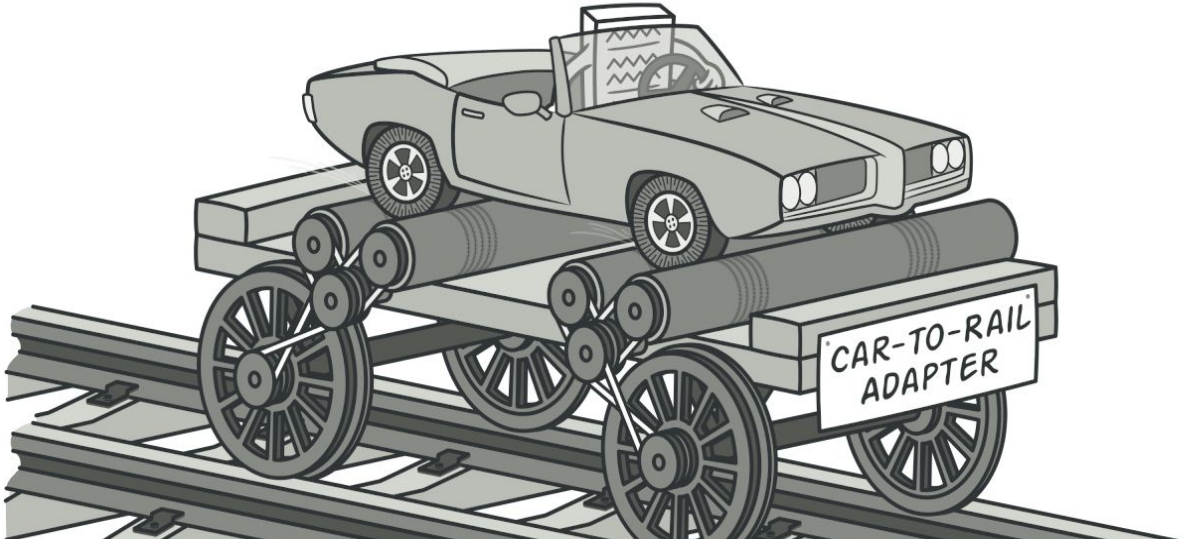
Design Pattern: Adapter

Gabriella Zola & Samara Ribeiro Antonio

01
Adaptor Introduction

What is the Adaptor Design Pattern

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



Applicability

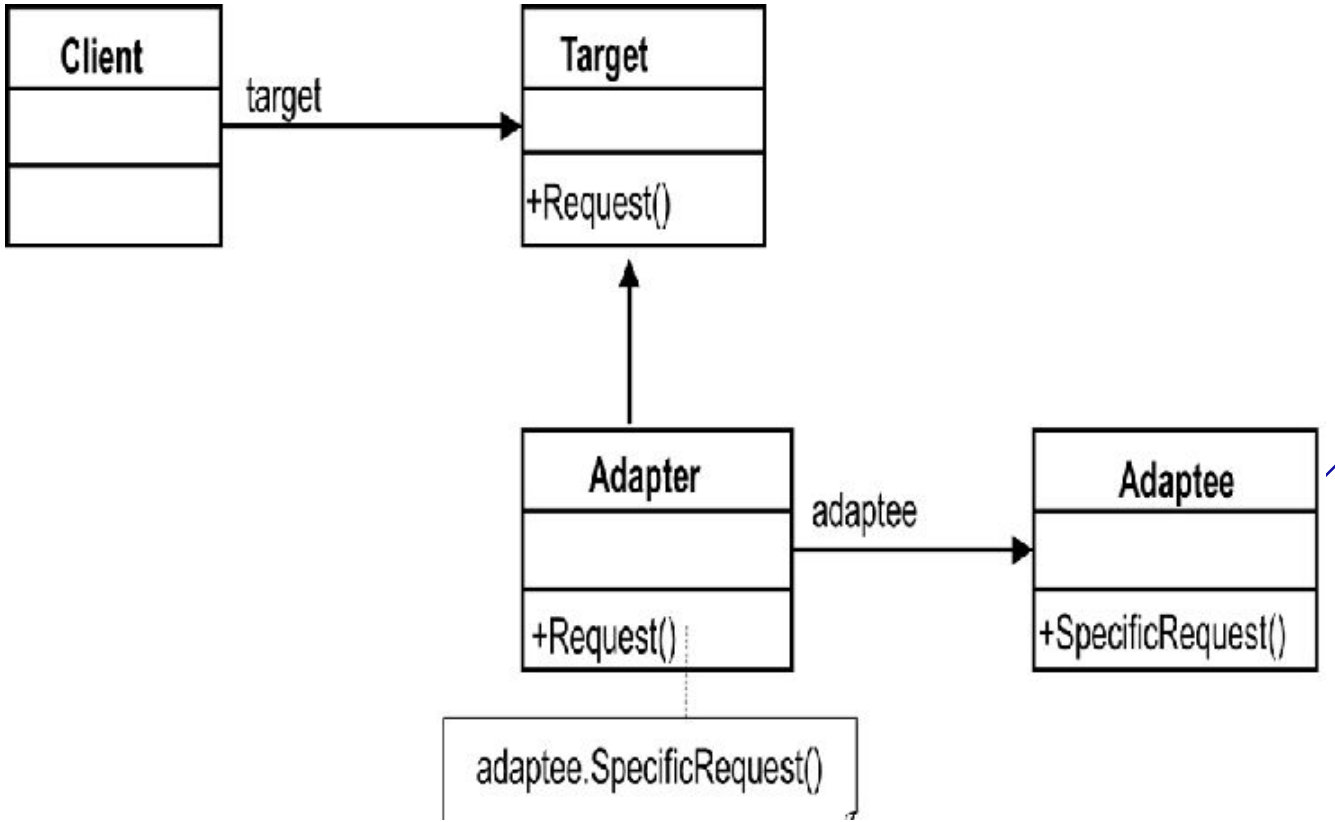
- Use the Adapter class when you want to use some existing class, but its interface isn’t compatible with the rest of your code.

