Expanding Our Horizons

CSCI 4448/5448: Object-Oriented Analysis & Design

Lecture 20

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

Goals of the Lecture

- Material is from Chapter 8 in Shalloway/Trott
  - 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

Head First Design Patterns: OO Principles

- Encapsulate what varies
- Favor composition (delegation) over inheritance
- Program to interfaces not implementations
- Strive for loosely coupled designs between objects that interact
- Classes should be open for extension, but closed for modification
- Depend on abstractions, not concrete classes
- Only talk to your (immediate) friends
  - Law of Demeter, Principle of Least Knowledge
- Don’t call us, we’ll call you
- A class should have only one reason to change

Look for these design principles in what we review today…

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 is too simple
  - It 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 often consider attributes as private by default
  - Focus on behavior
    - And how those behaviors allow you to fulfill responsibilities that the system must meet
  - Stay at a conceptual level as long as you can before dropping down to specification and implementation

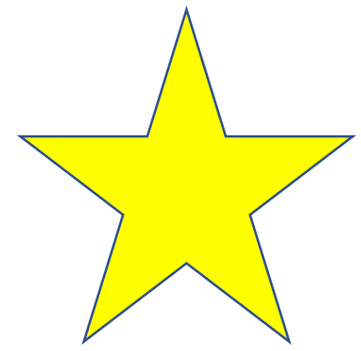
New Perspective on Objects (II)

- 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?
- Focus on motivation NOT implementation is a theme of design patterns
  - Hiding the implementation behind an interface decouples it from the object using that interface
- 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

- A pixel object may be 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

[Static method – belongs to the class, can be invoked without a class instance]

A "-" indicates an instance method

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

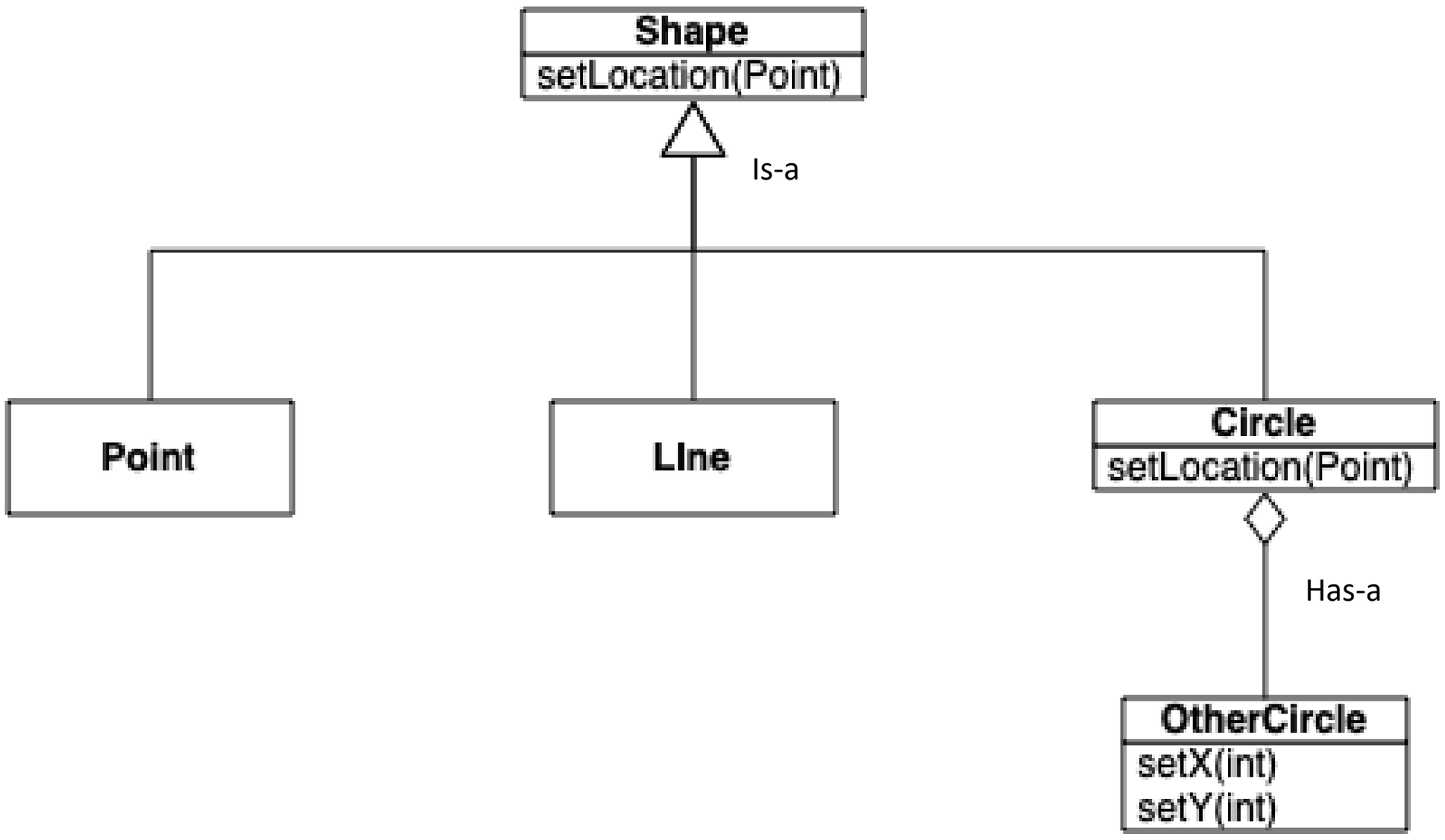
Traditional View on Encapsulation

- Traditionally, encapsulation implies “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
  - A person describes a car as an umbrella, a mobile device to keep them dry
  - 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!
      - Any definitions can be limiting
    - Encapsulation can do more too…

