# Software Design Patterns

*Lecture 6*

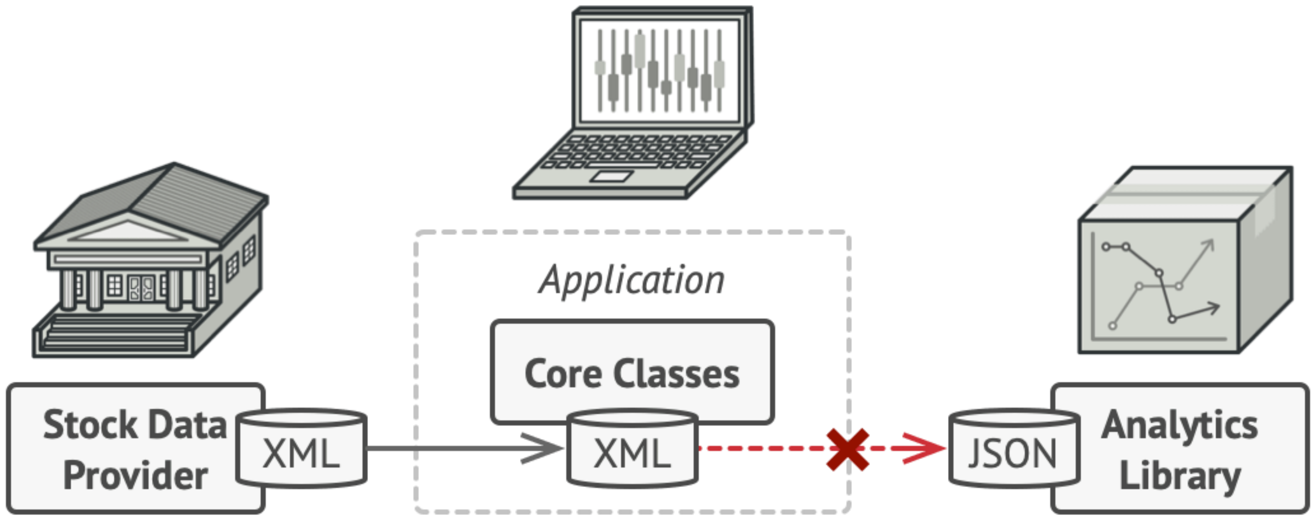
***Adaptor***
***Bridge***

**Dr. Fan Hongfei**

**10 October 2024**

## Adaptor: Problem

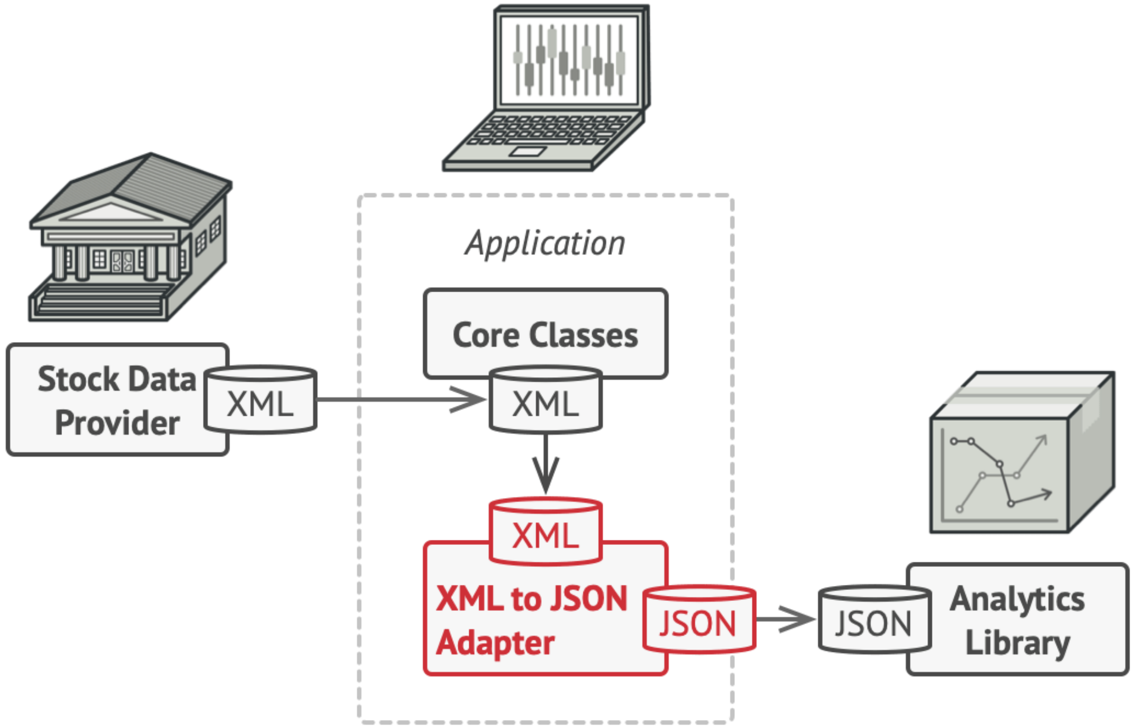
- **Example: a stock market monitoring app**
  - The app downloads stock data from multiple sources in XML format and displays nice-looking charts and diagrams
- New requirement: integrating a smart 3rd-party analytics library that only works with data in JSON format



