Real World Patterns; Bridge, Builder

CSCI 4448/5448: Object-Oriented Analysis & Design

Lecture 28

Acknowledgement & Materials Copyright

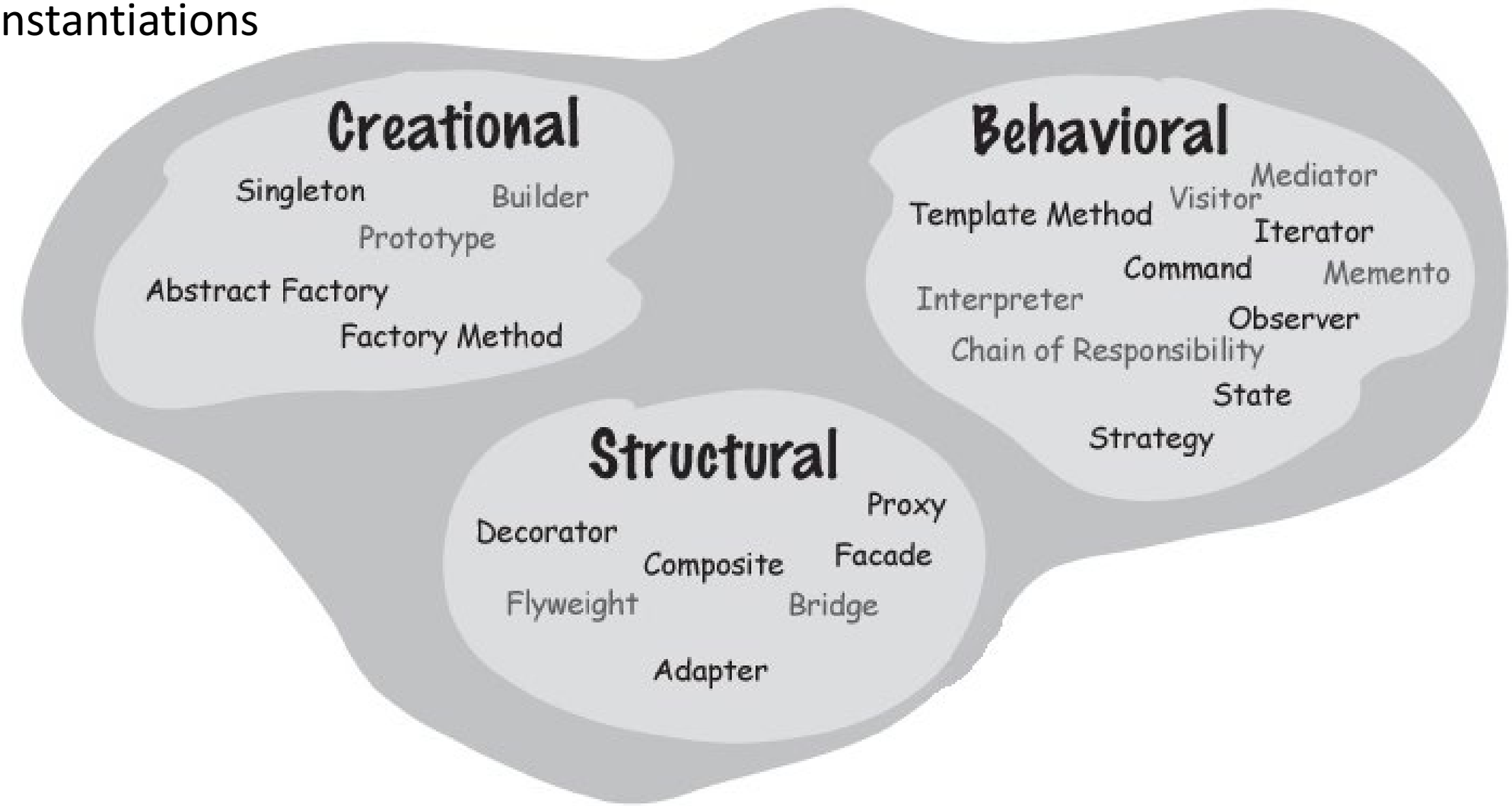
- I'd like to start by acknowledging Dr. Ken Anderson
- Ken is a Professor and the Chair of the Department of Computer Science
- Ken taught OOAD on several occasions, and has graciously allowed me to use his copyrighted material for this instance of the class
- Although I will modify the materials to update and personalize this class, the original materials this class is based on are all copyrighted © Kenneth M. Anderson; the materials are used with his consent; and this use in no way challenges his copyright

Head First Design Patterns

- Chapter 13 – Better Living with Patterns: Patterns in the Real World
  - Not too much here we haven't covered…
- Appendix A – Leftover Patterns
  - Bridge
  - Builder
  - Flyweight
  - Chain of Responsibility
  - Interpreter
  - Mediator
  - Memento
  - Prototype
  - Visitor

Pattern Classification

- The Gang of Four classified patterns in three ways
  - The **behavioral** patterns are used to manage variation in behaviors, interaction, and distribution of responsibility
  - The **structural** patterns are useful to compose classes or objects in larger structures
  - The **creational** patterns are used to create objects and decouple clients from instantiations

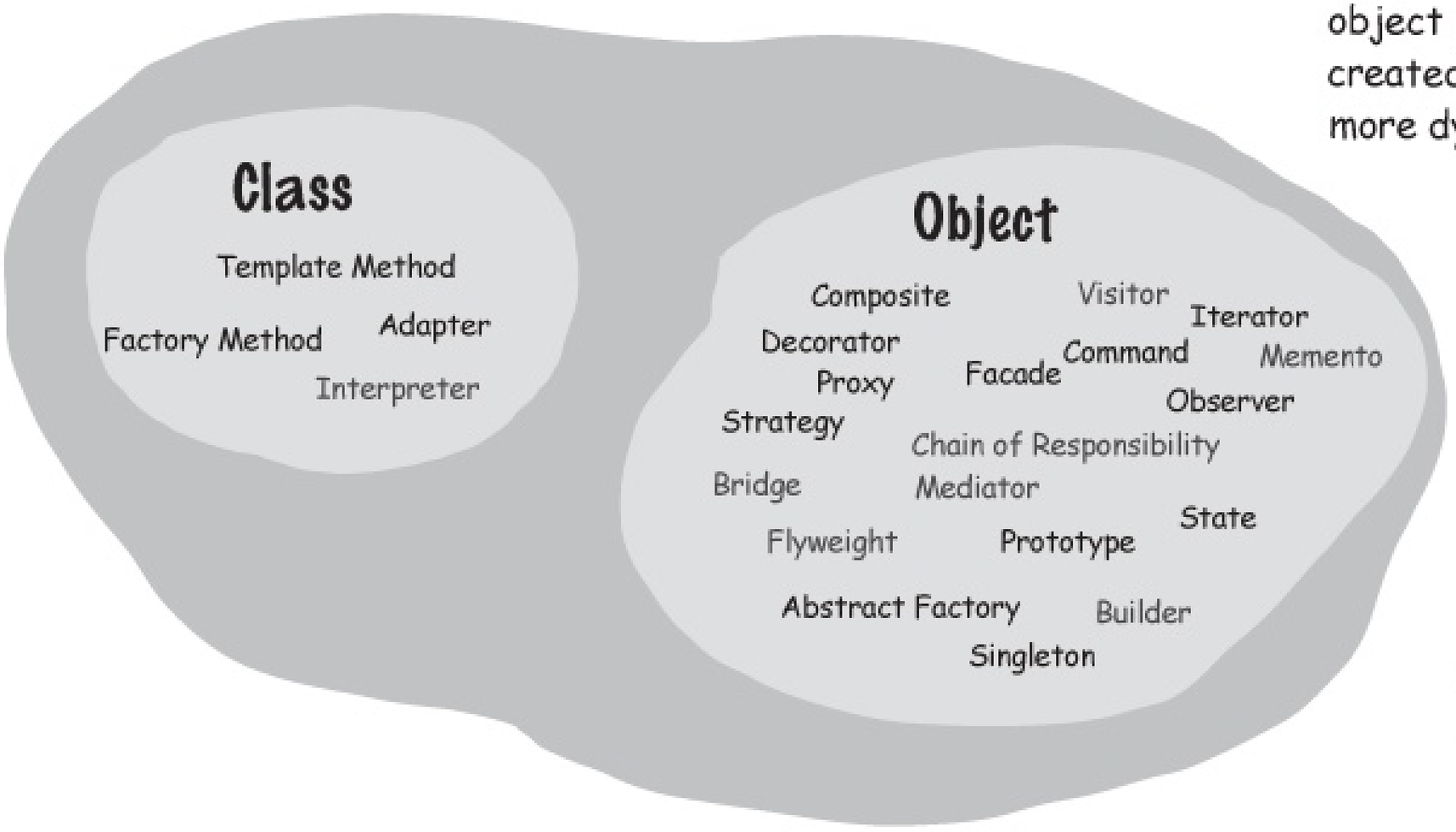


Pattern Classification

- Are the patterns related to Classes or Objects?

Class Patterns describe how relationships between classes are defined via inheritance. Relationships in class patterns are established at compile time.

Object Patterns describe relationships between objects and are primarily defined by composition. Relationships in object patterns are typically created at runtime and are more dynamic and flexible.



Notice there are a lot more object patterns than class patterns!