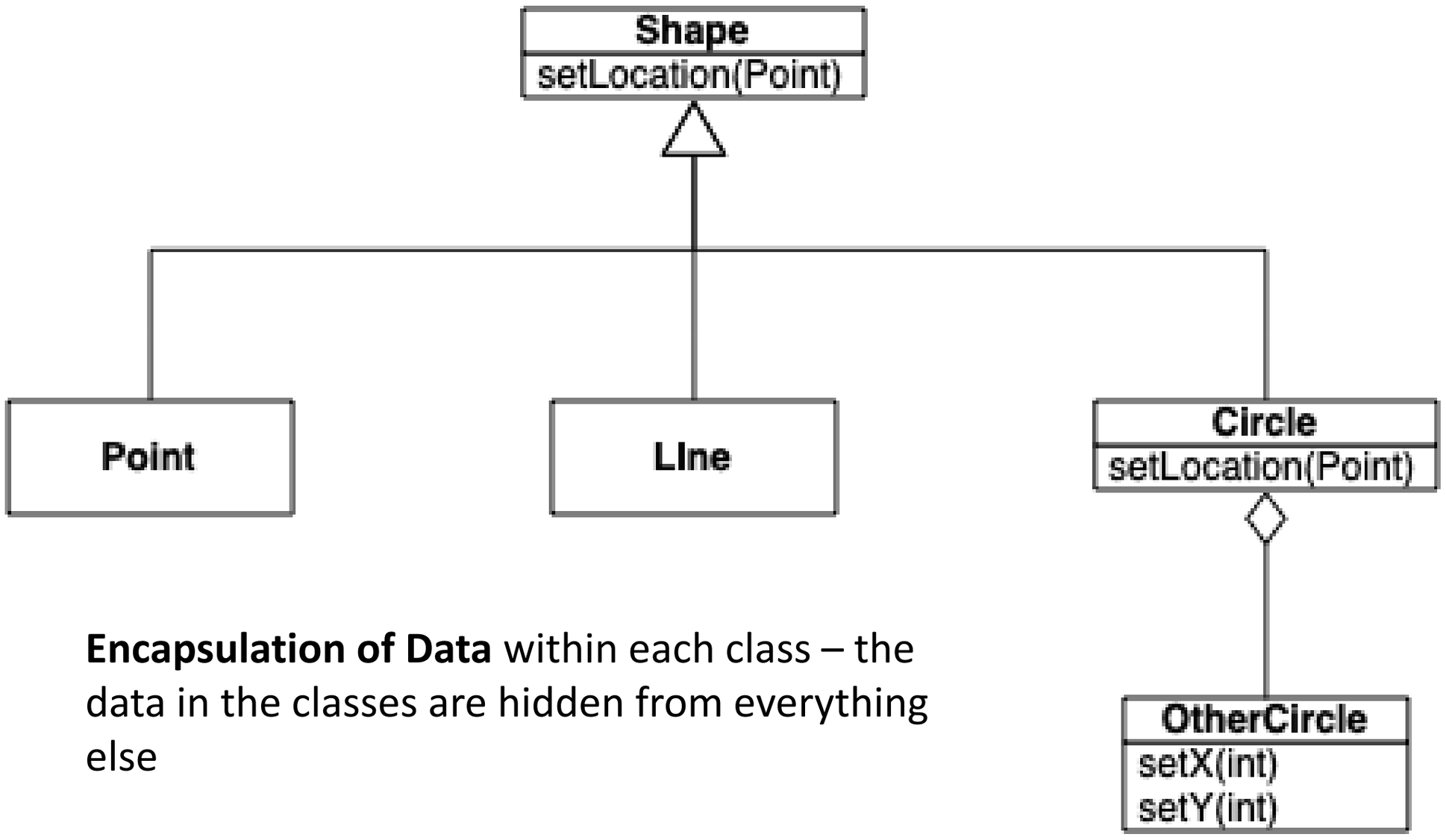
New Perspective on Encapsulation (I)

- 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
    - Derived classes
    - Instantiation rules
    - 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
    - Uses an Abstract Factory pattern to provide an abstract class that groups a set of private concrete subclasses, hiding implementation detail behind a public interface
    - Objective-C programmers may use a class cluster and not realize it… The point.

Multiple Types of Encapsulation (I)

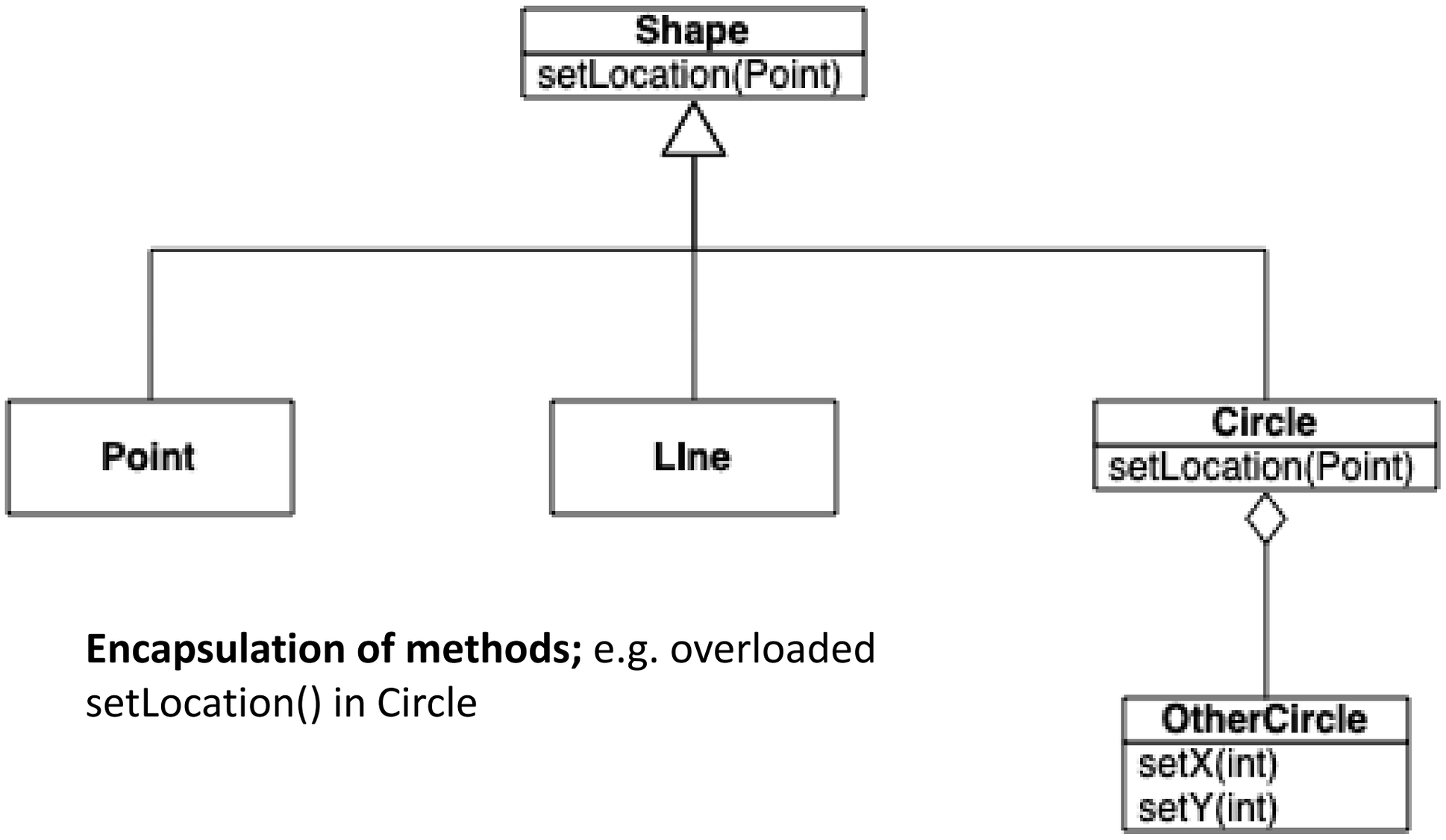


Multiple Types of Encapsulation (II)



