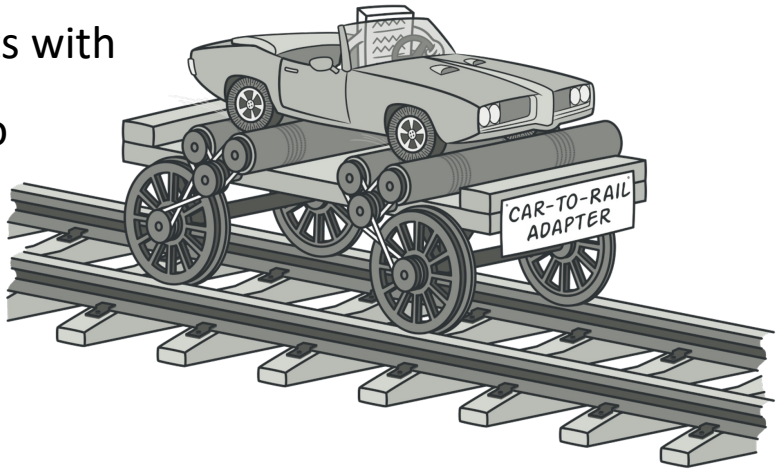
Advanced Software Engineering Part 09 – Adapter Pattern

Dr. Amjad AbuHassan

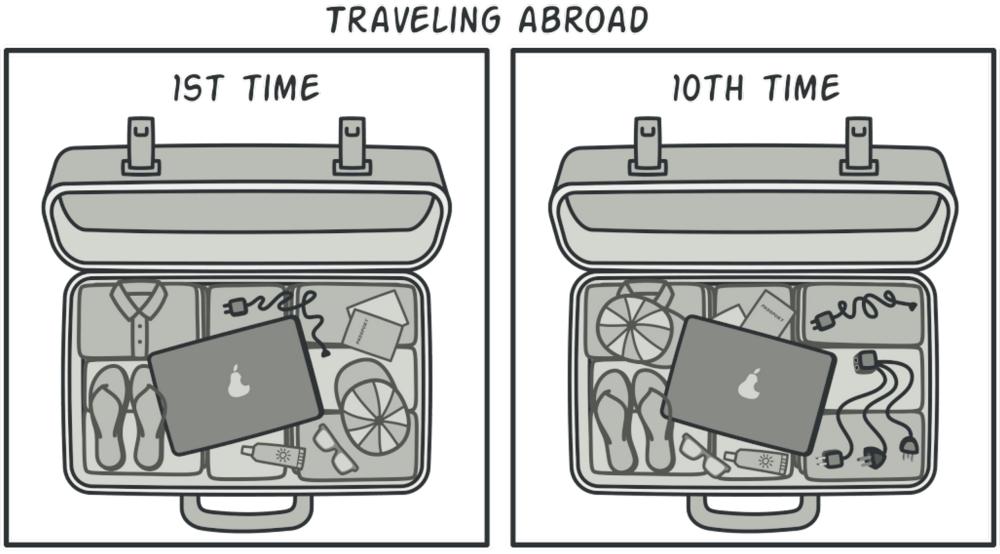
Also Known as: **Wrapper**

Intent

- **Adapter** is a structural design pattern that allows objects with incompatible interfaces to collaborate.



Real-World Analogy

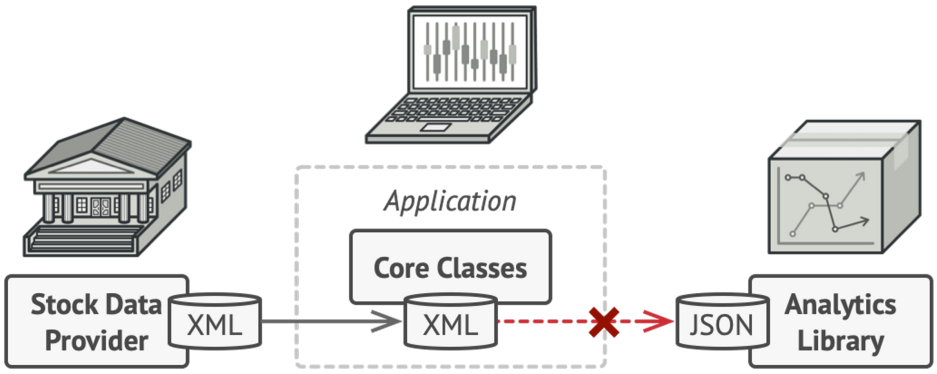


Problem

- Imagine that you're creating a stock market monitoring app. The app downloads the stock data from multiple sources in XML format and then displays nice-looking charts and diagrams for the user.
- At some point, you decide to improve the app by integrating a smart 3rd-party analytics library. But there's a catch: the analytics library only works with data in JSON format.

Problem cont.

- You can't use the analytics library "as is" because it expects the data in a format that's incompatible with your app.



Problem cont.

You could change the library to work with XML.

