## Преимущества:

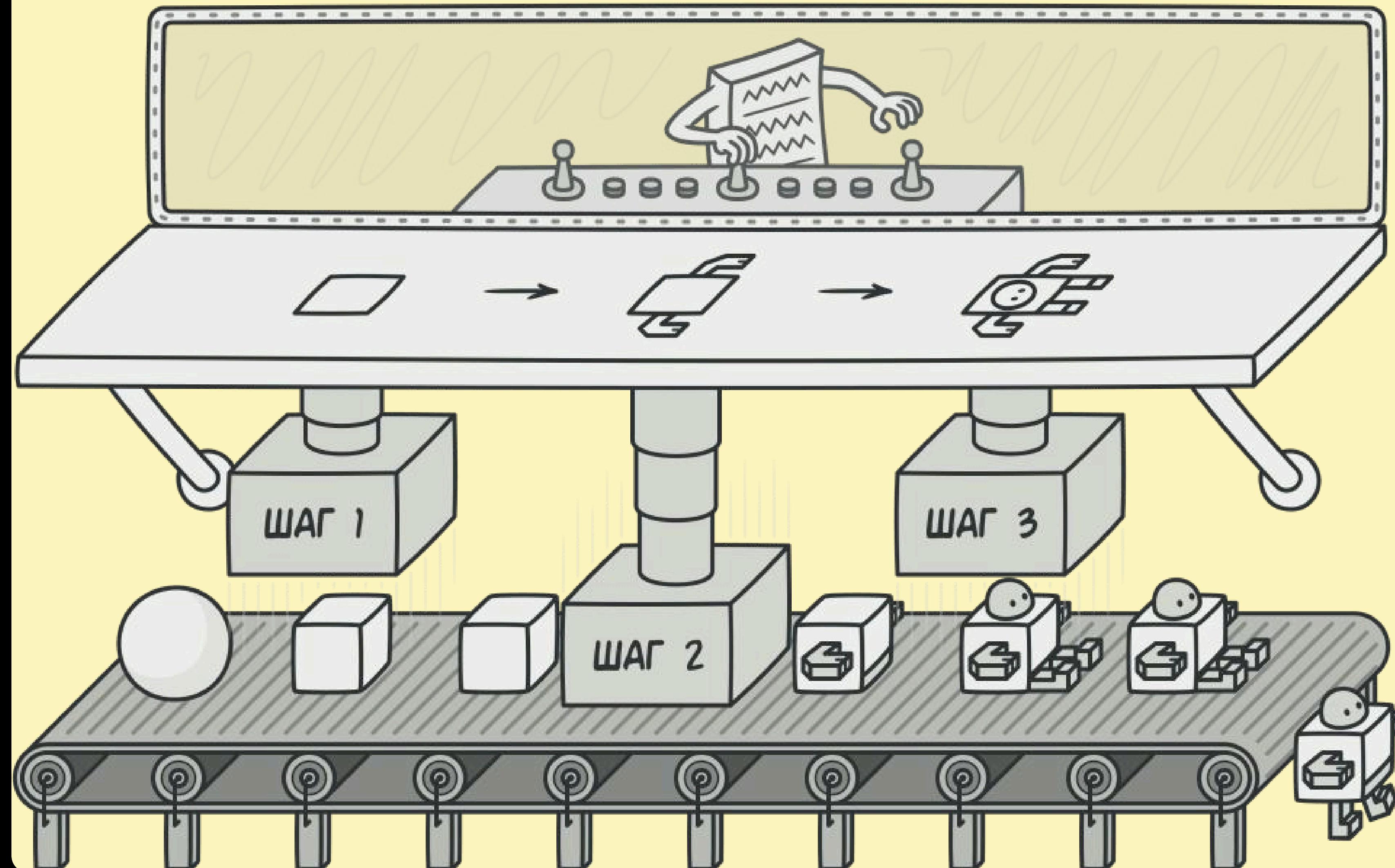
- Избавляет от привязки к конкретным классам
- Упрощает добавление новых продуктов
- Соответствует принципу открытости/закрытости

