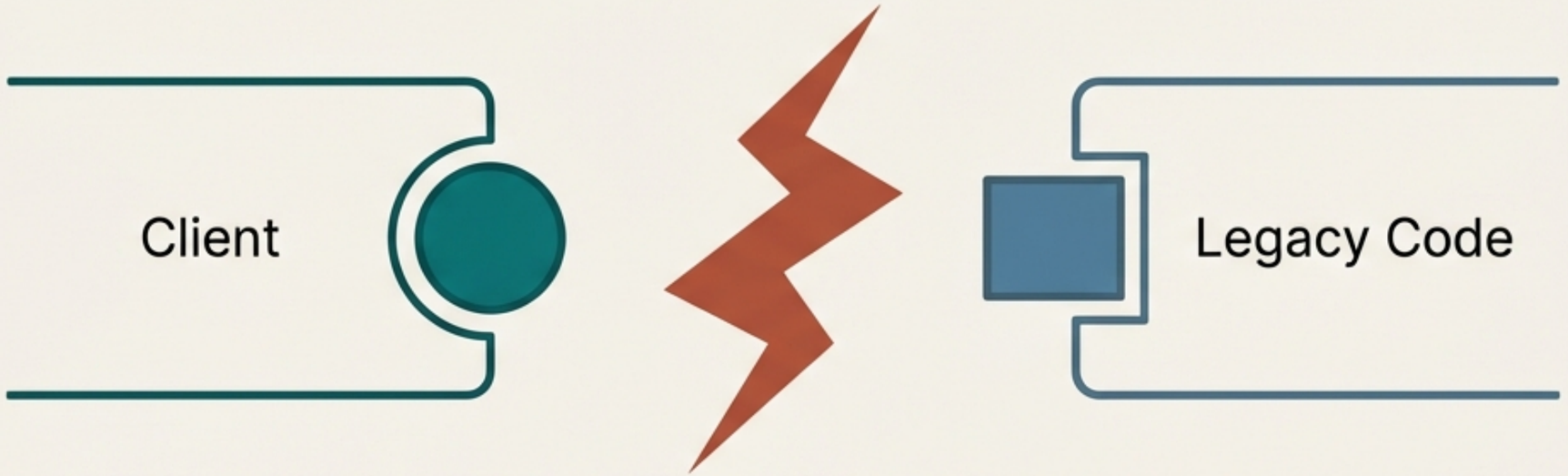
Rethinking the Adapter Pattern

Moving Beyond Hardware Analogies to a Clearer Software Model

The Fundamental Challenge: Making Incompatible Interfaces Cooperate



- You have a client that expects a specific interface (`Target`).
- You have an existing, valuable class with a different, incompatible interface (`Adaptee`).
- **The Constraint:** You cannot—or should not—change the source code of the existing class. This is common when dealing with:
  - Legacy systems.
  - Third-party libraries.

