Курс лекций:

Язык программирования

C++Лекция 12: Шаблоны проектирования: Порождающие.

Преподаватель: Пысин Максим Дмитриевич, Краснов Дмитрий Олегович, аспиранты кафедры ИКТ.

Что запомнилось?

- **Что такое парадигма программирования? Какие парадигмы программирования есть? Какую изучаем мы?**
- **Абстрагирование и абстракция что это и для чего?**
- **Что такое класс? Объект? Наследования? Полиморфизм? Инкапсуляции?**
- **Что такое лямбда функция? Что такое захват[проброс] переменных?**
- **Что такое абстрактный класс? Для чего он нужен?**
- **Что такое итератор и для чего он используется?**
- **Что такое последовательные контейнеры? Что такое ассоциативные контейнеры? На чем основаны?**
- **С чем в основном работает стандартная библиотека алгоритмов?**
- **Что такое поток ввода вывода? Примеры?**
- **Что такое исключение?**
- **Можно ли порождать исключения самостоятельно?**
- **Какая конструкция отвечает за обработку исключений?**
- **Какая проблема есть у исключений? Вариант ее решения в C++?**

Паттерны проектирования

Шаблон проектирования, или **паттерн**, в разработке программного обеспечения — **повторяемая архитектурная конструкция**, представляющая собой решение проблемы проектирования, в рамках некоторого часто возникающего контекста.

Шаблоны проектирования — это руководства по решению повторяющихся проблем. Это не классы, пакеты или библиотеки, которые можно было бы подключить к вашему приложению и сидеть в ожидании чуда. Они скорее являются методиками, как решать определенные проблемы в определенных ситуациях.

Будьте осторожны

Шаблоны проектирования не являются решением всех ваших проблем;

не пытайтесь использовать их в обязательном порядке — это может привести к негативным последствиям. Шаблоны — это подходы к решению проблем, а не решения для поиска проблем.

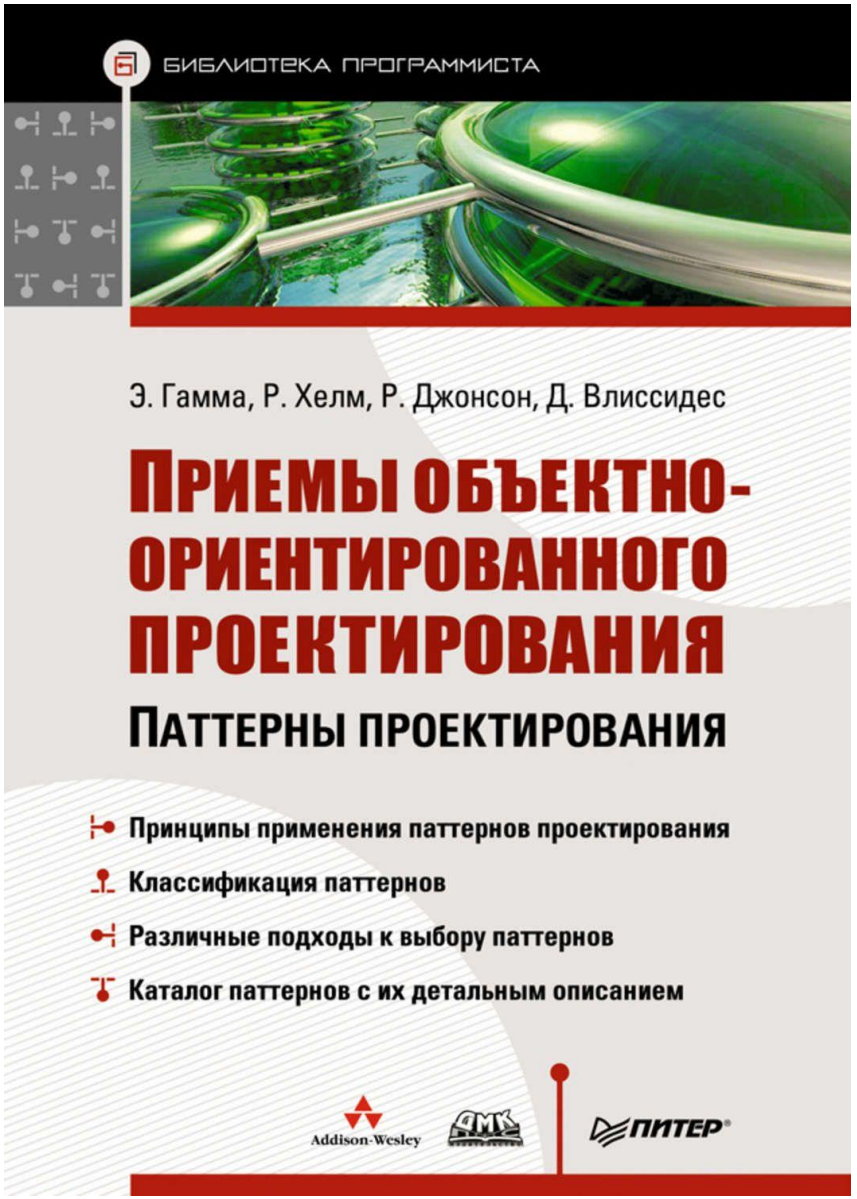
если их правильно использовать в нужных местах, то они могут стать спасением, а иначе могут привести к ужасному беспорядку.