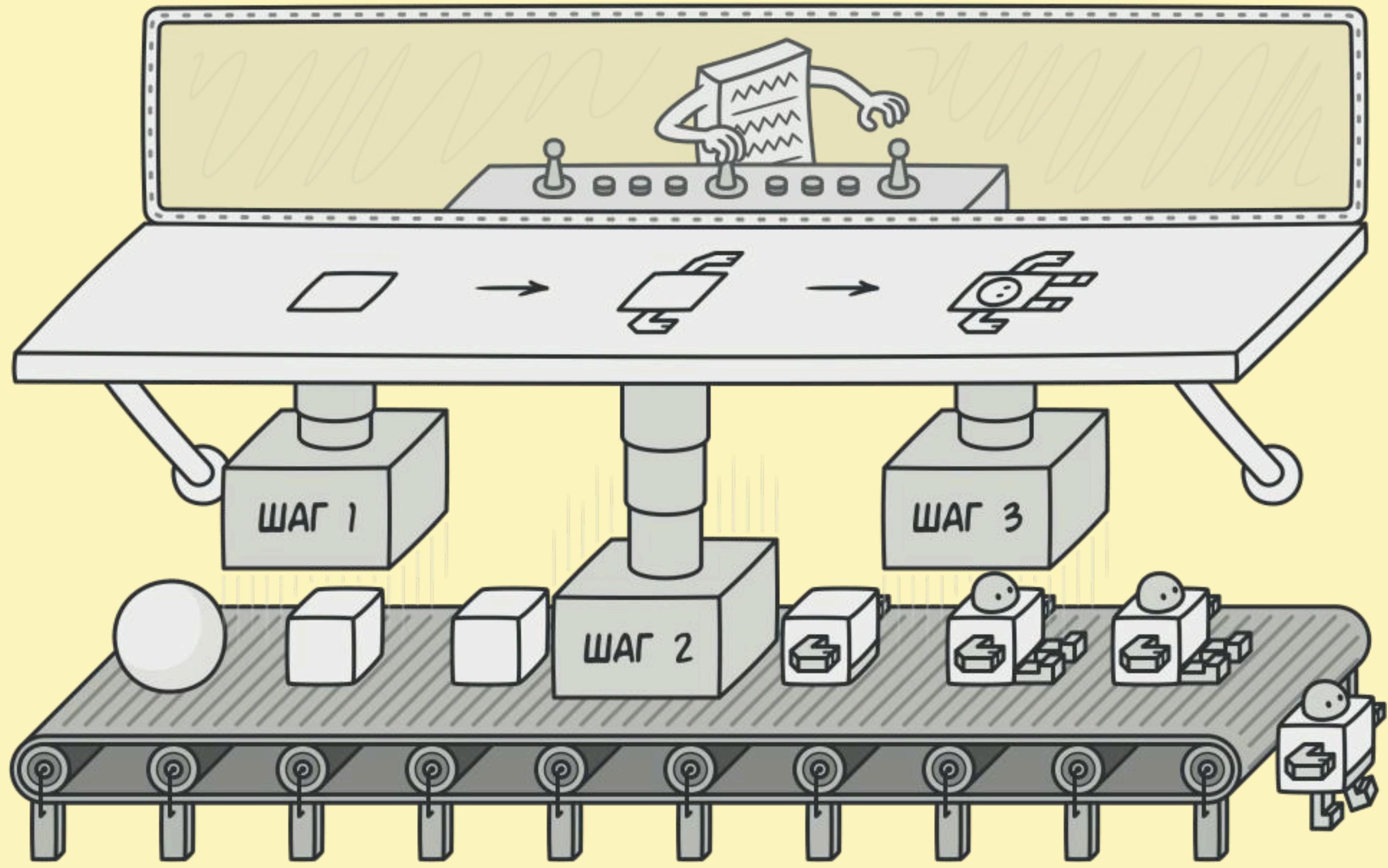
# Строитель

Позволяет создавать сложные объекты пошагово, используя один и тот же процесс строительства для получения разных представлений



