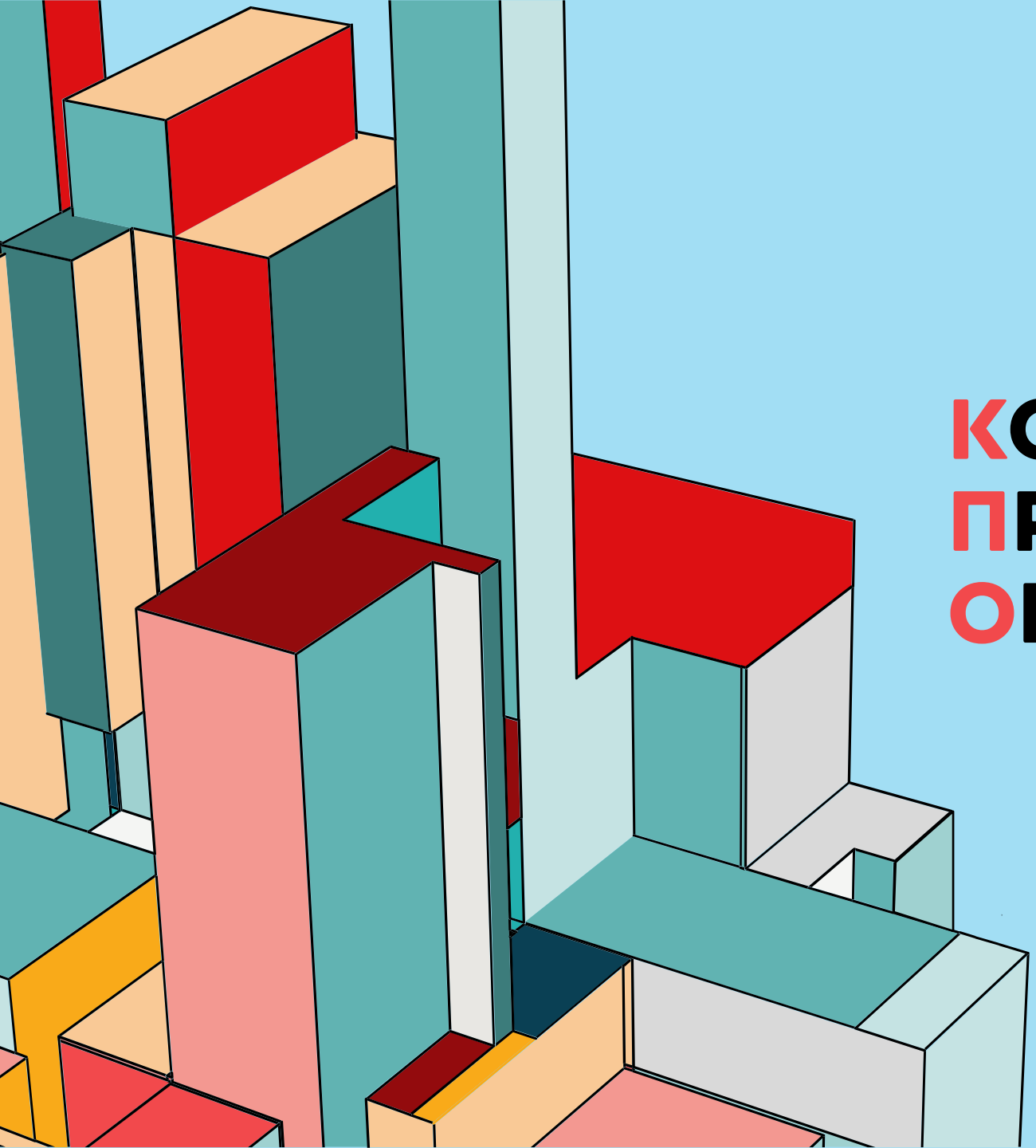
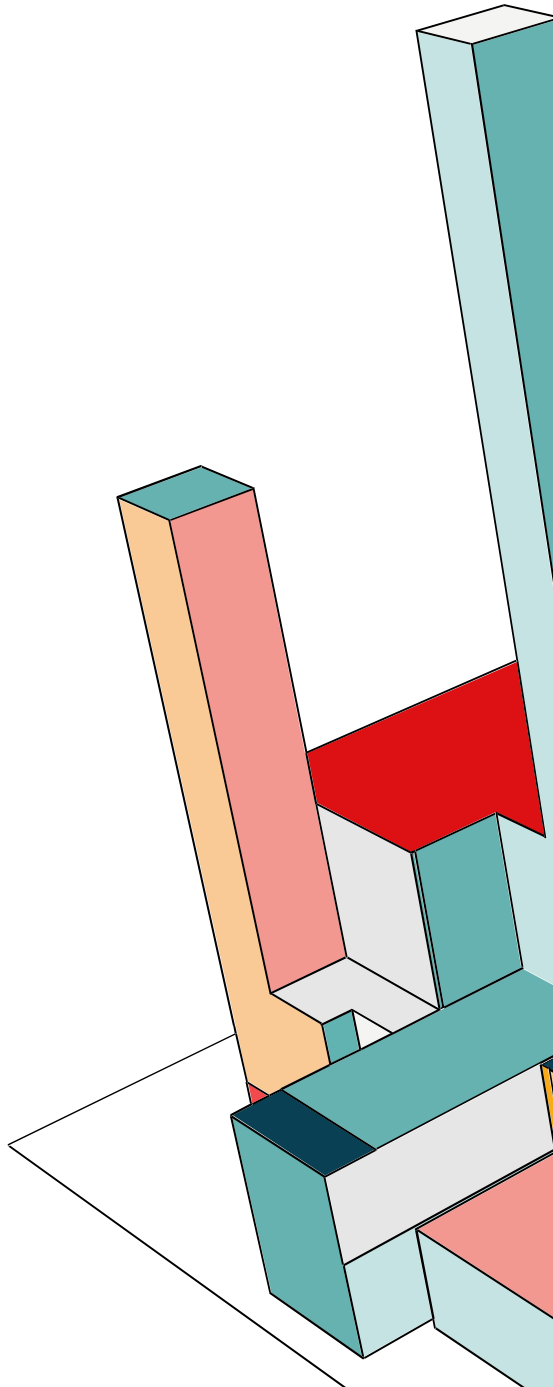


КОНСТРУИРОВАНИЕ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

ПЛАН ЛЕКЦИИ № 5

1. Что такое Паттерн?
2. Классификация паттернов
3. Порождающие паттерны проектирования
   - фабричный метод
   - абстрактная фабрика
   - строитель
   - прототип
   - одиночка

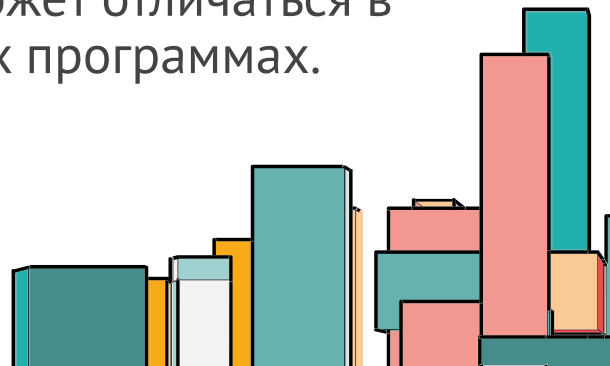


ЧТО ТАКОЕ ПАТТЕРН

Паттерн проектирования — это часто встречающееся решение определённой проблемы при проектировании архитектуры программ.

Не путать с алгоритмами!

Алгоритм — это чёткий набор действий, а паттерн — это описание решения, реализация которого может отличаться в двух разных программах.



ИСТОРИЯ ПАТТЕРНОВ

Концепцию паттернов впервые описал Кристофер Александер в книге «Язык шаблонов. Города. Здания. Строительство».

Идея показалась заманчивой четвёрке авторов и в 1994 году они написали книгу «Приемы объектно-ориентированного проектирования.

ЗАЧЕМ ЗНАТЬ ПАТТЕРНЫ?

- **Проверенные решения.** Вы тратите меньше времени, используя готовые решения, вместо повторного изобретения велосипеда. До некоторых решений вы смогли бы додуматься и сами, но многие могут быть для вас открытием.

- **Стандартизация кода.** Вы делаете меньше просчётов при проектировании, используя типовые унифицированные решения, так как все скрытые проблемы в них уже давно найдены.

- **Общий программистский словарь.** Вы произносите название паттерна, вместо того, чтобы час объяснять другим программистам, какой крутой дизайн вы придумали и какие классы для этого нужны.

КРИТИКА ПАТТЕРНОВ

- **Костыли** для слабого языка программирования

  стратегия = анонимная (лямбда) функция

- **Неэффективные** решения

  реализуют паттерны «как в книжке»

- **Неоправданное** применение

  Если у тебя в руках молоток, то все предметы вокруг начинают напоминать гвозди.

ВИДЫ ПАТТЕРНОВ

- **Порождающие** паттерны беспокоятся о гибком создании объектов без внесения в программу лишних зависимостей.

- **Структурные** паттерны показывают различные способы построения связей между объектами.

- **Поведенческие** паттерны заботятся об эффективной коммуникации между объектами.

СПИСОК ШАБЛОНОВ

- C Абстрактная фабрика
- S Адаптер
- S Мост
- C Строитель
- B Цепочка обязанностей
- B Команда
- S Компоновщик
- S Декоратор
- S Фасад
- C Фабричный метод
- S Приспособленец
- B Интерпретатор
- B Итератор
- B Посредник
- B Хранитель
- C Прототип
- S Прокси
- B Наблюдатель
- C Одиночка
- B Состояние
- B Стратегия
- B Шаблонный метод
- B Посетитель

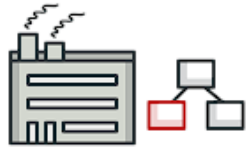
- B — поведенческие (behavioral);
- C — порождающие (creational);
- S — структурные (structural).

СПИСОК ПОРОЖДАЮЩИХ ПАТТЕРНОВ ПРОЕКТИРОВАНИЯ

Отвечают за удобное и безопасное создание новых объектов.

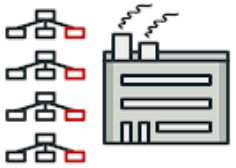
Одиночка
Singleton

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



Фабричный метод
Factory Method

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

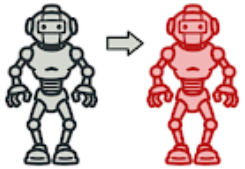


Абстрактная фабрика
Abstract Factory

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

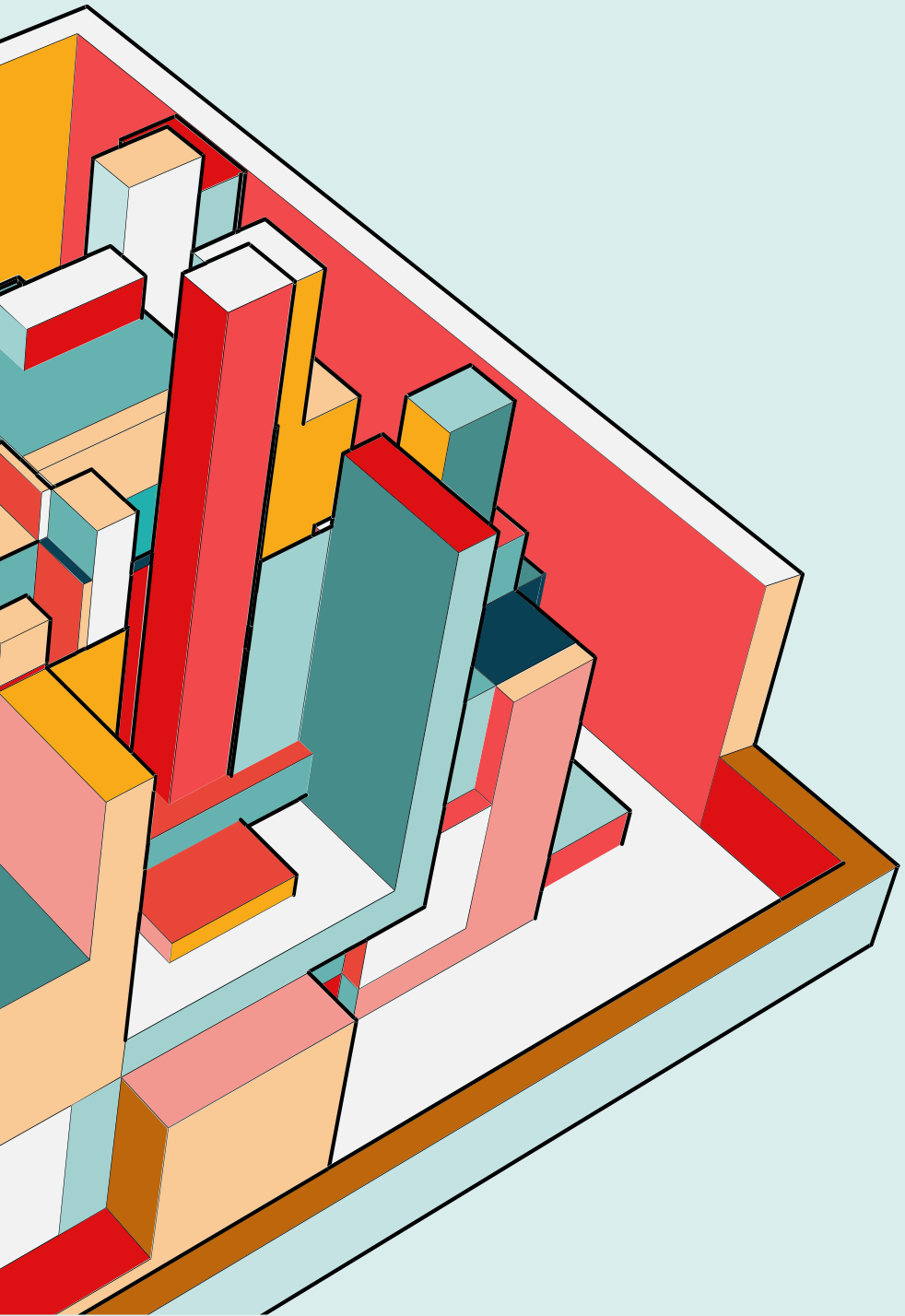
Строитель
Builder

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 ПРОЕКТИРОВАНИЯ

Factory Method

ФАБРИЧНЫЙ МЕТОД

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

Сперва вы рассчитываете перевозить товары только на автомобилях.

