

Паттерны Проектирования

Design Patterns

Что такое паттерны проектирования

Паттерны проектирования — это повторно используемые решения типичных задач проектирования ПО.

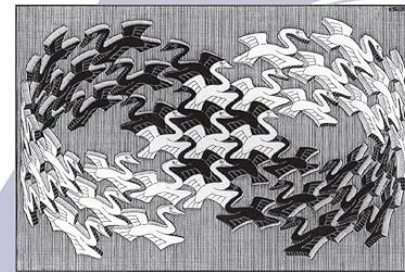
Помогают писать гибкий, расширяемый и поддерживаемый код.

Термин стал популярен после книги
“Design Patterns: Elements of Reusable Object-Oriented Software”

Design Patterns

Elements of Reusable
Object-Oriented Software

Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides



Cover art © 1994 M.C. Escher / Cordon Art - Baarn - Holland. All rights reserved.

Foreword by Grady Booch

Признаки плохого кода

- дублирование кода;
- длинный метод;
- большой класс;
- длинный список параметров;
- классы данных;
- несгруппированные данные.

Причины возникновения плохого кода

- Частые изменения в требованиях, противоречащие исходной архитектуре;
- недостаточно времени сделать работу качественно;
- глупый менеджер/начальник/заказчик и т.д.

Настоящие причины возникновения плохого кода

- **Непрофессионализм**
- **Лень**

Закон Леблана

«Потом равносильно никогда»

Зачем нужны паттерны

- Ускоряют разработку — не нужно изобретать велосипед.
- Повышают читаемость и понятность кода.
- Облегчают командную работу.
- Помогают говорить с коллегами “на одном языке”:
- «здесь используем *Singleton*» и т.п.

Паттерны и Python

- Python — динамический язык с простыми конструкциями.
- Многие паттерны можно реализовать проще, чем в Java или C++.
- Некоторые паттерны встроены в язык (например, Decorator через `@decorator`).
- Главное — понимать принцип, а не заучивать форму

Классификация паттернов

Порождающие — как создавать объекты.

Структурные — как объединять классы и объекты.

Поведенческие — как объекты взаимодействуют между собой.

Порождающие паттерны



Фабричный метод

Factory Method

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



Абстрактная фабрика

Abstract Factory

Позволяет создавать семейства связанных объектов, не привязываясь к конкретным классам создаваемых объектов.

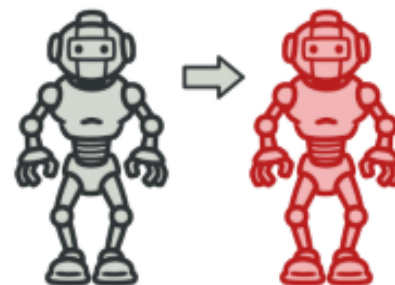
Порождающие паттерны



Строитель

Builder

Позволяет создавать сложные объекты пошагово. Строитель даёт возможность использовать один и тот же код строительства для получения разных представлений объектов.



Прототип

Prototype

Позволяет копировать объекты, не вдаваясь в подробности их реализации.

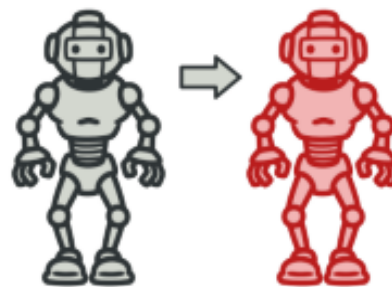
Порождающие паттерны



Строитель

Builder

Позволяет создавать сложные объекты пошагово. Строитель даёт возможность использовать один и тот же код строительства для получения разных представлений объектов.



Прототип

Prototype

Позволяет копировать объекты, не вдаваясь в подробности их реализации.

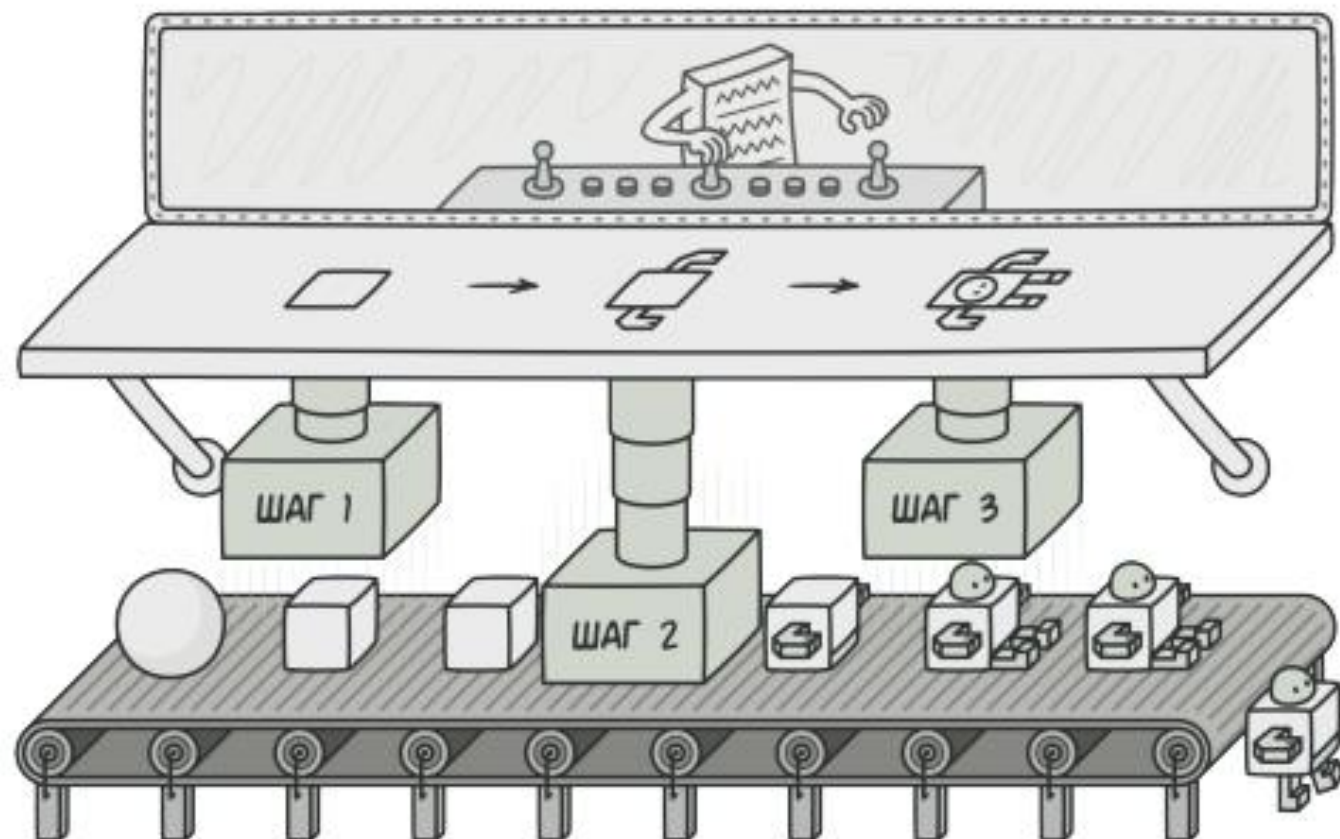
Порождающие паттерны



Одиночка

Singleton

Гарантирует, что у класса есть только один экземпляр, и предоставляет к нему глобальную точку доступа.

