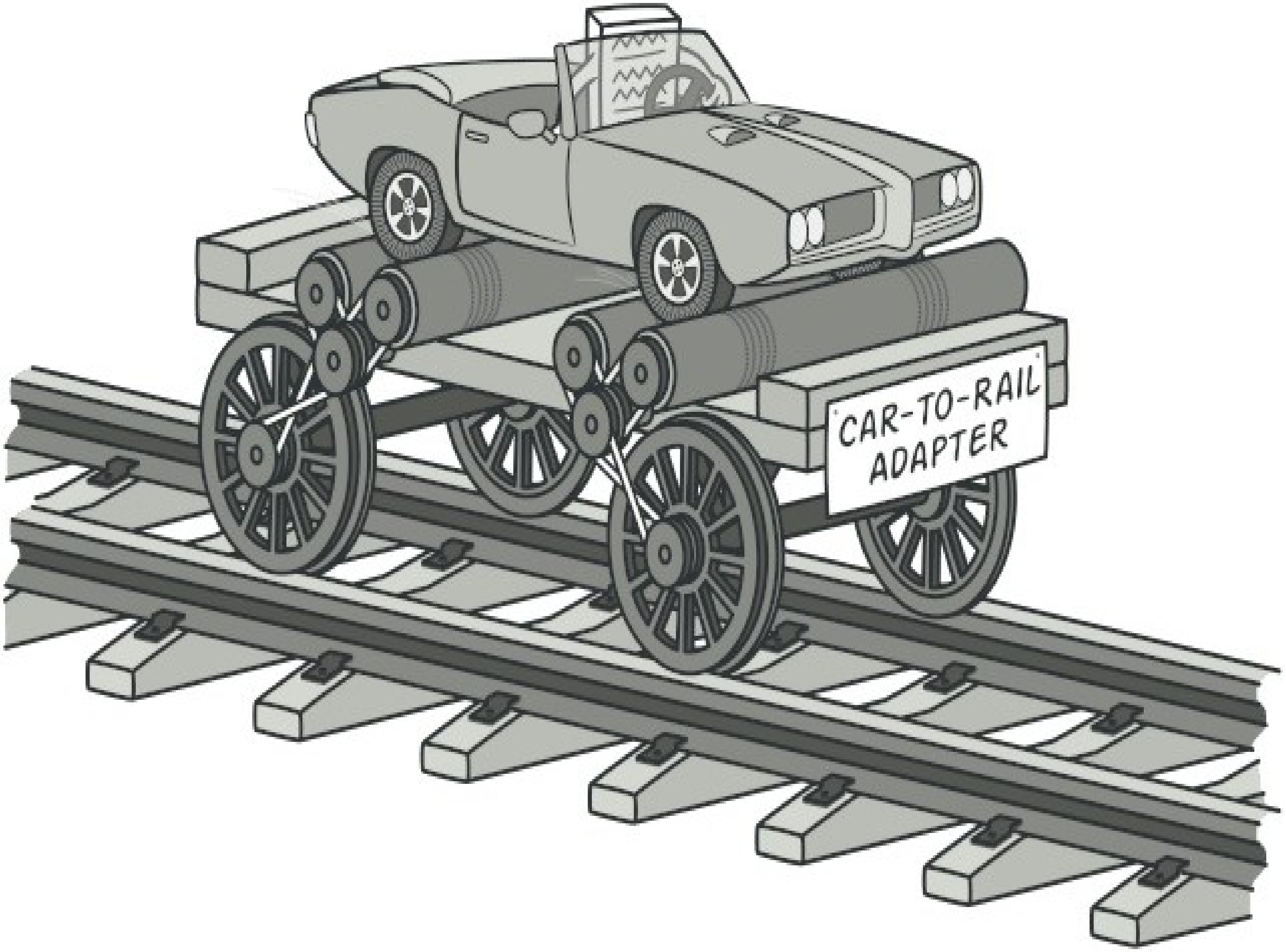
Adaptor

Lecture 19

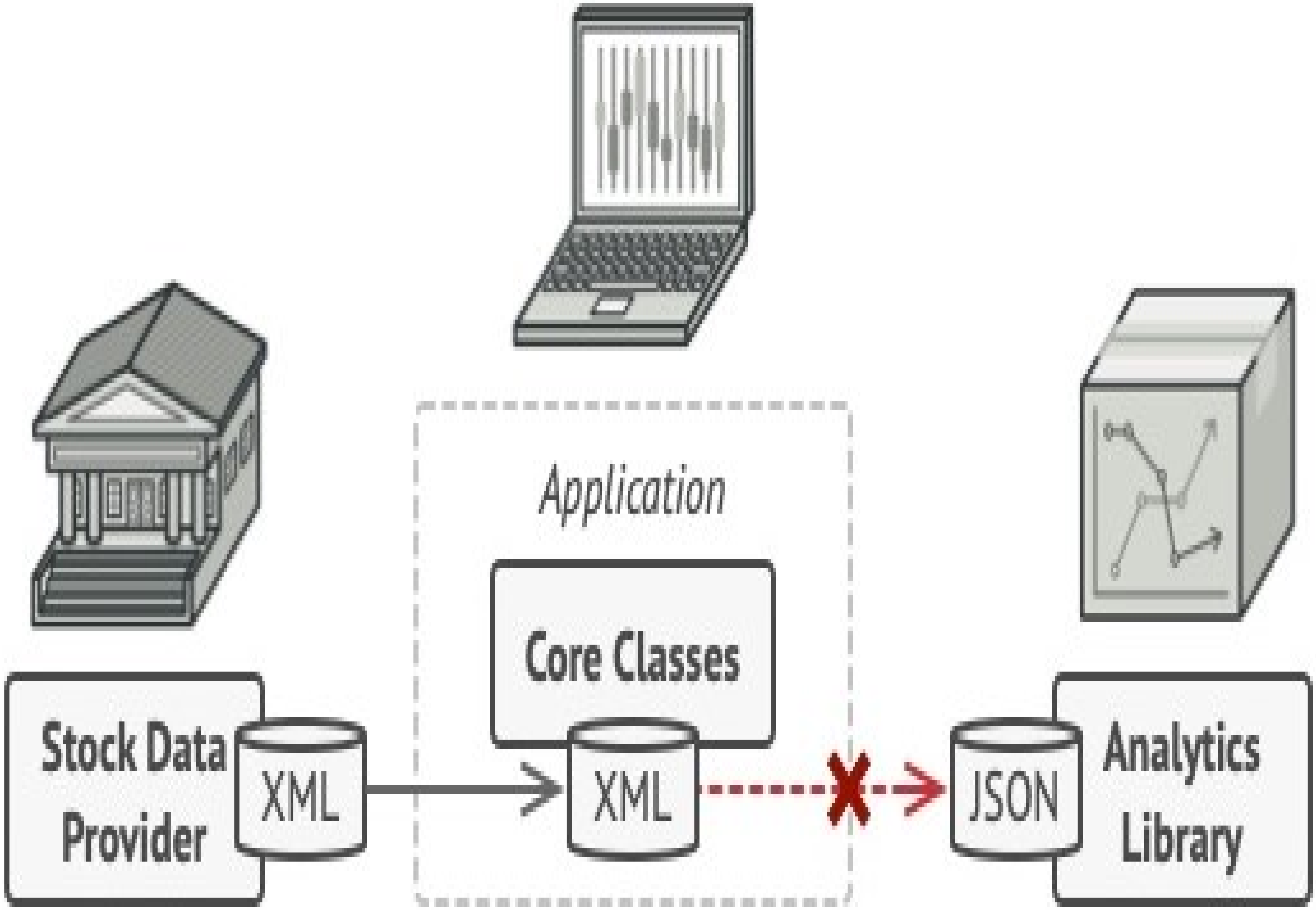
Intent

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


CAR-TO-RAIL ADAPTER

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.