[Figures in this slide are extracted from https://refactoring.guru/design-patterns/adapter]

## Adaptor: Solution

- **Adaptor:** a special objects that converts the interface
  - Wrapping one of the objects to hide the complexity of conversion
  - The wrapped object is not even aware of the adapter
- **Working mechanism**
  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



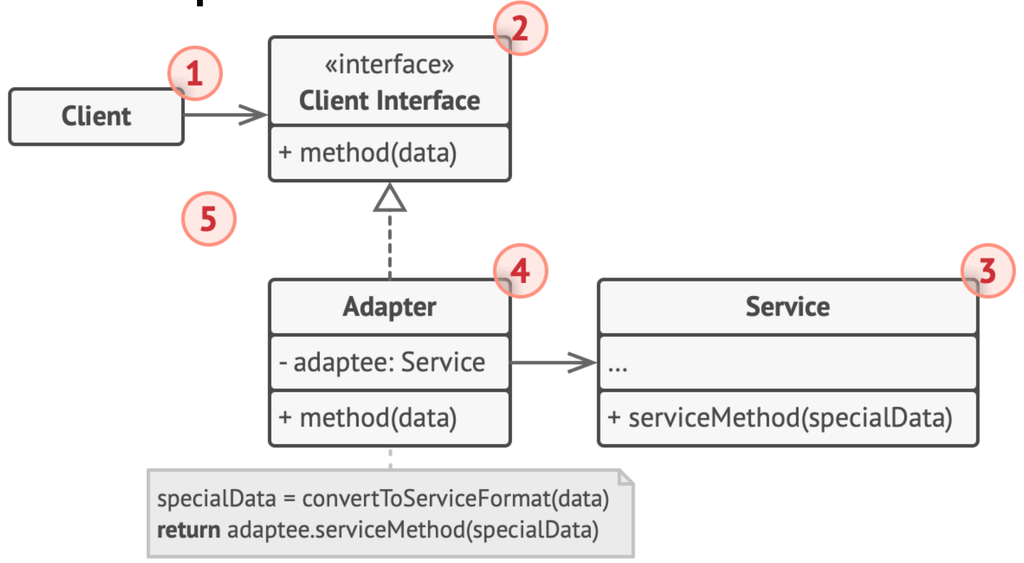
- When an XML-to-JSON adapter receives a call, it translates the incoming XML data into a JSON structure and passes the call to the methods of a wrapped analytics object

[Figures in this slide are extracted from https://refactoring.guru/design-patterns/adapter]

# Adaptor: Structure

## 1) Object adaptor

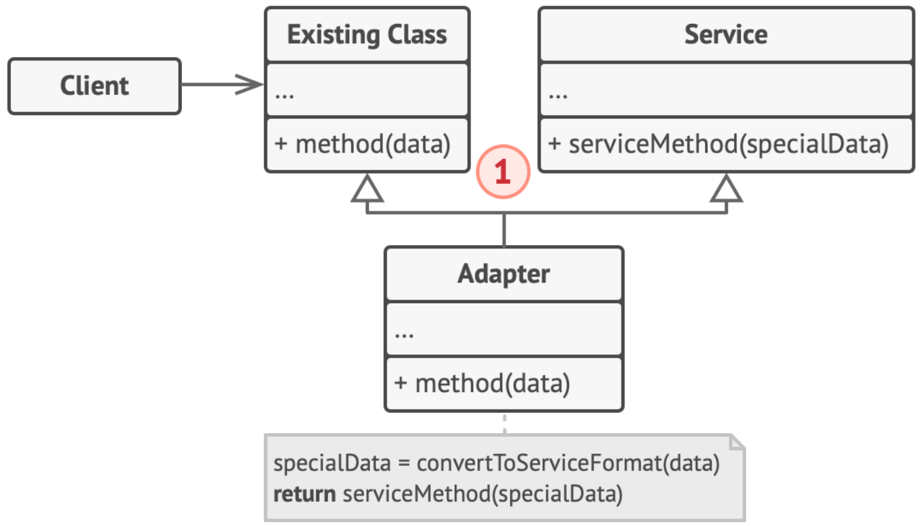
- **Composition-based**: the adapter implements the interface of one object and wraps the other one


