

Погружение в

# ПАТТЕРНЫ ПРОЕКТИРОВАНИЯ



Александр Швец

Погружение в

# **ПАТТЕРНЫ ПРОЕКТИРОВАНИЯ**

v2019-1.1

ДЕМО-ВЕРСИЯ

# Вместо копирайта

Привет! Меня зовут Александр Швец, я автор книги **Погружение в Паттерны**, а также онлайн-курса **Погружение в Рефакторинг**.



Эта книга предназначена для вашего личного пользования. Пожалуйста, не передавайте книгу третьим лицам, за исключением членов своей семьи. Если вы хотите поделиться книгой с друзьями или коллегами, то купите и подарите им легальную копию.

Все деньги, вырученные от продаж моих книг и курсов, идут на развитие **Refactoring.Guru**. Это один из немногих ресурсов с авторским контентом, доступным на русском языке. Каждая легально приобретённая копия помогает проекту жить дальше и приближает момент выхода нового курса или книги.

© Александр Швец, Refactoring.Guru, 2019

✉ [support@refactoring.guru](mailto:support@refactoring.guru)

🖼️ Иллюстрации: Дмитрий Жарт

✎ Редактор: Эльвира Мамонтова

*Посвящаю эту книгу своей жене, Марии, без  
которой я бы не смог довести дело до  
конца ещё лет тридцать.*

# Содержание

<b>Содержание .....</b>	<b>4</b>
<b>Как читать эту книгу .....</b>	<b>6</b>
<b>ВВЕДЕНИЕ В ООП .....</b>	<b>7</b>
Вспоминаем ООП .....	8
Краеугольные камни ООП .....	13
Отношения между объектами .....	20
<b>ОСНОВЫ ПАТТЕРНОВ .....</b>	<b>23</b>
Что такое паттерн? .....	24
Зачем знать паттерны? .....	28
<b>ПРИНЦИПЫ ПРОЕКТИРОВАНИЯ .....</b>	<b>29</b>
Качества хорошей архитектуры .....	30
<b>Базовые принципы проектирования .....</b>	<b>35</b>
§ Инкапсулируйте то, что меняется .....	36
§ Прографируйте на уровне интерфейса .....	40
§ Предпочитайте композицию наследованию .....	45
<b>Принципы SOLID .....</b>	<b>49</b>
§ S: Принцип единственной ответственности .....	50
§ O: Принцип открытости/закрытости .....	52
§ L: Принцип подстановки Лисков .....	55
§ I: Принцип разделения интерфейса .....	61
§ D: Принцип инверсии зависимостей .....	64

<b>КАТАЛОГ ПАТТЕРНОВ .....</b>	<b>67</b>
<b>Порождающие паттерны .....</b>	<b>68</b>
§ Фабричный метод.....	70
§ Абстрактная фабрика .....	85
§ Строитель.....	100
§ Прототип .....	117
§ Одиночка .....	131
<b>Структурные паттерны .....</b>	<b>139</b>
§ Адаптер .....	142
§ Мост .....	155
§ Компоновщик .....	170
§ Декоратор .....	183
§ Фасад .....	201
§ Легковес.....	211
§ Заместитель.....	225
<b>Поведенческие паттерны .....</b>	<b>237</b>
§ Цепочка обязанностей.....	241
§ Команда .....	260
§ Итератор .....	280
§ Посредник.....	294
§ Снимок.....	309
§ Наблюдатель .....	324
§ Состояние .....	339
§ Стратегия .....	355
§ Шаблонный метод .....	367
§ Посетитель .....	379
<b>Заключение .....</b>	<b>394</b>

# Как читать эту книгу?

Эта книга состоит из описания 22-х классических паттернов проектирования, впервые открытых «Бандой Четырёх» (“Gang of Four” или просто GoF) в 1994 году.

Каждая глава книги посвящена только одному паттерну. Поэтому книгу можно читать как последовательно, от края до края, так и в произвольном порядке, выбирая только интересные в данный момент паттерны.

Многие паттерны связаны между собой, поэтому вы сможете с лёгкостью прыгать по связанным темам, используя ссылки, которых в книге предостаточно. В конце каждой главы приведены отношения текущего паттерна с остальными. Если вы видите там название паттерна, до которого ещё не добрались, то попросту читайте дальше — этот пункт будет повторён в другой главе.

Паттерны проектирования универсальны. Поэтому все примеры кода в этой книге приведены на псевдокоде, без привязки к конкретному языку программирования.

Перед изучением паттернов вы можете освежить память, пройдясь по **основным терминам объектного программирования**. Параллельно я расскажу об UML-диаграммах, которых в этой книге множество. Если вы всё это уже знаете, смело приступайте к **изучению паттернов**.