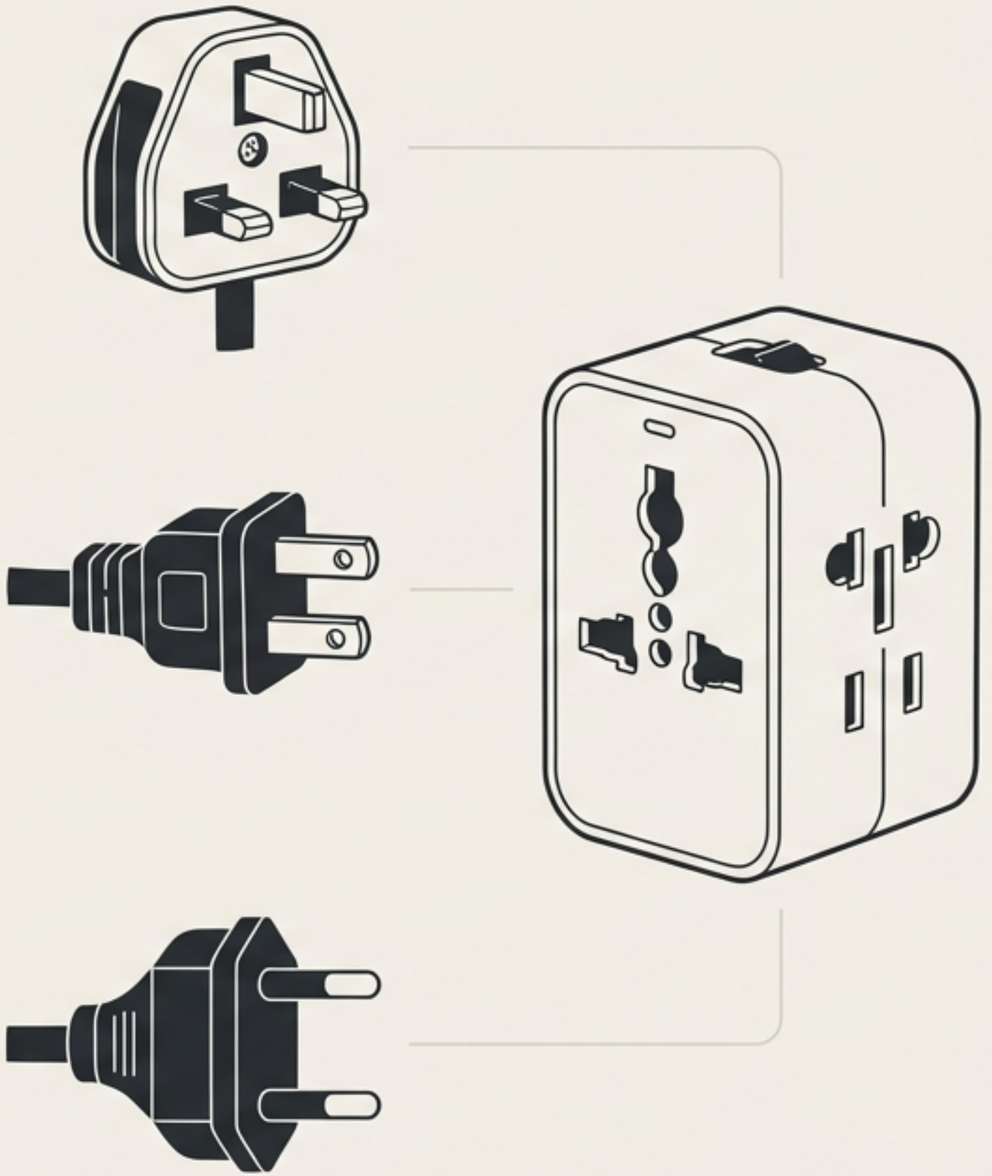
How do you bridge this gap without modifying existing, stable code?

The Familiar Analogy: The International Travel Adapter

The problem is simple: a device plug from one country doesn’t fit the wall socket of another.

The solution is an adapter that sits in between, converting the plug’s physical shape to match the socket.

This is often the first way developers learn to think about the pattern.



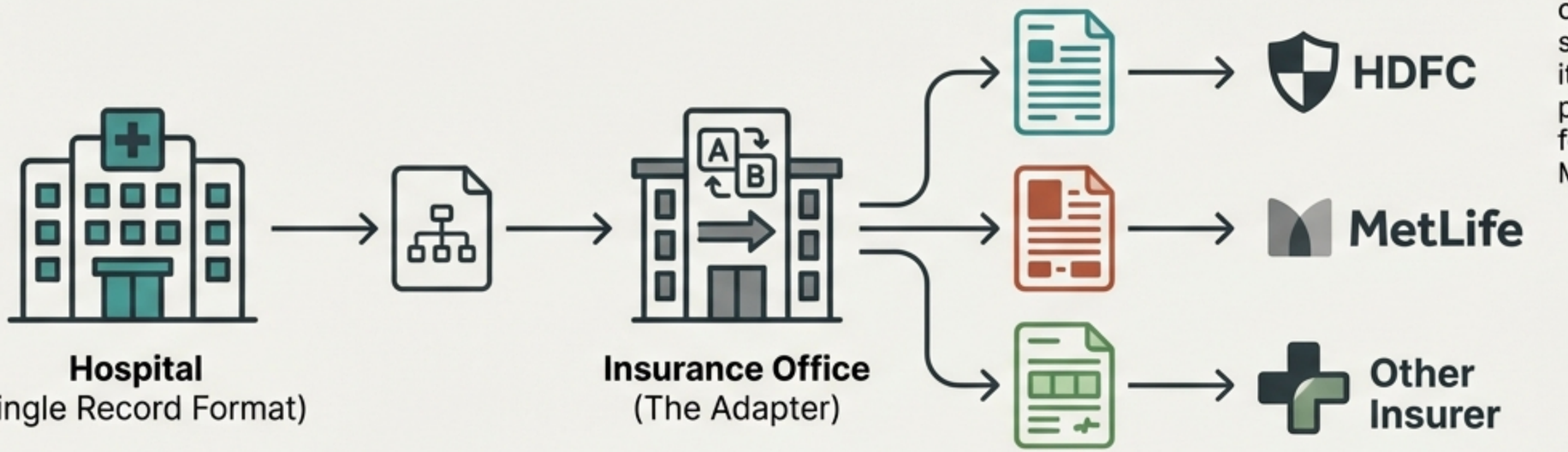
But is a Hardware Analogy Right for Software?