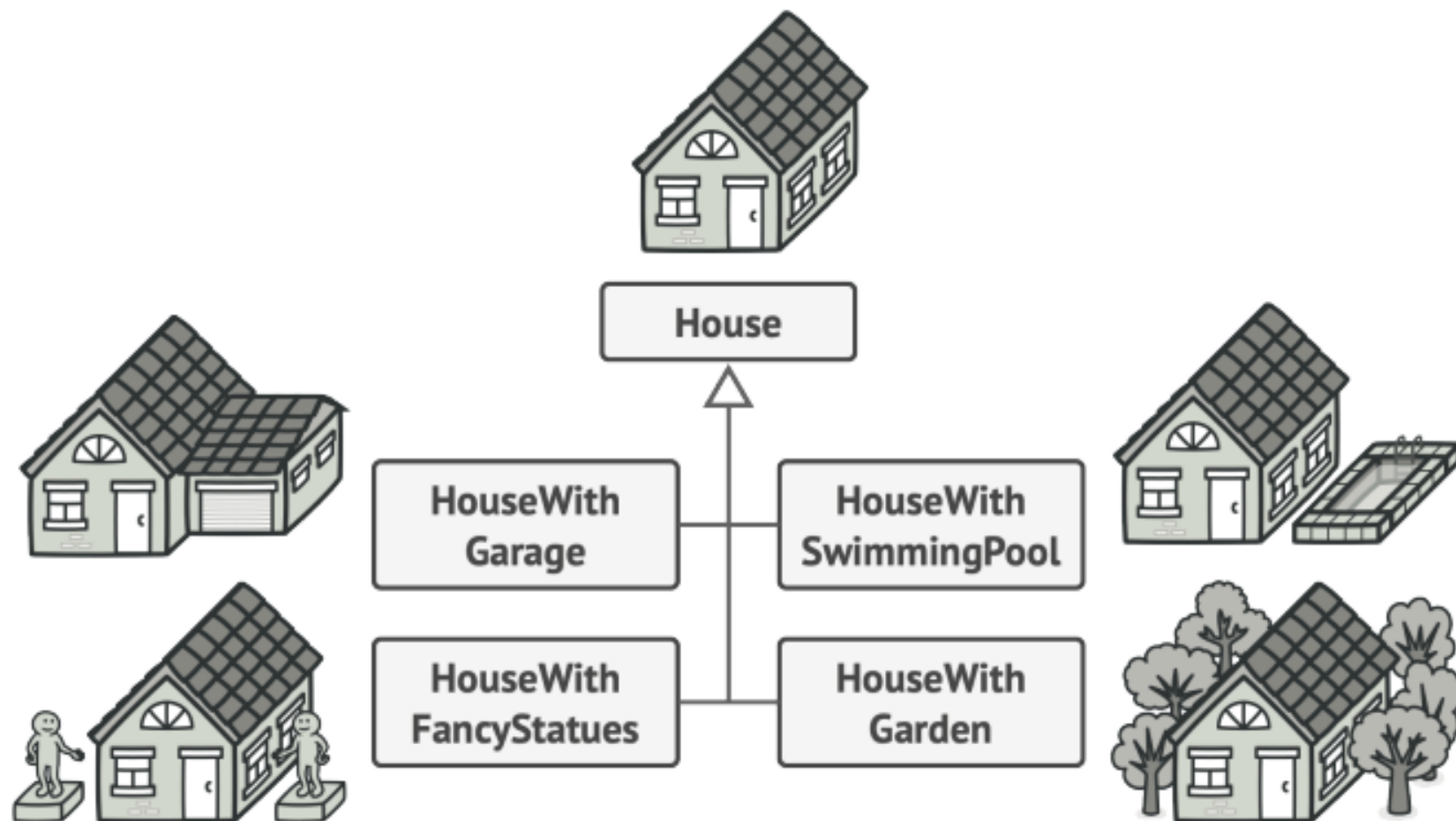


СТРОИТЕЛЬ

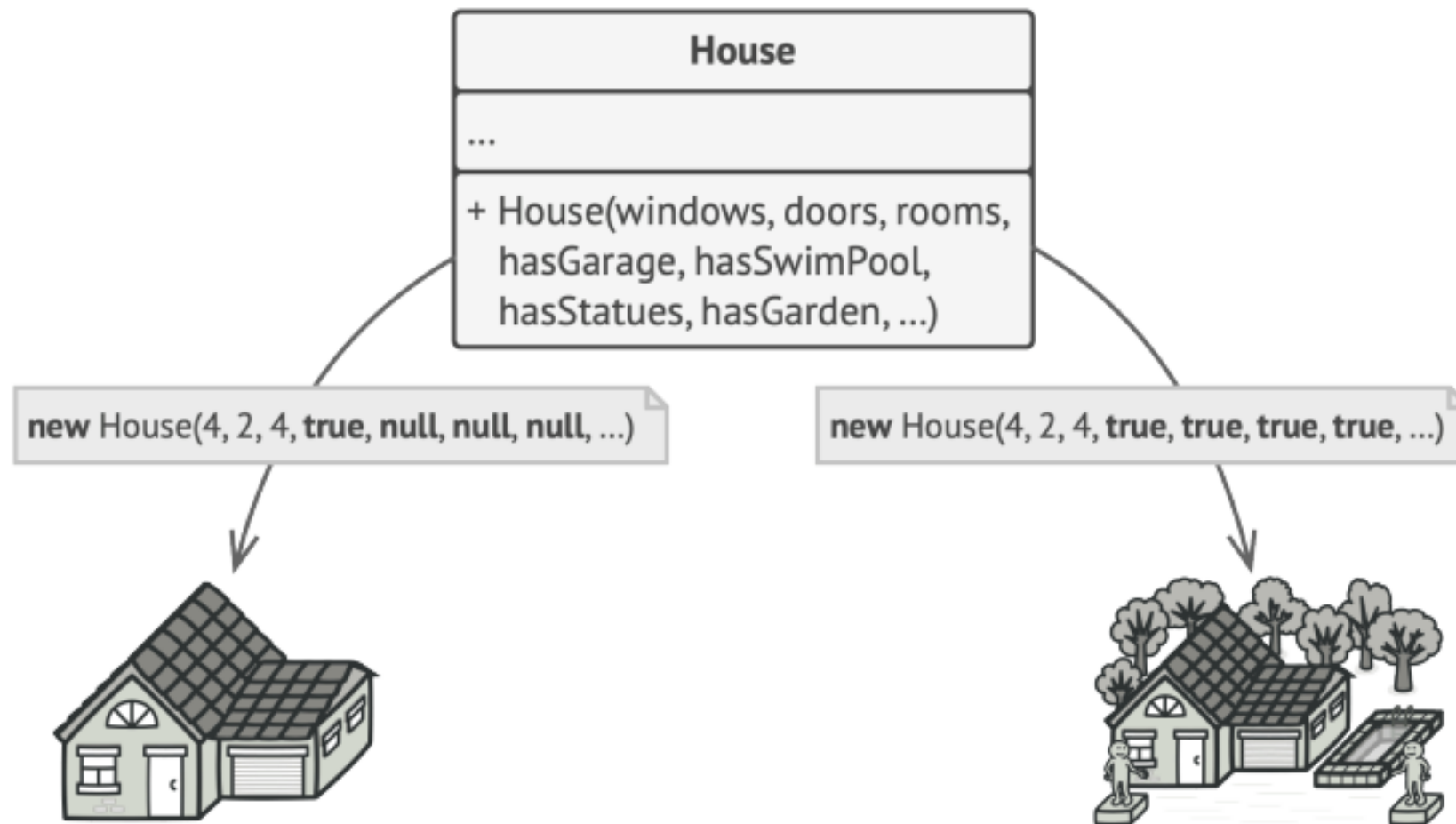
Также известен как: Builder

Строитель — это порождающий паттерн проектирования, который позволяет создавать сложные объекты пошагово. Строитель даёт возможность использовать один и тот же код строительства для получения разных представлений объектов

