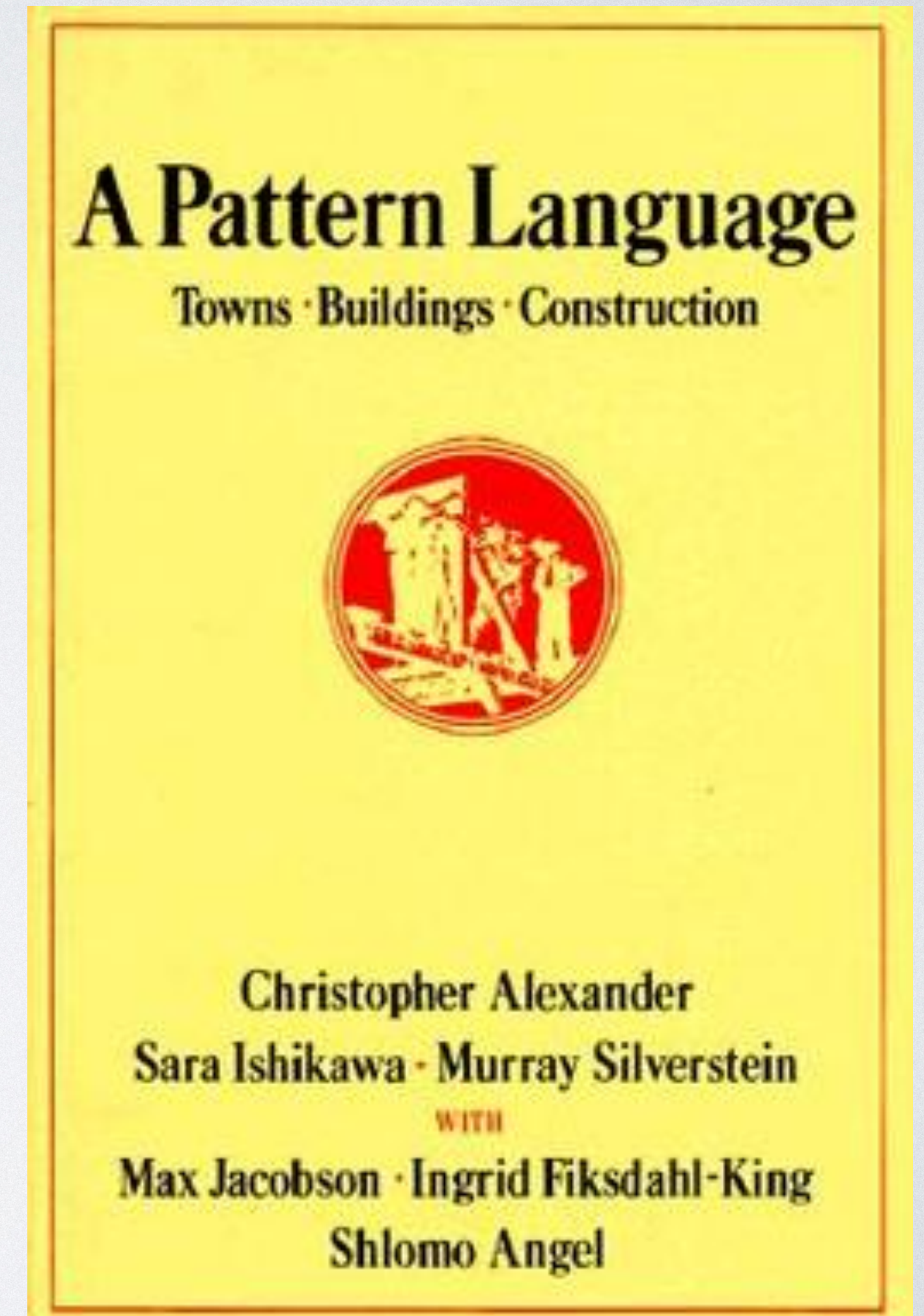


# Design Patterns (I)

# Patterns

- Often, the same problem arises in multiple contexts
- "A Pattern Language" describes 253 patterns for architects: "All 253 patterns together form a language."
- "each pattern represents our current best guess as to what arrangement of the physical environment will work to solve the problem presented. The empirical questions center on the problem—does it occur and is it felt in the way we describe it?—and the solution—does the arrangement we propose solve the problem? "



# A Pattern Language Example

- "When they have a choice, people will always gravitate to those rooms which have light on two sides, and leave the rooms which are lit only from one side unused and empty."
- "Locate each room so that it has outdoor space outside it on at least two sides, and then place windows in these outdoor walls so that natural light falls into every room from more than one direction."