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

Key Features

1. **Target Interface:** The target interface represents the expected interface that the client wants to interact with.
2. **Adaptee:** The adaptee is the existing class or object that has an incompatible interface with the client.
3. **Adapter:** The adapter class wraps the adaptee and implements the target interface.
4. **Client:** The code that uses the target interface to interact with objects.

How Adapter Design Pattern works?



Pros

✔ Single Responsibility Principle

✔Open/Closed Principle

✔Compatibility and Integration

✔Code Reuse

Cons

✘ The overall complexity of the code increases.

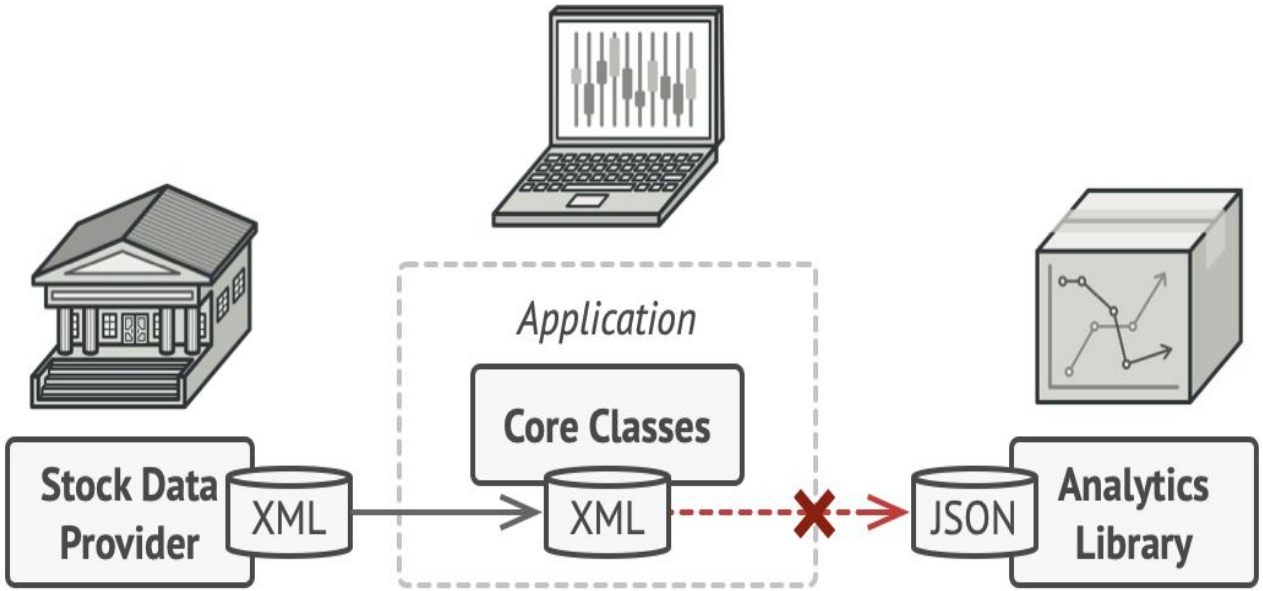
✘ It creates coupling and dependency between the two interfaces.

