Expanding Our Horizons

CSCI 4448/5448: Object-Oriented Analysis & Design

Lecture 8 — 02/03/2011

Goals of the Lecture

- Cover the material in Chapter 8 of our textbook
  - New perspective on objects and encapsulation
  - How to handle variation in behavior
  - New perspective on inheritance
  - Commonality and Variability Analysis
  - Relationship between Design Patterns and Agile

Traditional View of Objects

- "Data with Methods" or "Smart Data"
  - Based on the mechanics of OO languages
    - In C, you have structs (data) and then you have functions that operate on the structs (methods)
    - In C++, you could combine the two into a single unit… hence "data with methods"
- But this view limits your ability to design with objects
  - The focus is mainly on the data not the behavior!

Example

```
 1 public class Pixel {
 2
 3   private double red;
 4   private double green;
 5   private double blue;
 6   private double alpha;
 7
 8   public Pixel(double red, double green, double blue, double alpha) {
 9     this.red = red;
10     this.green = green;
11     this.blue = blue;
12     this.alpha = alpha;
13   }
14
15   public double getRed() {
16     return red;
17   }
18
19   public void setRed(double red) {
20     this.red = red;
21   }
22
23   ...
24 }
25
```

“Dumb Data Holder”

This is a class that exists solely to help some other class. It is the worst form of “data with methods”

Part of the problem is this “concept” is too low level to be useful.

New Perspective on Objects (I)

- Objects are "Things with Responsibilities"
  - Don't focus on the data; it is subject to change as the implementation evolves to meet non-functional constraints
    - this is why we set attributes as private by default
- **focus on behavior**
  - And **how** those **behaviors** allow you to **fulfill responsibilities** that the **system must meet**

New Perspective on Objects (II)

- The responsibilities come from the requirements
  - If you have a requirement to create profiles for your users then somewhere in your design, you have
    - an object with the responsibility of creating profiles and managing the workflow related to that task
    - an object with the responsibility of storing and manipulating the data of a profile
    - an object with the responsibility of storing and manipulating multiple profiles

New Perspective on Objects (III)

- Responsibilities help you design
  - Requirements lead to responsibilities
  - And responsibilities need to "go somewhere"
- The process of analysis becomes
  - finding all of the responsibilities of the system
- The process of design becomes
  - finding a home for each responsibility (object/subsystem)

New Perspective on Objects (IV)

- A focus on responsibilities also promotes a focus on **defining the public interface** of an object
  - What methods will I need to meet my responsibilities?
  - How will I be used?
- This focus early in design **matches the external perspective we need to maintain**
  - **see the system from the user's point of view**
    - A rush to implementation obscures that perspective

Example, continued

- Pixel was too low level to be useful
  - but a collection of pixels… an image
  - Now you're talking
- With an image class you can specify useful services
  - stretch, flip, distort, change to black and white, add a shadow, produce a mirror image effect, move, display yourself on this canvas, …

Tasks

Cached Image Loading Routines

+ imageNamed:

Creating New Images

+ imageWithContentsOfFile:
+ imageWithData:
+ imageWithCGImage:
+ imageWithCGImage:scale:orientation:
– stretchableImageWithLeftCapWidth:topCapHeight:

Initializing Images

– initWithContentsOfFile:
– initWithData:
– initWithCGImage:
– initWithCGImage:scale:orientation:

Image Attributes

imageOrientation *property*
size *property*
scale *property*
CGImage *property*
leftCapWidth *property*
topCapHeight *property*

Drawing Images

– drawAtPoint:
– drawAtPoint:blendMode:alpha:
– drawInRect:
– drawInRect:blendMode:alpha:
– drawAsPatternInRect:

Example, continued.

Here's the public interface of the UIImage class in Apple's Cocoa touch library

Note that they refer to the public interface as "Tasks"

A "+" in front of a method name indicates a static method

A "-" indicates an instance method

This class is designed to be used with UIImageView to be displayed and CoreAnimation to be manipulated/animated