# **ВВЕДЕНИЕ В ООП**

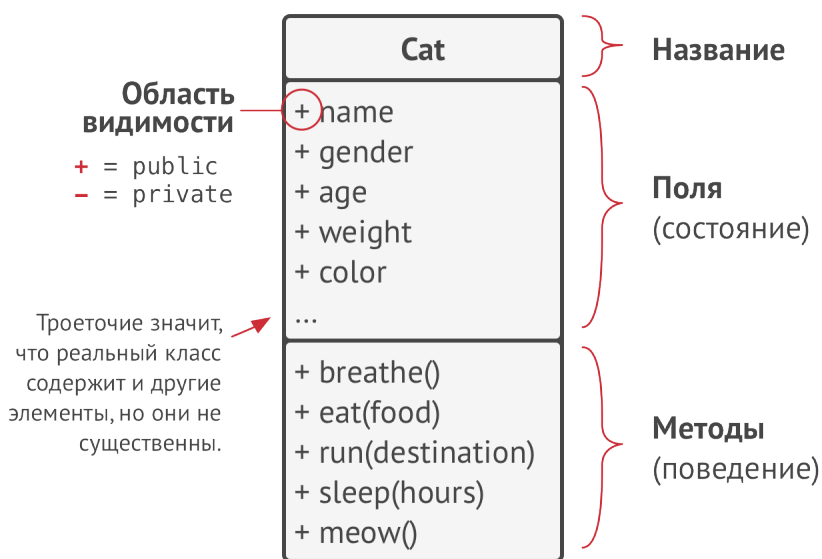


# Вспоминаем ООП

*Объектно-ориентированное программирование* — это методология программирования, в которой все важные вещи представлены **объектами**, каждый из которых является экземпляром определенного **класса**, а классы образуют **иерархию** наследования.

## Объекты, классы

Вы любите котиков? Надеюсь, да, потому что я попытаюсь объяснить все эти вещи на примерах с кошками.



Это UML-диаграмма класса. В книге будет много таких диаграмм.

Итак, у вас есть кот Пушистик. Он является *объектом класса Кот*. Все коты имеют одинаковый набор свойств: имя, пол, возраст, вес, цвет, любимая еда и прочее. Это — *поля класса*.

Почти все коты ведут себя схожим образом: бегают, дышат, спят, едят и мурчат. Всё это — *методы класса*. Обобщённо, поля и методы иногда называют *членами класса*.

Значения полей определённого объекта обычно называют его *состоянием*, а совокупность методов — *поведением*.



**Pushistik: Cat**

```
name  = "Pushistik"  
sex    = "male"  
age    = 3  
weight = 5.5  
color  = gray
```



**Murka: Cat**

```
name  = "Murka"  
sex    = "female"  
age    = 1  
weight = 3.5  
color  = white
```

*Объекты — это экземпляры классов.*

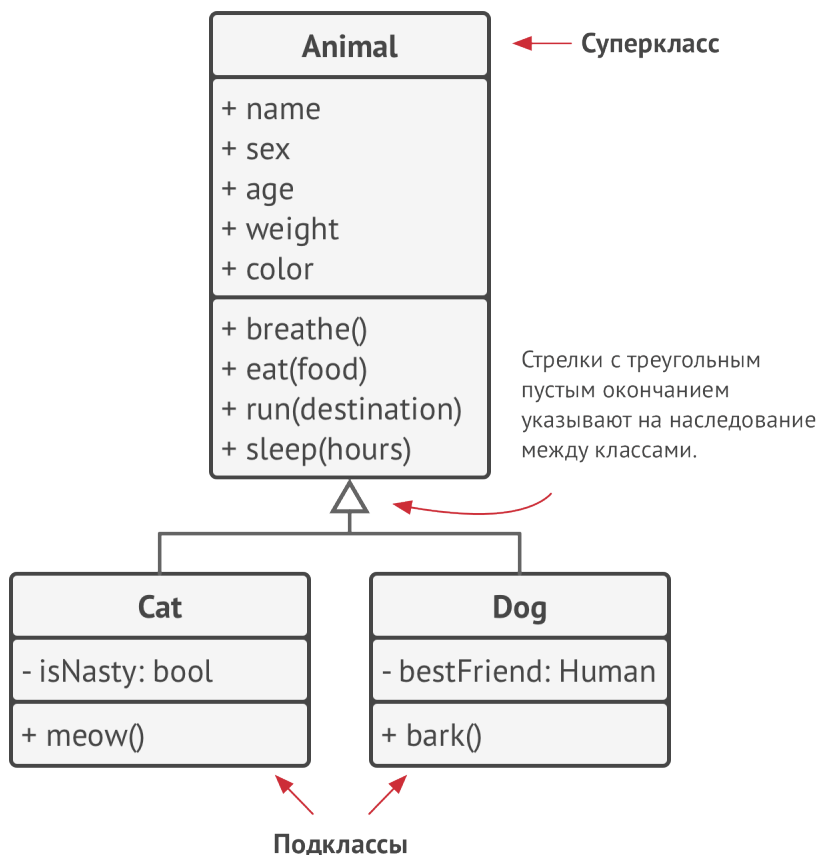
Мурка, кошка вашей подруги, тоже является экземпляром класса `Кот`. Она имеет такой же набор свойств и поведений, что и Пушистик, а отличается от него лишь значениями этих свойств: она другого пола, имеет другой окрас, вес и т. д.

Итак, **класс** — это своеобразный «чертёж», по которому строятся **объекты** — экземпляры этого класса.