1
Client
2
«interface»
Client Interface
+ method(data)
5
4
Adapter
- adaptee: Service
+ method(data)
3
Service
...
+ serviceMethod(specialData)
specialData = convertToServiceFormat(data)
return adaptee.serviceMethod(specialData)

- The client code does not get coupled to the concrete adapter class, and you can introduce new types of adapters, especially when the interface of the service class gets changed

## 2) Class adaptor

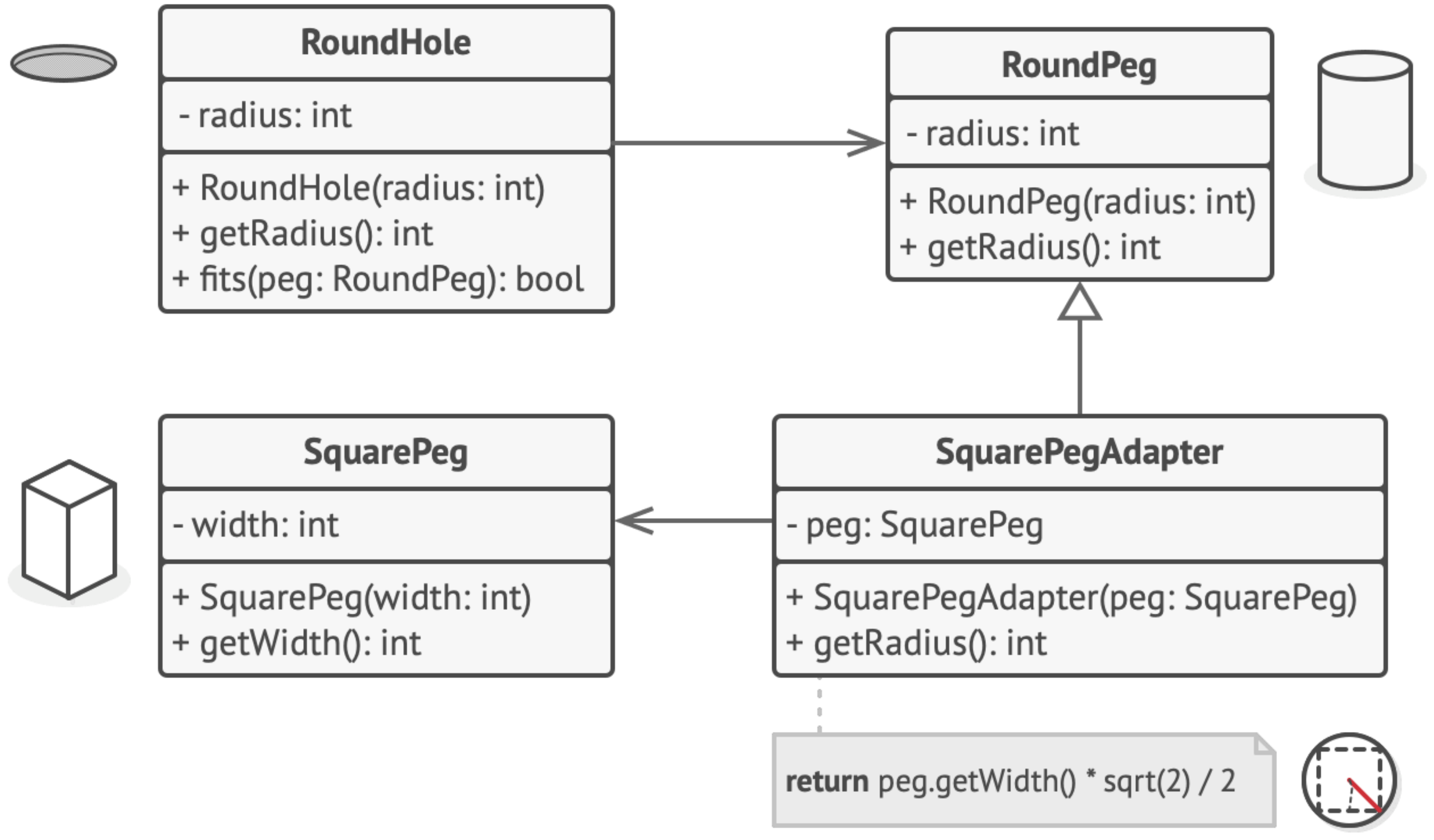
- **Inheritance-based**: the adapter inherits interfaces from both objects


