

# Паттерны #1

Порождающие: одиночка, строитель, прототип

Неделя 5

## **Порождающие**

Помогают реализовать гибкое  
создание объектов без  
внесения в программу лишних  
зависимостей

# **Паттерны**

## **Порождающие**

Помогают реализовать гибкое  
создание объектов без  
внесения в программу лишних  
зависимостей

## **Поведенческие**

Заботятся об эффективной  
коммуникации между  
объектами

# **Паттерны**

# Паттерны

## Порождающие

Помогают реализовать гибкое создание объектов без внесения в программу лишних зависимостей

## Поведенческие

Заботятся об эффективной коммуникации между объектами

## Структурные

Показывают различные способы построения связей между объектами.

# Паттерны

## Порождающие

Помогают реализовать гибкое создание объектов без внесения в программу лишних зависимостей

## Поведенческие

Заботятся об эффективной коммуникации между объектами

мы тут

## Структурные

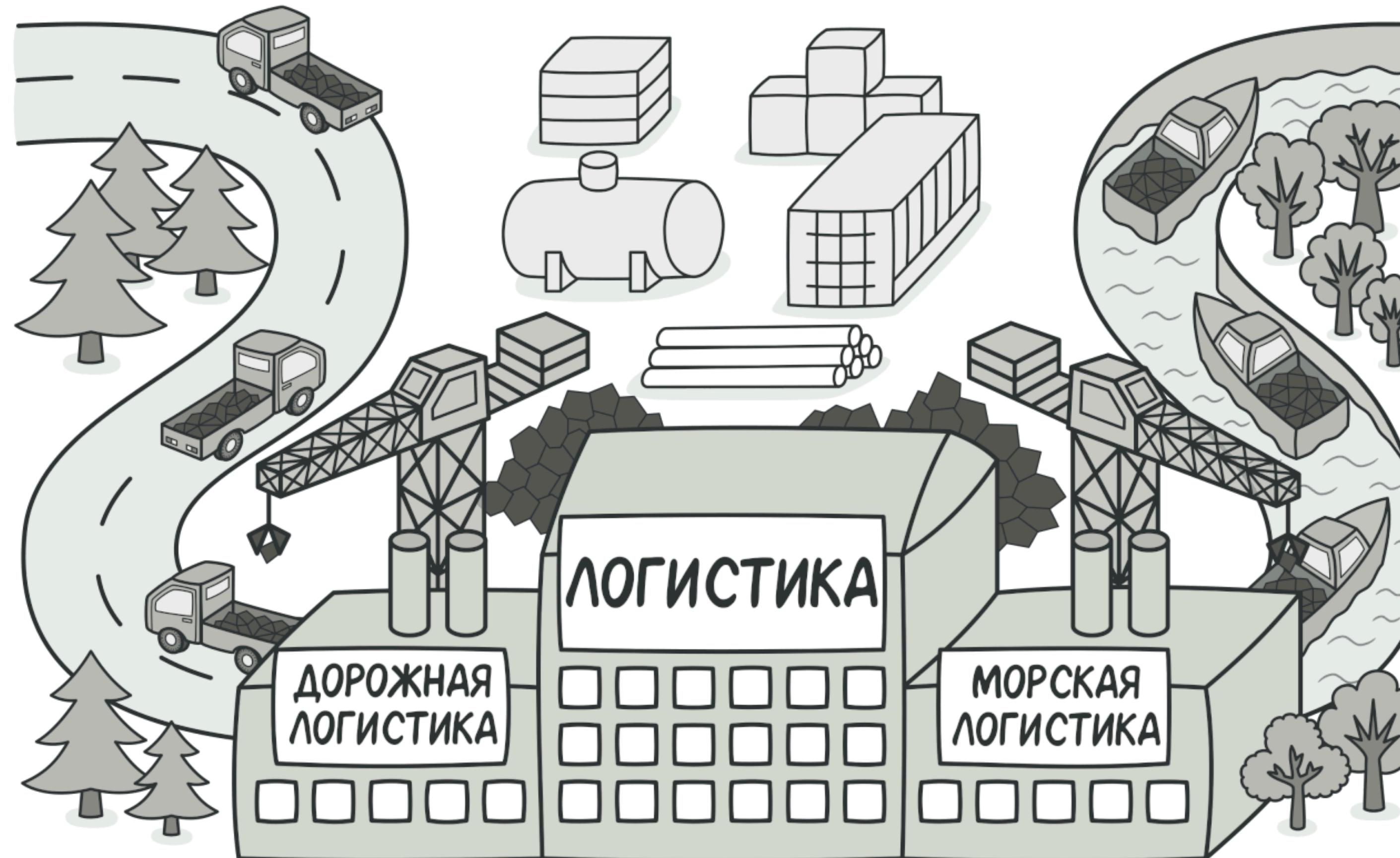
Показывают различные способы построения связей между объектами.