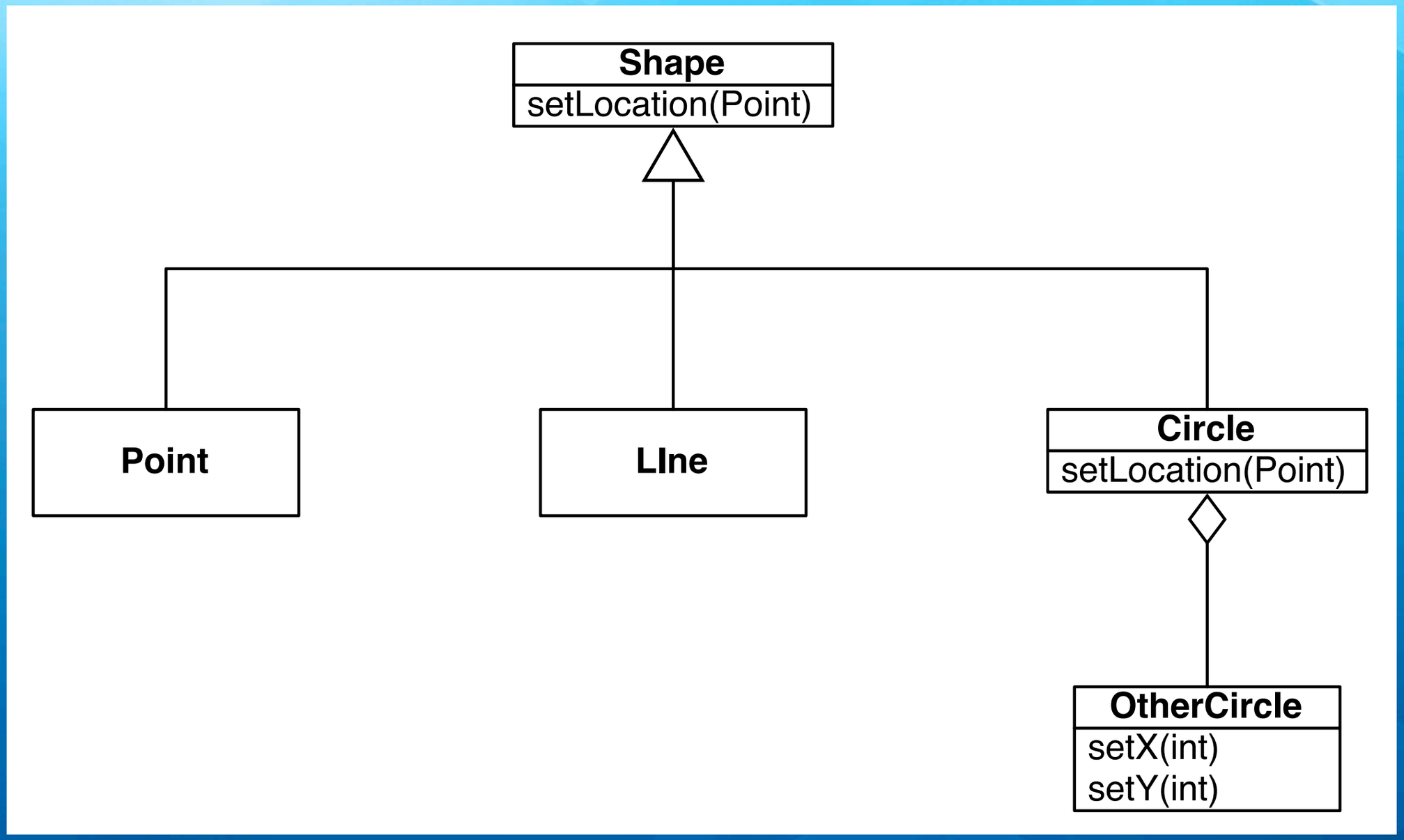
Traditional View on Encapsulation

- Encapsulation means "hiding data"
  - This view is too limited and again focuses on the data when we want to focus on behavior and responsibilities
- The Umbrella Example
  - In the analogy, the car plays the role of "encapsulation"
    - Thinking of a car as an "umbrella" is too limiting; it can do so much more! The same is true of encapsulation

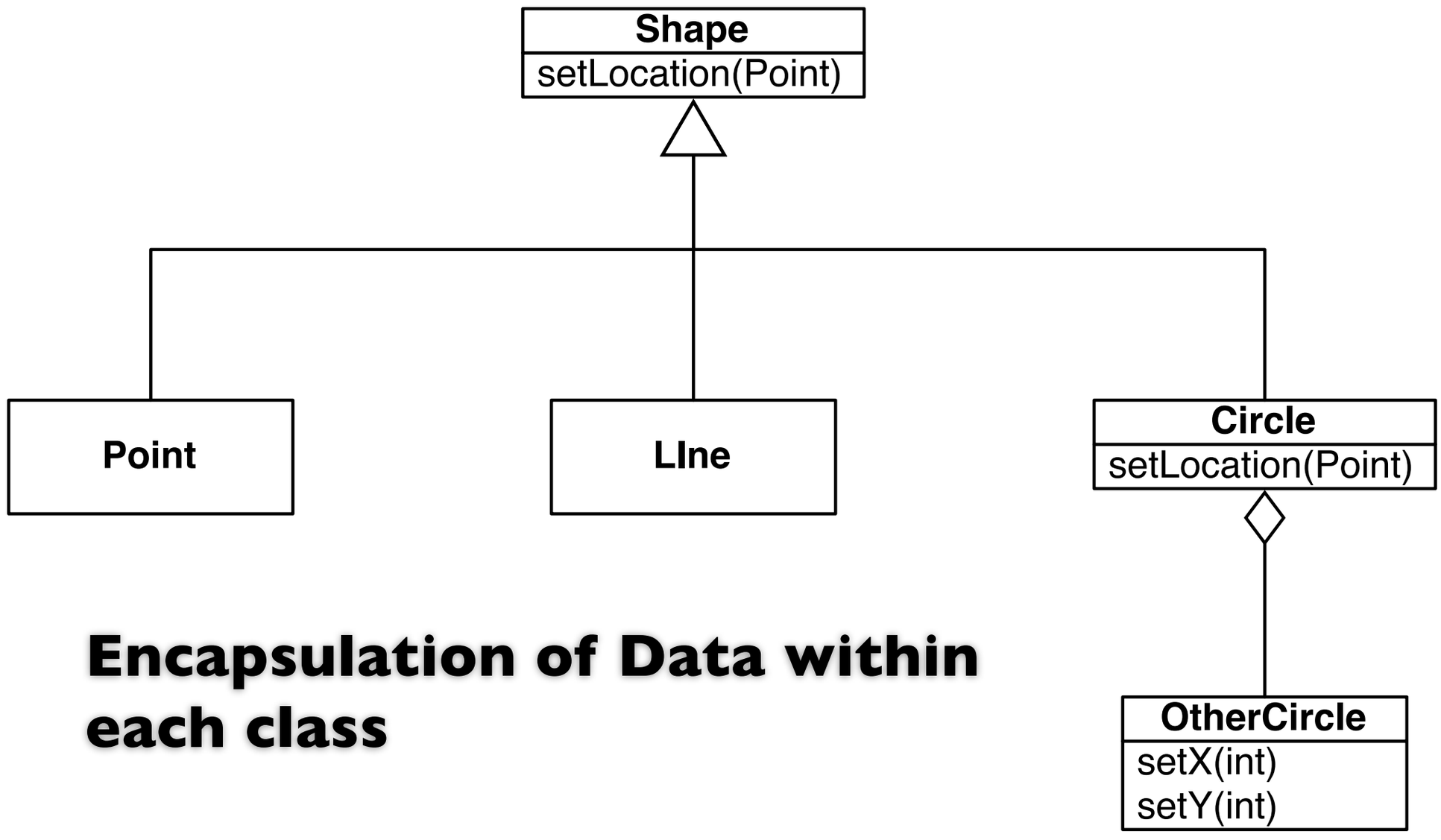
New Perspective on Encapsulation (I)