В какой-то момент вас просят добавить поддержку морской логистики в программу.

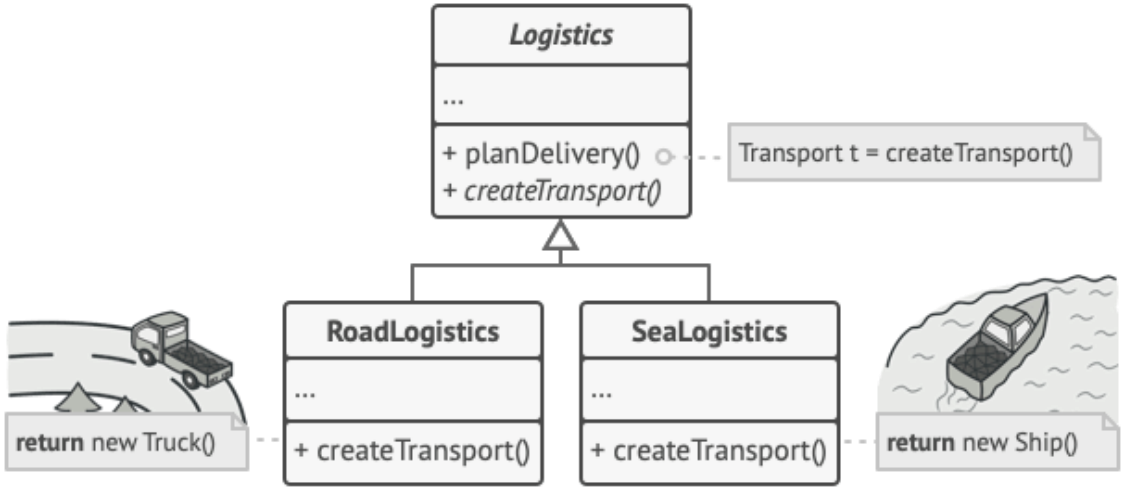
Пишем ужасающий код, наполненный условными операторами, которые выполняют действие, в зависимости от класса транспорта.

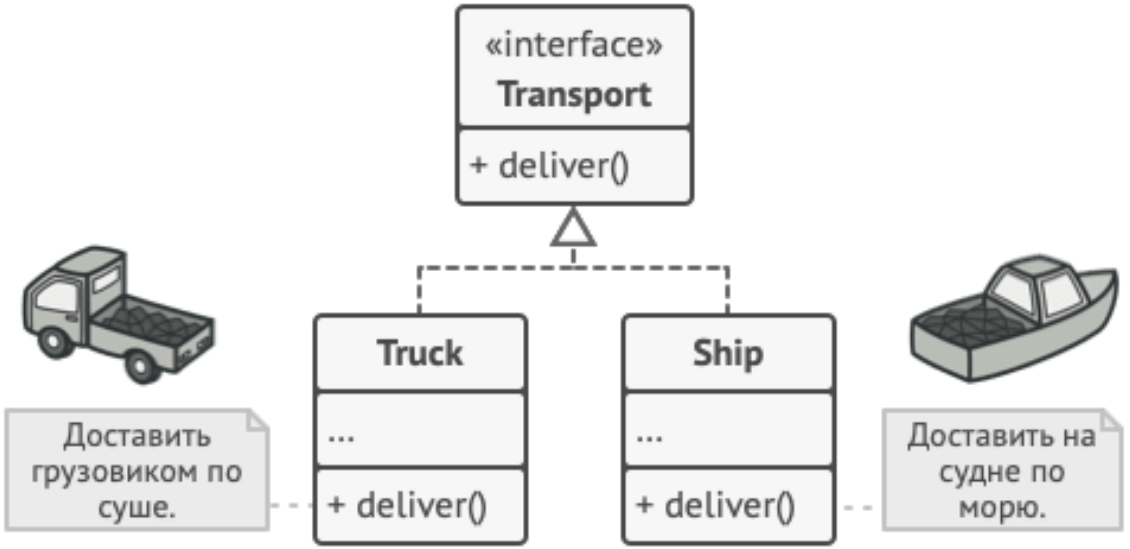
Нужно создавать объекты не напрямую, используя оператор *new*, а через вызов особого фабричного метода.

Не пугайтесь, объекты всё равно будут создаваться при помощи new, но делать это будет фабричный метод.

РЕШЕНИЕ

Мы просто переместили вызов конструктора
из одного конца программы в другой

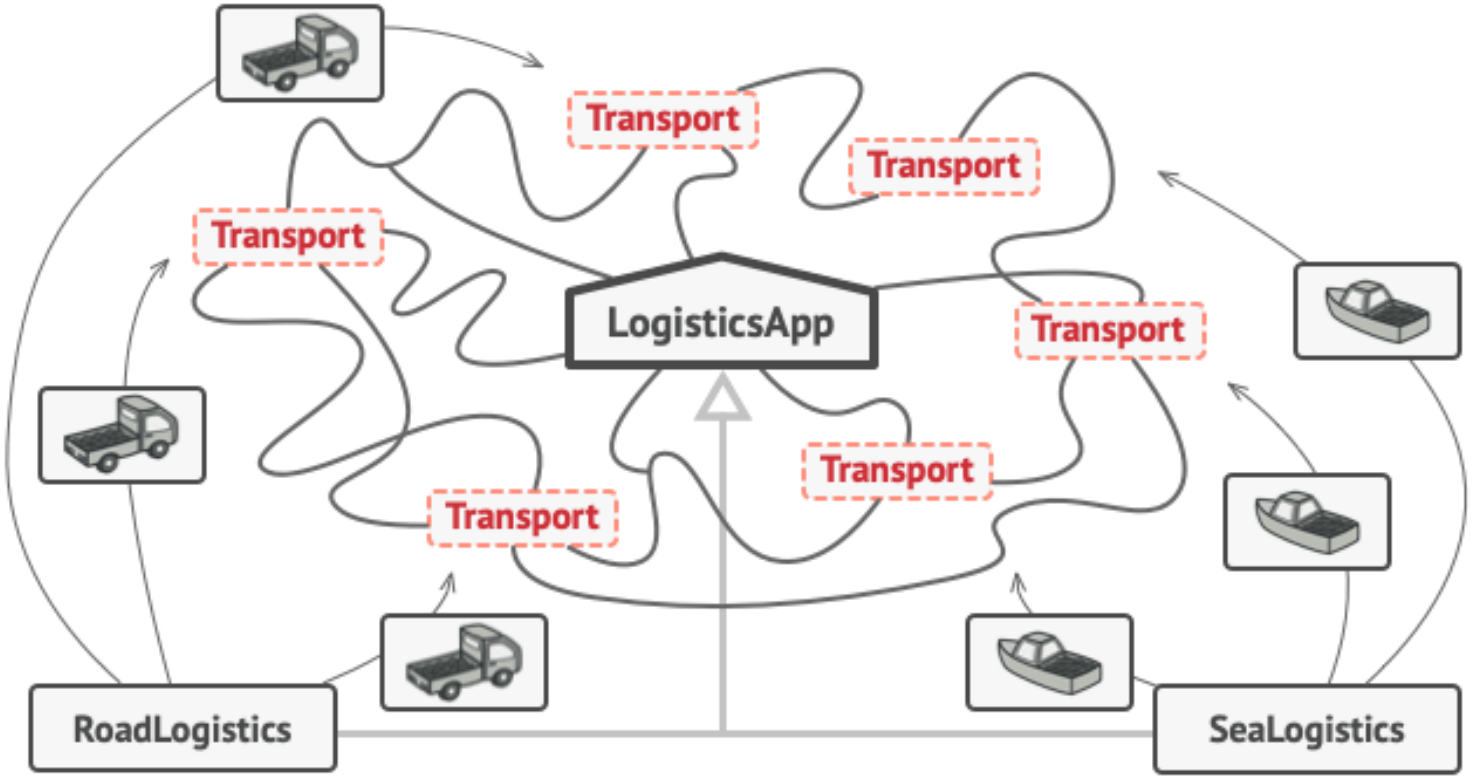




Чтобы эта система заработала, все возвращаемые объекты должны иметь общий интерфейс.

ПОКА ВСЕ ПРОДУКТЫ РЕАЛИЗУЮТ ОБЩИЙ ИНТЕРФЕЙС, ИХ ОБЪЕКТЫ МОЖНО ВЗАИМОЗАМЕНЯТЬ В КЛИЕНТСКОМ КОДЕ.

Для клиента фабричного метода нет разницы между этими объектами, так как он будет трактовать их как некий абстрактный Транспорт.



СТРУКТУРА

1. **Продукт** определяет общий интерфейс.

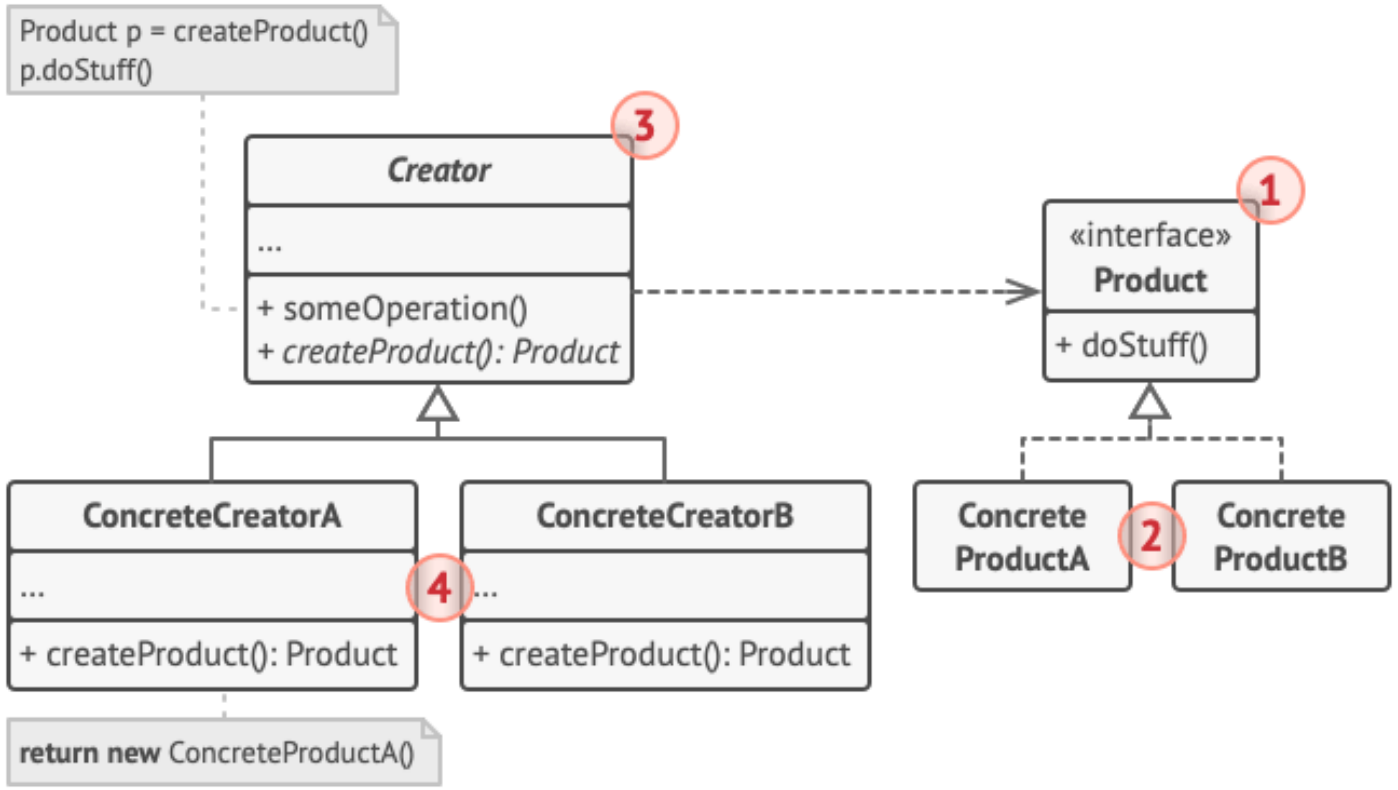
2. **Конкретные продукты** содержат код различных продуктов.