https://refactoring.guru/ru/design-patterns/catalog

Книга

Книга 1994 года о программной инженерии, описывающая шаблоны проектирования программного обеспечения. Авторами книги, которых прозвали «Бандой четырёх»[1], являются Эрих Гамма, Ричард Хелм, Ральф Джонсон (англ.)русск., Джон Влиссидес. Предисловие написал Гради Буч.

Книга состоит из двух частей, в первых двух главах рассказывается о возможностях и недостатках объектно-ориентированного программирования, а во второй части описаны 23 классических шаблона проектирования. Примеры в книге написаны на языках программирования C++ и Smalltalk.



Отношения между классами

- **агрегация** (aggregation) — описывает связь **«часть»–«целое»**, в котором «часть» может существовать отдельно от «целого». Ромб указывается со стороны «целого».

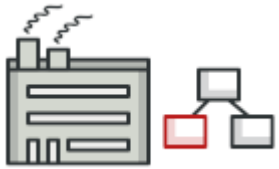
- **композиция** (composition) — подвид агрегации, в которой **«части» не могут существовать отдельно от «целого»**.

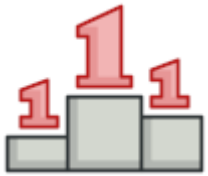
- **зависимость** (dependency) — **изменение в одной** сущности (независимой) может **влиять на состояние или поведение другой** сущности (зависимой). Со стороны стрелки указывается независимая сущность.

- **обобщение** (generalization) — **отношение наследования или реализации интерфейса**. Со стороны стрелки находится суперкласс или интерфейс.

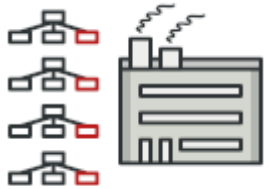
Порождающие паттерны

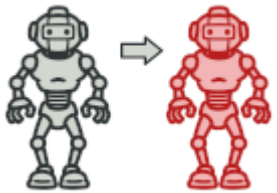
Порождающие шаблоны — шаблоны проектирования, которые **абстрагируют процесс создания объекта класса**.
Они позволяют сделать систему независимой от способа создания, композиции и представления объектов. Шаблон, порождающий классы, использует наследование, чтобы изменять наследуемый класс, а шаблон, порождающий объекты, делегирует процесс создания другому объекту.
Порождающие шаблоны это шаблоны, которые предназначены для создания экземпляра объекта или группы связанных объектов





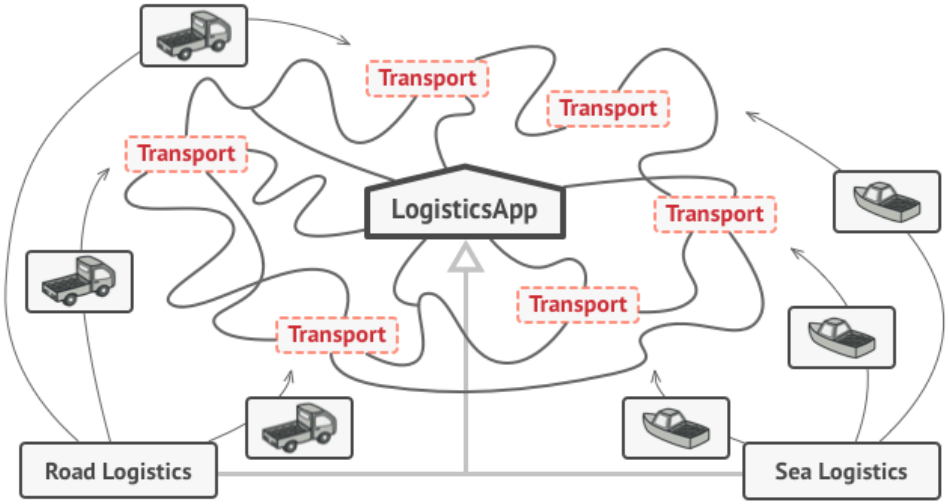
- Фабричный метод
- Абстрактная фабрика
- Строитель
- Прототип
- Одиночка





