Creational Design Patterns

📌 Упрощают создание объектов, уменьшая зависимость от конкретных классов.

👤 Singleton (Одиночка)

Цель: Гарантирует 1 экземпляр класса

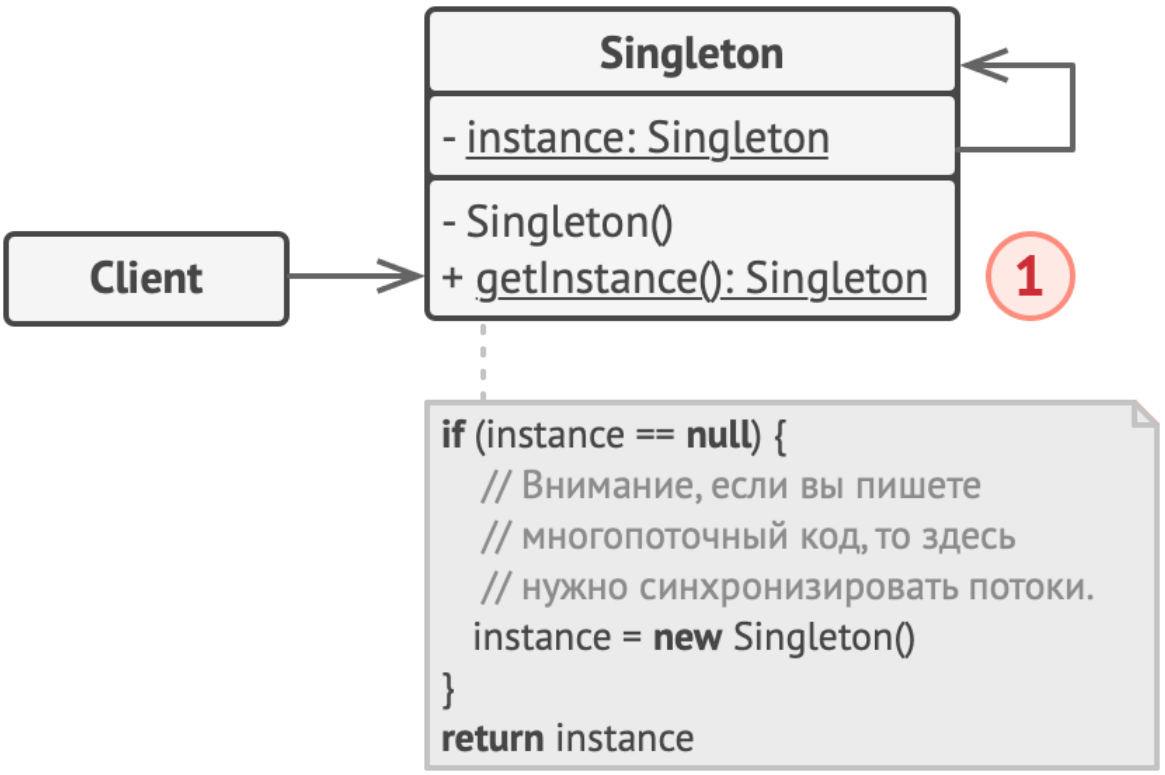
🔷**Когда:** Логгеры, глобальные настройки, кэш

👥 **Участники:**

- **Singleton** — класс с приватным конструктором и статическим методом доступа.

✅ **Плюсы:** Глобальный доступ

❌ **Минусы:** Трудно тестировать




```
public sealed class ApplicationConfig
{
    private static ApplicationConfig _instance;
    public string ConfigData { get; set; }

    private ApplicationConfig() {} // Приватный конструктор

    public static ApplicationConfig Instance => _instance ??= new Application
Config();
}

// Использование:
ApplicationConfig.Instance.ConfigData = "Settings";
Console.WriteLine(ApplicationConfig.Instance.ConfigData); // "Settings"
```

🏭 Factory Method (Фабричный метод)