- Encapsulation should be thought of as "any kind of hiding" especially the hiding of "things that can change"
  - We certainly can hide data but also
    - behavior, implementations, design details, etc.
- and the mechanisms can involve more than just attribute and method visibility annotations
  - design patterns, subsystem boundaries, interfaces
    - for example, Objective-C's class clusters

Multiple Types of Encapsulation (I)

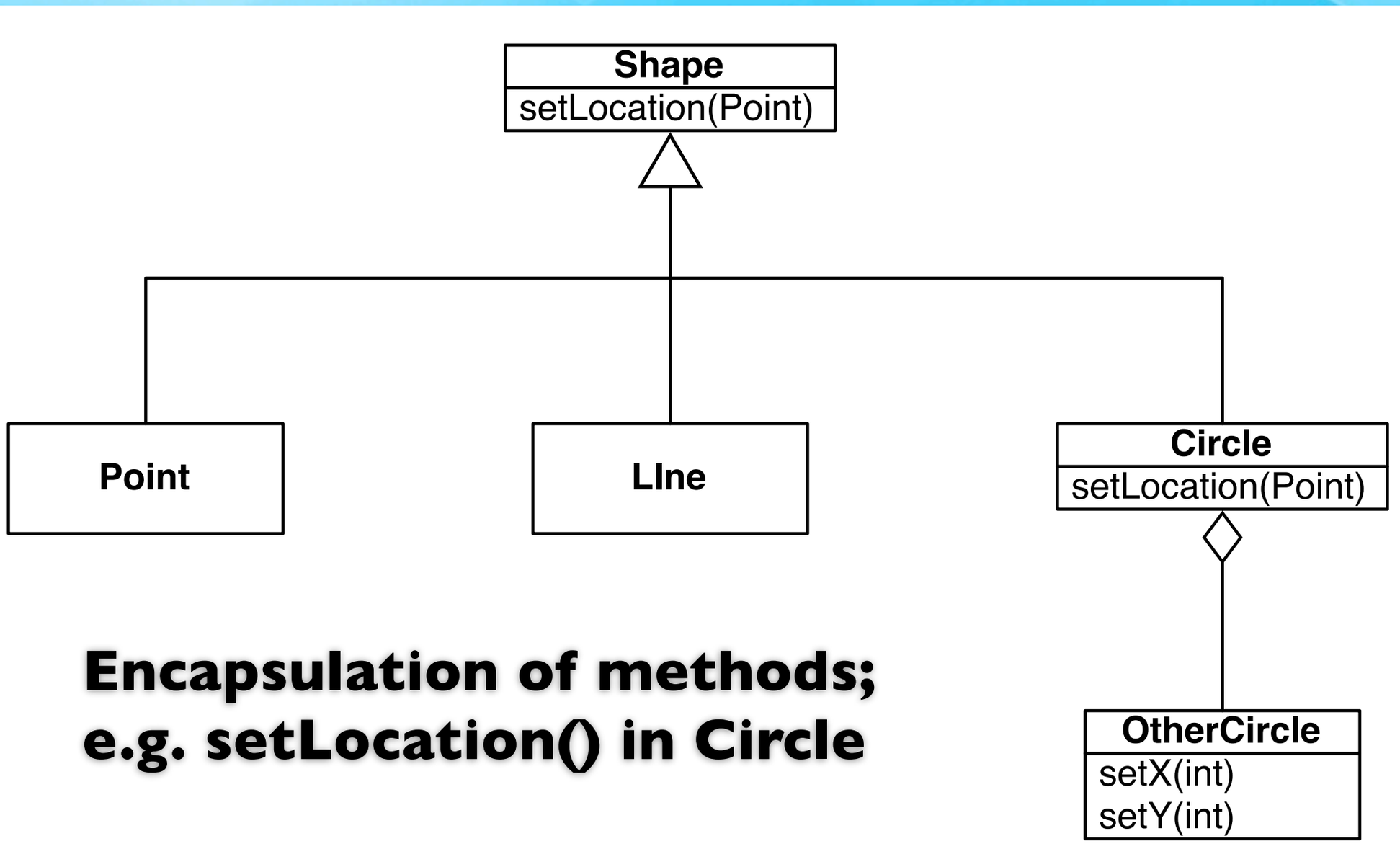


Multiple Types of Encapsulation (II)



