Паттерны Проектирования

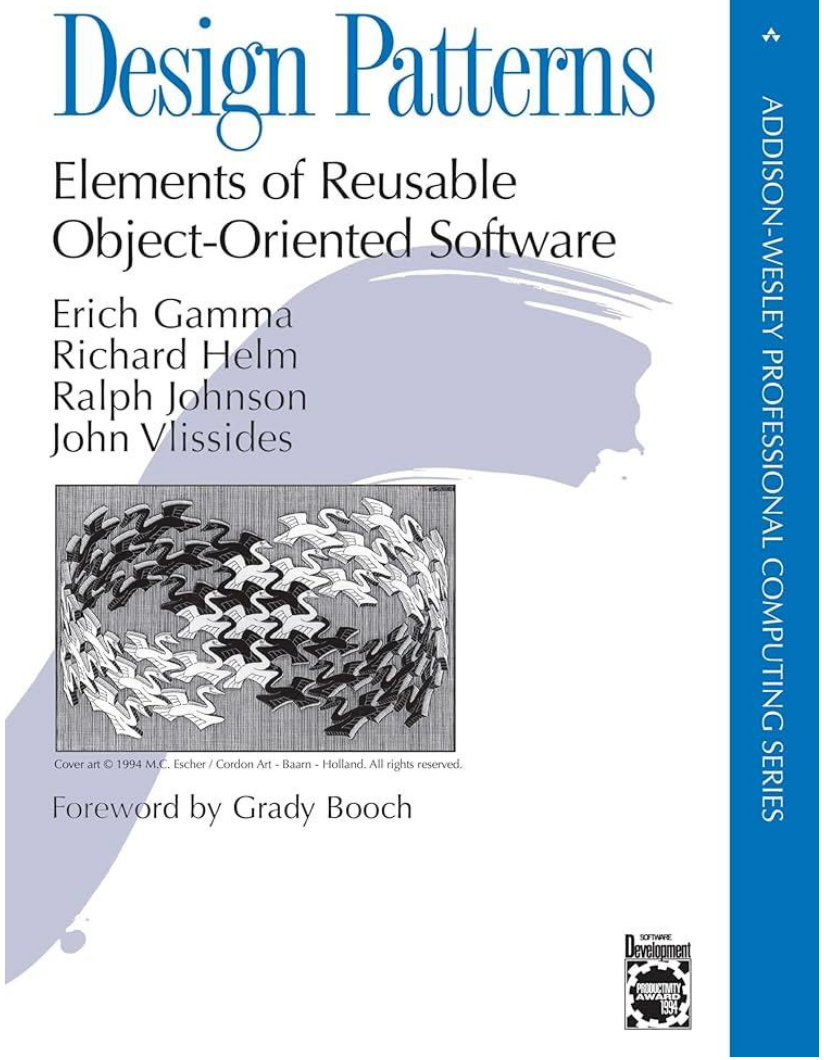
Design Patterns

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”



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

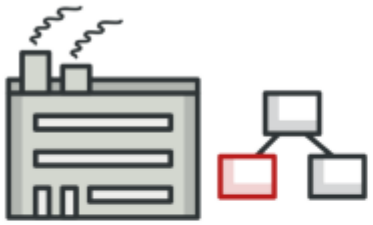
Классификация паттернов

Порождающие — как создавать объекты.

Структурные — как объединять классы и объекты.

Поведенческие — как объекты взаимодействуют между собой.

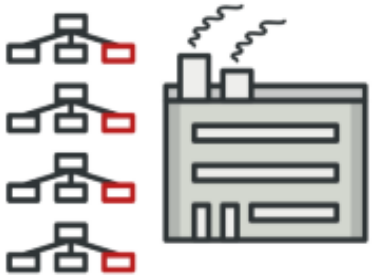
Порождающие паттерны



Фабричный метод

Factory Method

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



Абстрактная фабрика

Abstract Factory

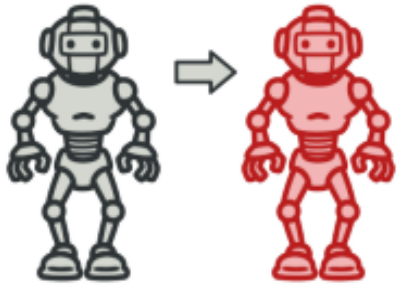
Позволяет создавать семейства связанных объектов, не привязываясь к конкретным классам создаваемых объектов.

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

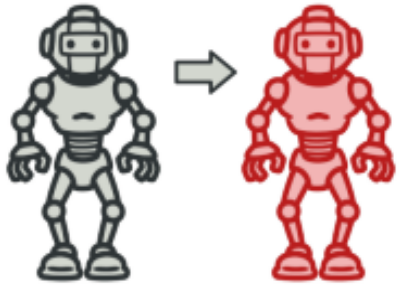
Позволяет копировать объекты, не вдаваясь в подробности их реализации.