Цель: Делегирует создание подклассам

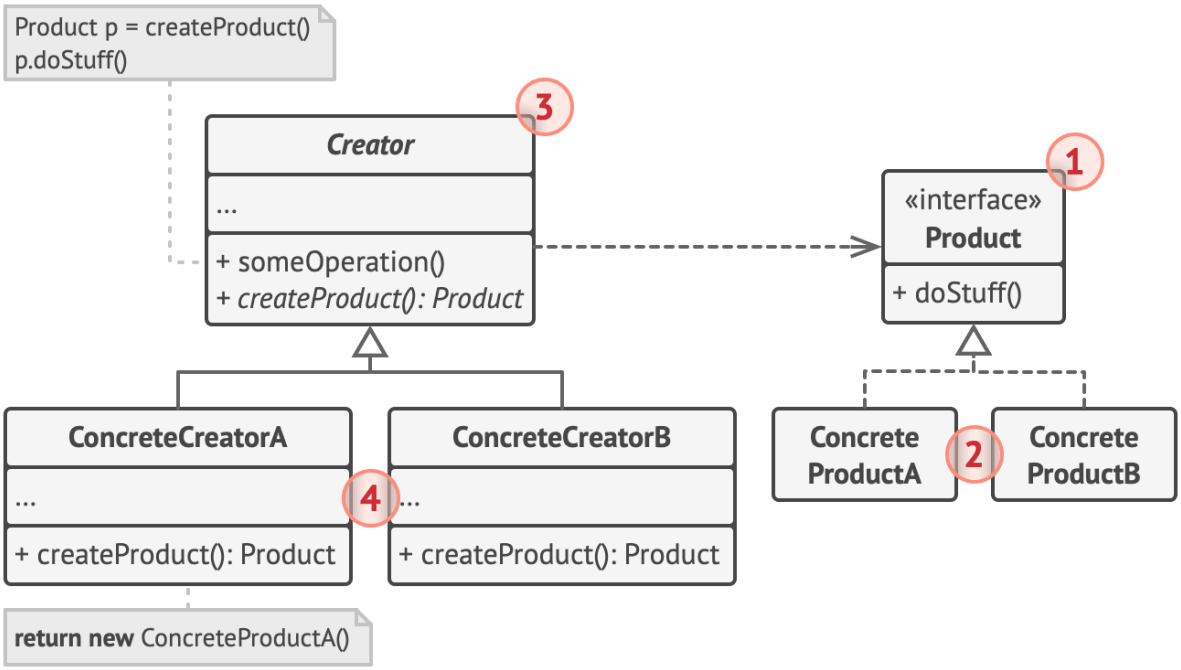
🔷 **Когда:** Разные реализации одного интерфейса

👥 **Участники:**

- **Product** — интерфейс создаваемого объекта (напр., `IButton`).
- **ConcreteProduct** — конкретная реализация (`WindowsButton`).
- **Creator** — абстрактный класс с фабричным методом (`Dialog`).
- **ConcreteCreator** — реализует фабричный метод (`WindowsDialog`).

✅ **Плюсы:** Гибкость

❌ **Минусы:** Много классов




```
public interface IButton { void Render(); }

public class WindowsButton : IButton
{
    public void Render() => Console.WriteLine("Windows button rendered");
}

public abstract class Dialog
{
    public abstract IButton CreateButton(); // Фабричный метод
}
```

```
public class WindowsDialog : Dialog
{
    public override IButton CreateButton() => new WindowsButton();
}

// Использование:
Dialog dialog = new WindowsDialog();
IButton button = dialog.CreateButton();
button.Render(); // "Windows button rendered"
```

🏢 Abstract Factory (Абстрактная фабрика)