# Lighting: Two Sides vs One Side



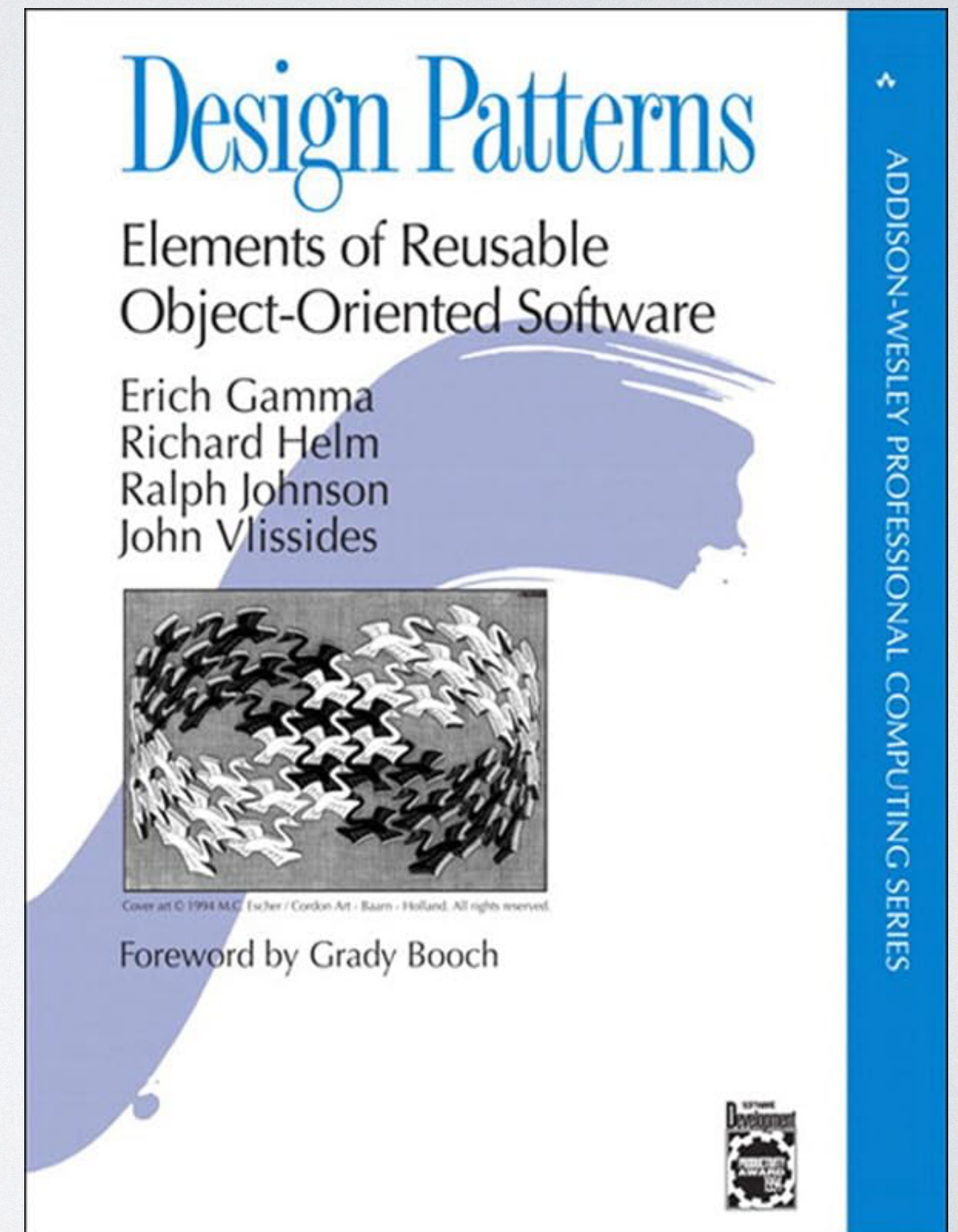
# Wrinkle the Edge

- "Wrinkling the edge" of a building enables natural light from more than one side of each room