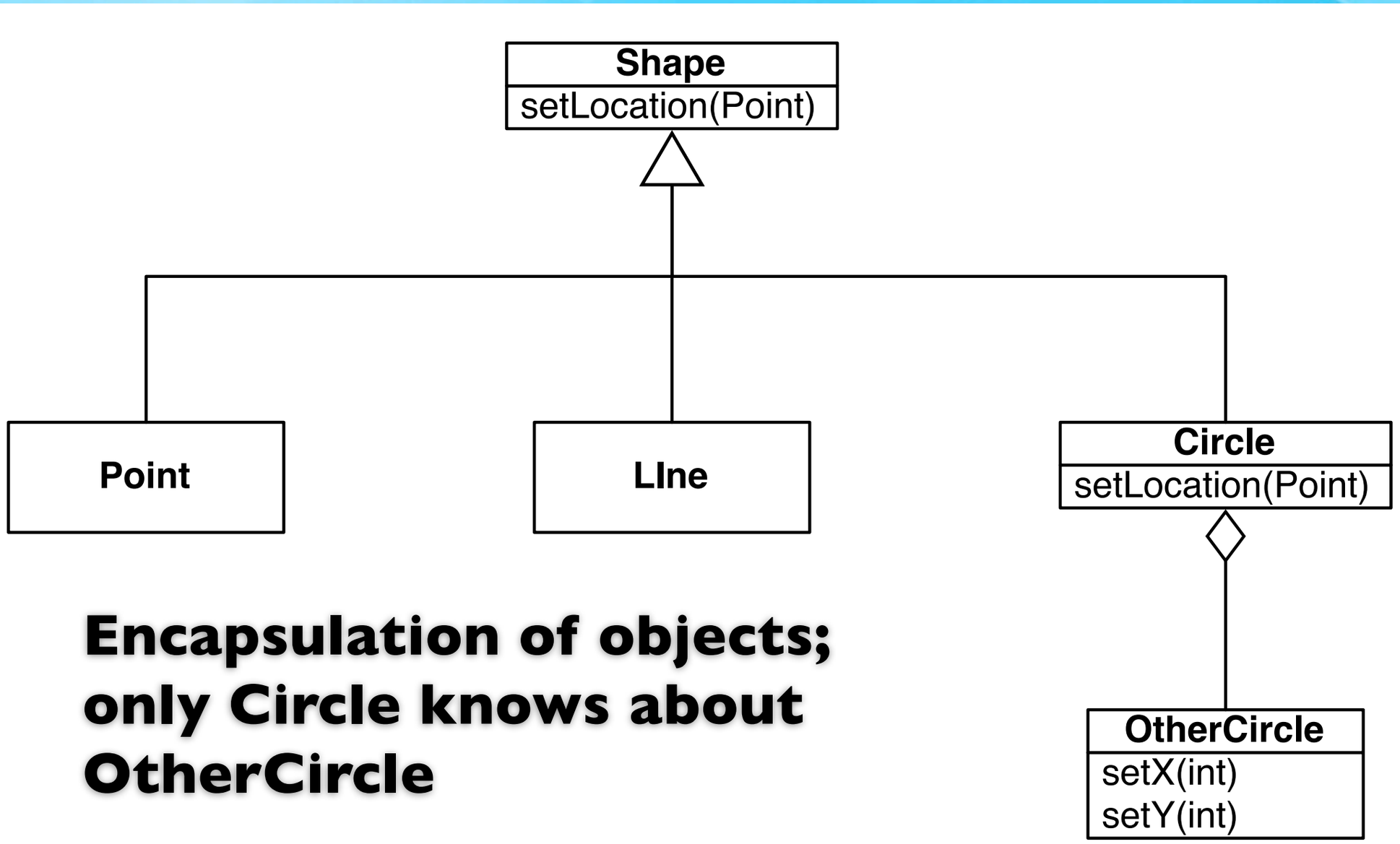
Encapsulation of Data within each class

Multiple Types of Encapsulation (III)



