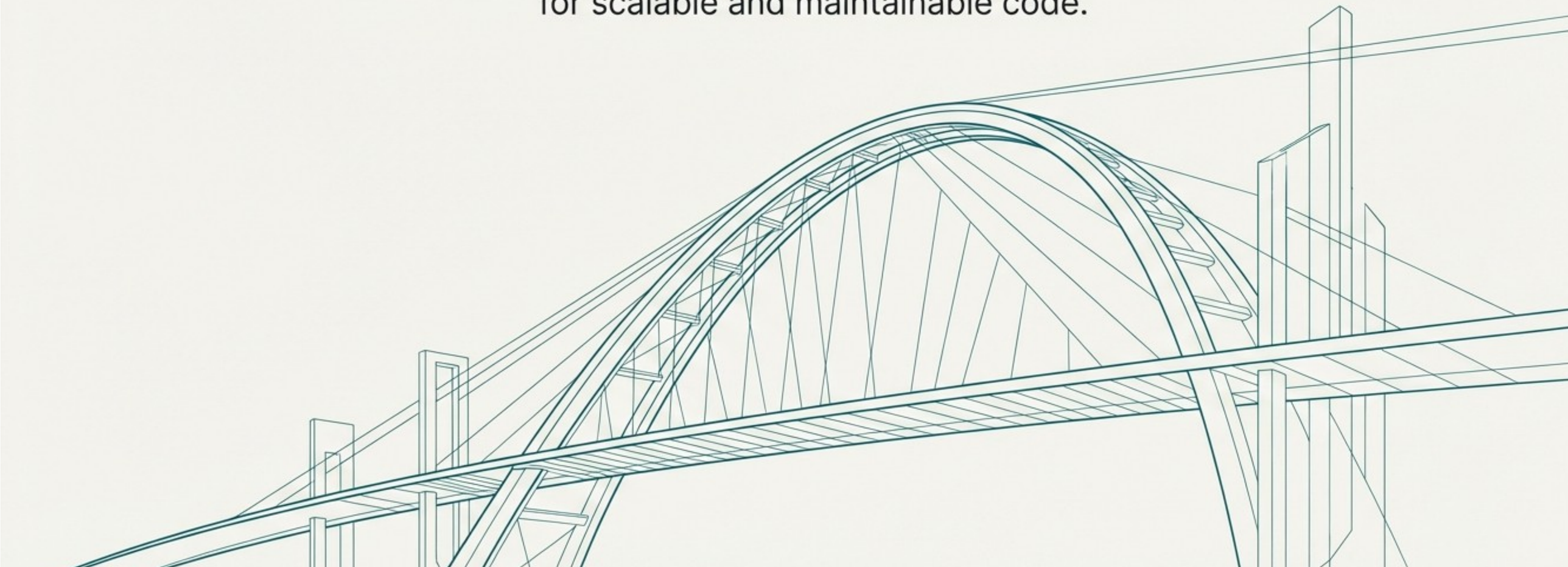
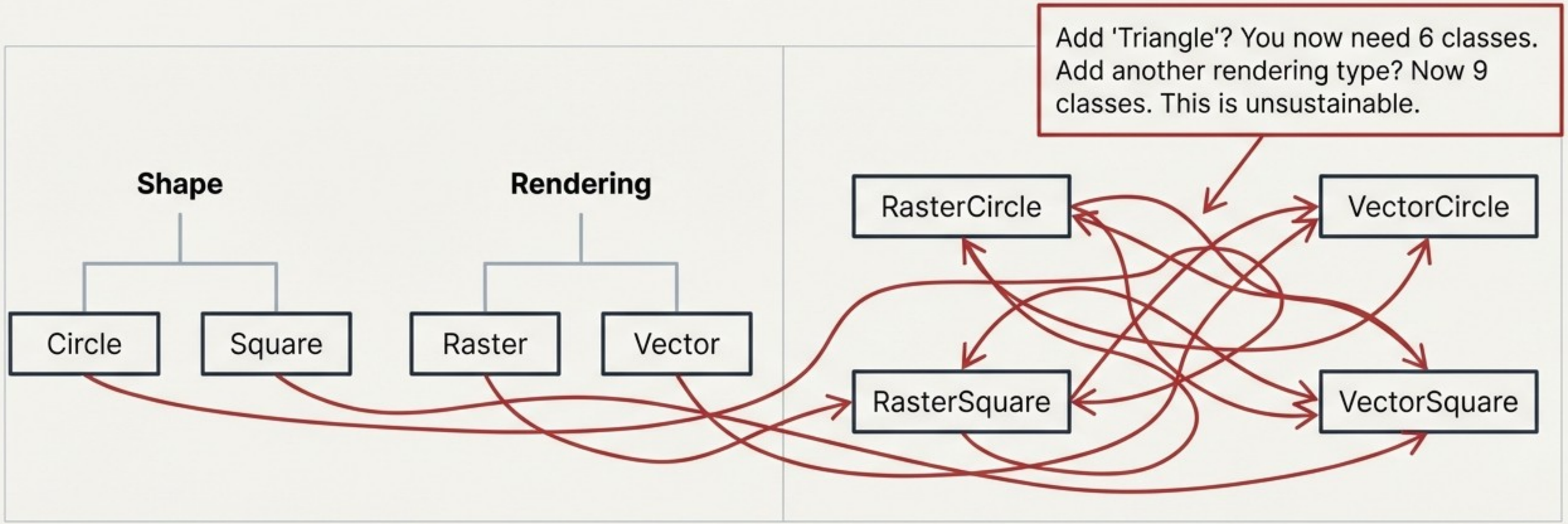
Beyond Inheritance: Building Flexible Systems with the Bridge Pattern

