

Введение в паттерны проектирования

Содержание

1. Паттерны проектирования
2. История паттернов
3. Классификация паттернов
4. Отношения между классами
5. Порождающие паттерны:
 - Одиночка
 - Фабричный метод
 - Строитель
 - Прототип
 - Абстрактная Фабрика

Паттерны проектирования

- это часто встречающееся решение определенной проблемы при проектировании архитектуры программ
- В отличие от готовых функций или библиотек, паттерн нельзя просто взять и скопировать в программу. Паттерн представляет собой не какой-то конкретный код, а общую концепцию решения той или иной проблемы, которую нужно будет еще подстроить под нужды программы.

История паттернов

Концепцию паттернов впервые описал Кристофер Александер в книге «Язык шаблонов. Города. Здания. Строительство». В книге описан «язык» для проектирования окружающей среды, единицы которого — шаблоны (или паттерны) — отвечают на архитектурные вопросы.

Идея показалась заманчивой четверке авторов: Эриху Гамме, Ричарду Хелму, Ральфу Джонсону, Джону Влиссидесу. В 1995 году они написали книгу «Приемы объектно-ориентированного проектирования. Паттерны проектирования», в которую вошли 23 паттерна, решающие различные проблемы объектно-ориентированного дизайна. Вскоре все стали называть её «book by the gang of four», то есть «книга от банды четырёх», а затем и вовсе «GOF book».

С тех пор были найдены десятки других объектных паттернов. «Паттерновый» подход стал популярен и в других областях программирования, поэтому сейчас можно встретить всевозможные паттерны и за пределами объектного проектирования.

Классификация паттернов

- **порождающие (6)** - беспокоятся о гибком создании объектов без внесения в программу лишних зависимостей.
- **структурные (7)** - показывают различные способы построения связей между объектами.
- **поведенческие (11)** - заботятся об эффективной коммуникации между объектами.

Порождающие:

- Singleton (Одиночка) - ограничивает создание одного экземпляра класса, обеспечивает доступ к его единственному объекту.
- Simple Factory (Простая Фабрика) - используется, когда у нас есть суперкласс с несколькими подклассами и на основе ввода, нам нужно вернуть один из подкласса.
- Abstract Factory (Абстрактная фабрика) - используем супер фабрику для создания фабрики, затем используем созданную фабрику для создания объектов.
- Builder (Строитель) - используется для создания сложного объекта с использованием простых объектов. Постепенно он создает большой объект от малого и простого объекта.
- Prototype (Прототип) - помогает создать дублированный объект с лучшей производительностью, вместо нового создается возвращаемый клон существующего объекта.
- Fabric Method (Фабричный метод) - менеджер предоставляет способ делегирования логики создания экземпляра дочерним классам.

Структурные:

- Adapter (Адаптер) - это конвертер между двумя несовместимыми объектами. Используя паттерн адаптера, мы можем объединить два несовместимых интерфейса.
- Composite (Компоновщик) - использует один класс для представления древовидной структуры.
- Proxy (Заместитель) - представляет функциональность другого класса.
- Flyweight (Легковес) - вместо создания большого количества похожих объектов, объекты используются повторно.
- Facade (Фасад) - обеспечивает простой интерфейс для клиента, и клиент использует интерфейс для взаимодействия с системой.
- Bridge (Мост) - делает конкретные классы независимыми от классов реализации интерфейса.
- Decorator (Декоратор) - добавляет новые функциональные возможности существующего объекта без привязки его структуры.





Поведенческие:

- Template Method (Шаблонный метод) - определяющий основу алгоритма и позволяющий наследникам переопределять некоторые шаги алгоритма, не изменяя его структуру в целом.
- Mediator (Посредник) - предоставляет класс посредника, который обрабатывает все коммуникации между различными классами.
- Chain of Responsibility (Цепочка обязанностей) - позволяет избежать жесткой зависимости отправителя запроса от его получателя, при этом запрос может быть обработан несколькими объектами.
- Observer (Наблюдатель) - позволяет одним объектам следить и реагировать на события, происходящие в других объектах.
- Strategy (Стратегия) - алгоритм стратегии может быть изменен во время выполнения программы.

Поведенческие продолжение:

- Command (Команда) - интерфейс команды объявляет метод для выполнения определенного действия.
- State (Состояние) - объект может изменять свое поведение в зависимости от его состояния.
- Visitor (Посетитель) - используется для упрощения операций над группировками связанных объектов.
- Interpreter (Интерпретатор) - определяет грамматику простого языка для проблемной области.
- Iterator (Итератор) - последовательно осуществляет доступ к элементам объекта коллекции, не зная его основного представления.
- Memento (Хранитель) - используется для хранения состояния объекта, позже это состояние можно восстановить.