Client
Existing Class
...
+ method(data)
Service
...
+ serviceMethod(specialData)
1
Adapter
...
+ method(data)
specialData = convertToServiceFormat(data)
return serviceMethod(specialData)

- Limitation: can only be implemented in languages supporting multiple inheritance
- The adaptation happens within overridden methods
- The resulting adapter can be used in place of an existing client class

[Figures in this slide are extracted from https://refactoring.guru/design-patterns/adapter]

## Adaptor: Example



- The Adapter **pretends** to be a round peg

## Adaptor: Applicability

- When you want to **use a existing class**, but its interface is not compatible with the rest of your code
  - The Adapter pattern lets you create **a middle-layer class serving as a translator** between your code and a legacy class, a 3rd-party class or any other class with a weird interface
- When you want to **reuse several existing subclasses that lack some common functionality** that cannot be added to the superclass
  - You could extend each subclass and put the missing functionality into new child classes, but it is not a good design
  - A better solution: put the missing functionality into an adapter class, and then wrap objects with missing features (note: the target classes must have a common interface for the adapter’s field to follow)

## Adaptor: Implementation

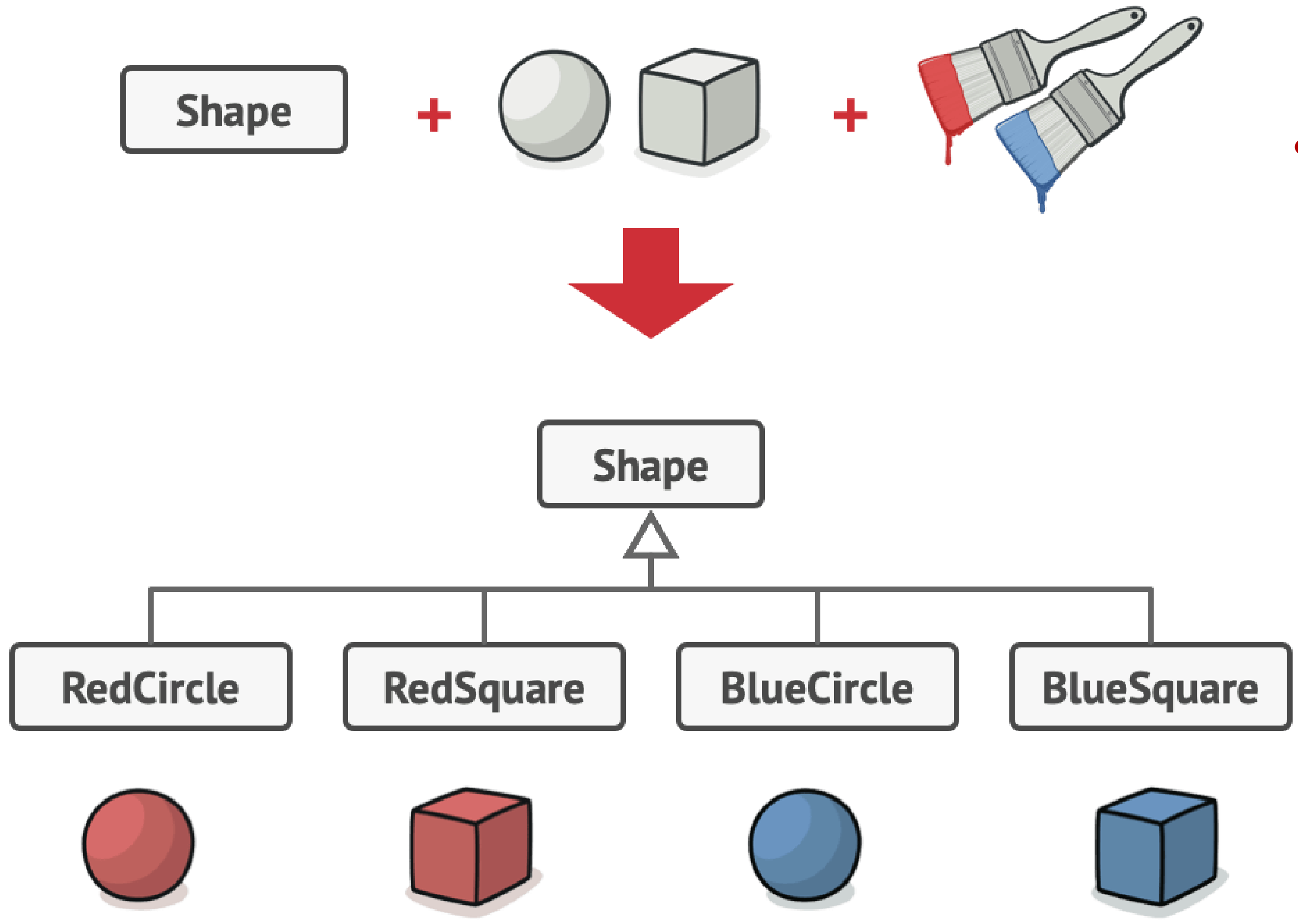
1. Make sure that you have at least two classes with incompatible interfaces:
   - A useful **service** class, which you **cannot change** (often 3rd-party, legacy or with lots of existing dependencies)
   - One or several **client** classes that would benefit from using the service class
2. Declare the client interface and describe how clients communicate with the service
3. Create the adapter class and make it follow the client interface
4. Add a field to the adapter class to store a reference to the service object
5. Implement all methods of the client interface in the adapter class
   - **Delegating** most of the real work to the service object, and **handling** only the interface or data format conversion
6. Clients should use the adapter via the client interface