Pattern Libraries & Documenting Patterns

- **Name** – Identifies the pattern
- **Classification** – Category of pattern
- **Intent** – Short form of what pattern does
- **Motivation** – Scenario with problem and solution
- **Applicability** – Situations to use the pattern
- **Structure** – UML diagram showing relationships
- **Participants** – Classes/objects in design
- **Collaborations** – How participants work together
- **Consequences** – Any good/bad effects from using the pattern
- **Implementation/Sample Code** – How to use it
- **Known Uses** – Examples in real systems
- **Related Patterns** – How this pattern relates to others

Stages of Using Patterns

- Beginners – try to use patterns everywhere
- Intermediate – see where patterns are needed and where they aren’t
- Experienced – see where patterns fit naturally AND don’t let pattern knowledge overly influence overall design decisions

The Bridge Pattern

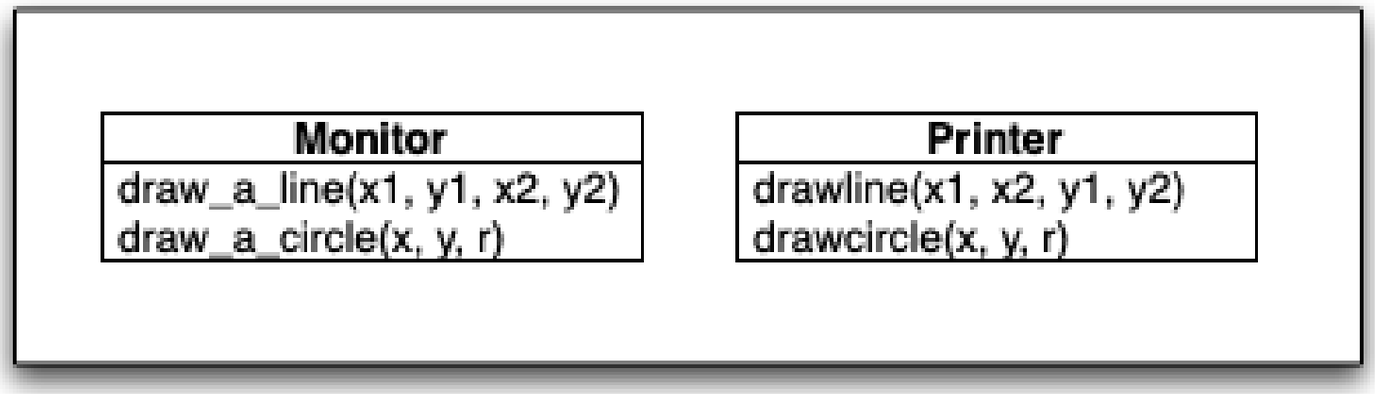
- The Gang of Four book says the intent of the pattern is to “decouple an abstraction from its implementation so that the two can vary independently”
- Tough one to get a handle on - break it down
  - Decouple – things behaving independently
  - Abstraction – conceptual relation between things
  - Implementation – here, speaking to objects the abstract class and derivations of that class are using (not the derivations of the abstract class – i.e. concrete classes)
- What does it mean
  - Allows a set of varying abstract objects to implement their operations in a number of ways **in a scalable fashion**

Bottom-Up Design

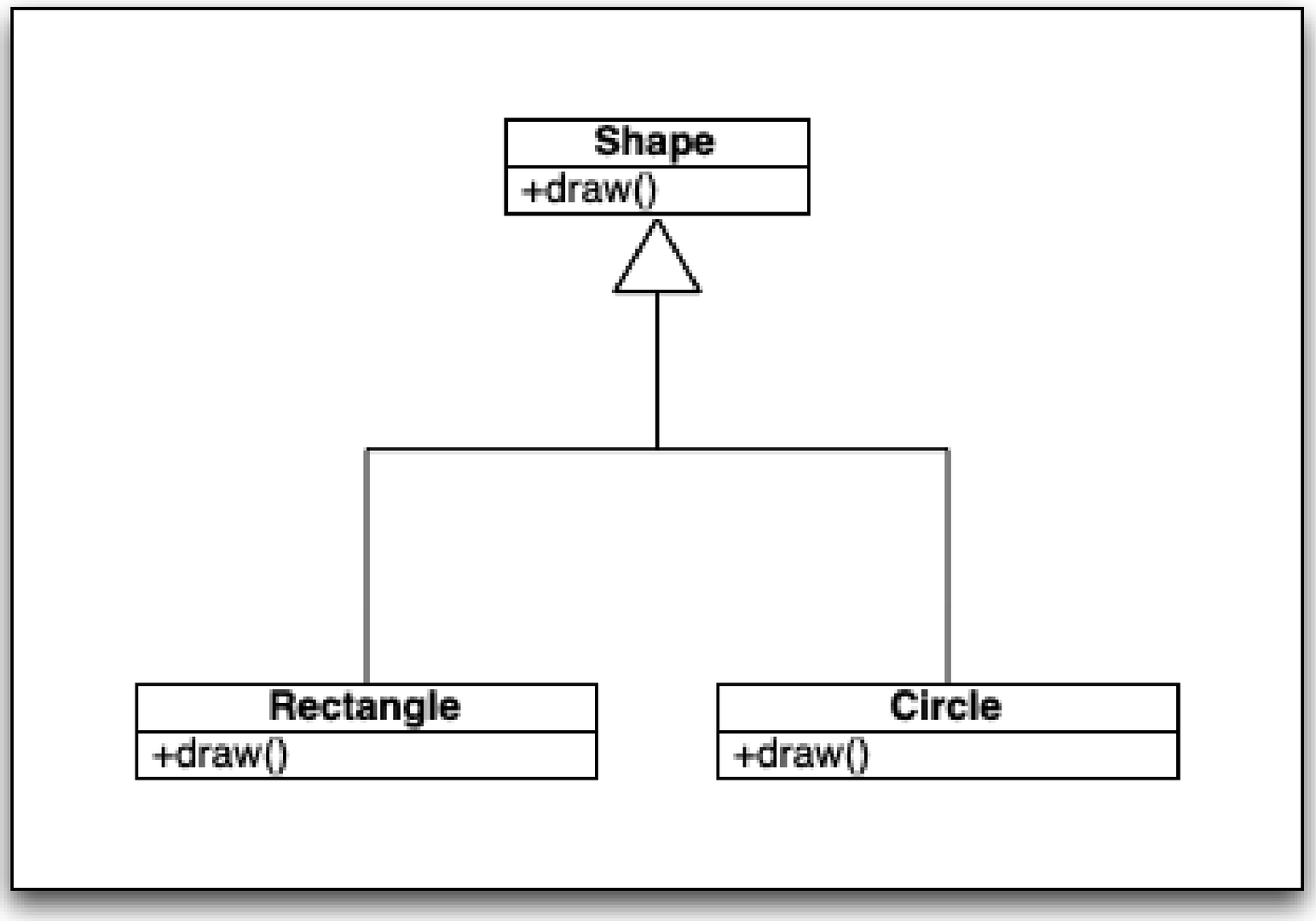
- The Shalloway/Trott book presents an example that derives the bridge pattern
  - The goal of the example is to consider
  - Variations in concept abstractions
  - Variations in implementing the concepts
- Let a set of shapes draw themselves using different drawing libraries
  - Think of the libraries as items such as Monitor, Printer, OffScreenBuffer, etc.
  - Imagine a world where each of these might have slightly different methods and method signatures

Examples of Drawing Library

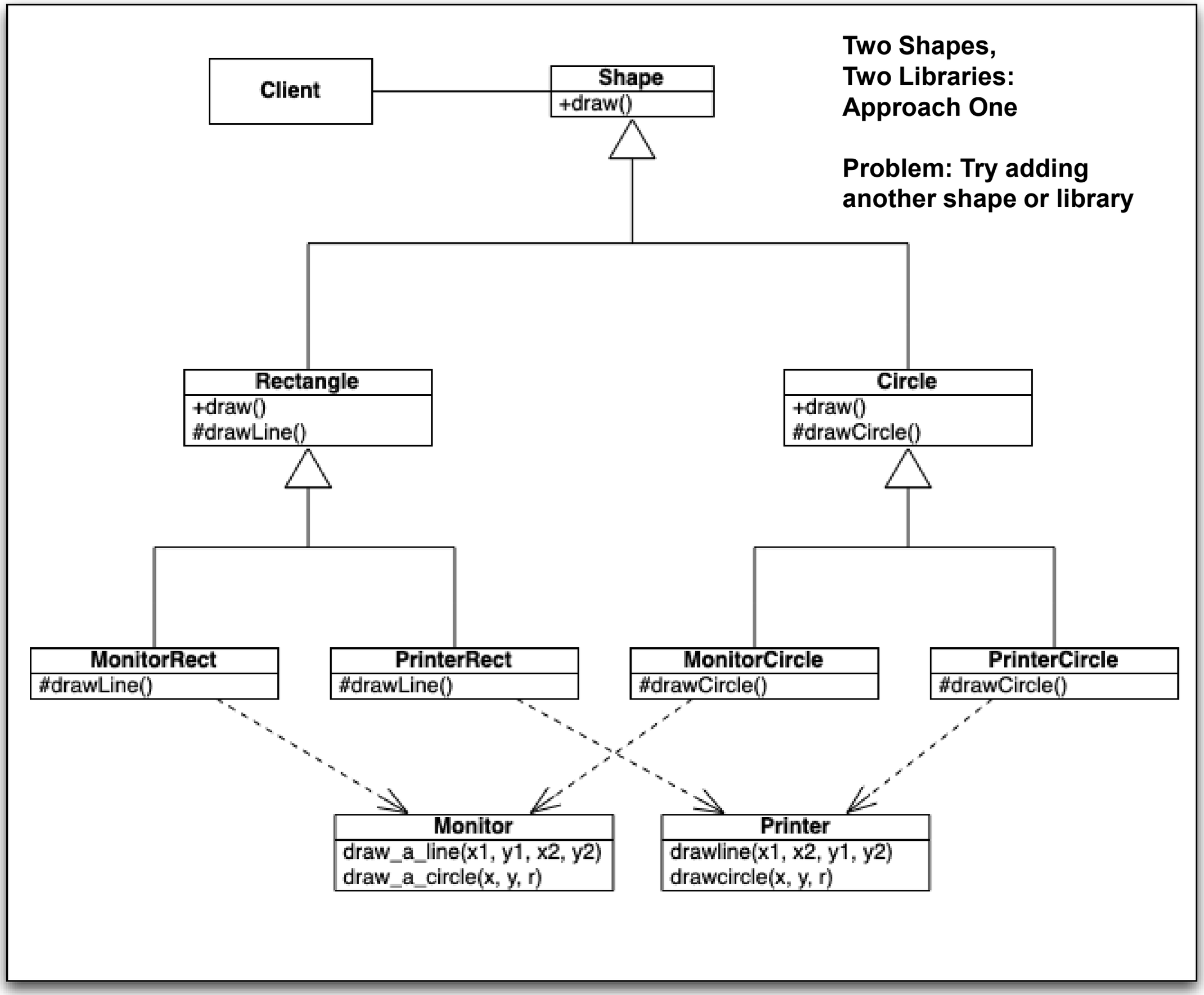
- The drawing library for Monitor has these methods
  - draw_a_line(x1, y1, x2, y2)
  - draw_a_circle(x, y, r)
- The drawing library for Printer has these methods
  - drawline(x1, x2, y1, y2)
  - drawcircle(x, y, r)