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

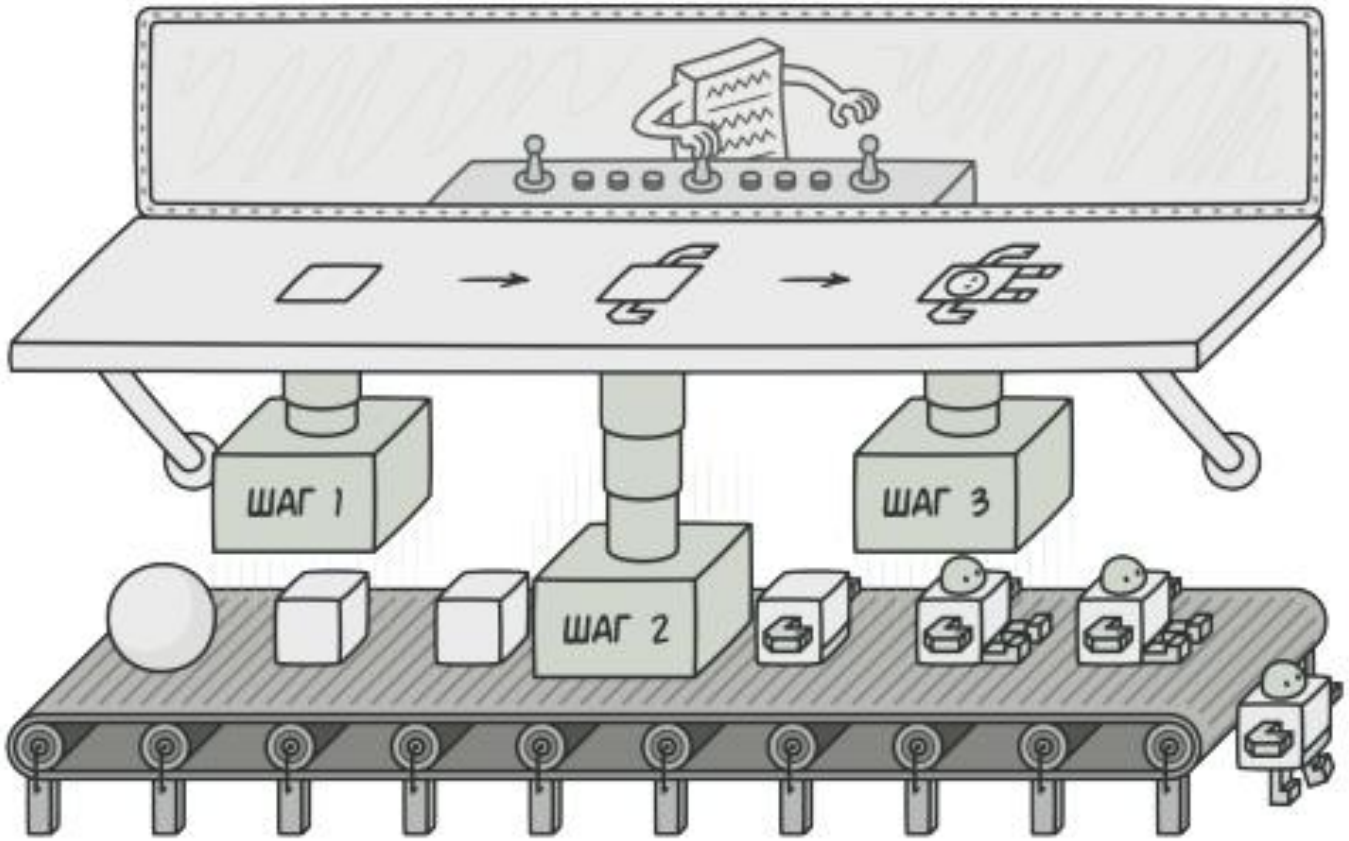
Позволяет копировать объекты, не вдаваясь в подробности их реализации.

