



НОВОГОДНЯЯ РАСПРОДАЖА!



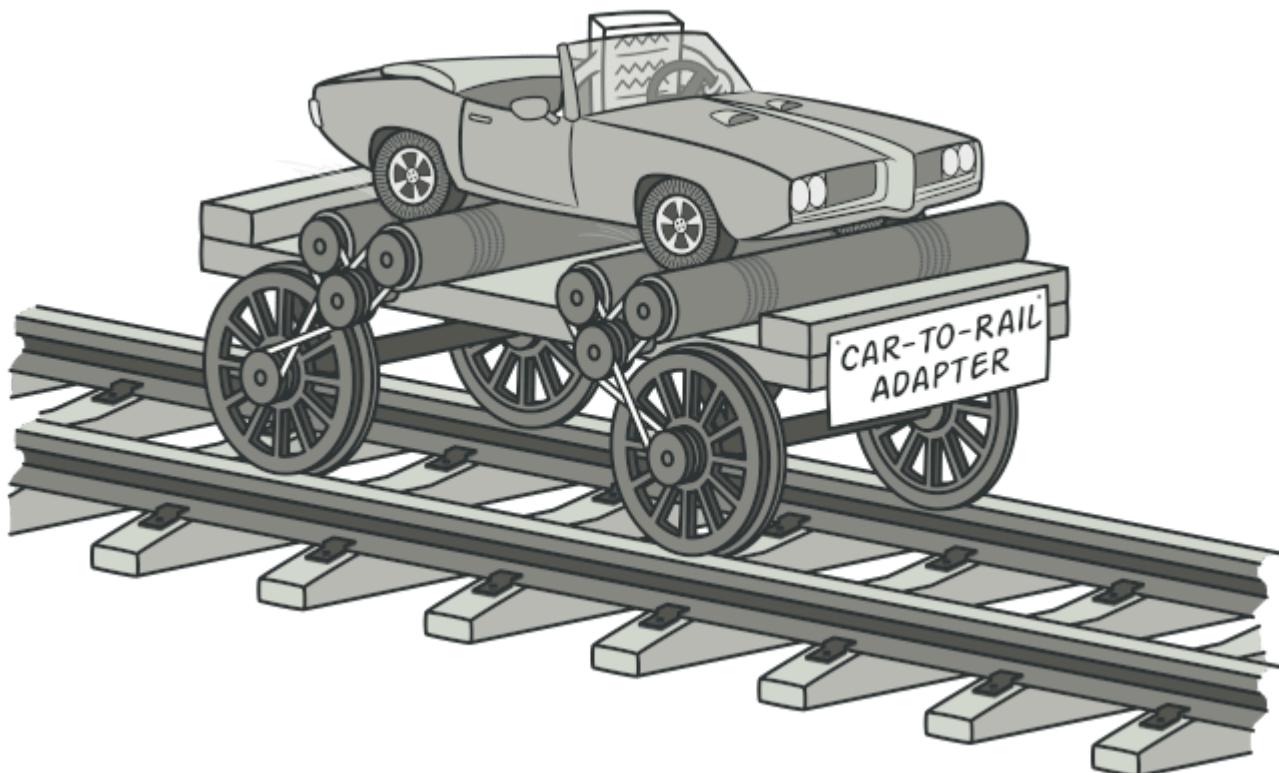
[Home](#) / Паттерны проектирования / Структурные паттерны