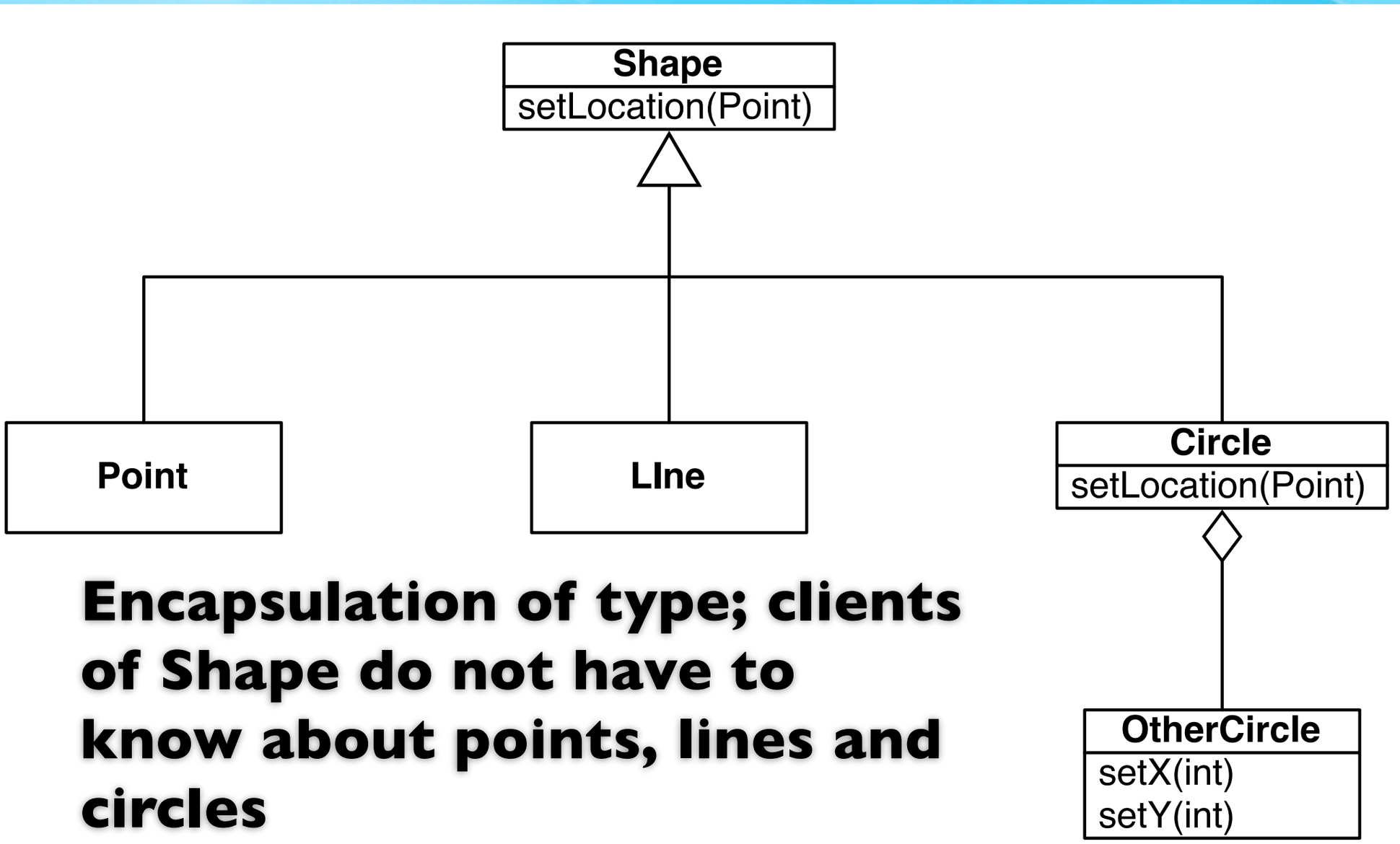
Encapsulation of methods; e.g. setLocation() in Circle

Multiple Types of Encapsulation (IV)



Encapsulation of objects; only Circle knows about OtherCircle

Multiple Types of Encapsulation (V)



Encapsulation of type; clients of Shape do not have to know about points, lines and circles

Encapsulation of Type

- Encapsulation of Type occurs
  - when there is an abstract class with derivations (subclasses) or an interface with implementations
- AND
  - the abstract class or interface is used polymorphically
- When you encounter the term "encapsulation" in design patterns, this is typically what they are referring to
- These abstract types provide the means for decomposing designs around the major services the system provides

Inheritance: Specialization vs. Behavior

- Encapsulation of type provides a new way of looking at inheritance
  - Subclasses of the abstract types are grouped because they all behave the same way (as defined by the methods of the abstract type)
- This contrasts with inheritance used to "specialize" (make more specific) an existing class
  - Pentagon → SpecialBorderPentagon

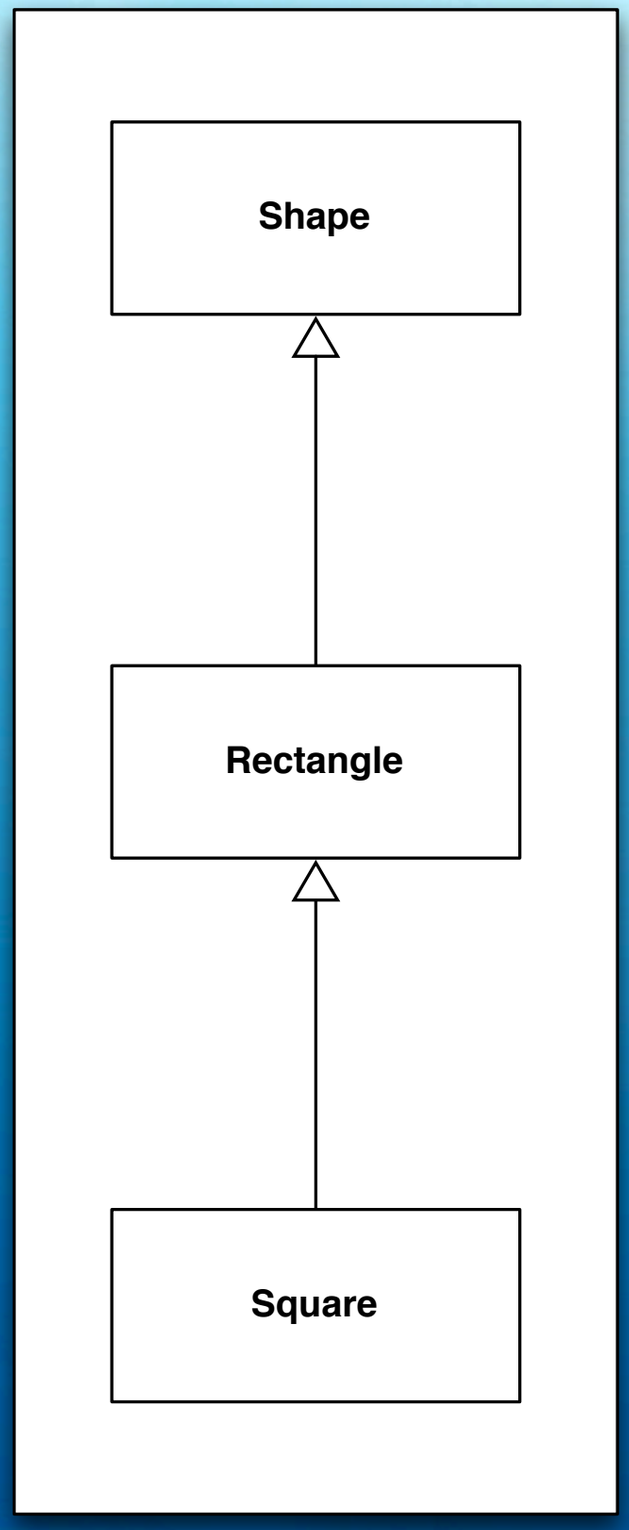
Specialization vs. Behavior (II)