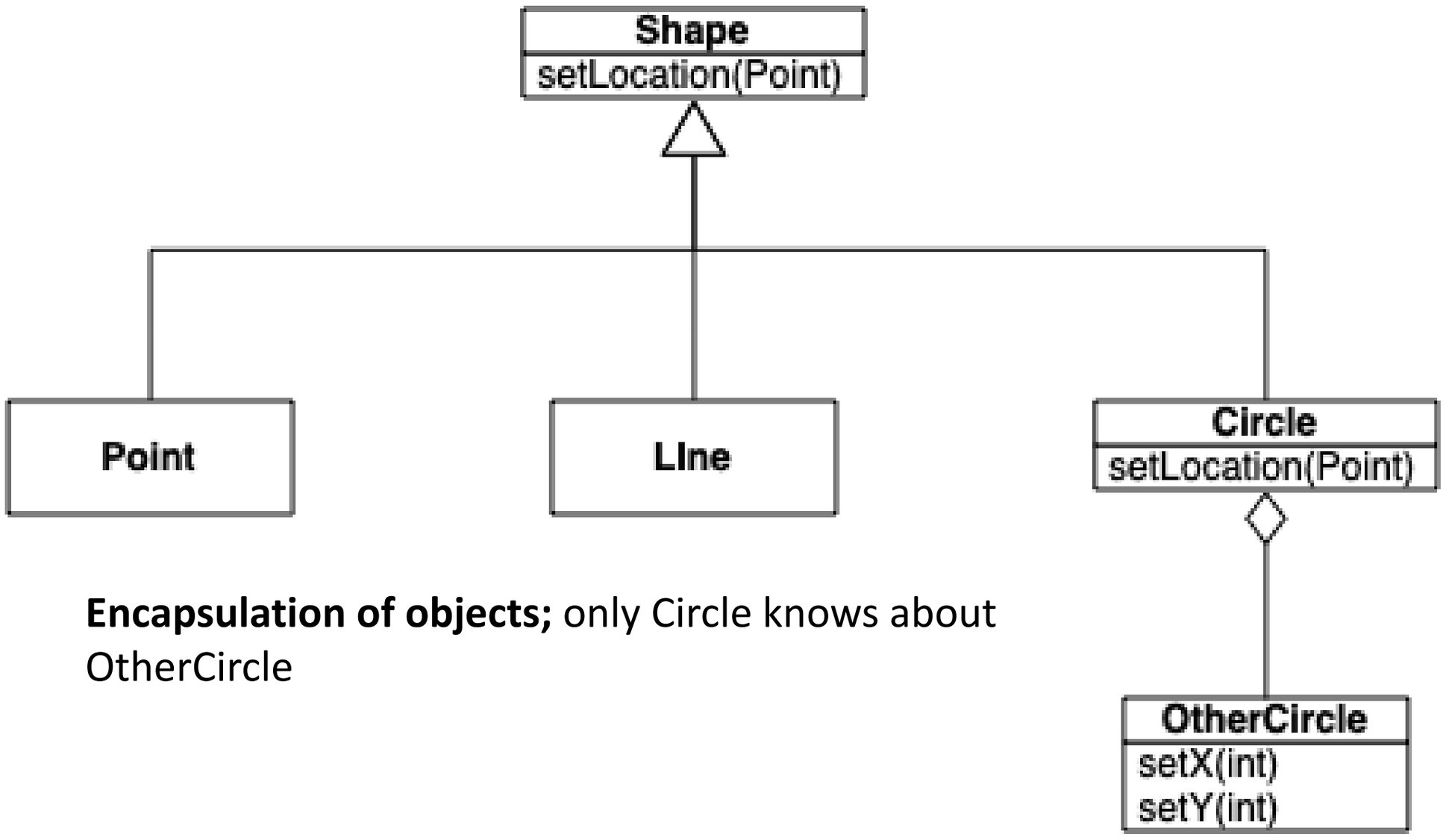
Encapsulation of Data within each class – the data in the classes are hidden from everything else

Multiple Types of Encapsulation (III)



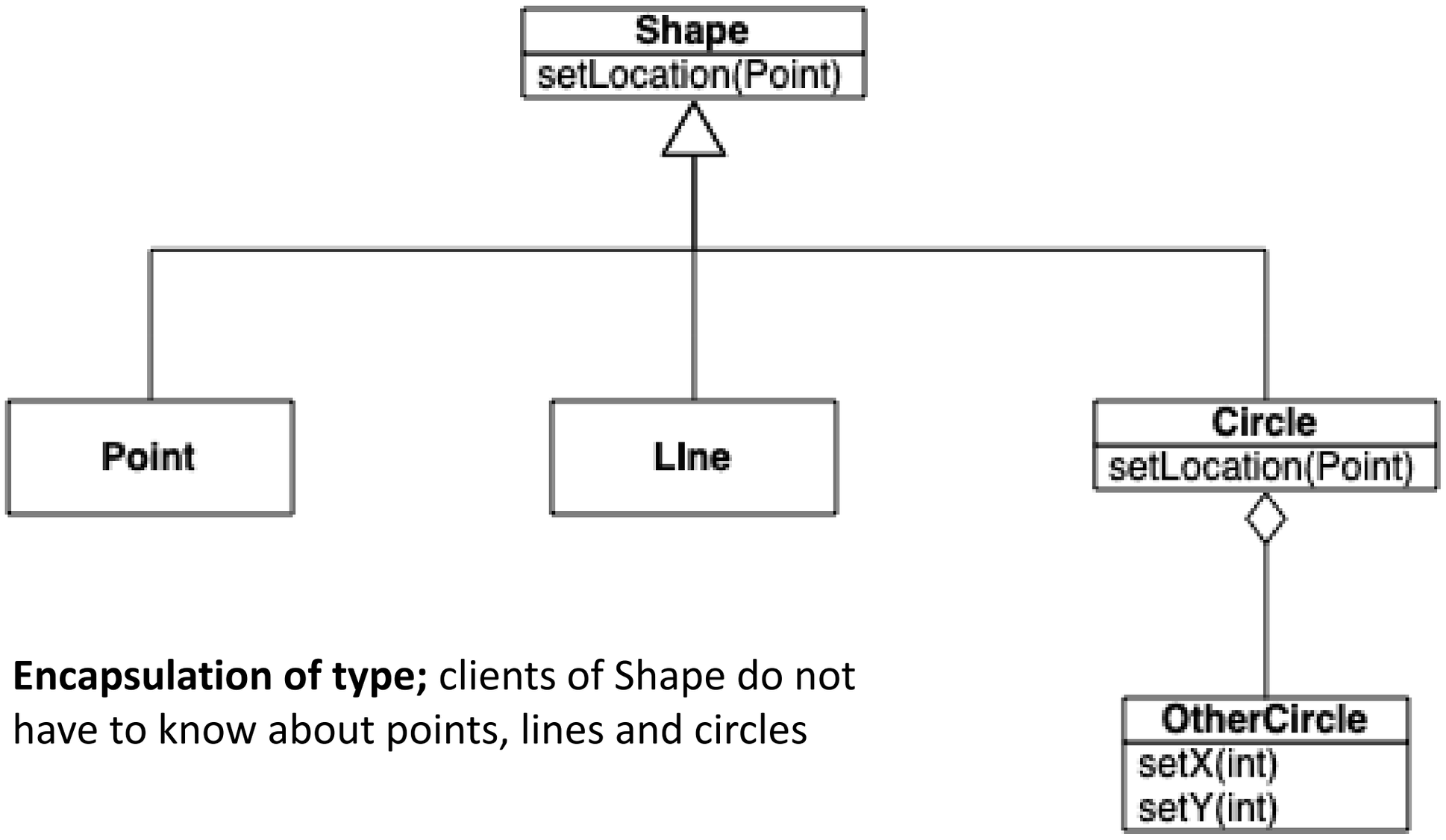
Encapsulation of methods; e.g. overloaded setLocation() in Circle

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
    - E.g. looping through a collection of Shapes
- Often (especially in the Gang of Four book), 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