# Суть паттерна





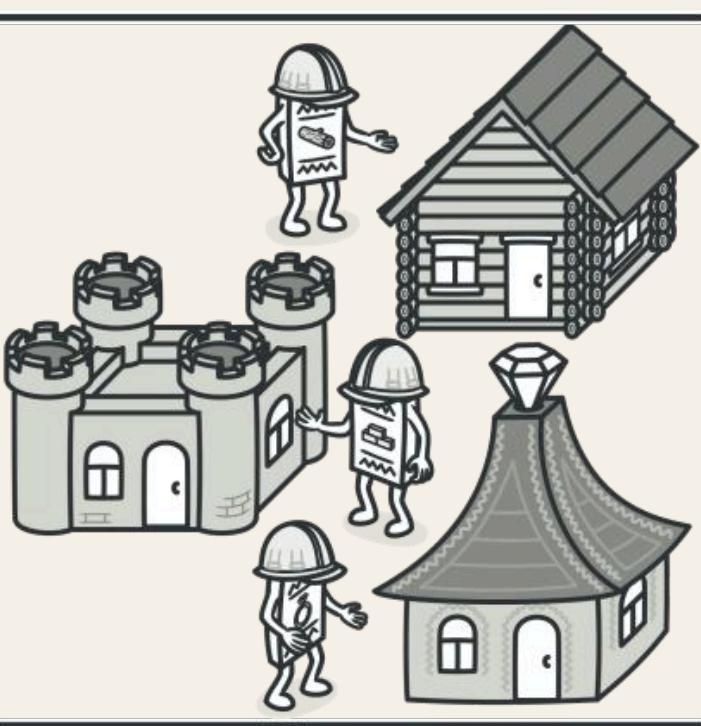
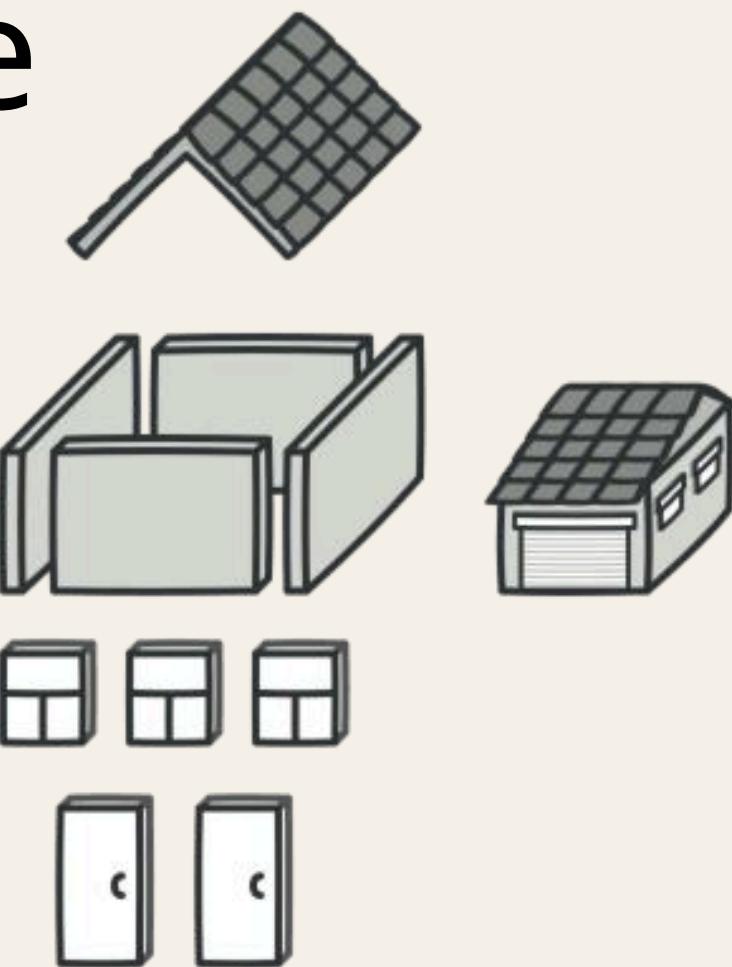
## Проблема