Отношения между классами

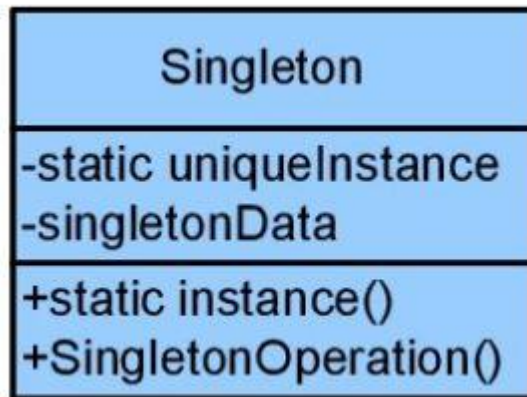
-  — агрегация (aggregation) — описывает связь «часть»–«целое», в котором «часть» может существовать отдельно от «целого». Ромб указывается со стороны «целого».
-  — композиция (composition) — подвид агрегации, в которой «части» не могут существовать отдельно от «целого».
-  — зависимость (dependency) — 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

шаблоны проектирования, которые абстрагируют процесс инстанцирования. 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 инстанцирование другому объекту.

Одиночка (Singleton)

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



Проблема

1. **Гарантирует наличие единственного экземпляра класса.** Чаще всего это полезно для доступа к какому-то общему ресурсу, например, базе данных.

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.

2. **Предоставляет глобальную точку доступа.** Это не просто глобальная переменная, через которую можно достигаться к определенному объекту. Глобальные переменные не защищены от записи, поэтому любой код может подменять их значения без вашего ведома.

Но есть и другой нюанс. Неплохо бы хранить в одном месте и код, который решает проблему №1, а также иметь к нему простой и доступный интерфейс.

Решение

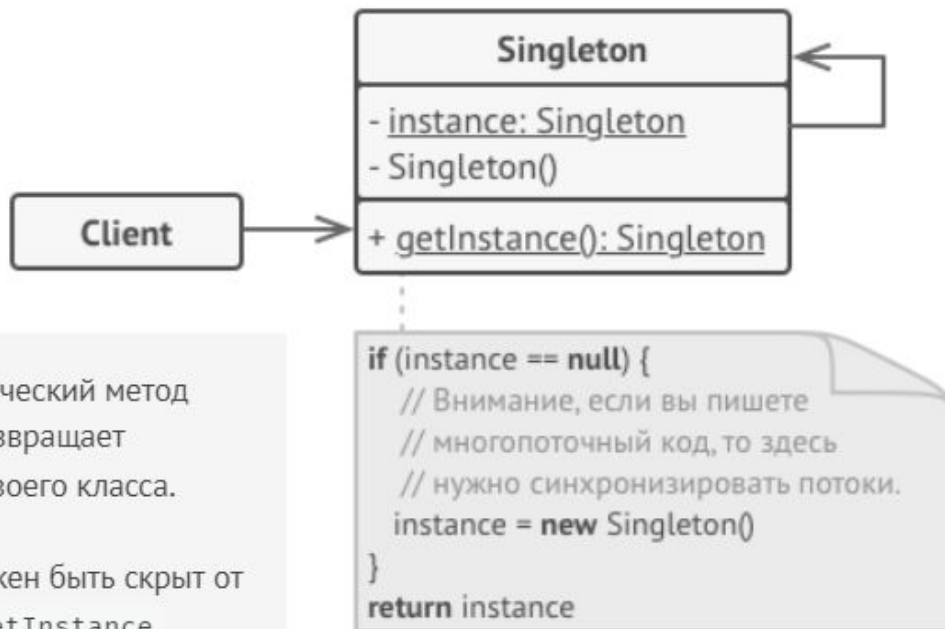
Все реализации одиночки сводятся к тому, чтобы скрыть конструктор по умолчанию и создать публичный статический метод, который и будет контролировать жизненный цикл объекта-одиночки. Если у вас есть доступ к классу одиночки, значит, будет доступ и к этому статическому методу. Из какой точки кода вы бы его ни вызвали, он всегда будет отдавать один и тот же объект.

Пример из жизни: В стране одновременно может быть только один президент. Один и тот же президент должен действовать, когда того требуют обстоятельства. Президент здесь является одиночкой.

Простыми словами: Обеспечивает тот факт, что создаваемый объект является единственным объектом своего класса.

