C# Professional

Порождающие шаблоны проектирования

otus.ru

Проверить, идет ли запись

Меня хорошо видно && слышно?

Ставим “+”, если все хорошо
“-”, если есть проблемы

Тема вебинара

Порождающие шаблоны проектирования

Михаил Дмитриев

Ведущий программист НИПК Электрон

Разрабатываю и поддерживаю приложения для работы с радиологическими комплексами

https://t.me/sf321

Правила вебинара

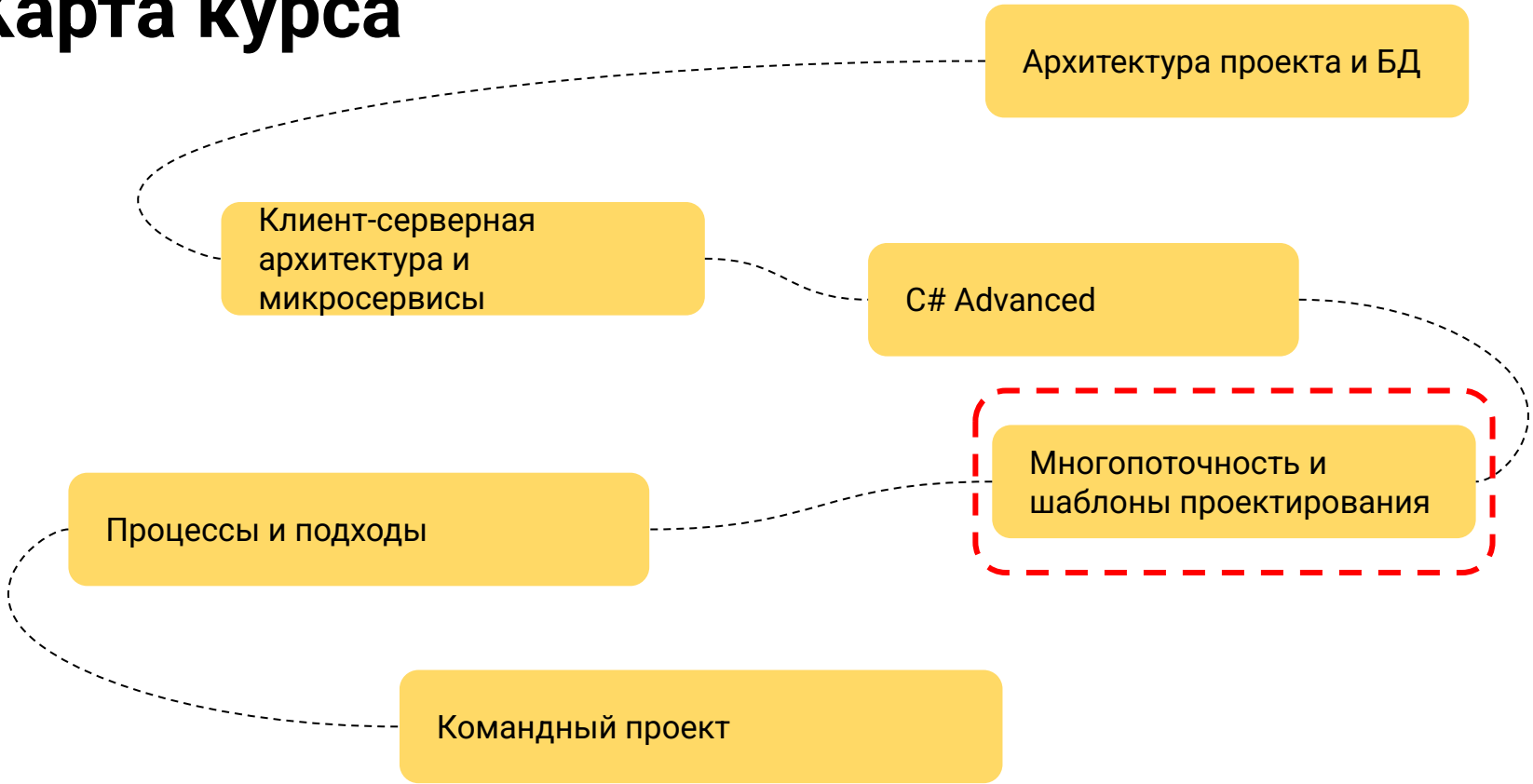
Активно
участвуем

Off-topic обсуждаем
в общем чате учебной группы
в telegram

Задаем вопрос
в чат или голосом

Вопросы вижу в чате,
могу ответить не сразу

Карта курса


Архитектура проекта и БД
Клиент-серверная архитектура и микросервисы
C# Advanced
Многопоточность и шаблоны проектирования
Процессы и подходы
Командный проект

Маршрут вебинара

- Знакомство
- Немного теории
- Проблема сложности
- Порождающие шаблоны проектирования
- Примеры
- Рефлексия

Цели вебинара

К концу занятия вы сможете

1. Расширить знания о применении порождающих шаблонов проектирования при разработке ПО
2. Применять на практике порождающие шаблоны проектирования
3. Глубже понимать влияние дизайна на эффективность разработки

Смысл

Зачем вам это уметь

1. Управлять сложностью разрабатываемых вами решений
2. Снижать издержки на поддержку вашего ПО
3. Быть в контексте с участниками вашей команды

Тестирование

Управление сложностью разработки ПО на этапе проектирования архитектуры

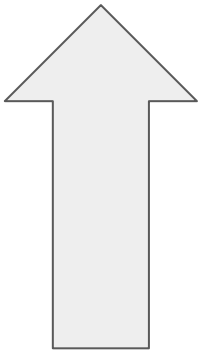
Проблема сложности при разработке ПО

Количество ресурсов , связей и компонентов, которые требуются для решения какой-либо задачи растет нелинейно по мере роста проекта.