## Решение

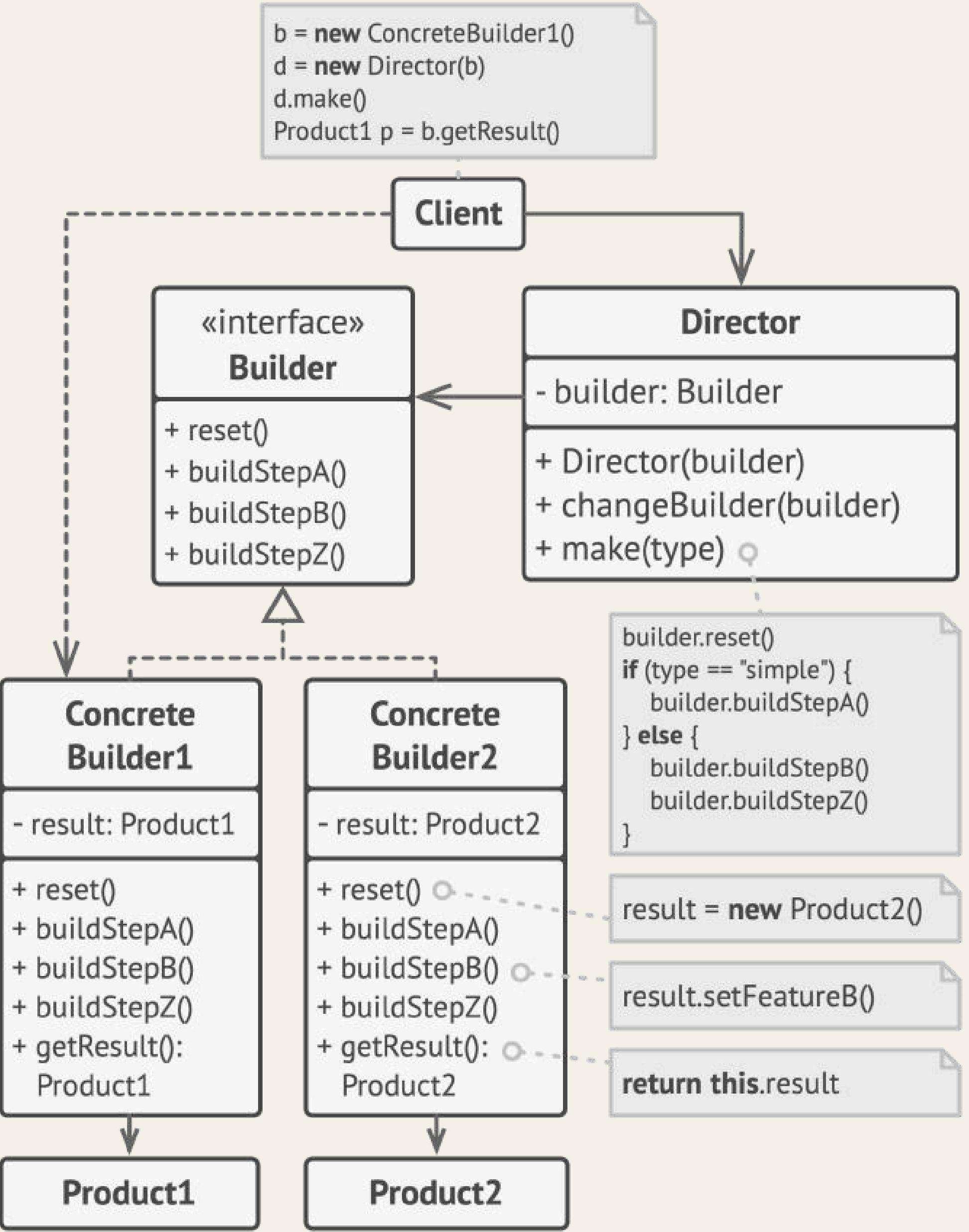
### HouseBuilder

```
+ buildWalls()
+ buildDoors()
+ buildWindows()
+ buildRoof()
+ buildGarage()
+ getResult(): House
```