Вообще шаблон одиночка признан антипаттерном, необходимо избегать его чрезмерного использования. Он необязательно плох и может иметь полезные применения, но использовать его надо с осторожностью, потому что он вводит глобальное состояние в ваше приложение и его изменение в одном месте может повлиять на другие части приложения, что вызовет трудности при отладке. Другой минус — это то, что он делает ваш код связанным.

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.

Пример кода

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

Применимость

Когда в программе должен быть единственный экземпляр какого-то класса, доступный всем клиентам (например, общий доступ к базе данных из разных частей программы).

Одиночка скрывает от клиентов все способы создания нового объекта, кроме специального метода. Этот метод либо создаёт объект, либо отдаёт существующий объект, если он уже был создан.

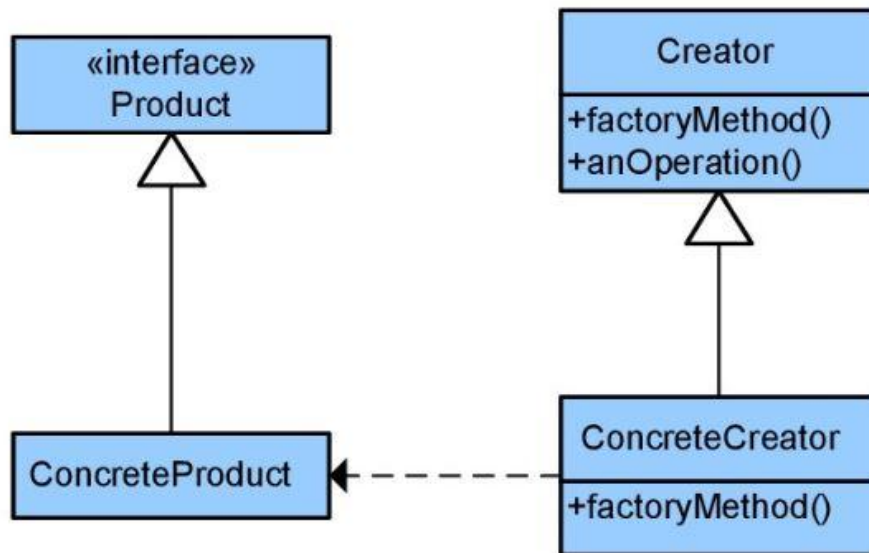
Когда вам хочется иметь больше контроля над глобальными переменными.

В отличие от глобальных переменных, Одиночка гарантирует, что никакой другой код не заменит созданный экземпляр класса, поэтому вы всегда уверены в наличии лишь одного объекта-одиночки.

Тем не менее, в любой момент вы можете расширить это ограничение и позволить любое количество объектов-одиночек, поменяв код в одном месте (метод `getInstance`).