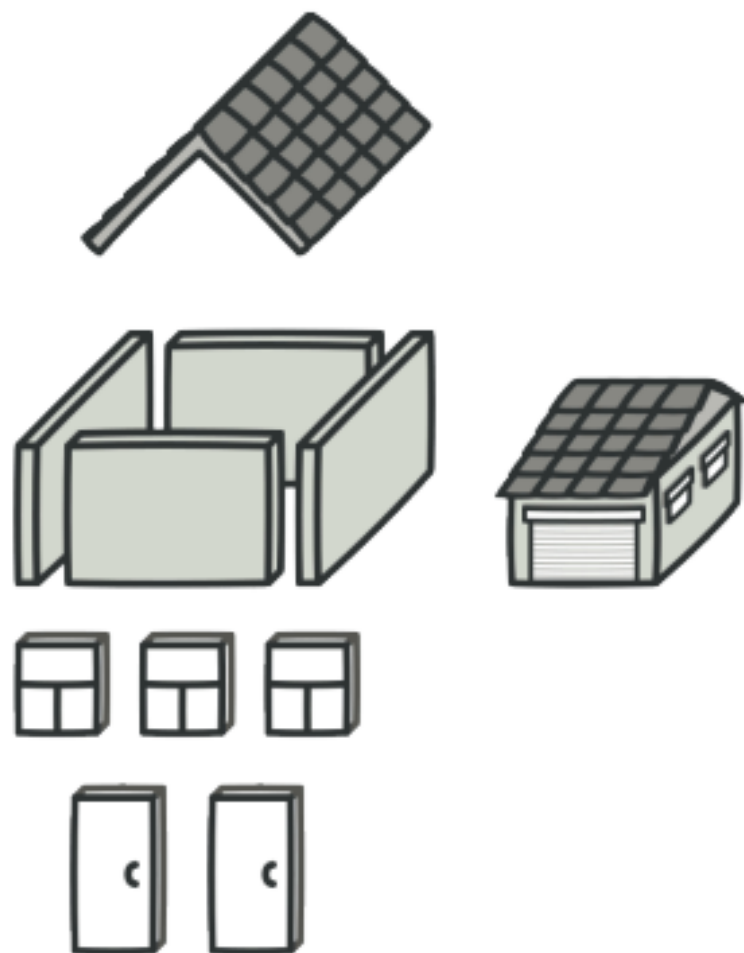
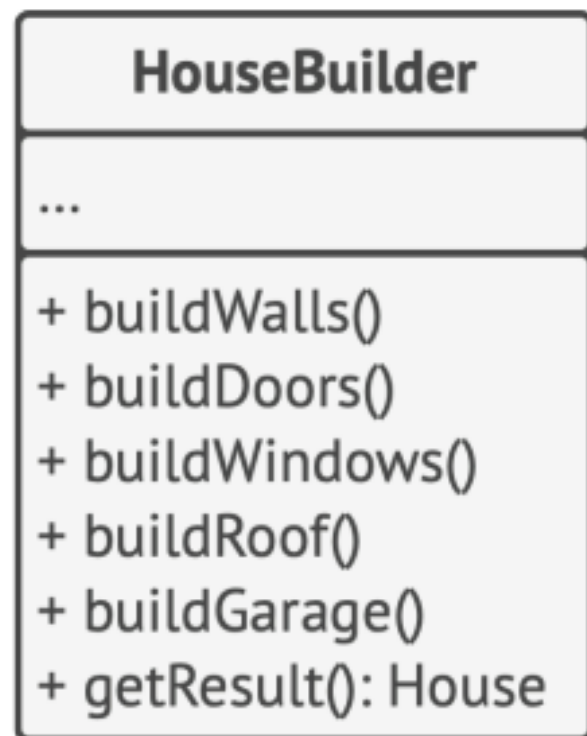
Представьте сложный объект, требующий кропотливой пошаговой инициализации множества полей и вложенных объектов. Код инициализации таких объектов обычно спрятан внутри монструозного конструктора с десятком параметров. Либо ещё хуже — расплён по всему клиентскому коду. .



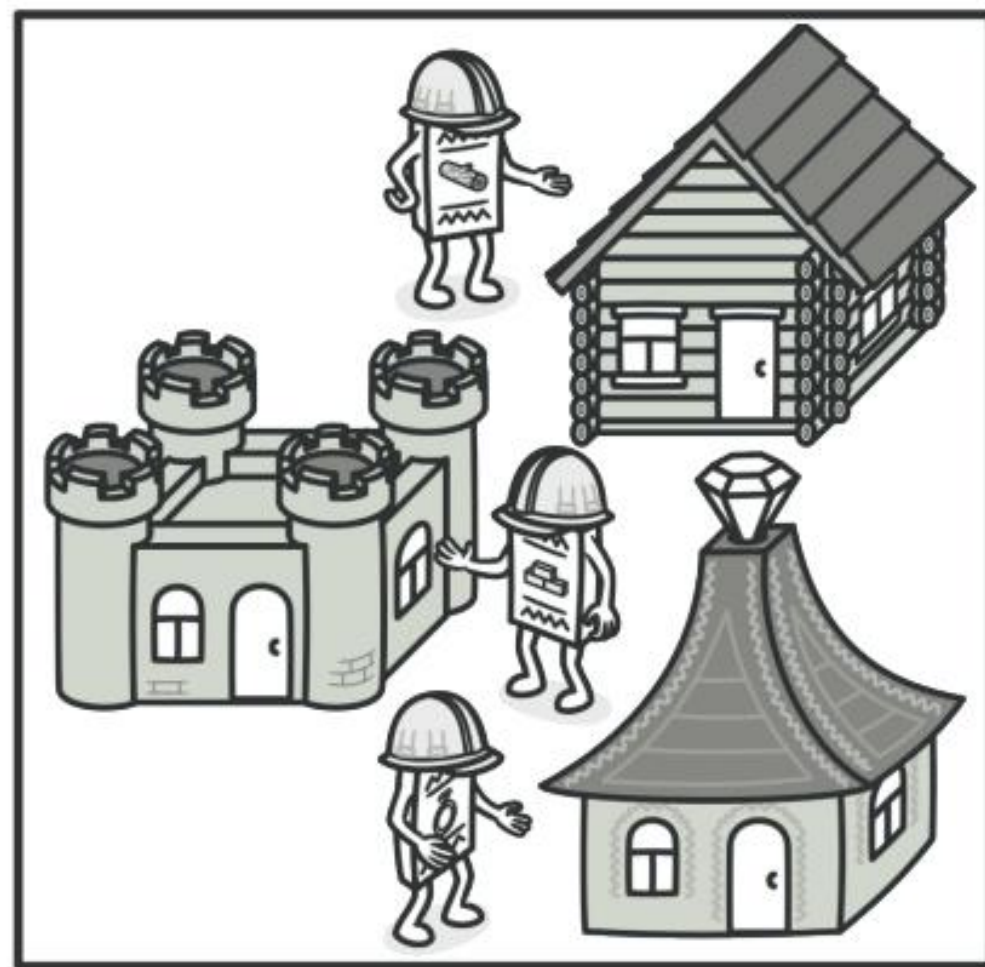
Создав кучу подклассов для всех конфигураций объектов, вы можете излишне усложнить программу.



*Конструктор со множеством параметров имеет свой недостаток:
не все параметры нужны большую часть времени.*



*Строитель позволяет создавать сложные объекты пошагово.
Промежуточный результат всегда остаётся защищён.*

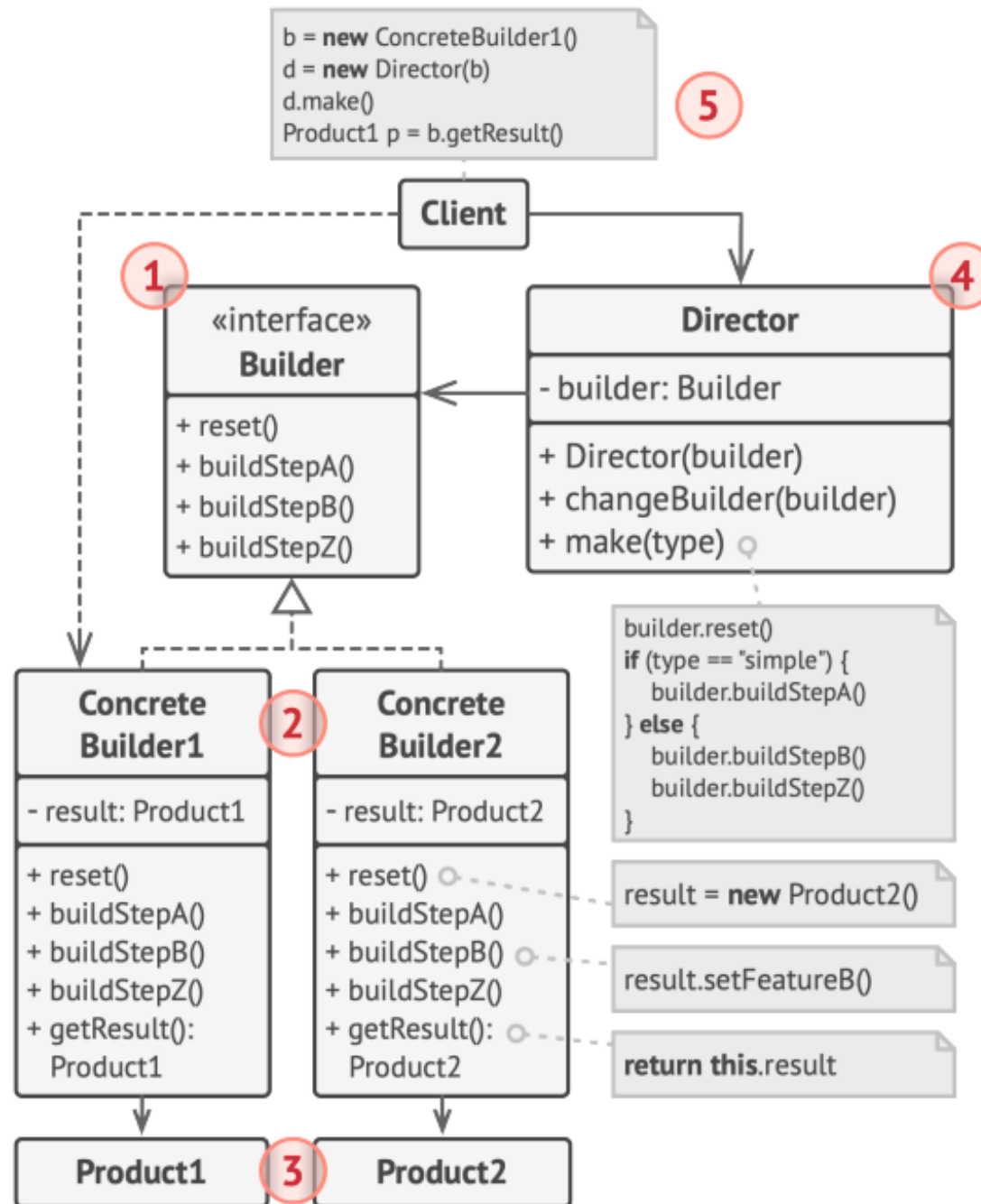


Разные строители выполняют одну и ту же задачу по-разному.