## Директор





```

1 PUBLIC CLASS BUILDERSHORT {
2     STATIC CLASS HOUSE {
3         PRIVATE STRING WALLS, DOORS, WINDOWS, ROOF;
4
5         VOID SETWALLS(STRING WALLS) { THIS.WALLS = WALLS; }
6         VOID SETDOORS(STRING DOORS) { THIS.DOORS = DOORS; }
7         VOID SETWINDOWS(STRING WINDOWS) { THIS.WINDOWS = WINDOWS; }
8         VOID SETROOF(STRING ROOF) { THIS.ROOF = ROOF; }
9
10    @OVERRIDE
11    PUBLIC STRING TOSTRING() {
12        RETURN "ДОМ: " + WALLS + " СТЕНЫ, " + DOORS + " ДВЕРИ";
13    }
14
15    ABSTRACT STATIC CLASS HOUSEBUILDER {
16        PROTECTED HOUSE HOUSE = NEW HOUSE();
17        ABSTRACT VOID BUILDWALLS();
18        ABSTRACT VOID BUILDDOORS();
19        ABSTRACT VOID BUILDWINDOWS();
20        ABSTRACT VOID BUILDROOF();
21        HOUSE GETHOUSE() { RETURN HOUSE; }
22
23    STATIC CLASS WOODENHOUSEBUILDER EXTENDS HOUSEBUILDER {
24        VOID BUILDWALLS() { HOUSE.SETWALLS("ДЕРЕВЯННЫЕ"); }
25        VOID BUILDDOORS() { HOUSE.SETDOORS("ДЕРЕВЯННЫЕ"); }
26        VOID BUILDWINDOWS() { HOUSE.SETWINDOWS("ДЕРЕВЯННЫЕ"); }
27        VOID BUILDROOF() { HOUSE.SETROOF("ДЕРЕВЯННАЯ"); }
28    }
29
30    STATIC CLASS DIRECTOR {
31        HOUSE BUILDSHOT(HOUSEBUILDER BUILDER) {
32            BUILDER.BUILDWALLS();
33            BUILDER.BUILDDOORS();
34            BUILDER.BUILDWINDOWS();
35            BUILDER.BUILDROOF();
36            RETURN BUILDER.GETHOUSE();
37        }
38        PUBLIC STATIC VOID MAIN(STRING[] ARGS) {
39            DIRECTOR DIRECTOR = NEW DIRECTOR();
40            HOUSE HOUSE = DIRECTOR.BUILDSHOT(NEW WOODENHOUSEBUILDER());
41            SYSTEM.OUT.PRINTLN(HOUSE);
42        }
43    }
44
45}
  
```

# Применимость

1. Когда объект имеет много опциональных параметров
2. Когда нужны разные представления одного объекта
3. Когда нужно создавать сложные составные объекты
4. Когда процесс создания должен быть независим от частей объекта

# Преимущества

- Позволяет создавать объекты пошагово
  - Изолирует сложный код сборки
- Позволяет использовать один код для разных продуктов