3. **Создатель** объявляет *фабричный метод*, который должен возвращать новые объекты продуктов.

   Зачастую **фабричный метод объявляют абстрактным**, чтобы заставить все подклассы реализовать его по-своему.

   Несмотря на название, важно понимать, что создание продуктов **не является** его единственной функцией.

4. **Конкретные создатели** по-своему реализуют фабричный метод.



ФОРМАЛЬНОЕ ОПРЕДЕЛЕНИЕ ПАТТЕРНА

```
abstract class Product {}

class ConcreteProductA : Product
{}

class ConcreteProductB : Product
{}


abstract class Creator
{
    public abstract Product FactoryMethod();
}


class ConcreteCreatorA : Creator
{
    public override Product FactoryMethod() { return new ConcreteProductA(); }
}

class ConcreteCreatorB : Creator
{
    public override Product FactoryMethod() { return new ConcreteProductB(); }
}
```

РАССМОТРИМ НА ПРИМЕРЕ

Допустим, мы создаем программу для сферы строительства.

Возможно, вначале мы захотим построить многоэтажный панельный дом. И для этого выбирается соответствующий подрядчик, который возводит каменные дома.

Затем нам захочется построить деревянный дом и для этого также надо будет выбрать нужного подрядчика.

Назовите недостаток паттерна на этом примере.

```
// абстрактный класс строительной компании
abstract class Developer
{
    public string Name { get; set; }

    public Developer (string n)
    {
        Name = n;
    }
    // фабричный метод
    abstract public House Create();
}
// строит панельные дома
class PanelDeveloper : Developer
{
    public PanelDeveloper(string n) : base(n)
    { }

    public override House Create()
    {
        return new PanelHouse();
    }
}
// строит деревянные дома
class WoodDeveloper : Developer
{
    public WoodDeveloper(string n) : base(n)
    { }

    public override House Create()
    {
        return new WoodHouse();
    }
}

abstract class House
{ }

class PanelHouse : House
{
    public PanelHouse()
    {
        Console.WriteLine("Панельный дом построен");
    }
}
class WoodHouse : House
{
    public WoodHouse()
    {
        Console.WriteLine("Деревянный дом построен");
    }
}
```

```
// абстрактный класс строительной компании

abstract class Developer
{
    public string Name { get; set; }

    public Developer (string n)
    {
        Name = n;
    }

    // фабричный метод
    abstract public House Create();
}
```

```
// абстрактный класс строительной компании
abstract class Developer
{
    public string Name { get; set; }

    public Developer (string n)
    {
        Name = n;
    }
    // фабричный метод
    abstract public House Create();
}
// строит панельные дома
class PanelDeveloper : Developer
{
    public PanelDeveloper(string n) : base(n)
    { }

    public override House Create()
    {
        return new PanelHouse();
    }
}
// строит деревянные дома
class WoodDeveloper : Developer
{
    public WoodDeveloper(string n) : base(n)
    { }

    public override House Create()
    {
        return new WoodHouse();
    }
}

abstract class House
{ }

class PanelHouse : House
{
    public PanelHouse()
    {
        Console.WriteLine("Панельный дом построен");
    }
}
class WoodHouse : House
{
    public WoodHouse()
    {
        Console.WriteLine("Деревянный дом построен");
    }
}
```

```
// строит панельные дома
class PanelDeveloper : Developer
{
    public PanelDeveloper(string n) : base(n)
    { }

    public override House Create()
    {
        return new PanelHouse();
    }
}

// строит деревянные дома
class WoodDeveloper : Developer
{
    public WoodDeveloper(string n) : base(n)
    { }

    public override House Create()
    {
        return new WoodHouse();
    }
}
```

```
// абстрактный класс строительной компании
abstract class Developer
{
    public string Name { get; set; }

    public Developer (string n)
    {
        Name = n;
    }
    // фабричный метод
    abstract public House Create();
}
// строит панельные дома
class PanelDeveloper : Developer
{
    public PanelDeveloper(string n) : base(n)
    { }

    public override House Create()
    {
        return new PanelHouse();
    }
}
// строит деревянные дома
class WoodDeveloper : Developer
{
    public WoodDeveloper(string n) : base(n)
    { }

    public override House Create()
    {
        return new WoodHouse();
    }
}

abstract class House
{ }

class PanelHouse : House
{
    public PanelHouse()
    {
        Console.WriteLine("Панельный дом построен");
    }
}
class WoodHouse : House
{
    public WoodHouse()
    {
        Console.WriteLine("Деревянный дом построен");
    }
}
```

```
abstract class House
{ }

class PanelHouse : House
{
    public PanelHouse()
    {
        Console.WriteLine("Панельный дом построен");
    }
}

class WoodHouse : House
{
    public WoodHouse()
    {
        Console.WriteLine("Деревянный дом построен");
    }
}
```

```
// абстрактный класс строительной компании
abstract class Developer
{
    public string Name { get; set; }

    public Developer (string n)
    {
        Name = n;
    }
    // фабричный метод
    abstract public House Create();
}
// строит панельные дома
class PanelDeveloper : Developer
{
    public PanelDeveloper(string n) : base(n)
    { }

    public override House Create()
    {
        return new PanelHouse();
    }
}
// строит деревянные дома
class WoodDeveloper : Developer
{
    public WoodDeveloper(string n) : base(n)
    { }

    public override House Create()
    {
        return new WoodHouse();
    }
}

abstract class House
{ }

class PanelHouse : House
{
    public PanelHouse()
    {
        Console.WriteLine("Панельный дом построен");
    }
}
class WoodHouse : House
{
    public WoodHouse()
    {
        Console.WriteLine("Деревянный дом построен");
    }
}
```

```
Developer dev = new PanelDeveloper("ООО КирпичСтрой");
House house2 = dev.Create();

dev = new WoodDeveloper("Частный застройщик");
House house = dev.Create();

Console.ReadLine();
```

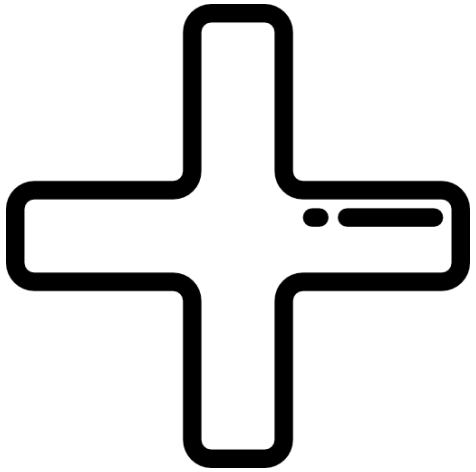
ПРИМЕНИМОСТЬ

1. Когда заранее неизвестны типы и зависимости объектов, с которыми должен работать ваш код.

2. Когда вы хотите дать возможность пользователям расширять части вашего фреймворка или библиотеки.

3. Когда вы хотите экономить системные ресурсы, повторно используя уже созданные объекты, вместо порождения новых.

ПРЕИМУЩЕСТВА И НЕДОСТАТКИ



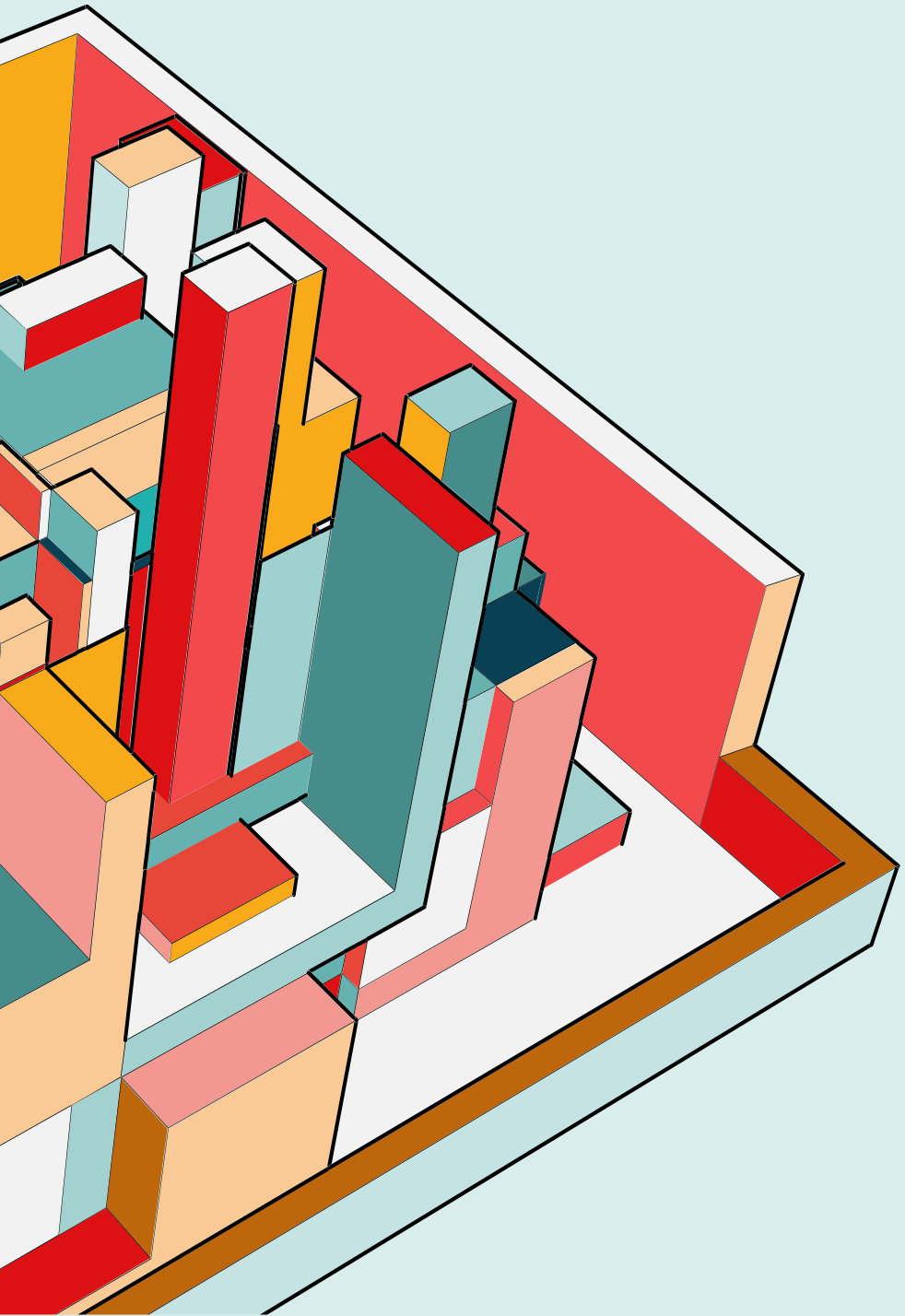
Избавляет класс от привязки к конкретным классам продуктов.

Выделяет код производства продуктов в одно место, упрощая поддержку кода.

Упрощает добавление новых продуктов в программу.

Реализует *принцип открытости/закрытости*.

Может привести к созданию больших параллельных иерархий классов, так как для каждого класса продукта надо создать свой подкласс создателя.



ПОРОЖДАЮЩИЕ ПАТТЕРНЫ ПРОЕКТИРОВАНИЯ

Abstract Factory

АБСТРАКТНАЯ ФАБРИКА

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

Ваш код содержит:

Семейство зависимых продуктов:
Кресло + Диван + Столик.

Продукты представлены в разных стилях:
Ар-деко, Викторианском и Модерне..