https://refactoring.guru/ru/design-patterns/creational-patterns

Фабричный метод



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

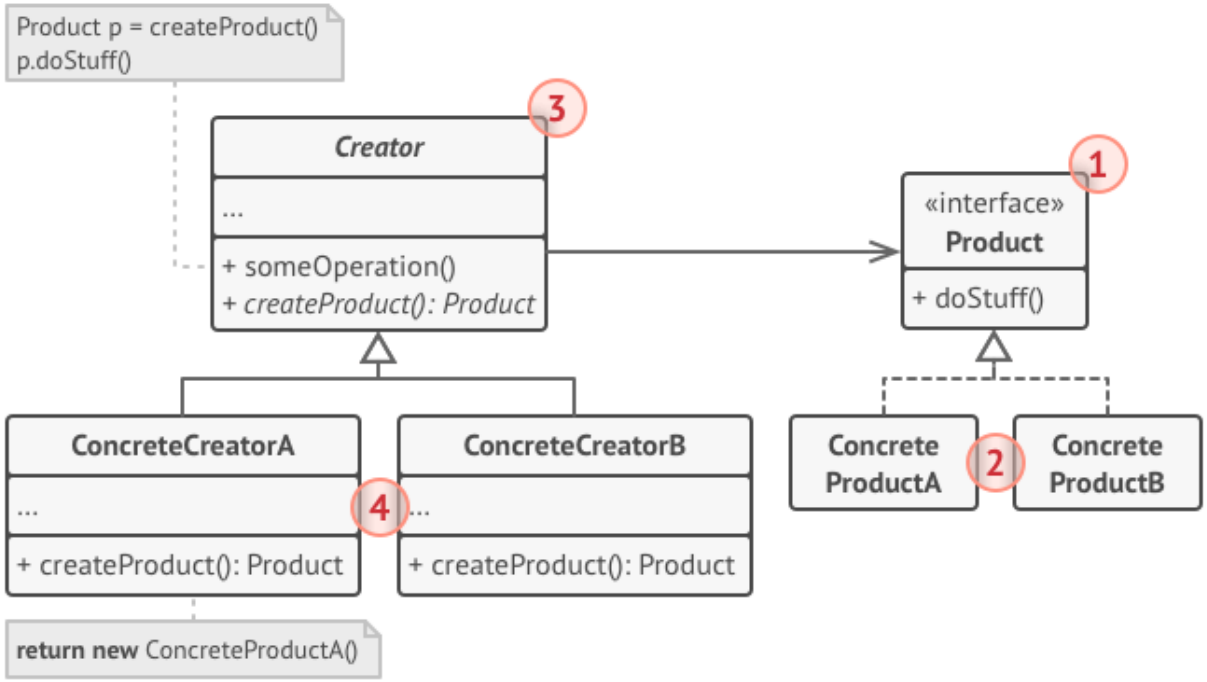
Представьте, что вы создаёте программу управления грузовыми перевозками. Сперва вы рассчитываете перевозить товары только на автомобилях. Поэтому весь ваш код работает с объектами класса Грузовик.

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

Фабричный метод: Структура

Продукт определяет общий интерфейс объектов, которые может произвести создатель и его подклассы.
Конкретные продукты содержат код различных продуктов. Продукты будут отличаться реализацией, но интерфейс у них будет общий.
Создатель объявляет фабричный метод, который должен возвращать новые объекты продуктов. Важно, чтобы тип результата совпадал с общим интерфейсом продуктов. Зачастую фабричный метод объявляют абстрактным.
Несмотря на название, важно понимать, что создание продуктов не является единственной функцией создателя.

Конкретные создатели по-своему реализуют фабричный метод, производя те или иные конкретные продукты. Фабричный метод не обязан всё время



Фабричный метод: Использование

Плюсы.

- 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.

Минусы

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

Когда использовать.

- Когда заранее неизвестны типы и зависимости объектов, с которыми должен работать ваш код.
- Когда вы хотите дать возможность пользователям расширять части вашего фреймворка или библиотеки.
- Когда вы хотите экономить системные ресурсы, повторно используя уже созданные объекты, вместо порождения новых.