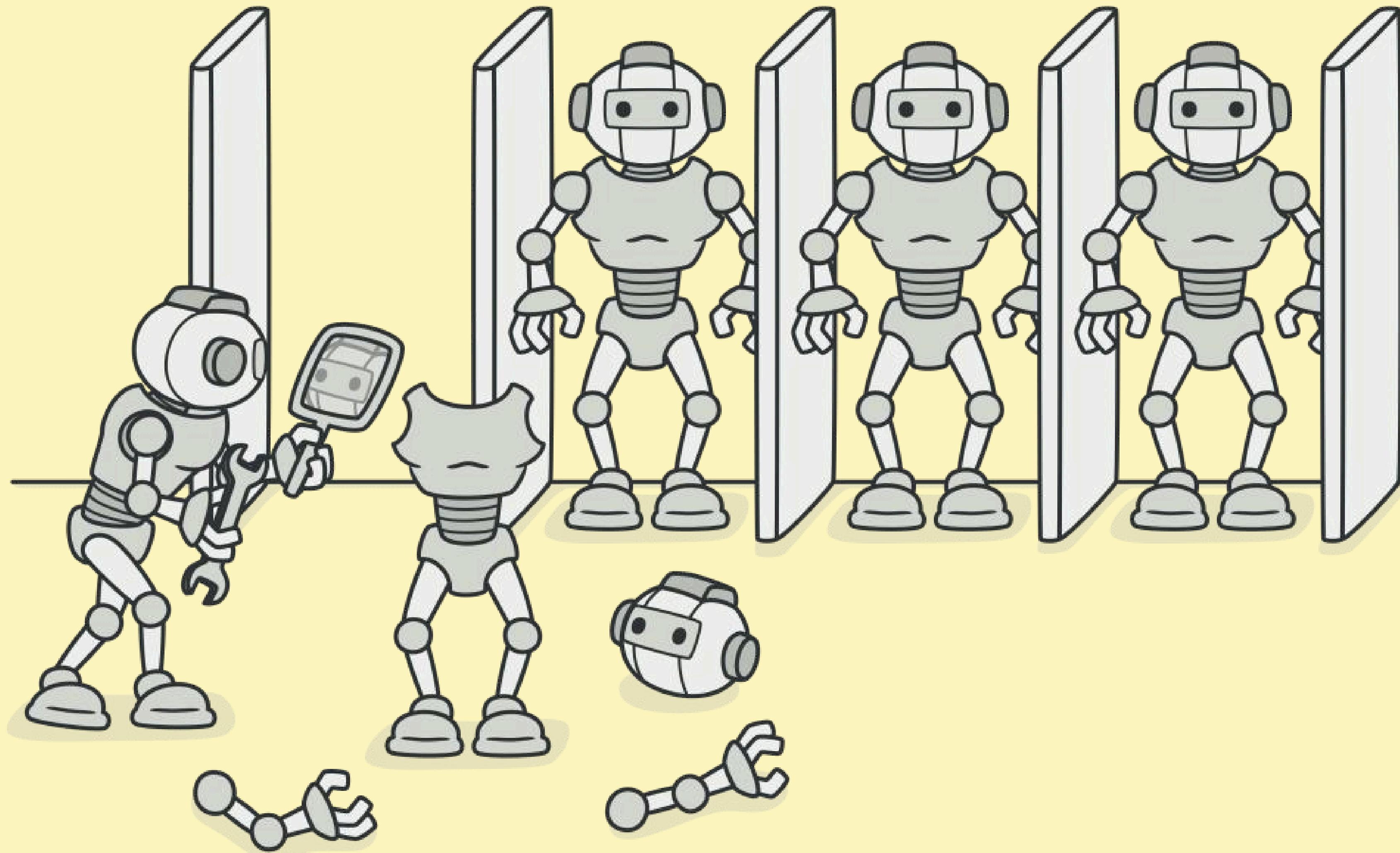
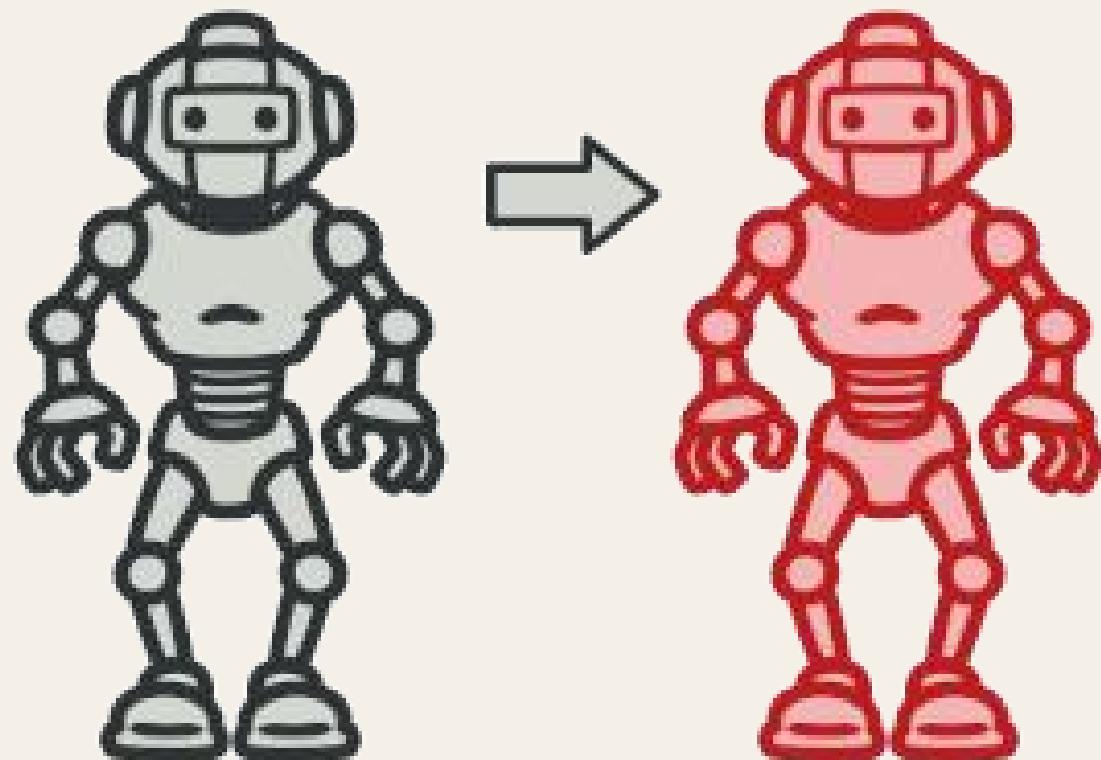
# Недостатки