Application
Core Classes
Stock Data Provider
XML
XML
JSON
Analytics Library

Problem

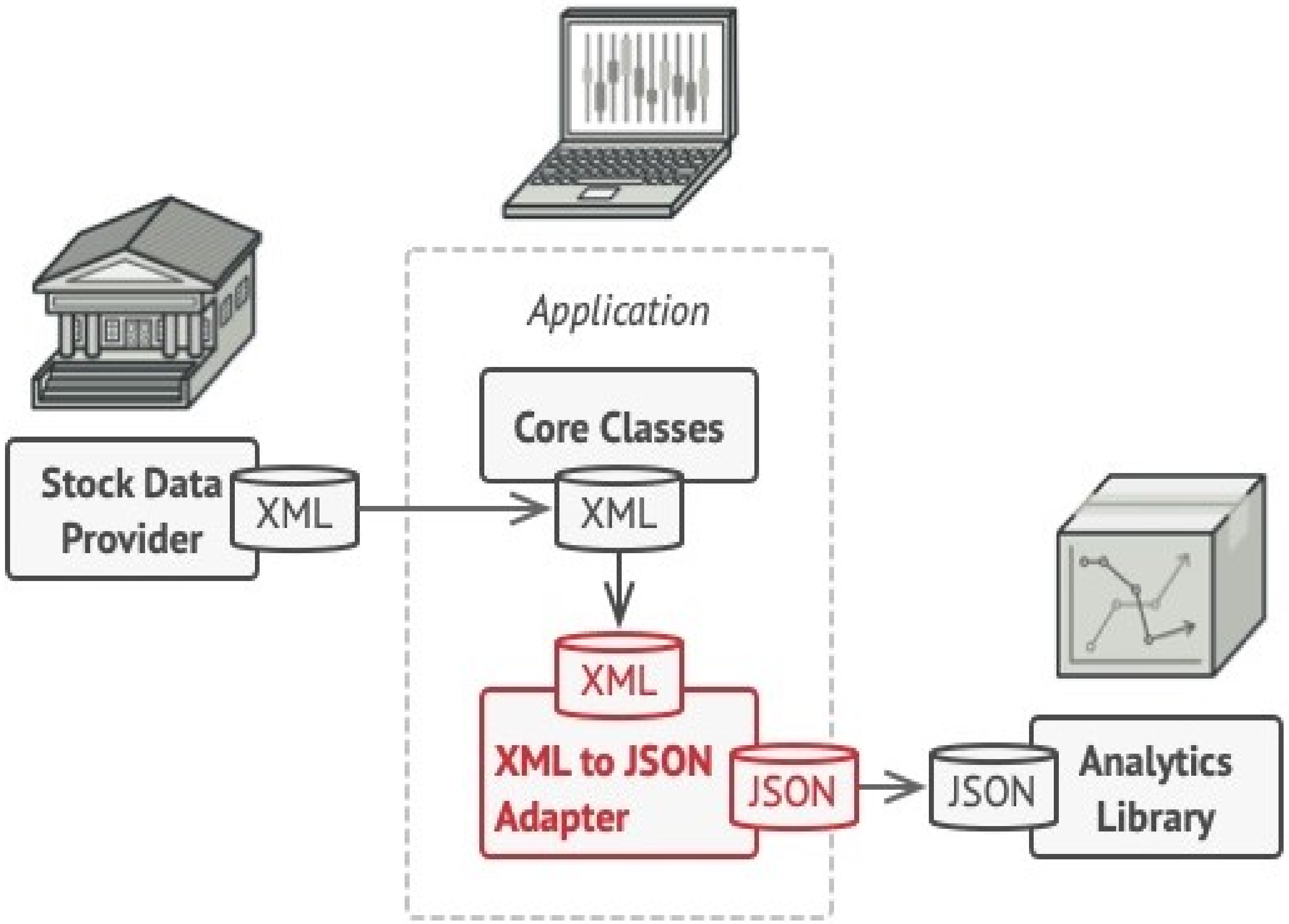
- You could change the library to work with XML.
- However, this might break some existing code that relies on the library.
- And worse, you might not have access to the library’s source code in the first place, making this approach impossible.