Директор



*Директор знает, какие шаги должен выполнить объект-строитель,
чтобы произвести продукт.*





ФАБРИЧНЫЙ МЕТОД

Также известен как: Виртуальный конструктор, Factory Method

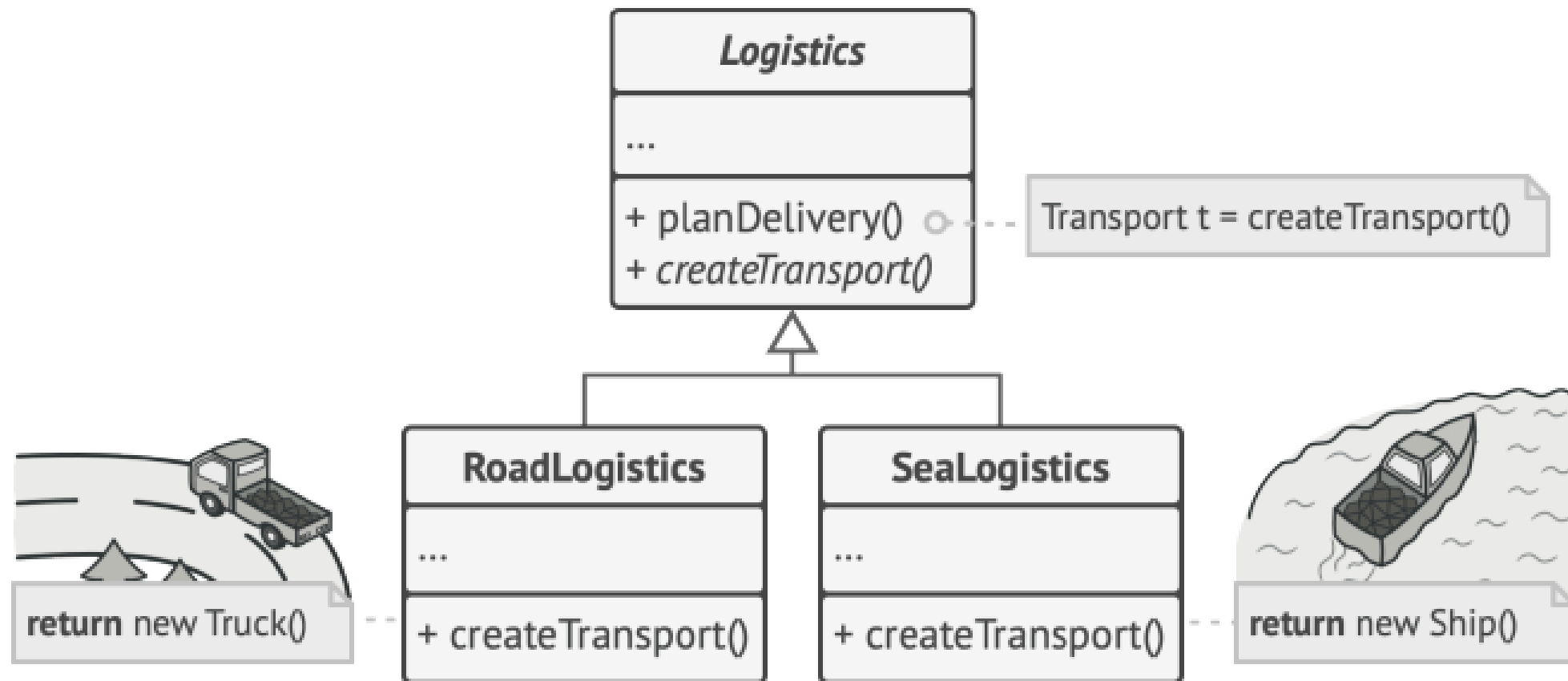
- Фабричный метод — это порождающий паттерн проектирования, который определяет общий интерфейс для создания объектов в суперклассе, позволяя подклассам изменять тип создаваемых объектов.

Проблема

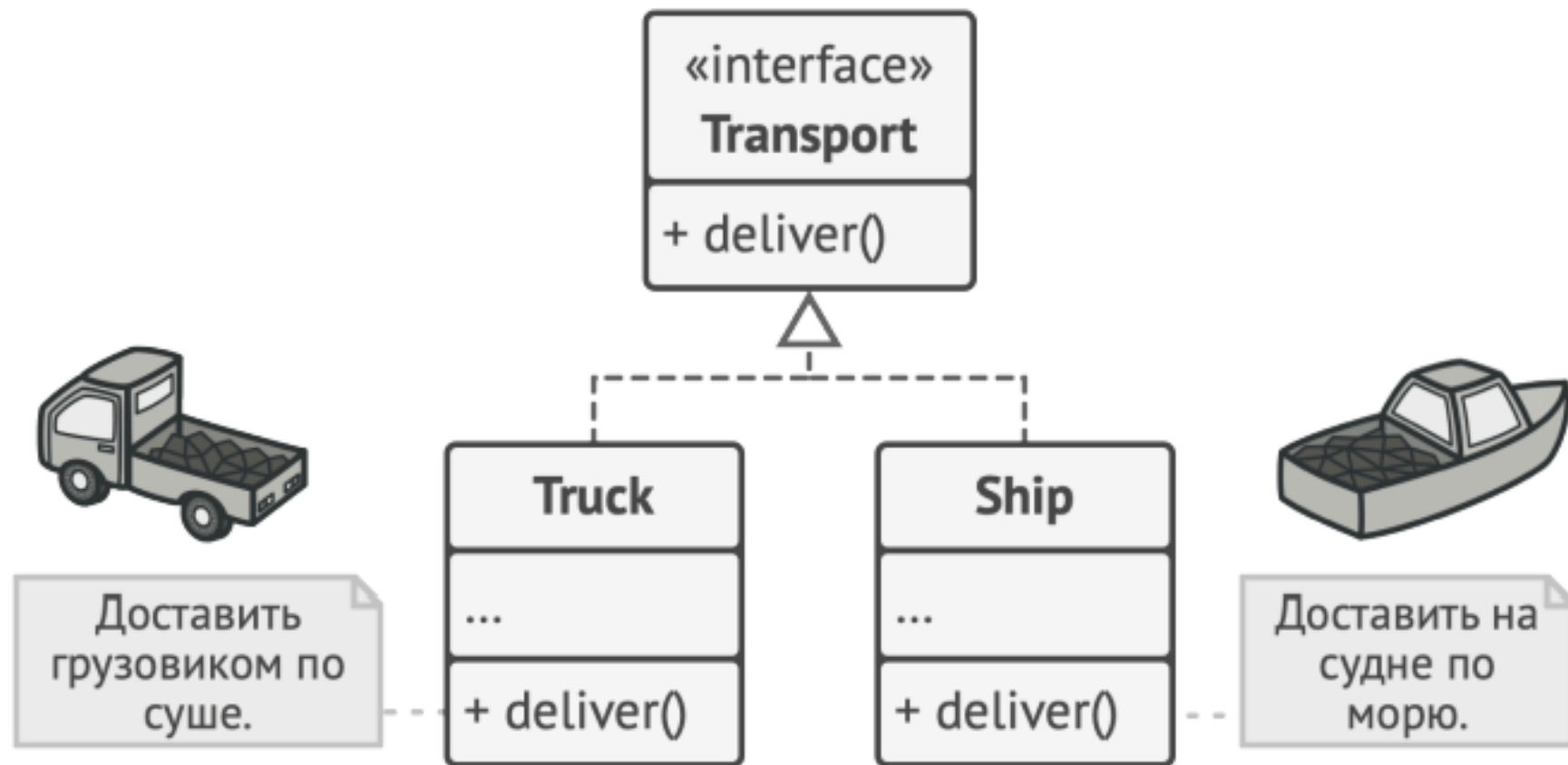
- Представьте, что вы создаёте программу управления грузовыми перевозками. Сперва вы рассчитываете перевозить товары только на автомобилях. Поэтому весь ваш код работает с объектами класса Грузовик .
- В какой-то момент ваша программа становится настолько известной, что морские перевозчики выстраиваются в очередь и просят добавить поддержку морской логистики в программу.