Абстрактная фабрика

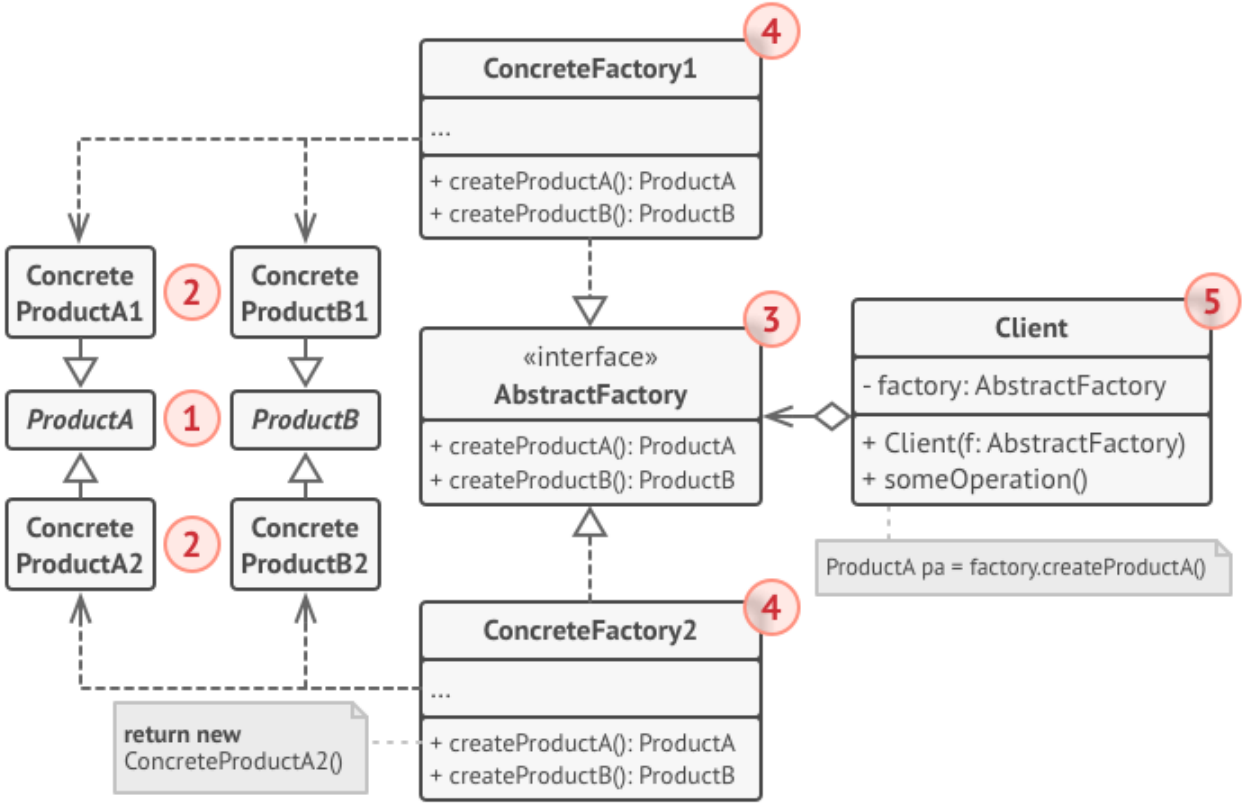
Абстрактная фабрика — это порождающий паттерн проектирования, который позволяет **создавать семейства связанных объектов**, не привязываясь к конкретным классам создаваемых объектов.

Представьте, что вы пишете симулятор мебельного магазина. Ваш код содержит:
Семейство зависимых продуктов. Скажем, Кресло + Диван + Столик.
Несколько вариаций этого семейства. Например, продукты Кресло, Диван и Столик представлены в трёх разных стилях: Ар-деко, Викторианском и Модерне.
Вам нужен такой способ создавать объекты продуктов, чтобы они сочетались с другими продуктами того же семейства. Это важно, так как клиенты расстраиваются, если получают несочетающуюся мебель.

Абстрактная фабрика: 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, но имеют одни и те же вариации.
3. Абстрактная фабрика объявляет методы создания различных абстрактных продуктов.
4.Конкретные фабрики относятся каждая к своей вариации продуктов и реализуют методы абстрактной фабрики, позволяя создавать все продукты определённой вариации.

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



Абстрактная фабрика: Использование

Плюсы

- 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.