The charger analogy focuses on *physical conversion* (changing the shape of pins). Software, however, is fundamentally about *information translation*. The core task isn't changing electricity; it's **re-interpreting calls** and **re-formatting data**. A hardware example can be a mental barrier.

Hardware converts power. Software translates information.

A Better Analogy: The Medical Insurance Claims Office

1. **The Hospital:** Has its own standard format for patient medical records. This is their established, internal system.



Hospital
(Single Record Format)

Insurance Office
(The Adapter)

Insurance Companies
(Multiple Claim Formats)

2. **The Insurance Companies:** Each requires claims to be submitted in its own unique, proprietary format (HDFC, MetLife, etc.).

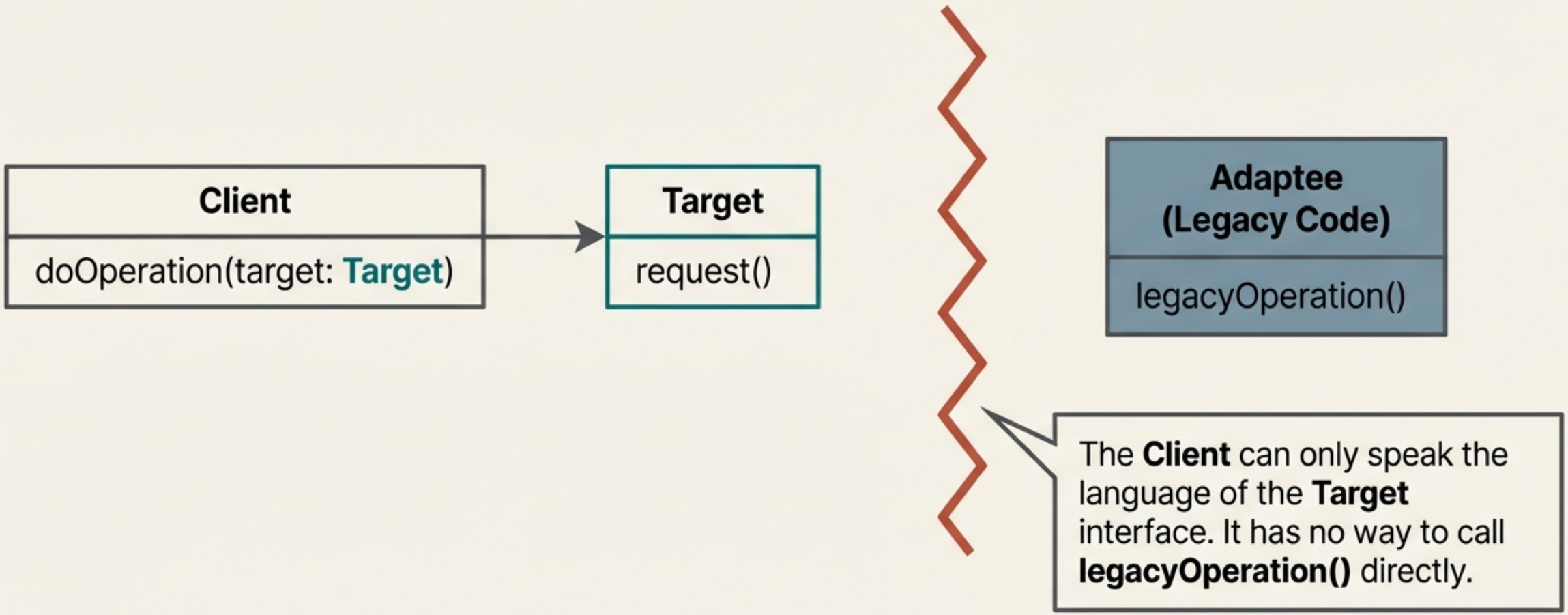
3. **The Bridge:** The hospital's insurance office takes the standard hospital records and *adapts* them into the specific format required by the patient's insurance provider.