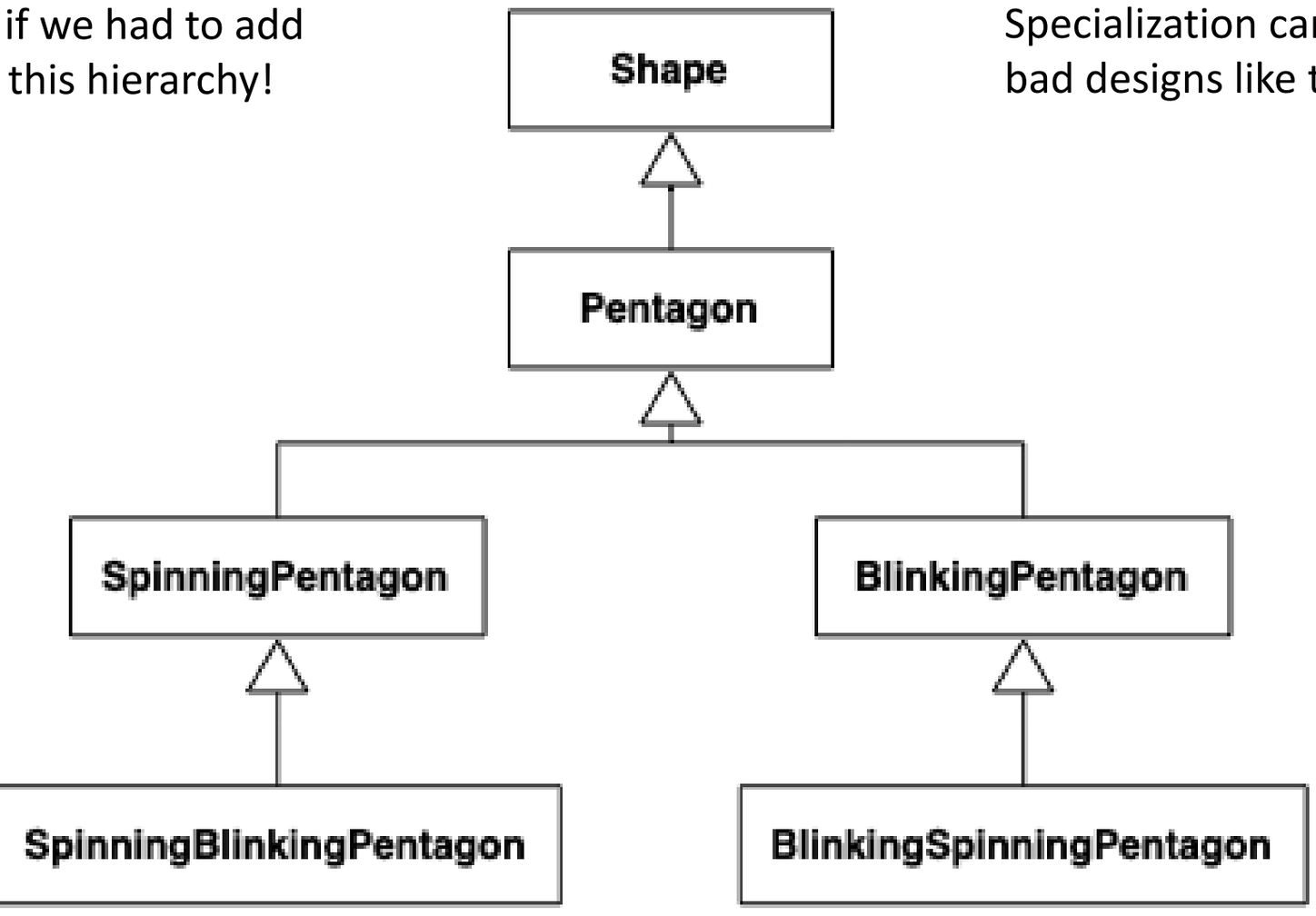
- Encapsulation of type provides a new way of looking at inheritance
  - Subclasses of the abstract types are grouped because they all behave the same way (as defined by the methods of the abstract type)
- This contrasts with inheritance used to “specialize” (make more specific) an existing class
  -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

Example

Imagine if we had to add Circle to this hierarchy!

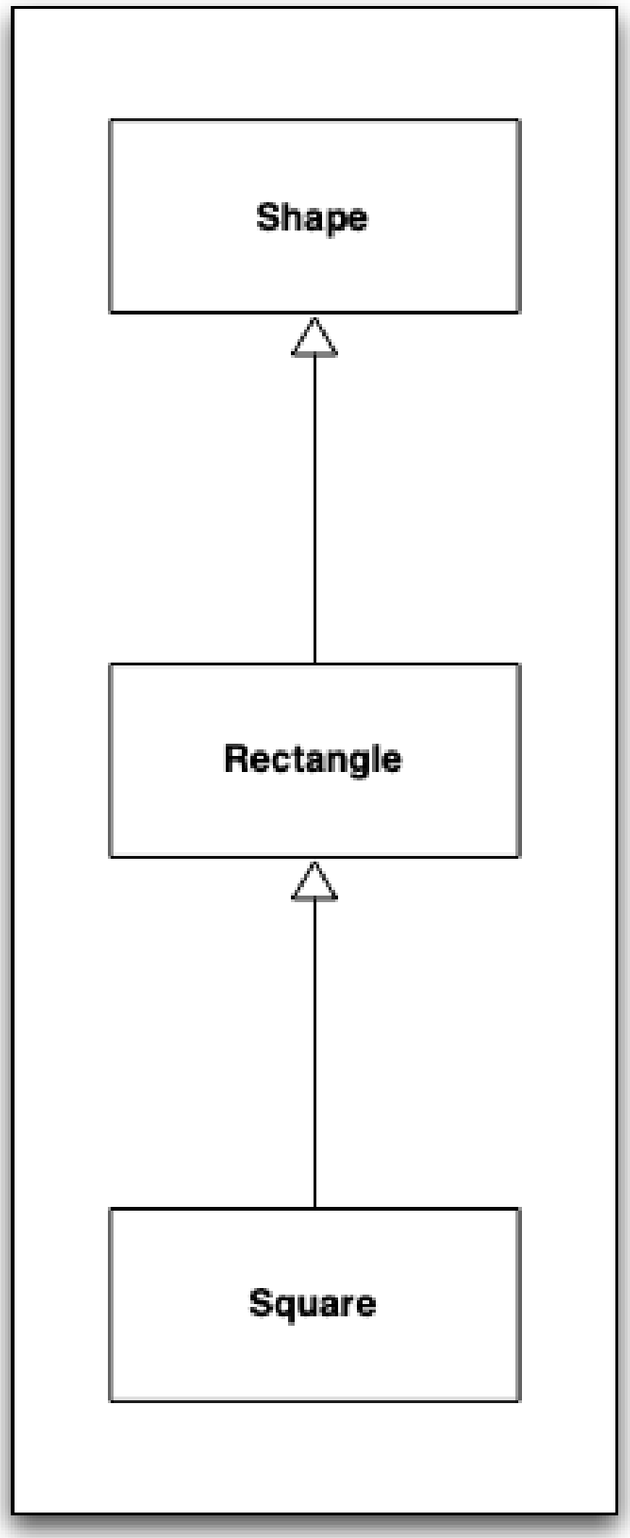


Specialization can lead to bad designs like this

Specialization vs. Behavior (III)

- To avoid the trap of SpecialBorderBlinkingSpinningPentagon
- Encapsulate variation in behavior using the Strategy pattern
  - 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 fit with these classes in as well
  - This approach scales; one new abstract type, one concrete subclass for each new behavior that varies