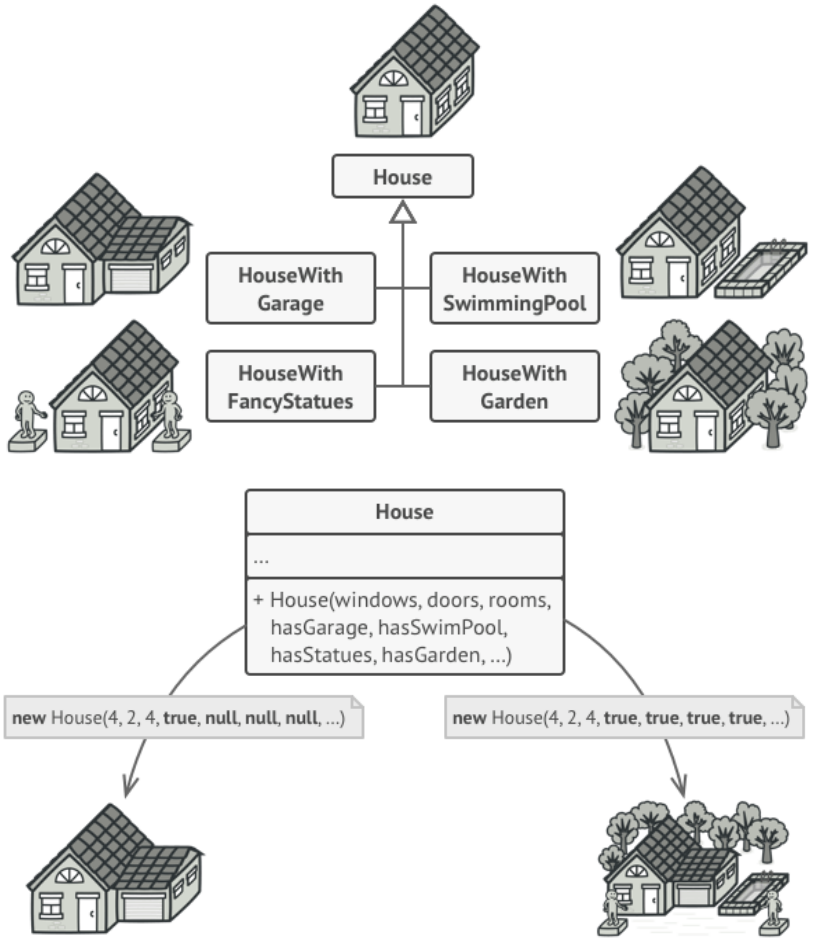
Минусы

- Реализует принцип открытости/закрытости.
- Усложняет код программы из-за введения множества дополнительных классов.
- Требует наличия всех типов продуктов в каждой вариации.

Когда использовать?

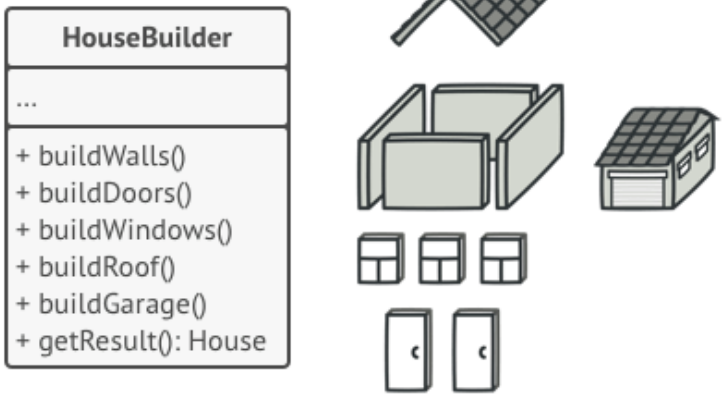
- Когда бизнес-логика программы должна работать с разными видами связанных друг с другом продуктов, не завися от конкретных классов продуктов.
- Когда в программе уже используется Фабричный метод, но очередные изменения предполагают введение новых типов продуктов.