Examples of Shape



We want to be able to create collections of rectangles and circles and then tell the collection to draw itself and have it work regardless of the medium



Two Shapes,
Two Libraries:
Approach One

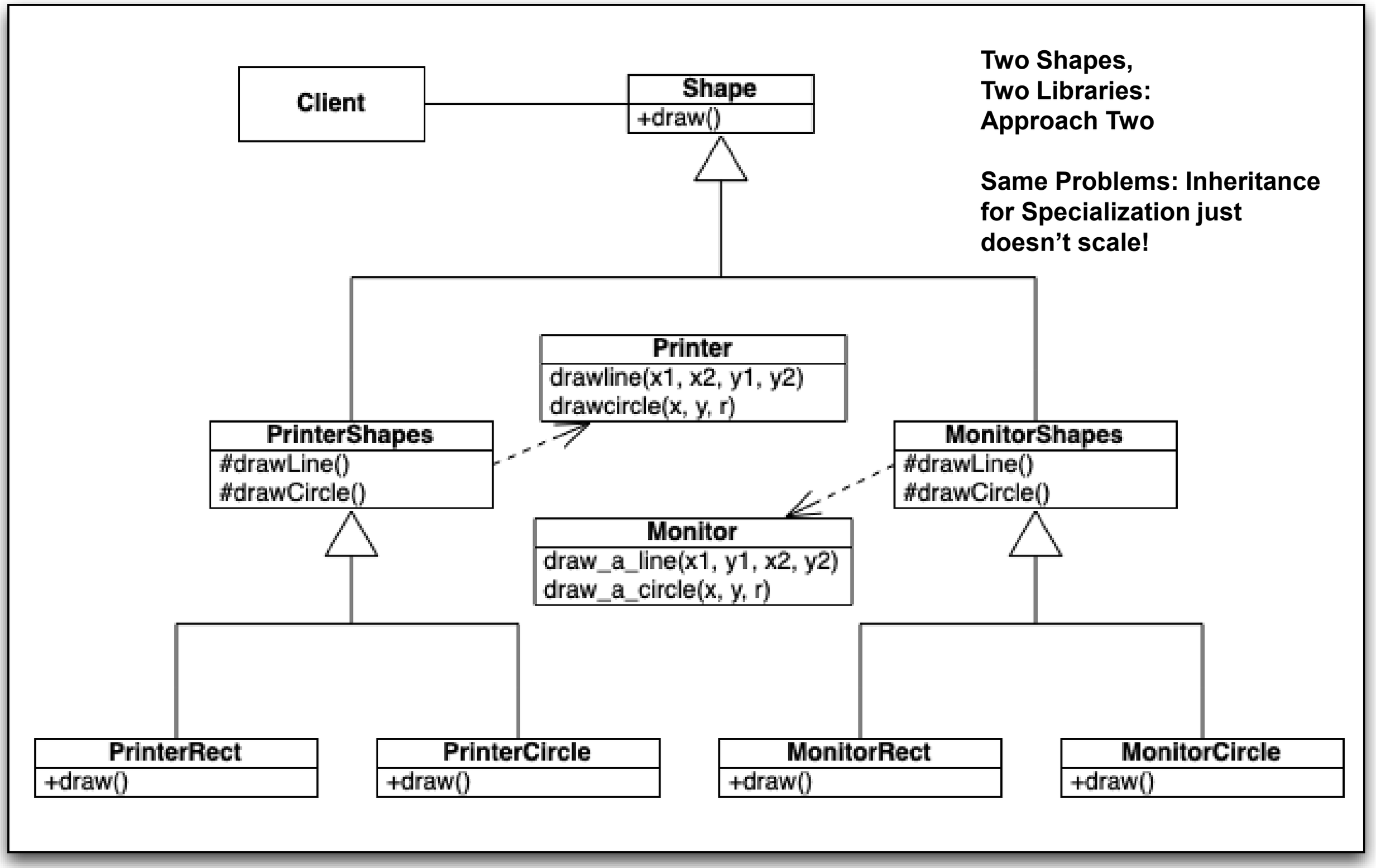
Problem: Try adding another shape or library

Emphasis of Problem (I)

- We are using inheritance to specialize for implementation
  - And, surprise, we encounter the combinatorial subclass program once again
  - A “class explosion”
  - 2 shapes, 2 libraries: 4 subclasses
  - 3 shapes, 3 libraries: 9 subclasses
  - 100 shapes, 10 libraries: 1000 subclasses
- Use inheritance for behavior, not specialization

Emphasis of Problem (II)

- Is there redundancy (duplication) in this design?
  - Yes, each subclass method is VERY similar
- Tight Coupling
  - You bet… each subclass highly dependent on the drawing libraries
    - change a library, change a lot of subclasses
  - Strong Cohesion? Not completely, shapes need to know about their drawing libraries; no single location for drawing
  - Would you want to have to maintain this code?



Two Shapes,
Two Libraries:
Approach Two

Same Problems: Inheritance for Specialization just doesn't scale!

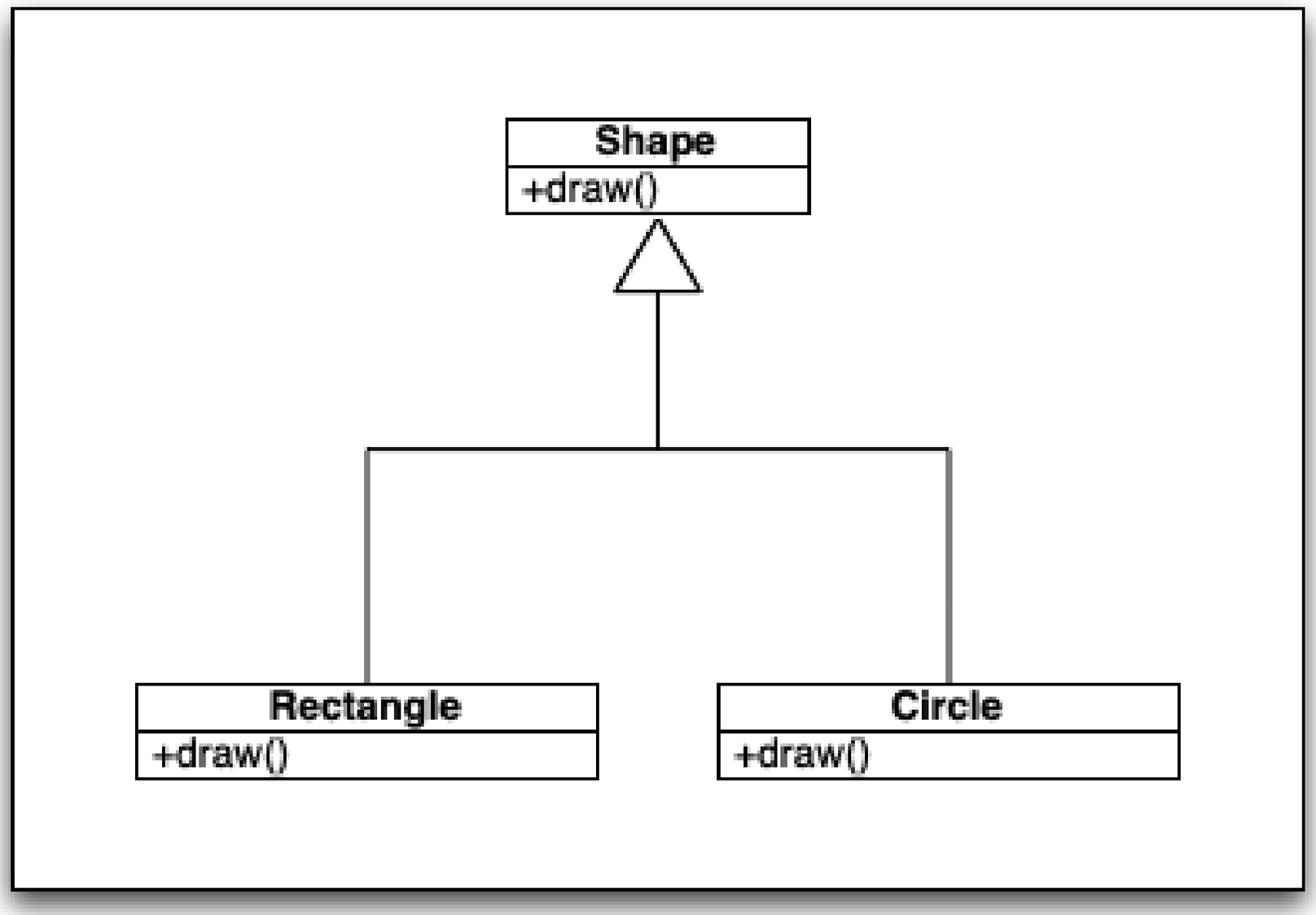
Finding a Solution

- The example applies two strategies to find the right solution
  - Find what varies and encapsulate it
  - Favor delegation (aggregation) over inheritance
  - Recognize those?
- What varies?
  - Shapes and Drawing Libraries
- We've seen two approaches to using inheritance
  - But neither worked, let's try delegation instead