## Иерархии классов

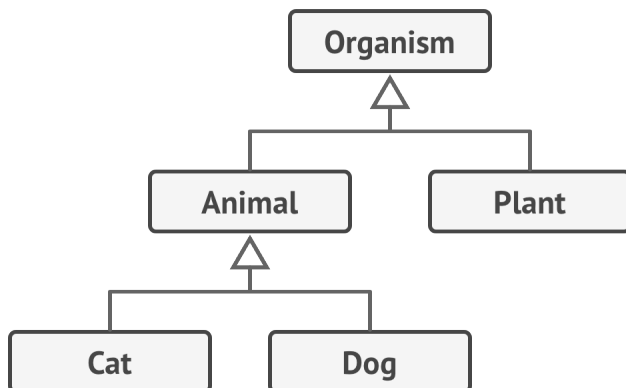
Идём дальше. У вашего соседа есть собака Жучка. Как известно, и собаки, и коты имеют много общего: имя, пол, возраст, цвет есть не только у котов, но и у собак. Да и бегать, дышать, спать и есть могут не только коты. Получается, эти свойства и поведения присущи общему классу `Животных`.

Такой родительский класс принято называть **суперклассом**, а его потомков — **подклассами**. Подклассы наследуют свойства и поведения своего родителя, поэтому в них содержится лишь то, чего нет в суперклассе. Например, только коты могут мурчать, а собаки — лаять.



*UML-диаграмма иерархии классов. Все классы на этой диаграмме являются частью иерархии Животных.*

Мы можем пойти дальше и выделить ещё более общий класс живых Организмов, который будет родительским и для Животных, и для Рыб. Такую «пирамиду» классов обычно называют **иерархией**. Класс Котов унаследует всё как из Животных, так из Организмов.



*Классы на UML-диаграмме можно упрощать, если важно показать отношения между ними.*

Следует упомянуть, что подклассы могут переопределять поведение методов, которые им достались от суперкласса. При этом они могут как полностью заменить поведение метода, так и просто добавить что-то к результату выполнения родительского метода.

**16 страниц**

из полной книги пропущена в демо-версии

# **ПРИНЦИПЫ ПРОЕКТИРОВАНИЯ**

# Качества хорошей архитектуры

Прежде чем перейти к изучению конкретных паттернов, давайте поговорим о самом процессе проектирования, о том, к чему надо стремиться, и о том, чего надо избегать.

## Повторное использование кода

Не секрет, что стоимость и время разработки — это наиболее важные метрики при разработке любых программных продуктов. Чем меньше оба этих показателя, тем конкурентнее продукт будет на рынке и тем больше прибыли получит разработчик.

**Повторное использование** программной архитектуры и кода — это один из самых распространённых способов снижения стоимости разработки. Логика проста: вместо того, чтобы разрабатывать что-то по второму разу, почему бы не использовать прошлые наработки в новом проекте?

Идея выглядит отлично на бумаге, но, к сожалению, не всякий код можно приспособить к работе в новых условиях. Слишком тесные связи между компонентами, зависимость кода от конкретных классов, а не более абстрактных интерфейсов, вшитые в код операции, которые невозможно расширить — всё это уменьшает гибкость вашей архитектуры и препятствует её повторному использованию.



На помощь приходят паттерны проектирования, которые ценой усложнения кода программы повышают гибкость её частей, упрощая дальнейшее повторное использование.

Приведу цитату Эриха Гаммы<sup>1</sup>, одного из первооткрывателей паттернов, о повторном использовании кода и роли паттернов в нём.

“

Существует три уровня повторного использования кода. На самом нижнем уровне находятся классы: полезные библиотеки классов, контейнеры, а также «команды» классов, вроде контейнеров/итераторов.

Фреймворки стоят на самом верхнем уровне. В них важна только архитектура. Они определяют ключевые абстракции для решения определённых бизнес-задач, представленных в виде классов и отношений между ними. Возьмите JUnit, это маленький фреймворк. Он имеет всего несколько классов: `Test`, `TestCase` и `TestSuite`. Обычно фреймворк имеет гораздо больший охват, чем один класс. Вы должны вклиниться в фреймворк, расширив какой-то из его классов. Всё работает по так называемому голливудскому принципу «не звоните нам, мы сами вам перезвоним». Фреймворк позволяет вам задать какое-то своё поведение, а затем сам вызывает его, когда приходит черёд что-то делать. То же про-

---

1. Erich Gamma on Flexibility and Reuse: <https://refactoring.guru/gamma-interview>

исходит и в JUnit. Он обращается к вашему классу, когда нужно выполнить тест, но всё остальное происходит внутри фреймворка.

Есть ещё средний уровень. Это то, где я вижу паттерны. Паттерны проектирования и меньше, и более абстрактные, чем фреймворки. Они, на самом деле, просто описание того, как парочка классов относится и взаимодействует друг с другом. Уровень повторного использования повышается, когда вы двигаетесь в направлении от конкретных классов к паттернам, а затем к фреймворкам.

Что ещё замечательно в этом среднем уровне так это то, что паттерны — это менее рискованный способ повторного использования, чем фреймворки. Разработка фреймворка — это крайне рискованная и дорогая инвестиция. В то же время, паттерны позволяют вам повторно использовать идеи и концепции в отрыве от конкретного кода.

