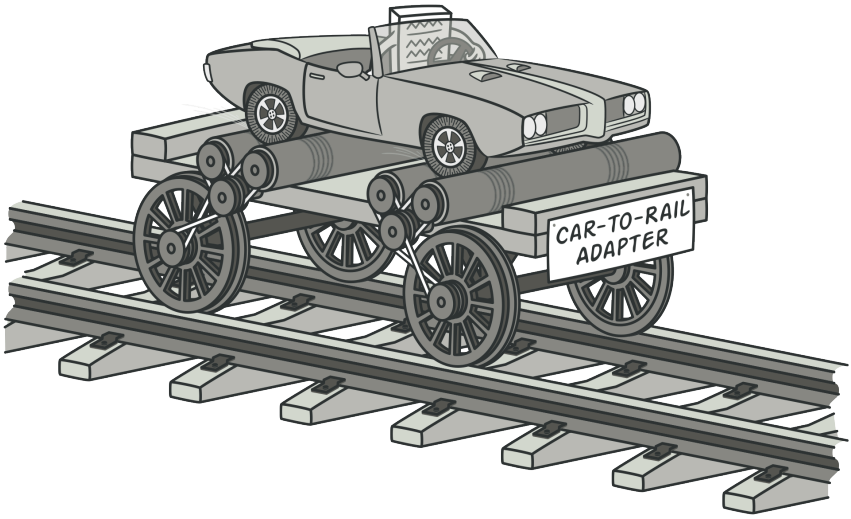


АДАПТЕР

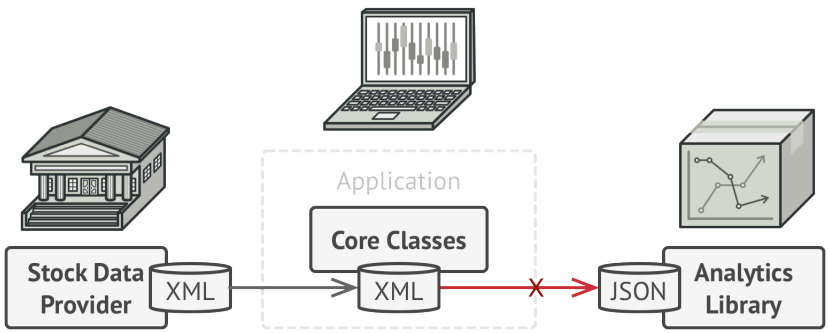
Также известен как: Обёртка, Adapter

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.

В какой-то момент вы решаете улучшить приложение, применив стороннюю библиотеку аналитики. Но вот беда, библиотека поддерживает только формат данных JSON, несовместимый с вашим приложением.



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

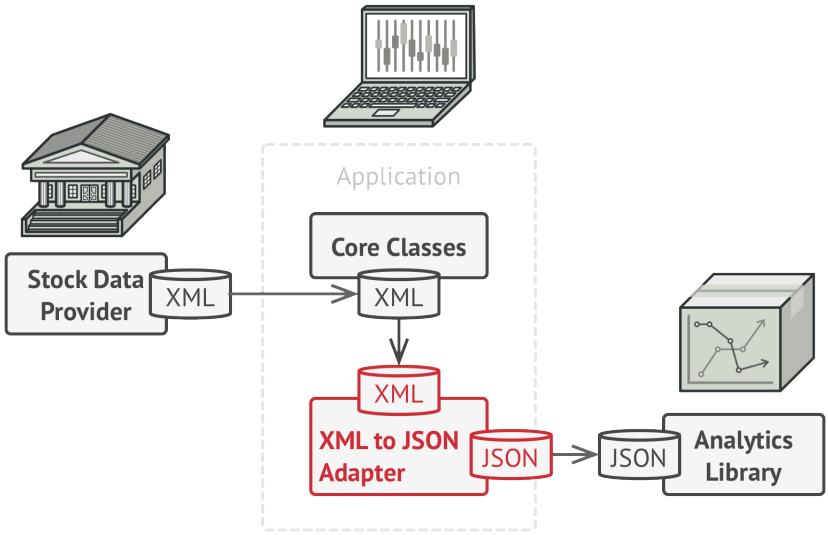
Вы можете создать *адаптер*. Это объект-переводчик, который трансформирует интерфейс или данные одного объекта в такой вид, чтобы он стал понятен другому объекту.