## Adaptor: Pros and Cons

- **Pros**
  - Single Responsibility Principle: separate the interface or data conversion code from the primary business logic
  - Open/Closed Principle: introduce new types of adapters into the program without breaking the existing client code
- **Cons**
  - The overall complexity of the code increases

# Bridge: Problem

- **Example: a hierarchy of shape classes**

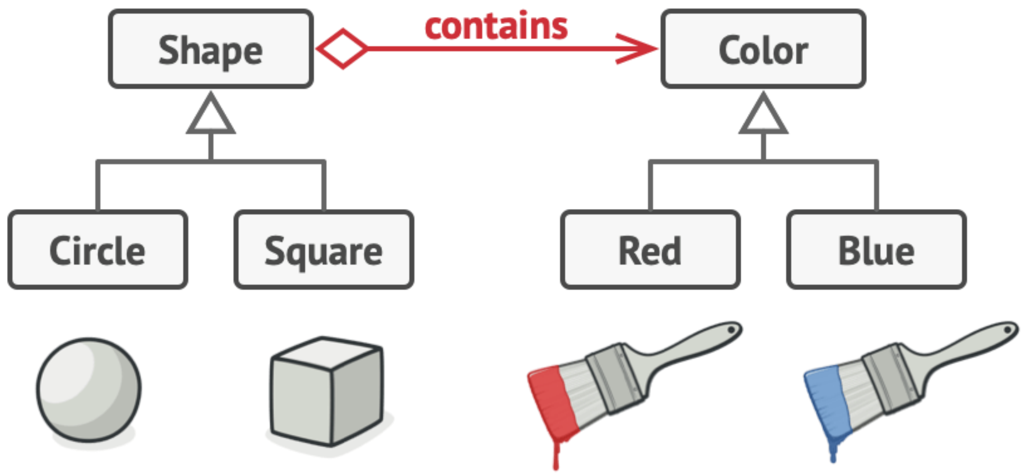


- **Problem:** adding new shape types and colors to the hierarchy will grow it exponentially

[Figures in this slide are extracted from https://refactoring.guru/design-patterns/bridge]

# Bridge: Solution

- **Essence of the problem:** we try to extend the shape classes in two independent dimensions
- **The Bridge pattern:** solve this problem by switching from inheritance to composition
- Extract one of the dimensions into a **separate** class hierarchy