✘ It can make testing and debugging more difficult.

02
Adaptor Examples and Structure

What it is? Which problems does it solve?

Stock Market App Problem

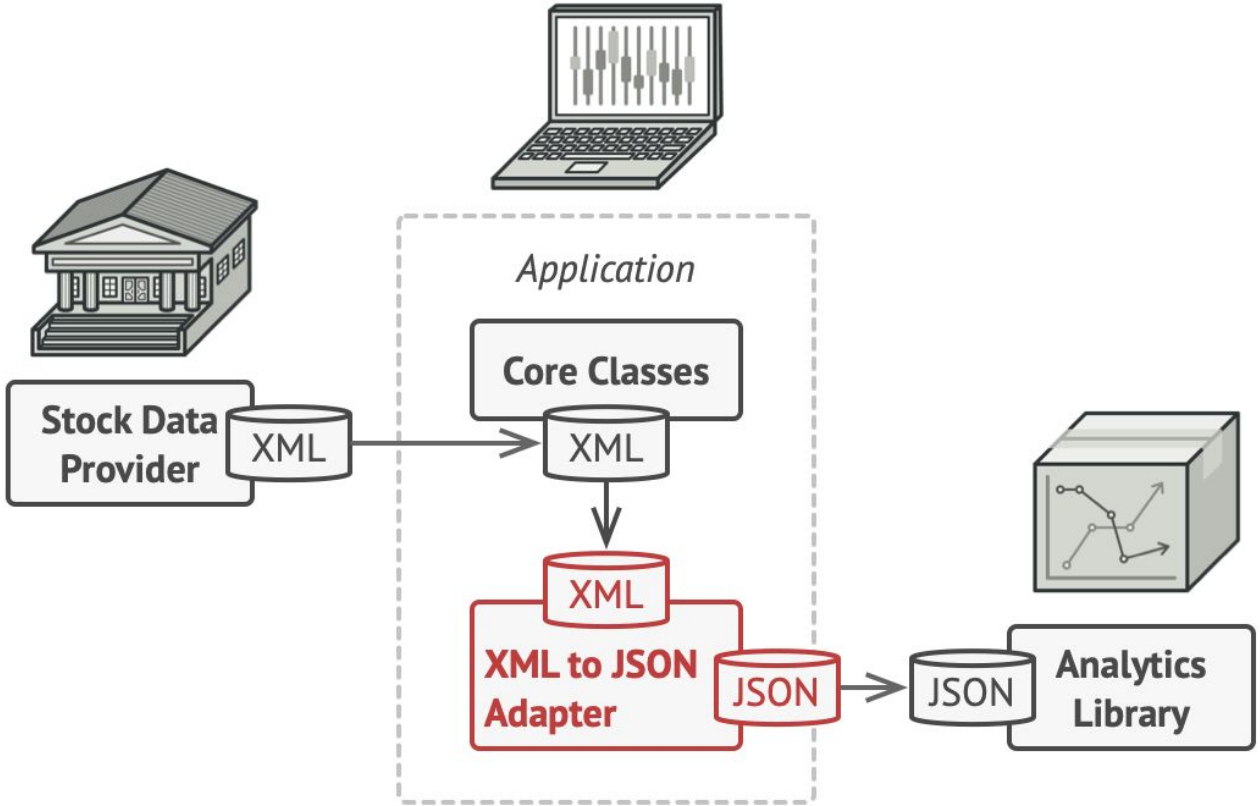


You can't use the analytics library "as is" because it expects the data in a format that's incompatible with your app.

Problem Solution: You can create an adapter!

To solve the dilemma of incompatible formats for the Stock Market App Problem, you can create XML-to-JSON adapters for every class of the analytics library that your code works with directly.

Then you adjust your code to communicate with the library only via these adapters. When an adapter receives a call, it translates the incoming XML data into a JSON structure and passes the call to the appropriate methods of a wrapped analytics object.