Порождающие паттерны

Одиночка

Singleton

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

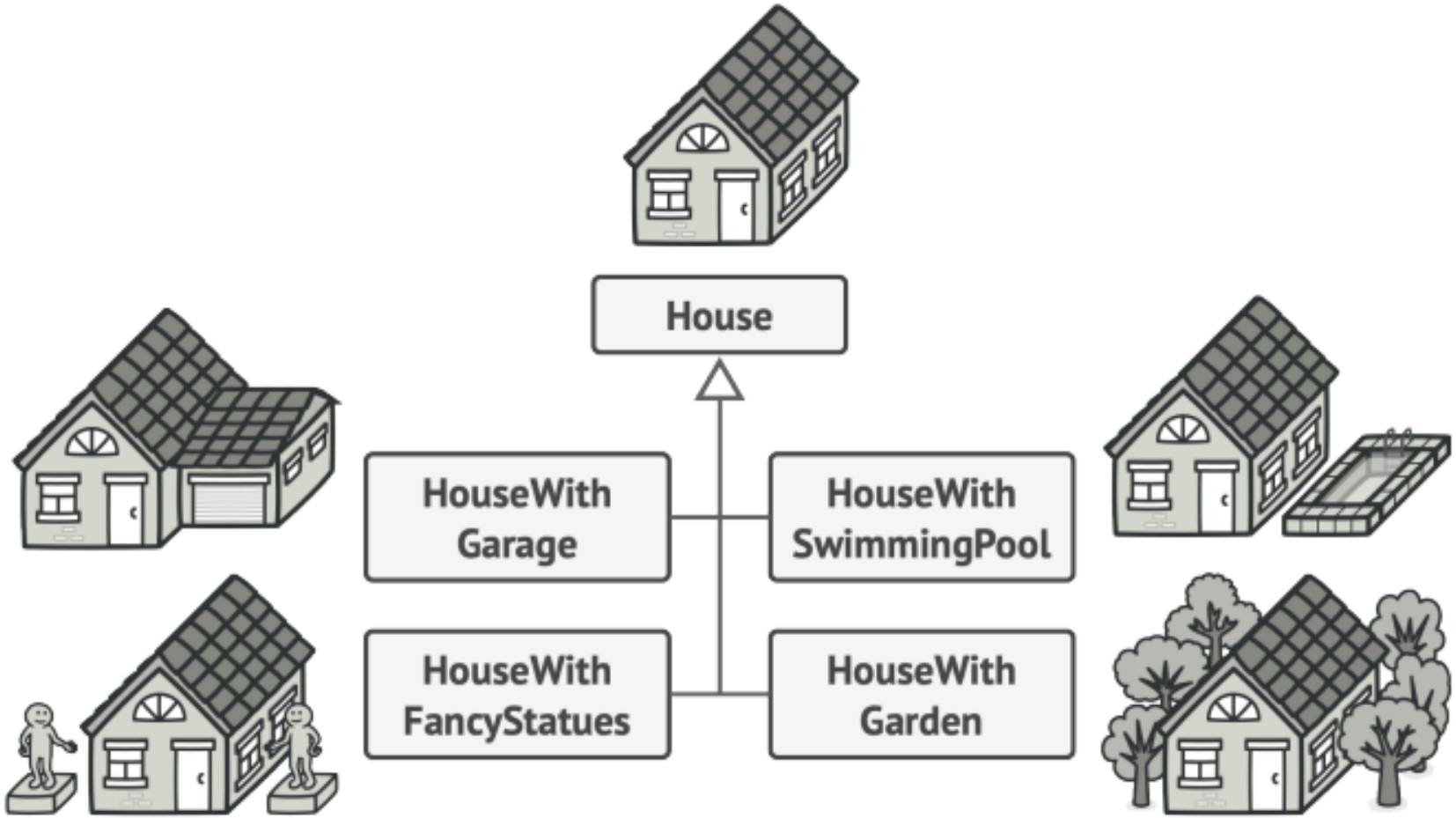


СТРОИТЕЛЬ

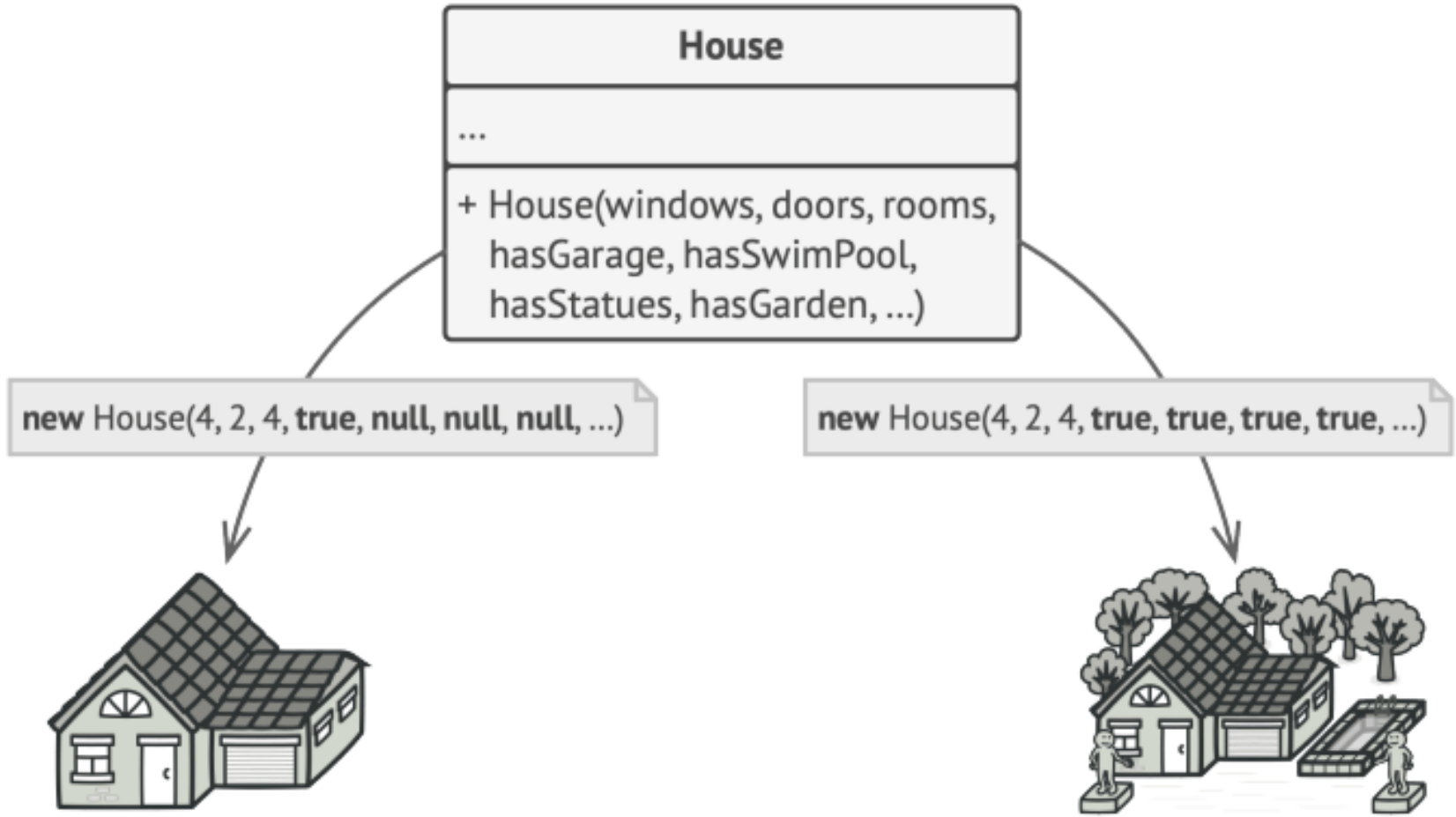
Также известен как: Builder

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

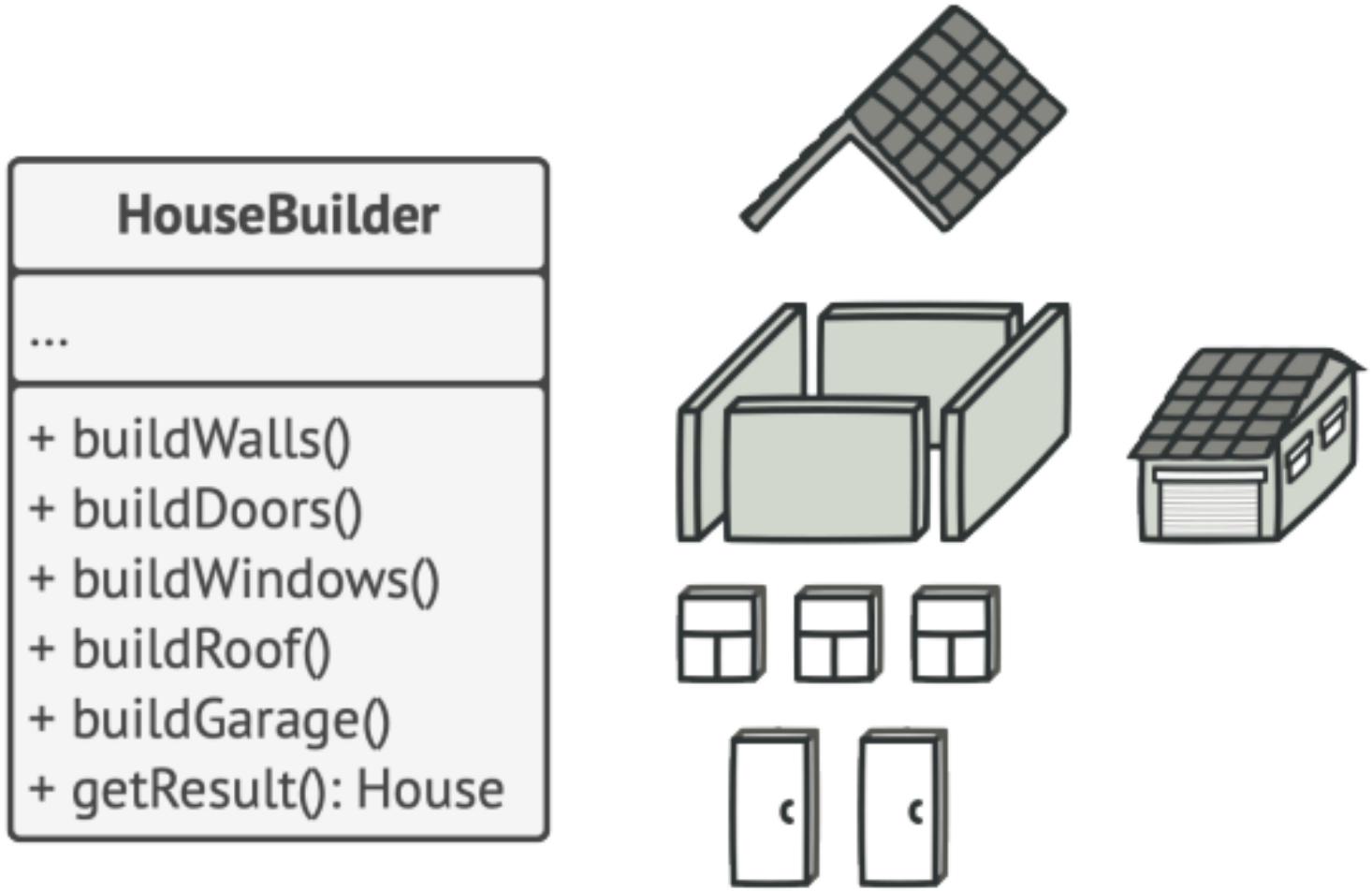