- Pentagon → SpecialBorderPentagon
  - Pros
    - Reuse pentagon's behavior; enable variation with borders
  - Cons
    - **Weak Cohesion:** If I specialize again with another border, I've got classes that all deal with both pentagons and borders
    - **Poor Reuse:** How do I share my borders with Circles?
    - **Does not scale** across multiple dimensions: *SpecialBorderBlinkingSpinningPentagon* (give me a break!)

Specialization vs. Behavior (III)

- To avoid the trap of SpecialBorderBlinkingSpinningPentagon
- Encapsulate variation in behavior using the Strategy pattern we saw in Lecture 6
  - Subclasses become manageable as they are partitioned across multiple abstract types (FlyBehavior)
  - Lots of polymorphic behavior is enabled since classes like Pentagon become customizable
  - Reuse is enabled because Circle can plug these classes in as well
  - This approach scales; one new abstract type, one concrete subclass for each new behavior that varies

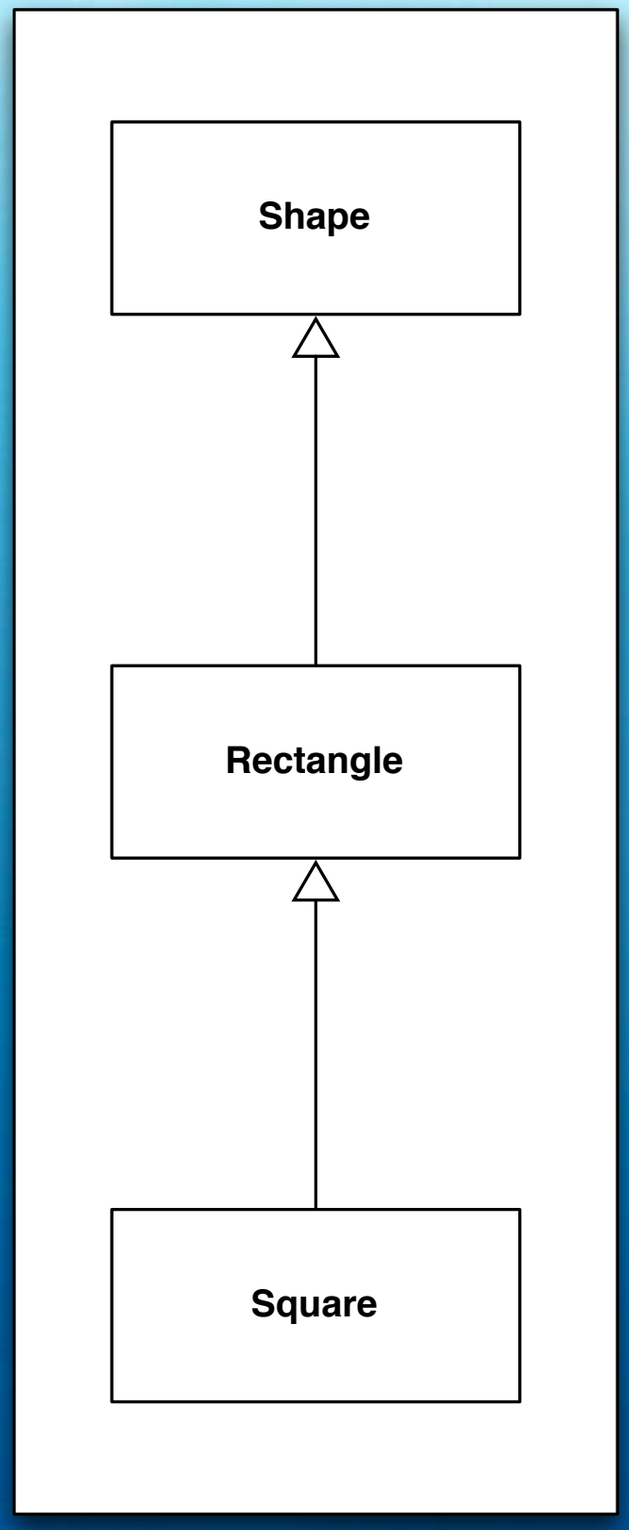
Example: Rectangle and Square



Rectangle IS-A Shape
Square IS-A Rectangle

Is there a problem with this design?