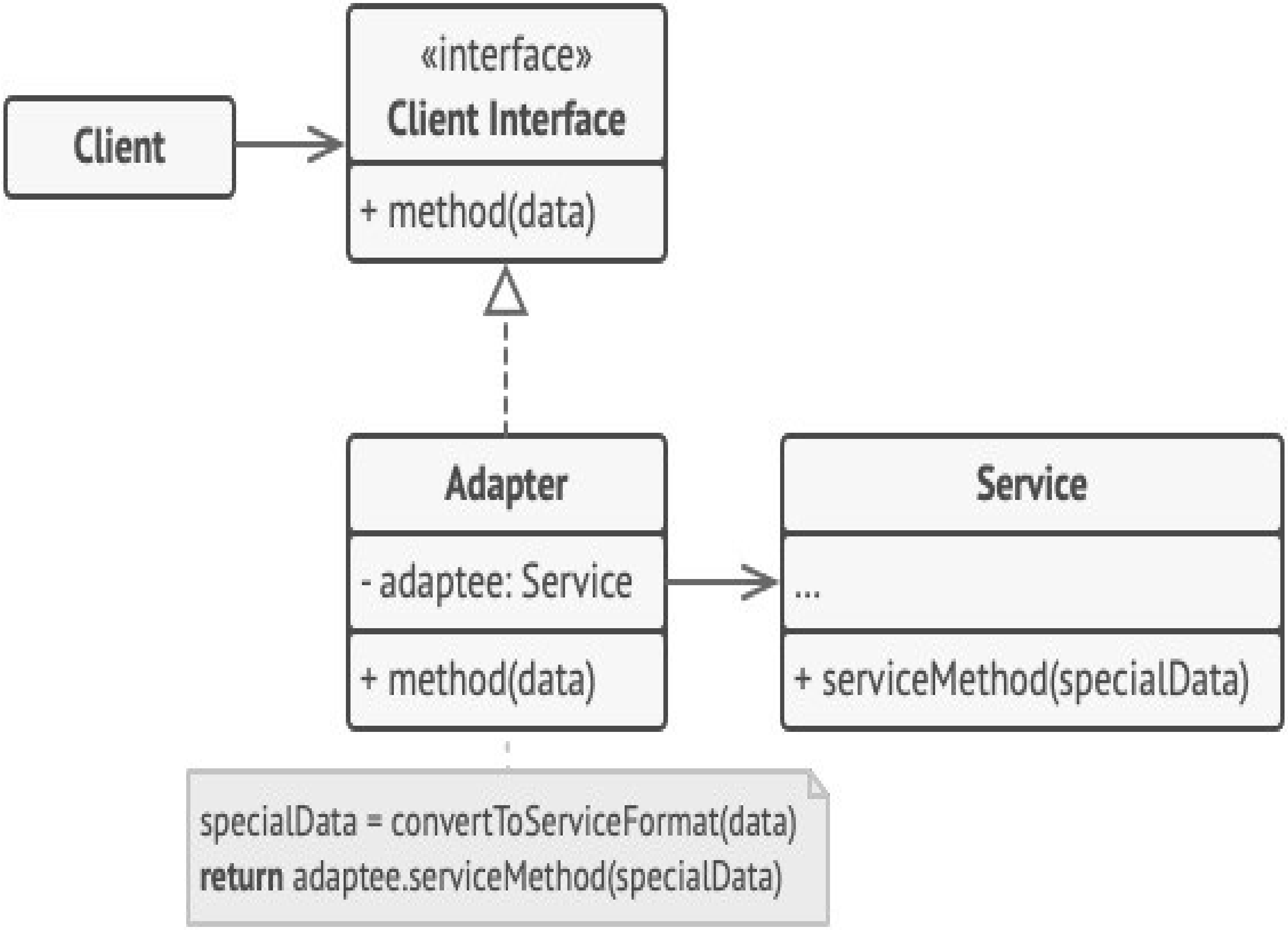
Solution

- You can create an *adapter*. This is a special object that converts the interface of one object so that another object can understand it.