- Усложняет код из-за дополнительных классов
- Клиент может быть привязан к конкретным строителям

# Суть паттерна

## Прототип

Позволяет копировать существующие объекты, не вдаваясь в подробности их реализации



# Проблема:

- Изначально только грузовики
- Потом добавились корабли
- Затем самолёты

```

1 if transport_type == "truck":
2     vehicle = Truck()
3 elif transport_type == "ship":
4     vehicle = Ship()
5 # Добавление нового типа
# требует изменения кода
    
```

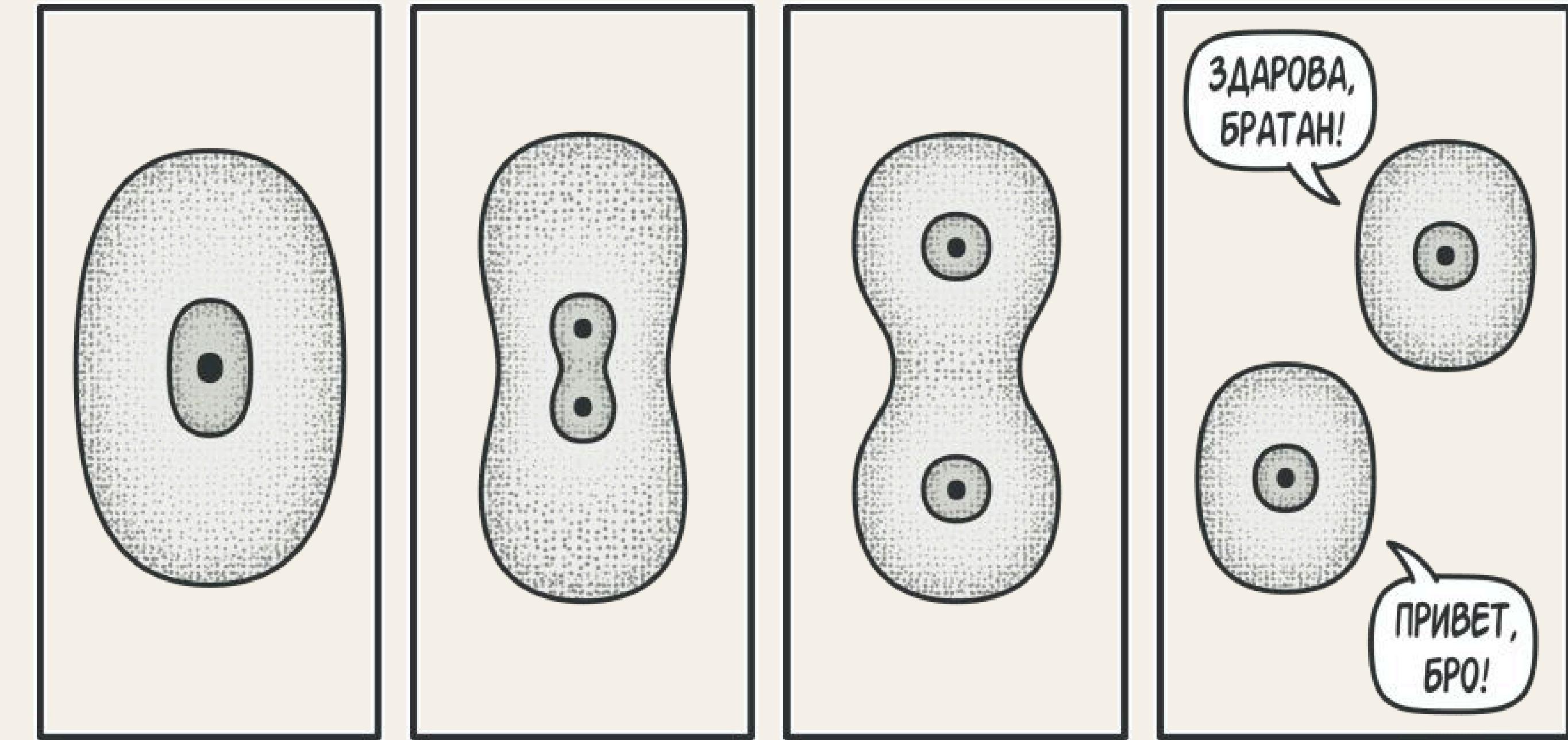
# Решение:

- Новые типы транспорта добавляются без изменения существующего кода



# Ключевые компоненты

1. Прототип — интерфейс с методом `clone()`
2. Конкретный прототип — реализует клонирование
3. Клиент — создаёт копию через вызов `clone()`



```
1      clone()  
2      |      |  
3  clone() field1 field2
```

```

1 // Прототип: клонирование вместо создания с нуля
2 public class PrototypeShort {
3     interface Shape extends Cloneable {
4         Shape clone() throws CloneNotSupportedException;
5         void draw();
6     }
7     static class Circle implements Shape {
8         private int x, y, radius;
9         public Circle(int x, int y, int r) {
10            this.x = x; this.y = y; this.radius = r;
11            System.out.println("Создаём круг (медленно)");
12        }
13        private Circle(Circle source) {
14            this.x = source.x;
15            this.y = source.y;
16            this.radius = source.radius;
17            System.out.println("Клонируем круг (быстро)");
18        }
19
20        public Shape clone() {
21            return new Circle(this);
22        }
23
24        public void draw() {
25            System.out.println("Круг в (" + x + "," + y + "), r=" + radius);
26        }
27
28        void move(int x, int y) { this.x = x; this.y = y; }
29    }
30
31    public static void main(String[] args) throws CloneNotSupportedException {
32        Circle original = new Circle(10, 20, 15);
33        Circle clone = (Circle) original.clone();
34
35        clone.move(100, 100); // Меняем только клон
36
37        System.out.print("Оригинал: "); original.draw();
38        System.out.print("Клон:      "); clone.draw();
39    }
40 }
41

```

## Применимость:

1. Когда создание объекта дороже копирования
2. Когда код не должен зависеть от классов копируемых объектов
3. Когда есть множество подклассов с разными начальными значениями
4. Когда нужно избежать построения иерархий фабрик

## Преимущества:

- Позволяет клонировать объекты без привязки к их классам
- Уменьшает повторяющийся код инициализации
- Ускоряет создание объектов

Критерий	Прототип	Фабричный метод	Строитель
Цель	Копирование существующих объектов	Делегирование создания подклассам	Пошаговая сборка сложных объектов
Основа	Клонирование	Наследование	Композиция
Гибкость	Высокая в копировании	Высокая в выборе типа	Высокая в конфигурации
Сложность	Средняя	Низкая	Высокая
Когда использовать	Создание объекта дорого	Неизвестен тип объекта	Много параметров у объекта

Спасибо за  
внимание!

Использованные материалы:

<https://refactoringguru.cn>

<https://metanit.com/java/tutorial/>

<https://course.modelware.ru>