

☹ Проблема

Представьте, что вы пишете симулятор мебельного магазина. Ваш код содержит:

1. Семейство зависимых продуктов. Скажем, **Кресло** + **Диван** + **Столик**.
2. Несколько вариаций этого семейства. Например, продукты **Кресло**, **Диван** и **Столик** представлены в трёх разных стилях: **Ар-деко**, **Викторианском** и **Модерне**.

	Кресло	Столик	Диван
Ар-деко			
Викторианский			
Модерн			

Семейства продуктов и их вариации.

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.

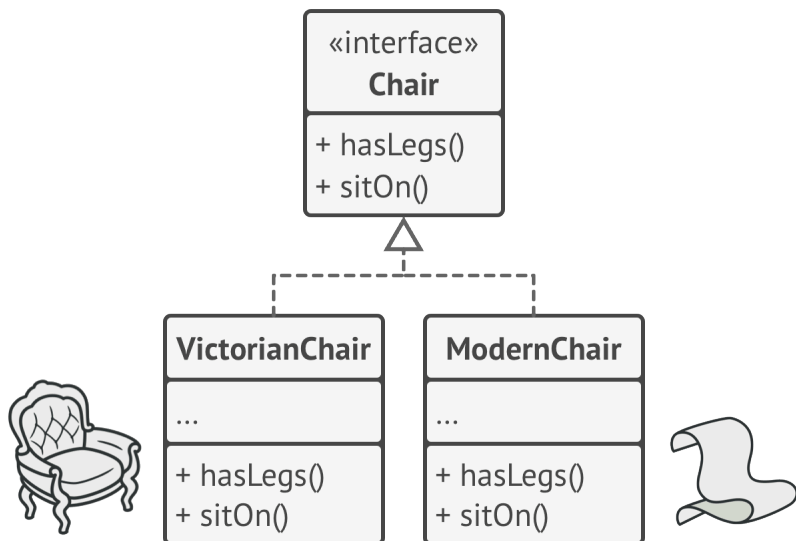


Клиенты расстраиваются, если получают несочетающиеся продукты.

Кроме того, вы не хотите вносить изменения в существующий код при добавлении новых продуктов или семейств в программу. Поставщики часто обновляют свои каталоги, и вам бы не хотелось менять уже написанный код при получении новых моделей мебели.

😊 Решение

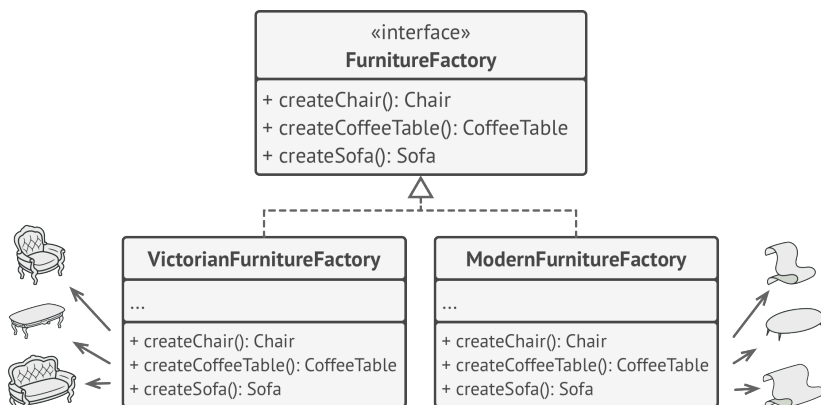
Для начала, паттерн **Абстрактная фабрика** предлагает выделить общие интерфейсы для отдельных продуктов, составляющих семейства. Так, все вариации кресел получают общий интерфейс `Кресло`, все диваны реализуют интерфейс `Диван` и так далее.



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

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`.