## Порождающие паттерны

- Фабричный метод, абстрактная фабрика
- Одиночка
- Строитель
- Прототип

# Паттерн Фабрика

**Фабричный метод** – это порождающий паттерн проектирования, который определяет общий интерфейс для создания объектов в суперклассе, позволяя подклассам изменять тип создаваемых объектов.



## Когда используем?

- Когда создание объекта подразумевает какую-то логику, а не просто несколько присваиваний, то имеет смысл делегировать задачу выделенной фабрике, а не повторять повсюду один и тот же код.
- Когда заранее неизвестны типы и зависимости объектов, с которыми должен работать ваш код.



*Добавить новый класс не так-то просто, если весь код уже завязан на конкретные классы.*

# Общая структура

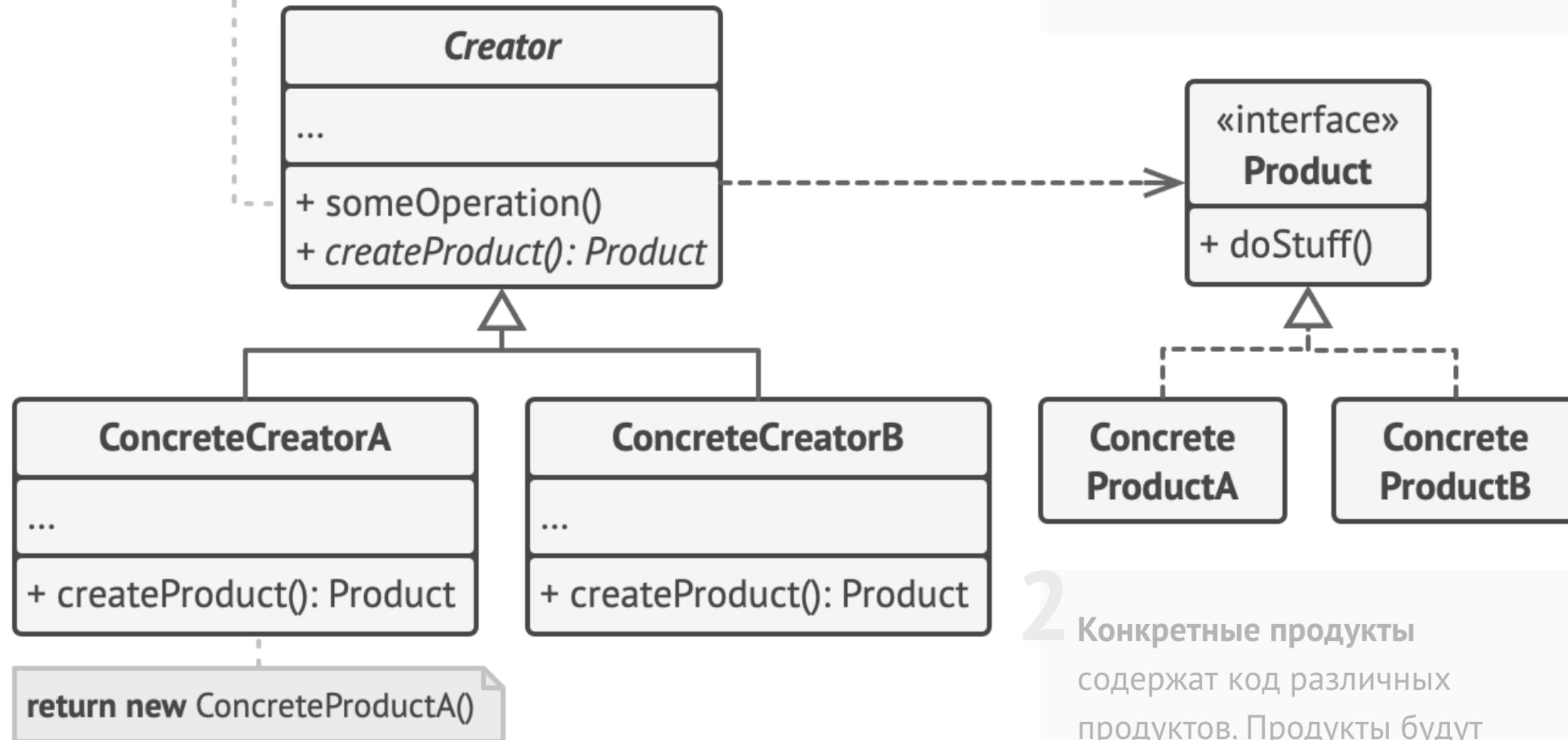
3

Создатель объявляет фабричный метод, который должен возвращать новые объекты продуктов. Важно, чтобы тип результата совпадал с общим интерфейсом продуктов.