Фабричный метод

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



Проблема

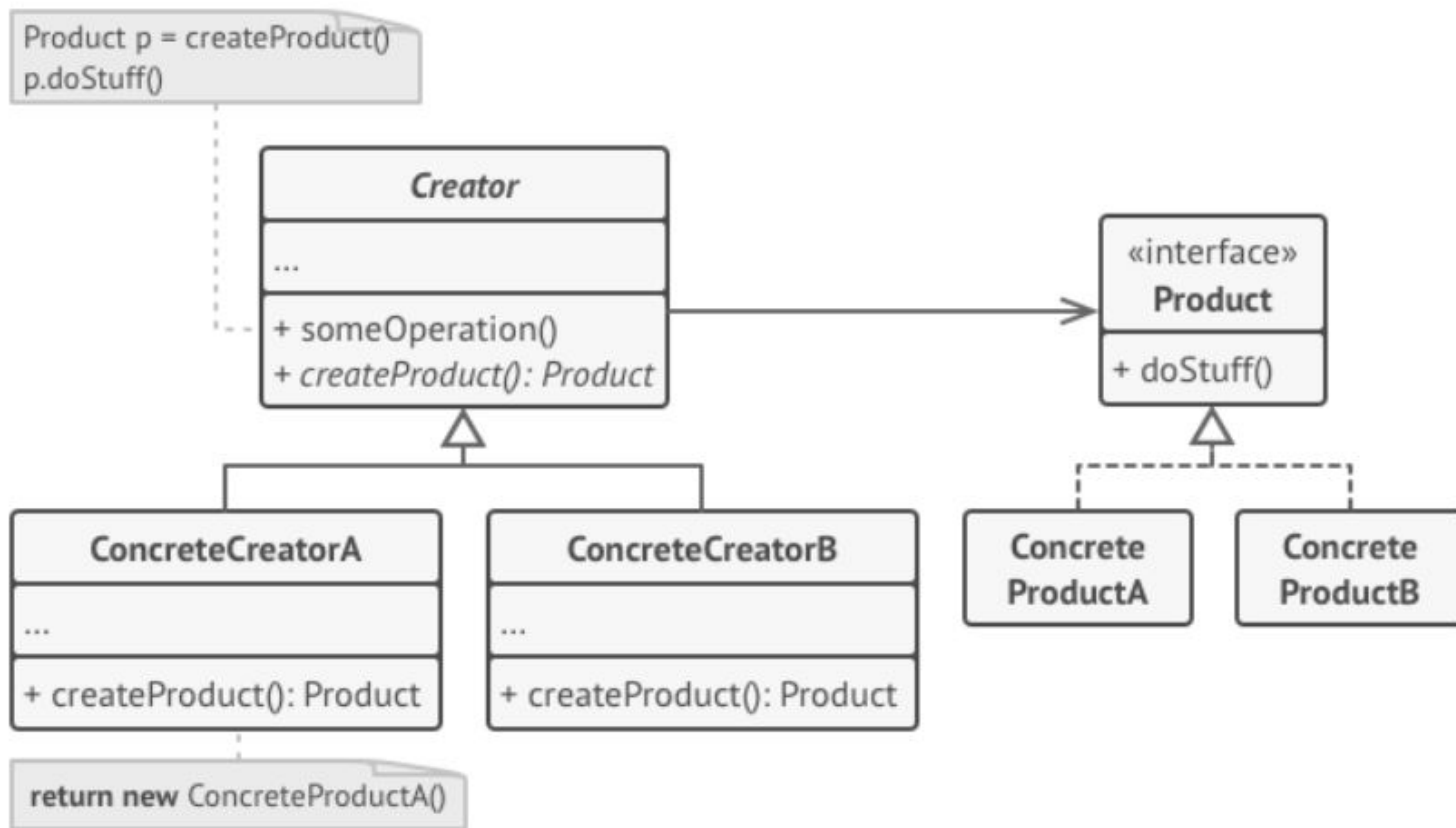
Представьте, что вы создаете программу управления грузовыми перевозками. Сперва вы рассчитываете перевозить товары только на автомобилях. Поэтому весь ваш код работает с объектами класса Грузовик.

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

Отличные новости, правда?! Но как насчёт кода? Большая часть существующего кода жёстко привязана к классам Грузовиков. Чтобы добавить в программу классы морских Судов, понадобится перелопатить всю программу. Более того, если вы потом решите добавить в программу еще один вид транспорта, то всю эту работу придется повторить.

В итоге вы получите ужасающий код, наполненный условными операторами, которые выполняют то или иное действие, в зависимости от класса транспорта.

Решение



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.
3. Создатель объявляет фабричный метод, который должен возвращать новые объекты продуктов. Важно, чтобы тип результата совпадал с общим интерфейсом продуктов.

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.

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

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

```
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(); }
```

```
}
```

Применимость

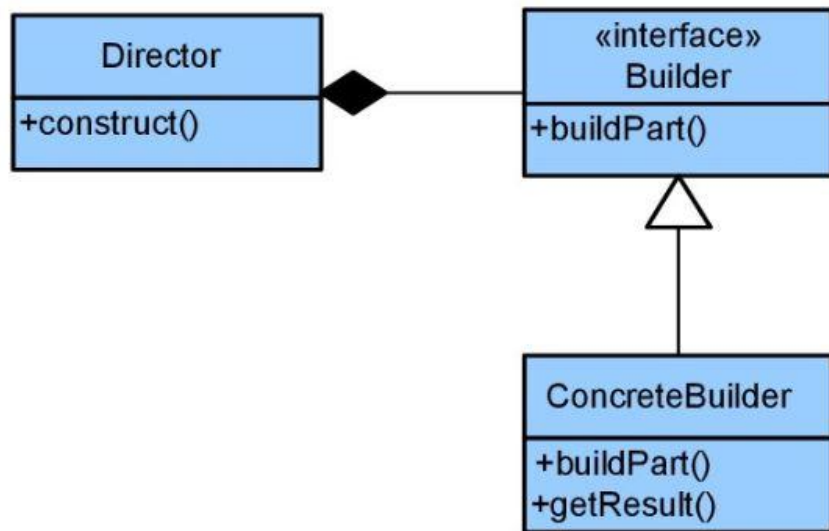
Когда заранее неизвестны типы и зависимости объектов, с которыми должен работать ваш код.

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.

Строитель

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



Проблема

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 — распылен по всему клиентскому коду.