However, this might break some existing code that relies on the library.

And worse, you might not have access to the library's source code in the first place, making this approach impossible.

Solution

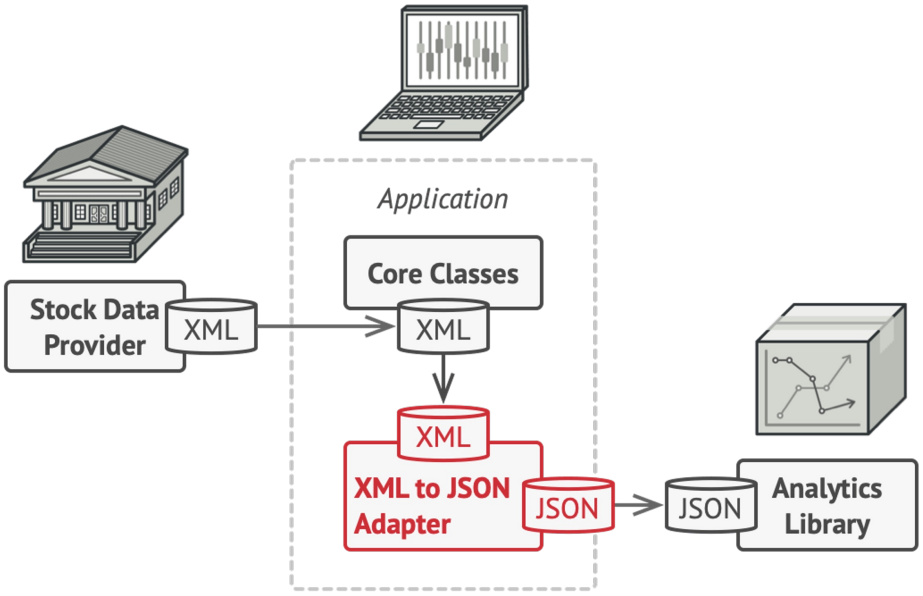
- Create an adapter: Which is a special object that converts the interface of one object so that another object can understand it.
- It wraps one of the objects to hide the complexity of conversion happening behind the scenes.
- The wrapped object isn't even aware of the adapter.
- For example, you can wrap an object that operates in meters and kilometers with an adapter that converts all of the data to imperial units such as feet and miles.
- Adapters can not only convert data into various formats but can also help objects with different interfaces collaborate.

How it Works

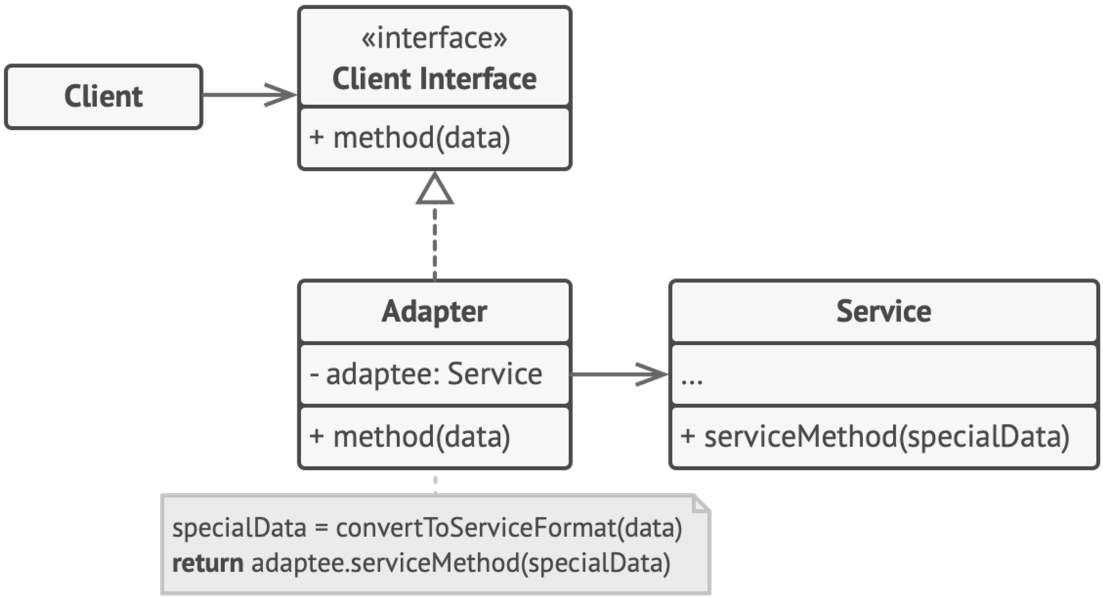
1. The adapter gets an interface, compatible with one of the existing objects.
2. Using this interface, the existing object can safely call the adapter's methods.
3. Upon receiving a call, the adapter passes the request to the second object, but in a format and order that the second object expects.

Sometimes it's even possible to create a two-way adapter that can convert the calls in both directions.