Зачастую фабричный метод объявляют абстрактным, чтобы заставить все подклассы реализовать его по-своему. Но он может возвращать и некий стандартный продукт.

Несмотря на название, важно понимать, что создание продуктов **не является** единственной функцией создателя. Обычно он содержит и другой полезный код работы с продуктом. Аналогия: большая софтверная компания может иметь центр подготовки программистов, но основная задача компании – создавать программные продукты, а не готовить программистов.

```
Product p = createProduct()  
p.doStuff()
```



4

Конкретные создатели по-своему реализуют фабричный метод, производя те или иные конкретные продукты.

Фабричный метод не обязан всё время создавать новые объекты. Его можно переписать так, чтобы возвращать существующие объекты из какого-то хранилища или кэша.

1

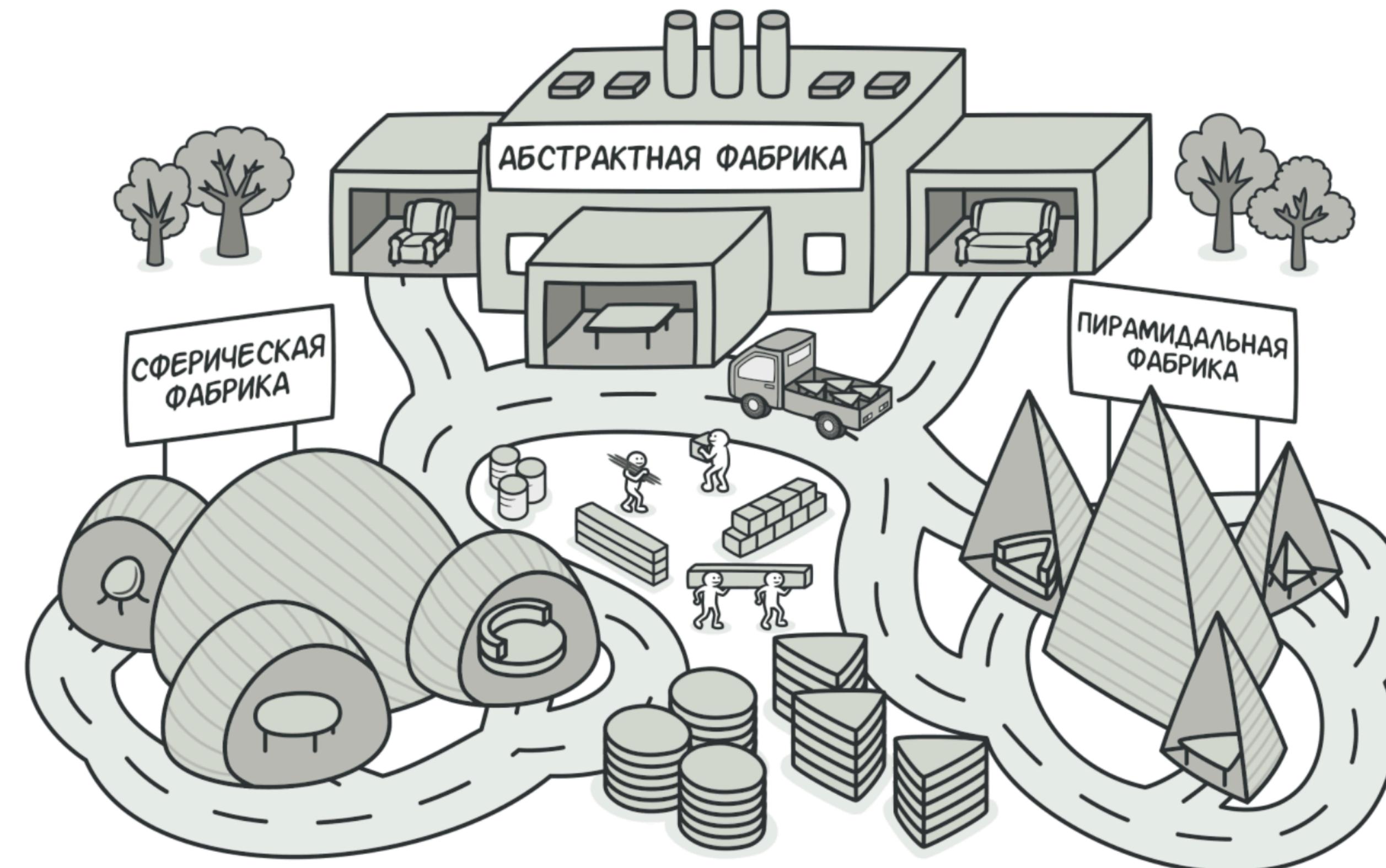
Продукт определяет общий интерфейс объектов, которые может произвести создатель и его подклассы.