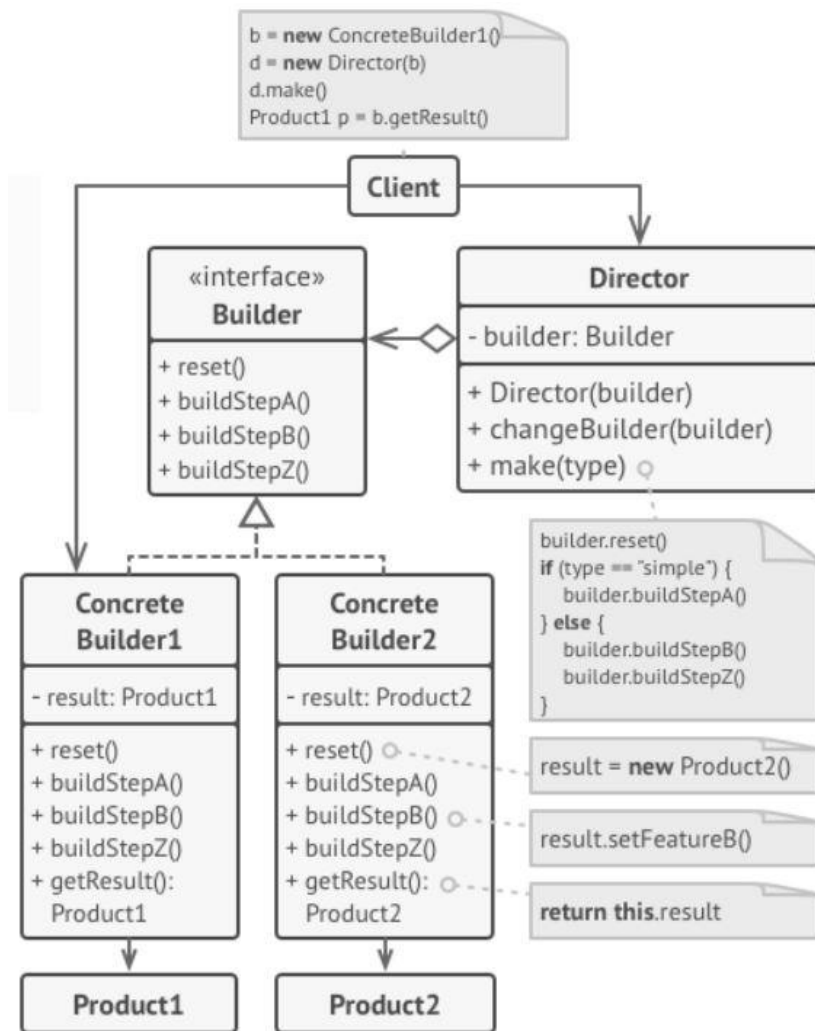
Самое простое решение — расширить класс, создав подклассы для всех комбинаций параметров объекта. Проблема такого подхода — это громадное количество классов, которые вам придётся создать.

Чтобы не плодить подклассы, вы можете подойти к решению с другой стороны. Вы можете создать гигантский конструктор класса, принимающий уйму параметров для контроля над создаваемым продуктом.

Решение

Паттерн предлагает разбить процесс конструирования объекта на отдельные шаги (например, построить Стены, вставить Двери и другие). Чтобы создать объект, вам нужно поочередно вызывать методы строителя. Причем не нужно запускать все шаги, а только те, что нужны для производства объекта определенной конфигурации.

Вы можете пойти дальше и выделить вызовы методов строителя в отдельный класс, называемый директором. В этом случае директор будет задавать порядок шагов строительства, а строитель — выполнять их.



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

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

Продукт — создаваемый объект. Продукты, сделанные разными строителями, не обязаны иметь общий интерфейс.

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 передает строителя через параметр строительного метода директора. В этом случае можно каждый раз применять разных строителей для производства различных представлений объектов.

Применимость

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.

Прототип

порождающий паттерн проектирования, который позволяет копировать объекты, не вдаваясь в подробности их реализации.

Использовать:

- Когда конкретный тип создаваемого объекта должен определяться динамически во время выполнения
- Когда нежелательно создание отдельной иерархии классов фабрик для создания объектов-продуктов из параллельной иерархии классов (как это делается, например, при использовании паттерна Абстрактная фабрика)
- Когда клонирование объекта является более предпочтительным вариантом нежели его создание и инициализация с помощью конструктора. Особенно когда известно, что объект может принимать небольшое ограниченное число возможных состояний.

Абстрактная фабрика

порождающий паттерн проектирования, который позволяет создавать семейства связанных объектов, не привязываясь к конкретным классам создаваемых объектов.

Использовать:

- Когда система не должна зависеть от способа создания и компоновки новых объектов
- Когда создаваемые объекты должны использоваться вместе и являются взаимосвязанными