Stock Market App

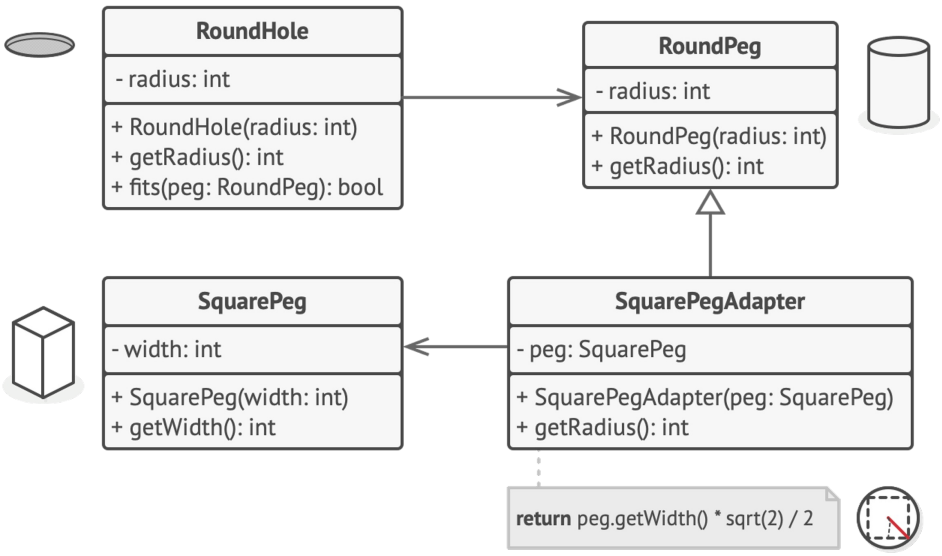


Object Adapter Structure



Pseudocode

This example of the **Adapter** pattern is based on the classic conflict between square pegs and round holes



```
public class RoundHole {
    private double radius;

    public RoundHole(double radius) {
        this.radius = radius;
    }

    public double getRadius() {
        return radius;
    }

    public boolean fits(RoundPeg peg) {
        boolean result;
        result = (this.getRadius() >= peg.getRadius());
        return result;
    }
}
```



```
public class RoundPeg {
    private double radius;

    public RoundPeg() {}

    public RoundPeg(double radius) {
        this.radius = radius;
    }

    public double getRadius() {
        return radius;
    }
}
```

```java
public class SquarePeg {
    private double width;

    public SquarePeg(double width) {
        this.width = width;
    }

    public double getWidth() {
        return width;
    }

    public double getSquare() {
        double result;
        result = Math.pow(this.width, 2);
        return result;
    }
}
```



```java
public class SquarePegAdapter extends RoundPeg {
    private SquarePeg peg;

    public SquarePegAdapter(SquarePeg peg) {
        this.peg = peg;
    }

    @Override
    public double getRadius() {
        double result;
        // Calculate a minimum circle radius, which can fit this peg.
        result = (Math.sqrt(Math.pow((peg.getWidth() / 2), 2) * 2));
        return result;
    }
}
```

```
public class Demo {
    public static void main(String[] args) {
        // Round fits round, no surprise.
        RoundHole hole = new RoundHole(5);
        RoundPeg rpeg = new RoundPeg(5);
        if (hole.fits(rpeg)) {
            System.out.println("Round peg r5 fits round hole r5.");
        }

        SquarePeg smallSqPeg = new SquarePeg(2);
        SquarePeg largeSqPeg = new SquarePeg(20);
        // hole.fits(smallSqPeg); // Won't compile.

        // Adapter solves the problem.
        SquarePegAdapter smallSqPegAdapter = new SquarePegAdapter(smallSqPeg);
        SquarePegAdapter largeSqPegAdapter = new SquarePegAdapter(largeSqPeg);
        if (hole.fits(smallSqPegAdapter)) {
            System.out.println("Square peg w2 fits round hole r5.");
        }
        if (!hole.fits(largeSqPegAdapter)) {
            System.out.println("Square peg w20 does not fit into round hole r5.");
        }
    }
}
```

```
Round peg r5 fits round hole r5.
Square peg w2 fits round hole r5.
Square peg w20 does not fit into
round hole r5.
```


Applicability

Use the Adapter class when you want to use some existing class, but its interface isn’t compatible with the rest of your code.

- The Adapter pattern lets you create a middle-layer class that serves as a translator between your code and a legacy class, a 3rd-party class or any other class with a weird interface.

Applicability cont.

Use the pattern when you want to reuse several existing subclasses that lack some common functionality that can't be added to the superclass.

- You could extend each subclass and put the missing functionality into new child classes. However, you'll need to duplicate the code across all of these new classes, which smells really bad.

Applicability cont.

- The much more elegant solution would be to put the missing functionality into an adapter class. Then you would wrap objects with missing features inside the adapter, gaining needed features dynamically. For this to work, the target classes must have a common interface, and the adapter's field should follow that interface. This approach looks very similar to the Decorator pattern.