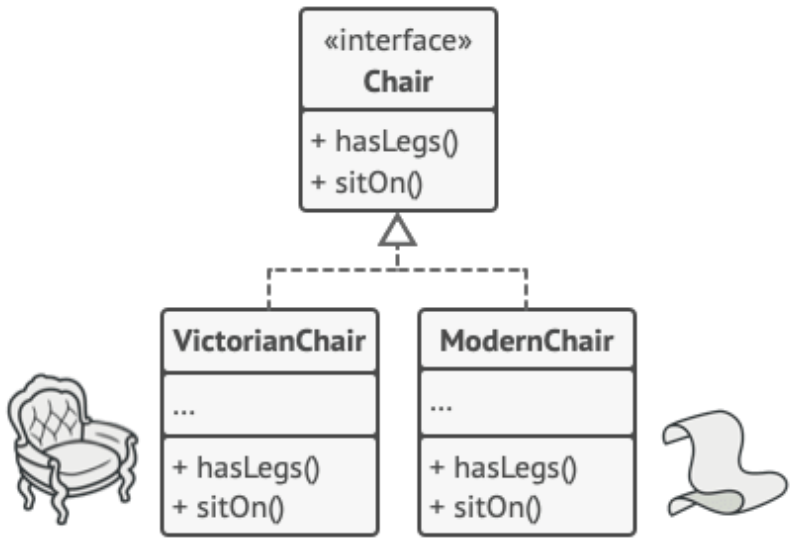
клиенты расстраиваются, если получают несочетающуюся мебель и вы не хотите вносить изменения в существующий код при добавлении новых продуктов или семейств в программу

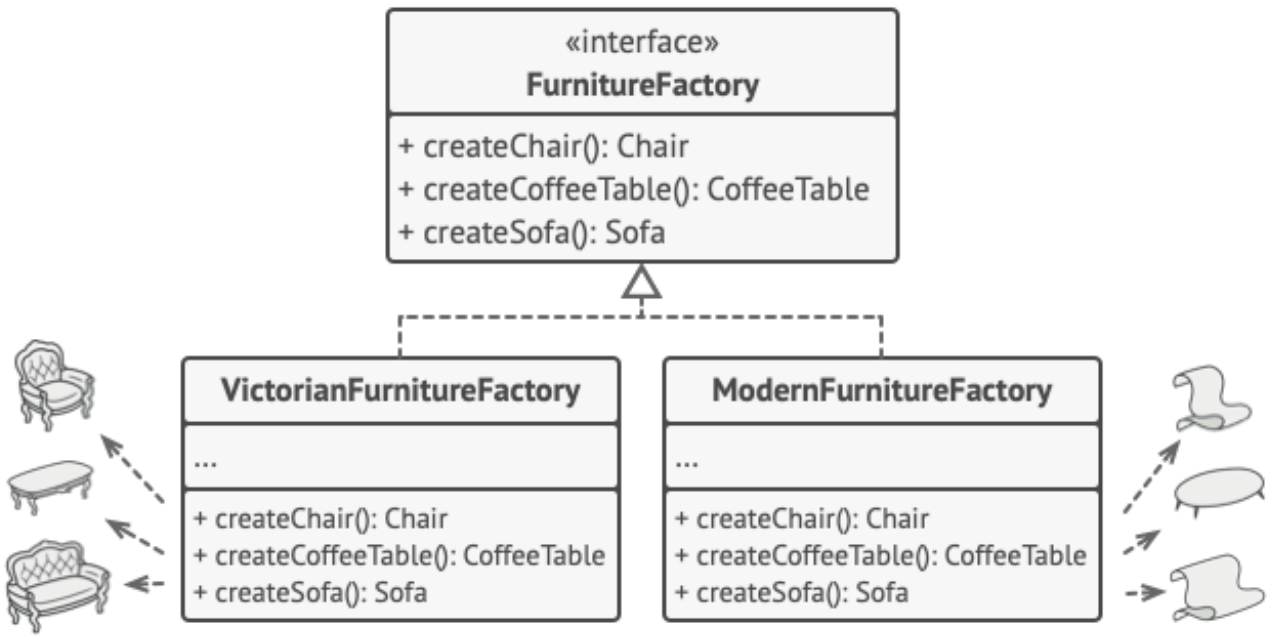
Нужно выделить общие интерфейсы для отдельных продуктов, составляющих семейства.

Далее вы создаёте общий интерфейс, который содержит методы создания всех продуктов. Эти операции должны возвращать абстрактные типы продуктов

РЕШЕНИЕ

Все вариации одного и того же объекта должны жить в одной иерархии классов.





Конкретные фабрики соответствуют определённой вариации семейства продуктов.

ДЛЯ КЛИЕНТСКОГО КОДА ДОЛЖНО БЫТЬ БЕЗРАЗЛИЧНО, С КАКОЙ ФАБРИКОЙ РАБОТАТЬ

Обычно программа создаёт конкретный объект фабрики при запуске, причём тип фабрики выбирается, исходя из параметров окружения или конфигурации.

СТРУКТУРА

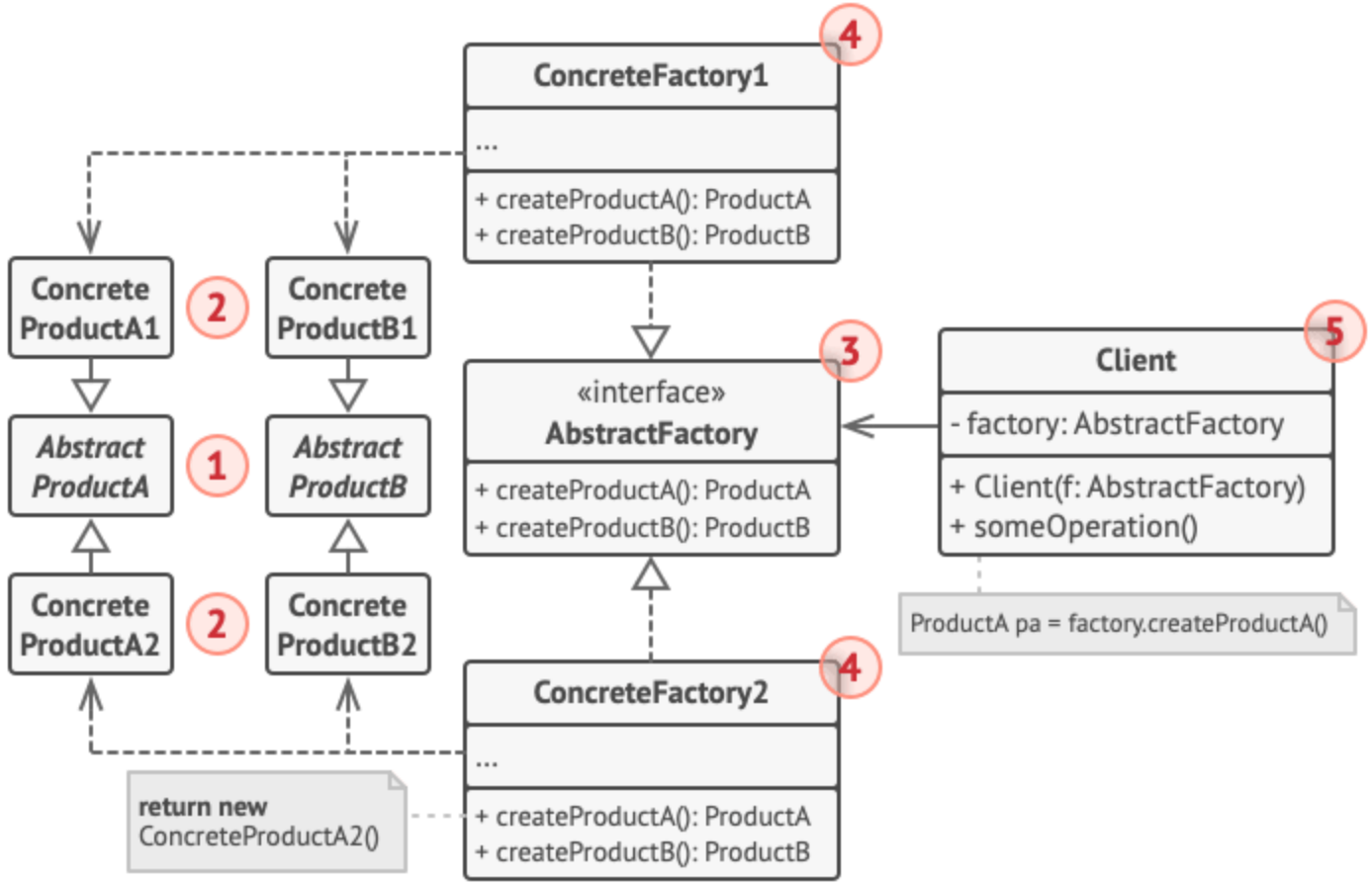
1. **Абстрактные продукты** объявляют интерфейсы продуктов, которые связаны друг с другом по смыслу, но выполняют разные функции.

2. **Конкретные продукты** — большой набор классов, которые относятся к различным абстрактным продуктам (кресло/столик), но имеют одни и те же вариации (Викторианский/Модерн).

3. **Абстрактная фабрика** объявляет методы создания различных абстрактных продуктов (кресло/столик).

4. **Конкретные фабрики** относятся каждая к своей вариации продуктов (Викторианский/Модерн).

5. Несмотря на то, что конкретные фабрики порождают конкретные продукты, сигнатуры их методов должны возвращать соответствующие абстрактные продукты.



ФОРМАЛЬНОЕ ОПРЕДЕЛЕНИЕ ПАТТЕРНА

```
abstract class AbstractProductA
{}
abstract class AbstractProductB
{}


class ProductA1: AbstractProductA
{}
class ProductB1: AbstractProductB
{}


class ProductA2: AbstractProductA
{}
class ProductB2: AbstractProductB
{}
```

```
abstract class AbstractProductA
{}
abstract class AbstractProductB
{}


class ProductA1: AbstractProductA
{}
class ProductB1: AbstractProductB
{}


class ProductA2: AbstractProductA
{}
class ProductB2: AbstractProductB
{}
```

```
abstract class AbstractFactory
{
    public abstract AbstractProductA CreateProductA();
    public abstract AbstractProductB CreateProductB();
}

class ConcreteFactory1: AbstractFactory
{
    public override AbstractProductA CreateProductA()
    {
        return new ProductA1();
    }

    public override AbstractProductB CreateProductB()
    {
        return new ProductB1();
    }
}

class ConcreteFactory2: AbstractFactory
{
    public override AbstractProductA CreateProductA()
    {
        return new ProductA2();
    }

    public override AbstractProductB CreateProductB()
    {
        return new ProductB2();
    }
}
```

```
abstract class AbstractProductA
{}
abstract class AbstractProductB
{}


class ProductA1: AbstractProductA
{}
class ProductB1: AbstractProductB
{}


class ProductA2: AbstractProductA
{}
class ProductB2: AbstractProductB
{}


class Client
{
   private AbstractProductA abstractProductA;
   private AbstractProductB abstractProductB;

   public Client(AbstractFactory factory)
   {
      abstractProductB = factory.CreateProductB();
      abstractProductA = factory.CreateProductA();
   }

   public void Run() { }
}
```

```
abstract class AbstractFactory
{
   public abstract AbstractProductA CreateProductA();
   public abstract AbstractProductB CreateProductB();
}

class ConcreteFactory1: AbstractFactory
{
   public override AbstractProductA CreateProductA()
   {
      return new ProductA1();
   }

   public override AbstractProductB CreateProductB()
   {
      return new ProductB1();
   }
}

class ConcreteFactory2: AbstractFactory
{
   public override AbstractProductA CreateProductA()
   {
      return new ProductA2();
   }

   public override AbstractProductB CreateProductB()
   {
      return new ProductB2();
   }
}
```

РАССМОТРИМ НА ПРИМЕРЕ

Например, мы делаем игру, где пользователь должен управлять некими супергероями, при этом каждый супергерой имеет определенное оружие и определенную модель передвижения.

Различные супергерои могут определяться комплексом признаков. Например, эльф может летать и должен стрелять из арбалета, другой супергерой должен бегать и управлять мечом.

Таким образом, получается, что сущность оружия и модель передвижения являются взаимосвязанными и используются в комплексе.

```
//абстрактный класс - оружие
abstract class Weapon
{
    public abstract void Hit();
}
// абстрактный класс движение
abstract class Movement
{
    public abstract void Move();
}

// класс арбалет
class Arbalet : Weapon
{
    public override void Hit() ...
}

// класс меч
class Sword : Weapon
{
    public override void Hit() ...
}

// движение полета
class FlyMovement : Movement
{
    public override void Move() ...
}
// движение - бег
class RunMovement : Movement
{
    public override void Move() ...
}
```

```
//абстрактный класс - оружие
abstract class Weapon
{
    public abstract void Hit();
}
// абстрактный класс движение
abstract class Movement
{
    public abstract void Move();
}

// класс арбалет
class Arbalet : Weapon
{
    public override void Hit() ...
}

// класс меч
class Sword : Weapon
{
    public override void Hit() ...
}

// движение полета
class FlyMovement : Movement
{
    public override void Move() ...
}
// движение - бег
class RunMovement : Movement
{
    public override void Move() ...
}
```

```
// класс абстрактной фабрики
abstract class HeroFactory
{
    public abstract Movement CreateMovement();
    public abstract Weapon CreateWeapon();
}

// Фабрика создания летящего героя с арбалетом
class ElfFactory : HeroFactory
{
    public override Movement CreateMovement()
    {
        return new FlyMovement();
    }

    public override Weapon CreateWeapon()
    {
        return new Arbalet();
    }
}
// Фабрика создания бегущего героя с мечом
class VoinFactory : HeroFactory
{
    public override Movement CreateMovement() ...

    public override Weapon CreateWeapon() ...
}
```

```
// клиент - сам супергерой
class Hero
{
    private Weapon weapon;
    private Movement movement;

    public Hero(HeroFactory factory)
    {
        weapon = factory.CreateWeapon();
        movement = factory.CreateMovement();
    }
    public void Run()
    {
        movement.Move();
    }
    public void Hit()
    {
        weapon.Hit();
    }
}
```

```
// клиент - сам супергерой
class Hero
{
    private Weapon weapon;
    private Movement movement;

    public Hero(HeroFactory factory)
    {
        weapon = factory.CreateWeapon();
        movement = factory.CreateMovement();
    }
    public void Run()
    {
        movement.Move();
    }
    public void Hit()
    {
        weapon.Hit();
    }
}
```

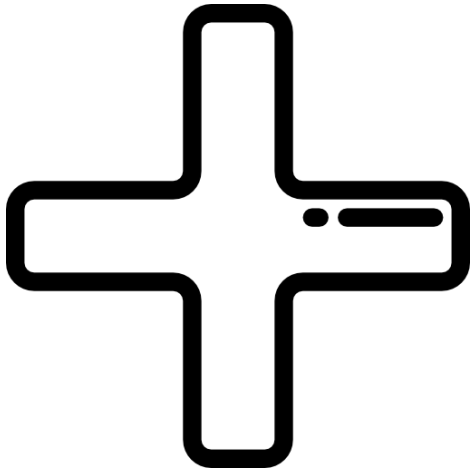
```
Hero elf = new Hero(new ElfFactory());
elf.Hit();
elf.Run();

Hero voin = new Hero(new VoinFactory());
voin.Hit();
voin.Run();
```

ПРИМЕНИМОСТЬ

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

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

ПРЕИМУЩЕСТВА И НЕДОСТАТКИ



Гарантирует сочетаемость создаваемых продуктов.