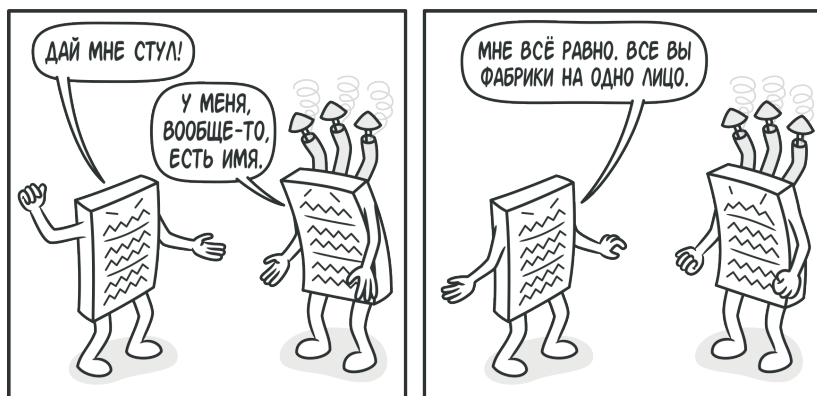
Как насчёт вариаций продуктов? Для каждой вариации семейства продуктов мы должны создать свою собственную фабрику, реализовав абстрактный интерфейс. Фабрики создают продукты одной вариации. Например, `ФабрикаМодерн` будет возвращать только `КреслаМодерн`, `ДиваныМодерн` и `СтоликиМодерн`.



Конкретные фабрики соответствуют определённой вариации семейства продуктов.

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 фабрика это была.

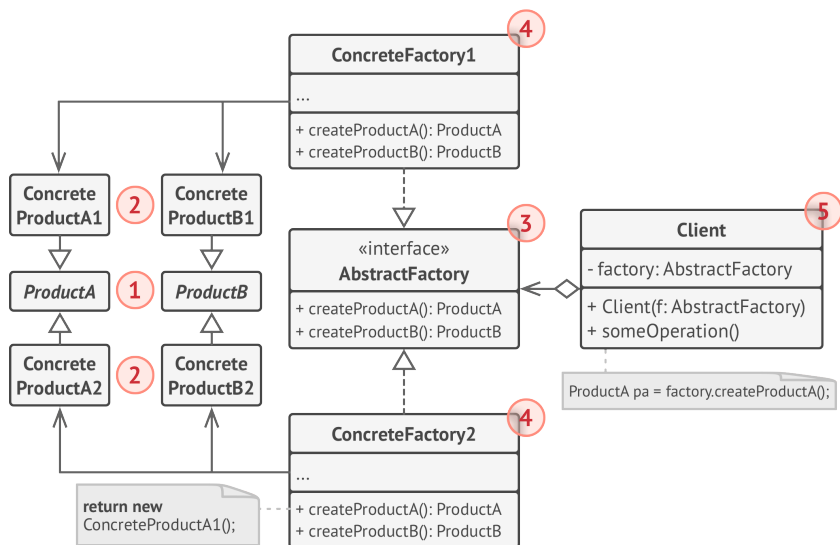


Клиентскому коду должно быть всё равно с какой фабрикой работать.

Он не знает, получит викторианский или современный стул. Для него важно, чтобы на этом стуле можно 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.

Структура

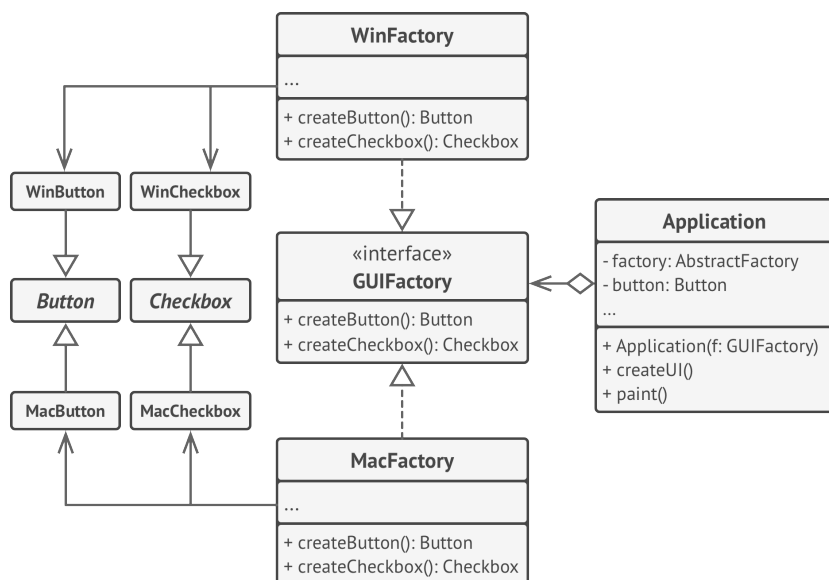


1. **Абстрактные продукты** объявляют интерфейсы продуктов, которые связаны друг с другом по смыслу, но выполняют разные функции.