Строитель

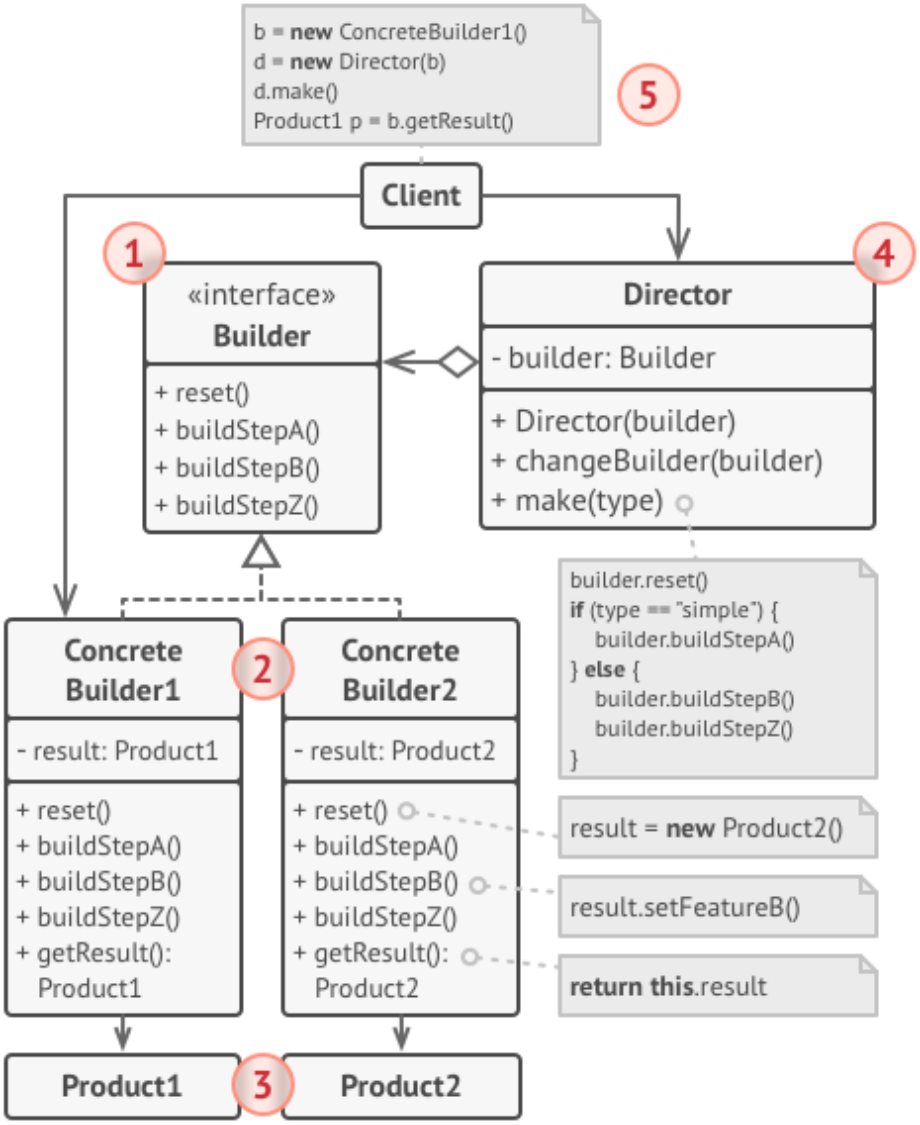


Строитель — это порождающий паттерн проектирования, который позволяет **создавать сложные объекты пошагово**. Строитель даёт возможность использовать один и тот же код строительства для получения разных представлений объектов.

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.



Строитель: Структура



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

2.Конкретные строители реализуют строительные шаги, каждый по-своему. Конкретные строители могут производить разнородные объекты, не имеющие общего интерфейса.

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. Обычно Клиент подаёт в конструктор директора уже готовый объект-строитель, и в дальнейшем данный директор использует только его. Но возможен и другой вариант, когда клиент передаёт строителя через параметр строительного метода директора.

Строитель: Использование

Плюсы

- 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.

Минусы

- Усложняет код программы из-за введения дополнительных классов.
- Клиент будет привязан к конкретным классам строителей, так как в интерфейсе строителя может не быть метода получения результата.

Когда применять?

- Когда вы хотите избавиться от «телескопического конструктора».
- Когда ваш код должен создавать разные представления какого-то объекта. Например, деревянные и железобетонные дома.
- Когда вам нужно собирать сложные составные объекты, например, деревья Компоновщика.

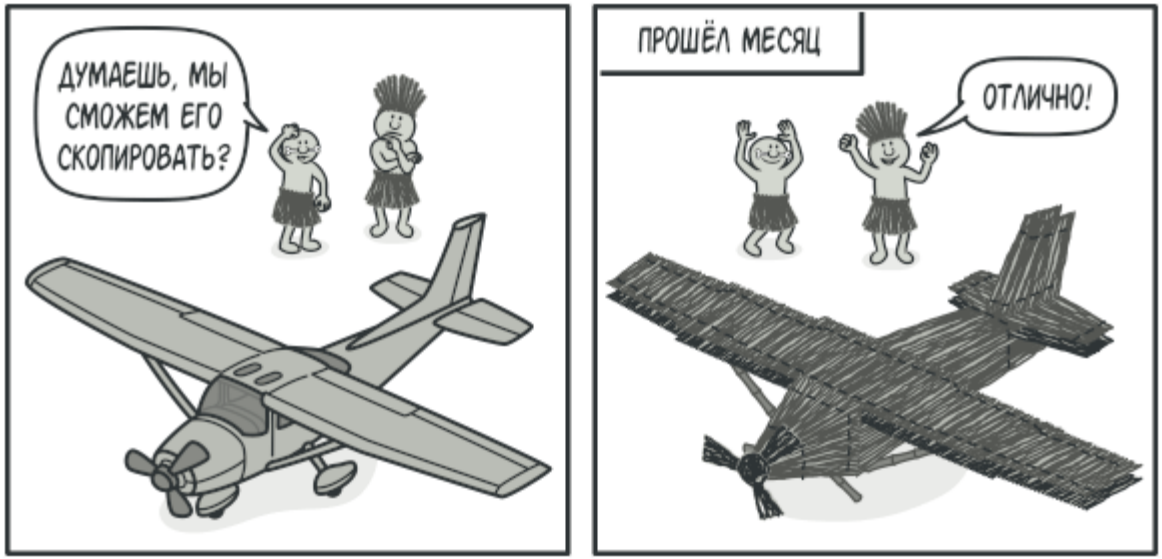
Прототип

Прототип — это порождающий паттерн проектирования, который **позволяет копировать объекты**, не вдаваясь в подробности их реализации.