Избавляет клиентский код от привязки к конкретным классам продуктов.

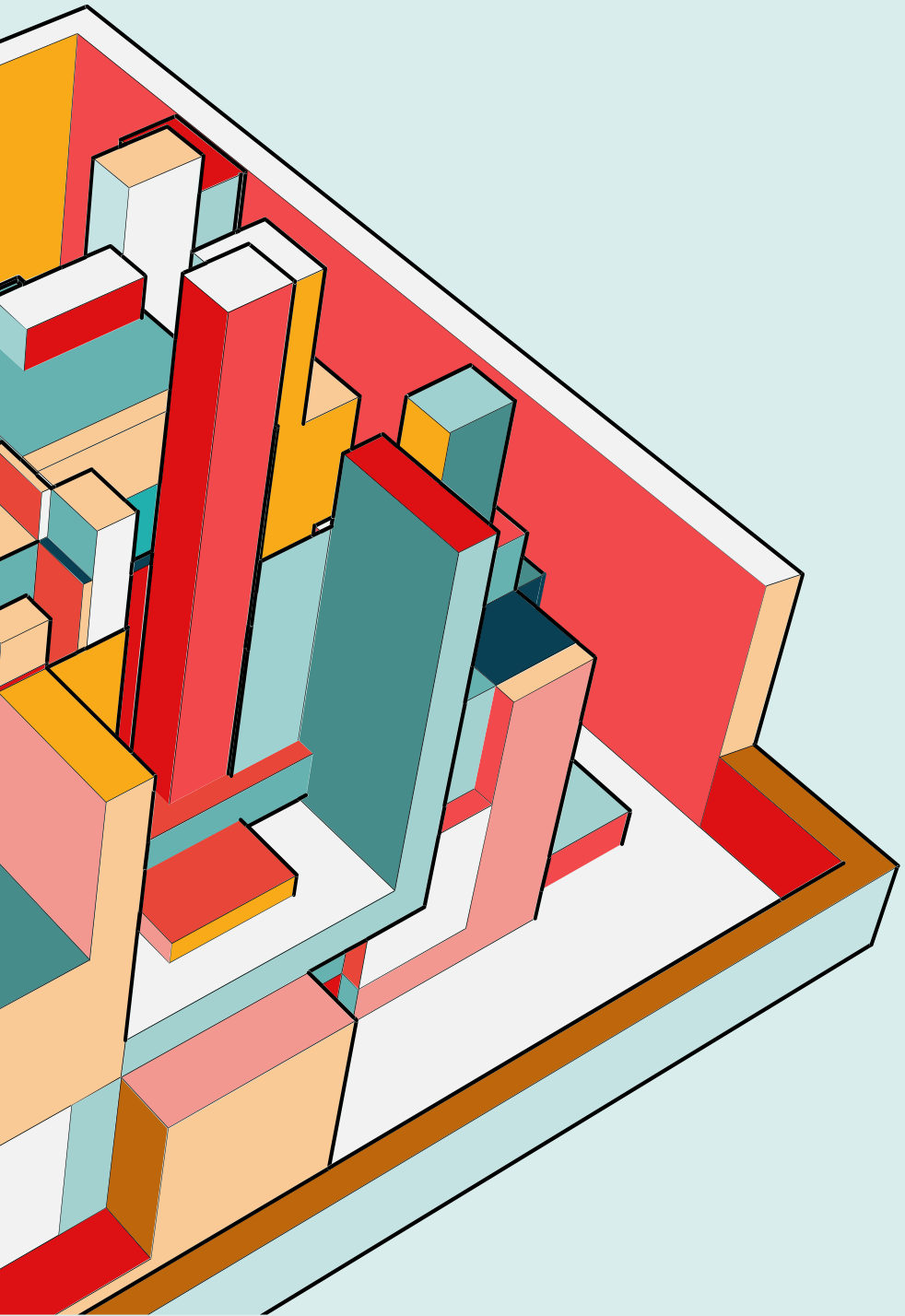
Выделяет код производства продуктов в одно место, упрощая поддержку кода.

Упрощает добавление новых продуктов в программу.

Реализует принцип открытости/закрытости.

Усложняет код программы.

Требует наличия всех типов продуктов в каждой вариации.

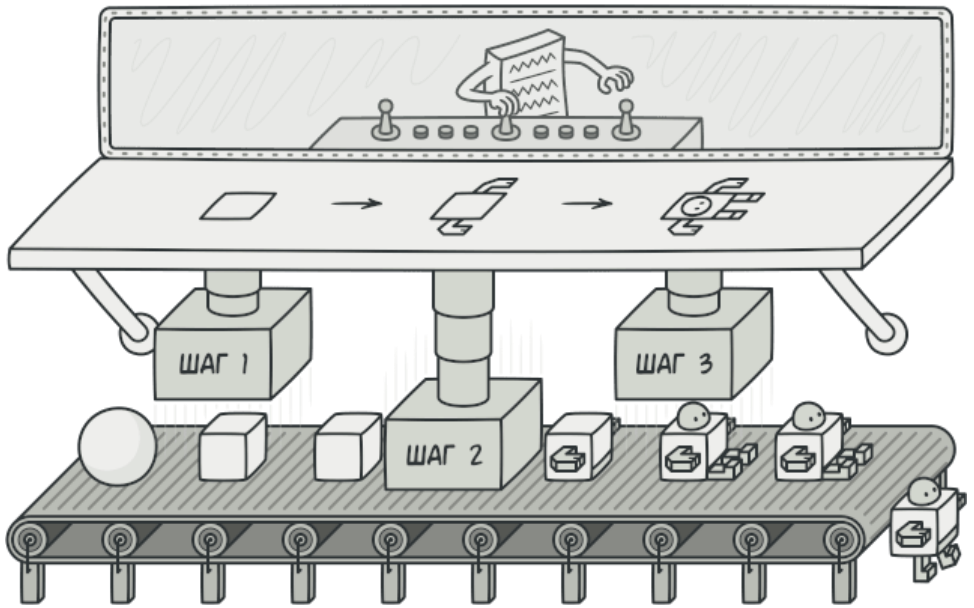


ПОРОЖДАЮЩИЕ ПАТТЕРНЫ ПРОЕКТИРОВАНИЯ

Builder

СТРОИТЕЛЬ

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



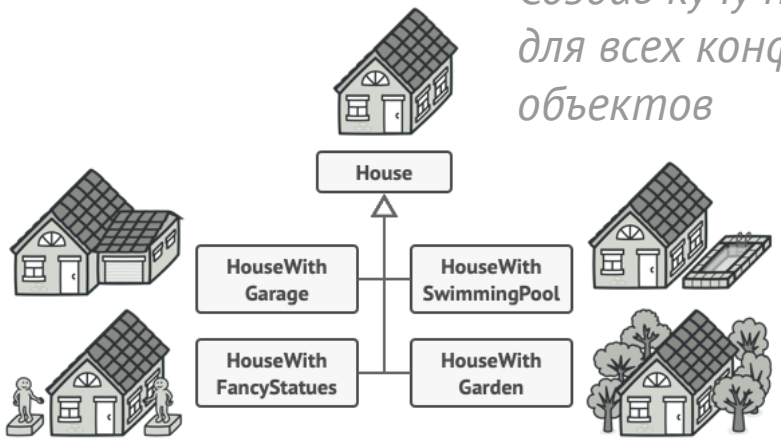
Ваш код содержит сложный объект, требующий кропотливой пошаговой инициализации.

Код инициализации таких объектов обычно спрятан внутри монструозного конструктора с десятком параметров.

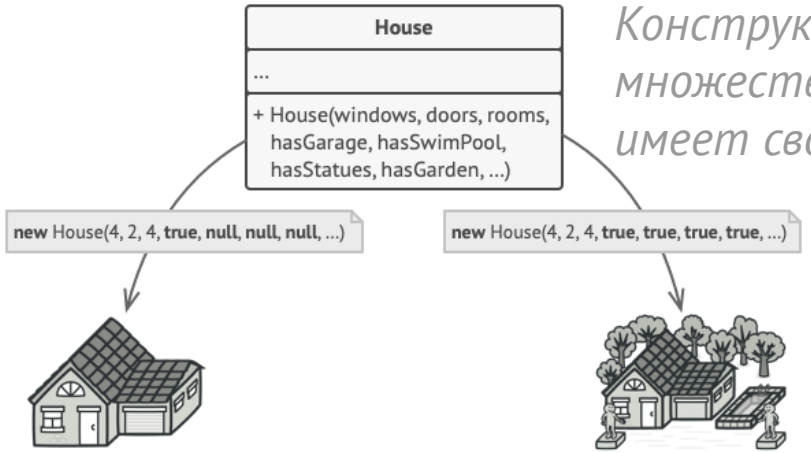
Нужно вынести конструирование объекта за пределы его собственного класса, поручив это дело отдельным объектам, которые следует называть строителями.