2. **Конкретные продукты** — большой набор классов, которые относятся к различным абстрактным продуктам (кресло/столик), но имеют одни и те же вариации (Викториан./Модерн).
3. **Абстрактная фабрика** объявляет методы создания различных абстрактных продуктов (кресло/столик).
4. **Конкретные фабрики** относятся каждая к своей вариации продуктов (Викториан./Модерн) и реализуют методы абстрактной фабрики, позволяя создавать все продукты определённой вариации.
5. Несмотря на то, что конкретные фабрики порождают конкретные продукты, сигнатуры их методов должны возвращать соответствующие абстрактные продукты. Это позволит клиентскому коду, использующему фабрику, не привязываться к конкретным классам продуктов. Клиент сможет работать с любыми вариациями продуктов через абстрактные интерфейсы.

Псевдокод

В этом примере **Абстрактная фабрика** создаёт кросс-платформенные элементы интерфейса и следит за тем, чтобы они соответствовали выбранной операционной системе.



Пример кросс-платформенного графического интерфейса пользователя.

Кросс-платформенная программа может показывать одни и те же элементы интерфейса, выглядящие чуточку по-другому в различных операционных системах. В такой программе, важно, чтобы все создаваемые элементы всегда соответствовали текущей операционной системе. Вы бы не хотели, чтобы программа, запущенная на Windows, вдруг начала показывать чекбоксы в стиле macOS.

Абстрактная фабрика объявляет список продуктов, которые может запрашивать клиентский код. Конкретные фабрики относятся к различным операционным системам и создают элементы одного и того же вида.

В самом начале, программа определяет, какая из фабрик соответствует текущей операционке. Затем, создаёт эту

фабрику и отдаёт её клиентскому коду. В дальнейшем, клиент будет работать только с этой фабрикой, чтобы исключить несовместимость возвращаемых продуктов.

Клиентский код не зависит от конкретных классов фабрик и элементов интерфейса. Он общается с ними через абстрактные интерфейсы. Благодаря этому, клиент может работать любой разновидностью фабрик и элементов интерфейса.

Чтобы добавить в программу новую вариацию элементов (например, для поддержки Linux), вам не нужно трогать клиентский код. Достаточно создать ещё одну фабрику, производящую эти элементы.

```
1 // Этот паттерн предполагает, что у вас есть несколько
2 // семейств продуктов, находящихся в отдельных иерархиях
3 // классов (Button/Checkbox). Продукты одного семейства
4 // должны иметь общий интерфейс.
5 interface Button is
6     method paint()
7
8 // Все семейства продуктов имеют одинаковые
9 // вариации (macOS/Windows).
10 class WinButton implements Button is
11     method paint() is
12         // Отрисовать кнопку в стиле Windows.
13
14 class MacButton implements Button is
15     method paint() is
16         // Отрисовать кнопку в стиле macOS.
```







```
17 interface Checkbox is
18     method paint()
19
20 class WinCheckbox implements Checkbox is
21     method paint() is
22         // Отрисовать чекбокс в стиле Windows.
23
24 class MacCheckbox implements Checkbox is
25     method paint() is
26         // Отрисовать чекбокс в стиле macOS.
27
28
29 // Абстрактная фабрика знает обо всех (абстрактных)
30 // типах продуктов.
31 interface GUIFactory is
32     method createButton():Button
33     method createCheckbox():Checkbox
34
35
36 // Каждая конкретная фабрика знает и создаёт только продукты
37 // своей вариации.
38 class WinFactory implements GUIFactory is
39     method createButton():Button is
40         return new WinButton()
41     method createCheckbox():Checkbox is
42         return new WinCheckbox()
43
44 // Несмотря на то что фабрики оперируют конкретными
45 // классами, их методы возвращают абстрактные типы
46 // продуктов. Благодаря этому, фабрики можно взаимозаменять,
47 // не изменяя клиентский код.
48 class MacFactory implements GUIFactory is
49     method createButton():Button is
50         return new MacButton()
```

```

51     method createCheckbox():Checkbox is
52         return new MacCheckbox()
53
54
55     // Код, использующий фабрику, не волнует с какой конкретно
56     // фабрикой он работает. Все получатели продуктов работают с
57     // продуктами через абстрактный интерфейс.
58     class Application is
59         private field button: Button
60         constructor Application(factory: GUIFactory) is
61             this.factory = factory
62         method createUI()
63             this.button = factory.createButton()
64         method paint()
65             button.paint()
66
67
68     // Приложение выбирает тип и создаёт конкретные фабрики
69     // динамически исходя из конфигурации или окружения.
70     class ApplicationConfigurator is
71         method main() is
72             config = readApplicationConfigFile()
73
74             if (config.OS == "Windows") then
75                 factory = new WinFactory()
76             else if (config.OS == "Web") then
77                 factory = new MacFactory()
78             else
79                 throw new Exception("Error! Unknown operating system.")
80
81             Application app = new Application(factory)

