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

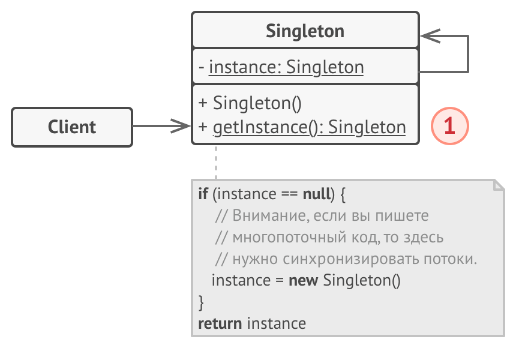
Одиночка решает сразу две проблемы:

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

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

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.



**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.

**--Проблемы мультипоточности.**

**LAZY**

Таким образом, и главный, и дополнительный поток пытаются инициализровать синглтон нужным значением - "Windows 10", либо "Windows 8.1". Какое значение сиглтон получит в итоге, пресказать в данном случае невозможно.

Он будет создан только тогда, когда кто-то вызовет метод getInstance (). Он проверит, имеет ли он значение null (означает первый вызов метода getInstance ()), тогда объект будет создан и будет возвращен. Но если это не NULL, это означает, что экземпляр Singleton уже инициализирован кем-то другим, поэтому он просто вернет уже инициализированную ссылку на экземпляр Singleton.

**EAGER**

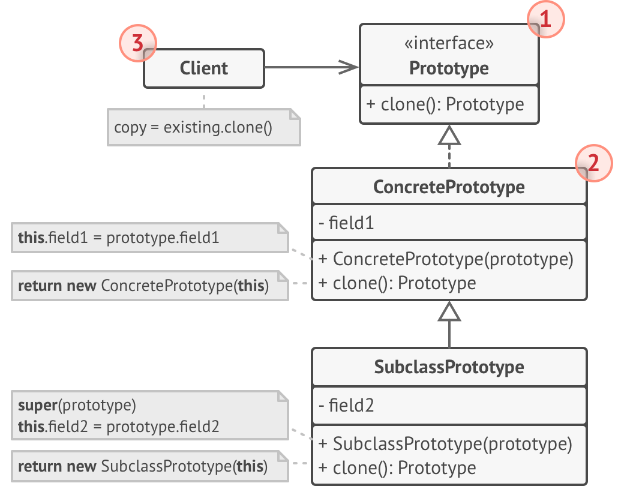
Согласно принципам Eager Instantiation, объект должен быть создан заранее и должен быть готов к использованию. Вот что здесь делается. Мы знаем, что перед выполнением приложения все классы загружаются в первую очередь. И при загрузке классов создаются все их элементы данных. Поэтому, когда мы запустим наше приложение, класс Singleton также будет загружен, и его экземпляры-члены будут также создавать частный статический экземпляр Singleton и открытый статический набор Set. Классы загружаются JVM только один раз автоматически, поэтому в приложении будет создан только один объект, и он будет следовать принципам Singleton Design Pattern.

Поскольку этот объект будет создан заранее, поэтому мы не знаем, будем ли мы его использовать или нет, но этот объект готов к использованию (Eagerly Instantiated), и я могу получить ссылку на этот объект, используя открытый статический метод Singleton getInstance ().

Еще одна вещь, о которой нам нужно помнить, - это то, что Eager - это потокобезопасность. Потокобезопасный означает, что не имеет значения, сколько потоков пытается вызвать метод getInstance (), они всегда получат тот единственный объект, который создан с нетерпением. Следовательно, нет никаких шансов, что будут созданы два объекта, и не будет никакого нарушения принципа Singleton Design Pattern.

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

Паттерн Прототип поручает создание копий самим копируемым объектам. Он вводит общий интерфейс для всех объектов, поддерживающих клонирование. Это позволяет копировать объекты, не привязываясь к их конкретным классам. Обычно такой интерфейс имеет всего один метод clone.