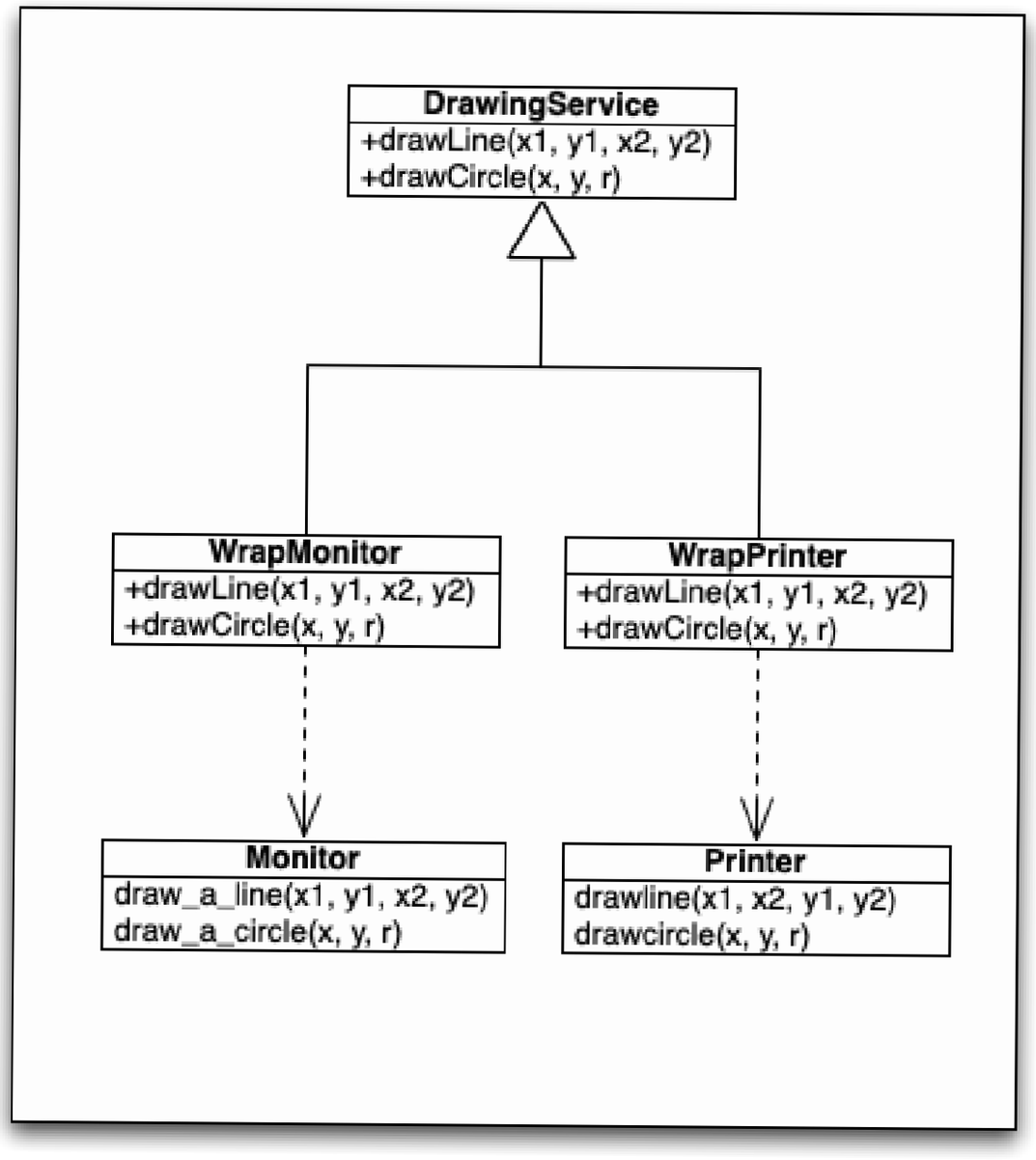
An aside: Using Design Patterns

- With patterns, designers often focus on the solutions the patterns present
- This is starting at the wrong end
- First, understand the problem
  - Looking for where to apply a pattern tells you what to do, but not when or why
- Try to focus instead on the context of a pattern, that is, what problem is the pattern trying to solve?
  - Understand the when and why I would use this
- For the Bridge, it's useful for abstractions that have different implementations, and it allows the abstractions and implementations to vary independently…