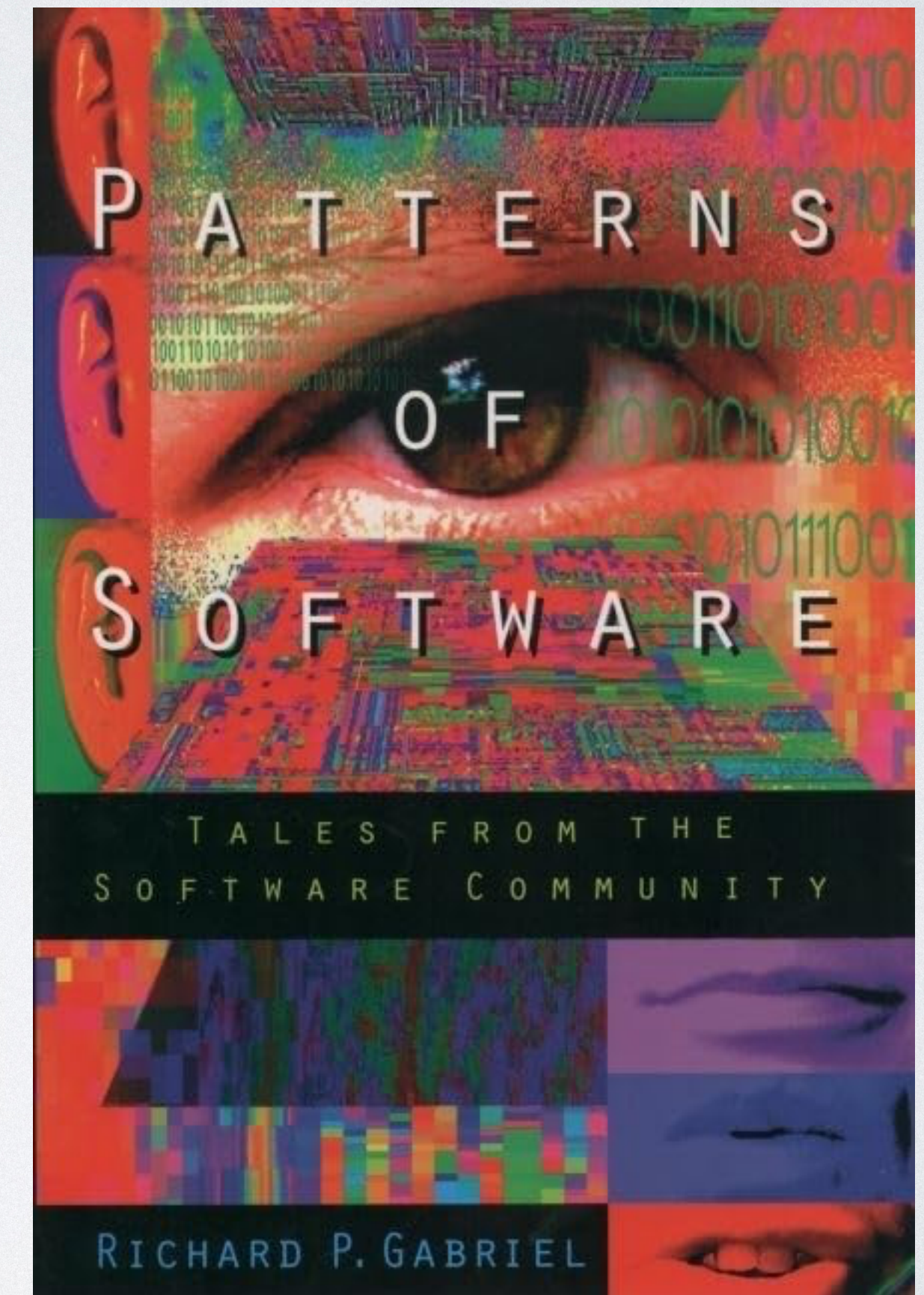
# Object-Oriented Patterns

- The "Gang of Four" book (1994) describes 23 patterns
- The problems they address are still common — and so are the patterns!
- Some patterns solve multiple different problems



# Software Patterns

- "Programs live and grow, and their inhabitants—the programmers—need to work with that program the way the farmer works with the homestead." (Richard P. Gabriel)



# Each Pattern Solves Certain Problems

- With practice, you will see those problems and think "aha! I need THIS pattern!"
- Not every problem has a pattern-based solution.
  - At least, not a named pattern in the book.
- But when you now might think "now what?" eventually you'll think "the usual way to do this is...".