Example: Rectangle and Square

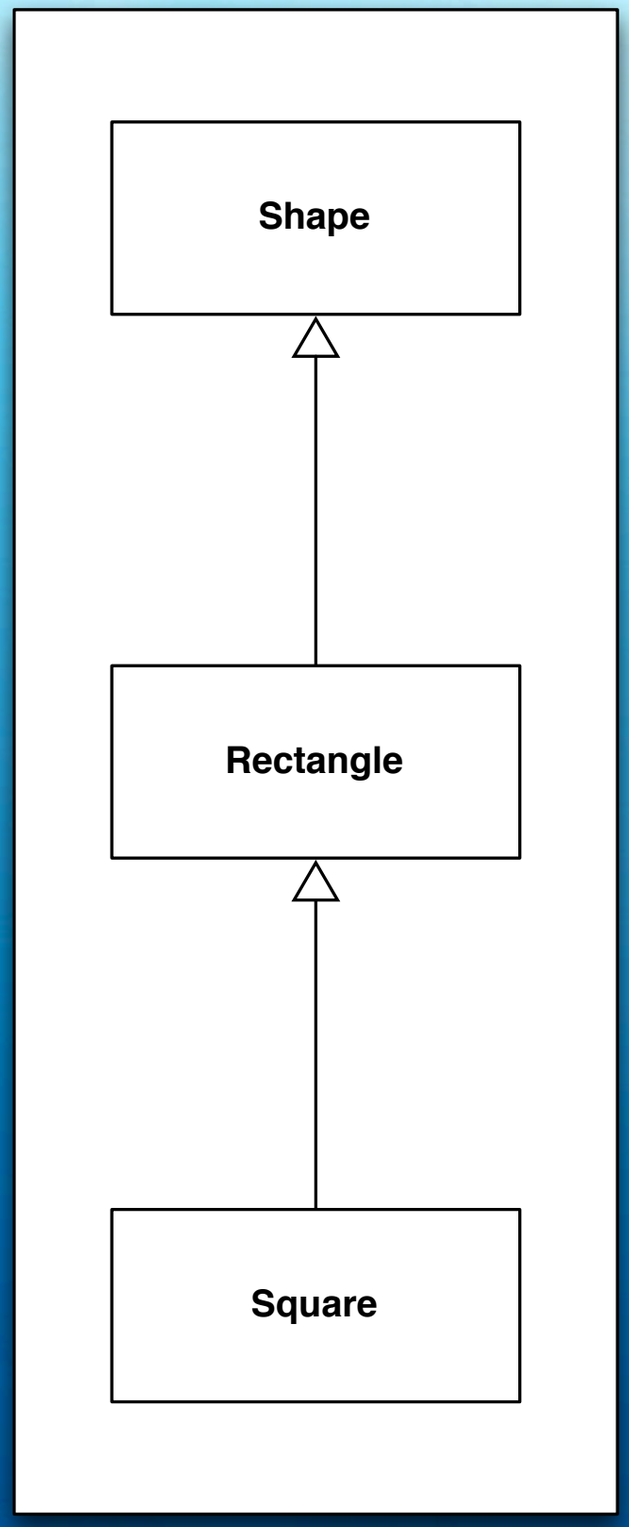


What would happen if we did something like this?

```
List<Shape> shapes = (list of squares/rects)

// set width to 5; leave length the same
for (Shape s: shapes) {
   s.setWidth(5);
}
```

Example: Rectangle and Square



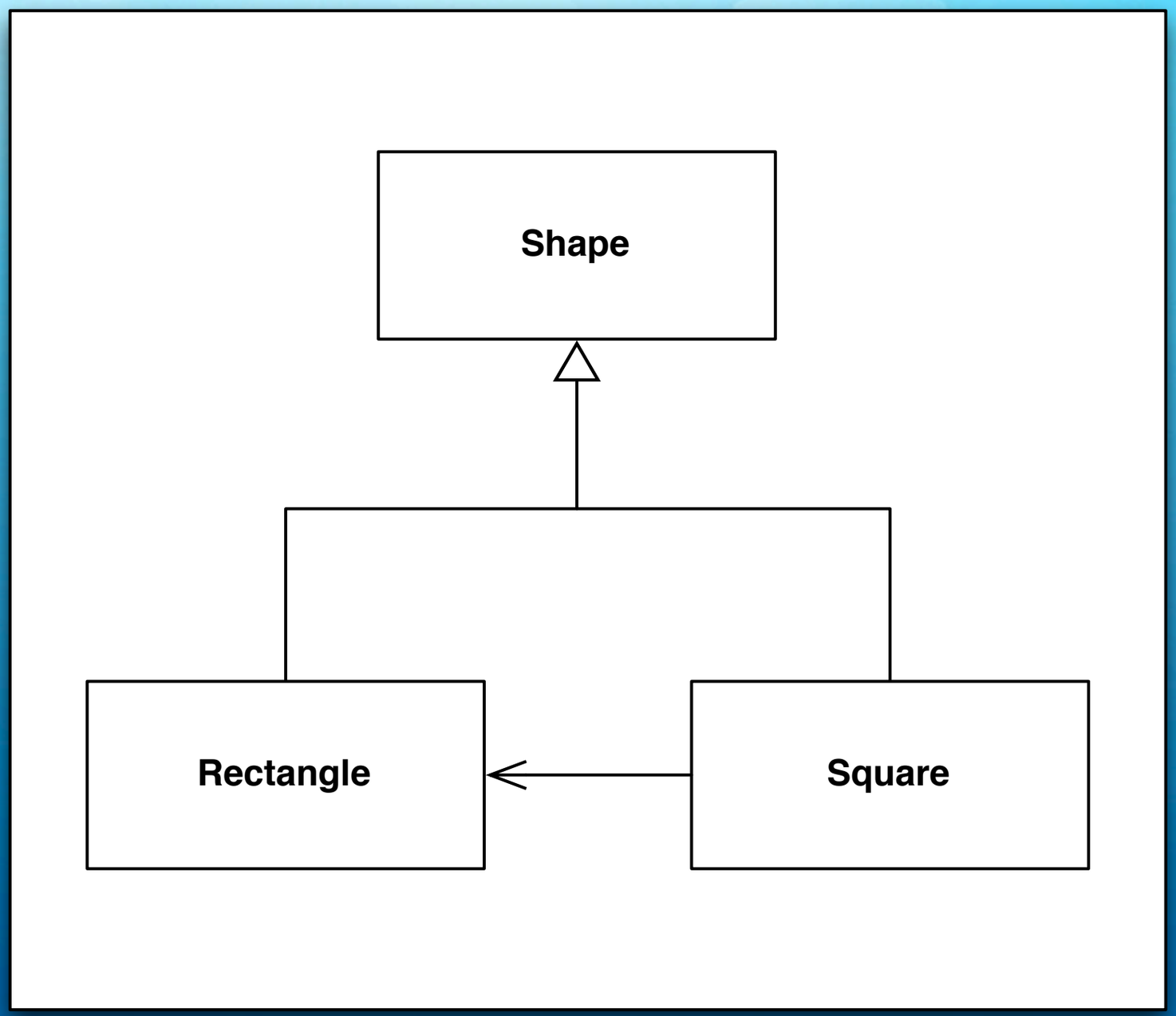
Squares share properties of rectangles but they don't BEHAVE the same