Некоторые виды “сложности”:

- алгоритмическая
- трудоемкость разработки
- информационная
- сложность тестирования

Уровни архитектуры



- инфраструктура (система) (паттерны межсервисного взаимодействия)
- компоненты системы (сервисы) (+ solid / patterns)
- модули проекта (+ solid / patterns)
- классы (+ solid / patterns)
- методы (+ clean code)

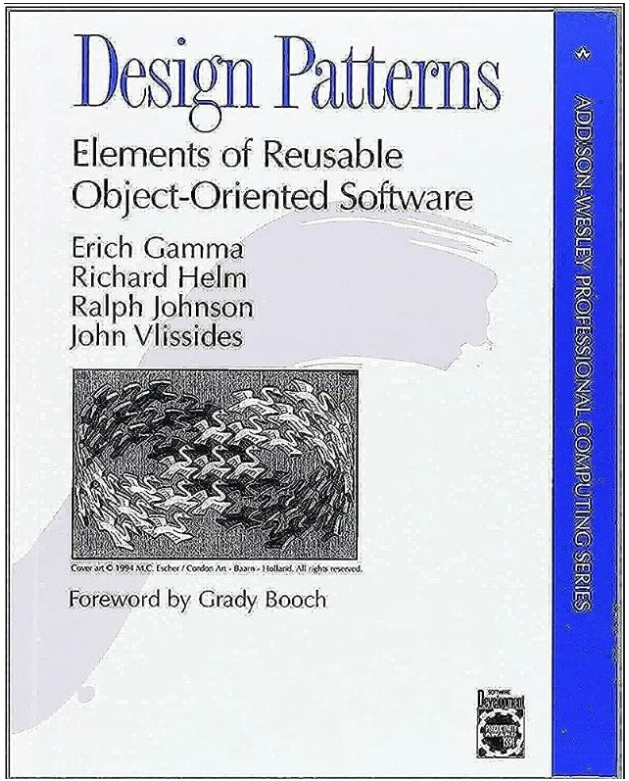
SOLID

SOLID – 5 основных принципов объектно-ориентированного программирования и проектирования:

- Принцип единственной ответственности (single responsibility principle, SRP)
- Принцип открытости/закрытости (open-closed principle, OCP)
- Принцип подстановки Лисков (Liskov substitution principle, LSP)
- Принцип разделения интерфейса (interface segregation principle, ISP)
- Принцип инверсии зависимостей (dependency inversion principle, DIP)

GoF Паттерны объектно-ориентированного проектирования

Design Patterns: Elements of Reusable Object-Oriented Software (1994)
Книгу написали
Эрих Гамма,
Ричард Хелм,
Ральф Джонсон и
Джон Влиссидес



Классификация паттернов ООП по GoF

- Порождающие паттерны дают возможность выполнять инициализацию объектов наиболее удобным и оптимальным способом.

- Структурные паттерны описывают взаимоотношения между различными классами или объектами, позволяя им совместно реализовывать поставленную задачу.

- Поведенческие паттерны позволяют грамотно организовать связь между сущностями для оптимизации и упрощения их взаимодействия.

Цель / Уровень	Порождающие паттерны	Структурные паттерны	Паттерны поведения
Класс	Фабричный метод (135)	Адаптер (171)	Интерпретатор (287) Шаблонный метод (373)
Объект	Абстрактная фабрика (113) Одиночка (157) Прототип (146) Строитель (124)	Адаптер (171) Декоратор (209) Заместитель (246) Компоновщик (196) Мост (184) Приспособленец (231) Фасад (221)	Итератор (302) Команда (275) Наблюдатель (339) Посетитель (379) Посредник (319) Состояние (352) Стратегия (362) Хранитель (330) Цепочка обязанностей (263)

Порождающие паттерны проектирования

Порождающие паттерны проектирования

Порождающие паттерны отвечают за удобное и безопасное создание новых объектов или даже целых семейств объектов.

При этом могут использоваться следующие механизмы:

- Наследование, когда базовый класс определяет интерфейс, а подклассы - реализацию. Структуры на основе наследования получаются статичными.