Цель: Создает семейства объектов

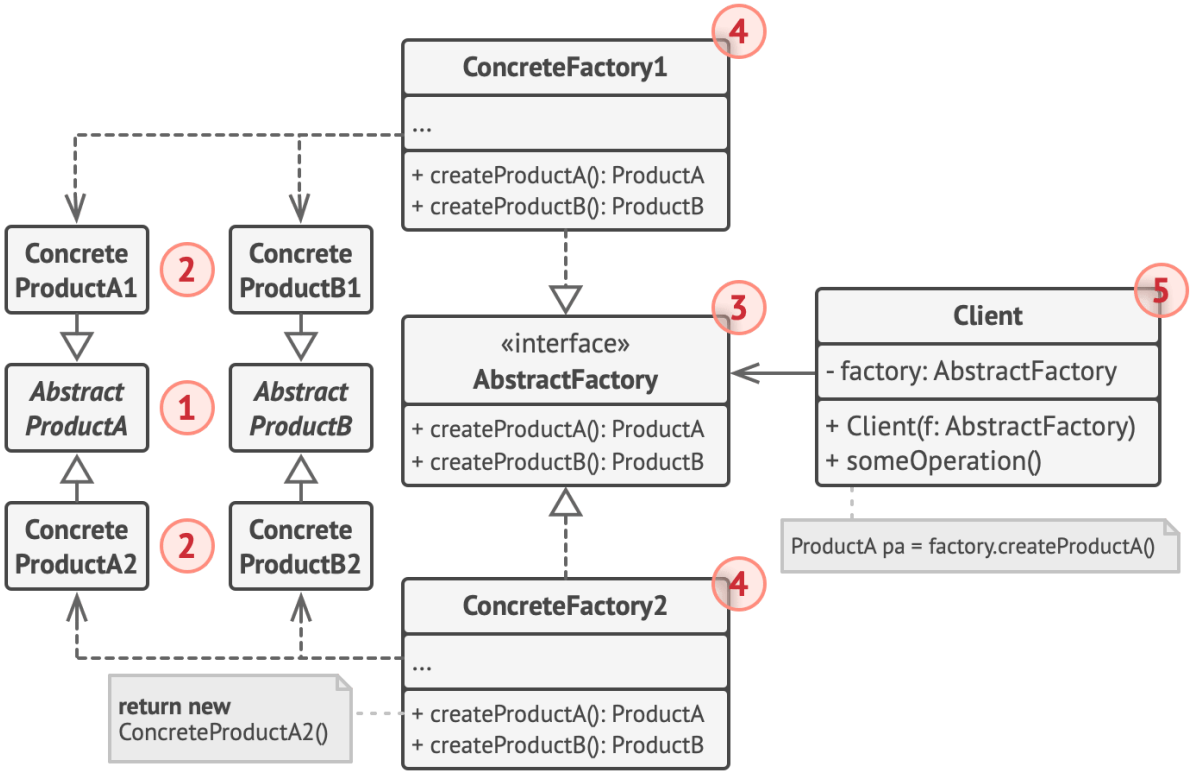
🔷 **Когда:** UI-библиотеки, темы

👥 **Участники**

- **AbstractFactory** — интерфейс фабрики (`IGUIFactory`).
- **ConcreteFactory** — реализация фабрики (`WindowsFactory`).
- **AbstractProduct** — интерфейс продукта (`IButton`).
- **ConcreteProduct** — конкретный продукт (`WindowsButton`).

✅ **Плюсы:** Совместимость компонентов

❌ **Минусы:** Сложность




```
public interface IButton { void Render(); }
public interface ICheckbox { void Render(); }

public class WindowsButton : IButton
{
    public void Render() => Console.WriteLine("Windows button");
}

public class MacCheckbox : ICheckbox
{
    public void Render() => Console.WriteLine("Mac checkbox");
}

public interface IGUIFactory
{
    IButton CreateButton();
    ICheckbox CreateCheckbox();
}

public class WindowsFactory : IGUIFactory
```

```
{
    public IButton CreateButton() => new WindowsButton();
    public ICheckbox CreateCheckbox() => new MacCheckbox();
}

// Использование:
IGUIFactory factory = new WindowsFactory();
factory.CreateButton().Render();   // "Windows button"
factory.CreateCheckbox().Render(); // "Mac checkbox"
```

🏗️ Builder (Строитель)