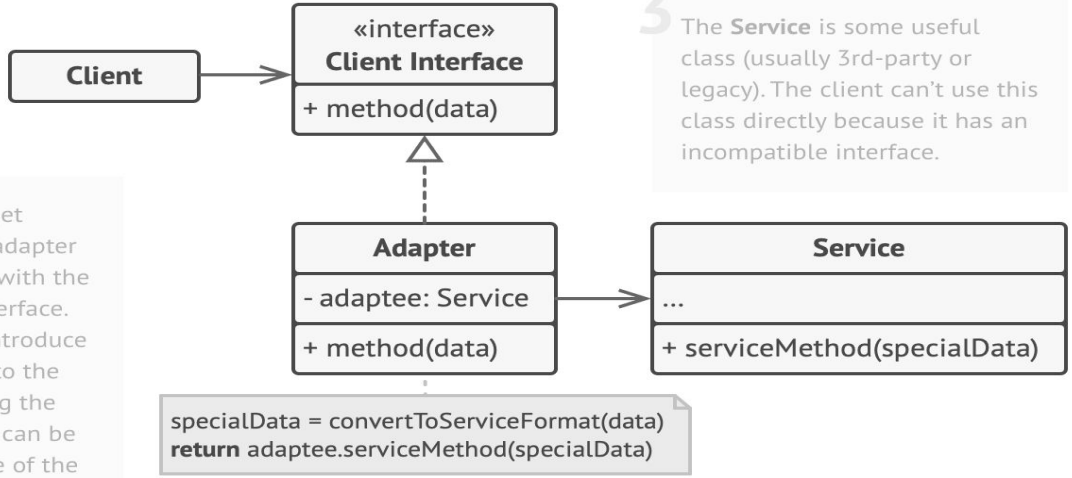

Application
Core Classes
Stock Data Provider
XML
XML
XML
XML to JSON Adapter
JSON
JSON
Analytics Library

Structure Visualization

Object Adapter

2 The **Client Interface** describes a protocol that other classes must follow to be able to collaborate with the client code.