- Композиция, когда структуры строятся путем объединения объектов некоторых классов. Композиция позволяет получать структуры, которые можно изменять во время выполнения (рекомендуемый метод)

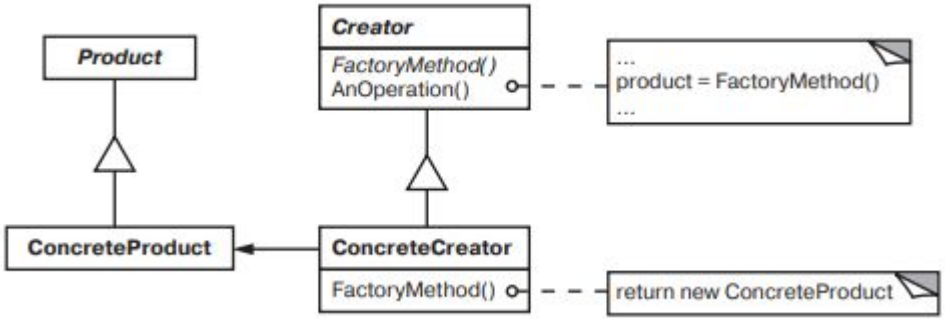
Порождающие паттерны проектирования

Список порождающих шаблонов проектирования:

- Фабричный метод (Fabric Method)(Virtual Constructor)
- Абстрактная фабрика (Abstract Factory)(Kit)
- Строитель(Builder)
- Прототип (Prototype)
- Одиночка (Singleton)

Фабричный метод (Factory Method)(Virtual Constructor)

Определяет интерфейс для создания объекта, но оставляет подклассам решение о том, экземпляры какого класса должны создаваться. Фабричный метод позволяет классу делегировать создание экземпляров подклассам.

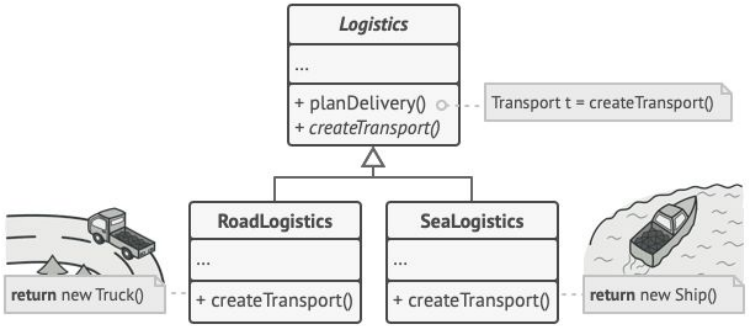


Основные условия для применения паттерна фабричный метод:

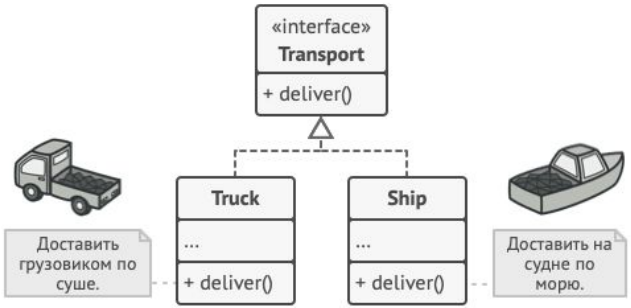
- классу заранее неизвестно, объекты каких классов ему нужно создавать;
- класс спроектирован так, чтобы объекты, которые он создает, определялись подклассами;
- класс делегирует свои обязанности одному из нескольких вспомогательных подклассов, и вам нужно локализовать информацию о том, какой класс принимает эти обязанности на себя

Фабричный метод (Factory Method)(Virtual Constructor)

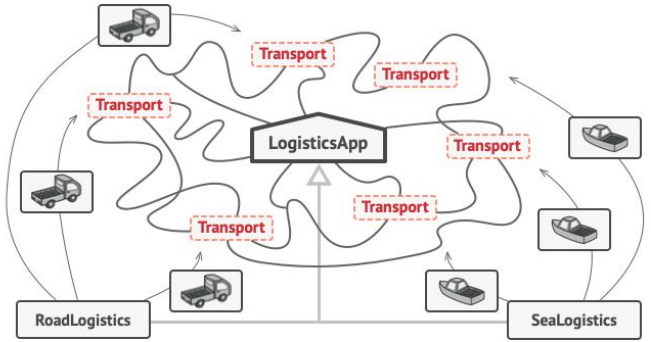
Добавить новый класс не так-то просто, если весь код уже завязан на конкретные классы.



Подклассы могут изменять класс создаваемых объектов.



Все объекты-продукты должны иметь общий интерфейс.



