

# КОМПОНОВЩИК

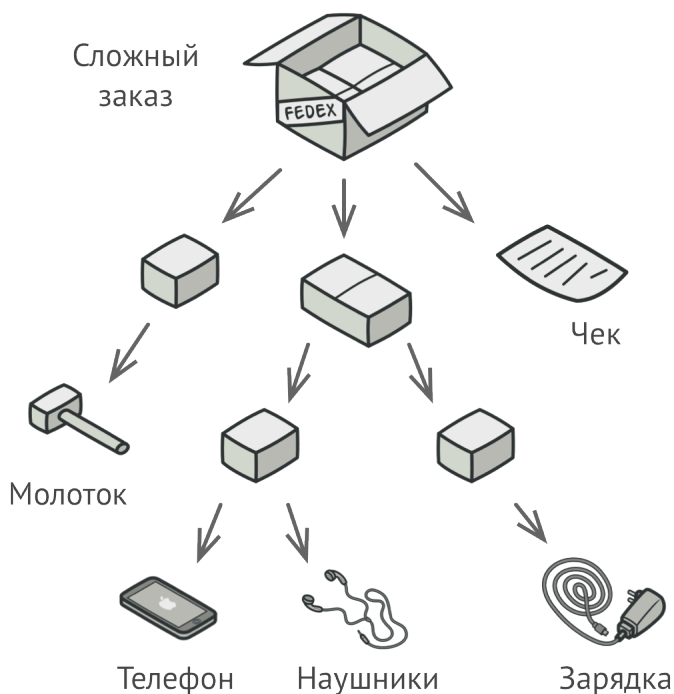
*Также известен как: Дерево, Composite*

**Компоновщик** — это структурный паттерн проектирования, который позволяет сгруппировать объекты в древовидную структуру, а затем работать с ними так, если бы это был единичный объект.

## ☹ Проблема

Паттерн Компоновщик имеет смысл только тогда, когда основная модель вашей программы может быть структурирована в виде дерева.

Например, есть два объекта — `Продукт` и `Коробка`. `Коробка` может содержать несколько `Продуктов` и других `Коробок` поменьше. Те, в свою очередь, тоже содержат либо `Продукты`, либо `Коробки` и так далее.



*Заказ может состоять из различных позиций, упакованных в собственные коробки.*

Теперь, предположим, ваши `Продукты` и `Коробки` могут быть частью заказов. Ваша задача в том, чтобы узнать цену всего заказа. Причём в заказе может быть как просто `Продукт` без упаковки, так и пустая или составная `Коробка` .

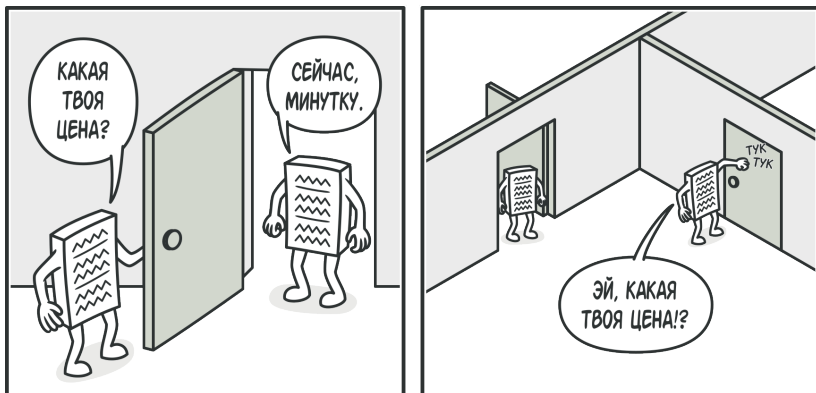
Если решать задачу в лоб, то вам потребуется открыть все коробки заказа, перебрать все продукты и посчитать их суммарную цену. Но это слишком хлопотно, так как типы коробок и их содержимого могут быть вам неизвестны. Кроме того, наперёд неизвестно и количество уровней вложенности коробок, поэтому перебрать коробки простым циклом не выйдет.

## Решение

Компоновщик предлагает рассматривать `Продукт` и `Коробку` через единый интерфейс с общим методом получения цены.

`Продукт` просто вернёт свою цену. `Коробка` спросит цену каждого предмета внутри себя и вернёт сумму результатов.