Представьте сложный объект, требующий кропотливой пошаговой инициализации множества полей и вложенных объектов. Код инициализации таких объектов обычно спрятан внутри монструозного конструктора с десятком параметров. Либо ещё хуже — распылён по всему клиентскому коду. .



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.



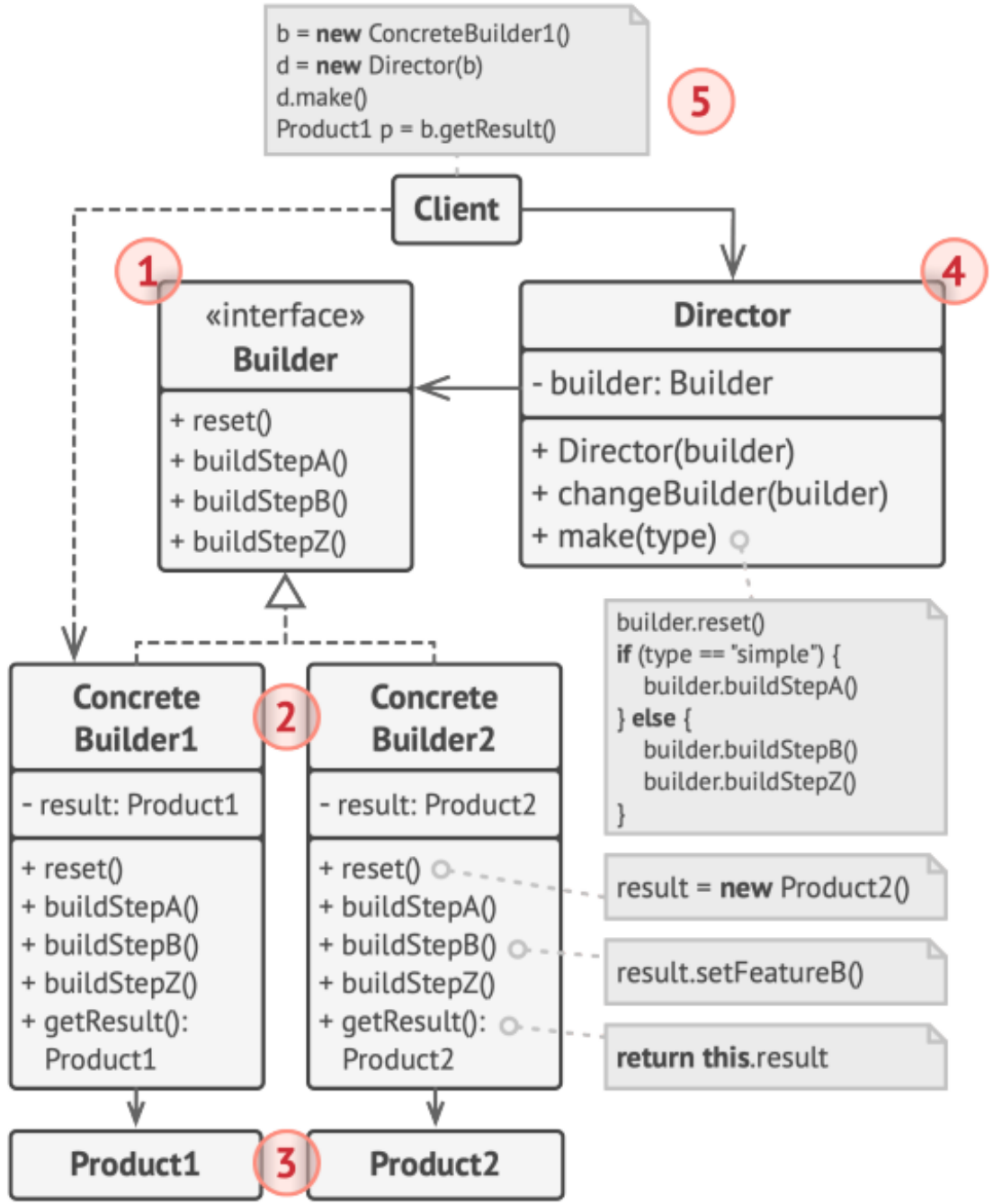
Строитель позволяет создавать сложные объекты пошагово.