У вас есть объект, который нужно скопировать. Как это сделать? Нужно создать пустой объект такого же класса, а затем поочерёдно скопировать значения всех полей из старого объекта в новый.

Прекрасно! Но есть нюанс. Не каждый объект удастся скопировать таким образом, ведь часть его состояния может быть приватной, а значит — недоступной для остального кода программы.

Но есть и другая проблема. Копирующий код станет зависим от классов копируемых объектов. Ведь, чтобы перебрать все поля объекта, нужно привязаться к его классу. Из-за этого вы не сможете копировать объекты, зная только их интерфейсы, а не конкретные классы.

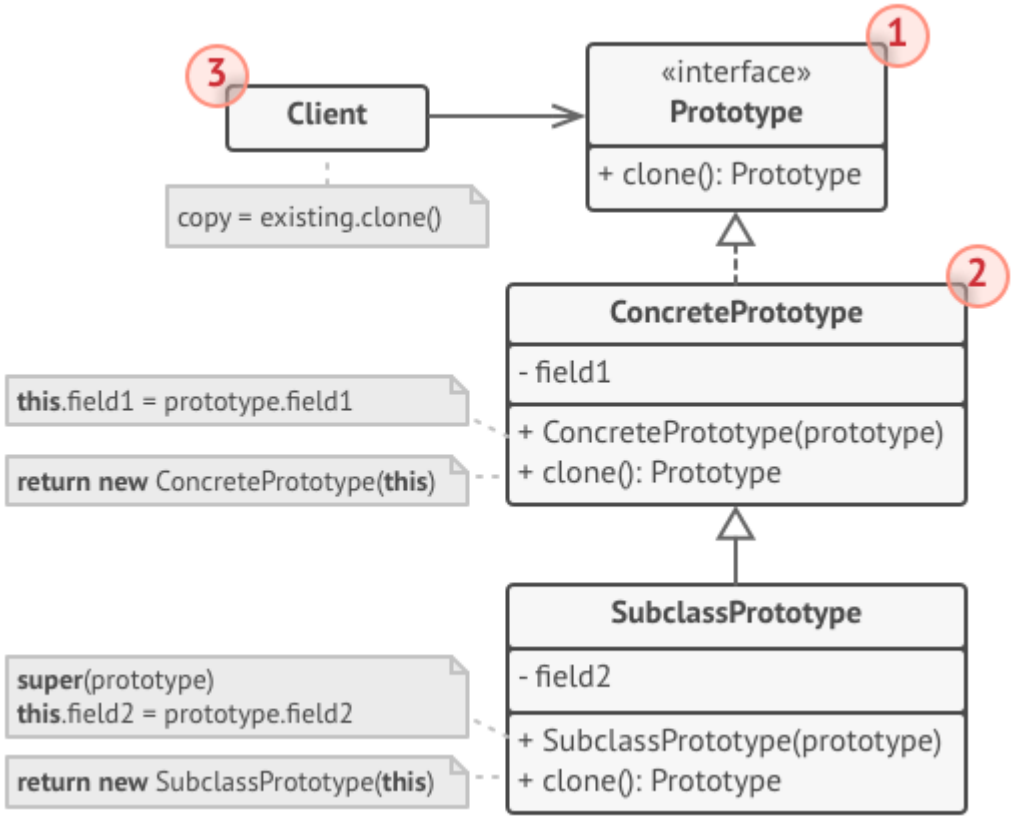


Прототип: Структура

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

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.

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



Прототип: Использование

Плюсы

- Позволяет клонировать объекты, не привязываясь к их конкретным классам.
- Меньше повторяющегося кода инициализации объектов.
- Ускоряет создание объектов.
- Альтернатива созданию подклассов для конструирования сложных объектов.

Минусы

- Сложно клонировать составные объекты, имеющие ссылки на другие объекты.

Когда использовать?

- 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.