Цель: Пошаговое создание сложных объектов

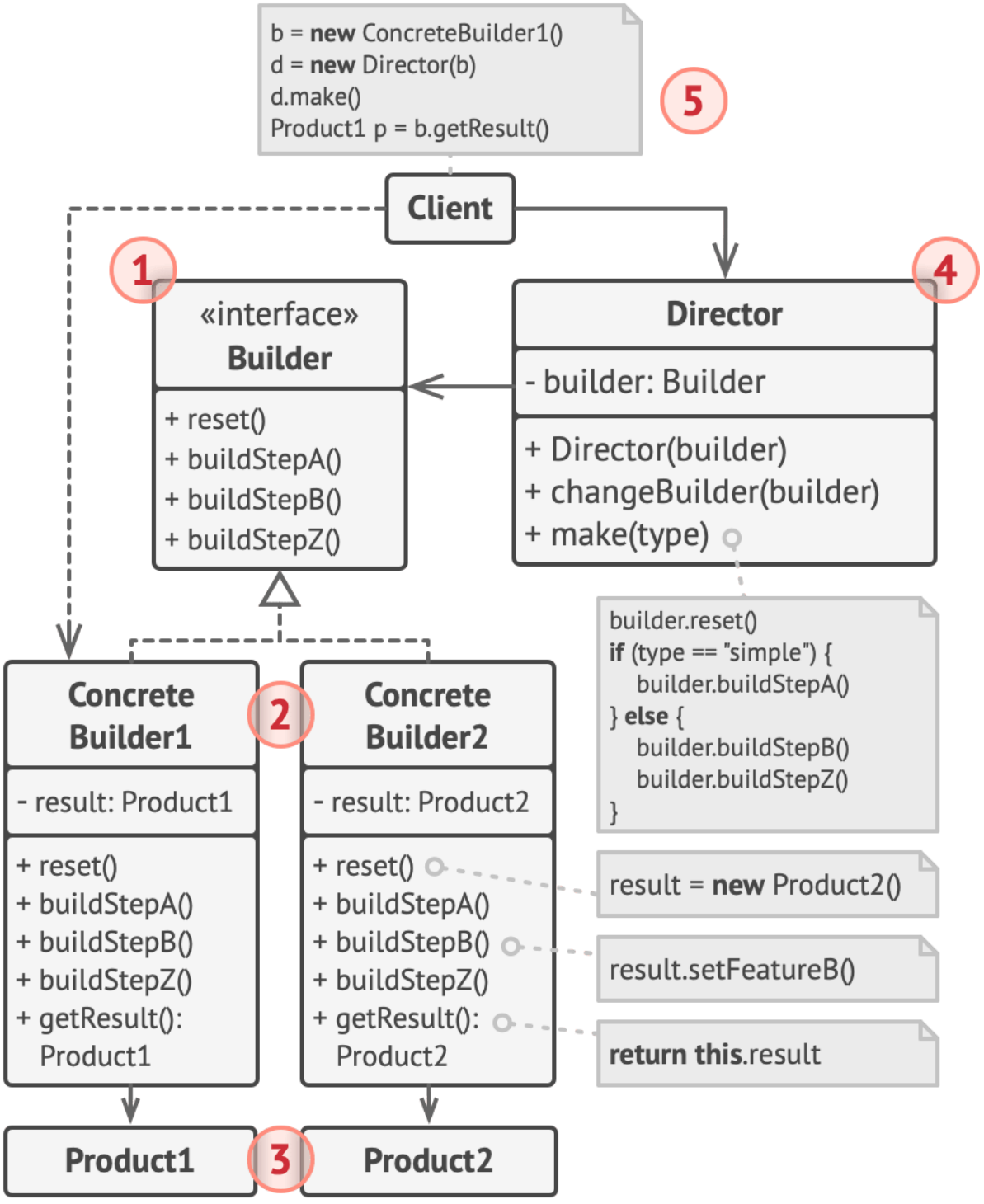
🔷 **Когда:** Многоэтапная конфигурация, Конструирование SQL-запросов, Генерация отчетов

👥 **Участники:**

- **Builder** — интерфейс строителя (`IPizzaBuilder`).
- **ConcreteBuilder** — конкретная реализация (`MargheritaBuilder`).
- **Director** — управляет процессом (`Cook`).
- **Product** — создаваемый объект (`Pizza`).

✅ **Плюсы:** Контроль процесса

❌ **Минусы:** Избыточность для простых объектов




```
public class Pizza
{
    public List<string> Toppings { get; } = new List<string>();
}

public interface IPizzaBuilder
```

```
{
    void AddCheese();
    void AddPepperoni();
    Pizza GetResult();
}

public class MargheritaBuilder : IPizzaBuilder
{
    private Pizza _pizza = new Pizza();

    public void AddCheese() => _pizza.Toppings.Add("Mozzarella");
    public void AddPepperoni() => _pizza.Toppings.Add("Pepperoni");
    public Pizza GetResult() => _pizza;
}

public class Cook
{
    public Pizza MakePizza(IPizzaBuilder builder)
    {
        builder.AddCheese();
        builder.AddPepperoni();
        return builder.GetResult();
    }
}

// Использование:
var builder = new MargheritaBuilder();
var pizza = new Cook().MakePizza(builder);
Console.WriteLine(string.Join(", ", pizza.Toppings)); // "Mozzarella, Pepper
oni"
```

🧬 Prototype (Прототип)