What varies? Shapes



What varies? Drawing Libraries

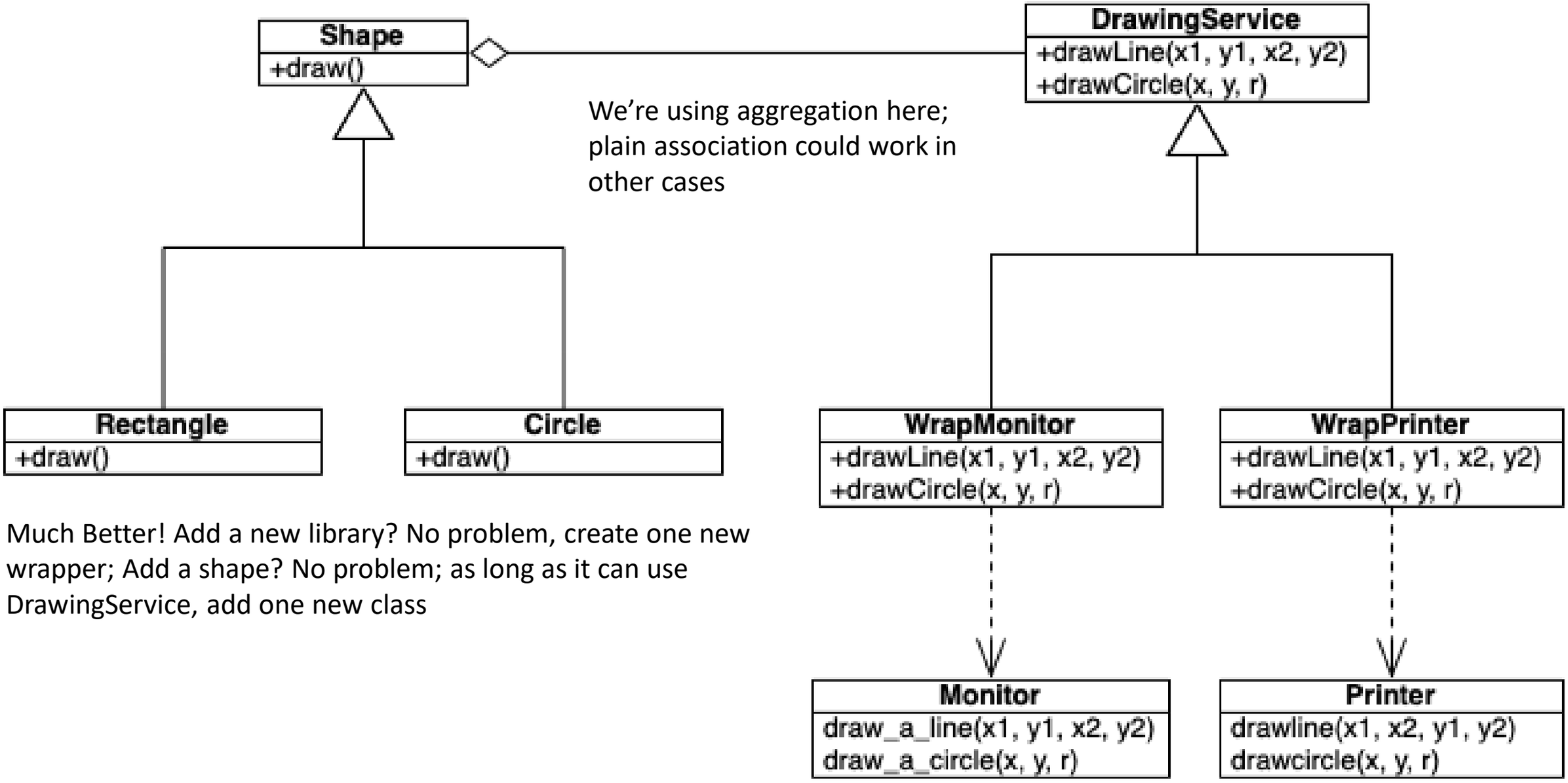


One abstract service which defines a uniform interface

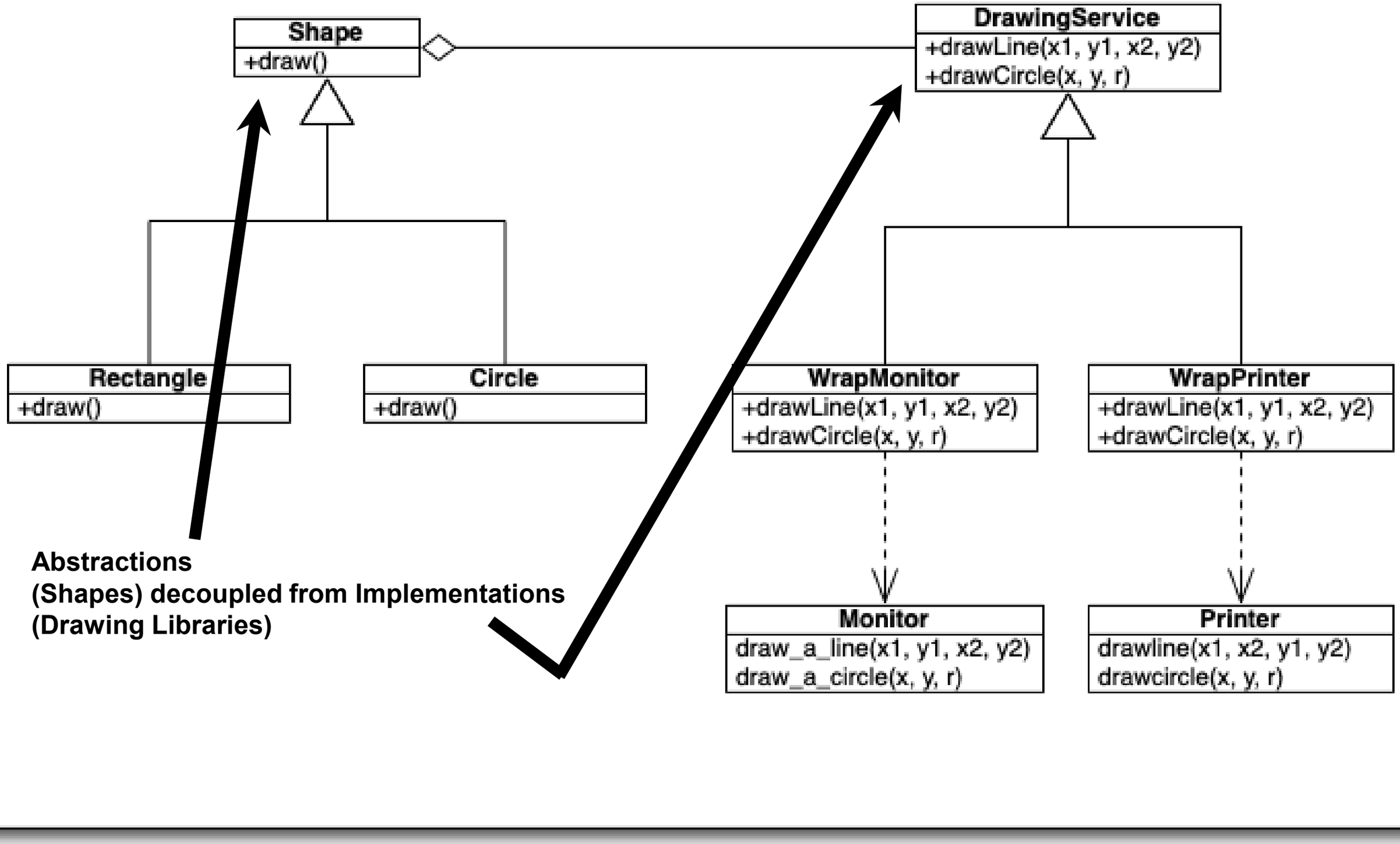
The next level down consists of classes that wrap each library behind the uniform interface

Favor delegation

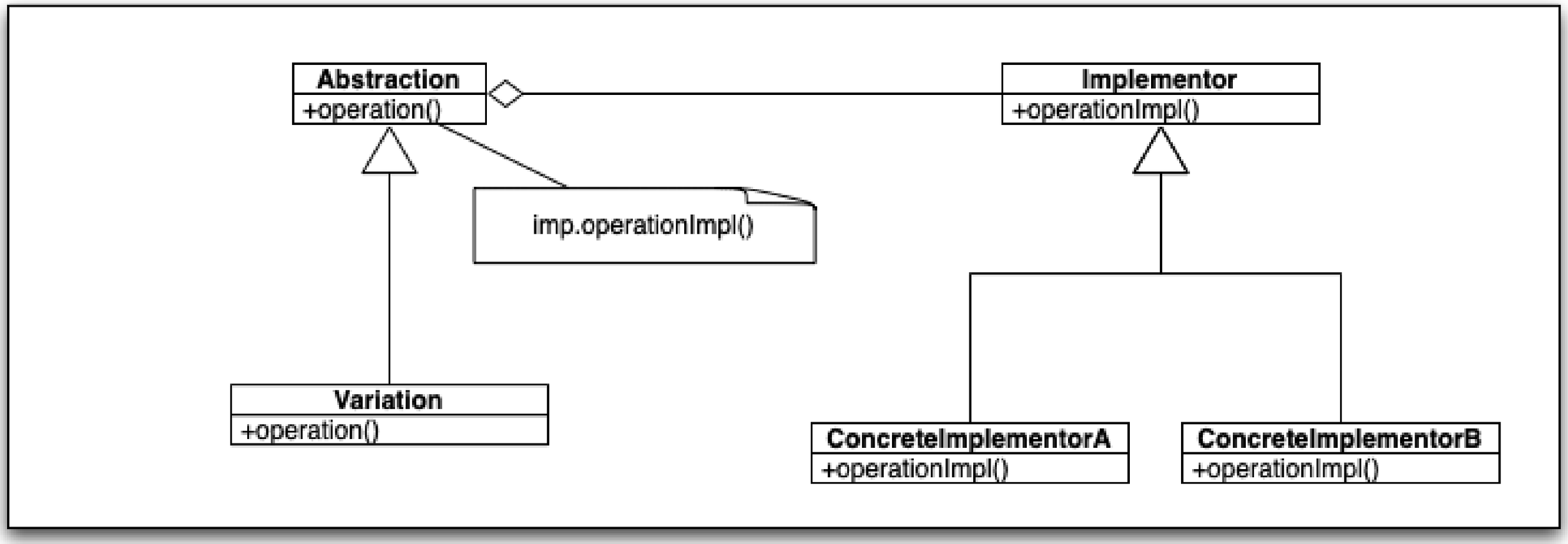
- Two choices
  - DrawingLibrary delegates to Shape
    - That doesn’t sound right
  - Shape delegates to DrawingLibrary
    - That sounds better
  - So…



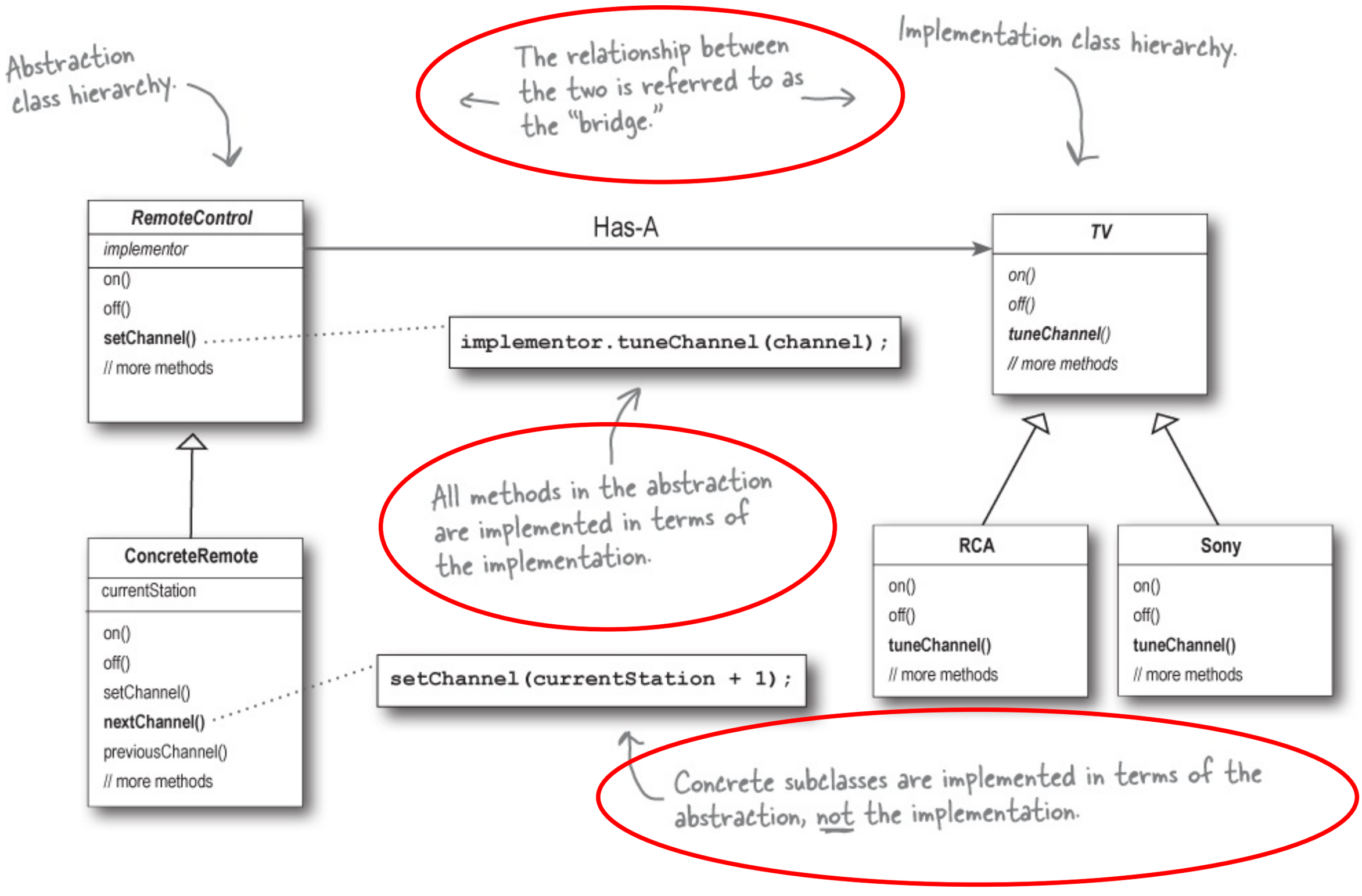
Much Better! Add a new library? No problem, create one new wrapper; Add a shape? No problem; as long as it can use DrawingService, add one new class