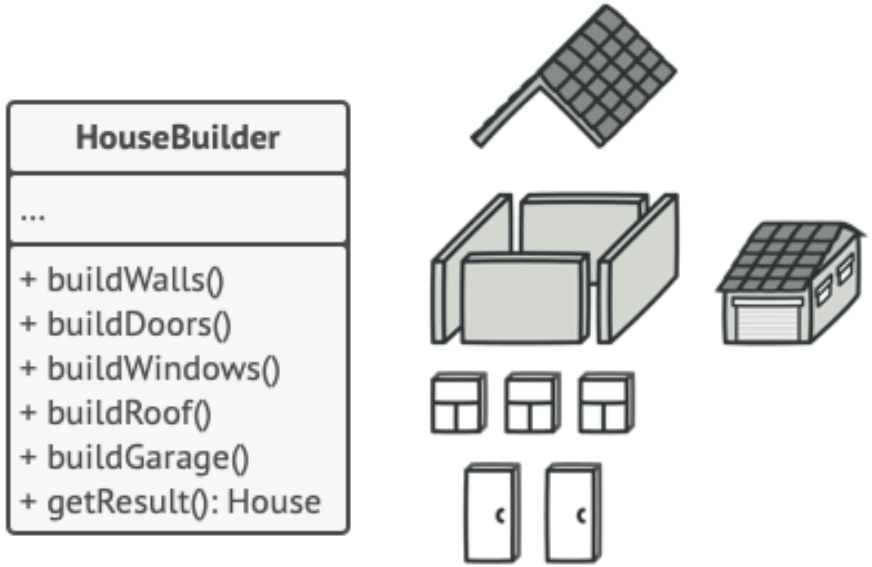
РЕШЕНИЕ

Создав кучу подклассов для всех конфигураций объектов



Конструктор со множеством параметров имеет свой недостаток

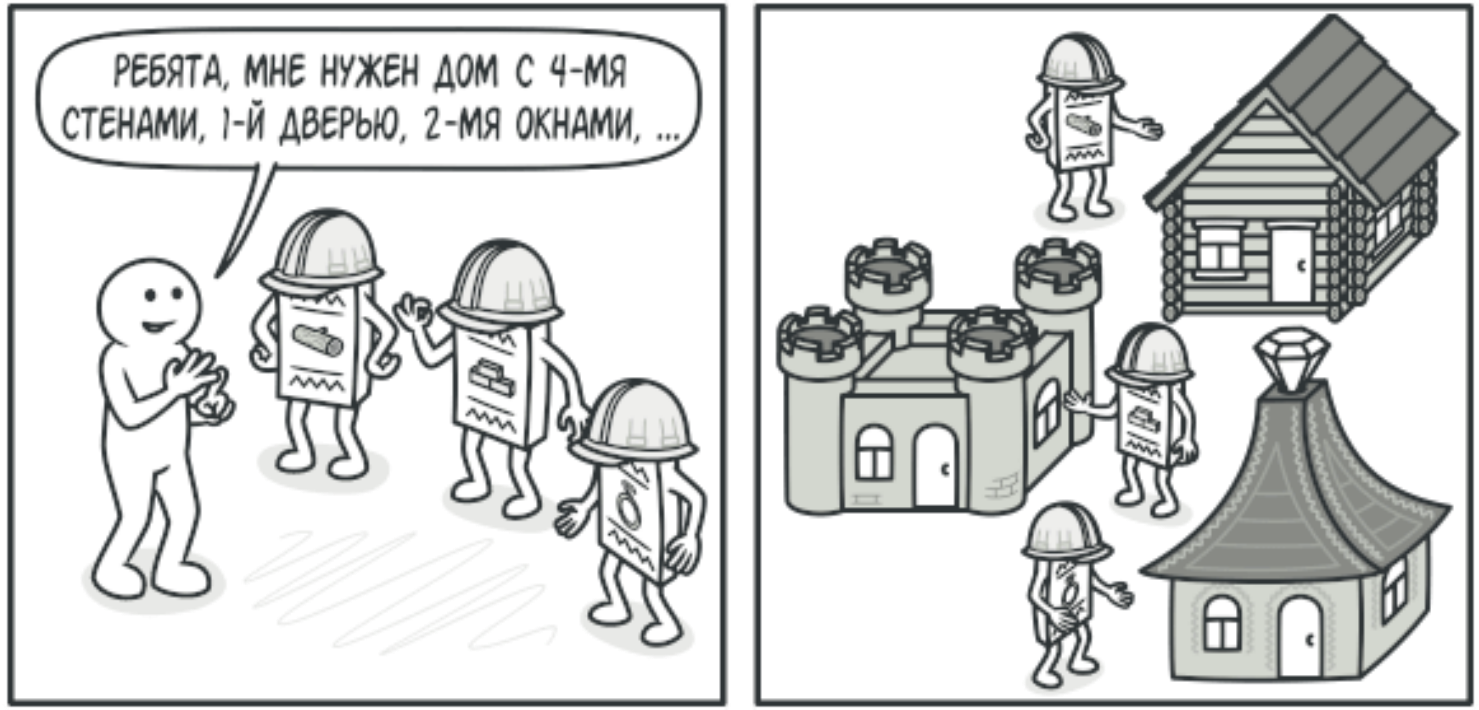




Строитель позволяет создавать сложные объекты пошагово. Промежуточный результат всегда остаётся защищён.

РАЗНЫЕ СТРОИТЕЛИ ВЫПОЛНЯТ ОДНУ И ТУ ЖЕ ЗАДАЧУ ПО-РАЗНОМУ.

Код, который вызывает шаги строительства, должен работать со строителями через общий интерфейс, чтобы их можно было свободно взаимозаменять.



ДИРЕКТОР

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

Такая структура классов полностью скроет от клиентского кода процесс конструирования объектов.

Клиенту останется только привязать желаемого строителя к директору, а затем получить у строителя готовый результат.

СТРУКТУРА

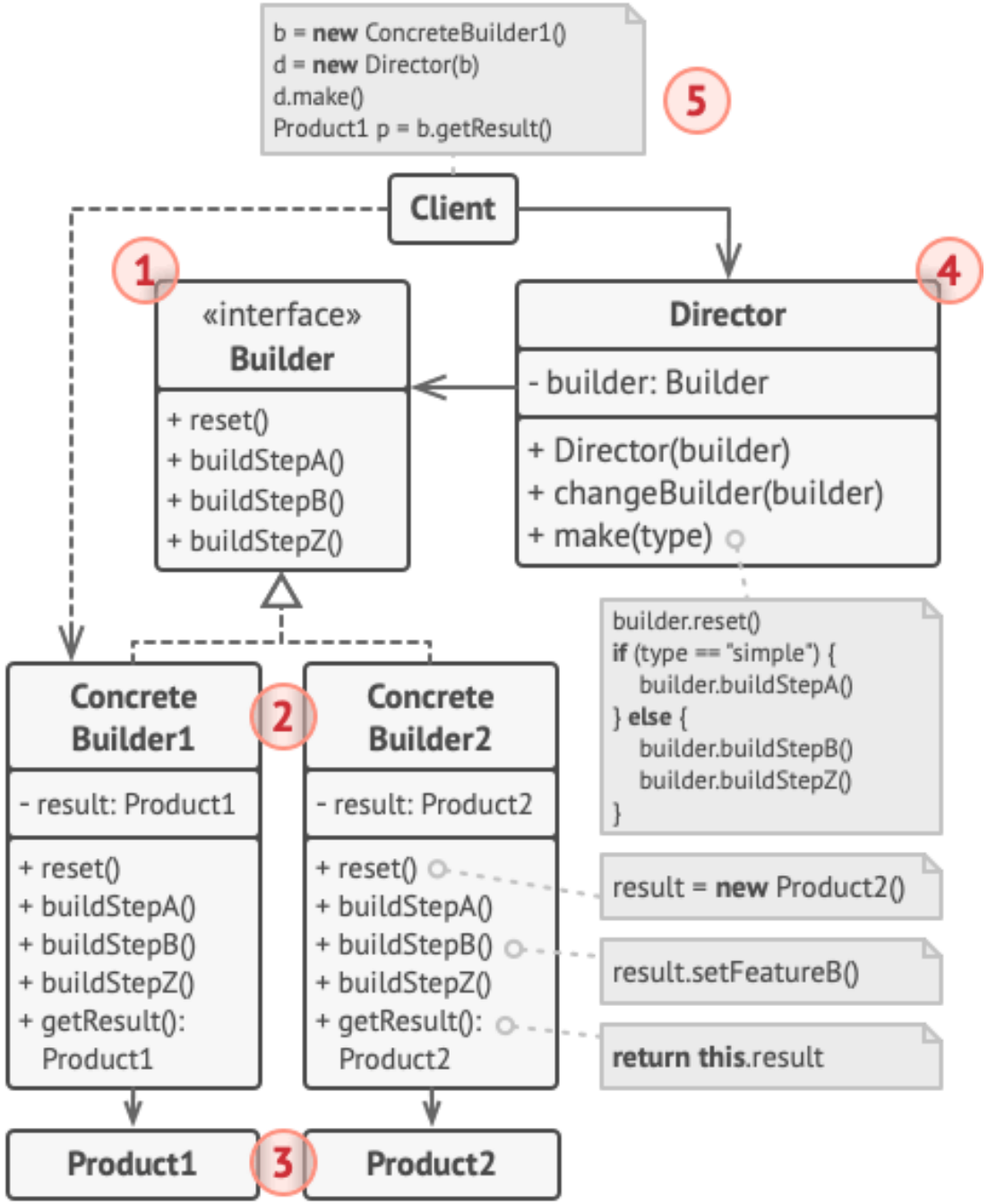
1. **Интерфейс строителя** объявляет шаги конструирования продуктов, общие для всех видов строителей.

2. **Конкретные строители** реализуют строительные шаги, каждый по-своему.

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.



```
class Product
{
    List<object> parts = new List<object>()
    public void Add(string part)
    {
        parts.Add(part);
    }
}

abstract class Builder
{
    public abstract void BuildPartA();
    public abstract void BuildPartB();
    public abstract void BuildPartC();
    public abstract Product GetResult();
}

class ConcreteBuilder : Builder
{
    Product product = new Product();

    public override void BuildPartA()
    {
        product.Add("Part A");
    }
    public override void BuildPartB() ...
    public override void BuildPartC() ...

    public override Product GetResult()
    {
        return product;
    }
}
```

```
class Product
{
    List<object> parts = new List<object>()
    public void Add(string part)
    {
        parts.Add(part);
    }
}

abstract class Builder
{
    public abstract void BuildPartA();
    public abstract void BuildPartB();
    public abstract void BuildPartC();
    public abstract Product GetResult();
}

class ConcreteBuilder : Builder
{
    Product product = new Product();

    public override void BuildPartA()
    {
        product.Add("Part A");
    }
    public override void BuildPartB() ...
    public override void BuildPartC() ...

    public override Product GetResult()
    {
        return product;
    }
}
```

```
class Director
{
    Builder builder;
    public Director(Builder builder)
    {
        this.builder = builder;
    }
    public void Construct()
    {
        builder.BuildPartA();
        builder.BuildPartB();
        builder.BuildPartC();
    }
}
```

```
Builder builder = new ConcreteBuilder();
Director director = new Director(builder);
director.Construct();
Product product = builder.GetResult();
```

РАССМОТРИМ НА ПРИМЕРЕ

Рассмотрим применение паттерна на примере выпечки хлеба.

```
class Flour
{
    // какого сорта мука
    public string Sort { get; set; }
}
// соль
class Salt
{ }

// пищевые добавки
class Additives
{
    public string Name { get; set; }
}

class Bread
{
    // мука
    public Flour Flour { get; set; }
    // соль
    public Salt Salt { get; set; }
    // пищевые добавки
    public Additives Additives { get; set; }
    public override string ToString() ...
}
```

```
abstract class BreadBuilder
{
    public Bread Bread { get; private set; }
    public void CreateBread()
    {
        Bread = new Bread();
    }
    public abstract void SetFlour();
    public abstract void SetSalt();
    public abstract void SetAdditives();
}
// пекарь
class Baker
{
    public Bread Bake(BreadBuilder breadBuilder)
    {
        breadBuilder.CreateBread();
        breadBuilder.SetFlour();
        breadBuilder.SetSalt();
        breadBuilder.SetAdditives();
        return breadBuilder.Bread;
    }
}
```

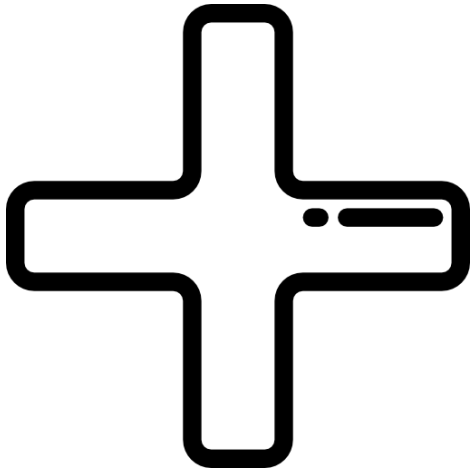
```
// строитель для ржаного хлеба
class RyeBreadBuilder : BreadBuilder
{
    public override void SetFlour()
    {
        this.Bread.Flour = new Flour { Sort = "Ржаная мука 1 сорт" }
    }

    public override void SetSalt()
    {
        this.Bread.Salt = new Salt();
    }

    public override void SetAdditives()
    {
        // не используется
    }
}
// строитель для пшеничного хлеба
class WheatBreadBuilder ...
```

```
// содаем объект пекаря
Baker baker = new Baker();

// создаем билдер для ржаного хлеба
BreadBuilder builder = new RyeBreadBuilder();

// выпекаем
Bread ryeBread = baker.Bake(builder);

// оздаем билдер для пшеничного хлеба
builder = new WheatBreadBuilder();

Bread wheatBread = baker.Bake(builder);
```

ПРИМЕНИМОСТЬ

- Когда вы хотите избавиться от «телескопического конструктора».
- Когда ваш код должен создавать разные представления какого-то объекта. Например, деревянные и железобетонные дома.

ПРЕИМУЩЕСТВА И НЕДОСТАТКИ



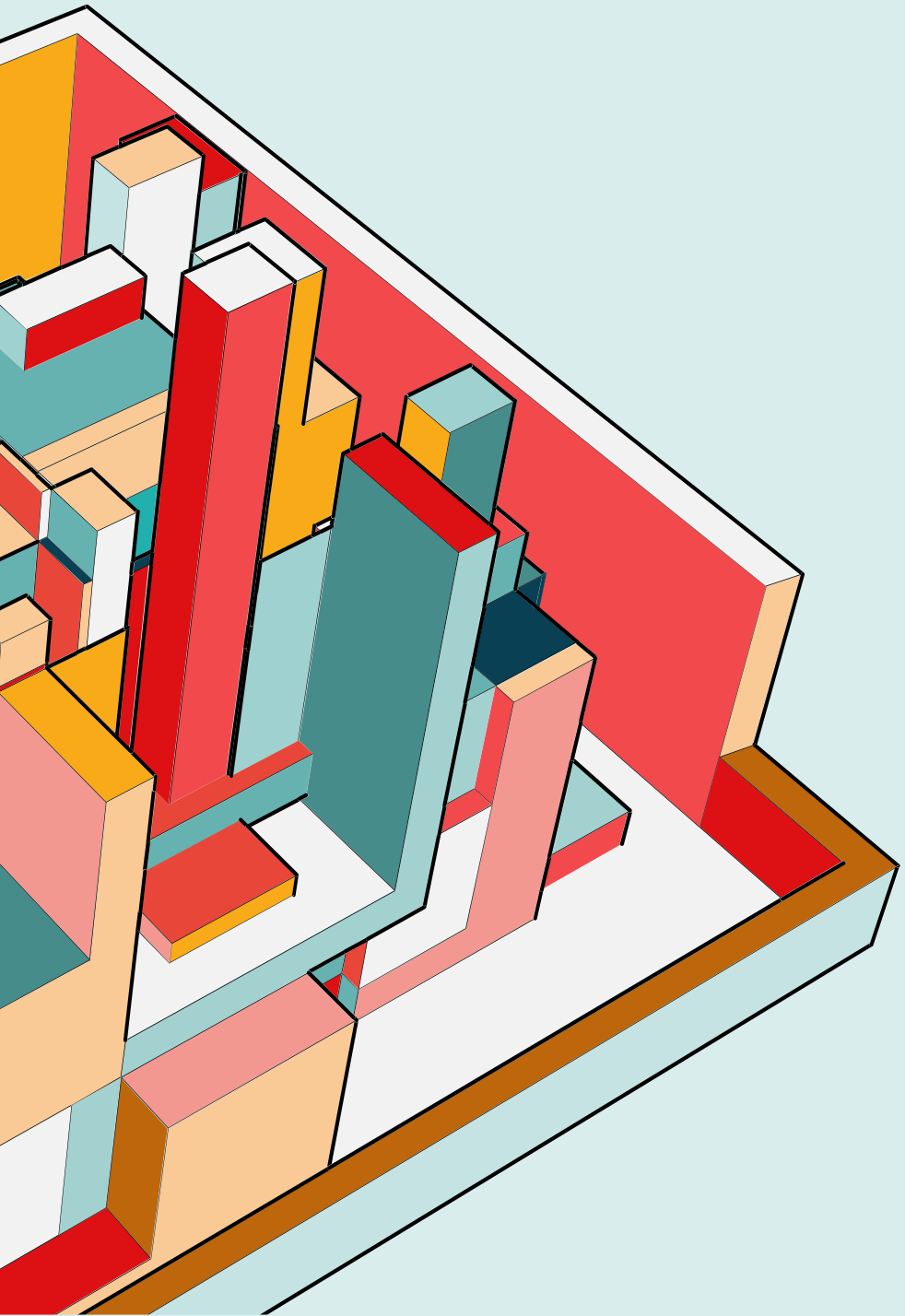
Позволяет создавать продукты пошагово.