При этом адаптер оборачивает один из объектов, так что другой объект даже не знает о его наличии. 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 следует интерфейсу, который один объект ожидает от другого.
2. Когда первый объект вызывает методы адаптера, адаптер передаёт выполнение второму объекту, вызывая в нём те или иные методы в том порядке, который важен для второго объекта.

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 класс библиотеки аналитики. Ваш код посылал бы запросы этому объекту в формате XML, а адаптер бы сначала транслировал входящие данные в формат JSON, а затем передавал бы их определённым методам библиотеки.

Аналогия из жизни

Когда вы в первый раз летите за границу, вас может ждать сюрприз при попытке зарядить ноутбук. Стандарты розеток в разных странах отличаются.

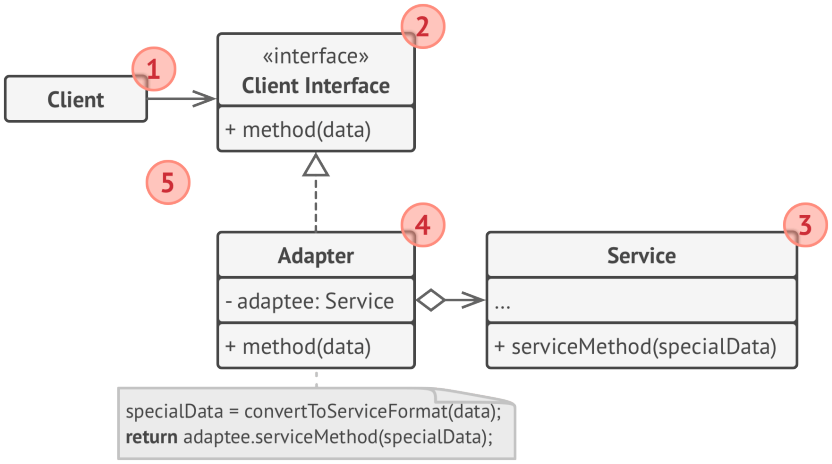
Содержимое чемоданов до и после поездки за границу.

Ваша европейская зарядка будет бесполезна в США без специального адаптера, позволяющего подключиться к розетке другого типа.

Структура

Адаптер объектов

Эта реализация использует композицию: объект адаптера «оборачивает», 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.

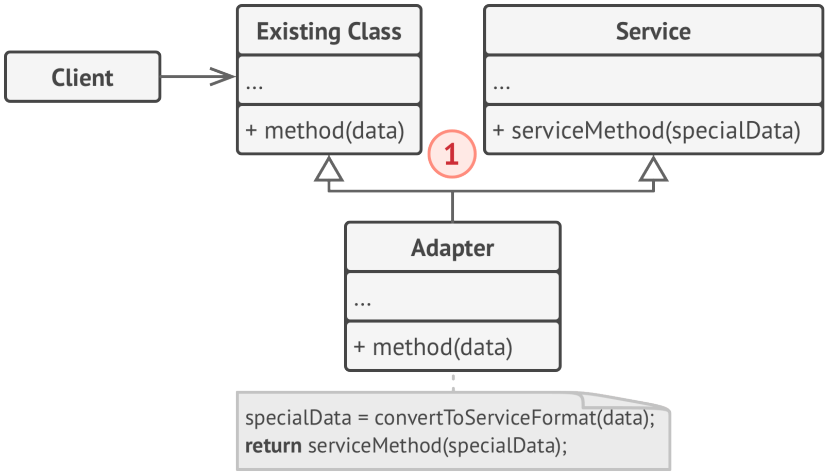
3. **Сервис** – это какой-то полезный класс, обычно сторонний. Клиент не может использовать этот класс напрямую, так как сервис имеет непонятный ему интерфейс.