Shape
+draw()
DrawingService
+drawLine(x1, y1, x2, y2)
+drawCircle(x, y, r)
Rectangle
+draw()
Circle
+draw()
WrapMonitor
+drawLine(x1, y1, x2, y2)
+drawCircle(x, y, r)
WrapPrinter
+drawLine(x1, y1, x2, y2)
+drawCircle(x, y, r)
Monitor
draw_a_line(x1, y1, x2, y2)
draw_a_circle(x, y, r)
Printer
drawline(x1, x2, y1, y2)
drawcircle(x, y, r)
Abstractions
(Shapes) decoupled from Implementations
(Drawing Libraries)

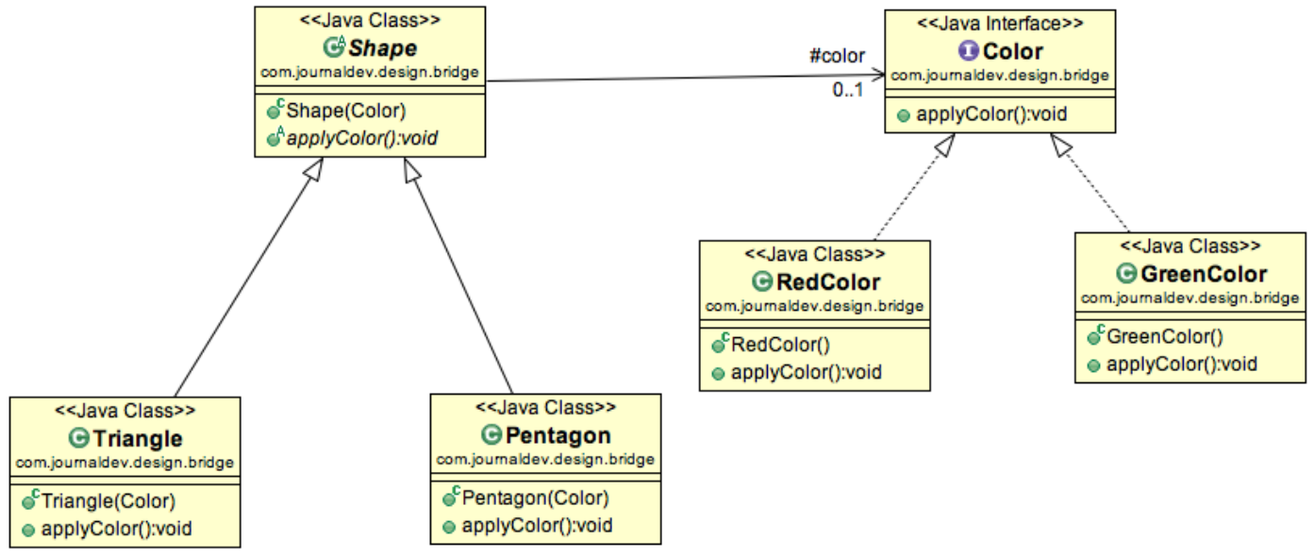
The Structure of the Bridge Pattern



Bridge example from Head First



Java Code for Bridge




```
public interface Color {
        public void applyColor();
}
public abstract class Shape {
        protected Color color;                          //composition – implementor

        public Shape(Color c) { this.color=c; }         //constructor with implementor
        abstract public void applyColor();
}
public class Triangle extends Shape{
        public Triangle(Color c) { super(c); }
        @Override
        public void applyColor() {
                System.out.print("Triangle filled with color ");
                color.applyColor();     }
}
public class RedColor implements Color{
        public void applyColor(){
                System.out.println("red.");     }
}
```

https://www.journaldev.com/1491/bridge-design-pattern-java

Summary of Bridge

- Decouples an implementation so that it is not bound permanently to an interface
- Abstraction and implementation can be extended independently
- Changes to the concrete abstraction classes don’t affect the client
- Useful any time you need to vary an interface and an implementation in different ways
- Can increases complexity

One Rule, One Place

- One important implementation strategy is to have only one place where a given rule is implemented
- This lines up with prior discussions about the “bad smell” of duplicate code (or logic)
- Generally, following this rule may result in code with a greater number of smaller methods
- This is a minimal cost that eliminates duplication and prevents future scaling and maintenance problems

- The Bridge design process example also highlights use of commonality and variability analysis…

Factories & Their Role in OO Design

- It is important to manage the creation of objects
  - Code that **mixes** object *creation* with the *use* of objects can become quickly non-cohesive
  - A system may have to deal with a variety of different contexts
  - With each context requiring a different set of objects
  - In design patterns, the context determines which concrete implementations need to be present
- The code to determine the current context, and thus which objects to instantiate, can become complex
  - with many different conditional statements

Factories & Their Role in OO Design

- If you mix this type of code with the **use** of the instantiated objects, your code becomes cluttered with responsibilities
  - often the use scenarios can happen in a **few lines of code**
  - if combined with creational code, the operational code **gets buried** behind the creational code
- Similar issues arise when mixing user interface creation and management with underlying business logic or data handling
  - One of the reasons for MVC, for instance

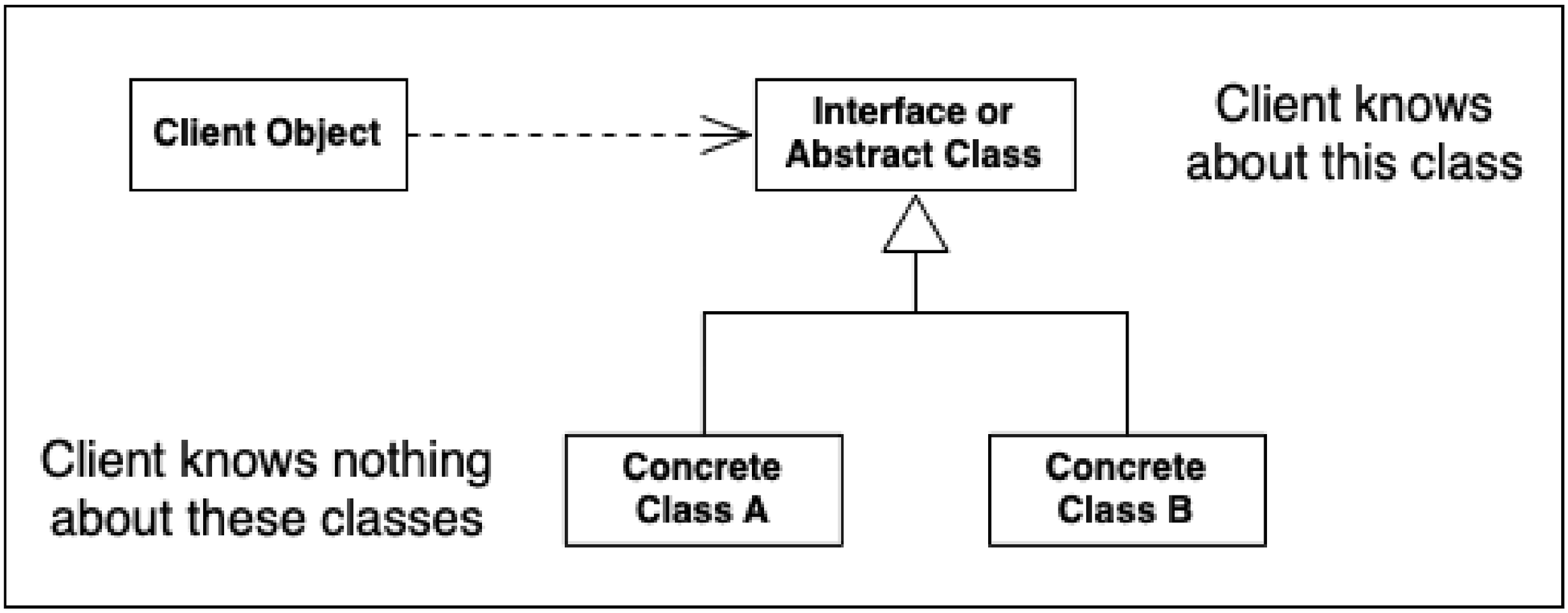
Factories provide Cohesion

- The use of factories can address these issues
  - The conditional code can be hidden within them
  - pass in the parameters associated with the current context
  - and get back the objects you need for the situation
  - Then use those objects to get your work done
- Factories concern themselves just with creation, letting your code focus on other things