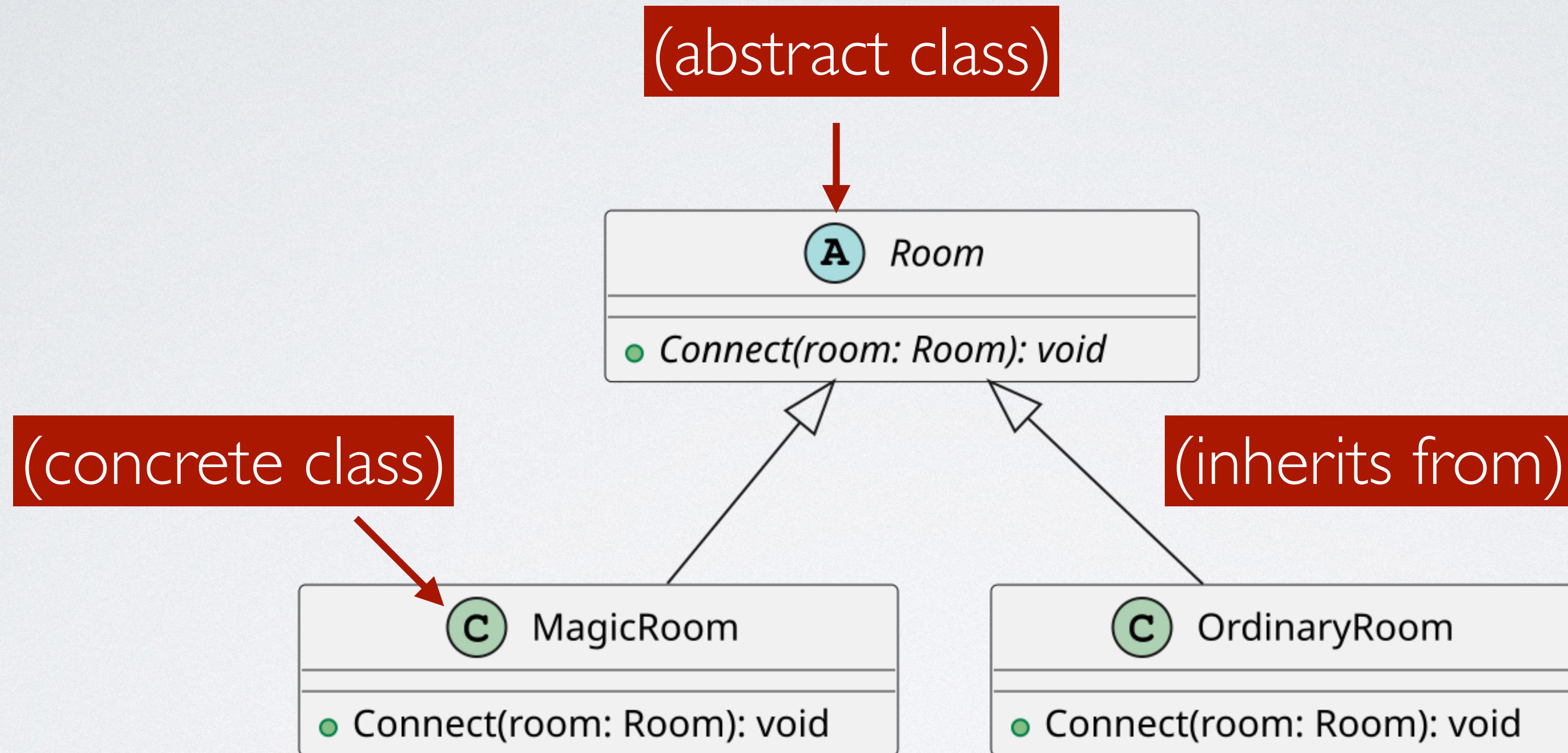
# Factory Pattern

- Sometimes object creation is complicated
  - Object needs to be "hooked up" or which object to create depends on something
  - Putting this logic everywhere would violate DRY
- Solution: put complicated logic in a "factory"

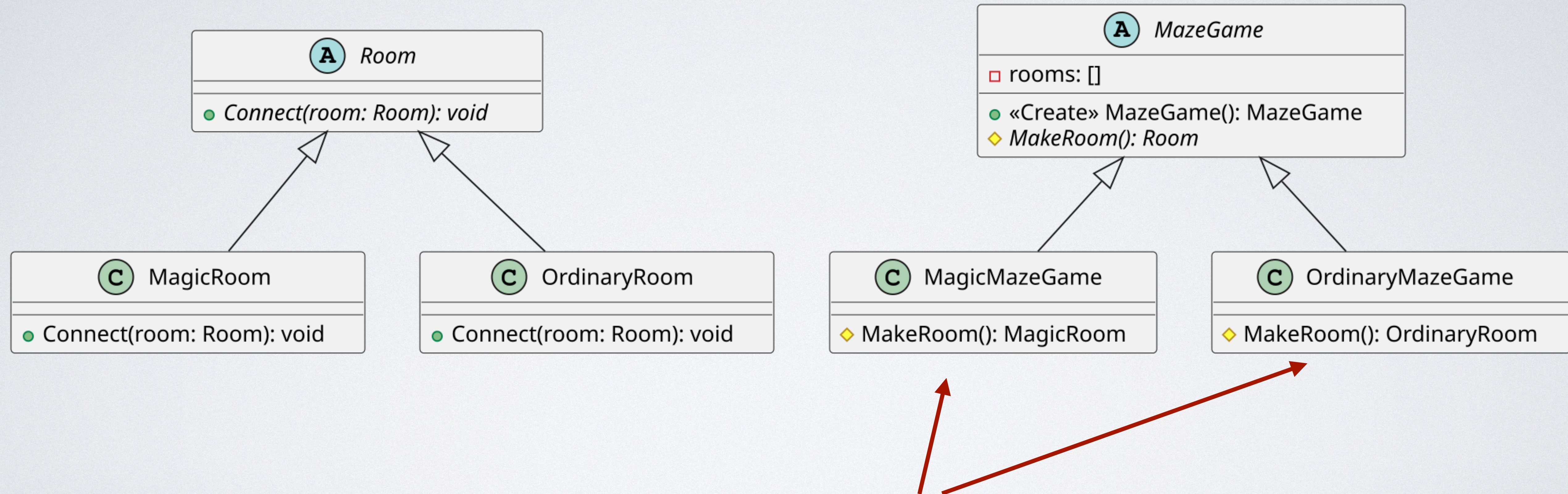
# Example

- Two varieties of maze games
  - OrdinaryMazeGame uses OrdinaryRoom instances
  - MagicMazeGame uses MagicMazeRoom instances
- Want to re-use the rest of the game logic (don't want two game implementations)