2

Конкретные продукты содержат код различных продуктов. Продукты будут отличаться реализацией, но интерфейс у них будет общий.

# Абстрактная фабрика

**Abstract Factory** - фабрика фабрик. То есть фабрика, группирующая индивидуальные, но взаимосвязанные/взаимозависимые фабрики без указания для них конкретных классов.



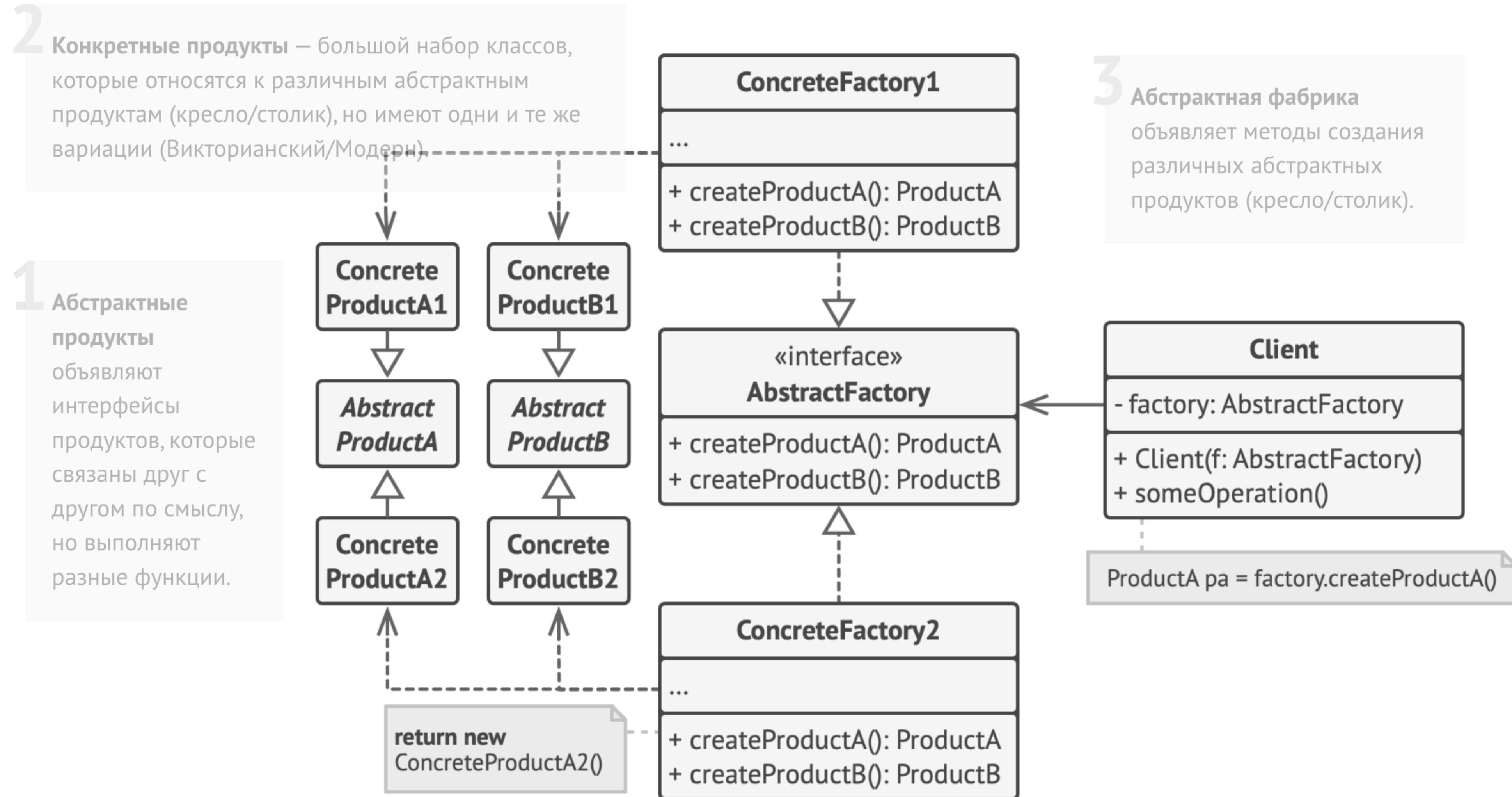
# Абстрактная фабрика. Применимость

**Когда бизнес-логика программы должна работать с разными видами связанных друг с другом продуктов, не завися от конкретных классов продуктов.**

Абстрактная фабрика скрывает от клиентского кода подробности того, как и какие конкретно объекты будут созданы. Но при этом клиентский код может работать со всеми типами создаваемых продуктов, поскольку их общий интерфейс был заранее определён.

**Когда в программе уже используется Фабричный метод, но очередные изменения предполагают введение новых типов продуктов.**

В хорошей программе каждый класс отвечает только за одну вещь. Если класс имеет слишком много фабричных методов, они способны затуманить его основную функцию. Поэтому имеет смысл вынести всю логику создания продуктов в отдельную иерархию классов, применив абстрактную фабрику.