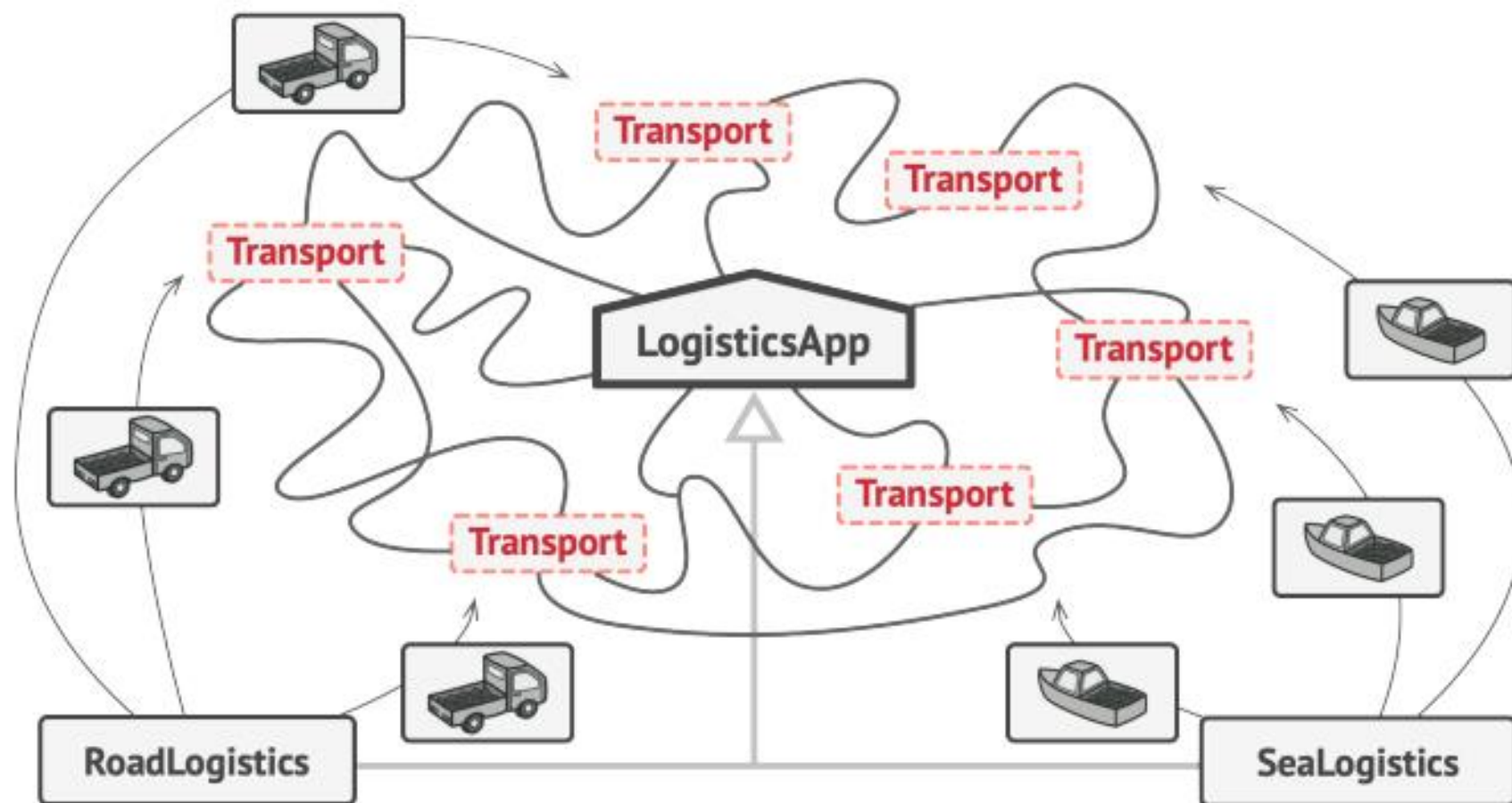
Добавить новый класс не так-то просто, если весь код уже завязан на конкретные классы.



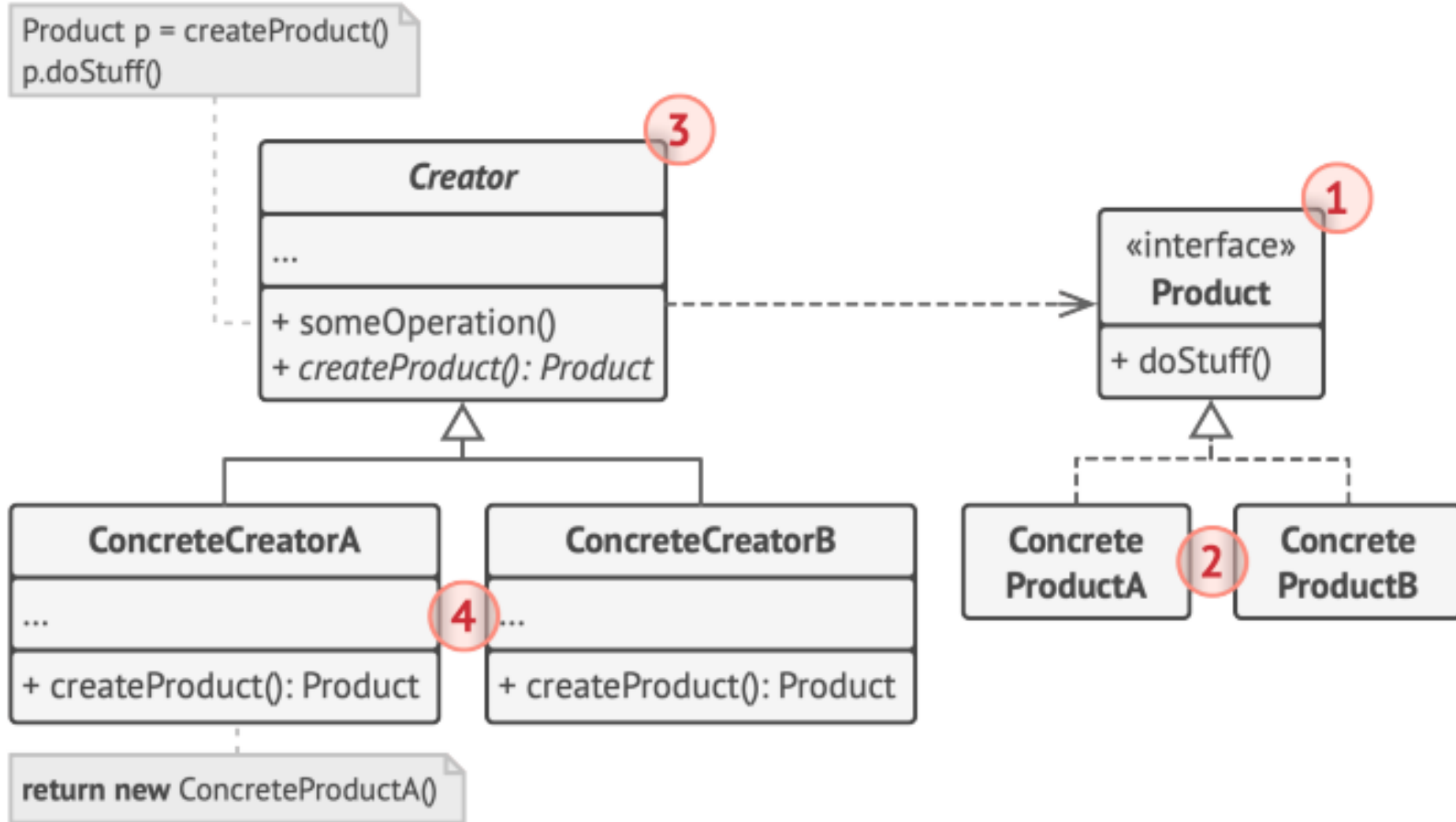
Подклассы могут изменять класс создаваемых объектов.