Адаптер

Также известен как: Wrapper, Обёртка, Adapter

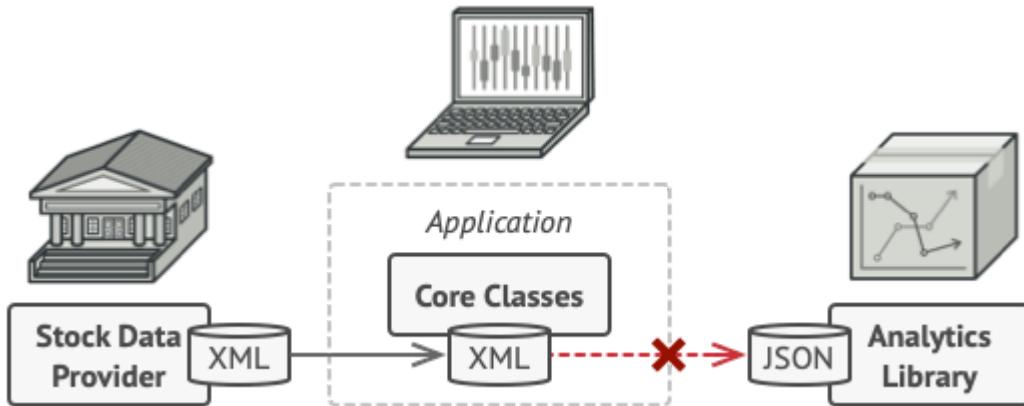
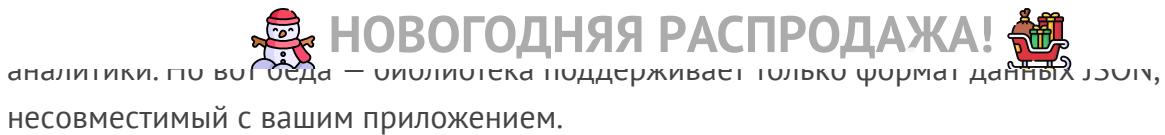
💬 Суть паттерна

Адаптер – это структурный паттерн проектирования, который позволяет объектам с несовместимыми интерфейсами работать вместе.



😢 Проблема

Представьте, что вы делаете приложение для торговли на бирже. Ваше приложение скачивает биржевые котировки из нескольких источников в XML, а затем рисует красивые графики.



Подключить стороннюю библиотеку не выйдет из-за несовместимых форматов данных.

Вы смогли бы переписать библиотеку, чтобы та поддерживала формат XML. Но, во-первых, это может нарушить работу существующего кода, который уже зависит от библиотеки. А во-вторых, у вас может просто не быть доступа к её исходному коду.

😊 Решение

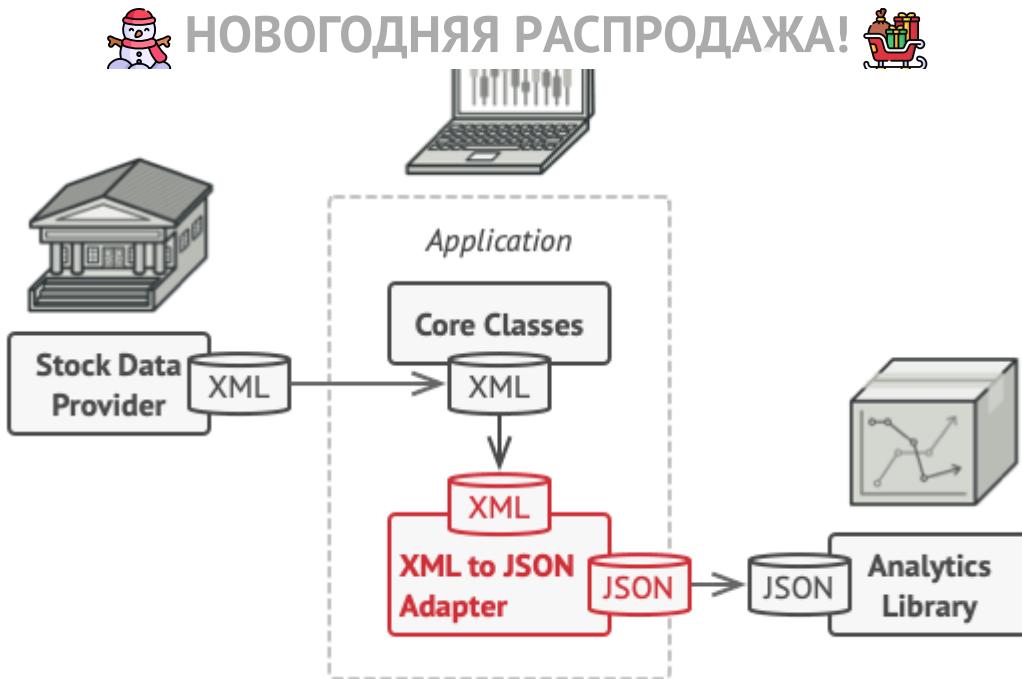
Вы можете создать *адаптер*. Это объект-переводчик, который трансформирует интерфейс или данные одного объекта в такой вид, чтобы он стал понятен другому объекту.