A practical guide to decoupling abstraction from implementation for scalable and maintainable code.



The Hidden Threat of Rigidity: The Class Explosion Problem

When two dimensions of change exist in a system, a common approach using inheritance creates a maintenance nightmare. For every new type added to one dimension, the number of required classes multiplies.



An Intuitive Bridge from the Real World

The Game Controller and the Console

Controller (The Abstraction - WHAT you do)

Play
Pause
Move

You don’t build a separate `PlayStationController` and `XboxController`. A single controller design works with any console. The Bridge pattern applies this same powerful idea to your code.

Consoles (The Implementation - HOW it’s done)

The Solution: Decoupling ‘What’ from ‘How’

The Bridge pattern separates a system's high-level logic (the **Abstraction**) from its low-level platform-specific details (the **Implementation**), allowing both to evolve independently.

Abstraction
(WHAT we do)

Implementation
(HOW it is done)

This structure replaces a rigid inheritance tree with a flexible composition, creating a “bridge” between two independent hierarchies.

Blueprint for Decoupling: An Enterprise Reporting System

We need to generate different types of reports and export them in various formats.

Dimension 1: Report Types

`SalesReport`

`InventoryReport`

Dimension 2: Export Formats

`PDF`

`Excel`

To freely combine any report type with any export format without creating a class for every combination (e.g., `PdfSalesReport`, `ExcelInventoryReport`).

Step 1: Build the ‘How’ – The Implementation Layer

First, we define an interface for all our low-level implementation details. These are the concrete ‘exporters’ that do the actual work.

```
// The Implementation Interface
public interface IReportExporter
{
    void Export(string content);
}

// Concrete Implementations
public class PdfExporter : IReportExporter
{
    public void Export(string content)
    {
        Console.WriteLine("Exporting PDF: " + content);
    }
}

public class ExcelExporter : IReportExporter
{
    public void Export(string content)
    {
        Console.WriteLine("Exporting Excel: " + content);
    }
}
```

Step 2: Define the ‘What’ – The Abstraction Layer

Next, we create the high-level abstraction. Instead of inheriting from an exporter, it holds a reference to one.

This composition is the bridge.

```
// The Abstraction
public abstract class Report
{
    // The "Bridge" is this reference to the implementation
    protected readonly IReportExporter _exporter;

    protected Report(IReportExporter exporter)
    {
        _exporter = exporter; // Injected on construction
    }

    public abstract void Generate();
}

// Concrete Abstractions
public class SalesReport : Report
{
    public SalesReport(IReportExporter exporter) : base(exporter) {}
    public override void Generate() => _exporter.Export("Sales Report Data");
}
```

This is the bridge!

Step 3: Cross the Bridge – Connecting the Layers

The client code can now create any combination of report and exporter dynamically. The report's logic is separate from the export's logic.