”



## Расширяемость

**Изменения** часто называют главным врагом программиста.

- Вы придумали идеальную архитектуру интернет-магазина, но через месяц пришлось добавить интерфейс для заказов по телефону.
- Вы выпустили видеоигру под Windows, но затем понадобилась поддержка macOS.

- Вы сделали интерфейсный фреймворк с квадратными кнопками, но клиенты начали просить круглые.

У каждого программиста есть дюжина таких историй. Есть несколько причин, почему так происходит.

Во-первых, все мы понимаем проблему лучше в процессе её решения. Нередко к концу работы над первой версией программы, мы уже готовы полностью её переписать, так как стали лучше понимать некоторые аспекты, которые не были очевидны вначале. Сделав вторую версию, вы начинаете понимать проблему ещё лучше, вносите ещё изменения и так далее — процесс не останавливается никогда, ведь не только ваше понимание, но и сама проблема может измениться со временем.

Во-вторых, изменения могут прийти извне. У вас есть идеальный клиент, который с первого раза сформулировал то, что ему надо, а вы в точности это сделали. Прекрасно! Но вот выходит новая версия операционной системы, в которой ваша программа не работает. Чертыхаясь, вы лезете в код, чтобы внести кое-какие изменения.

Можно посмотреть на это с оптимистичной стороны: если кто-то просит вас что-то изменить в программе, значит, она всё ещё кому-то нужна.

Вот почему даже мало-мальски опытный программист проектирует архитектуру и пишет код с учётом будущих изменений.

# Базовые принципы проектирования

Что такое хороший дизайн? По каким критериям его оценивать и каких правил придерживаться при разработке? Как обеспечить достаточный уровень гибкости, связанности, управляемости, стабильности и понятности кода?

Всё это правильные вопросы, но для каждой программы ответ будет чуточку отличаться. Давайте рассмотрим универсальные принципы проектирования, которые помогут вам формулировать ответы на эти вопросы самостоятельно.

Кстати, большинство паттернов, приведённых в этой книге, основаны именно на перечисленных ниже принципах.

## Инкапсулируйте то, что меняется

Определите аспекты программы, класса или метода, которые меняются чаще всего, и отделите их от того, что остаётся постоянным.

Этот принцип преследует единственную цель — уменьшить последствия, вызываемые изменениями.

Представьте, что ваша программа — это корабль, а изменения — коварные мины, которые его подстерегают. Натыкаясь на мину, корабль заполняется водой и тонет.

Зная это, вы можете разделить корабль на независимые секции, проходы между которыми можно наглухо задраивать. Теперь при встрече с миной корабль останется на плаву. Вода затопит лишь одну секцию, оставив остальные без изменений.

Изолируя изменчивые части программы в отдельных модулях, классах или методах, вы уменьшаете количество кода, который затронут последующие изменения. Следовательно, вам нужно будет потратить меньше усилий на то, чтобы привести программу в рабочее состояние, отладить и протестировать изменившийся код. Где меньше работы, там меньше стоимость разработки. А где меньше стоимость, там и преимущество перед конкурентами.

## Пример инкапсуляции на уровне метода

Представьте, что вы разрабатываете интернет-магазин. Где-то внутри вашего кода может существовать метод `getOrderTotal`, который рассчитывает финальную сумму заказа, учитывая размер налога.

Мы можем предположить, что код вычисления налогов, скорее всего, будет часто меняться. Во-первых, логика начисления налога зависит от страны, штата и даже города, в котором находится покупатель. К тому же размер налога непостоянен: власти могут менять его, когда вздумается.

Из-за этих изменений вам постоянно придётся трогать метод `getOrderTotal`, который, по правде, не особо интересуется *деталими* вычисления налогов.

```
1  method getOrderTotal(order) is
2      total = 0
3      foreach item in order.lineItems
4          total += item.price * item.quantity
5
6      if (order.country == "US")
7          total += total * 0.07 // US sales tax
8      else if (order.country == "EU")
9          total += total * 0.20 // European VAT
10
11     return total
```

*ДО: правила вычисления налогов смешаны с основным кодом метода.*

Вы можете перенести логику вычисления налогов в собственный метод, скрыв детали от оригинального метода.

```
1  method getOrderTotal(order) is
2      total = 0
3      foreach item in order.lineItems
4          total += item.price * item.quantity
5
6      total += total * getTaxAmount(order.country)
7
8      return total
9
10 method getTaxAmount(country) is
11     if (country == "US")
12         return 0.07 // US sales tax
13     else if (country == "EU")
14         return 0.20 // European VAT
15     else
16         return 0
```

*ПОСЛЕ: размер налога можно получить, вызвав один метод.*

Теперь изменения налогов будут изолированы в рамках одного метода. Более того, если логика вычисления налогов станет ещё более сложной, вам будет легче извлечь этот метод в собственный класс.