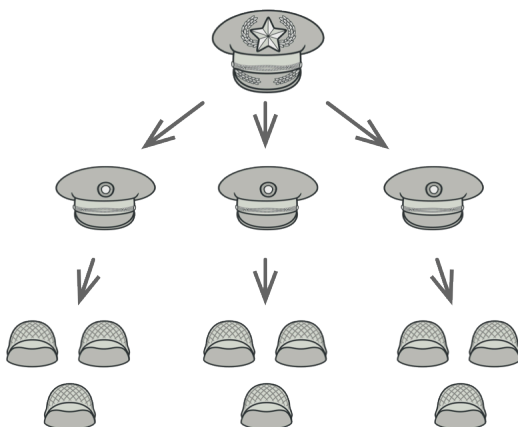
Если одним из внутренних предметов окажется коробка поменьше, она тоже будет перебирать своё содержимое, и так далее, пока не посчитаются все составные части.



*Компоновщик рекурсивно запускает действие по все элементы дерева от корня к листьям.*

Для вас, клиента, главное, что теперь не нужно ничего знать о структуре заказов. Вы вызываете метод получения цены, он возвращает цифру, а вы не тонете в горах картона и скотча.

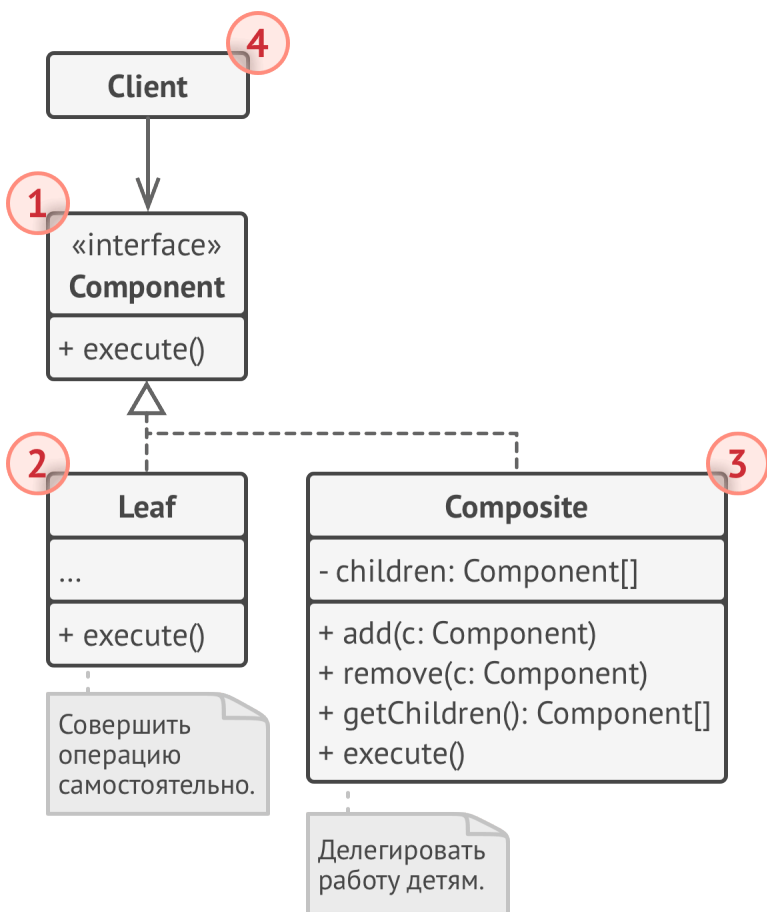
## Аналогия из жизни



*Пример армейской структуры.*

Армии большинства государств могут быть представлены в виде перевёрнутых деревьев. На нижнем уровне у вас есть солдаты, затем взводы, затем полки, а затем целые армии. Приказы отдаются сверху и спускаются вниз по структуре командования, пока не доходят до конкретного солдата.

## 🏰 Структура



1. **Компонент** определяет общий интерфейс для простых и составных компонентов дерева.
2. **Лист** – это простой элемент дерева, не имеющий ответвлений.

Из-за того, что им некому больше передавать выполнение, классы Листьев будут содержать большую часть полезного кода.

3. **Контейнер** (или «композит») — это составной элемент дерева. Он содержит набор дочерних компонентов, но ничего не знает об их типах. Это могут быть как простые компоненты-листья, так и другие компоненты-контейнеры. Но это не является проблемой, так как все дочерние элементы следуют общему интерфейсу.