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

Существует два типа клонирования шаблонов прототипов. Одним из них является мелкое клонирование, которое вы только что прочитали в первом вопросе. В мелкой копии клонируется только этот объект, любые объекты, содержащиеся в этом объекте, не клонируются. Например, рассмотрим рисунок «Глубокое клонирование в действии»: у нас есть класс клиентов, и у нас есть класс адресов, агрегированный внутри класса клиентов. MemberWiseClone будет клонировать только класс клиента "ClsCustomer", но не класс "ClsAddress". Поэтому мы добавили функцию «MemberWiseClone» в адресный класс. Теперь, когда мы вызываем функцию «getClone», мы вызываем родительскую функцию клонирования, а также дочернюю функцию клонирования, что приводит к клонированию всего объекта. Когда родительские объекты клонируются с содержащими их объектами, это называется глубоким клонированием, а когда только родительские объекты являются клонами, это называется мелким клонированием.

 Позволяет клонировать объекты, не привязываясь к их конкретным классам.

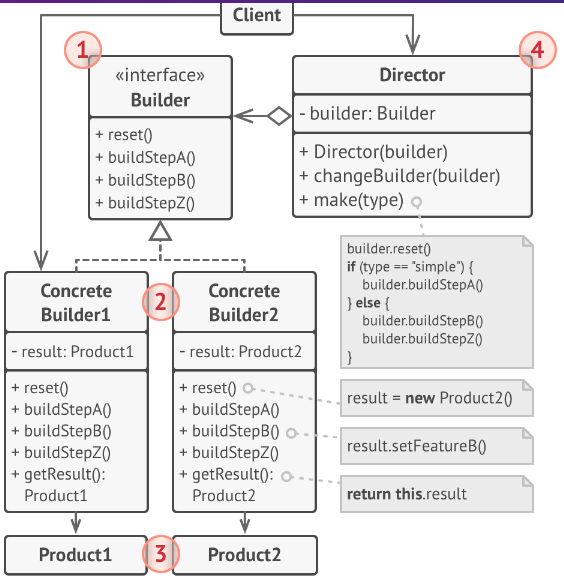
 Меньше повторяющегося кода инициализации объектов.

 Ускоряет создание объектов.

 Альтернатива созданию подклассов для конструирования сложных объектов.

**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.

Паттерн Строитель предлагает вынести конструирование объекта за пределы его собственного класса, поручив это дело отдельным объектам, называемым строителями. Паттерн предлагает разбить процесс конструирования объекта на отдельные шаги (например, построитьСтены, вставитьДвери и другие). Чтобы создать объект, вам нужно поочерёдно вызывать методы строителя. Причём не нужно запускать все шаги, а только те, что нужны для производства объекта определённой конфигурации. Зачастую один и тот же шаг строительства может отличаться для разных вариаций производимых объектов. В этом случае вы можете создать несколько классов строителей, выполняющих одни и те же шаги по-разному. Используя этих строителей в одном и том же строительном процессе, вы сможете получать на выходе различные объекты.



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.
5. 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. В этом случае можно каждый раз применять разных строителей для производства различных представлений объектов.

 Позволяет создавать продукты пошагово.

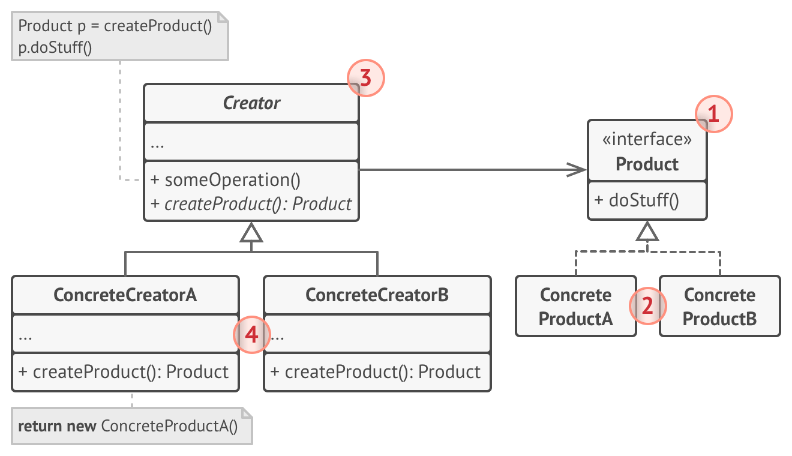
 Позволяет использовать один и тот же код для создания различных продуктов.

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

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

Паттерн Фабричный метод предлагает 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.