Промежуточный результат всегда остаётся защищён.

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

Директор

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


b = new ConcreteBuilder1()
d = new Director(b)
d.make()
Product1 p = b.getResult()
5
Client
1
«interface»
Builder
+ reset()
+ buildStepA()
+ buildStepB()
+ buildStepZ()
4
Director
- builder: Builder
+ Director(builder)
+ changeBuilder(builder)
+ make(type)
builder.reset()
if (type == "simple") {
builder.buildStepA()
} else {
builder.buildStepB()
builder.buildStepZ()
}
Concrete
Builder1
2
- result: Product1
+ reset()
+ buildStepA()
+ buildStepB()
+ buildStepZ()
+ getResult():
Product1
Concrete
Builder2
- result: Product2
+ reset()
+ buildStepA()
+ buildStepB()
+ buildStepZ()
+ getResult():
Product2
result = new Product2()
result.setFeatureB()
return this.result
Product1
3
Product2

ФАБРИЧНЫЙ МЕТОД

Также известен как: Виртуальный конструктор, Factory Method

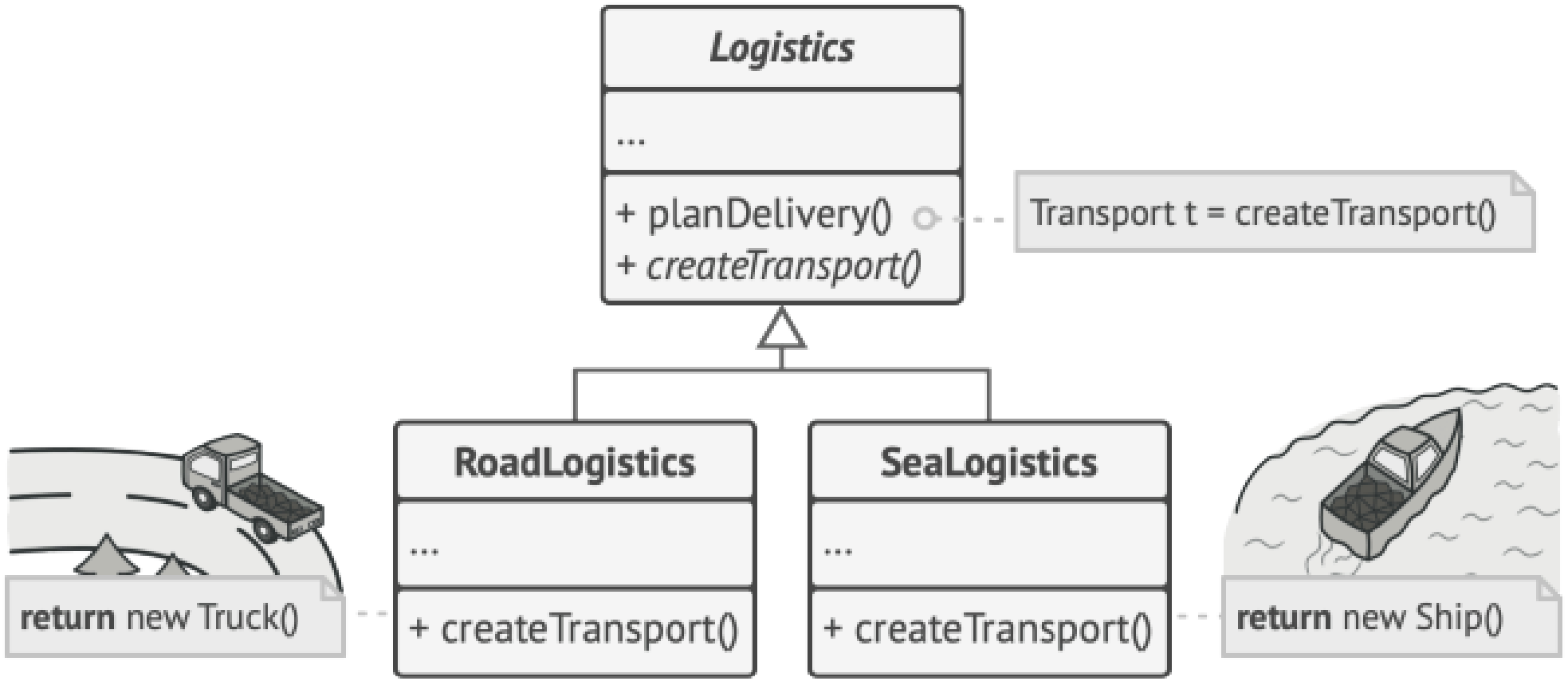
- Фабричный метод — это порождающий паттерн проектирования, который определяет общий интерфейс для создания объектов в суперклассе, позволяя подклассам изменять тип создаваемых объектов.