Позволяет использовать один и тот же код для создания различных продуктов.

Изолирует сложный код сборки продукта от его основной бизнес-логики.

Усложняет код программы из-за введения дополнительных классов.

Клиент будет привязан к конкретным классам строителей, так как в интерфейсе директора может не быть метода получения результата.

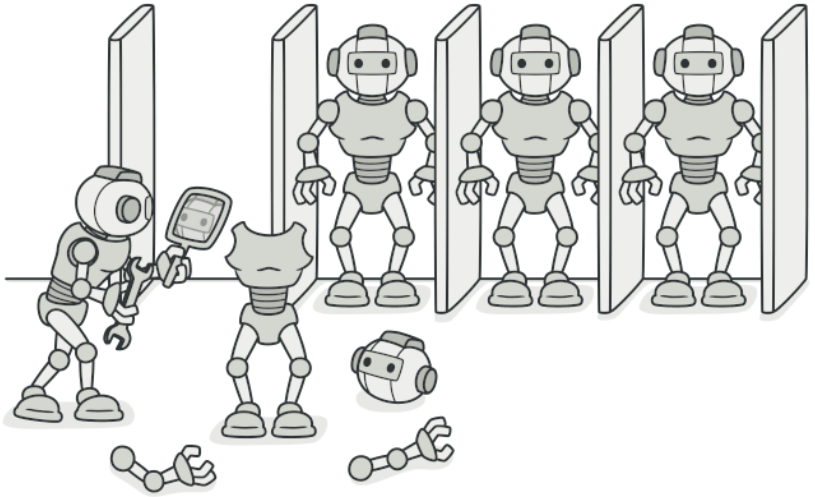


ПОРОЖДАЮЩИЕ ПАТТЕРНЫ ПРОЕКТИРОВАНИЯ

Prototype

ПРОТОТИП

Позволяет копировать объекты, не вдаваясь в подробности их реализации.

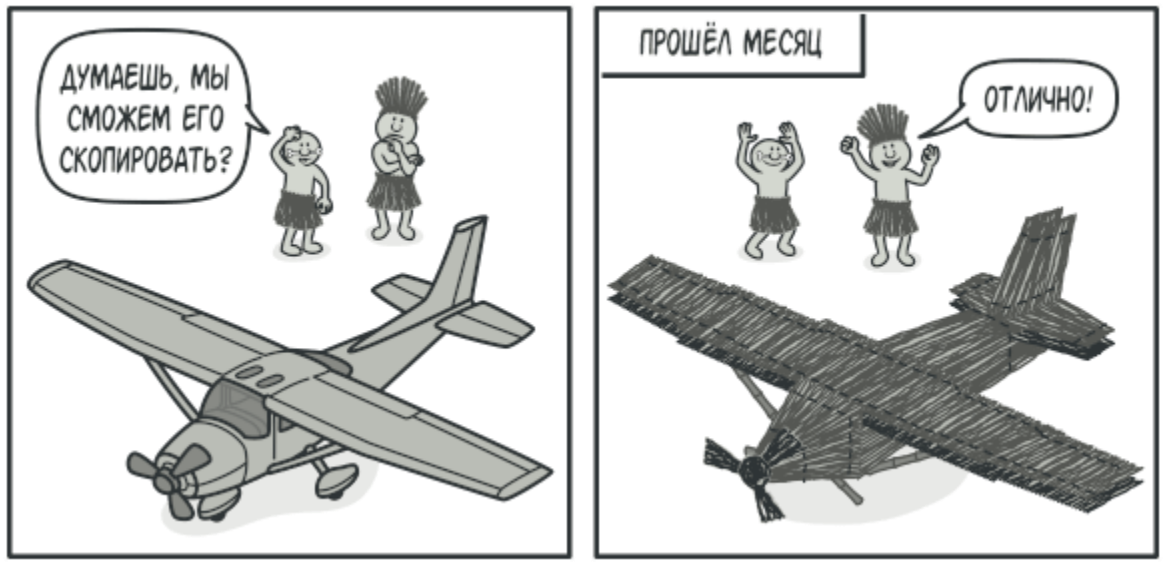


У вас есть объект, который нужно скопировать. Как это сделать?

Создать пустой объект такого же класса, а затем поочерёдно скопировать значения всех полей из старого объекта.

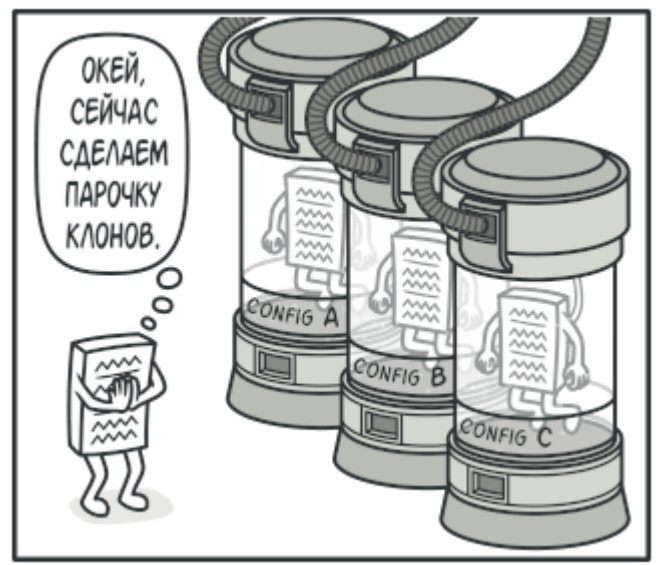
Поручить создание копий самим копируемым объектам.

РЕШЕНИЕ



Копирование «извне» не всегда возможно в реальности.

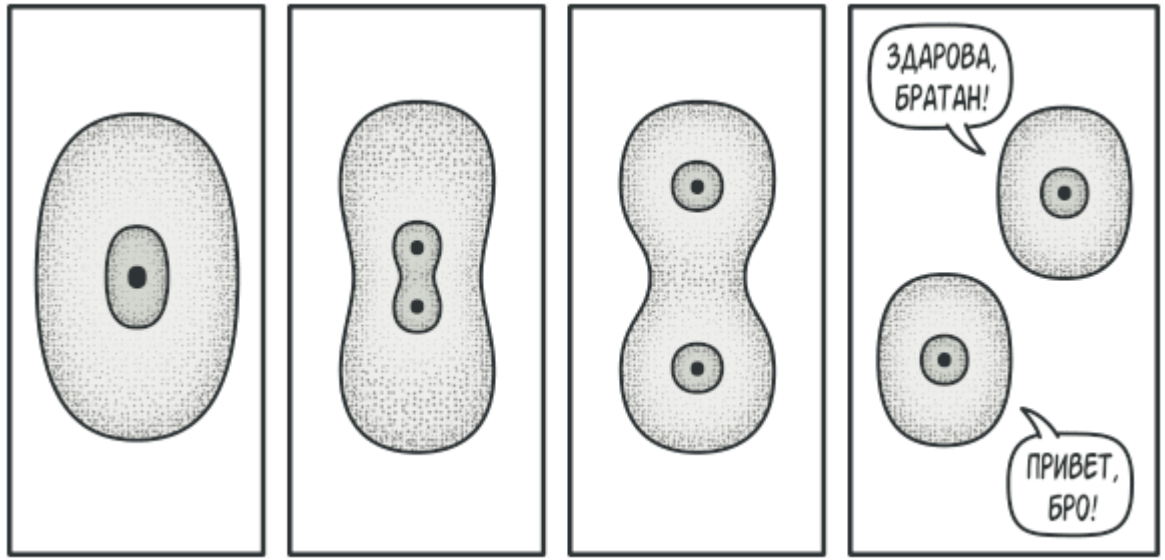
Предварительно заготовленные прототипы могут стать заменой подклассам.



ПРИМЕР ДЕЛЕНИЯ КЛЕТКИ.

После митозного деления клеток образуются две совершенно идентичные клетки.

Оригинальная клетка отыгрывает роль прототипа, принимая активное участие в создании нового объекта.

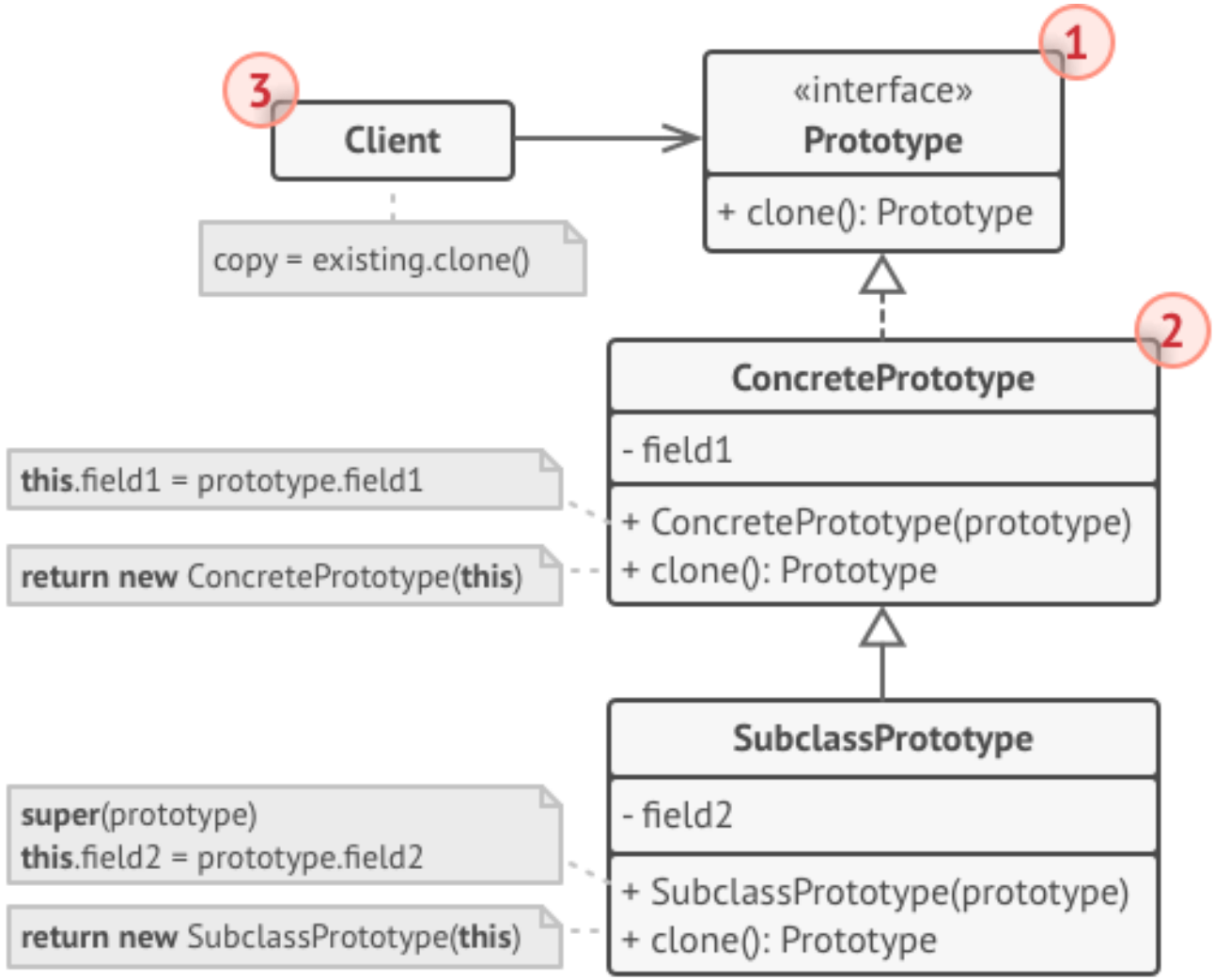


СТРУКТУРА

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.