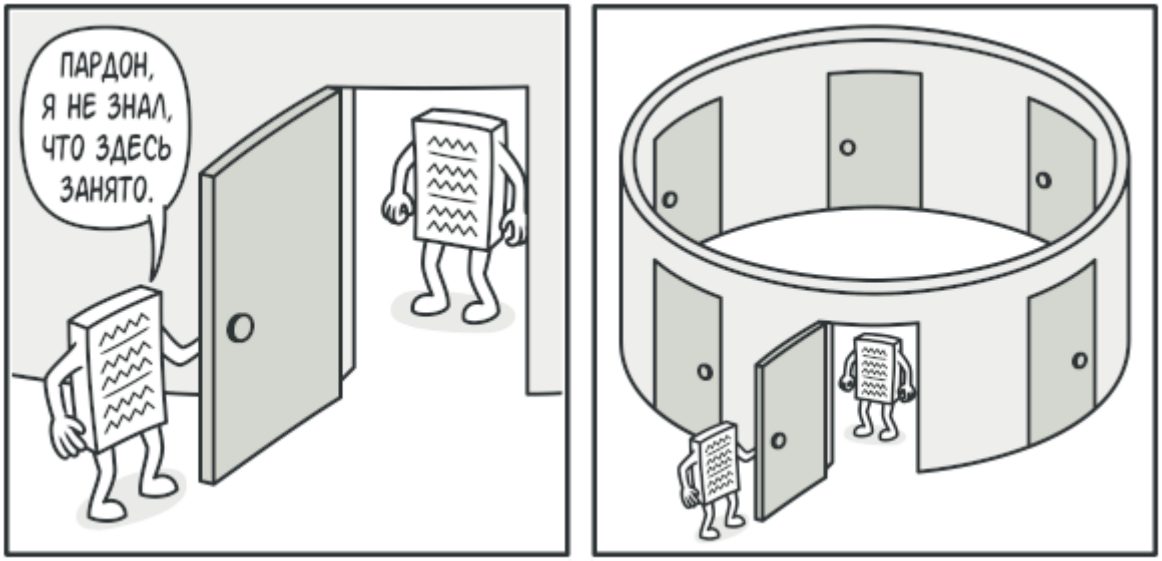
Одиночка

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

Гарантирует наличие единственного экземпляра класса.

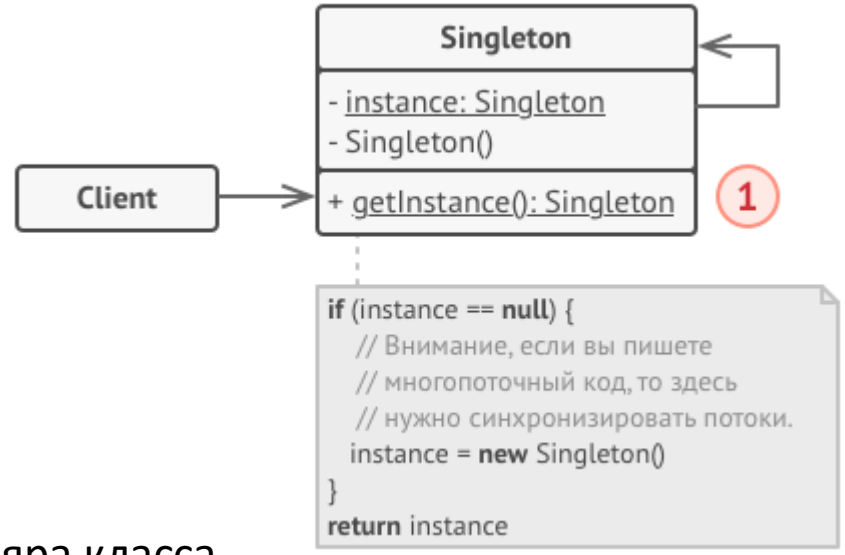
Представьте, что вы создали объект, а через некоторое время пробуете создать ещё один. В этом случае хотелось бы получить старый объект, вместо создания нового. Такое поведение невозможно реализовать с помощью обычного конструктора, так как конструктор класса всегда возвращает новый объект.

Предоставляет глобальную точку доступа. Глобальные переменные не защищены от записи, поэтому любой код может подменять их значения без вашего ведома.



Одиночка: Структура, Использование

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.

Плюсы

Гарантирует наличие единственного экземпляра класса.
Предоставляет к нему глобальную точку доступа.
Реализует отложенную инициализацию объекта-одиночки.

Минусы

Нарушает принцип единственной ответственности класса.
Маскирует плохой дизайн.
Проблемы мультипоточности.
Требует постоянного создания Mock-объектов при юнит-тестировании.

Когда использовать?

Когда в программе должен быть единственный экземпляр какого-то класса, доступный всем клиентам (например, общий доступ к базе данных из разных частей программы).
Когда вам хочется иметь больше контроля над глобальными переменными.