The Object Creation/Management Rule

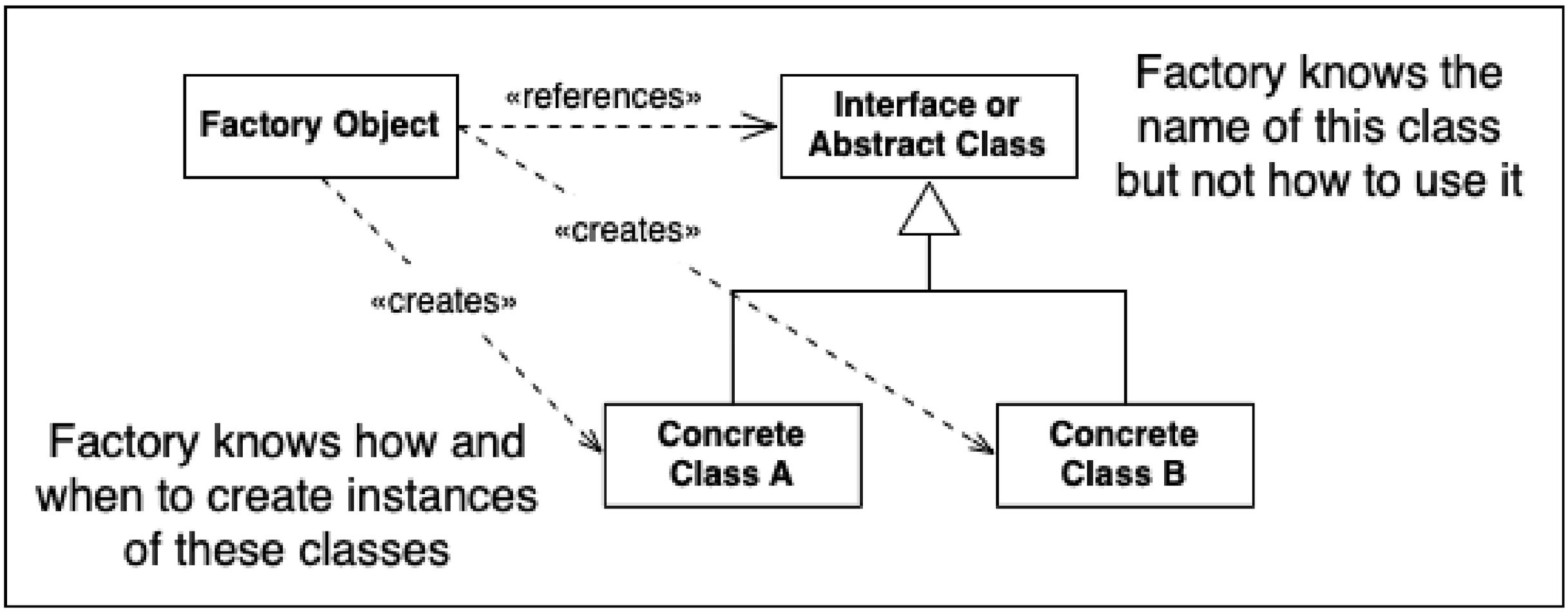
- This discussion brings us to a general design rule
  - An object should either create/manage a set of objects OR
  - It should use other objects
  - It should never do both
- Like most of the OO Principles, the Object Creation/Management rule is a **guideline** not an absolute
  - The latter is too difficult; an example is a view controllers for iOS
    - They exist to create a view and then respond to requests related to that view (which may involve making queries on the view)
    - This violates the rule, strictly speaking, as it both creates AND uses its associated view
- But as a guideline, the rule is useful
  - Look for ways to separate out the creation of objects from the code that makes use of those objects
    - encapsulate the creation process and you can change it as needed without impacting the code that then uses those objects
  - Demonstration of the advantages of the rule are in the following two diagrams

The perspective of a client object



The client is completely shielded from the concrete classes and only changes if the abstract interface changes

The perspective of a factory object



The factory knows nothing about how to use the abstract interface; it just creates the objects that implement it

Factories help to limit change

- If a change request relates to the creation of an object, the change will likely occur in a factory
  - all client code will remain unaffected
- If a change request does not relate to the creation of objects, the change will likely occur in the use of an object or the features it provides
  - your factories can be ignored as you work to implement the change

Abstract Factory and Factory Method

- We've already seen several factory pattern examples
  - **Factory Method**: Pizza and Pizza Store example
    - Have client code use an abstract method that returns a needed instance of an interface
    - Have a subclass implementation determine the concrete implementation that is returned
  - **Abstract Factory**: Pizza Ingredients Example
    - Pattern that creates groups or families of related objects

- Other Creational Patterns:
  - Singleton and Object Pool (we've already seen)
  - Builder
  - Flyweight

Builder (I)

- The Builder pattern comes from the Gang of Four book
- Its intent is
  - Separate the construction of a complex object from its representation so that the same construction process can create different representations
- Use the Builder pattern when
  - the algorithm for creating a complex object should be independent of the parts that make up the object and how they're assembled
  - the construction process must allow different representations for the object that's constructed
- Head First says
  - Use the Builder Pattern to encapsulate the construction of a product and allow it to be constructed in steps

Builder (II)

- An example of the Builder pattern is found in Android development – for Alert Dialogs
  - There are so many ways that an AlertDialog can be customized that Android offers a class called AlertDialog.Builder that makes the customization process easier

Builder (III)

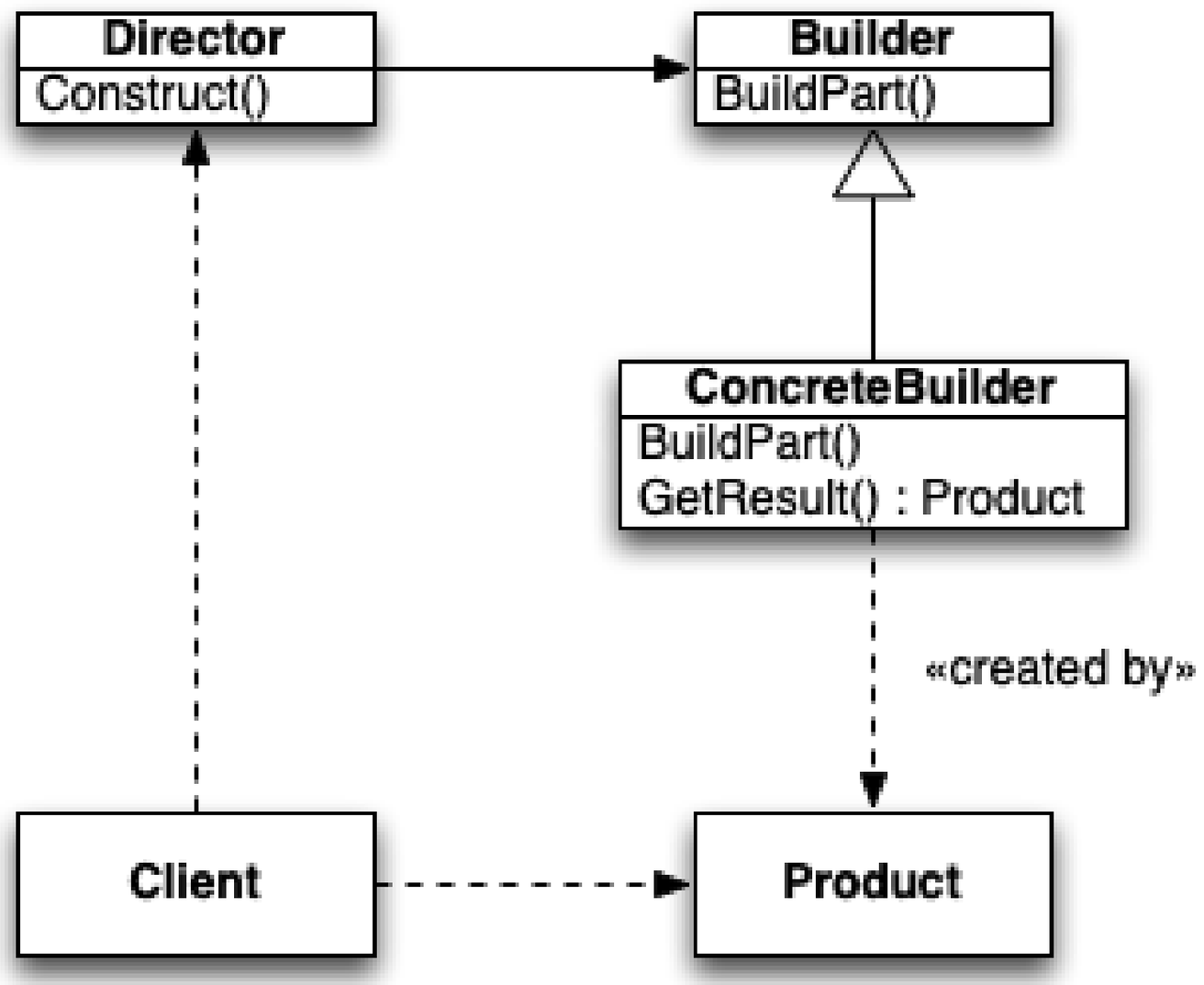
- Here's an example of using AlertDialog.Builder
  - `AlertDialog.Builder builder = new AlertDialog.Builder(this);`
  - `builder.setMessage("Do you want to cancel the download?").setTitle("Cancel Download?");`
  - `builder.setNegativeButton("No!", ...);`
  - `builder.setPositiveButton("YES!", ...);`
  - `return builder.create();`

Note the ability to chain calls to builder; each method on builder simply returns the builder object

- The **create()** method returns an instance of AlertDialog configured to match the results of the calls on the builder object
  - As you can see, this pattern can greatly simplify the creation/configuration process of complex objects

Builder (IV)