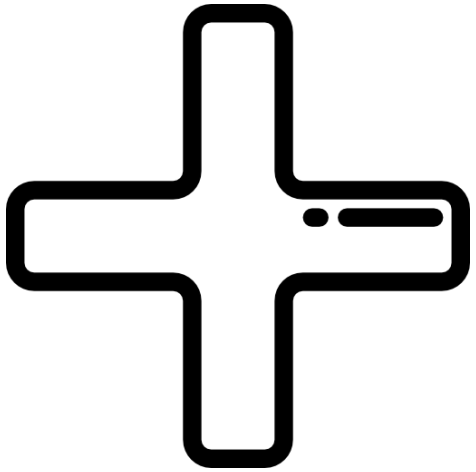
3. **Клиент** создаёт копию объекта, обращаясь к нему через общий интерфейс прототипов.



```
abstract class Prototype
{
    public int Id { get; private set; }
    public Prototype(int id)
    {
        this.Id = id;
    }
    public abstract Prototype Clone();
}

class ConcretePrototype1 : Prototype
{
    public ConcretePrototype1(int id)
        : base(id)
    { }
    public override Prototype Clone()
    {
        return new ConcretePrototype1(Id);
    }
}

class ConcretePrototype2 : Prototype
{
    public ConcretePrototype2(int id)
        : base(id)
    { }
    public override Prototype Clone() ...
}
```

```
Prototype prototype = new ConcretePrototype1(1);
Prototype clone = prototype.Clone();
prototype = new ConcretePrototype2(2);
clone = prototype.Clone();
```

ПРЕИМУЩЕСТВА И НЕДОСТАТКИ



Позволяет клонировать объекты, не привязываясь к их конкретным классам.

Меньше повторяющегося кода инициализации объектов.

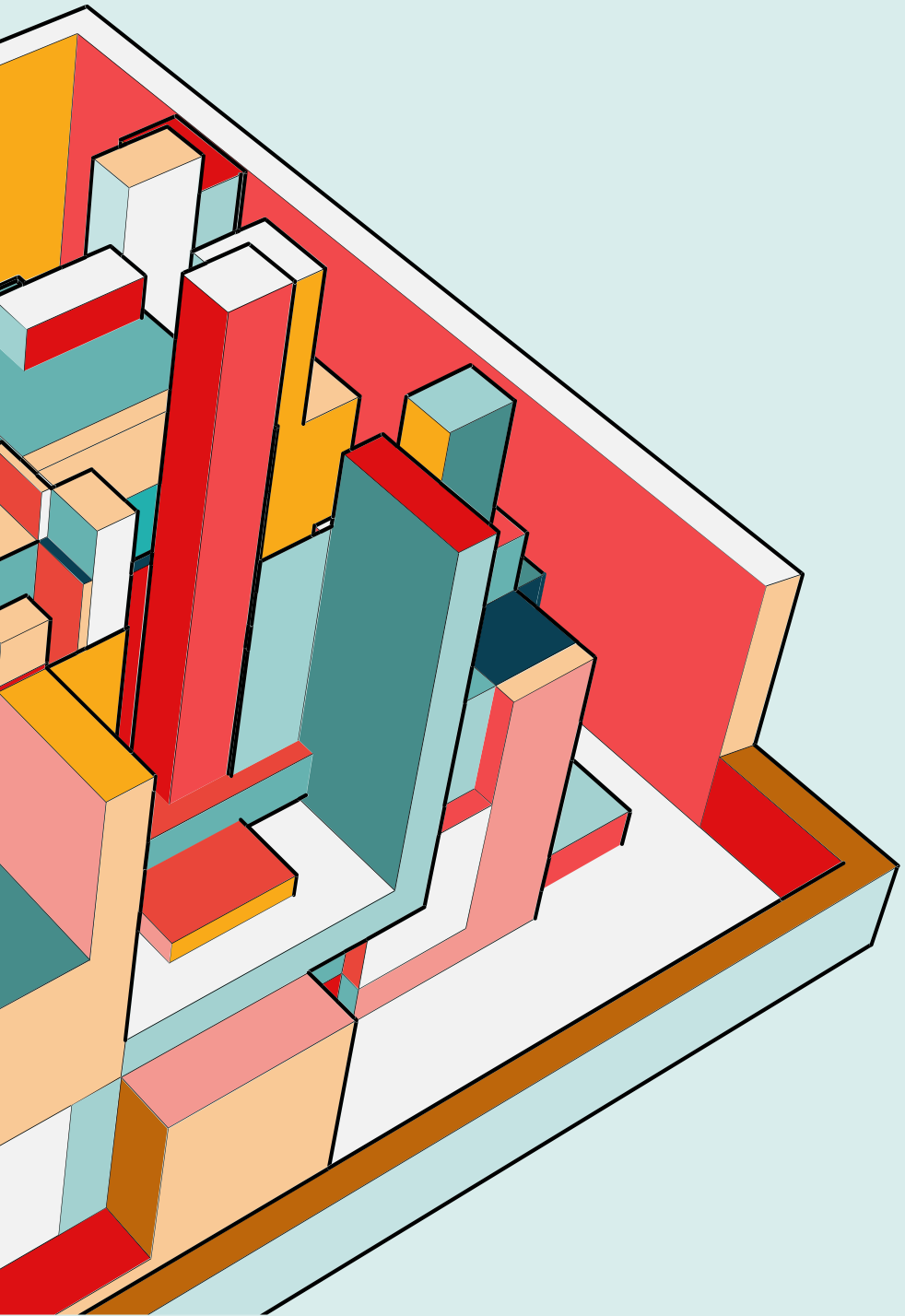
Ускоряет создание объектов.

Альтернатива созданию подклассов для конструирования сложных объектов.

Сложно клонировать составные объекты, имеющие ссылки на другие объекты.

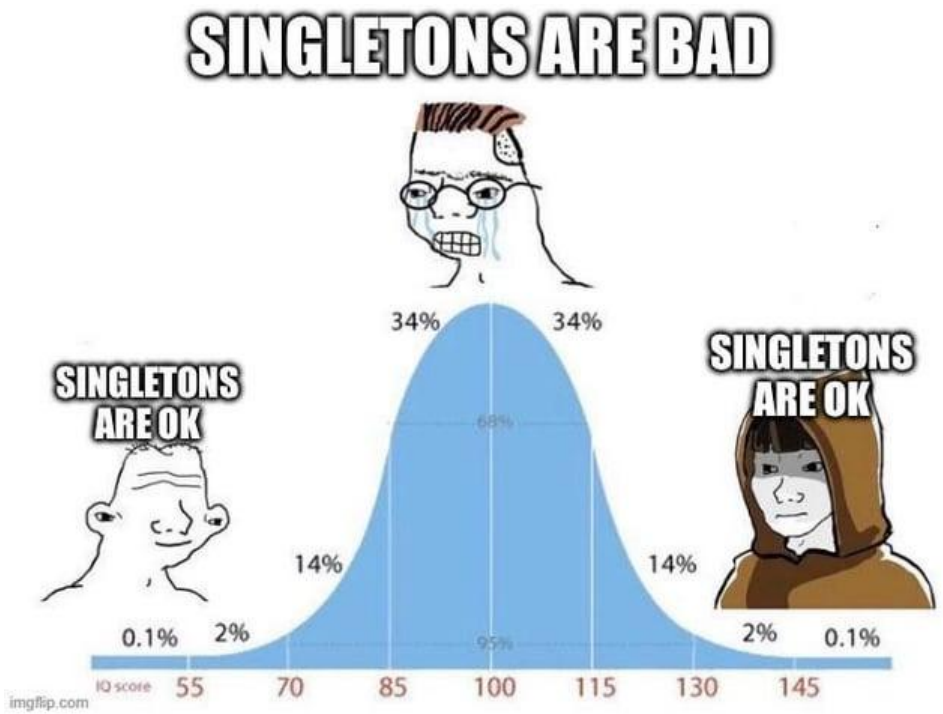
ПРИМЕНИМОСТЬ

- Когда ваш код не должен зависеть от классов копируемых объектов.
- Когда вы имеете уйму подклассов, которые отличаются начальными значениями полей. Кто-то мог создать все эти классы, чтобы иметь возможность легко порождать объекты с определённой конфигурацией.



ПОРОЖДАЮЩИЕ ПАТТЕРНЫ ПРОЕКТИРОВАНИЯ

Singleton


SINGLETONS ARE BAD
SINGLETONS ARE OK
SINGLETONS ARE OK
34%
34%
14%
14%
0.1%
2%
2%
0.1%
IQ score
55
70
85
100
115
130
145
imgflip.com

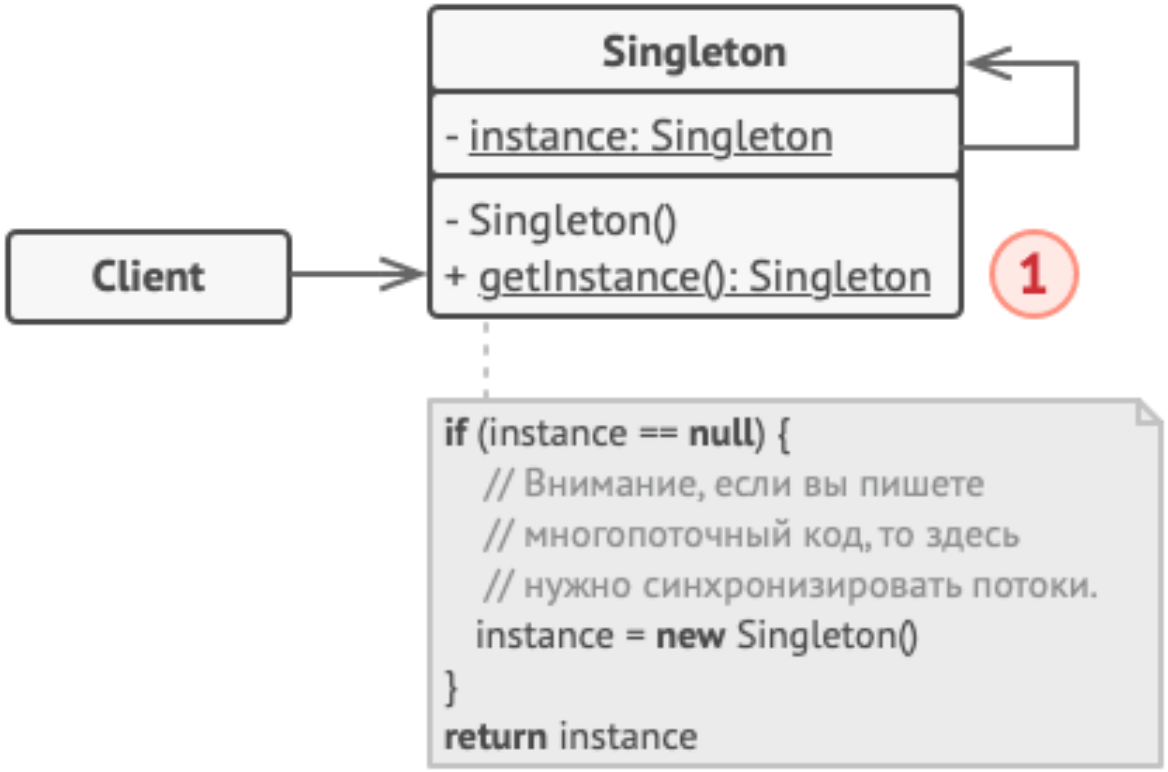

Singleton - антипаттерн, ты как вообще на работу к нам попал?

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.

.



КЛАССИЧЕСКАЯ РЕАЛИЗАЦИЯ ДАННОГО ШАБЛОНА

```
class Singleton
{
    private static Singleton instance;

    private Singleton()
    { }

    public static Singleton getInstance()
    {
        if (instance == null)
            instance = new Singleton();
        return instance;
    }
}
```

СИНГЛТОН И МНОГОПОТОЧНОСТЬ

```
class Computer
{
    public OS OS { get; set; }
    public void Launch(string osName)
    {
        OS = OS.getInstance(osName);
    }
}
class OS
{
    private static OS instance;

    public string Name { get; private set; }

    protected OS(string name)
    {
        this.Name = name;
    }

    public static OS getInstance(string name)
    {
        if (instance == null)
            instance = new OS(name);
        return instance;
    }
}
```

```
(new Thread(() =>
{
    Computer comp2 = new Computer();
    comp2.OS = OS.getInstance("Windows 10");
    Console.WriteLine(comp2.OS.Name);

})).Start();

Computer comp = new Computer();
comp.Launch("Windows 8.1");
Console.WriteLine(comp.OS.Name);
Console.ReadLine();
```

ИСТОЧНИКИ

https://refactoringguru.cn/ru/design-patterns/creational-patterns

https://metanit.com/sharp/patterns/2.1.php

https://habr.com/ru/articles/210288/

СПАСИБО!

Виденин Сергей

@videninserg