## Пример инкапсуляции на уровне класса

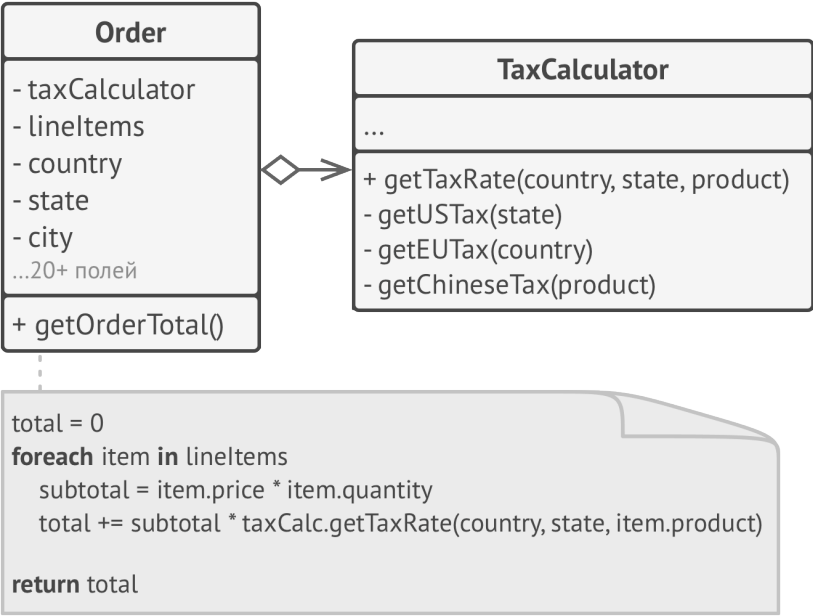
Извлечь логику налогов в собственный класс? Если логика налогов стала слишком сложной, то почему бы и нет?





ДО: вычисление налогов в классе заказов.

Объекты заказов станут делегировать вычисление налогов отдельному объекту-калькулятору налогов.



ПОСЛЕ: вычисление налогов скрыто в классе заказов.

**27 страниц**

из полной книги пропущена в демо-версии

# **КАТАЛОГ ПАТТЕРНОВ**

# Порождающие паттерны

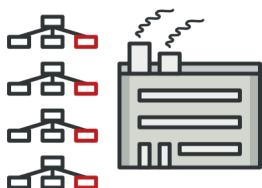
Эти паттерны отвечают за удобное и безопасное создание новых объектов или даже целых семейств объектов.



## Фабричный Метод

Factory Method

Определяет общий интерфейс для создания объектов в суперклассе, позволяя подклассам изменять тип создаваемых объектов.



## Абстрактная Фабрика

Abstract Factory

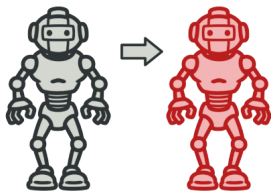
Позволяет создавать семейства связанных объектов, не привязываясь к конкретным классам создаваемых объектов.



## Строитель

Builder

Позволяет создавать сложные объекты пошагово. Строитель даёт возможность использовать один и тот же код строительства для получения разных представлений объектов.



## Прототип

Prototype

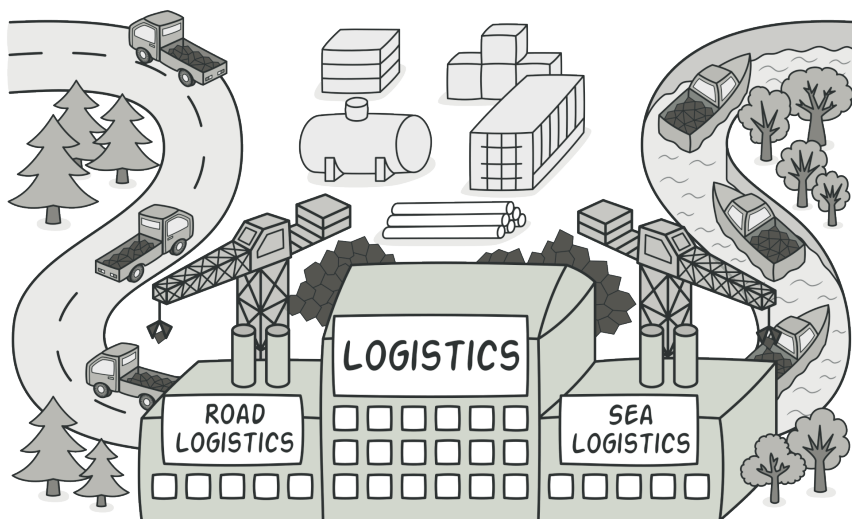
Позволяет копировать объекты, не вдаваясь в подробности их реализации.



## Одиночка

Singleton

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



# ФАБРИЧНЫЙ МЕТОД

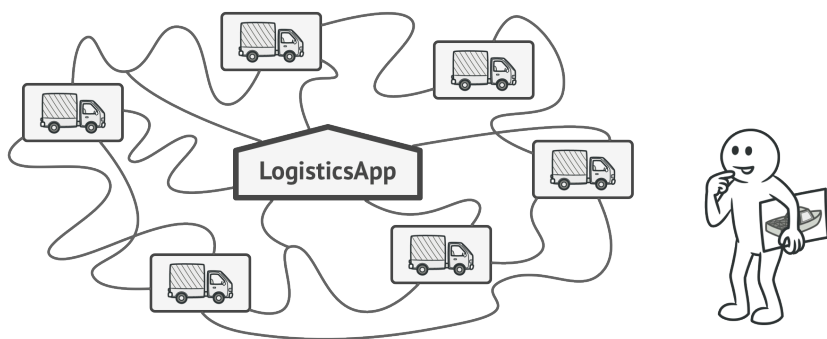
*Также известен как: Виртуальный конструктор, Factory Method*