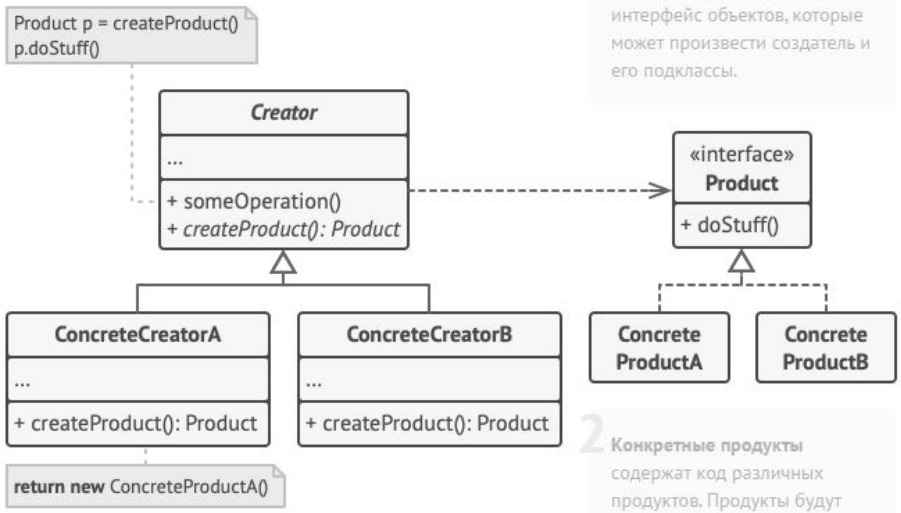
Пока все продукты реализуют общий интерфейс, их объекты можно взаимозаменять в клиентском коде.

Фабричный метод (Factory Method)(Virtual Constructor)

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.



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.

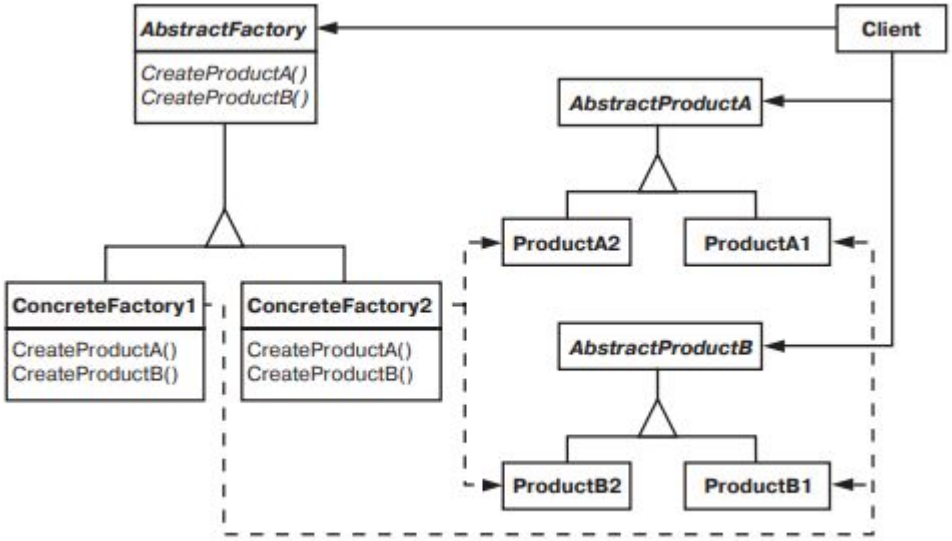
4 **Конкретные создатели** по-своему реализуют фабричный метод, производя те или иные конкретные продукты.

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

LIVE

Абстрактная фабрика (Abstract Factory)(Kit)

Предоставляет интерфейс для создания семейств взаимосвязанных или взаимозависимых объектов, не специфицируя их конкретных классов.



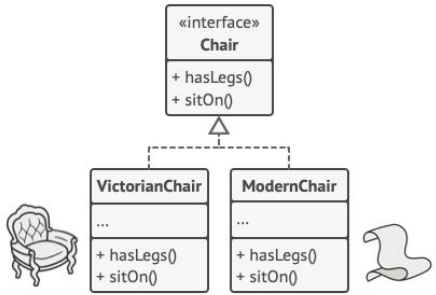
Основные условия применения паттерна:

- система не должна зависеть от того, как создаются, компонуются и представляются входящие в нее объекты;
- система должна настраиваться одним из семейств объектов;
- входящие в семейство взаимосвязанные объекты спроектированы для совместной работы, и вы должны обеспечить выполнение этого ограничения;
- вы хотите предоставить библиотеку объектов, раскрывая только их интерфейсы, но не реализацию.

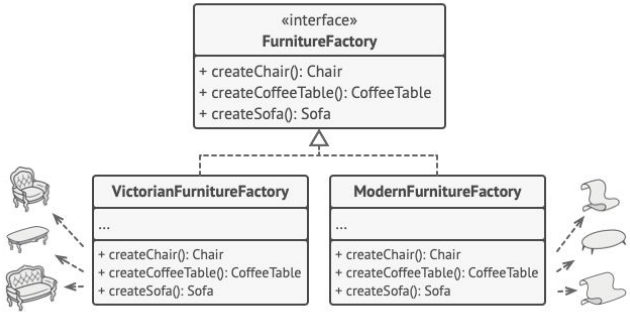
Абстрактная фабрика (Abstract Factory)(Kit)

Семейства продуктов и их вариации.

Клиенты расстраиваются, если получают несочетающиеся продукты.