**4** Конкретные фабрики относятся каждая к своей вариации продуктов (Викторианский/Модерн) и реализуют методы абстрактной фабрики, позволяя создавать все продукты определённой вариации.

**5** Несмотря на то, что конкретные фабрики порождают конкретные продукты, сигнатуры их методов должны возвращать соответствующие абстрактные продукты. Это позволит клиентскому коду, использующему фабрику, не привязываться к конкретным классам продуктов. Клиент сможет работать с любыми вариантами продуктов через абстрактные интерфейсы.

**Рассмотрим остальные паттерны...**

# Одиночка (Singleton)

## Почему одиночка?

**Singleton** (с англ. «одиночка») — это паттерн проектирования, гарантирующий, что у класса будет только **один экземпляр**.

К этому экземпляру будет предоставлена **глобальная**, то есть доступная из любой части программы, точка доступа. Если попытаться создать новый объект этого класса, то вернётся уже созданный существующий экземпляр.

# Где используется Singleton?

**Конфигурационные настройки.** Представим, что у нас есть класс с настройками приложения – параметрами базы данных или внешнего вида интерфейса. Имеет смысл реализовать его как Singleton. Это обеспечит одну точку доступа к настройкам, и весь код сможет ссылаться на одни и те же настройки.

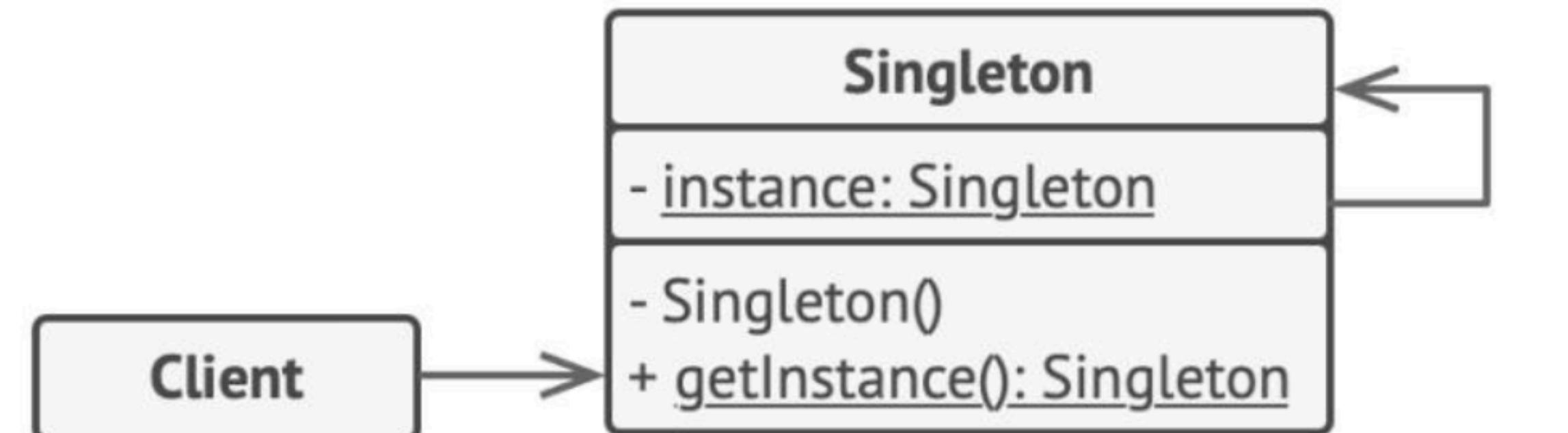
**Подключение к базе данных.** Если наше приложение использует базу данных, Singleton гарантированно создаст только один экземпляр класса, отвечающего за подключение к ней. Так мы предотвратим лишние соединения и упростим подключение в целом.