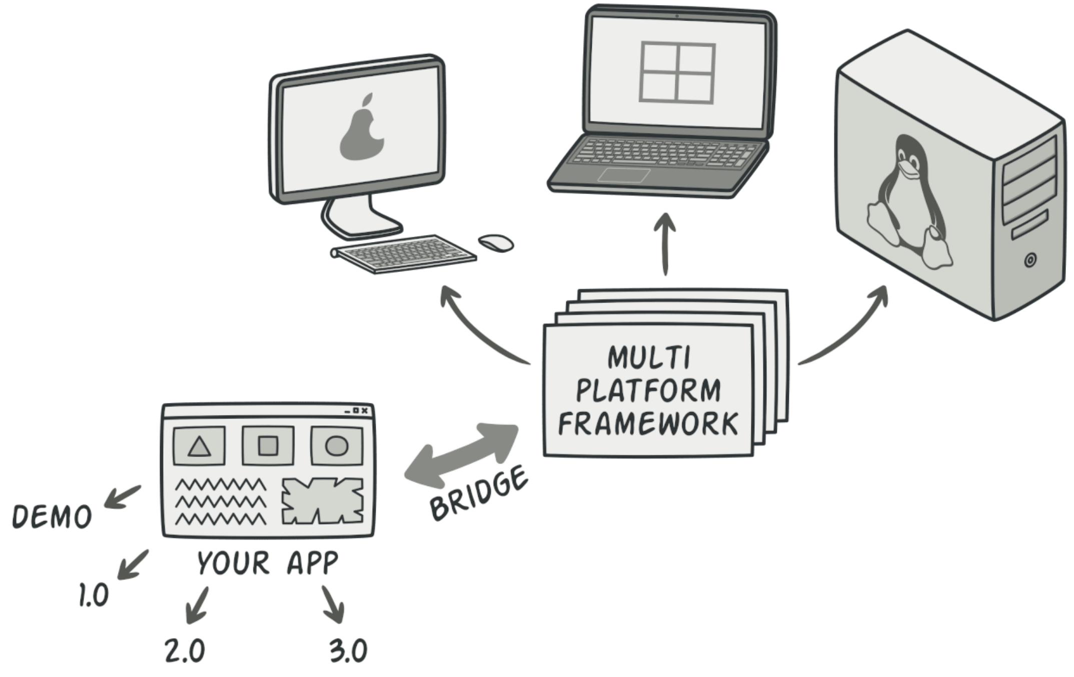


- The Shape class has a reference field pointing to a color object
- The reference **act as a bridge** between the Shape and Color classes

[Figures in this slide are extracted from https://refactoring.guru/design-patterns/bridge]