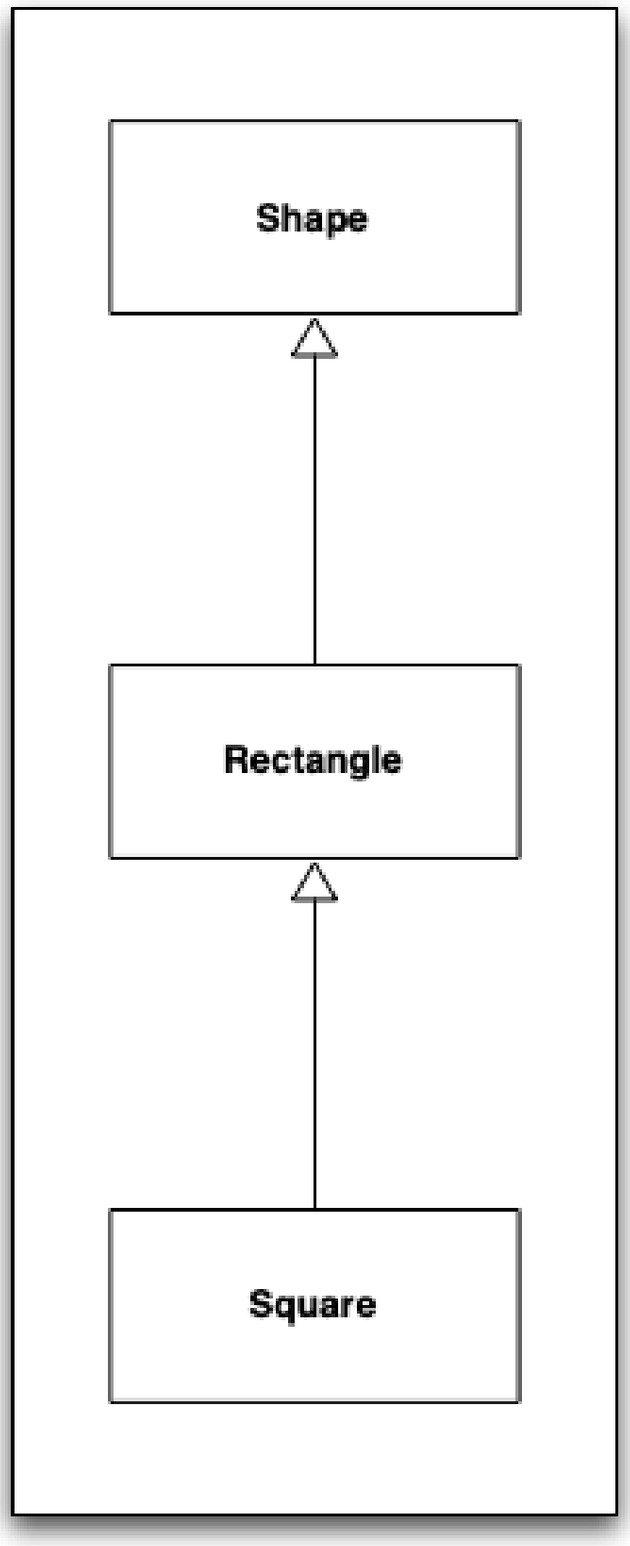
Example: Rectangle and Square



Rectangle IS-A Shape
Square IS-A Rectangle

Is there a problem with this design?

Example: Rectangle and Square

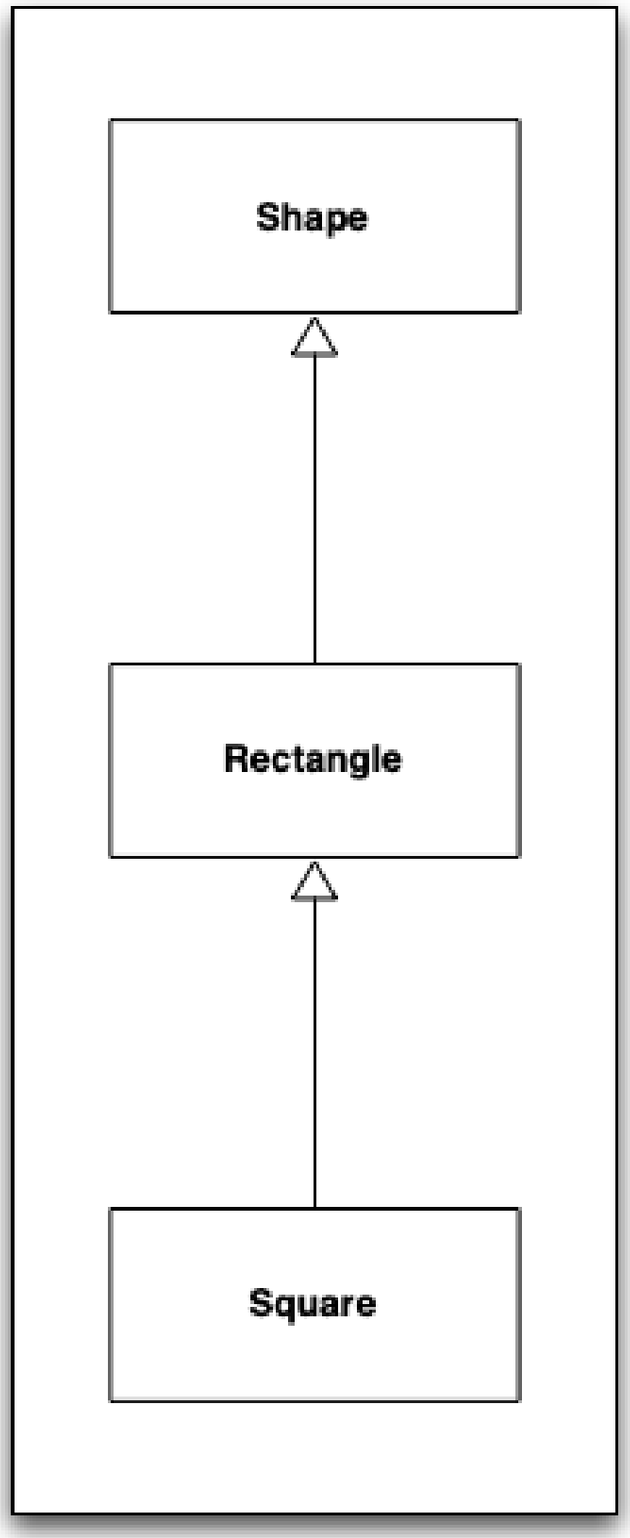


What would happen if we did something like this?