Проблема

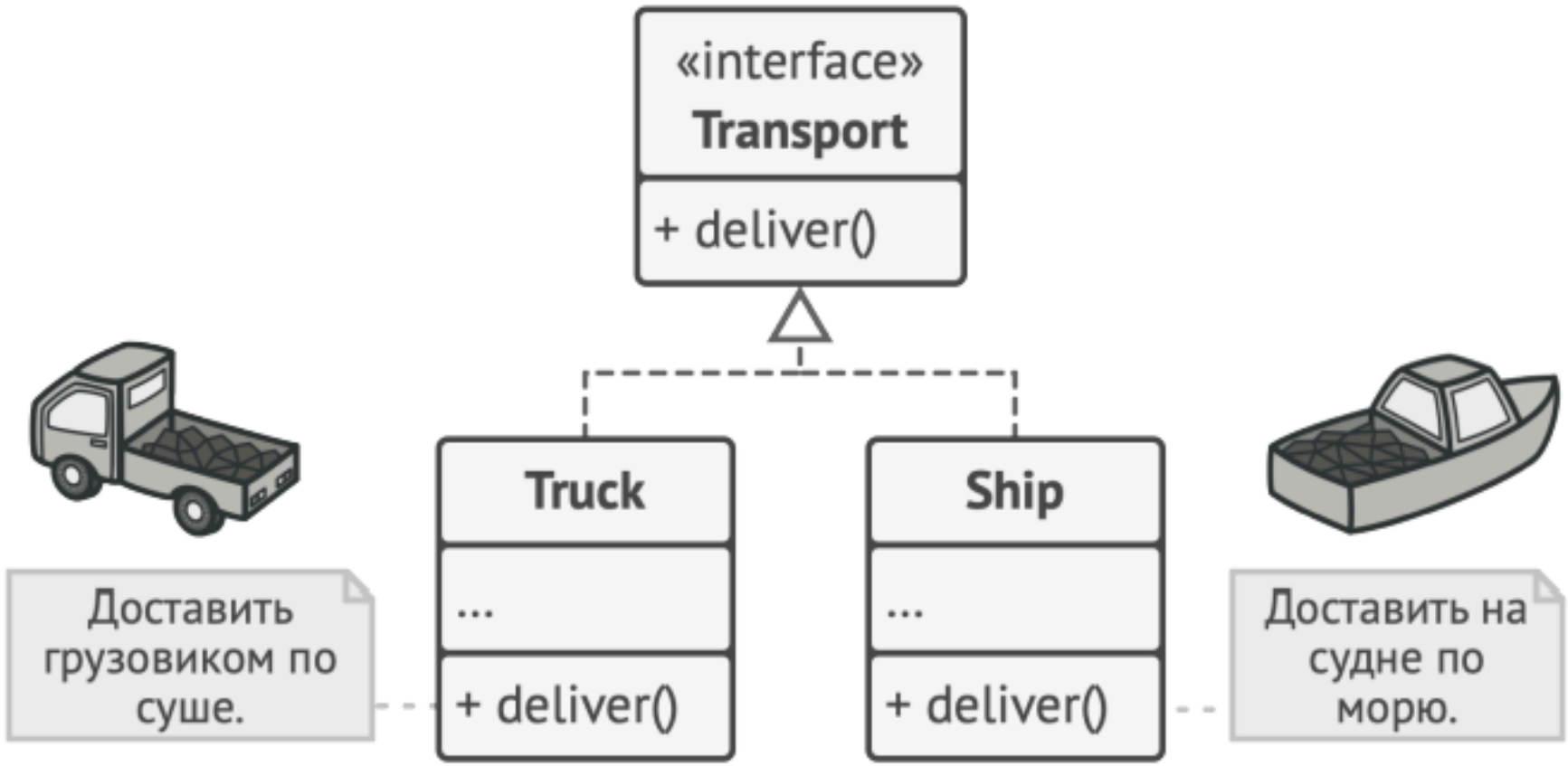
- Представьте, что вы создаёте программу управления грузовыми перевозками. Сперва вы рассчитываете перевозить товары только на автомобилях. Поэтому весь ваш код работает с объектами класса Грузовик .

- В какой-то момент ваша программа становится настолько известной, что морские перевозчики выстраиваются в очередь и просят добавить поддержку морской логистики в программу.