**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 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.



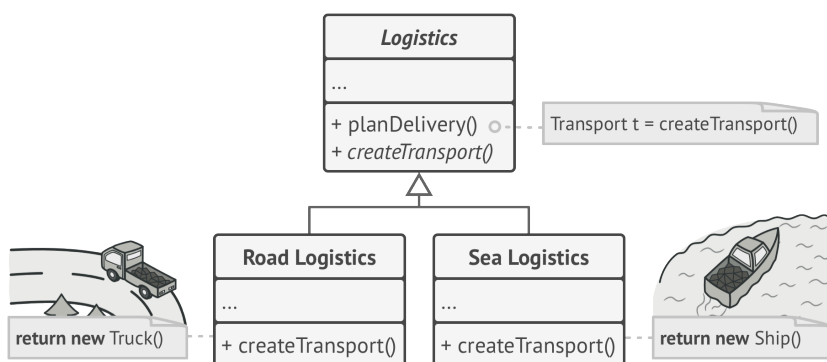
*Добавить новый класс не так-то просто, если весь код уже завязан на конкретные классы.*

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 ещё один вид транспорта, то всю эту работу придётся повторить.

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

## 😊 Решение

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.

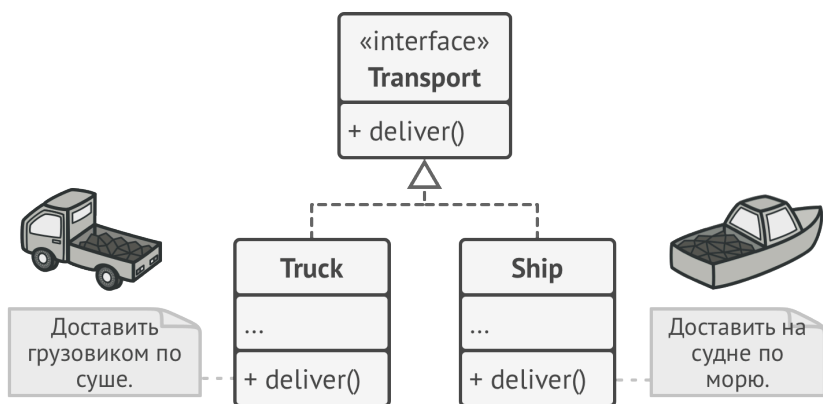


*Подклассы могут изменять класс создаваемых объектов.*

На первый взгляд, это может показаться бессмысленным: мы просто переместили вызов конструктора из одного конца программы в другой. Но теперь вы сможете переопределить фабричный метод в подклассе, чтобы изменить тип создаваемого продукта.



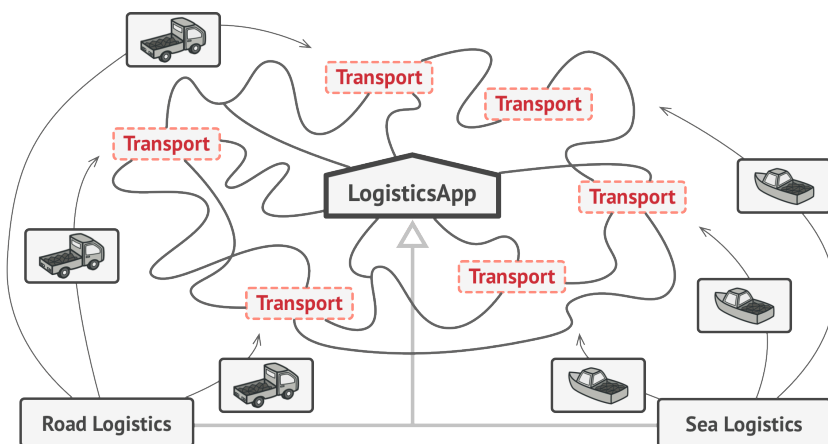
Чтобы эта система заработала, все возвращаемые объекты должны иметь общий интерфейс. Подклассы смогут производить объекты различных классов, следующих одному и тому же интерфейсу.



*Все объекты-продукты должны иметь общий интерфейс.*

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.

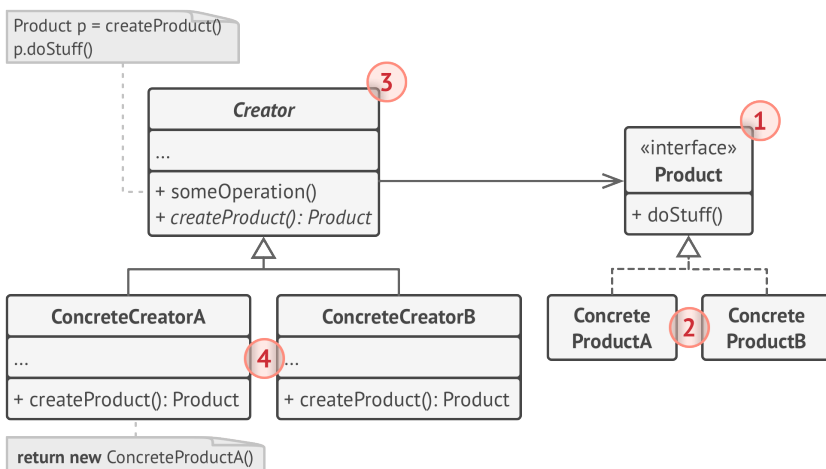
Для клиента фабричного метода нет разницы между этими объектами, так как он будет трактовать их как некий абстрактный `Транспорт`.