Solution

- Adapters can not only convert data into various formats but can also help objects with different interfaces collaborate. Here’s how it works:
  - The adapter gets an interface, compatible with one of the existing objects.
  - Using this interface, the existing object can safely call the adapter’s methods.
  - Upon receiving a call, the adapter passes the request to the second object, but in a format and order that the second object expects.


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


«interface»
Client Interface
+ method(data)
Client
Adapter
- adaptee: Service
+ method(data)
Service
...
+ serviceMethod(specialData)
specialData = convertToServiceFormat(data)
return adaptee.serviceMethod(specialData)

Structure

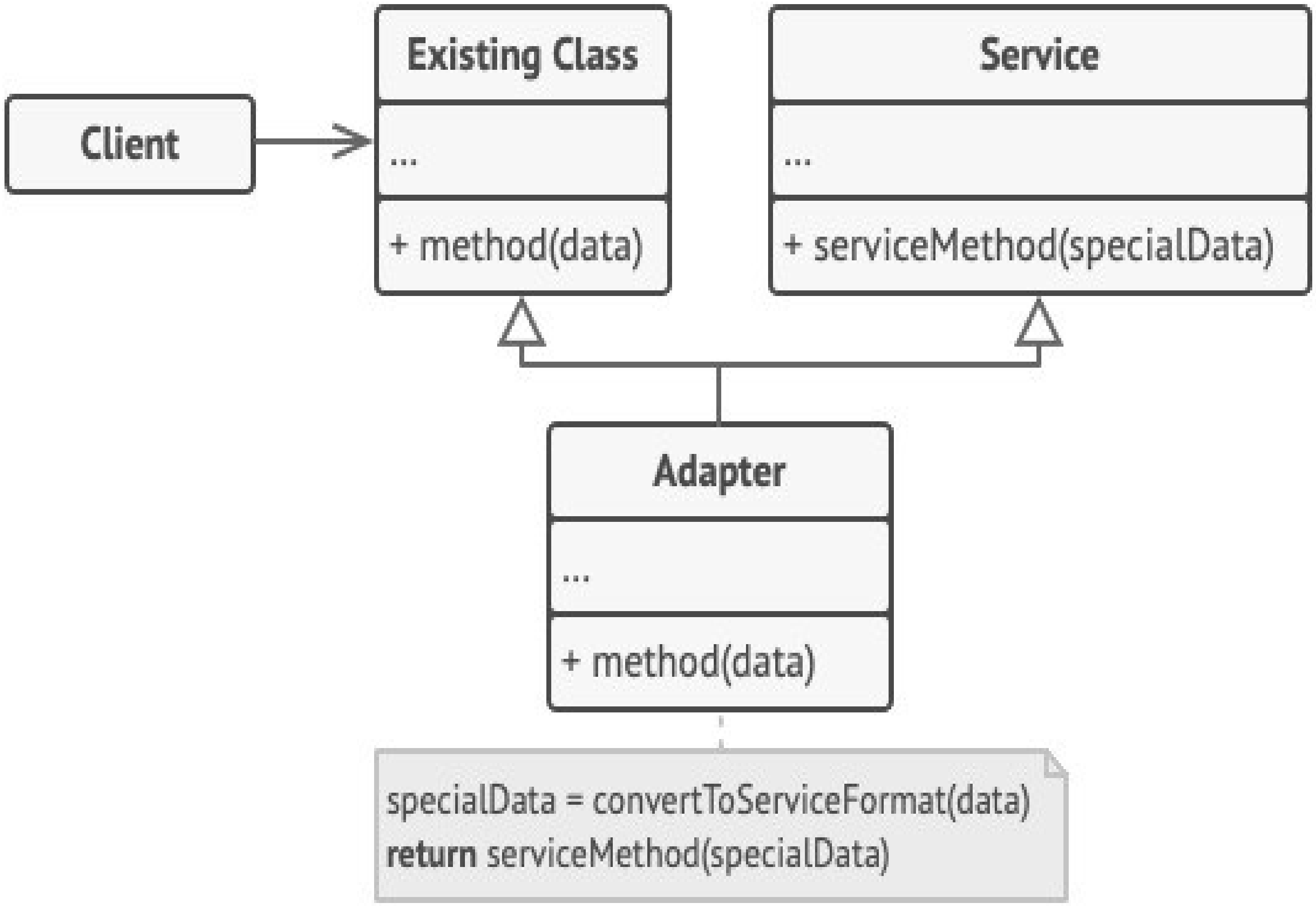
- The **Client** is a class that contains the existing business logic of the program.
- The **Client Interface** describes a protocol that other classes must follow to be able to collaborate with the client code.
- The **Service** is some useful class (usually 3rd-party or legacy). The client can’t use this class directly because it has an incompatible interface.