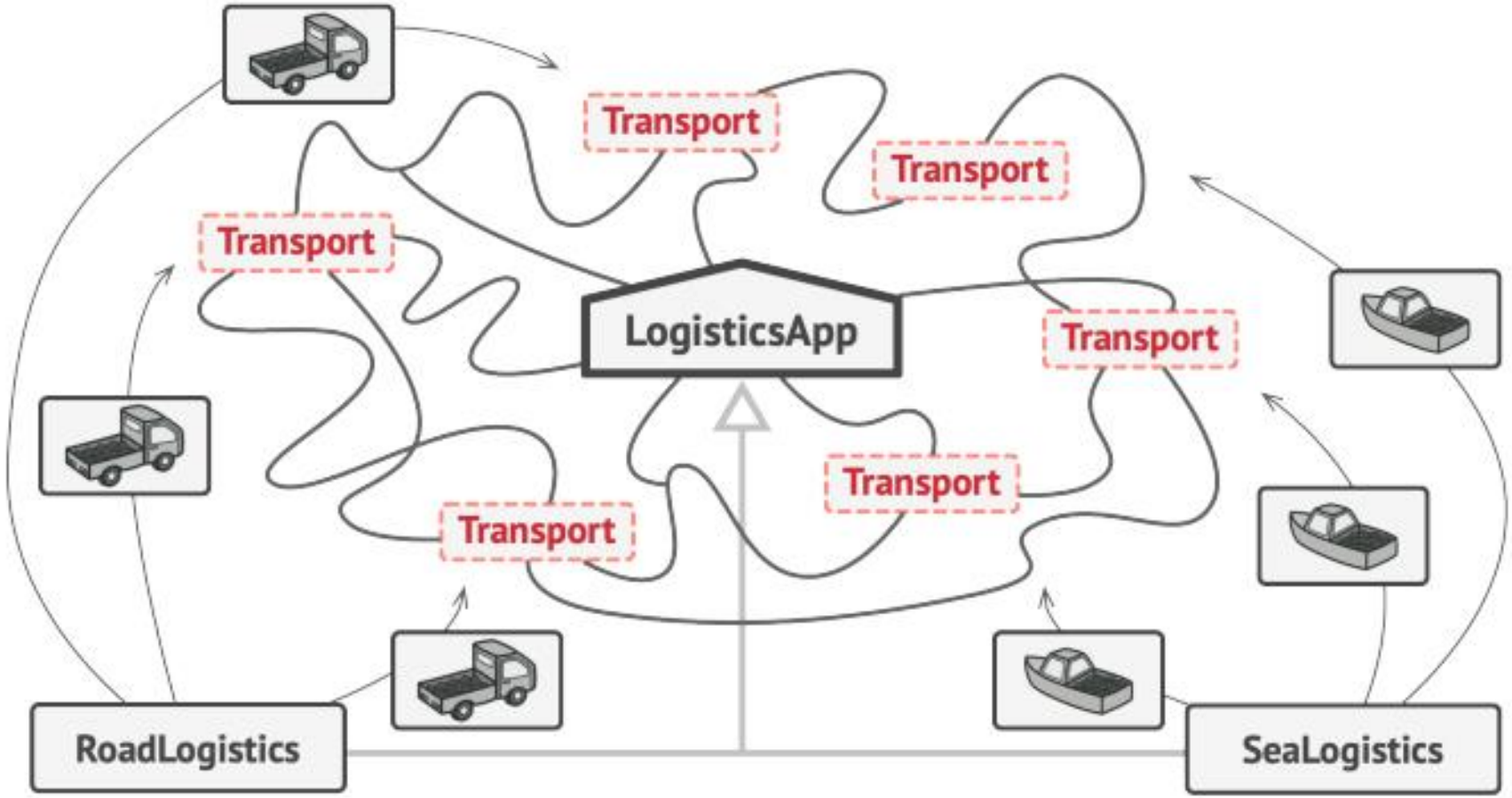
Добавить новый класс не так-то просто, если весь код уже завязан на конкретные классы.



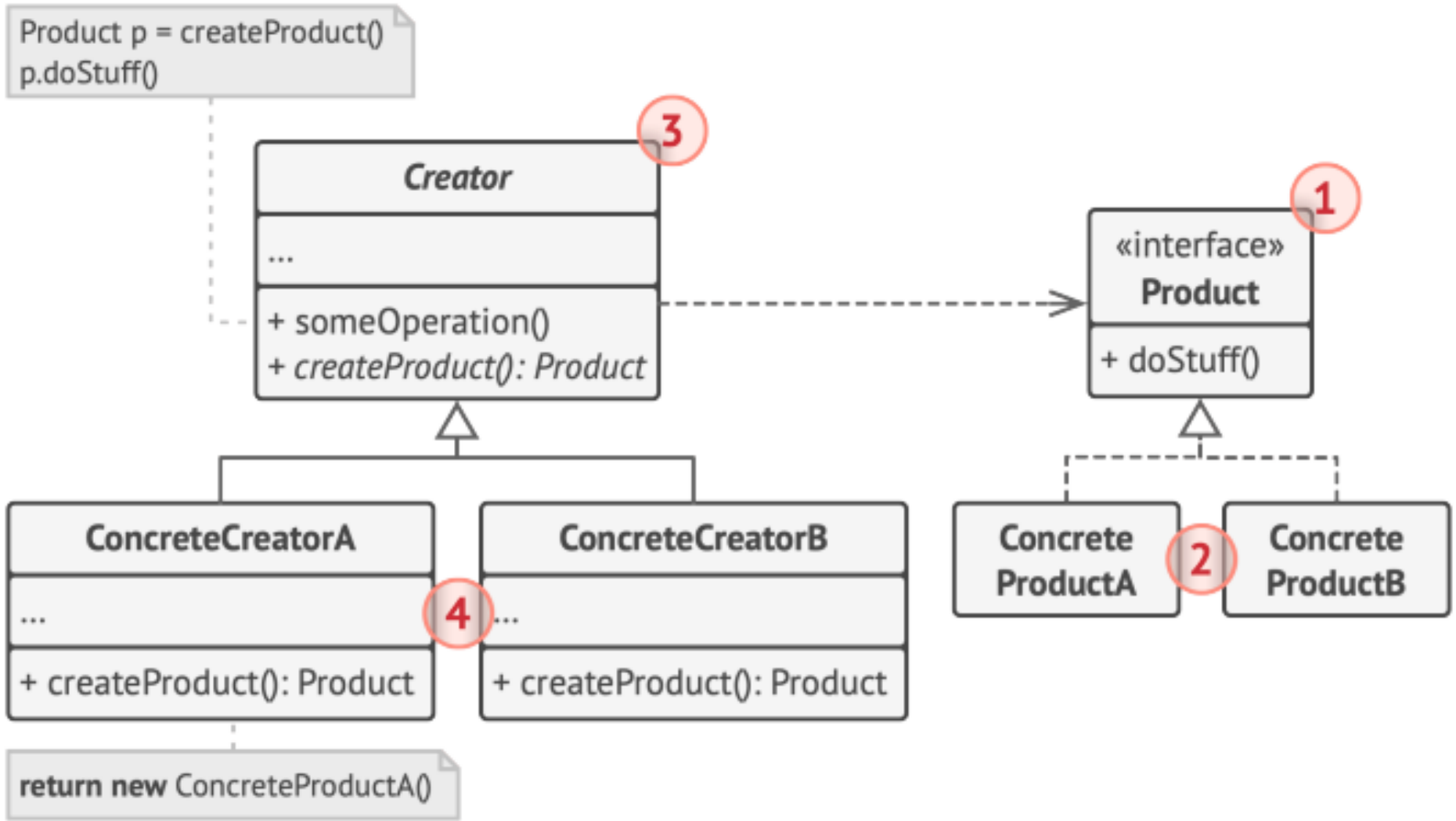
Подклассы могут изменять класс создаваемых объектов.