При этом адаптер обворачивает один из объектов, так что другой объект даже не знает о наличии первого. Например, вы можете обернуть объект, работающий в метрах, адаптером, который бы конвертировал данные в футы.

Адаптеры могут не только переводить данные из одного формата в другой, но и помогать объектам с разными интерфейсами работать сообща. Это работает так:

1. Адаптер имеет интерфейс, который совместим с одним из объектов.
2. Поэтому этот объект может свободно вызывать методы адаптера.
3. Адаптер получает эти вызовы и перенаправляет их второму объекту, но уже в том формате и последовательности, которые понятны второму объекту.

Иногда возможно создать даже *двухсторонний адаптер*, который работал бы в обе стороны.

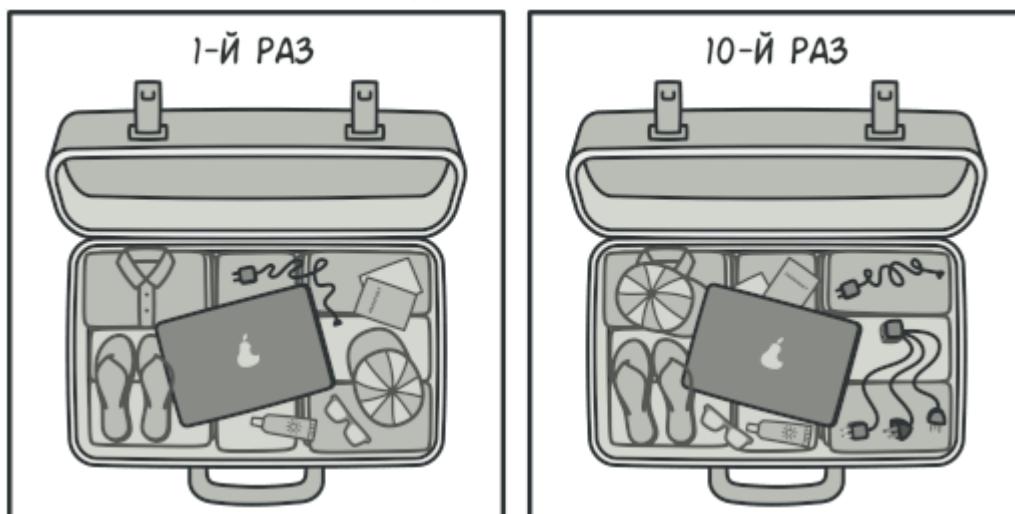


Программа может работать со сторонней библиотекой через адаптер.