Key Insight: The office doesn't change the medical facts; it translates them into a new structure.

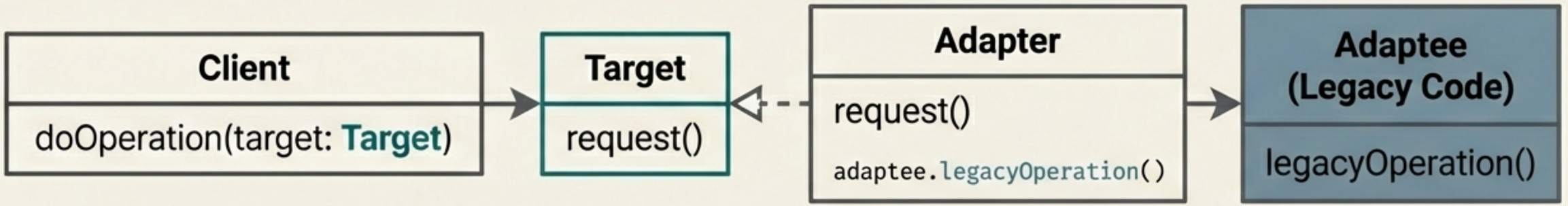
From Analogy to Architecture: Mapping the Concepts

Insurance Analogy	→	Adapter Pattern
The Client Requiring a Specific Format -> **You, the Patient**		The system using the target interface -> `Client`
The Desired Format -> **Insurance Company Claim Form**		The required interface -> `Target Interface`
The Existing Information Source -> **Hospital Medical Records**		The existing, incompatible class -> `Adaptee` (Legacy Code)
The Translator -> **Hospital Insurance Office**		The class that makes them compatible -> `Adapter`

The Structure of the Solution, Part 1: The Disconnect



The Structure of the Solution, Part 2: Bridging the Gap



The **Adapter** implements the **Target** interface the **Client** expects. Internally, it translates the `request()` call into a call on the wrapped **Adaptee** object.

Putting it to the Test: A Code Scenario

Goal: A `StudentClient` needs to process a list of `Student` objects.

The `Target` Interface

```
// The interface our client expects
public interface Student {
  String getName();
  String getSurname();
  String getEmail();
}
```

The Incompatible `Adaptee`

```
// An existing class we cannot change
public class SchoolStudent {
  public String getFirstName() { ... }
  public String getLastName() { ... }
  public String getEmailAddress() { ... }
}
```

The Problem

```
List<Student> students = new ArrayList<>();
students.add(new CollegeStudent(...)); // This works
students.add(new SchoolStudent(...)); // COMPILE ERROR!
```

The `SchoolStudent` class provides the right data, but through a different, incompatible interface. We can't add it to our list.

The Solution: Implementing the `SchoolStudentAdapter`

```
public class SchoolStudentAdapter implements Student { // Implements the Target
   private SchoolStudent schoolStudent; // Wraps the Adaptee

   public SchoolStudentAdapter(SchoolStudent schoolStudent) {
      this.schoolStudent = schoolStudent;
   }

   @Override
   public String getName() {
      return this.schoolStudent.getFirstName(); // Translation
   }

   @Override
   public String getSurname() {
      return this.schoolStudent.getLastName(); // Translation
   }

   @Override
   public String getEmail() {
      return this.schoolStudent.getEmailAddress(); // Translation
   }
}
```

The adapter implements `Student` and delegates the calls to the wrapped `SchoolStudent` instance, translating method names as needed.