**Логирование.** Singleton удобно использовать для логов. Вместо создания нового логгера каждый раз, когда нужно что-то залогировать, мы записываем всё в один объект.

**Счётчики и глобальные объекты.** Паттерн «одиночка» подходит для оценки состояния приложения или сбора статистики с его модулей. С классом можно будет взаимодействовать из любой части программы.

**Пул ресурсов.** Если у нас ограниченный пул соединений к внешнему сервису или к другим ресурсам, то Singleton гарантирует, что доступ к ним всегда будет идти через единственный экземпляр.

# Структура



1

Одиночка определяет статический метод `getInstance`, который возвращает единственный экземпляр своего класса.

Конструктор одиночки должен быть скрыт от клиентов. Вызов метода `getInstance` должен стать единственным способом получить объект этого класса.

```
if (instance == null) {
    // Внимание, если вы пишете
    // многопоточный код, то здесь
    // нужно синхронизировать потоки.
    instance = new Singleton()
}
return instance
```

# Простой пример

## Плюсы:

- Простота и прозрачность кода
- Потокобезопасность
- Высокая производительность в многопоточной среде

## Минусы:

- Не ленивая инициализация.

```
public class Singleton {  
    private static final Singleton INSTANCE = new Singleton();  
  
    private Singleton() {  
    }  
  
    public static Singleton getInstance() {  
        return INSTANCE;  
    }  
}
```

# Lazy Initialization

## Плюсы:

- Ленивая инициализация.

## Минусы:

- Не потокобезопасно

```
public class Singleton {  
    private static Singleton INSTANCE;  
  
    private Singleton() {}  
  
    public static Singleton getInstance() {  
        if (INSTANCE == null) {  
            INSTANCE = new Singleton();  
        }  
        return INSTANCE;  
    }  
}
```

# Synchronized Accessor

## Плюсы:

- Ленивая инициализация.
- Потокобезопасность

## Минусы:

- Низкая производительность в многопоточной среде

```
public class Singleton {  
    private static Singleton INSTANCE;  
  
    private Singleton() {  
    }  
  
    public static synchronized Singleton getInstance() {  
        if (INSTANCE == null) {  
            INSTANCE = new Singleton();  
        }  
        return INSTANCE;  
    }  
}
```

# Double Checked Locking

## Плюсы:

- Ленивая инициализация.
- Потокобезопасность
- Высокая производительность в многопоточной среде

```
public class Singleton {  
    private static Singleton INSTANCE;  
  
    private Singleton() {  
    }  
  
    public static Singleton getInstance() {  
        if (INSTANCE == null) {  
            synchronized (Singleton.class) {  
                if (INSTANCE == null) {  
                    INSTANCE = new Singleton();  
                }  
            }  
        }  
        return INSTANCE;  
    }  
}
```

# Проблемы Singleton

**Глобальное состояние.** Singleton доступен из любой части программы, то есть сломать его может кто угодно.

**Потокобезопасность.** Базовая реализация паттерна «одиночка» непотокобезопасна. Если не предусмотрены механизмы синхронизации, разные потоки могут наштамповать кучу синглтонов.

**Тестирование.** Для тестирования класса с Singleton потребуются мок-объекты (объекты созданные для тестирования). А заменить реальный объект на мок не всегда легко или вообще возможно.