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

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

Example: Rectangle and Square



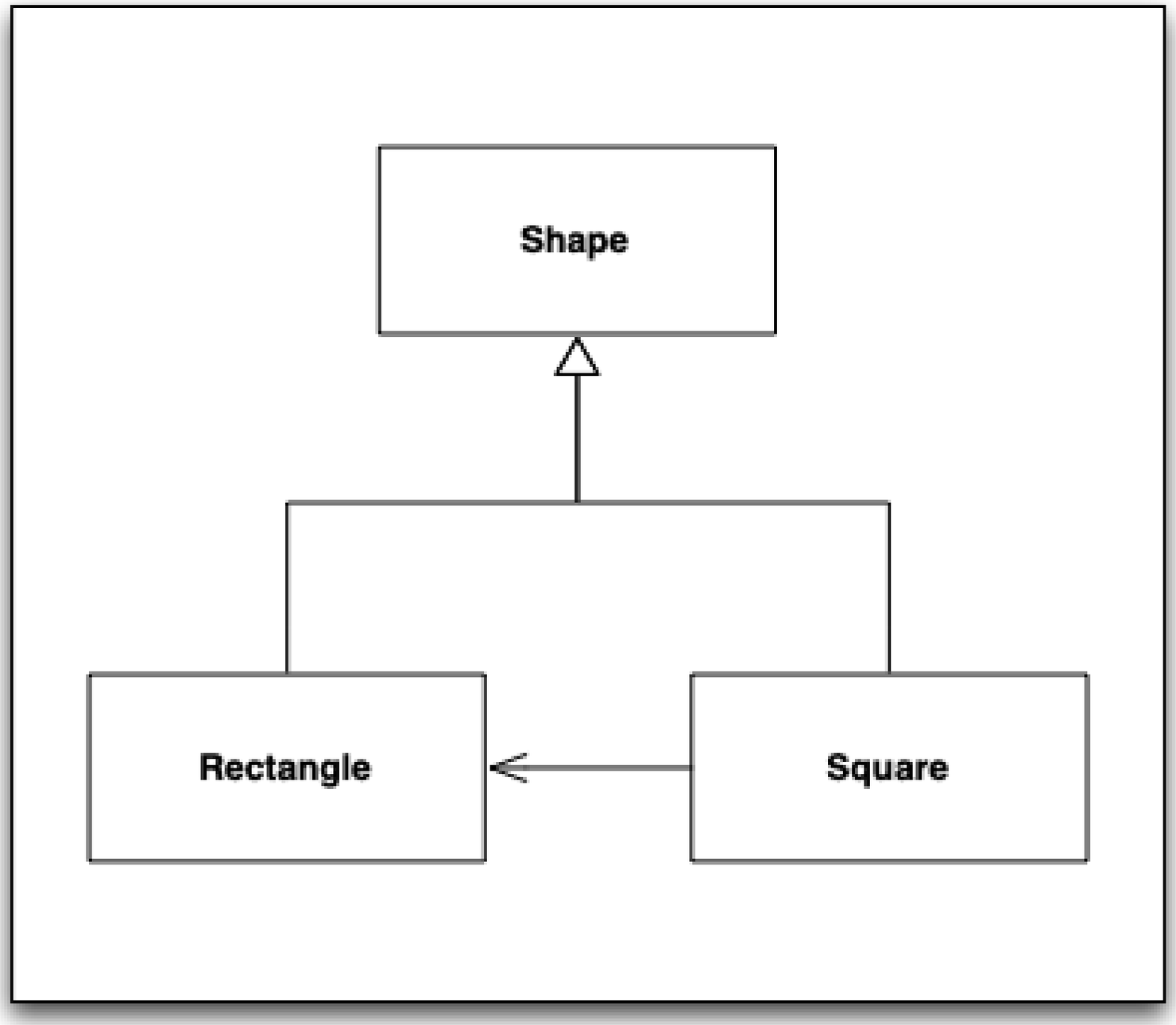
Squares share properties of rectangles but they don't BEHAVE the same

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

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

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

- Another OO Design Technique…
- Answers the question
  - What critical information do we need from a problem domain when we are performing analysis and design?
  - What do we need to know in order to be effective at modeling the domain and the problem we've been asked to tackle?
- Commonality Analysis: identifies the major archetypes of a domain
  - Look at ostensibly 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 the ways these common structures are likely to change
- Analysis and Design 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

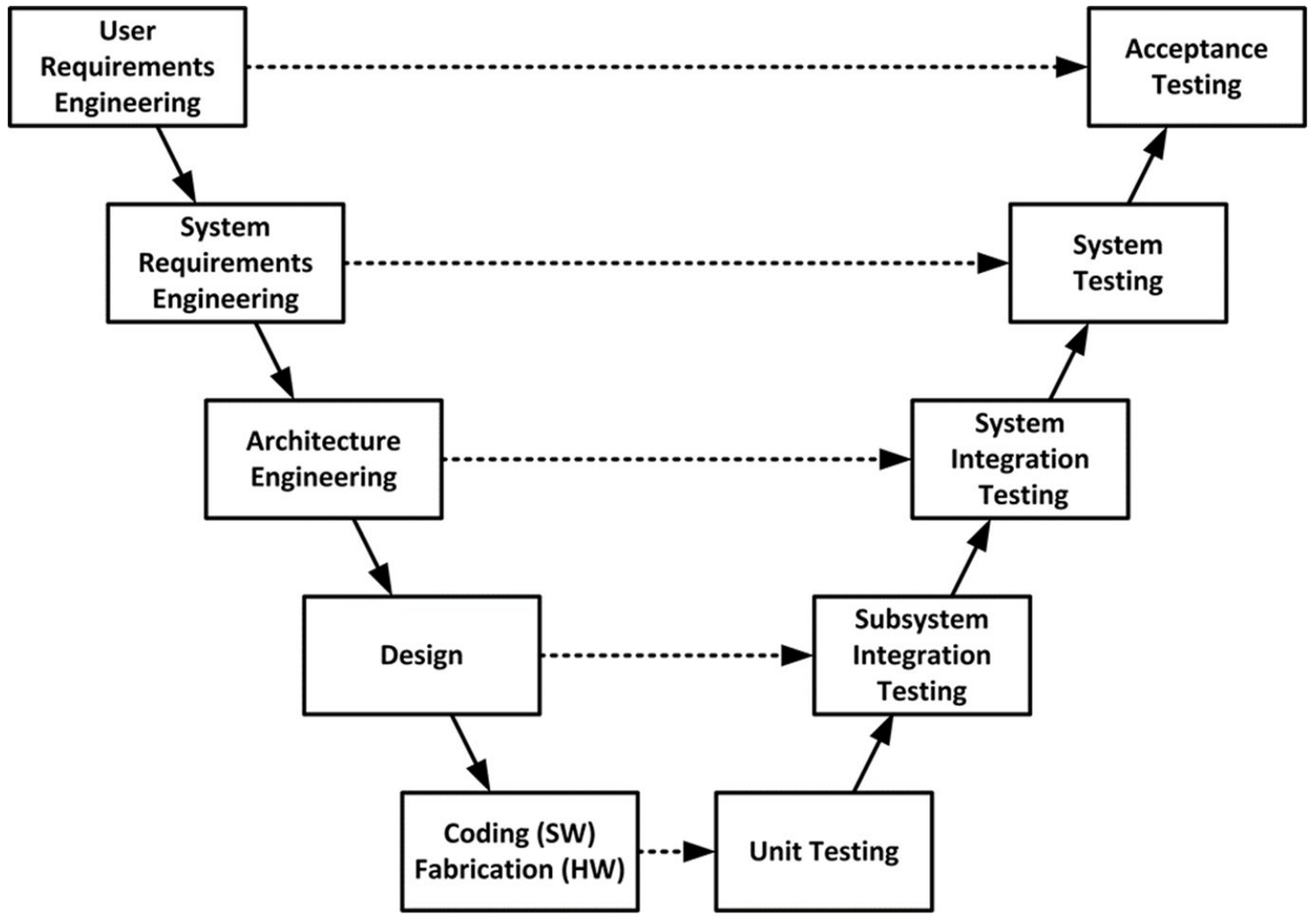
Design Up Front = Waterfall?

- The approach to A&D advocated here 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 and their tasks
  - 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

- Is this in conflict with Agile development?
- Does this mean working in a Waterfall process only?