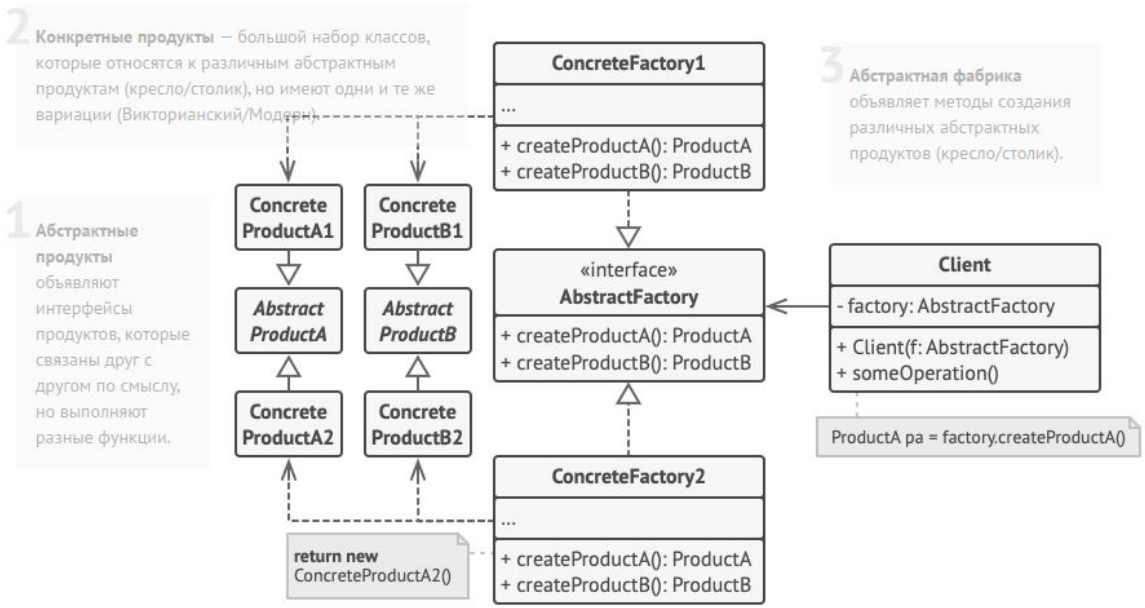
Все вариации одного и того же объекта должны жить в одной иерархии классов.



Конкретные фабрики соответствуют определённой вариации семейства продуктов.

Для клиентского кода должно быть безразлично, с какой фабрикой работать.

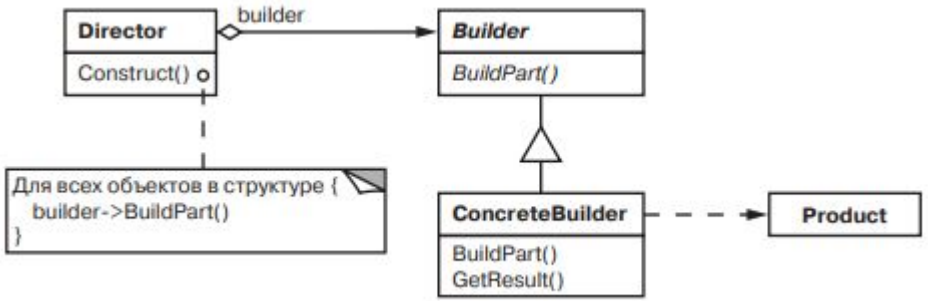
Абстрактная фабрика (Abstract Factory)(Kit)



LIVE

Строитель(Builder)

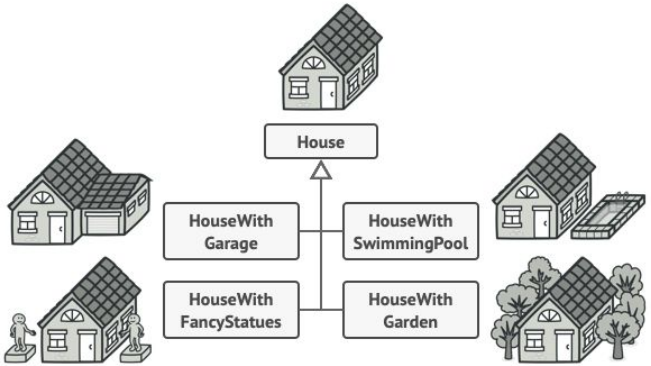
Отделяет конструирование сложного объекта от его представления, так что в результате одного и того же процесса конструирования могут получаться разные представления.



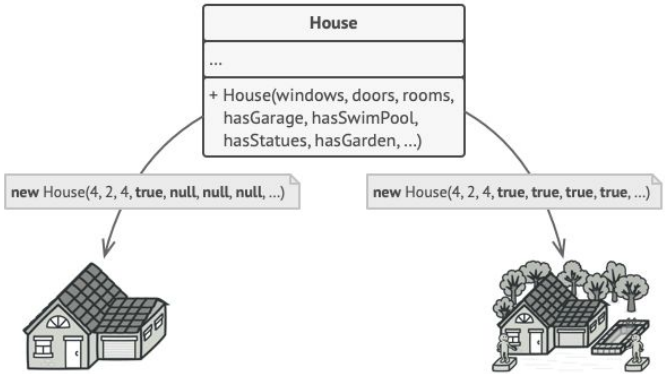
Основные условия для применения паттерна:

- алгоритм создания сложного объекта не должен зависеть от того, из каких частей состоит объект и как они стыкуются между собой;
- процесс конструирования должен обеспечивать различные представления конструируемого объекта.