```

Применимость

-  **Когда бизнес-логика программы должна работать с разными видами связанных друг с другом продуктов, не завися от конкретных классов продуктов.**
-  Абстрактная фабрика скрывает от клиентского кода подробности того, как и какие конкретно объекты будут созданы. Но при этом клиентский код может работать со всеми типами создаваемых продуктов, так как их общий интерфейс был заранее определён.
-  **Когда в программе уже используется Фабричный метод, но очередные изменения предполагают введение новых типов продуктов.**
-  В хорошей программе, каждый *класс отвечает только за одну вещь*. Если класс имеет слишком много фабричных методов, они способны затуманить его основную функцию. Поэтому имеет смысл вынести всю логику создания продуктов в отдельную иерархию классов, применив абстрактную фабрику.

Шаги реализации

1. Создайте таблицу соотношений типов продуктов к вариациям семейств продуктов.

2. Сведите все вариации продуктов к общим интерфейсам.
3. Определите интерфейс абстрактной фабрики. Он должен иметь фабричные методы для создания каждого из типов продуктов.
4. Создайте классы конкретных фабрик, реализовав интерфейс абстрактной фабрики. Этим классов должно быть столько же, сколько и вариаций семейств продуктов.
5. Измените код инициализации программы так, чтобы она создавала определённую фабрику и передавала её в клиентский код.
6. Замените в клиентском коде участки создания продуктов через конструктор вызовами соответствующих методов фабрики.



Преимущества и недостатки

- ✓ 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*.

- ✗ Усложняет код программы за счёт множества дополнительных классов.
- ✗ Требуется наличия всех типов продуктов в каждой вариации.

⇔ Отношения с другими паттернами

- Многие архитектуры начинаются с применения **Фабричного метода** (более простого и расширяемого через подклассы) и эволюционируют в сторону **Абстрактной фабрики**, **Прототипа** или **Строителя** (более гибких, но и более сложных).
- **Строитель** концентрируется на постройке сложных объектов шаг за шагом. **Абстрактная фабрика** специализируется на создании семейств связанных продуктов. *Строитель* возвращает продукт только после выполнения всех шагов, а *Абстрактная фабрика* возвращает продукт сразу же.
- Классы **Абстрактной фабрики** чаще всего реализуются с помощью **Фабричного метода**, хотя они могут быть построены и на основе **Прототипа**.
- **Абстрактная фабрика** может быть использована вместо **Фасада** для того, чтобы скрыть платформу-зависимые классы.

- **Абстрактная фабрика** может работать совместно с **Мостом**. Это особенно полезно, если у вас есть абстракции, которые могут работать только с некоторыми из реализаций. В этом случае фабрика будет определять типы создаваемых абстракций и реализаций.
- **Абстрактная фабрика**, **Строитель** и **Прототип** могут быть реализованы при помощи **Одиночки**.