If you set a square's width, you are also setting its length

Whereas with a Rectangle, setting width and length are independent objects

Since we should use inheritance to group classes that behave the same, how should we change our design?

Example: Rectangle and Square



Since squares do not behave like Rectangles they no longer are a subclass

But since they share lots of properties, Square will keep a private copy of rectangle and delegate to rectangle when their properties or behaviors ARE the same

Differences in behavior are then handled in Square itself

Commonality and Variability Analysis

- Answers the question
  - How do we find variations in a problem domain and identify what is common across the domain
- Commonality Analysis identifies *where* things vary
  - Look at different objects and find a supertype
- Variability Analysis identifies *how* things vary
  - Look at a supertype and identify variations

Example

- Objects
  - whiteboard marker, pencil, ballpoint pen
- Commonality Analysis
  - writing instruments
- Variability Analysis
  - appearance varies, writing surface varies, "ink" varies

Commonality and Variability Analysis

- Variability only makes sense within a given commonality
  - Commonality Analysis seeks structure in a problem domain that is *unlikely to change over time*
  - Variability Analysis identifies structures that are *likely to change*
- A&D becomes locating common concepts (abstract superclasses) and their likely variations (concrete subclasses)
  - The abstract classes identify important behavior (that fulfill responsibilities) within the domain; the subclasses outline the legal variations of that behavior

Comparison to Agile Techniques

- The approach to A&D advocated by this book is often called the "design up front" approach
  - You identify the primary domain concepts relevant to solving the problem
  - You identify the users of your system;
  - You then develop a design that uses those domain concepts to allow your users to complete their tasks
    - You iterate and flesh out the design until it is ready for implementation

Agile Techniques

- Agile methods are techniques/processes for developing software systems that rely on
  - communicating with your customer frequently
  - taking small steps (functionality wise)
  - validating the small steps with the customer before moving on
- They emphasize iteration, feedback, and communication over upfront design, detailed analysis, diagrams, etc.

Opposition?

- These two techniques seem to be in opposition
  - Up front design (top-down) vs. small steps (bottom-up)
- Yet, they are both driving towards the same goal
  - systems built from effective, robust, flexible code
- They differ in approach but value the same things
  - Design patterns produce flexible code; Agile values code that can change in a straightforward manner

Opposition? Not Really

- Agile techniques value characteristics in code that are valued by the design pattern approach
  - No redundancy and Highly Cohesive Code
  - Readability and Design to an Interface
  - Testability and <all the other qualities>
- While the two techniques use different names for these characteristics, they are really talking about the same thing…

No Redundancy

- When implementing code, don't repeat yourself
  - One Rule, One Place: do not duplicate behavior
- Once and Only Once
  - The system (code+tests) must communicate everything you want to communicate (about its responsibilites)
  - The system must contain no duplicate code
- Code with no redundancy is highly cohesive and loosely coupled

Readability (I)

- Program by Intention
  - You need to implement some functionality
  - Pretend it exists, give it an intention-revealing name
    - Write the method that calls it
    - Write the method itself
- Code becomes a series of calls to functions with highly descriptive names

Readability (II)

- Martin Fowler encourages Program by Intention when he says "Whenever [you] feel the need to [write a comment], write a method instead."
  - This encourages shorter and more cohesive methods in cohesive classes
- Using intention-revealing names is very similar to "Code to an Interface". By considering how the function is to be called/used before writing it, you establish its public interface…

Testability

- Testability is key in software development
  - The more you test the more confident you are in the software being developed
- Testable code (encouraged by Agile at every turn) is
  - cohesive (doing only one thing), loosely coupled (less dependencies on a class may mean it is easier to instantiate its objects), non redundant (each rule to be tested lives in one place), readable (intention-revealing names make it easier to target test cases), encapsulated

Wrapping Up (I)

- New perspective on objects and encapsulation
  - responsibilities; hide anything
- How to handle variation in behavior
  - strategy pattern
- New perspective on inheritance
  - Group via behavior

Wrapping Up (II)

- Commonality and Variability Analysis
  - Examine problem domain for structure that are resistant to change (commonality) and then identify ways in which they can legally vary
- Relationship between Design Patterns and Agile
  - They both value the same code qualities
    - loose coupling, high cohesiveness, no redundancy, testability, readability, code to an interface, etc.

Coming Up Next

- Lecture 9: Strategy, Bridge, Abstract Factory
  - Read Chapters 9, 10, & 11
- No Homework this week!
- Lecture 10: Introduction to Java