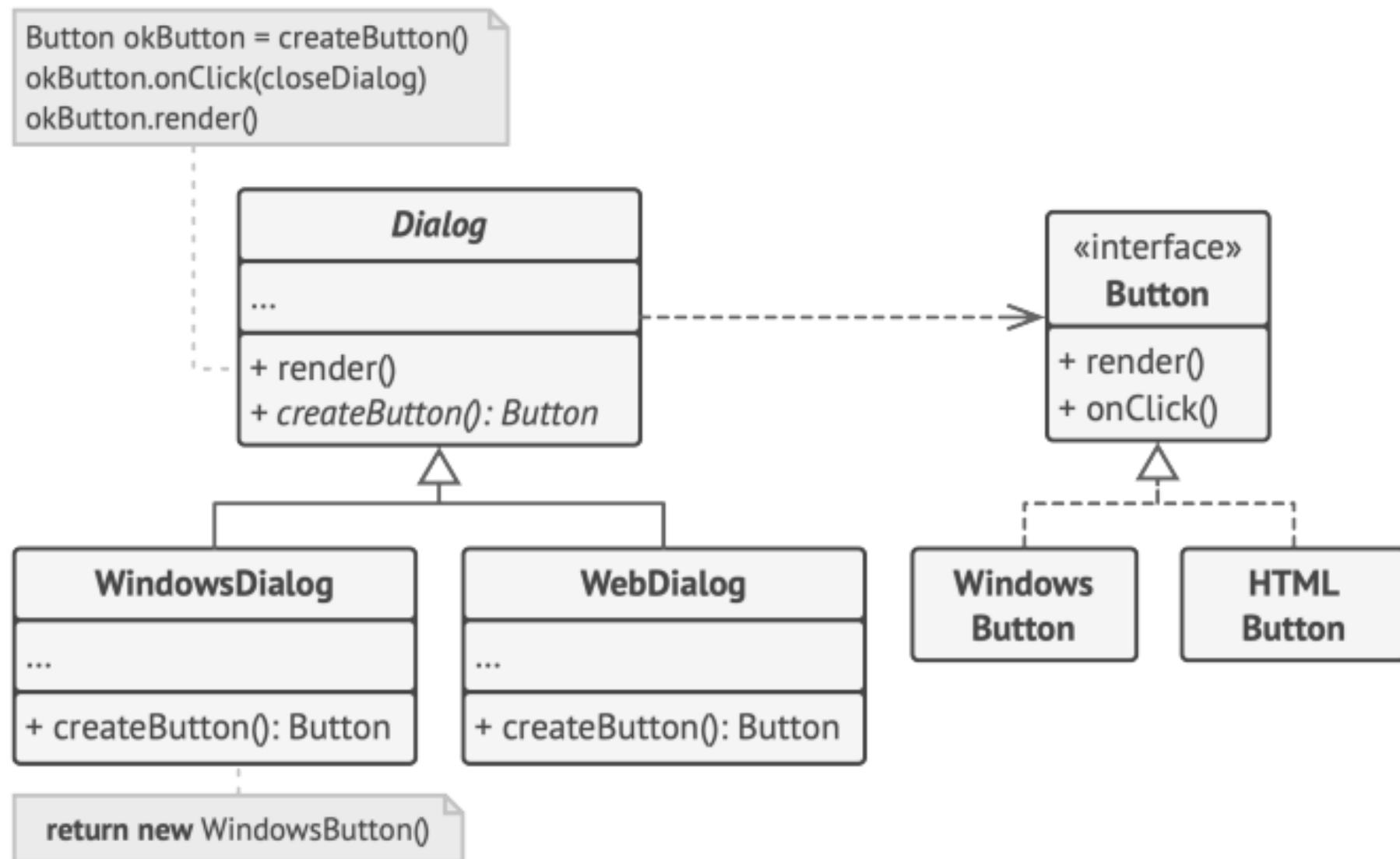


Все объекты-продукты должны иметь общий интерфейс.



*Пока все продукты реализуют общий интерфейс, их объекты можно
взаимозаменять в клиентском коде.*



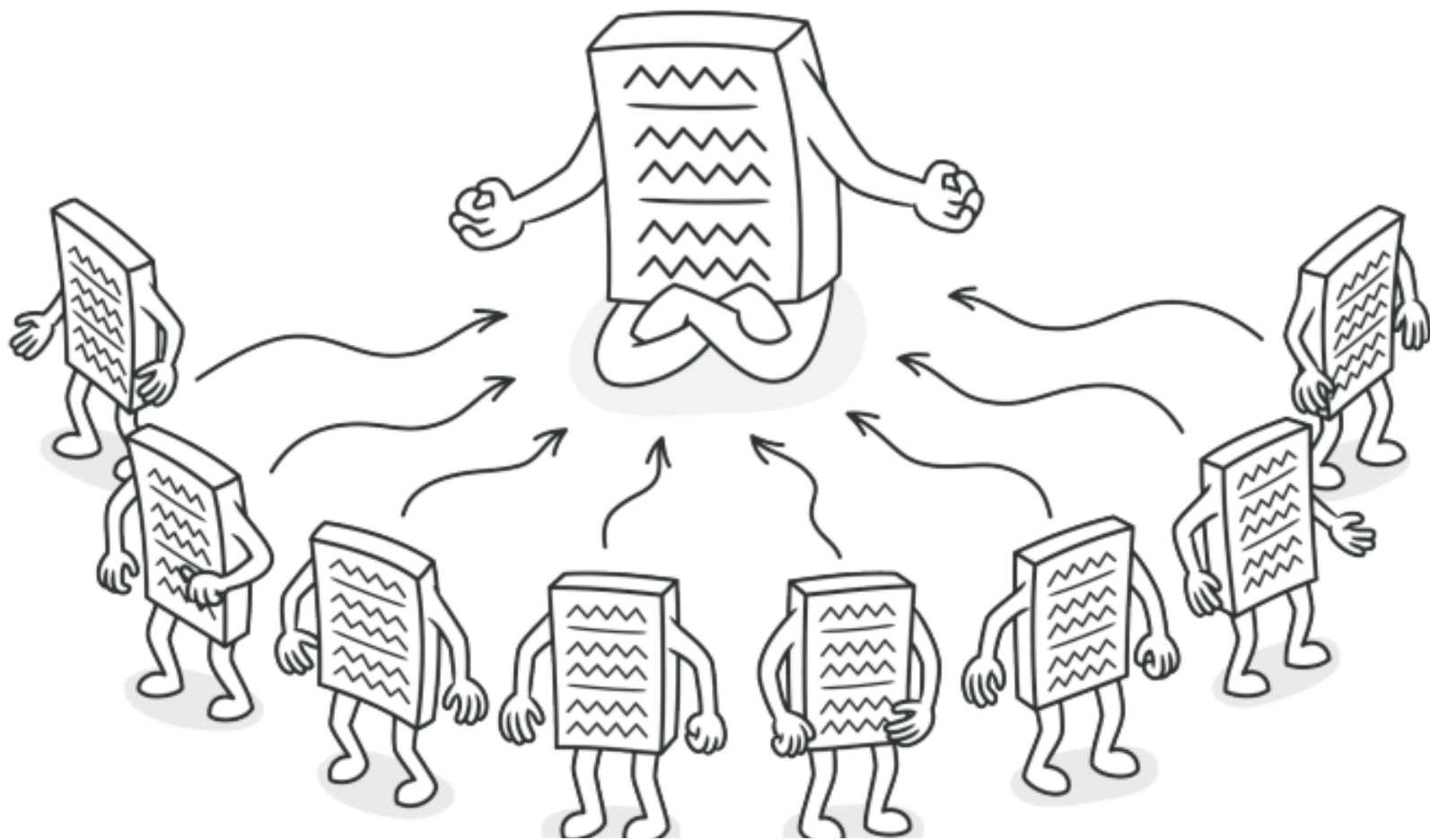


Пример кросс-платформенного диалога.

Что выводит на экран ?