**Нарушение принципа единственной ответственности.** Принципы грамотного проектирования и просто хороший тон гласят: каждый класс должен делать только одну важную вещь. Другими словами, класс должен быть специализированным.

## Реализуем Singleton

В нашем проекте Singleton можно использовать в качестве объекта-конфигурации, который будет отвечать за **создание контейнера зависимостей и его получение**.

**Строитель**

# Что за строитель?

**Строитель (Builder)** - шаблон проектирования, который инкапсулирует создание объекта и позволяет разделить его на различные этапы.

```
House houseBuilder = new HouseBuilder()  
    .setFrontPorch(HouseBuilder.COLOR.WHITE,  
                   HouseBuilder.SIZE.EXTRA_LARGE)  
    .build();
```

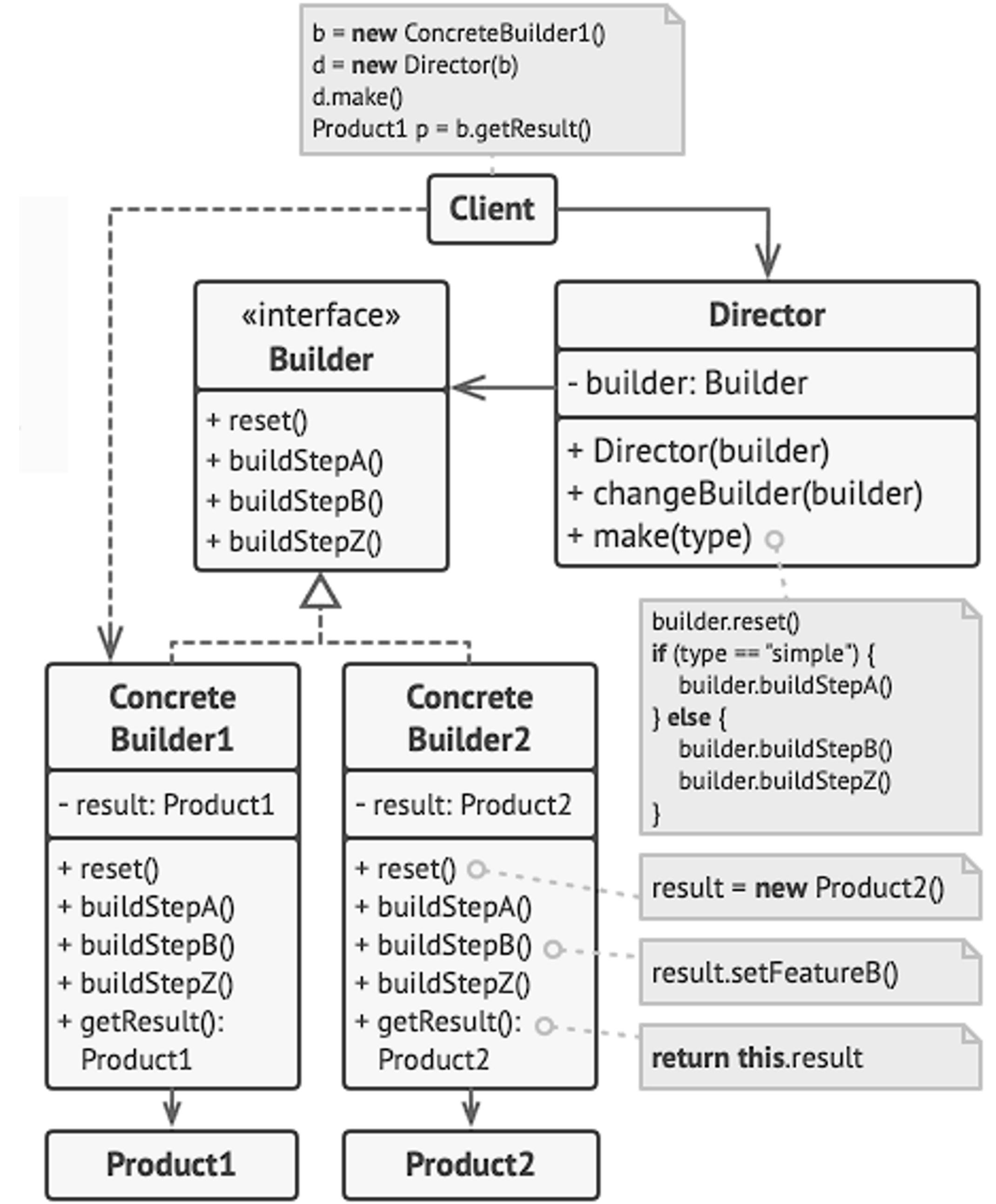


## Когда использовать?

- Когда процесс создания нового объекта не должен зависеть от того, **из каких частей этот объект состоит** и как эти части связаны между собой
- Когда необходимо обеспечить получение **различных вариаций** объекта в процессе его создания