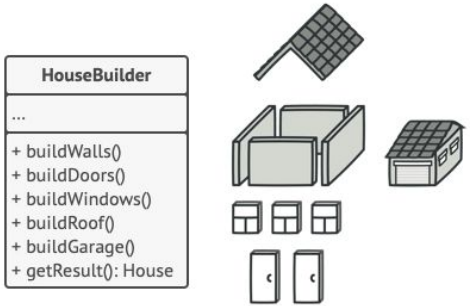
Строитель(Builder)



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.

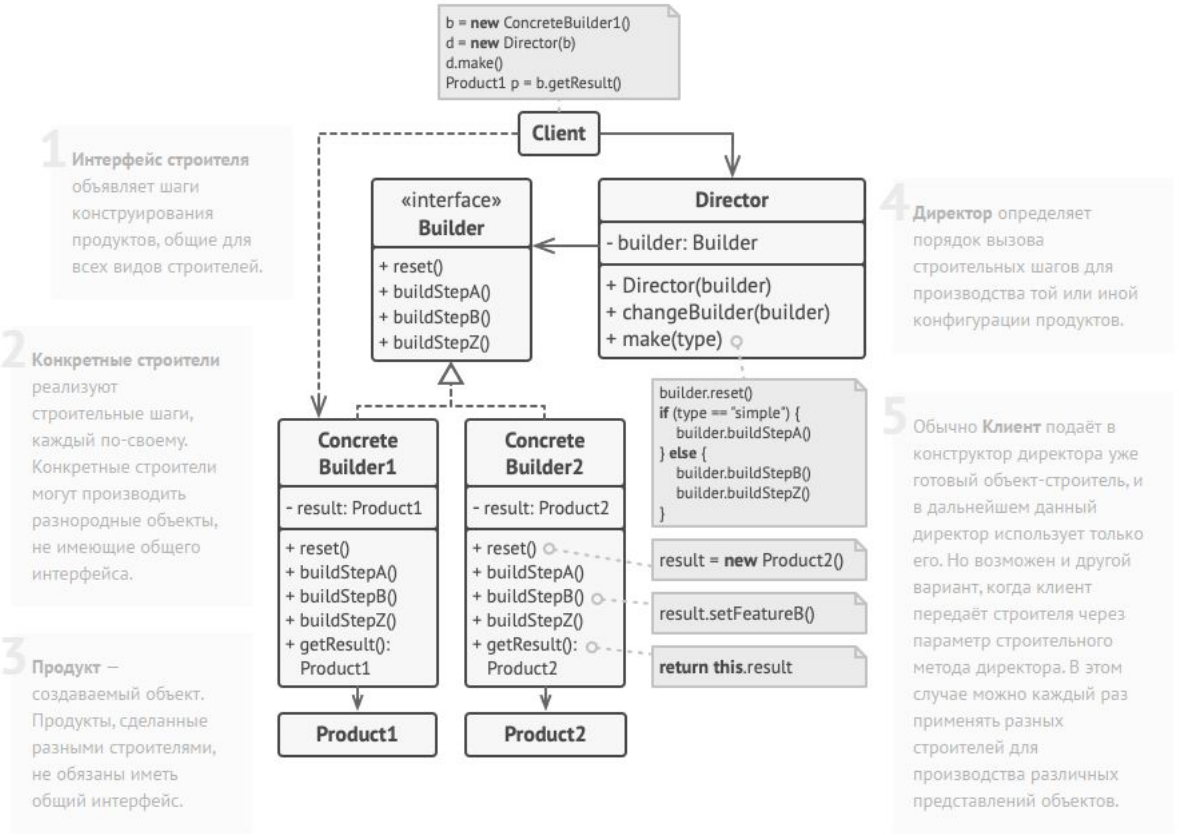


Строитель позволяет создавать сложные объекты пошагово. Промежуточный результат всегда остаётся защищён.

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

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

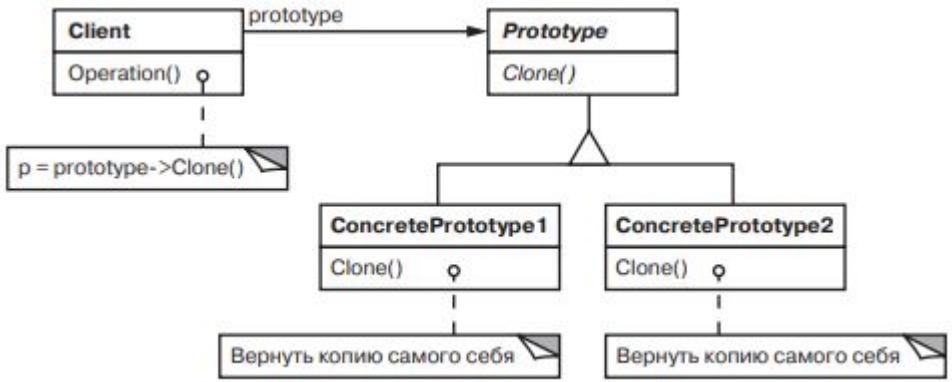
Строитель(Builder)