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.

5. Работая с адаптером через интерфейс, клиент не привязываться к конкретному классу адаптера. 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

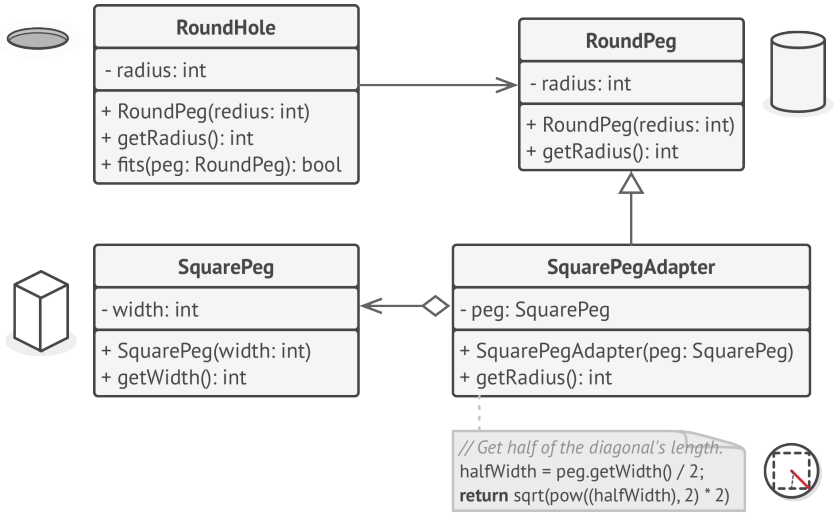
Эта реализация базируется на наследовании: адаптер наследует оба интерфейса одновременно. Такой подход возможен только в языках, поддерживающих множественное наследование, например C++.



1. **Адаптер классов** не нуждается во вложенном объекте, так как он одновременно наследует и существующий и сервисный интерфейс.

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 наименьшую окружность, в которую можно вписать квадратный колышек и представляет его как круглый колышек с таким диаметром.

```
// 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.radius()

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.


// Адаптер позволяет использовать квадратные колышки и
// круглые отверстия вместе.
class SquarePegAdapter extends RoundPeg is
    private field peg: SquarePeg

    constructor SquarePegAdapter(peg: SquarePeg) is
      this.peg = peg

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

```

```
// Где-то в клиентском коде.
hole = new RoundHole(5)
rpeg = new RoundPeg(5)
hole.fits(rpeg) // true

small_sqpeg = new SquarePeg(2)
large_sqpeg = new SquarePeg(5)
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 расширять 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.

Более элегантное решение — поместить недостающую функциональность в адаптер и приспособить его для работы с суперклассом. Такой адаптер сможет работать со всеми подклассами иерархии. Это решение будет сильно напоминать паттерн **Посетитель**.

Шаги реализации

1. Убедитесь, что у вас есть два класса с неудобными интерфейсами:
   - полезный *сервис* — служебный класс, который вы не можете изменять (он либо сторонний, либо от него зависит другой код);
   - один или несколько *клиентов* — классов приложения, несовместимых с сервисом из-за неудобного или несовпадающего интерфейса.
2. Опишите клиентский интерфейс, через который классы приложения смогли бы использовать сторонний класс.
3. Создайте класс адаптера, реализовав этот интерфейс.

4. Поместите в адаптер поле-ссылку на объект-сервис. В большинстве случаев, это поле заполняется 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 за счёт 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 вместе.

- **Адаптер** меняет интерфейс существующего объекта. **Декоратор** улучшает другой объект без изменения его интерфейса. Причём *Декоратор* поддерживает рекурсивную вложенность, чего не скажешь об *Адаптере*.

- **Адаптер** предоставляет классу альтернативный интерфейс. **Декоратор** предоставляет расширенный интерфейс. **Заместитель** предоставляет тот же 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.