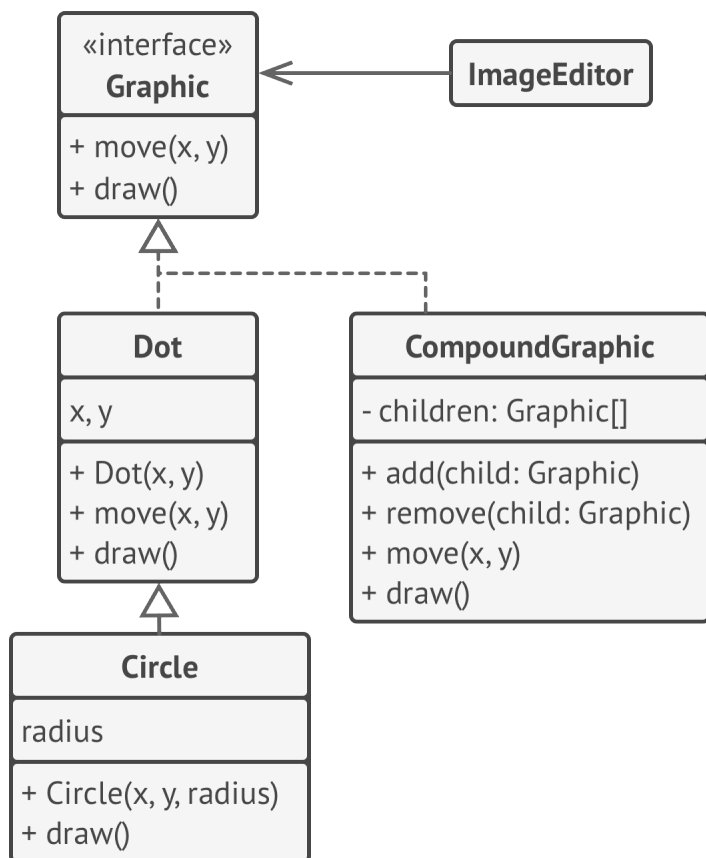
Методы контейнера переадресуют основную работу своим дочерним компонентам, хотя и могут добавлять что-то своё к результату.

4. **Клиент** работает с деревом через общий интерфейс компонентов.

Благодаря этому, клиенту без разницы что перед ним находится — простой или составной компонент дерева.

## # Псевдокод

В этом примере **Компоновщик** помогает реализовать вложенные геометрические фигуры.



*Пример редактора геометрических фигур.*

Класс `CompoundGraphic` может содержать любое количество подфигур, включая такие же контейнеры, как он сам. Контейнер реализует те же методы, что и простые фигуры. Но вместо непосредственного действия, он передаёт

вызовы всем вложенным компонентам, используя рекурсию. Затем он как бы «суммирует» результаты всех вложенных фигур.

Клиентский код работает со всеми фигурами через общий интерфейс фигур и не знает что перед ним — простая фигура или составная. Это позволяет клиентскому коду работать с деревьями объектов любой сложности, не привязываясь к конкретным классам объектов, формирующих дерево.

```
1  // Общий интерфейс компонентов.
2  interface Graphic is
3      method move(x, y)
4      method draw()
5
6  // Простой компонент.
7  class Dot implements Graphic is
8      field x, y
9
10     constructor Dot(x, y) { ... }
11
12     method move(x, y) is
13         this.x += x, this.y += y
14
15     method draw() is
16         // Нарисовать точку в координате X, Y.
17
18 // Компоненты могут расширять другие компоненты.
19 class Circle extends Dot is
20     field radius
21
```



```

22     constructor Circle(x, y, radius) { ... }
23
24     method draw() is
25         // Нарисовать окружность в координате X, Y и радиусом R.
26
27         // Контейнер содержит операции добавления/удаления дочерних
28         // компонентов. Все стандартные операции интерфейса
29         // компонентов он делегирует каждому из
30         // дочерних компонентов.
31     class CompoundGraphic implements Graphic is
32         field children: array of Graphic
33
34         method add(child: Graphic) is
35             // Добавить компонент в список дочерних.
36
37         method remove(child: Graphic) is
38             // Убрать компонент из списка дочерних.
39
40         method move(x, y) is
41             foreach (child in children) do
42                 child.move(x, y)
43
44         method draw() is
45             // 1. Для каждого дочернего компонента:
46             //     - Отрисовать компонент.
47             //     - Определить координаты максимальной границы.
48             // 2. Нарисовать пунктирную границу вокруг всей области.
49
50
51         // Приложение работает единообразно как с единичными
52         // компонентами, так и целыми группами компонентов.
53     class ImageEditor is
54         method load() is
55             all = new CompoundGraphic()