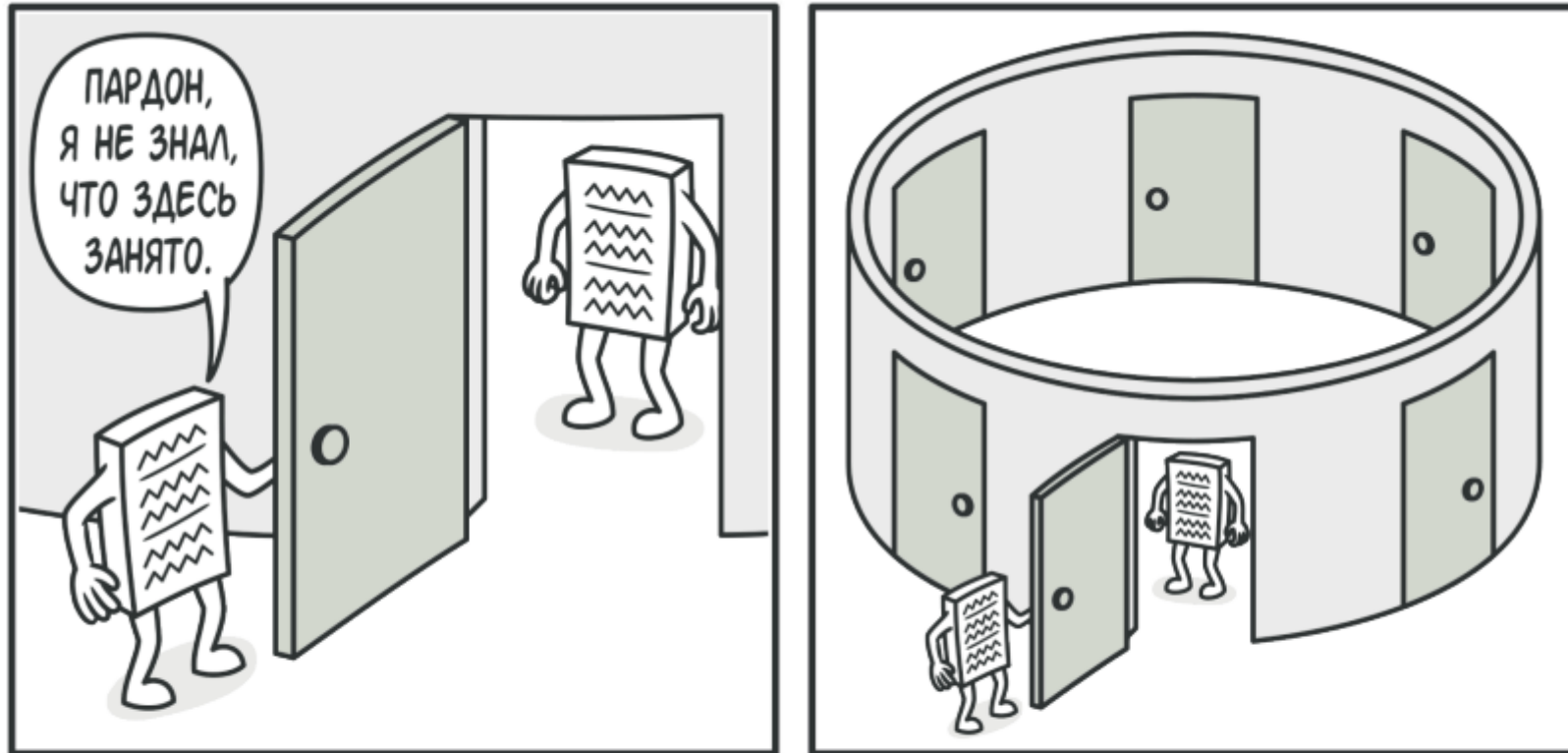
```
1  a_app = Application()
2  b_app = Application()
3
4  a_app.state = "running"
5  b_app.state = "stopped"
6
7  print(a_app.state)
8  print(b_app.state)
```



ОДИНОЧКА

Также известен как: Singleton

Одиночка — это порождающий паттерн проектирования, который гарантирует, что у класса есть только один экземпляр, и предоставляет к нему глобальную точку доступа.



Клиенты могут не подозревать, что работают с одним и тем же объектом.