Чтобы эта система заработала, все возвращаемые объекты должны иметь общий интерфейс. Подклассы смогут производить объекты различных классов, следующих одному и тому же интерфейсу. Например, классы Грузовик и Судно реализуют интерфейс Транспорт с методом доставить. Каждый из этих классов реализует метод по-своему: грузовики везут грузы по земле, а суда — по морю. Фабричный метод в классе ДорожнойЛогистики вернёт объект-грузовик, а класс МорскойЛогистики — объект-судно. Для клиента фабричного метода нет разницы между этими объектами, так как он будет трактовать их как некий абстрактный Транспорт. Для него будет важно, чтобы объект имел метод доставить, а как конкретно он работает — не важно.



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.
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.

 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*.

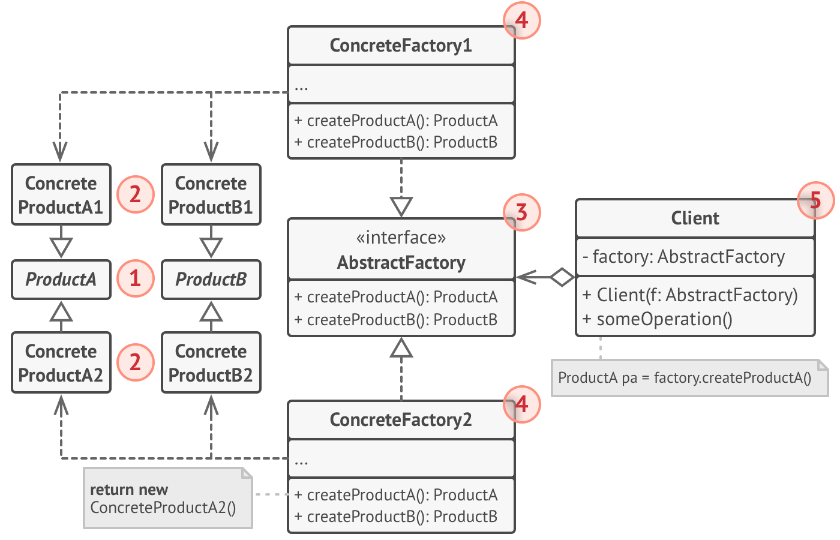
**Абстрактная фабрика** — это порождающий паттерн проектирования, который позволяет создавать семейства связанных объектов, не привязываясь к конкретным классам создаваемых объектов.

Для начала паттерн Абстрактная фабрика предлагает выделить общие интерфейсы для отдельных продуктов, составляющих семейства. Так, все вариации кресел получат общий интерфейс Кресло, все диваны реализуют интерфейс Диван и так далее. Далее вы создаёте абстрактную фабрику — общий интерфейс, который содержит методы создания всех продуктов семейства (например, создатьКресло, создатьДиван и создатьСтолик). Эти операции должны возвращать **абстрактные** типы продуктов, представленные интерфейсами, которые мы выделили ранее — Кресла, Диваны и Столики. Как насчёт вариаций продуктов? Для каждой вариации семейства продуктов мы должны создать свою собственную фабрику, реализовав абстрактный интерфейс. Фабрики создают продукты одной вариации. Например, ФабрикаМодерн будет возвращать только КреслаМодерн,ДиваныМодерн и СтоликиМодерн.

Клиентский код должен работать как с фабриками, так и с продуктами только через их общие интерфейсы. Это позволит подавать в ваши классы любой тип фабрики и производить любые продукты, ничего не ломая.

Например, клиентский код просит фабрику сделать стул. Он не знает, какого типа была эта фабрика. Он не знает, получит викторианский или модерновый стул. Для него важно, чтобы на стуле можно было сидеть и чтобы этот стул отлично смотрелся с диваном той же фабрики.

Осталось прояснить последний момент: кто создаёт объекты конкретных фабрик, если клиентский код работает только с интерфейсами фабрик? Обычно программа создаёт конкретный объект фабрики при запуске, причём тип фабрики выбирается, исходя из параметров окружения или конфигурации.



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) и реализуют методы абстрактной фабрики, позволяя создавать все продукты определённой вариации.

 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*.

**Объектный пул** ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *object pool*) — порождающий шаблон проектирования, набор инициализированных и готовых к использованию объектов. Когда системе требуется объект, он не создаётся, а берётся из пула. Когда объект больше не нужен, он не уничтожается, а возвращается в пул.

Объектный пул применяется для повышения производительности, когда создание объекта в начале работы и уничтожение его в конце приводит к большим затратам. Особенно заметно повышение производительности, когда объекты часто создаются-уничтожаются, но одновременно существует лишь небольшое их число.