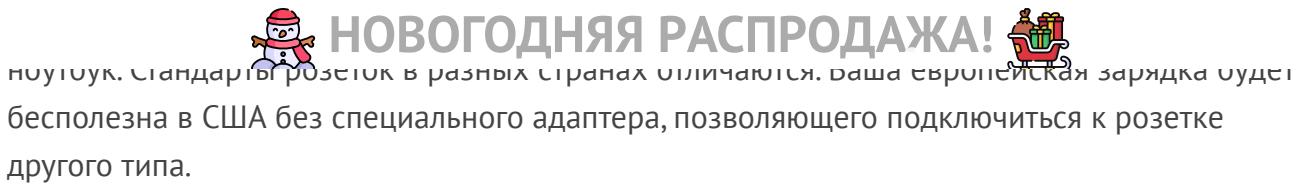
Таким образом, в приложении биржевых котировок вы могли бы создать класс `XML_To_JSON_Adapter`, который бы оборачивал объект того или иного класса библиотеки аналитики. Ваш код посыпал бы адаптеру запросы в формате XML, а адаптер сначала транслировал входящие данные в формат JSON, а затем передавал бы их методам обёрнутого объекта аналитики.

🚗 Аналогия из жизни

ПОЕЗДКА ЗА ГРАНИЦУ



Содержимое чемоданов до и после поездки за границу.

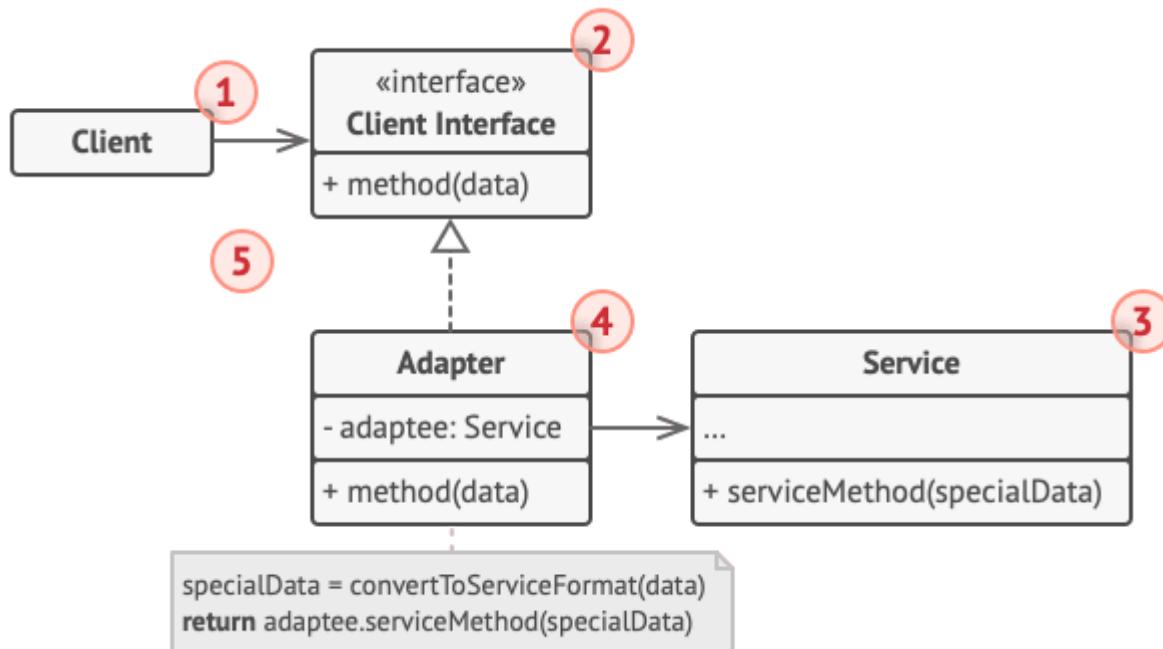


STRUCTURE

Структура

Адаптер объектов

Эта реализация использует агрегацию: объект адаптера «обворачивает», то есть содержит ссылку на служебный объект. Такой подход работает во всех языках программирования.