# Rooms



# MazeGame



MakeRoom() is a factory method: it always makes a new room of the right type



# Factories Are Beneficial Because

- A. They improve efficiency by keeping creation code in cache
- B. They consolidate complex construction logic in one place, avoiding repetitive code
- C. They enable sharing of immutable objects
- D. B and C
- E. A and B

# Singleton Pattern

- Sometimes there should be only ONE of something.
- Often, you only want one factory!
- Other examples: logger; cache; thread pool
- Use *sparingly*
  - *Singletons* smell an awful lot like *global variables*

# Singleton in Java

```
class ThreadPool {  
    static ThreadPool instance;
```

```
    private ThreadPool() {}
```

Private constructor forces singleton access

```
    public static ThreadPool getInstance() {  
        if (instance == null) {  
            instance = new ThreadPool();  
        }  
        return instance;  
    }  
}
```

Not thread-safe!

```
    // Instance methods go here  
}
```

# Combining Factory and Singleton

```
class Month {  
    private int month;  
  
    public String monthName() {...}  
}
```

- How many different Month instances do we need to allocate?
- No need to ever have more than 12!

# Want To Re-Use Month Objects

```
class MonthFactory {
    static MonthFactory instance;
    Month[] allMonths;

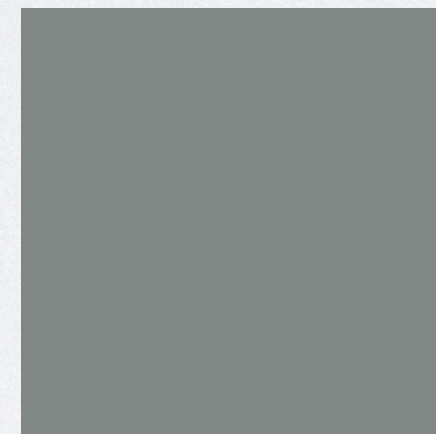
    private MonthFactory() {};
    public static MonthFactory getInstance() {
        if (instance == null) {
            instance = new MonthFactory();
        }
        return instance;
    }

    public Month createMonth(int month) {
        if (month >= 0 && month <= 11) {
            return allMonths[month];
        }
        else {
            // throw...
        }
    }
}
```

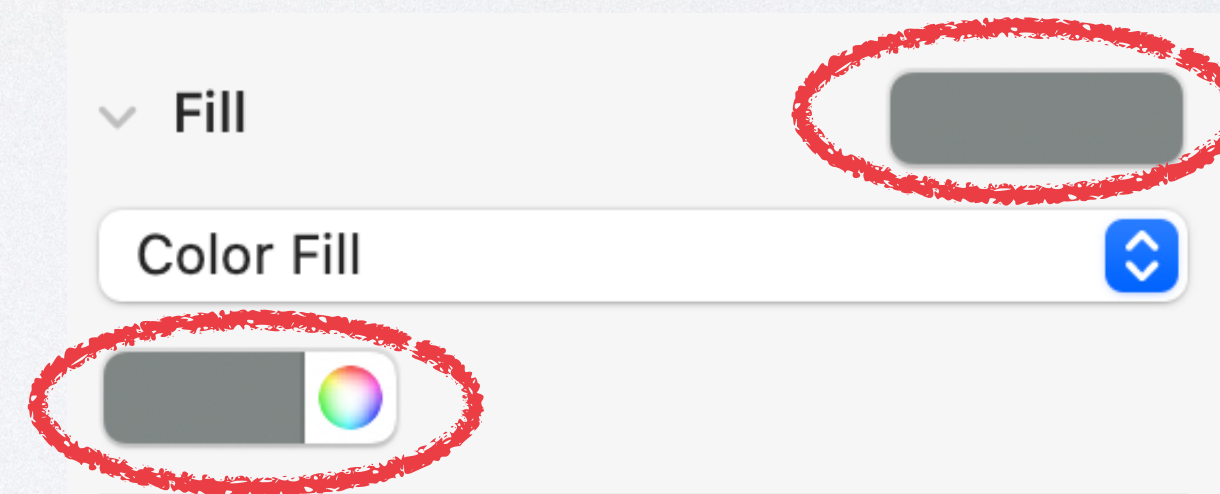
# Observer Pattern

- Suppose you have a slideshow application (like this one)