Explanation

- 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 adapter interface and translates them into calls to the wrapped service object in a format it can understand.

Explanation

- 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.

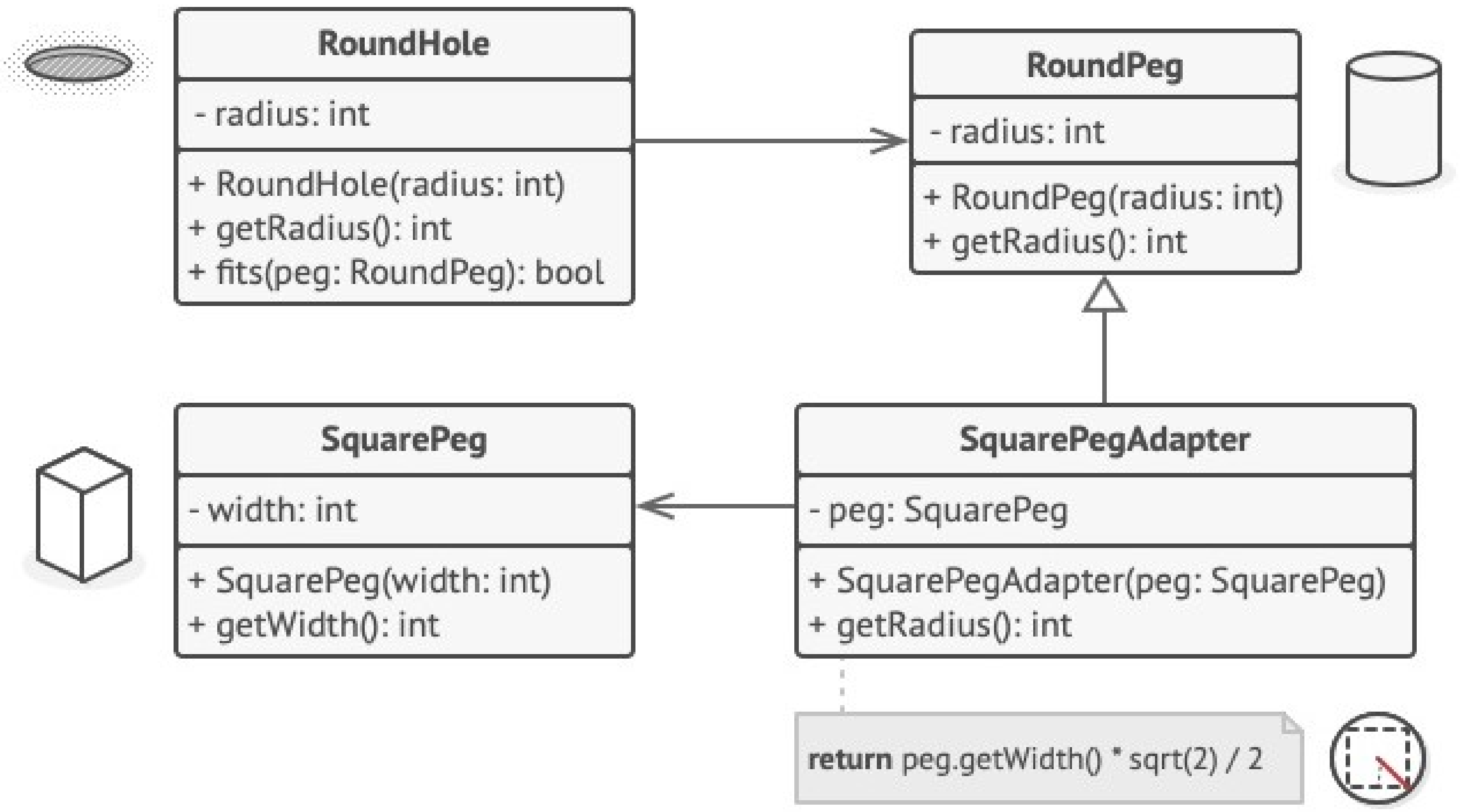


Client
Existing Class
...
+ method(data)
Service
...
+ serviceMethod(specialData)
Adapter
...
+ method(data)
specialData = convertToServiceFormat(data)
return serviceMethod(specialData)

Example

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

- The Adapter pretends to be a round peg, with a radius equal to a half of the square’s diameter (in other words, the radius of the smallest circle that can accommodate the square peg).


RoundHole
- radius: int
+ RoundHole(radius: int)
+ getRadius(): int
+ fits(peg: RoundPeg): bool
RoundPeg
- radius: int
+ RoundPeg(radius: int)
+ getRadius(): int
SquarePeg
- width: int
+ SquarePeg(width: int)
+ getWidth(): int
SquarePegAdapter
- peg: SquarePeg
+ SquarePegAdapter(peg: SquarePeg)
+ getRadius(): int
return peg.getWidth() * sqrt(2) / 2

```
// Say you have two classes with compatible interfaces:
// RoundHole and RoundPeg.
class RoundHole is
    constructor RoundHole(radius) { ... }

    method getRadius() is
        // Return the radius of the hole.

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

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

    method getRadius() is
        // Return the radius of the peg.
// But there's an incompatible class: SquarePeg.
class SquarePeg is
    constructor SquarePeg(width) { ... }

    method getWidth() is
        // Return the square peg width.


// An adapter class lets you fit square pegs into round holes.
// It extends the RoundPeg class to let the adapter objects act
// as round pegs.
class SquarePegAdapter extends RoundPeg is
    // In reality, the adapter contains an instance of the
    // SquarePeg class.
    private field peg: SquarePeg

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

    method getRadius() is
        // The adapter pretends that it's a round peg with a