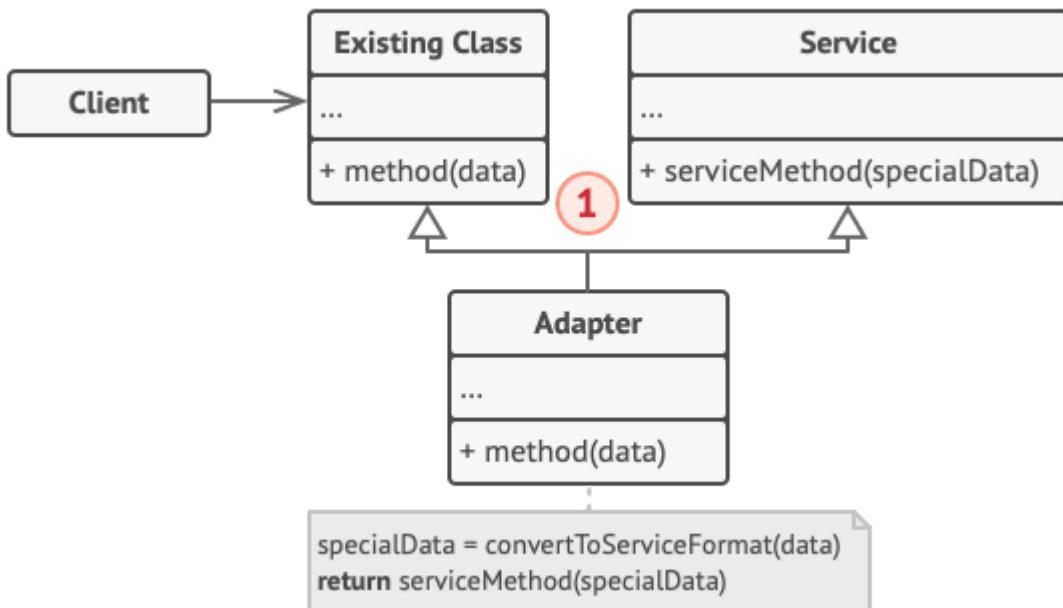


1. **Клиент** – это класс, который содержит существующую бизнес-логику программы.
2. **Клиентский интерфейс** описывает протокол, через который клиент может работать с другими классами.
3. **Сервис** – это какой-то полезный класс, обычно сторонний. Клиент не может использовать этот класс напрямую, так как сервис имеет непонятный ему интерфейс.
4. **Адаптер** – это класс, который может одновременно работать и с клиентом, и с сервисом. Он реализует клиентский интерфейс и содержит ссылку на объект сервиса. Адаптер получает вызовы от клиента через методы клиентского интерфейса, а затем переводит их в вызовы методов обёрнутого объекта в правильном формате.

 **НОВОГОДНЯЯ РАСПРОДАЖА!** 
 адаптера. Удачи! Однако благодаря этому, вы можете добавлять в программу новые виды адаптеров, независимо от клиентского кода. Это может пригодиться, если интерфейс сервиса вдруг изменится, например, после выхода новой версии сторонней библиотеки.