- The structure diagram for Builder identifies the following abstract roles



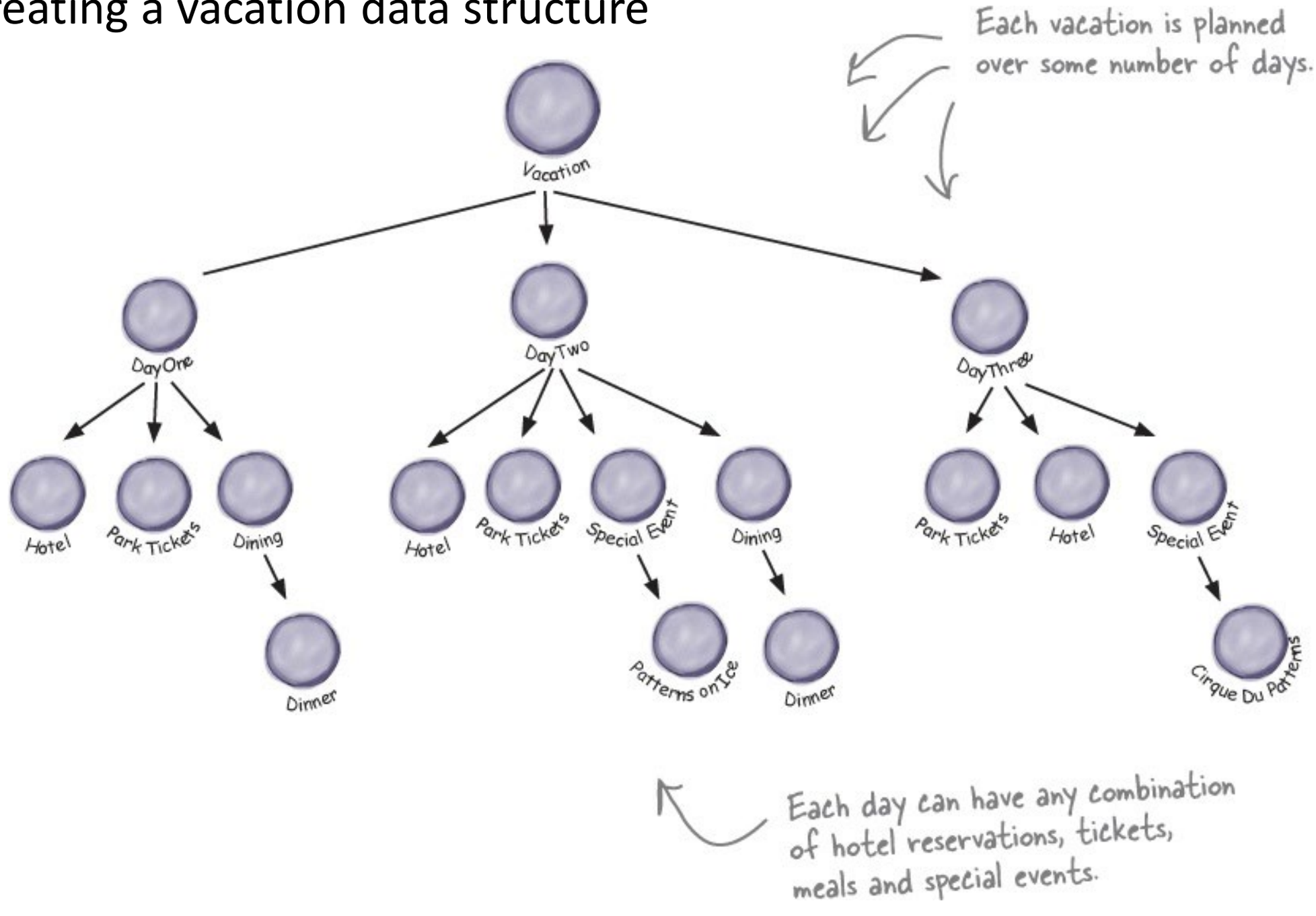
- A Director interacts with a Builder to guide the creation of a Product
- The Client creates the Director and a specific Builder and then asks the Director to create the Product
- The Client retrieves the Product directly from the ConcreteBuilder

Builder (V)

- How does this map back to the Android example?
  - AlertDialog.Builder is a ConcreteBuilder used to create/configure instances of AlertDialog (the Product)
  - Our Main activity played the roles of Client and Director
    - It created an instance of the Builder (Client responsibility)
    - It configured the Builder (Director responsibility)
    - It retrieved and used the Product (Client responsibility)
- This emphasizes something we've seen most of the semester
  - **Design Patterns outline the shape of a solution; but they can be implemented in multiple ways**

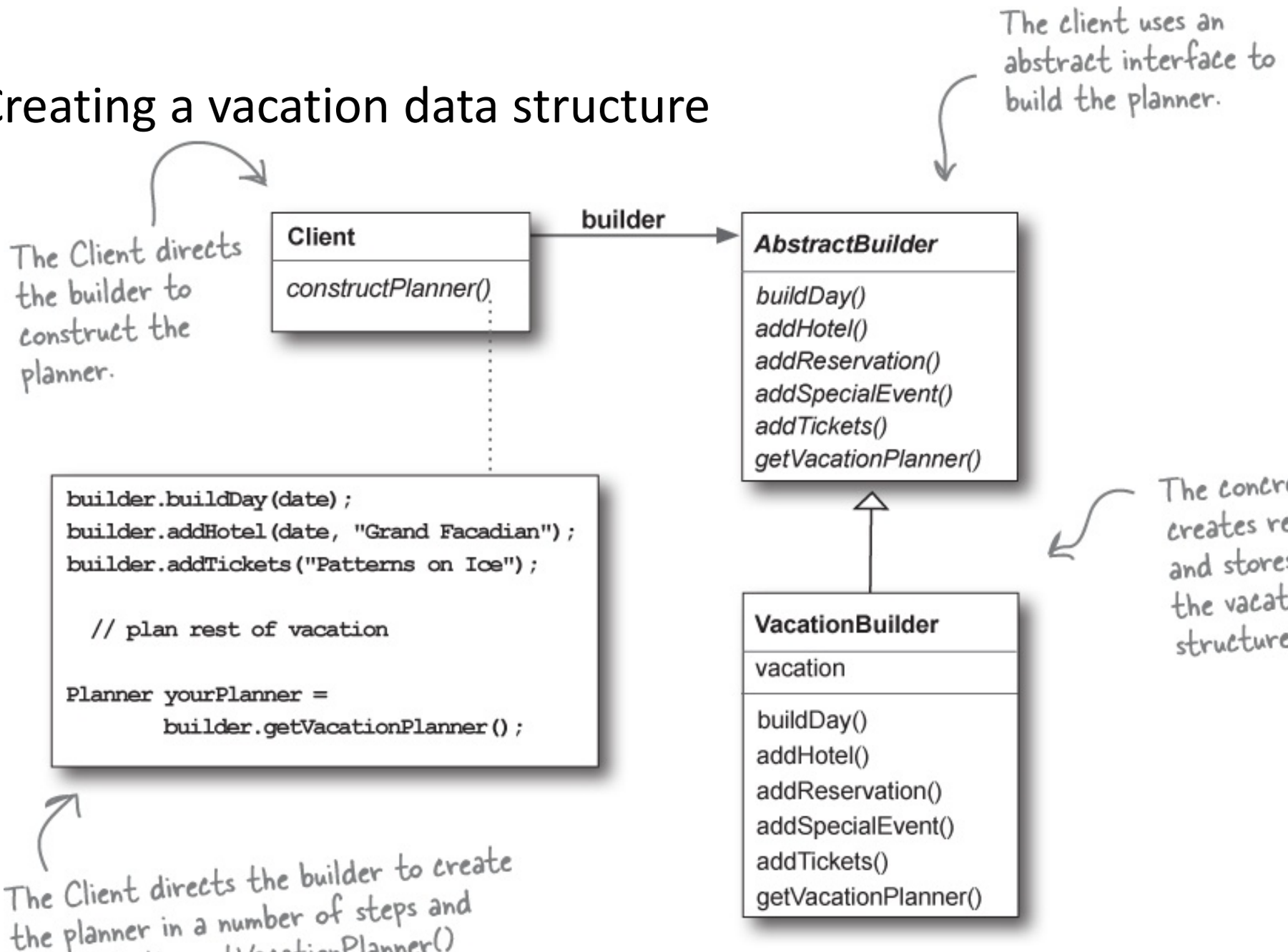
Builder in Head First

- Creating a vacation data structure



Builder in Head First

- Creating a vacation data structure



Java for Builder

Example of building out a bank account with varying fields

https://dzone.com/articles/design-patterns-the-builder-pattern

```
public class BankAccount {

   public static class Builder {

      private long accountNumber; // This is important, so we'll pass it to the constructor
      private String owner;
      private String branch;
      private double balance;
      private double interestRate;

      public Builder(long accountNumber) {
         this.accountNumber = accountNumber;
      }

      public Builder withOwner(String owner){
         this.owner = owner;
         return this;  // By returning the builder each time, we can create a fluent interface
      }

      public Builder atBranch(String branch){
         this.branch = branch;
         return this;
      }

      public Builder openingBalance(double balance){
         this.balance = balance;
         return this;
      }

      public Builder atRate(double interestRate){
         this.interestRate = interestRate;
         return this;
      }

      public BankAccount build(){
         // Here we create the actual bank account object, which is always in a
         // fully initialized state when it's returned.
         // Since the builder is in BankAccount, we can invoke its private constructor.
         BankAccount account = new BankAccount();
         account.accountNumber = this.accountNumber;
         account.owner = this.owner;
         account.branch = this.branch;
         account.balance = this.balance;
         account.interestRate = this.interestRate;
         return account;
      }
   }

   //Fields omitted for brevity.
   private BankAccount() {
      //Constructor is now private.
   }
   //Getters and setters omitted for brevity.
}
```

Java for Builder: Using It

```
BankAccount account = new BankAccount.Builder(1234L)
        .withOwner("Marge")
        .atBranch("Springfield")
        .openingBalance(100)
        .atRate(2.5)
        .build();
BankAccount anotherAccount = new BankAccount.Builder(4567L)
        .withOwner("Homer")
        .atBranch("Springfield")
        .openingBalance(100)
        .atRate(2.5)
        .build();
```

Python example at:
https://refactoring.guru/design-patterns/builder/python/example

Summary

- We learned about Bridge and saw how it allows a set of abstractions to make use of multiple implementations in a scalable way
- We used OO Principles as well as Commonality and Variability Analysis
  - Find what varies and encapsulate it
  - Favor delegation (aggregation) over inheritance
- We reviewed the Object Creation/Management Rule
- Looked at Builder for creating complex objects with varying representations