LIVE

Прототип (Клон, Prototype)

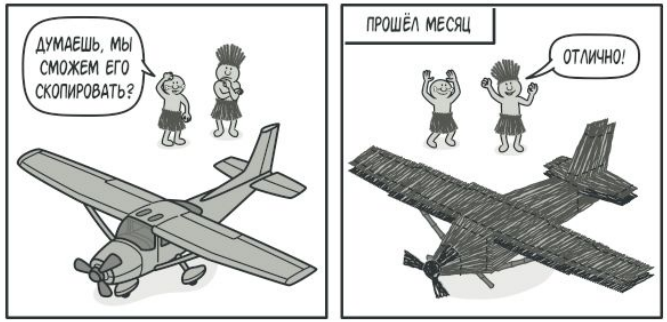
Задает виды создаваемых объектов с помощью экземпляра-прототипа и создает новые объекты путем копирования этого прототипа.



Система не должна зависеть от того, как в ней создаются, компонуются и представляются продукты; кроме того:

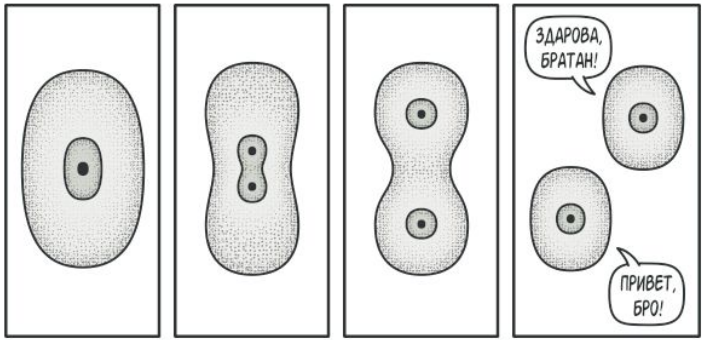
- классы для создания экземпляров определяются во время выполнения, например с помощью динамической загрузки; или
- для того чтобы избежать построения иерархий классов или фабрик, параллельных иерархии классов продуктов; или
- экземпляры класса могут находиться в одном из не очень большого числа различных состояний. Может быть удобнее установить соответствующее число прототипов и клонировать их, а не создавать экземпляр каждый раз вручную в подходящем состоянии.

Прототип (Клон, Prototype)



*Копирование «извне» **не всегда** возможно в реальности.*

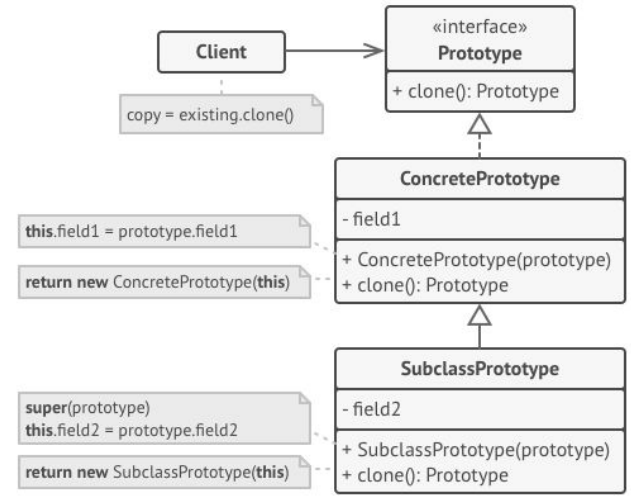
Предварительно заготовленные прототипы могут стать заменой подклассам.



Пример деления клетки.

Прототип (Клон, Prototype)

Базовая реализация

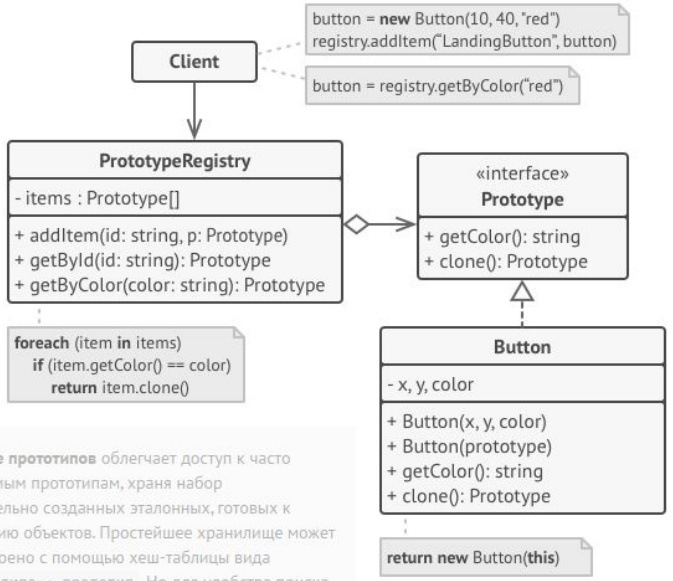


3 **Клиент** создаёт копию объекта, обращаясь к нему через общий интерфейс прототипов.