1 The **Client** is a class that contains the existing business logic of the program.

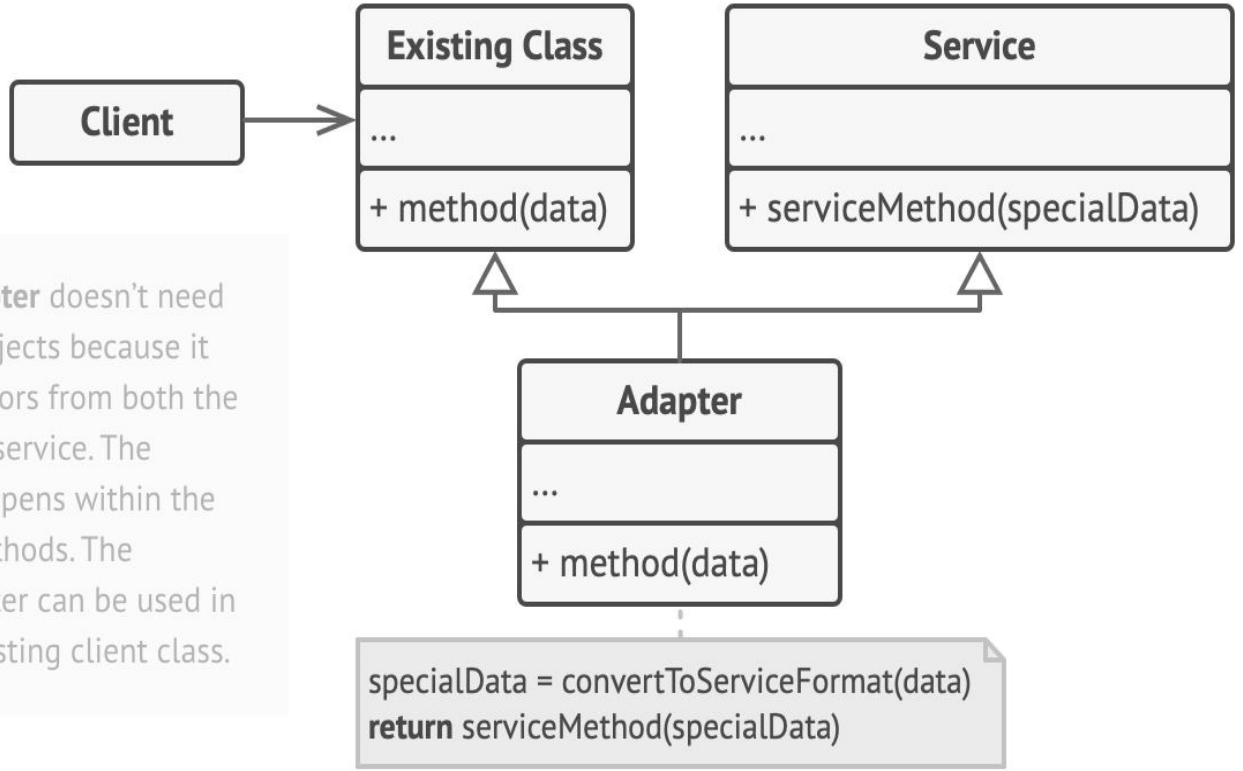


3 The **Service** is some useful class (usually 3rd-party or legacy). The client can't use this class directly because it has an incompatible interface.

5 The client code doesn't get coupled to the concrete adapter class as long as it works with the adapter via the client interface. Thanks to this, you can introduce new types of adapters into the program without breaking the existing client code. This can be useful when the interface of the service class gets changed or replaced: you can just create a new adapter class without changing the client code.

4 The **Adapter** is a class that's able to work with both the client and the service: it implements the client interface, while wrapping the service object. The adapter receives calls from the client via the client interface and translates them into calls to the wrapped service object in a format it can understand.

Class Adapter

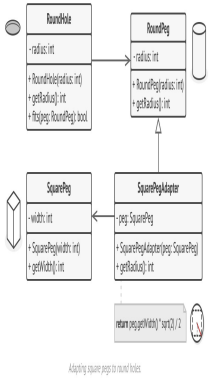


1 The **Class Adapter** doesn't need to wrap any objects because it inherits behaviors from both the client and the service. The adaptation happens within the overridden methods. The resulting adapter can be used in place of an existing client class.

Adaptor Example in Java:

Fitting square pegs into round holes

```java
package refactoring_guru.adapter.example;

import refactoring_guru.adapter.example.adapters.SquarePegAdapter;
import refactoring_guru.adapter.example.round.RoundHole;
import refactoring_guru.adapter.example.round.RoundPeg;
import refactoring_guru.adapter.example.square.SquarePeg;

/**
 * Somewhere in client code...
 */
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

round/RoundHole.java: Round holes

```
package refactoring_guru.adapter.example.round;

/**
 * RoundHoles are compatible with RoundPegs.
 */
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;
    }
}
```

round/RoundPeg.java: Round pegs

```
package refactoring_guru.adapter.example.round;

/**
 * RoundPegs are compatible with RoundHoles.
 */
public class RoundPeg {
    private double radius;

    public RoundPeg() {}

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

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

square

square/SquarePeg.java: Square pegs

```
package refactoring_guru.adapter.example.square;

/**
 * SquarePegs are not compatible with RoundHoles (they were implemented by
 * previous development team). But we have to integrate them into our program.
 */
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;
    }
}
```