## Bridge: Solution (cont.)

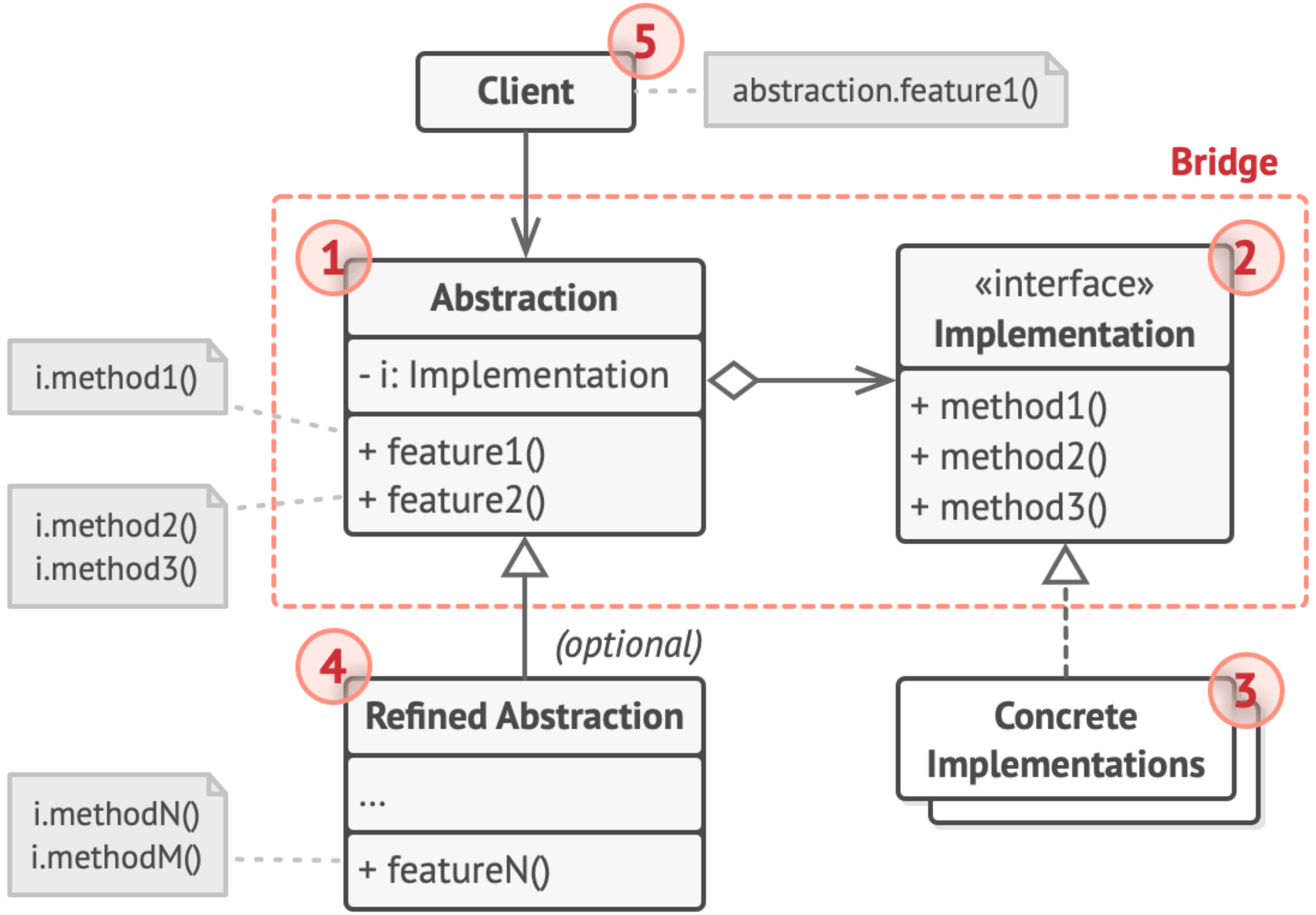
- **Abstraction and Implementation**
  - Introduced in the GoF book, as part of the Bridge definition
  - **Abstraction (interface)**: a high-level control layer for some entity
  - **Implementation (platform)**
- **Real-world example**
  - Extending an app in two independent directions
    - Implementing different GUIs
    - Supporting different APIs
  - Solution: dividing classes into two hierarchies
    - Abstraction: the GUI layer of the app
    - Implementation: the operating systems' APIs



- The abstraction object controls the appearance of the app, delegating the actual work to the linked implementation object
- Different implementations are interchangeable

[Figures in this slide are extracted from https://refactoring.guru/design-patterns/bridge]