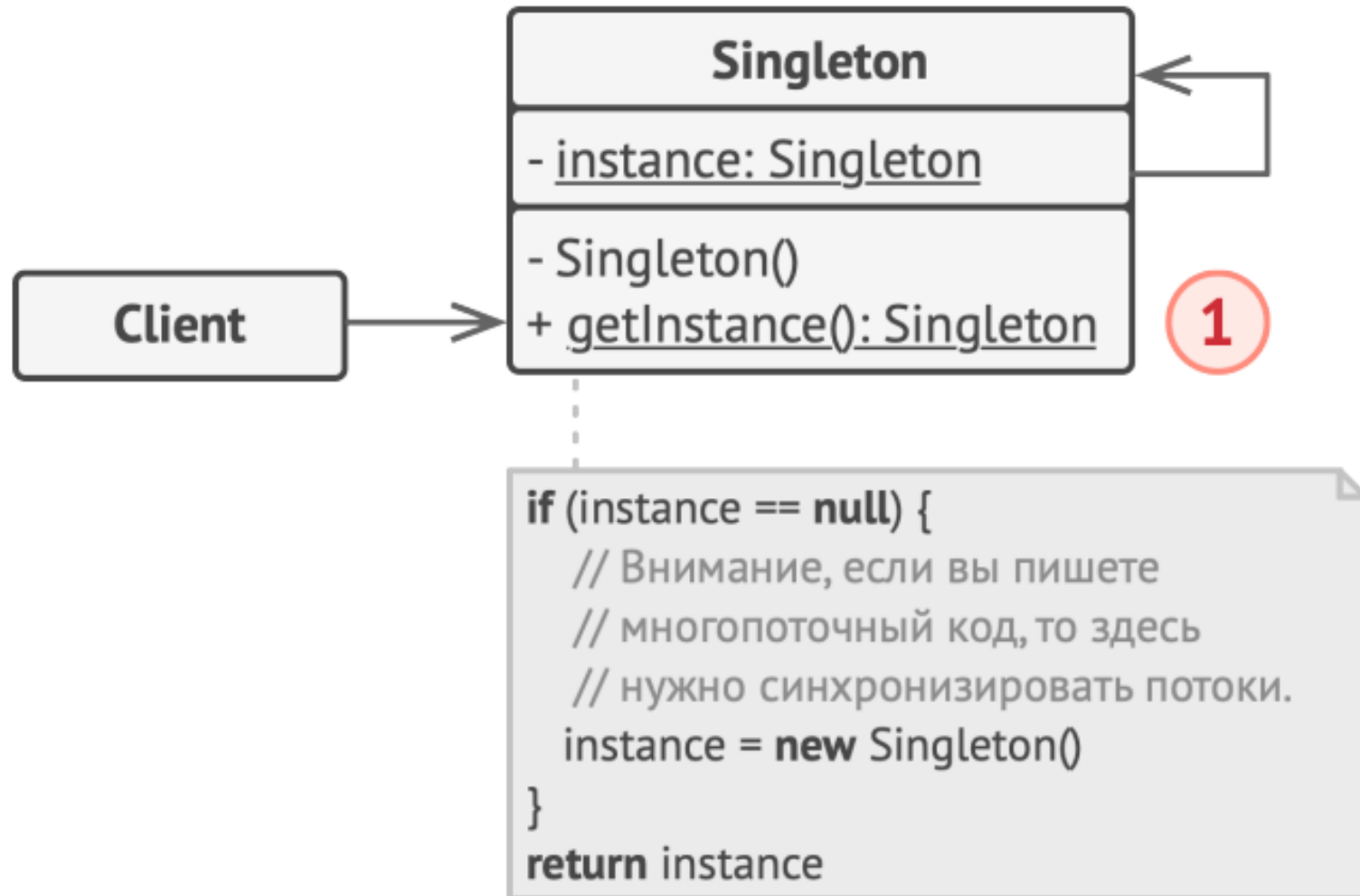
Проблема

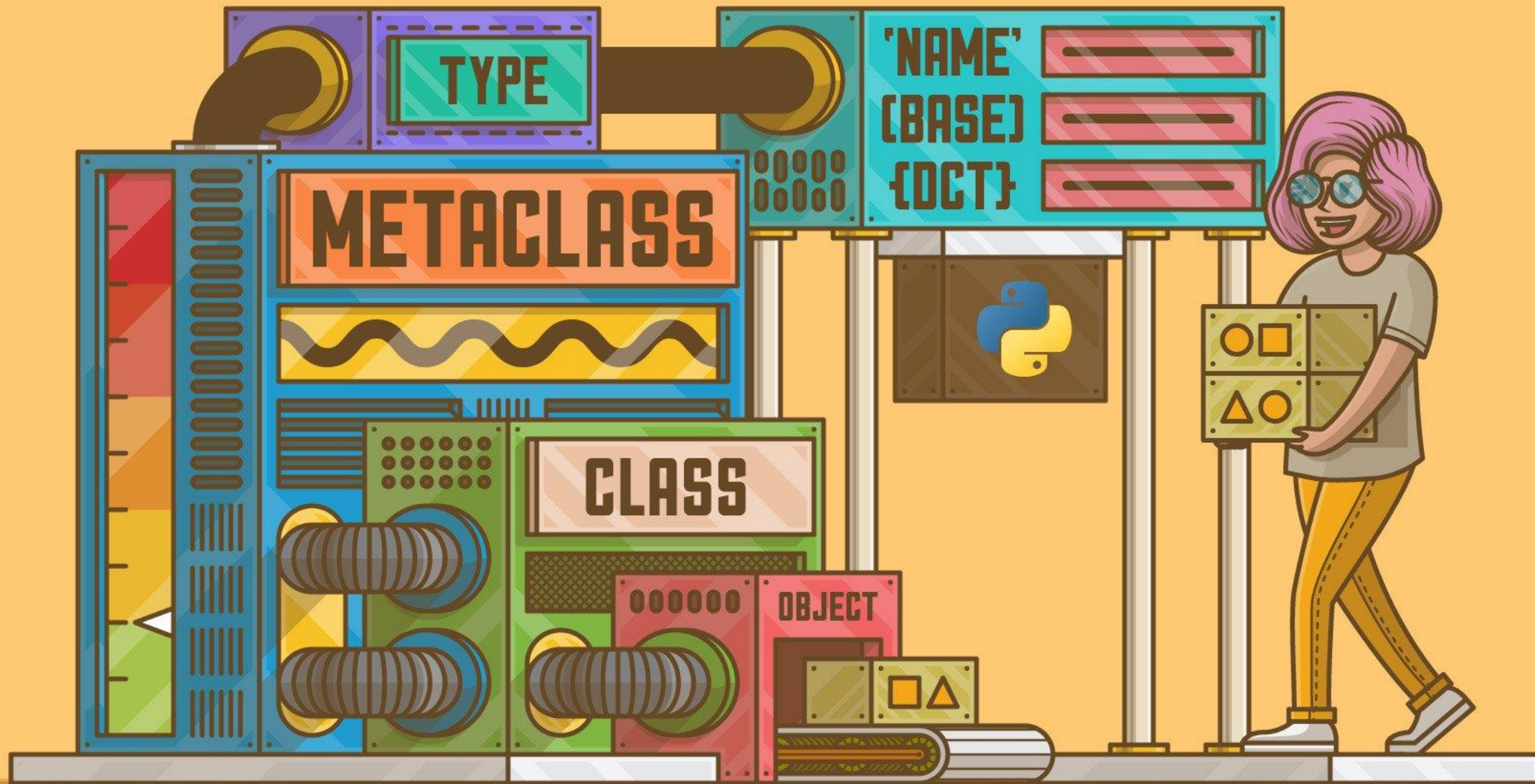
1. Гарантирует наличие единственного экземпляра класса.
2. Предоставляет глобальную точку доступа.

Аналогия из жизни

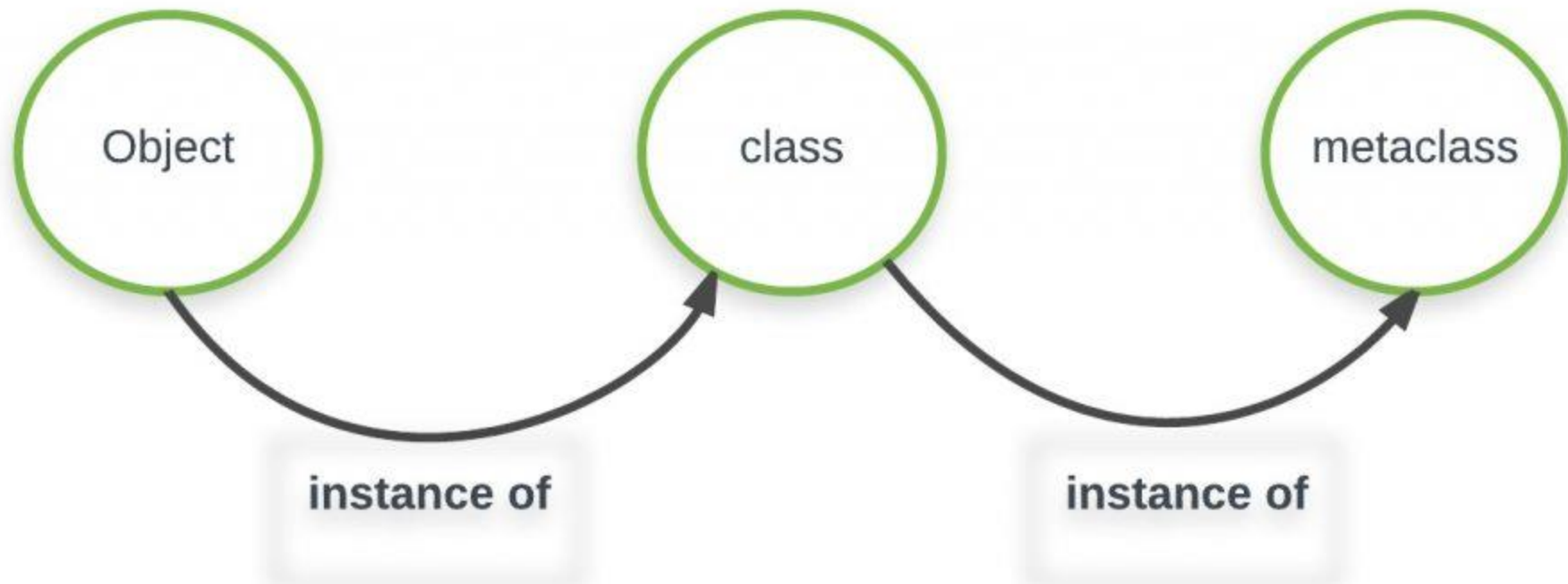
Правительство государства — хороший пример одиночки. В государстве может быть только одно официальное правительство. Вне зависимости от того, кто конкретно заседает в правительстве, оно имеет глобальную точку доступа «Правительство страны N».

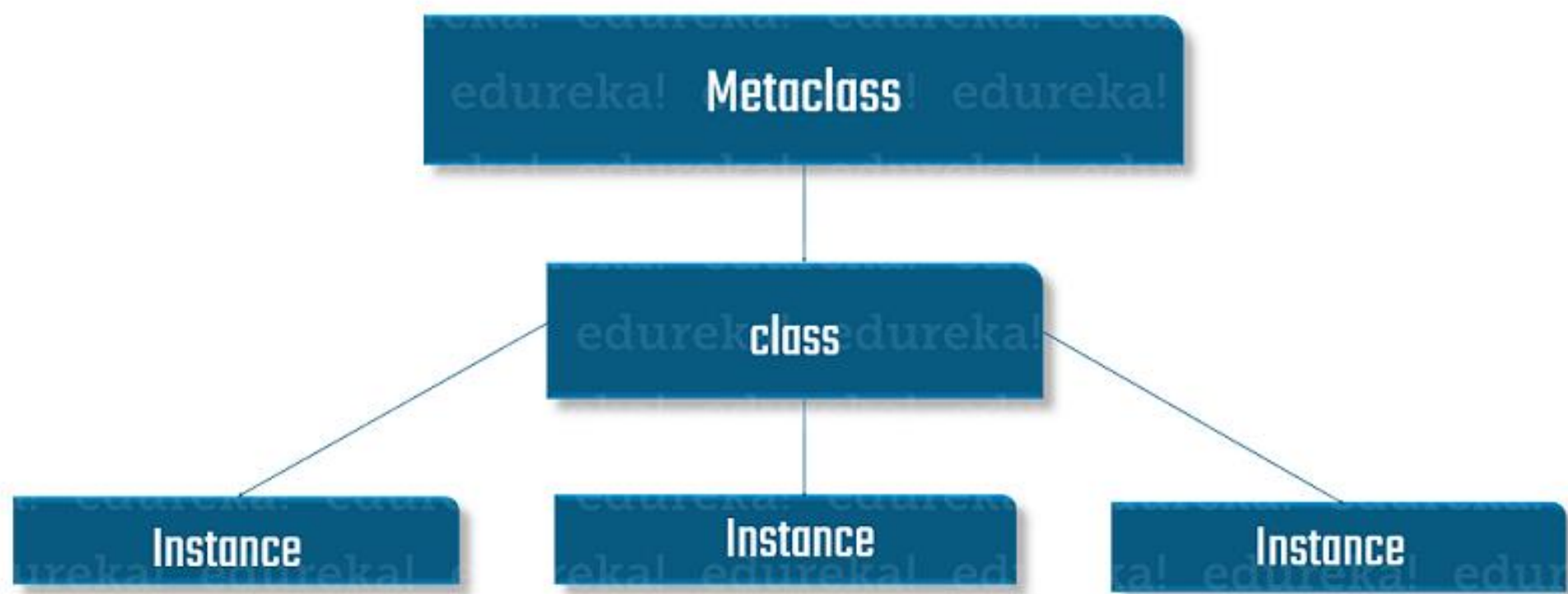
Структура





Real Python







```
1  class Singleton(type):
2      _instances = {}
3
4      def __call__(self, *args, **kwargs):
5          if self not in self._instances:
6              self._instances[self] = super().__call__(*args, **kwargs)
7          return self._instances[self]
```