- You can draw shapes

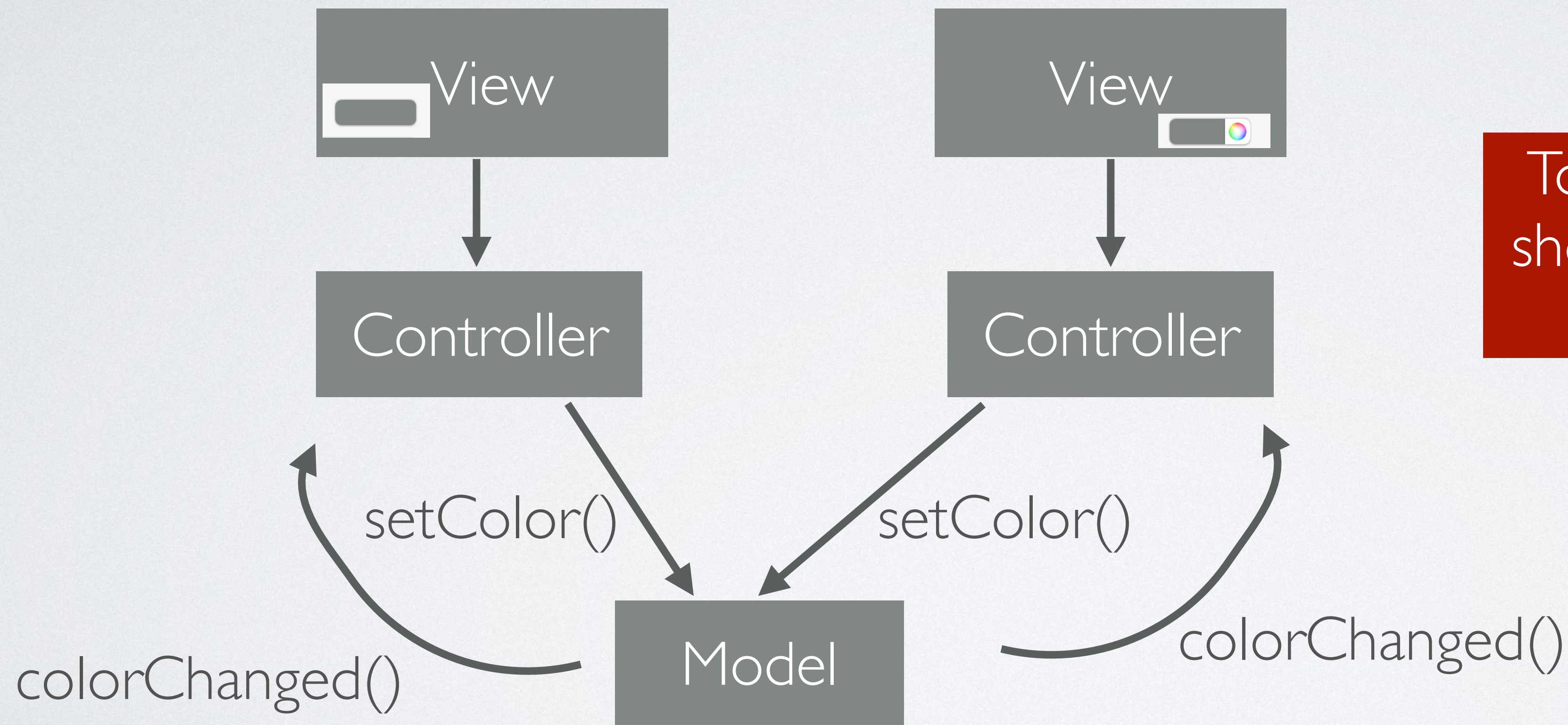


- and you can set their colors:



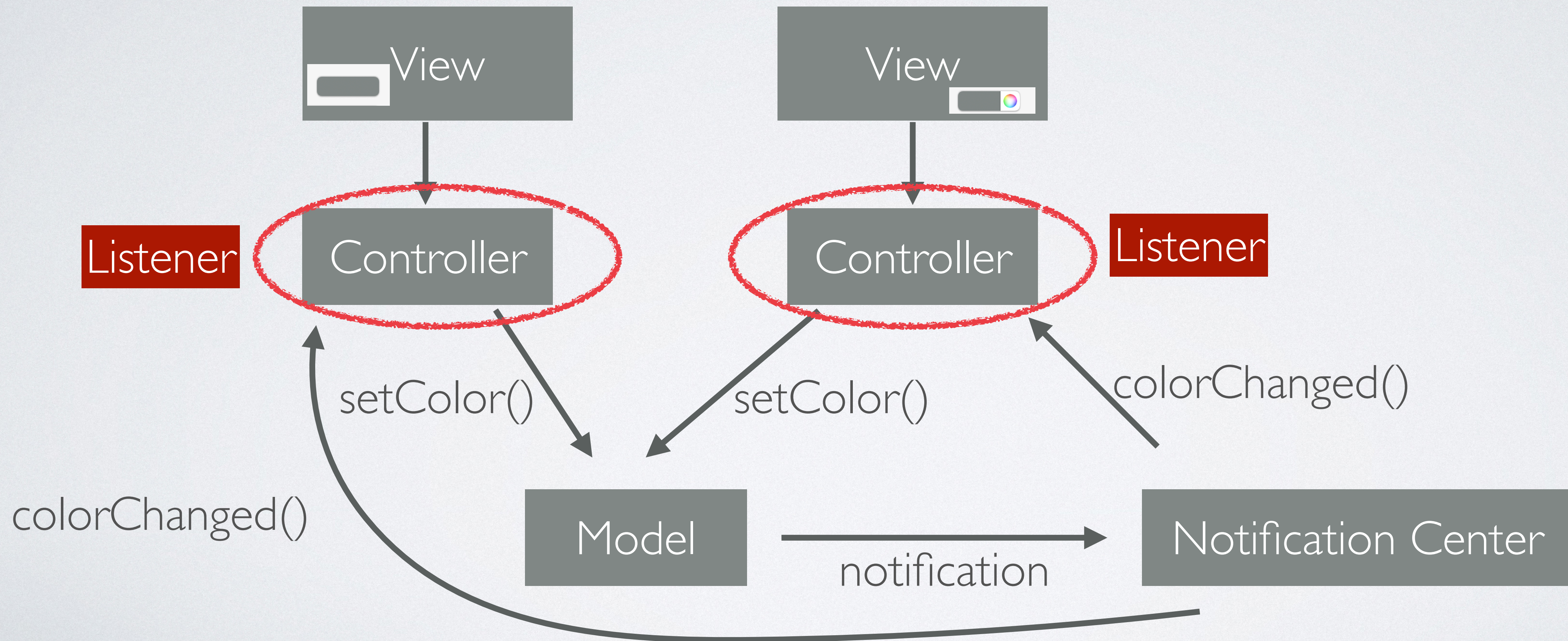
- Changing the color with either affordance updates both.