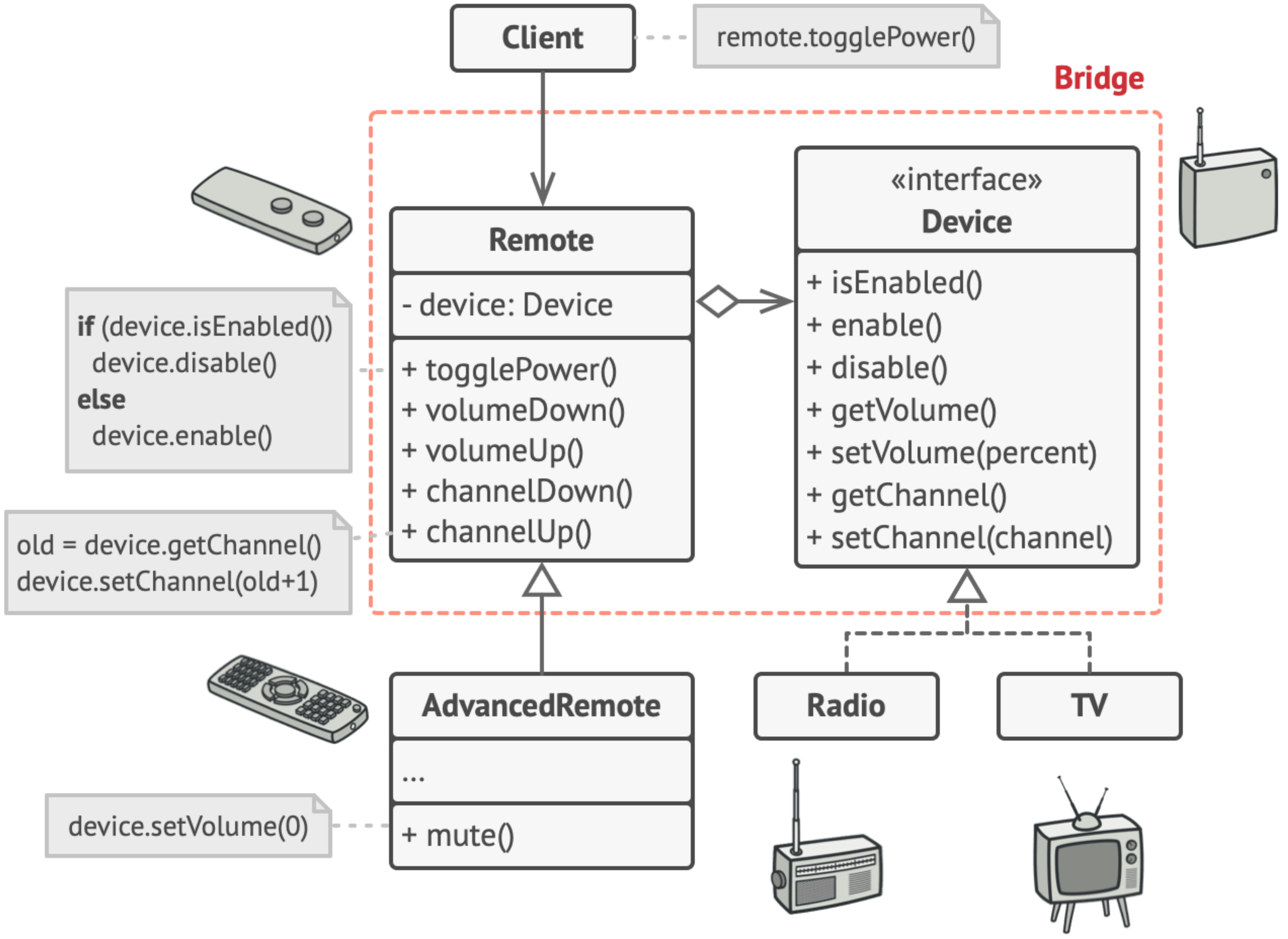
# Bridge: Structure



- The abstraction may list the same methods as the implementation, but usually the abstraction declares some **complex behaviors** that rely on a wide variety of **primitive operations** declared by the implementation

[Figures in this slide are extracted from https://refactoring.guru/design-patterns/bridge]

# Bridge: Example



[Figures in this slide are extracted from https://refactoring.guru/design-patterns/bridge]

## Bridge: Applicability

- When you want to **divide and organize** a monolithic class that has several variants of some functionalities
  - The bigger a class becomes, the harder it is to figure out how it works, and the longer it takes to make a change
  - The Bridge pattern lets you split the monolithic class into several class hierarchies
- When you need to **extend a class in several orthogonal (independent) dimensions**
  - The Bridge suggests that you extract a separate class hierarchy for each of the dimensions
  - The original class delegates the related work to the objects belonging to those hierarchies
- If you need to be able to **switch implementations at runtime**
  - Although optional, the Bridge pattern lets you replace the implementation object inside the abstraction: as easy as assigning a new value to a field

# Bridge: Implementation

1. Identify the **orthogonal dimensions** in your classes
   – Considerable independent concepts: **abstraction/platform, domain/infrastructure, front-end/back-end, or interface/implementation**
2. See what operations the client needs and **define them in the base abstraction class**
3. Determine the operations available on all platforms, and declare the ones that the abstraction needs in the **general implementation interface**
4. Create **concrete implementation classes**, following the implementation interface
5. Inside the abstraction class, add **a reference field** for the implementation type
6. If you have several variants of high-level logic, create **refined abstractions** for each variant by extending the base abstraction class
7. The client code should **pass an implementation object** to the abstraction's constructor

## Bridge: Pros and Cons

- **Pros**
  - You can create platform-independent classes and apps
  - The client code works with high-level abstractions without platform details
  - Open/Closed Principle: introducing new abstractions and implementations independently
  - Single Responsibility Principle: focusing on high-level logic in the abstraction, and on platform details in the implementation
- **Cons**
  - The code becomes more complicated by applying the pattern to a highly cohesive class