Waterfall Processes

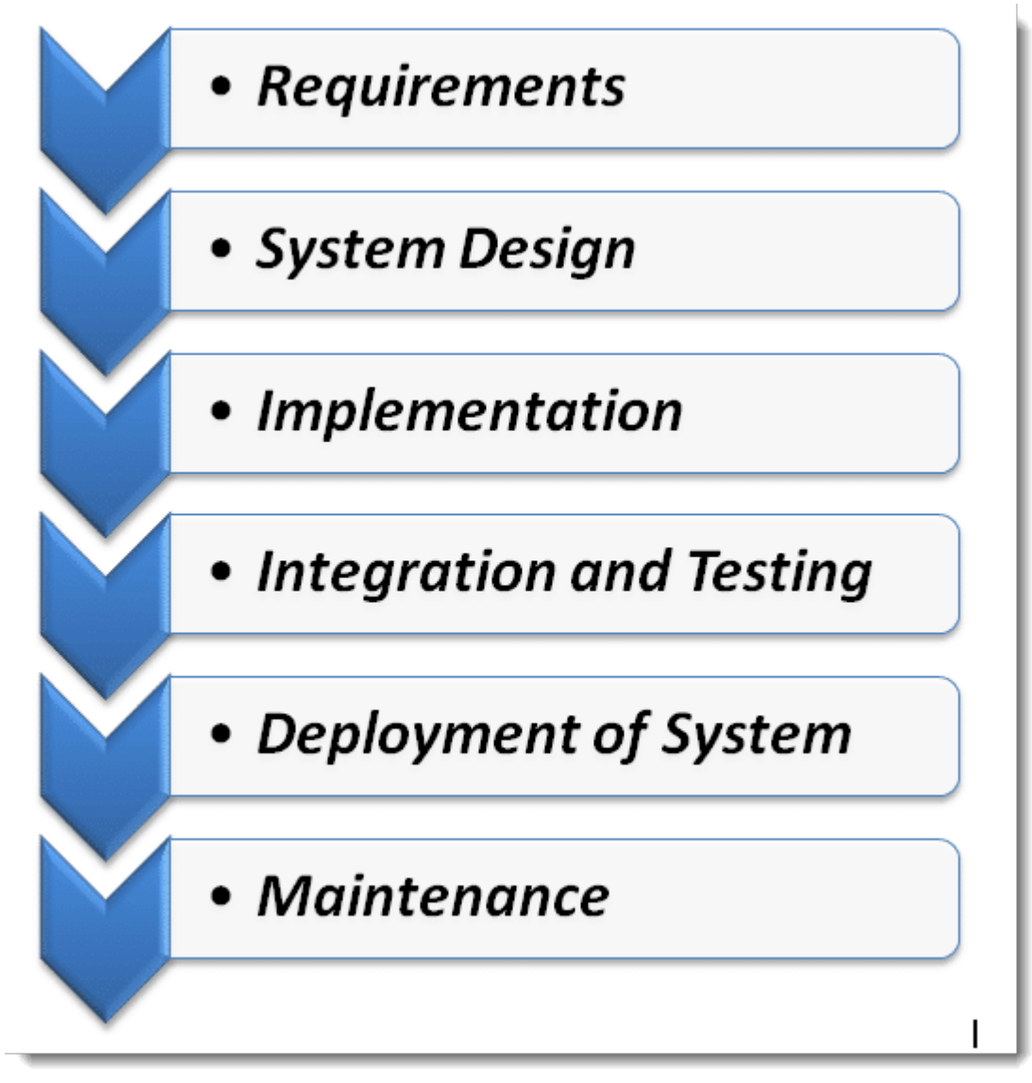
- Waterfall style processes usually move to levels of progressive design elaboration before development
- This is shown in the traditional “V” approach
- Usually a focus on up-front requirements
- Usually a focus on test planning for verifying and validating requirements



https://insights.sei.cmu.edu/sei_blog/2013/11/using-v-models-for-testing.html

Waterfall Processes

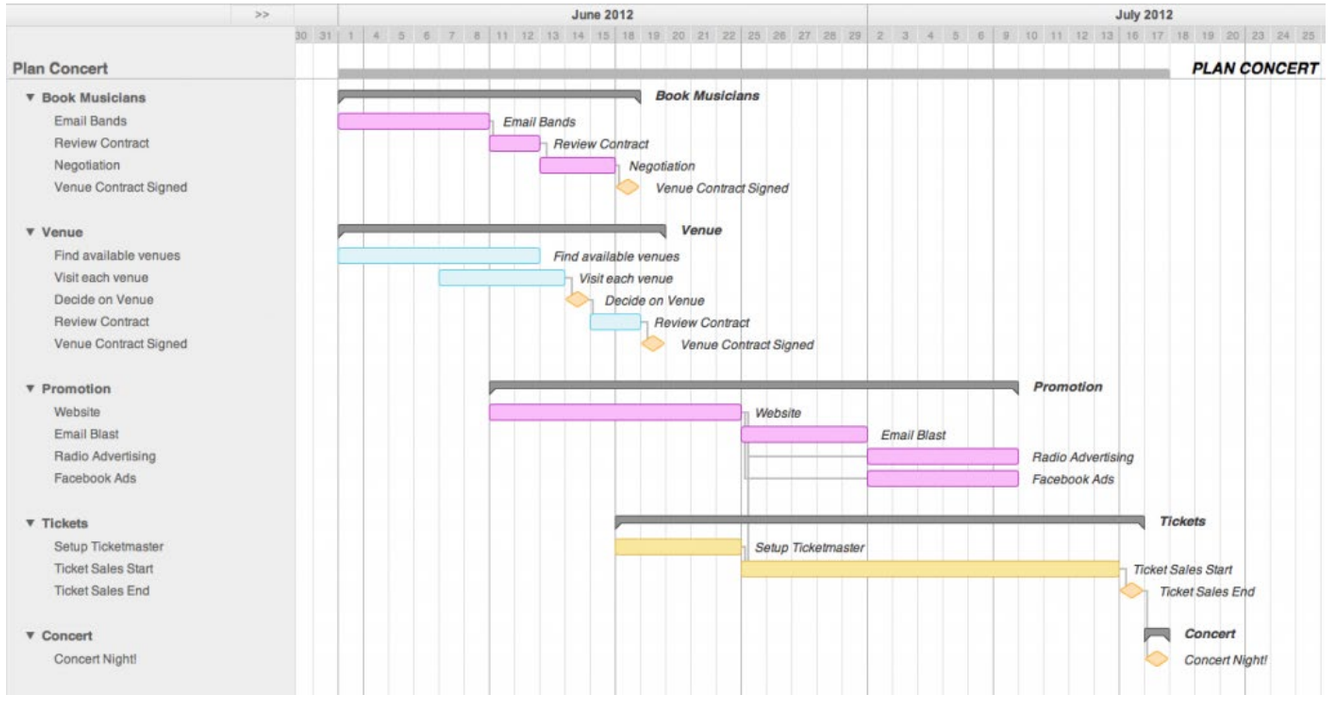
- Waterfall style processes usually move through development stages
- Stage gate reviews are used to decide if the project is ready to continue
- Most companies have tools and documentation standards for what supports each part of the process
- Most applications of this approach are carefully controlled with activities at each stage carefully managed