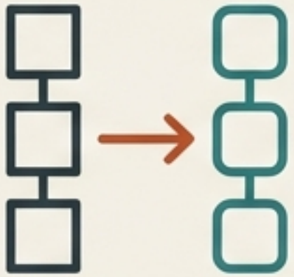
The Result: Seamless Integration

```
List<Student> students = new ArrayList<>();
SchoolStudent schoolStudent = new SchoolStudent(...);

students.add(new CollegeStudent(...)); // This still works

// Wrap the incompatible object in the adapter
students.add(new SchoolStudentAdapter(schoolStudent)); // SUCCESS!

// The client can now iterate and use all objects via the Student interface
for (Student student : students) {
    System.out.println(student.getSurname());
}
```

The `StudentClient` is completely unaware of the `SchoolStudent` class or the adapter. It continues to **work purely with the** `Student` interface, demonstrating that this is a client-focused pattern.

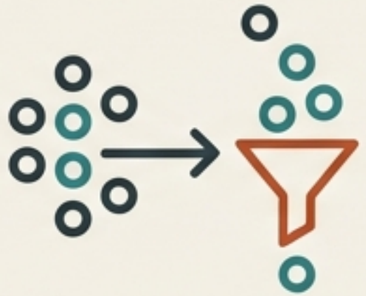
You’ve Probably Already Used This Pattern

The Adapter pattern isn't just a textbook concept; it's a practical tool used in core libraries.



`Arrays.asList(array)`

- **Function:** Takes a raw Java array and provides a `List` interface for it.
- **Mechanism:** It doesn't convert the array into a new list; it wraps it. The returned object is an adapter that makes the array *behave* like a list.



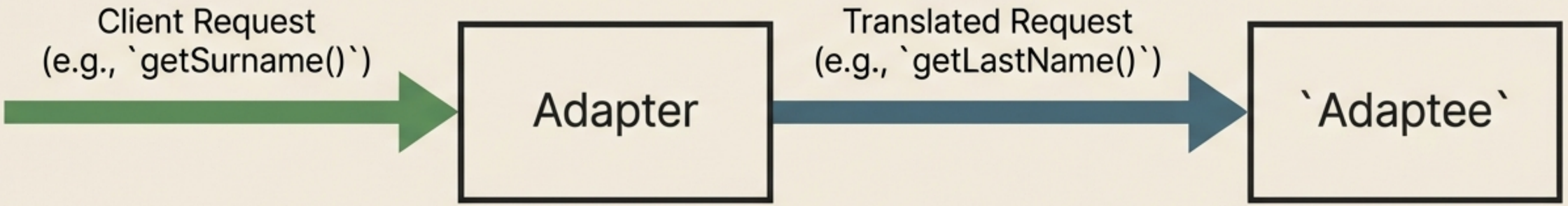
Java Stream Classes

- **Function:** Stream classes in Java often use adapters to connect various data sources (like collections or I/O channels) to the Stream API.

Explore the source code of these Java classes to see professional implementations of the Adapter pattern.

The Core Idea: It's About Translation, Not Conversion

The **Adapter pattern**'s primary role is to provide a different **interface** to an object, not to change its core **functionality** or **data**.



- Incorrect View (Charger): Converting one thing into another.
- **Correct View (Insurance Office)**: Translating requests so an existing object can be understood by a new client.

The Adapter Pattern: A Summary

What it is

A structural design pattern that allows objects with incompatible interfaces to collaborate.

When to Use It

- When you need to use an existing class, but its interface doesn't match the one you need.
- When you want to integrate legacy code into a new system without modifying the original code.
- When you need to create a reusable class that cooperates with unrelated or unforeseen classes.

Key Characteristics

- **Client-focused:** It shields the client from knowing about the incompatible class.
- **Simple to implement:** Often requires straightforward delegation.
- **Flexible:** You can write multiple adapters for a single class to have it work with many different systems.