# Структура

1. **Интерфейс строителя** объявляет шаги конструирования продуктов, общие для всех видов строителей.
2. **Конкретные строители** реализуют строительные шаги, каждый по-своему. Конкретные строители могут производить разнородные объекты, не имеющие общего интерфейса.
3. **Продукт** — создаваемый объект. Продукты, сделанные разными строителями, не обязаны иметь общий интерфейс.
4. **Директор** определяет порядок вызова строительных шагов для производства той или иной конфигурации продуктов.
5. **Клиент** подаёт в конструктор директора уже готовый объект-строитель, и в дальнейшем данный директор использует только его.



# Когда использовать? Кейс 1

## Когда вы хотите избавиться от «телескопического конструктора»

Допустим, у вас есть один конструктор с десятью опциональными параметрами. Его неудобно вызывать, поэтому вы создали ещё десять конструкторов с меньшим количеством параметров.

Всё, что они делают — это переадресуют вызов к базовому конструктору, подавая какие-то значения по умолчанию в параметры, которые пропущены в них самих.

```
class Pizza {  
    Pizza(int size) { ... }  
    Pizza(int size, boolean cheese) { ... }  
    Pizza(int size, boolean cheese, boolean pepperoni) { ... }  
    // ...
```

Паттерн Строитель позволяет собирать объекты пошагово, вызывая только те шаги, которые вам нужны.

А значит, больше не нужно пытаться «запихнуть» в конструктор все возможные опции продукта.

## Когда использовать? Кейс 2

**Когда ваш код должен создавать разные представления какого-то объекта.  
Например, деревянные и железобетонные дома.**

Строитель можно применить, если создание нескольких представлений объекта состоит из одинаковых этапов, которые отличаются в деталях.

Интерфейс строителей определит все возможные этапы конструирования.  
Каждому представлению будет соответствовать собственный класс-строитель.  
А порядок этапов строительства будет задавать класс-директор.

## **Когда использовать? Кейс 3**

**Когда вам нужно собирать сложные составные объекты, например, деревья Компоновщика.**

Строитель конструирует объекты пошагово, а не за один проход. Более того, шаги строительства можно выполнять рекурсивно. А без этого не построить древовидную структуру, вроде Компоновщика.

Заметьте, что Строитель не позволяет посторонним объектам иметь доступ к конструируемому объекту, пока тот не будет полностью готов.

Это предохраняет клиентский код от получения незаконченных «битых» объектов.

# Реализуем Builder

Сейчас в нашем проекте нет необходимости его использовать, придумаем кейс.

Предположим мы хотим строить отчеты о работе нашей системы в таком формате (при этом, чтобы возможно было изменять блоки местами, включать/не включать их.

Отчет за 2025-02-10

Операция: Инициализация системы

Покупатели:

- Имя: Ваня. Автомобиль: { Нет }
- Имя: Света. Автомобиль: { Нет }
- Имя: Сергей. Автомобиль: { Нет }
- Имя: Алексей. Автомобиль: { Нет }

Операция: Продажа автомобиля

Покупатели:

- Имя: Ваня. Автомобиль: { Номер: 1. Двигатель: Педальный двигатель. Размер педалей: 2 }
- Имя: Света. Автомобиль: { Номер: 3. Двигатель: Двигатель с ручным приводом }
- Имя: Сергей. Автомобиль: { Номер: 2. Двигатель: Педальный двигатель. Размер педалей: 3 }
- Имя: Алексей. Автомобиль: { Нет }

# Прототип

# Прототип

**Prototype** – позволяет создавать объекты на основе уже ранее созданных объектов-прототипов. То есть по сути данный паттерн предлагает технику клонирования объектов.



# Структура

1. **Интерфейс прототипов** описывает операции клонирования.
2. **Конкретный прототип** реализует операцию клонирования самого себя. Помимо банального копирования значений всех полей, здесь могут быть спрятаны различные сложности, о которых не нужно знать клиенту.
3. **Клиент** создаёт копию объекта, обращаясь к нему через общий интерфейс прототипов.