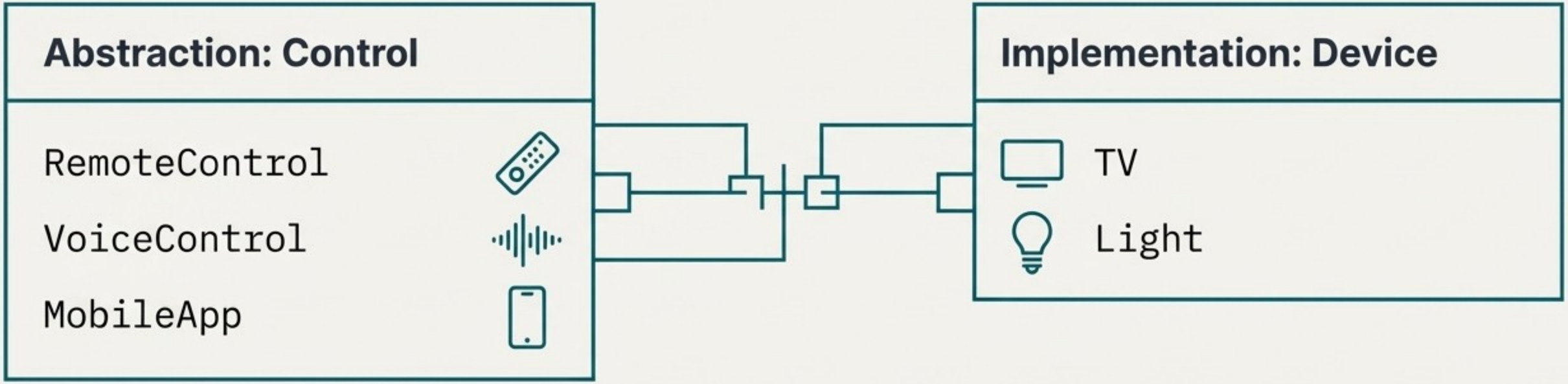
```
// Client code can now mix and match freely
var salesPdf = new SalesReport(new PdfExporter());
salesPdf.Generate();
// >> Output: Exporting PDF: Sales Report Data

var inventoryExcel = new InventoryReport(new ExcelExporter());
inventoryExcel.Generate();
// >> Output: Exporting Excel: Inventory Report Data
```

- Report logic is separate.
- Export logic is separate.
- Combine any report with any exporter.

The Pattern in Action: Smart Home Controls

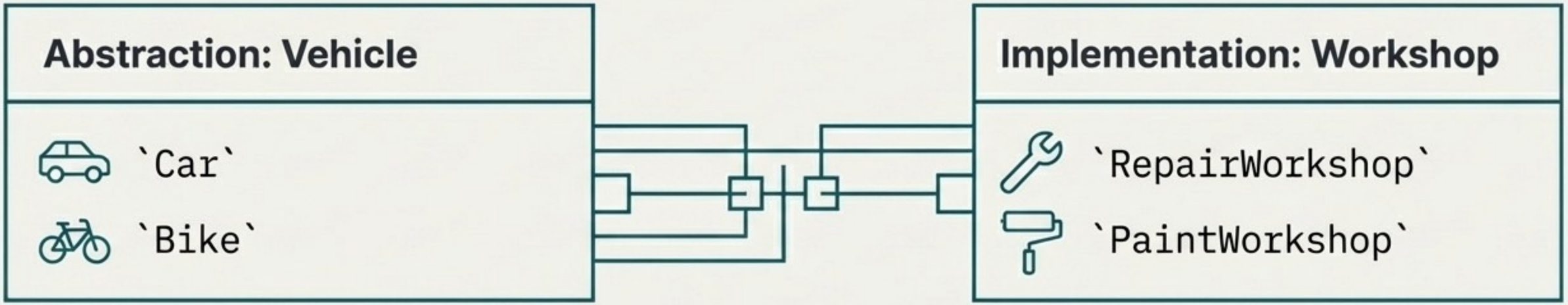
The same principle applies to controlling different devices. The type of control is separate from the device it operates.



A `VoiceControl` can operate a `TV` or a `Light` without needing specific `VoiceControlForTV` classes.

The Pattern in Action: Vehicle Service Centres

Decouple vehicle types from the types of service they can receive.
Avoids rigid classes like `CarRepair`, `CarPaint`, `BikeRepair`, and `BikePaint`.