1 **Интерфейс прототипов** описывает операции клонирования. В большинстве случаев — это единственный метод `clone`.

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. Например, клонирование связанных объектов, распутывание рекурсивных зависимостей и прочее.

Реализация с общим хранилищем прототипов

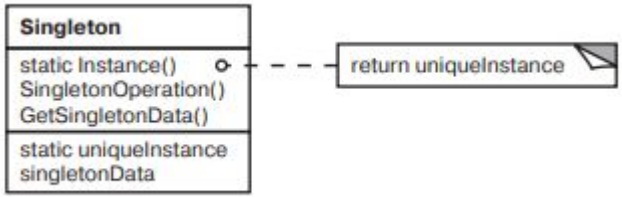


1 **Хранилище прототипов** облегчает доступ к часто используемым прототипам, храня набор предварительно созданных эталонных, готовых к копированию объектов. Простейшее хранилище может быть построено с помощью хеш-таблицы вида `имя-прототипа → прототип`. Но для удобства поиска прототипы можно маркировать и другими критериями, а не только условным именем.

LIVE

Одиночка (Singleton)

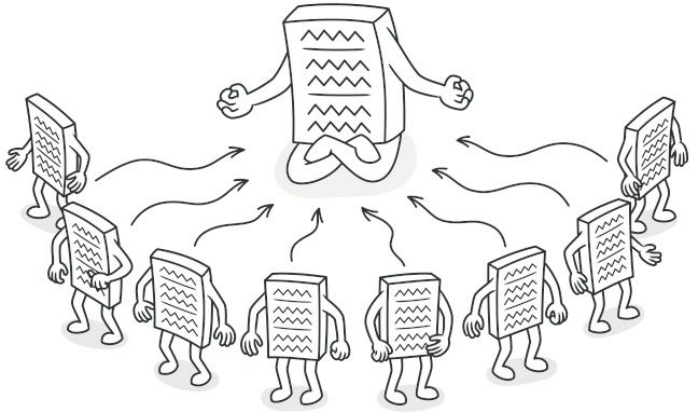
Гарантирует, что у класса существует только один экземпляр, и предоставляет к нему глобальную точку доступа.

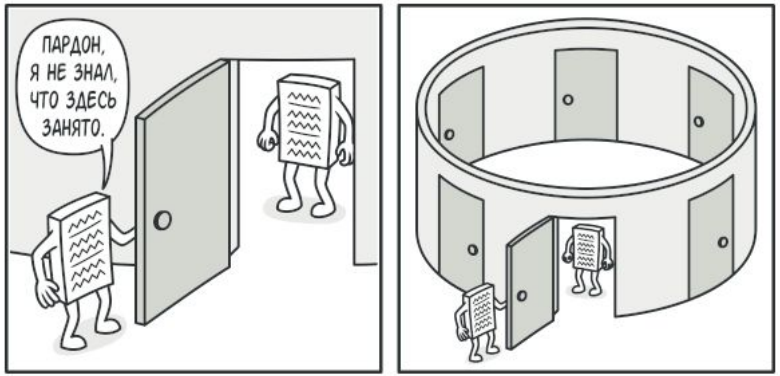


Основные условия для применения паттерна :

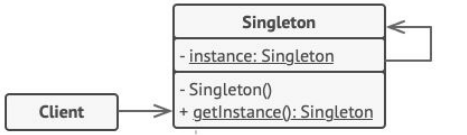
- должен существовать ровно один экземпляр некоторого класса, к которому может обратиться любой клиент через известную точку доступа;
- единственный экземпляр должен расширяться путем порождения подклассов, а клиенты должны иметь возможность работать с расширенным экземпляром без модификации своего кода.

Одиночка (Singleton)





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



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
```

LIVE

Список материалов для изучения

1. Гради Буч “Объектно-ориентированный анализ и проектирование”
2. Фредерик Брукс “Мифический человеко месяц”
3. Влиссидес Джон, Хелм Ричард “Паттерны объектно-ориентированного проектирования”
4. Крейг Ларман «Применение UML и шаблонов проектирования»
5. Сергей Тепляков “Паттерны проектирования на платформе .NET”
6. https://refactoring.guru/ru/design-patterns/creational-patterns
7. https://csharpindepth.com/Articles/Singleton
8. https://habr.com/ru/articles/210288/
9. https://github.com/kamranahmedse/design-patterns-for-humans
10. https://github.com/nemanjarogic/DesignPatternsLibrary
11. https://github.com/nemanjarogic/DesignPatternsLibrary/tree/main/src/CreationalPatterns/Builder/CustomSandwichBuilder
12. https://gitlab.com/otus-demo/professional/creational-design-patterns

Вопросы?

Ставим “+”,
если вопросы есть

Ставим “–”,
если вопросов нет

Рефлексия

Заполните, пожалуйста, опрос о занятии по ссылке в чате

Спасибо за внимание!

Приходите на следующие вебинары

Михаил Дмитриев

Ведущий программист НИПК Электрон

Разрабатываю и поддерживаю приложения для работы с радиологическими комплексами

https://t.me/sf321