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

Для него будет важно, чтобы объект имел метод `доставить`, а как конкретно он работает — не важно.

## Структура



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.

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.

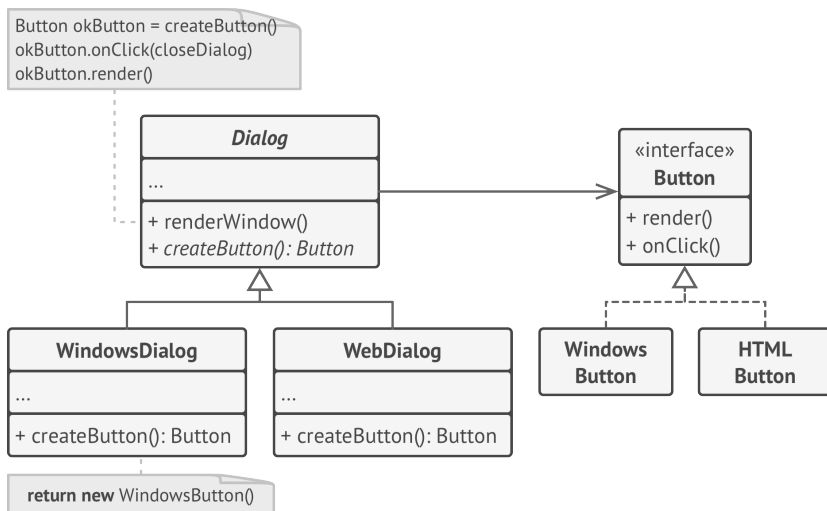
4. **Конкретные создатели** по-своему реализуют фабричный метод, производя те или иные конкретные продукты.

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

## # Псевдокод

В этом примере **Фабричный метод** помогает создавать кросс-платформенные элементы интерфейса, не привязывая основной код программы к конкретным классам элементов.

Фабричный метод объявлен в классе диалогов. Его подклассы относятся к различным операционным системам. Благодаря фабричному методу, вам не нужно переписывать логику диалогов под каждую систему. Подклассы могут наследовать почти весь код из базового диалога, изменяя типы кнопок и других элементов, из которых базовый код строит окна графического пользовательского интерфейса.



Пример кросс-платформенного диалога.

Базовый класс диалогов работает с кнопками через их общий программный интерфейс. Поэтому, какую вариацию кнопок бы ни вернул фабричный метод, диалог останется рабочим. Базовый класс не зависит от конкретных классов кнопок, оставляя подклассам решение о том, какой тип кнопок создавать.

Такой подход можно применить и для создания других элементов интерфейса. Хотя каждый новый тип элементов будет приближать вас к Абстрактной фабрике.

```
1  // Паттерн Фабричный метод применим тогда, когда в программе
2  // есть иерархия классов продуктов.
3  interface Button is
4      method render()
5      method onClick(f)
6
7  class WindowsButton implements Button is
8      method render(a, b) is
9          // Отрисовать кнопку в стиле Windows.
10     method onClick(f) is
11         // Навесить на кнопку обработчик событий Windows.
12
13  class HTMLButton implements Button is
14     method render(a, b) is
15         // Вернуть HTML-код кнопки.
16     method onClick(f) is
17         // Навесить на кнопку обработчик события браузера.
18
19
```

```

20 // Базовый класс фабрики. Заметьте, что "фабрика" – это всего
21 // лишь дополнительная роль для класса. Скорее всего, он уже
22 // имеет какую-то бизнес-логику, в которой требуется создание
23 // разнообразных продуктов.
24 class Dialog is
25     method renderWindow() is
26         // Чтобы использовать фабричный метод, вы должны
27         // убедиться в том, что эта бизнес-логика не зависит от
28         // конкретных классов продуктов. Button – это общий
29         // интерфейс кнопок, поэтому все хорошо.
30         Button okButton = createButton()
31         okButton.onClick(closeDialog)
32         okButton.render()
33
34         // Мы выносим весь код создания продуктов в особый метод,
35         // который называют "фабричным".
36         abstract method createButton()
37
38
39 // Конкретные фабрики переопределяют фабричный метод и
40 // возвращают из него собственные продукты.
41 class WindowsDialog extends Dialog is
42     method createButton() is
43         return new WindowsButton()
44
45 class WebDialog extends Dialog is
46     method createButton() is
47         return new HTMLButton()
48
49
50 class Application is
51     field dialog: Dialog

```

```

52 // Приложение создаёт определённую фабрику в зависимости от
53 // конфигурации или окружения.
54 method initialize() is
55     config = readApplicationConfigFile()
56
57     if (config.OS == "Windows") then
58         dialog = new WindowsDialog()
59     else if (config.OS == "Web") then
60         dialog = new WebDialog()
61     else
62         throw new Exception("Error! Unknown operating system.")
63
64 // Если весь остальной клиентский код работает с фабриками и
65 // продуктами только через общий интерфейс, то для него
66 // будет не важно, какая фабрика была создана изначально.
67 method main() is
68     this.initialize()
69     dialog.render()

```



## Применимость



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



Фабричный метод отделяет код производства продуктов от остального кода, который эти продукты использует.