```

```

56     all.add(new Dot(1, 2))
57     all.add(new Circle(5, 3, 10))
58     // ...
59
60     // Группировка выбранных компонентов в один
61     // сложный компонент.
62     method groupSelected(components: array of Graphic) is
63         group = new CompoundGraphic()
64         group.add(components)
65         all.remove(components)
66         all.add(group)
67         // Все компоненты будут отрисованы.
68         all.draw()

```



## Применимость




Когда вам нужно представить древовидную структуру объектов.



Паттерн Компоновщик предлагает хранить в составных объектах ссылки на другие простые или составные объекты. Те, в свою очередь, тоже могут хранить свои вложенные объекты и так далее. В итоге вы можете строить сложную древовидную структуру данных, используя всего две основные разновидности объектов.



Когда клиенты должны единообразно трактовать простые и составные объекты.

 Благодаря тому, что простые и составные объекты реализуют общий интерфейс, клиенту безразлично с каким именно объектом ему предстоит работать.

## Шаги реализации

1. Убедитесь, что вашу бизнес-логику можно представить как древовидную структуру. Попробуйте разбить её на простые элементы и контейнеры. Помните, что контейнеры могут содержать как простые элементы, так и другие контейнеры.
2. Создайте общий интерфейс компонентов, который объединит операции контейнеров и простых элементов дерева. Интерфейс будет удачным, если вы сможете взаимозаменять простые и составные компоненты без потери смысла.
3. Создайте класс компонентов-листьев, не имеющих дальнейших ответвлений. Имейте в виду, что программа может содержать несколько видов таких классов.
4. Создайте класс компонентов-контейнеров, и добавьте в него массив для хранения ссылок на вложенные компоненты. Этот массив должен быть способен содержать как простые, так и составные компоненты, поэтому убедитесь, что он объявлен с типом интерфейса компонентов.

Реализуйте в контейнере методы интерфейса компонентов, помня о том, что контейнеры должны делегировать основную работу своим дочерним компонентам.

5. Добавьте операции добавления и удаления дочерних элементов в класс контейнеров.

Имейте в виду, что методы добавления/удаления дочерних элементов можно поместить и в интерфейс компонентов. Да, это нарушит *принцип разделения интерфейса*, так как реализации методов будут пустыми в компонентах-листьях. Но зато все компоненты дерева станут действительно одинаковыми для клиента.



## Преимущества и недостатки

- ✓ Упрощает архитектуру клиента при работе со сложным деревом компонентов.
- ✓ Облегчает добавление новых видов компонентов.
- ✗ Создаёт слишком общий дизайн классов.



## Отношения с другими паттернами

- Строитель позволяет пошагово сооружать дерево **Компоновщика**.

- **Цепочку обязанностей** часто используют вместе с **Компоновщиком**. В этом случае, запрос передаётся от дочерних компонентов к их родителям.
- Вы можете обходить дерево **Компоновщика**, используя **Итератор**.
- Вы можете выполнить какое-то действие над всем деревом **Компоновщика** при помощи **Посетителя**.
- **Компоновщик** часто совмещают с **Легковесом**, чтобы реализовать общие ветки дерева и сэкономить при этом память.
- **Компоновщик** и **Декоратор** имеют похожие структуры классов из-за того, что оба построены на рекурсивной вложенности. Она позволяет связать в одну структуру бесконечное количество объектов.

*Декоратор* оборачивает только один объект, а узел *Компоновщика* может иметь много детей. *Декоратор* добавляет вложенному объекту новую функциональность, а *Компоновщик* не добавляет ничего нового, но «суммирует» результаты всех своих детей.

Но они могут и сотрудничать: *Компоновщик* может использовать *Декоратор*, чтобы переопределить функции отдельных частей дерева компонентов.

- Архитектура, построенная на **Компоновщиках** и **Декораторах**, часто может быть улучшена за счёт внедрения **Прототипа**. Он позволяет клонировать сложные структуры объектов, а не собирать их заново.