Цель: Клонирование объектов

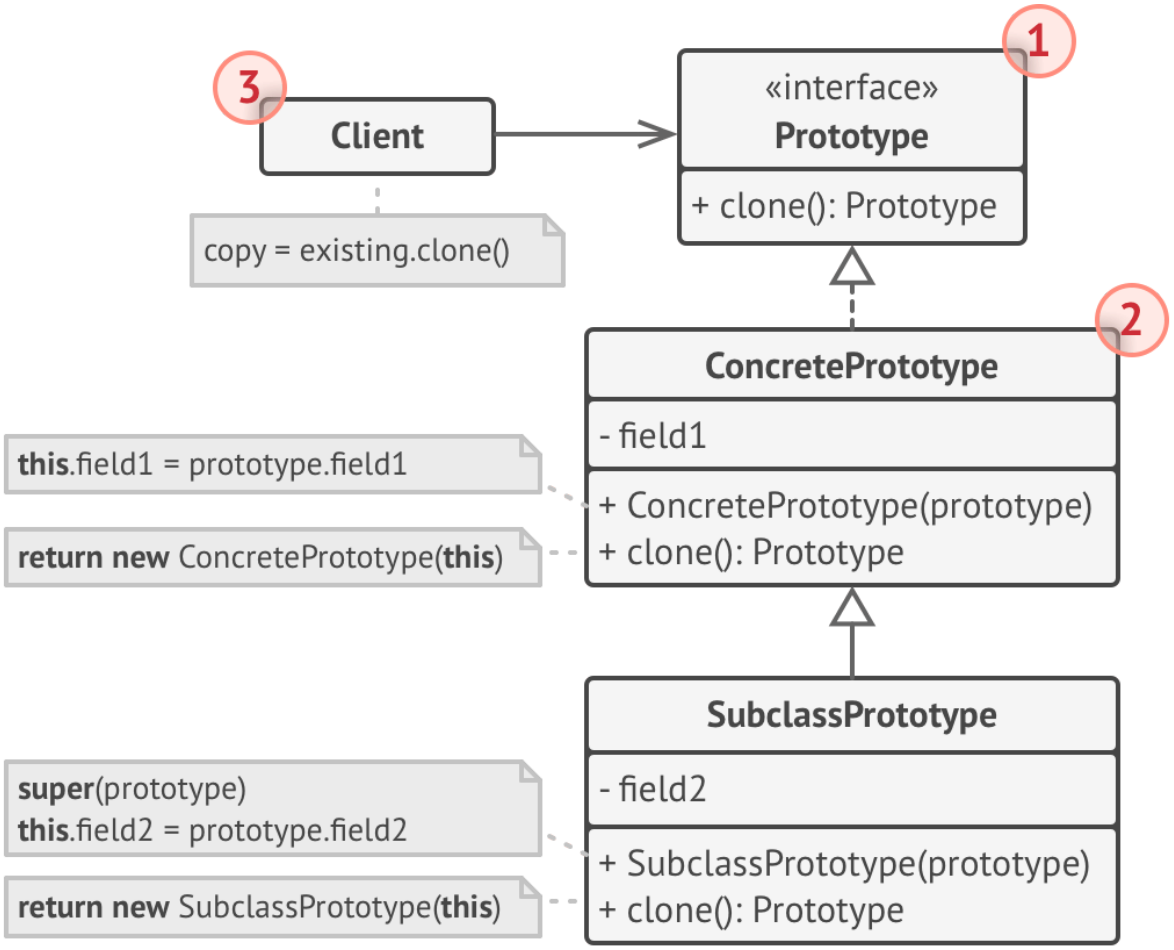
🔷 **Когда:** Копирование объектов без зависимости от их классов (Дорогое создание объектов).

👥 **Участники:**

- **Prototype** — интерфейс/класс с методом `Clone()` (`ISheep`).
- **ConcretePrototype** — конкретный объект для клонирования (`Sheep`).

✅ **Плюсы:** Быстрое копирование

❌ **Минусы:** Проблемы с глубоким копированием




```
public class Sheep : ICloneable
{
    public string Name { get; set; }
    public Sheep(string name) => Name = name;

    public object Clone() => new Sheep(Name); // Глубокое копирование
}

// Использование:
var original = new Sheep("Dolly");
```

```
var clone = (Sheep)original.Clone(); // Не зависит от конструктора Sheep!
Console.WriteLine(clone.Name); // "Dolly"
```

📊 Сравнение в одной таблице

Паттерн	Лучший случай использования	Главное преимущество
Singleton	Глобальные ресурсы	Единственный доступ
Factory	Неизвестный тип объекта	Гибкость
Abstract Factory	Связанные объекты	Совместимость
Builder	Сложные объекты	Пошаговое построение
Prototype	Копирование дорогих объектов	Быстрое создание

🛠️ Как выбрать паттерн?

1. **Singleton** → Когда нужен **единственный экземпляр** класса на всю программу
2. **Factory Method** → Когда **класс заранее не знает**, объекты каких подклассов ему создавать
3. **Abstract Factory** → Когда нужно **создавать семейства связанных объектов**
4. **Builder** → Когда **процесс создания объекта сложный** (много шагов, вариаций)
5. **Prototype** → Когда **клонирование дешевле**, чем создание через `new`

💡

Как запомнить?

1. **Singleton** - как президент страны (один на всех)
2. **Factory** - фабрика игрушек (создает что нужно)
3. **Abstract Factory** - мебельный гарнитур (все предметы в одном стиле)
4. **Builder** - конструктор LEGO (собираем по частям)
5. **Prototype** - ксерокс (делаем копии)