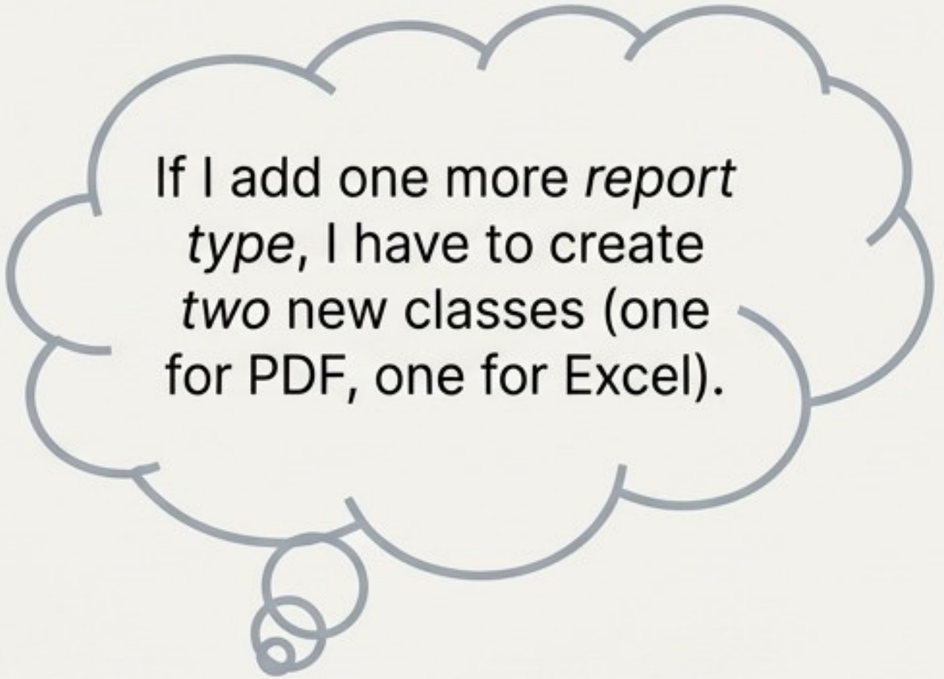


Any `Vehicle` can be sent to any `Workshop` dynamically, providing maximum service flexibility.

Your Checklist: When to Build a Bridge

Use the Bridge pattern when you need to vary two (or more) dimensions of a system independently.

- ☑ You want to decouple an abstraction from its implementation.
- ☑ You foresee an “inheritance explosion” where subclasses would multiply.
- ☑ You need to switch implementations at runtime.
- ☑ You want to share an implementation among multiple, different abstractions.



A Word of Caution: When to Avoid the Bridge

Like any pattern, the Bridge introduces a layer of complexity. Don’t force it where it isn’t needed.

Avoid If...

- ✕ The system is simple and unlikely to change.
- ✕ There is only one dimension of variation (a simpler pattern may suffice).
- ✕ The abstraction and implementation are intrinsically linked and will never need to vary independently.

Good design is about choosing the right tool for the job. Sometimes, simplicity is the best solution.

Built on a Foundation of SOLID Principles

The Bridge pattern is a practical application of core design principles that lead to robust, maintainable systems.

SRP (Single Responsibility Principle)

The Abstraction is concerned with high-level logic, while the Implementation handles platform details. Each has one clear job.

OCP (Open/Closed Principle)

You can introduce new abstractions or new implementations without modifying existing client code. The system is open for extension, closed for modification.

DIP (Dependency Inversion Principle)

The high-level Abstraction does not depend on low-level concrete Implementations. Both depend on interfaces.

The Bridge Pattern at a Glance

Problem	Two or more independent dimensions of change cause a 'class explosion' when using inheritance.
Solution	Separate the Abstraction (what) from the Implementation (how) and connect them via a 'bridge'.
Key Idea	Favour composition over inheritance to combine features dynamically.
Benefits	Creates flexible, scalable, and maintainable systems while avoiding code duplication.
Common Uses	Reporting systems, device drivers, UI frameworks, and any system managing multiple platforms or versions.

The Reward: Lasting Flexibility

The Bridge is more than just a pattern; it's a strategic decision to build systems that can adapt and grow. By decoupling what your system does from how it does it, you create software that is not only easier to maintain today, but prepared for the unknown requirements of tomorrow.