adapters

adapters/SquarePegAdapter.java: Adapter of square pegs to round holes

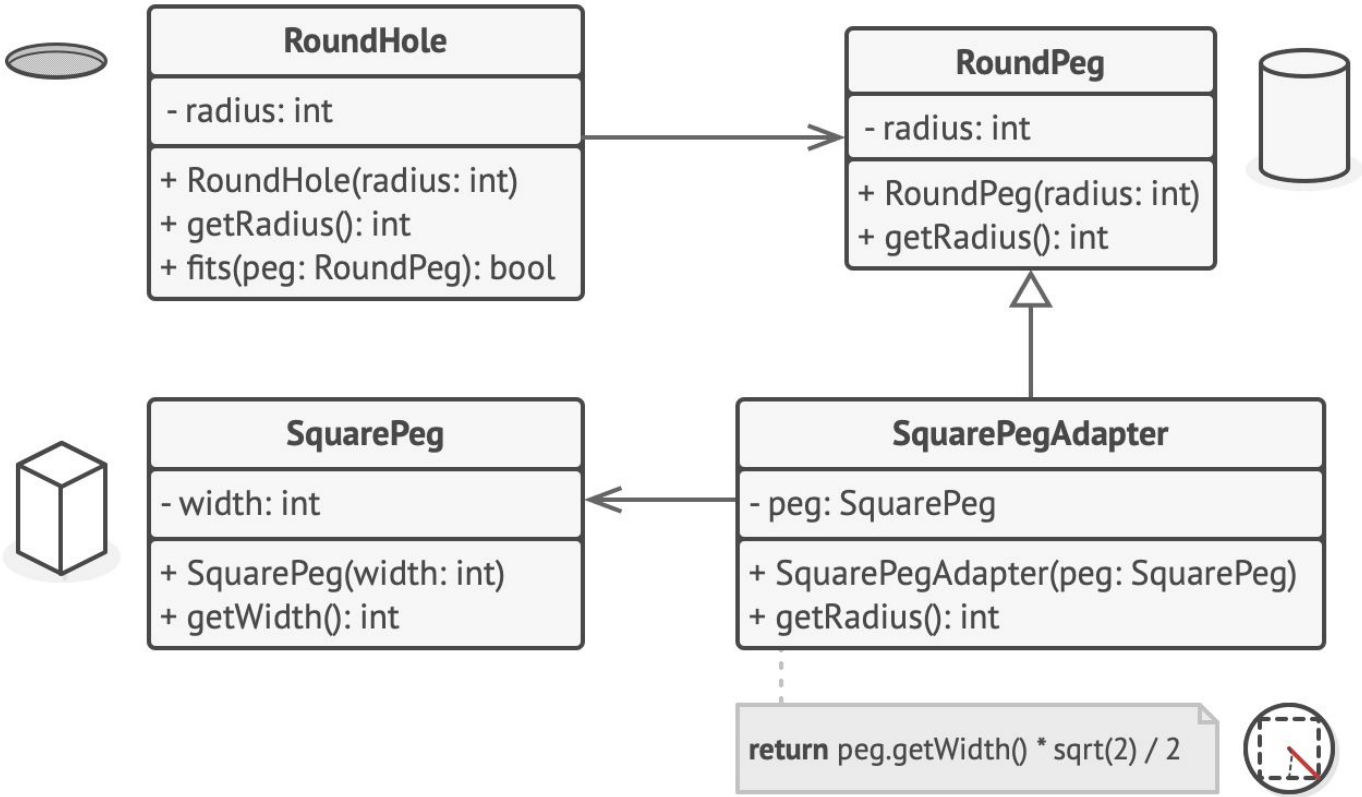
```
package refactoring_guru.adapter.example.adapters;

import refactoring_guru.adapter.example.round.RoundPeg;
import refactoring_guru.adapter.example.square.SquarePeg;

/**
 * Adapter allows fitting square pegs into round holes.
 */
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;
    }
}
```

Demo.java: Client code

```
package refactoring_guru.adapter.example;

import refactoring_guru.adapter.example.adapters.SquarePegAdapter;
import refactoring_guru.adapter.example.round.RoundHole;
import refactoring_guru.adapter.example.round.RoundPeg;
import refactoring_guru.adapter.example.square.SquarePeg;

/**
 * Somewhere in client code...
 */
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.");
        }
    }
}
```

OutputDemo.txt: Execution result

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



Adapting square pegs to round holes.

There are some standard Adapters in Java core libraries:

- java.util.Arrays#asList()
- java.util.Collections#list()
- java.util.Collections#enumeration()
- java.io.InputStreamReader(InputStream) (returns a Reader object)
- java.io.OutputStreamWriter(OutputStream) (returns a Writer object)
- javax.xml.bind.annotation.adapters.XmlAdapter#marshal() and #unmarshal()

Sources

- https://refactoring.guru/design-patterns/adapter/java/example
- https://refactoring.guru/design-patterns/adapter
- https://en.wikipedia.org/wiki/Adapter_pattern
- https://www.geeksforgeeks.org/adapter-pattern/
- https://www.linkedin.com/advice/0/what-drawbacks-using-adapter-pattern-performance#:~:text=The%20added%20adapters%20for%20communication,can%20introduce%20latency%20and%20complexity.&text=While%20the%20Adapter%20Pattern%20is,terms%20of%20performance%20and%20scalability.
- https://www.pentalog.com/blog/design-patterns/adapter-design-pattern/#:~:text=Advantages%20of%20the%20Adapter%20Design%20Pattern&text=Compatibility%20and%20Integration%3A%20One%20of,the%20rest%20of%20the%20codebase.
- https://medium.com/@rajeshvelmani/bridging-the-gap-exploring-the-adapter-design-pattern-in-java-823218841a9c#:~:text=The%20Adapter%20design%20pattern%20falls,as%20a%20bridge%20between%20them.

Thank You!

Do you have any questions?