# A First Try



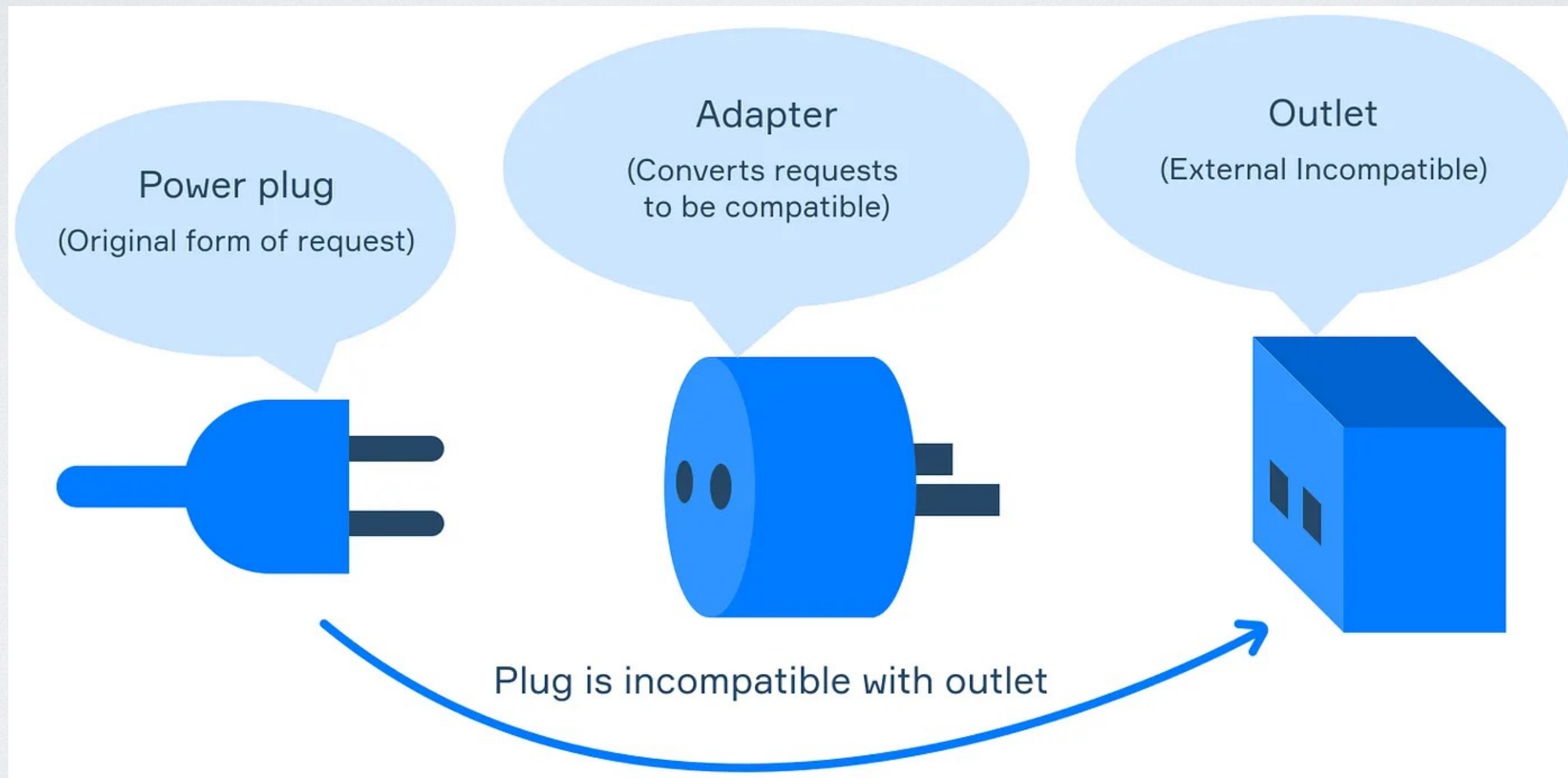
Too much coupling! Model shouldn't know about each controller.