Благодаря этому, код производства можно расширять, не трогая основной. Так, чтобы добавить поддержку нового продукта, вам нужно создать новый подкласс и определить

в нём фабричный метод, возвращая оттуда экземпляр нового продукта.

**✂** **Когда вы хотите дать возможность пользователям расширять части вашего фреймворка или библиотеки.**


**⚡** Пользователи могут расширять классы вашего фреймворка через наследование. Но как сделать так, чтобы фреймворк создавал объекты из этих новых классов, а не из стандартных?


Решением будет дать пользователям возможность расширять не только желаемые компоненты, но и классы, которые создают эти компоненты. А для этого создающие классы должны иметь конкретные создающие методы, которые можно определить.

Например, вы используете готовый UI-фреймворк для своего приложения. Но вот беда — требуется иметь круглые кнопки, вместо стандартных прямоугольных. Вы создаёте класс `RoundButton`. Но как сказать главному классу фреймворка `UIFramework`, чтобы он теперь создавал круглые кнопки, вместо стандартных?

Для этого вы создаёте подкласс `UIWithRoundButtons` из базового класса фреймворка, переопределяете в нём метод создания кнопки (а-ля `createButton`) и вписываете туда создание своего класса кнопок. Затем используете `UIWithRoundButtons` вместо стандартного `UIFramework`.



 **Когда вы хотите экономить системные ресурсы, повторно используя уже созданные объекты, вместо порождения новых.**

 Такая проблема обычно возникает при работе с тяжёлыми ресурсоёмкими объектами, такими, как подключение к базе данных, файловой системе и т. д.

Представьте, сколько действий вам нужно совершить, чтобы повторно использовать существующие объекты:

1. Сначала вам следует создать общее хранилище, чтобы хранить в нём все создаваемые объекты.
2. При запросе нового объекта нужно будет заглянуть в хранилище и проверить, есть ли там неиспользуемый объект.
3. А затем вернуть его клиентскому коду.
4. Но если свободных объектов нет — создать новый, не забыв добавить его в хранилище.

Весь этот код нужно куда-то поместить, чтобы не засорять клиентский код.

Самым удобным местом был бы конструктор объекта, ведь все эти проверки нужны только при создании объектов. Но, увы, конструктор всегда создаёт **новые** объекты, он не может вернуть существующий экземпляр.

Значит, нужен другой метод, который бы отдавал как существующие, так и новые объекты. Им и станет фабричный метод.

## Шаги реализации

1. Приведите все создаваемые продукты к общему интерфейсу.
2. В классе, который производит продукты, создайте пустой фабричный метод. В качестве возвращаемого типа укажите общий интерфейс продукта.
3. Затем пройдитесь по коду класса и найдите все участки, создающие продукты. Поочерёдно замените эти участки вызовами фабричного метода, перенося в него код создания различных продуктов.

В фабричный метод, возможно, придётся добавить несколько параметров, контролирующих, какой из продуктов нужно создать.

На этом этапе фабричный метод, скорее всего, будет выглядеть удручающе. В нём будет жить большой условный оператор, выбирающий класс создаваемого продукта. Но не волнуйтесь, мы вот-вот исправим это.

4. Для каждого типа продуктов заведите подкласс и переопределите в нём фабричный метод. Переместите туда код создания соответствующего продукта из суперкласса.

5. Если создаваемых продуктов слишком много для существующих подклассов создателя, вы можете подумать о введении параметров в фабричный метод, которые позволят возвращать различные продукты в пределах одного подкласса.

Например, у вас есть класс `Почта` с подклассами `АвиаПочта` и `НаземнаяПочта`, а также классы продуктов `Самолёт`, `Грузовик` и `Поезд`. `Авиа` соответствует `Самолётам`, но для `НаземнойПочты` есть сразу два продукта. Вы могли бы создать новый подкласс почты для поездов, но проблему можно решить и по-другому. Клиентский код может передавать в фабричный метод `НаземнойПочты` аргумент, контролирующий тип создаваемого продукта.

6. Если после всех перемещений фабричный метод стал пустым, можете сделать его абстрактным. Если в нём что-то осталось — не беда, это будет его реализацией по умолчанию.



## Преимущества и недостатки

- ✓ 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*.

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

## ⇔ 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).
- Классы Абстрактной фабрики чаще всего реализуются с помощью Фабричного метода, хотя они могут быть построены и на основе Прототипа.
- Фабричный метод можно использовать вместе с Итератором, чтобы подклассы коллекций могли создавать подходящие им итераторы.
- Прототип не опирается на наследование, но ему нужна сложная операция инициализации. Фабричный метод, наоборот, построен на наследовании, но не требует сложной инициализации.
- Фабричный метод можно рассматривать как частный случай Шаблонного метода. Кроме того, *Фабричный метод* нередко бывает частью большого класса с *Шаблонными методами*.

**310 страниц**

из полной книги пропущена в демо-версии