        // radius that could fit the square peg that the adapter
        // actually wraps.
        return peg.getWidth() * Math.sqrt(2) / 2
```

```
// Somewhere in client code.
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) // this won't compile (incompatible types)

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
```

How to Implement

1. Make sure that you have at least two classes with incompatible interfaces:

    - A useful *service* class, which you can't change (often 3rd-party, legacy or with lots of existing dependencies).
    - One or several *client* classes that would benefit from using the service class.

2. Declare the client interface and describe how clients communicate with the service.

3. Create the adapter class and make it follow the client interface. Leave all the methods empty for now.

4. Add a field to the adapter class to store a reference to the service object. The common practice is to initialize this field via the constructor, but sometimes it's more convenient to pass it to the adapter when calling its methods.

5. One by one, implement all methods of the client interface in the adapter class. The adapter should delegate most of the real work to the service object, handling only the interface or data format conversion.

6. Clients should use the adapter via the client interface. This will let you change or extend the adapters without affecting the client code.

Pros and Cons

- ✓ *Single Responsibility Principle*. You can separate the interface or data conversion code from the primary business logic of the program.
- ✓ *Open/Closed Principle*. You can introduce new types of adapters into the program without breaking the existing client code, as long as they work with the adapters through the client interface.

- ✗ The overall complexity of the code increases because you need to introduce a set of new interfaces and classes. Sometimes it's simpler just to change the service class so that it matches the rest of your code.

Reference

- https://refactoring.guru/design-patterns/adapter