# Observer



# Adapter Pattern

- Goal: re-use existing component in a new context
- Problem: new context assumes a different interface
- Solution: Adapter

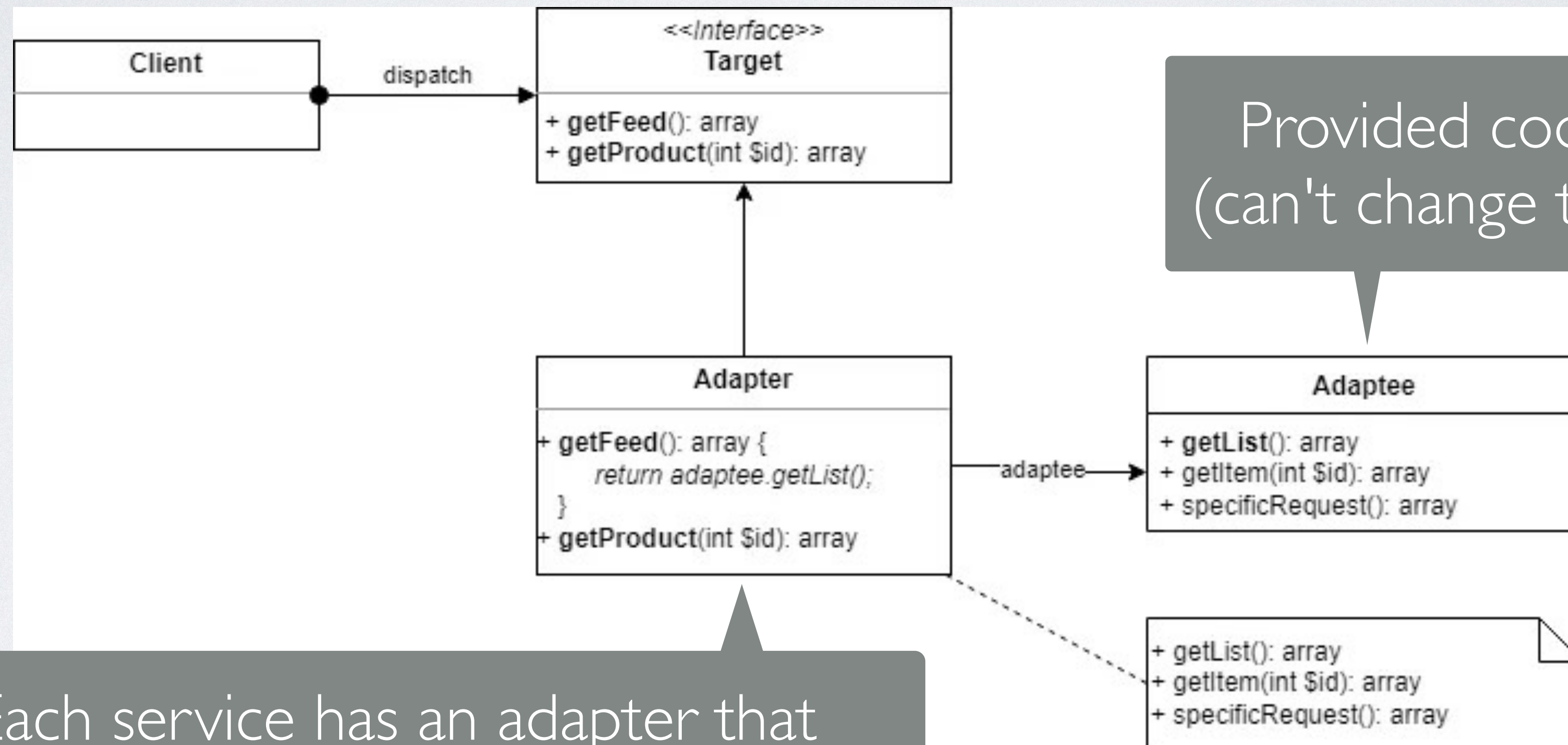


# Example

- Building a shopping app that aggregates data from different sources (Shopify, BigCommerce, etc.)
- Problem: different sources have different APIs
- Solution: define a common API; wrap each source in an adapter

Existing code (don't want to change this)

Generic interface



Provided code (can't change this)

Each service has an adapter that implements the generic interface

# ICA

- Think of your current project. Identify a place where one of these patterns might be appropriate, and explain why.