Адаптер классов

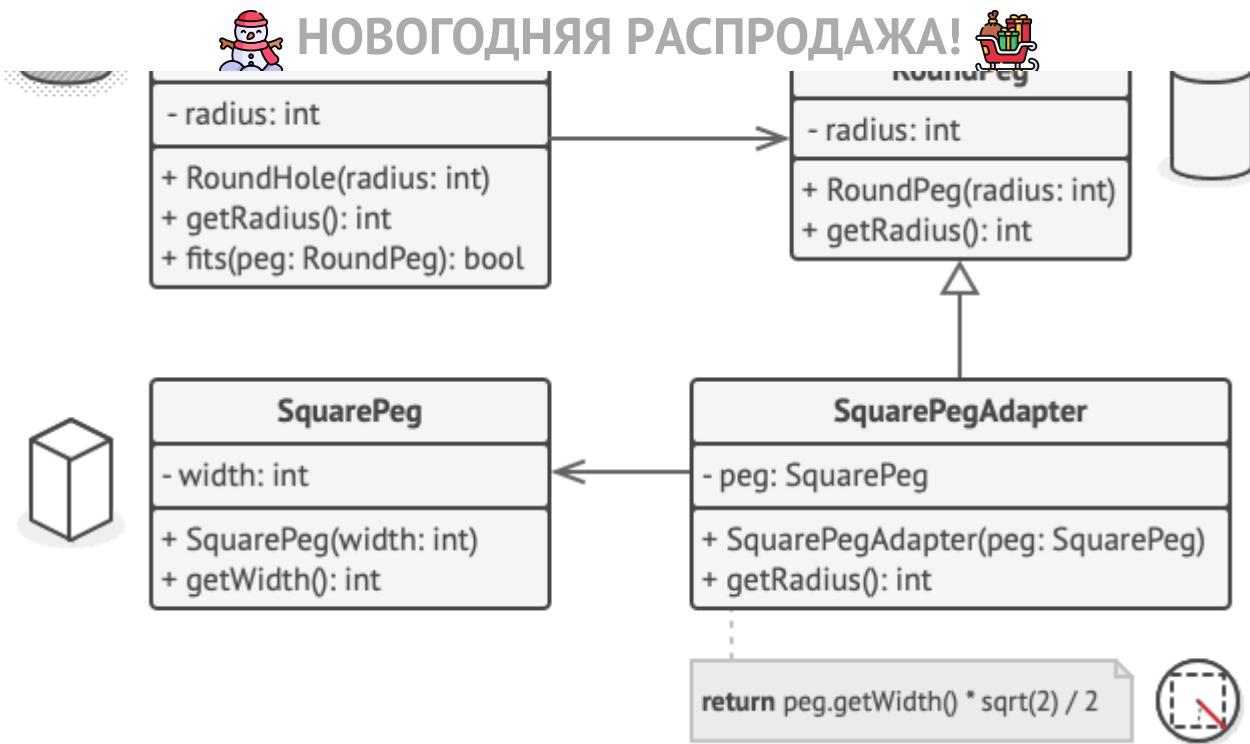
Эта реализация базируется на наследовании: адаптер наследует оба интерфейса одновременно. Такой подход возможен только в языках, поддерживающих множественное наследование, например, C++.



1. **Адаптер классов** не нуждается во вложенном объекте, так как он может одновременно наследовать и часть существующего класса, и часть сервиса.

Псевдокод

В этом шуточном примере **Адаптер** преобразует один интерфейс в другой, позволяя совместить квадратные колышки и круглые отверстия.



Пример адаптации квадратных колышков и круглых отверстий.

Адаптер вычисляет наименьший радиус окружности, в которую можно вписать квадратный колышек, и представляет его как круглый колышек с этим радиусом.

```

// Классы с совместимыми интерфейсами: КруглоеОтверстие и
// КруглыйКолышек.

class RoundHole is
    constructor RoundHole(radius) { ... }

    method getRadius() is
        // Вернуть радиус отверстия.

    method fits(peg: RoundPeg) is
        return this.getRadius() >= peg.getRadius()


class RoundPeg is
    constructor RoundPeg(radius) { ... }

    method getRadius() is
        // Вернуть радиус круглого колышка.


// Устаревший, несовместимый класс: КвадратныйКолышек.

class SquarePeg is
    constructor SquarePeg(width) { ... }

    method getWidth() is
        // Вернуть ширину квадратного колышка.
  
```



НОВОГОДНЯЯ РАСПРОДАЖА!



```
// отверстия вместе.

class SquarePegAdapter extends RoundPeg {
    private field peg: SquarePeg

    constructor(peg: SquarePeg) {
        this.peg = peg
    }

    method getRadius() {
        // Вычислить половину диагонали квадратного колышка по
        // теореме Пифагора.
        return peg.getWidth() * Math.sqrt(2) / 2
    }
}

// Где-то в клиентском коде.

hole = new RoundHole(5)
rpeg = new RoundPeg(5)
hole.fits(rpeg) // TRUE

small_sqpeg = new SquarePeg(5)
large_sqpeg = new SquarePeg(10)
hole.fits(small_sqpeg) // Ошибка компиляции, несовместимые типы

small_sqpeg_adapter = new SquarePegAdapter(small_sqpeg)
large_sqpeg_adapter = new SquarePegAdapter(large_sqpeg)
hole.fits(small_sqpeg_adapter) // TRUE
hole.fits(large_sqpeg_adapter) // FALSE
```

💡 Применимость

Когда вы хотите использовать сторонний класс, но его интерфейс не соответствует остальному коду приложения.