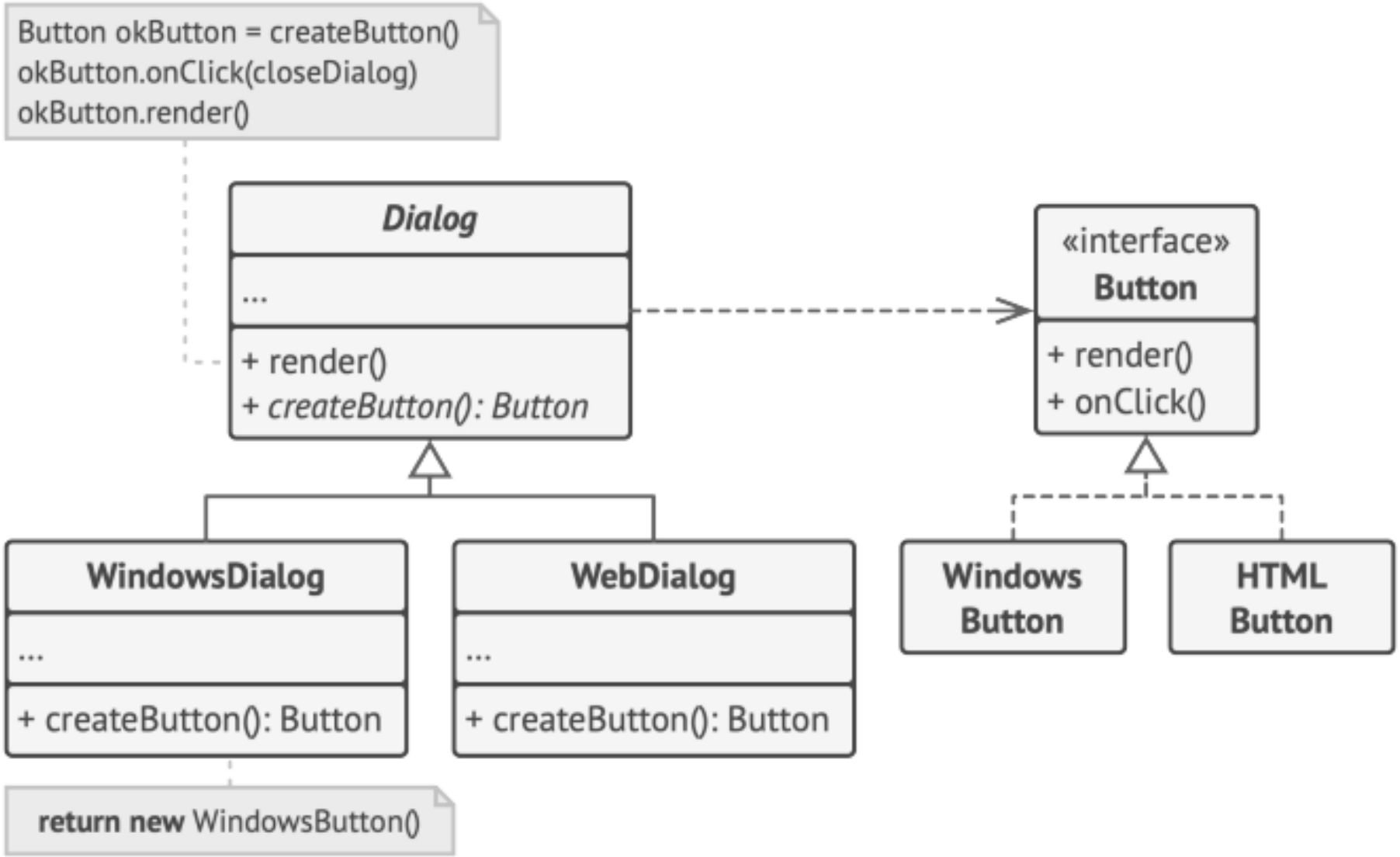


Все объекты-продукты должны иметь общий интерфейс.



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


Product p = createProduct()
p.doStuff()
Creator
...
+ someOperation()
+ createProduct(): Product
3
«interface»
Product
+ doStuff()
1
ConcreteCreatorA
...
+ createProduct(): Product
ConcreteCreatorB
...
+ createProduct(): Product
4
Concrete ProductA
2
Concrete ProductB
return new ConcreteProductA()



Пример кросс-платформенного диалога.