Адаптер позволяет создать объект-прокладку, который будет превращать вызовы приложения в формат, понятный стороннему классу.

Когда вам нужно использовать несколько существующих подклассов, но в них не хватает какой-то общей функциональности, причём расширить суперкласс вы не можете.

Вы могли бы создать ещё один уровень подклассов и добавить в них недостающую функциональность. Но при этом придётся дублировать один и тот же код в обеих ветках подклассов.



НОВОГОДНЯЯ РАСПРОДАЖА!



адаптер и присоединить его для работы с суперклассом. Такой адаптер сможет работать со всеми подклассами иерархии. Это решение будет сильно напоминать паттерн [Декоратор](#).

📋 Шаги реализации

1. Убедитесь, что у вас есть два класса с несовместимыми интерфейсами:

- полезный *сервис* – служебный класс, который вы не можете изменять (он либо сторонний, либо от него зависит другой код);
- один или несколько *клиентов* – существующих классов приложения, несовместимых с сервисом из-за неудобного или несовпадающего интерфейса.

2. Опишите клиентский интерфейс, через который классы приложения смогли бы использовать класс сервиса.

3. Создайте класс адаптера, реализовав этот интерфейс.

4. Поместите в адаптер поле, которое будет хранить ссылку на объект сервиса. Обычно это поле заполняют объектом, переданным в конструктор адаптера. В случае простой адаптации этот объект можно передавать через параметры методов адаптера.

5. Реализуйте все методы клиентского интерфейса в адаптере. Адаптер должен делегировать основную работу сервису.

6. Приложение должно использовать адаптер только через клиентский интерфейс. Это позволит легко изменять и добавлять адаптеры в будущем.

⚠️ Преимущества и недостатки

✓ Отделяет и скрывает от клиента подробности преобразования различных интерфейсов.

✗ Усложняет код программы из-за введения дополнительных классов.

↔ Отношения с другими паттернами

- [Мост](#) проектируют загодя, чтобы развивать большие части приложения отдельно друг от друга. [Адаптер](#) применяется постфактум, чтобы заставить несовместимые классы работать



НОВОГОДНЯЯ РАСПРОДАЖА!



- **Адаптер** предоставляет совершенно другой интерфейс для доступа к существующему объекту. С другой стороны, при использовании паттерна **Декоратор** интерфейс либо остается прежним, либо расширяется. Причём **Декоратор** поддерживает рекурсивную вложенность, чего не скажешь об **Адаптере**.
- С **Адаптером** вы получаете доступ к существующему объекту через другой интерфейс. Используя **Заместитель**, интерфейс остается неизменным. Используя **Декоратор**, вы получаете доступ к объекту через расширенный интерфейс.
- **Фасад** задаёт новый интерфейс, тогда как **Адаптер** повторно использует старый. **Адаптер** оборачивает только один класс, а **Фасад** оборачивает целую подсистему. Кроме того, **Адаптер** позволяет двум существующим интерфейсам работать сообща, вместо того, чтобы задать полностью новый.
- **Мост, Стратегия и Состояние** (а также слегка и **Адаптер**) имеют схожие структуры классов — все они построены на принципе «композиции», то есть делегирования работы другим объектам. Тем не менее, они отличаются тем, что решают разные проблемы. Помните, что паттерны — это не только рецепт построения кода определённым образом, но и описание проблем, которые привели к данному решению.

</> Примеры реализации паттерна