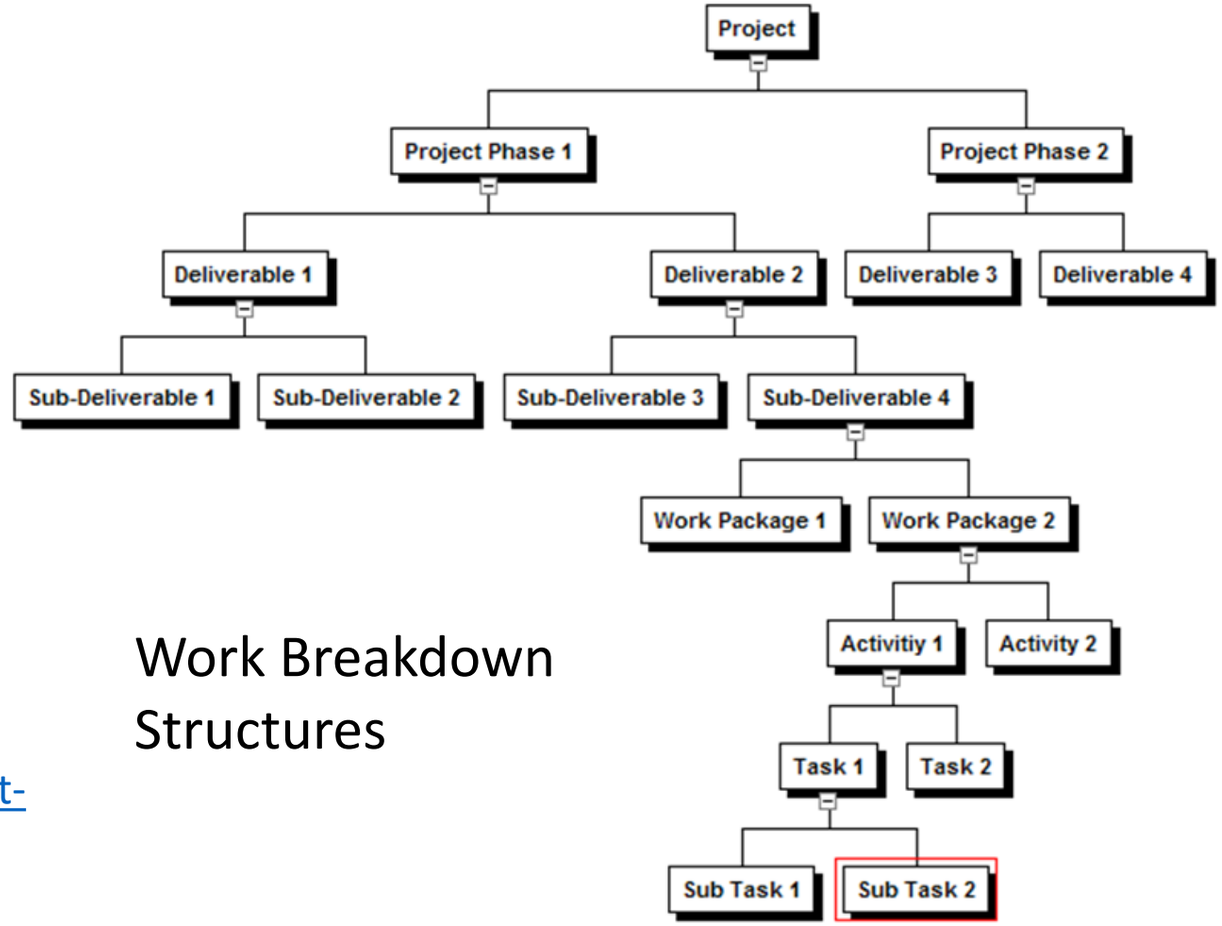
https://www.toolsqa.com/software-testing/waterfall-model/

Waterfall Process Tools

- Tools for Waterfall processes tend to focus on planning and control:
- Scope
- Schedule
- Cost
- Quality
- Resources
- Communications
- Risk
- Sourcing
- Stakeholders



Gantt Charts

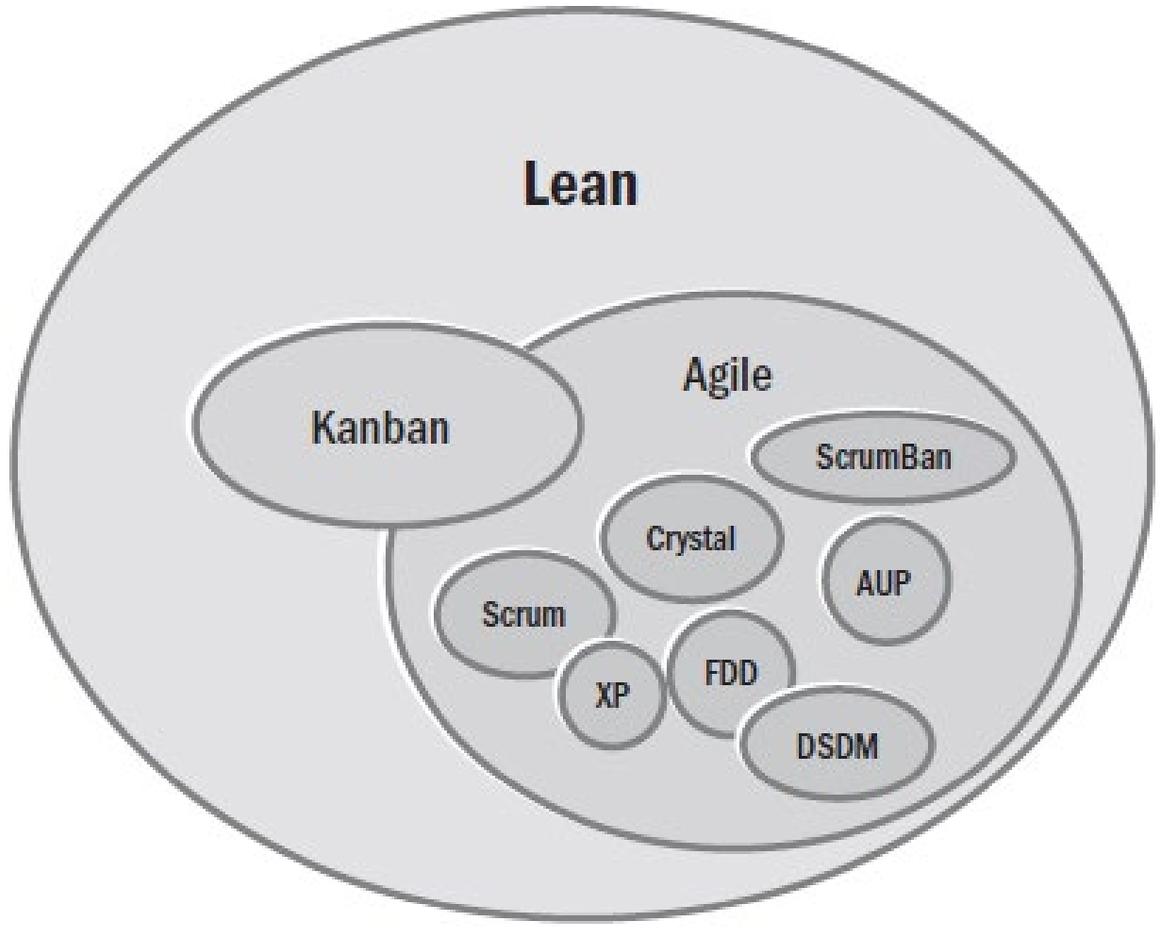


Work Breakdown Structures

https://blog.masterofproject.com/ten-project-management-knowledge-areas-pmbok/

Compare this to Agile Methods

- Many different processes in the category of Agile
- Most common are Scrum and Kanban
- FDD Feature Driven Development
- DSDM Dynamic Systems Development Method
- Image from PMI Agile Practice Guide, 2017



Agile Techniques

- Agile methods are techniques/processes for developing software systems that rely on
  - communicating with your customer (or product owner, representing the 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.

Manifesto for Agile Software Development

We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

Individuals and interactions over processes and tools
Working software over comprehensive documentation
Customer collaboration over contract negotiation
Responding to change over following a plan

That is, while there is value in the items on the right, we value the items on the left more.

https://agilemanifesto.org/

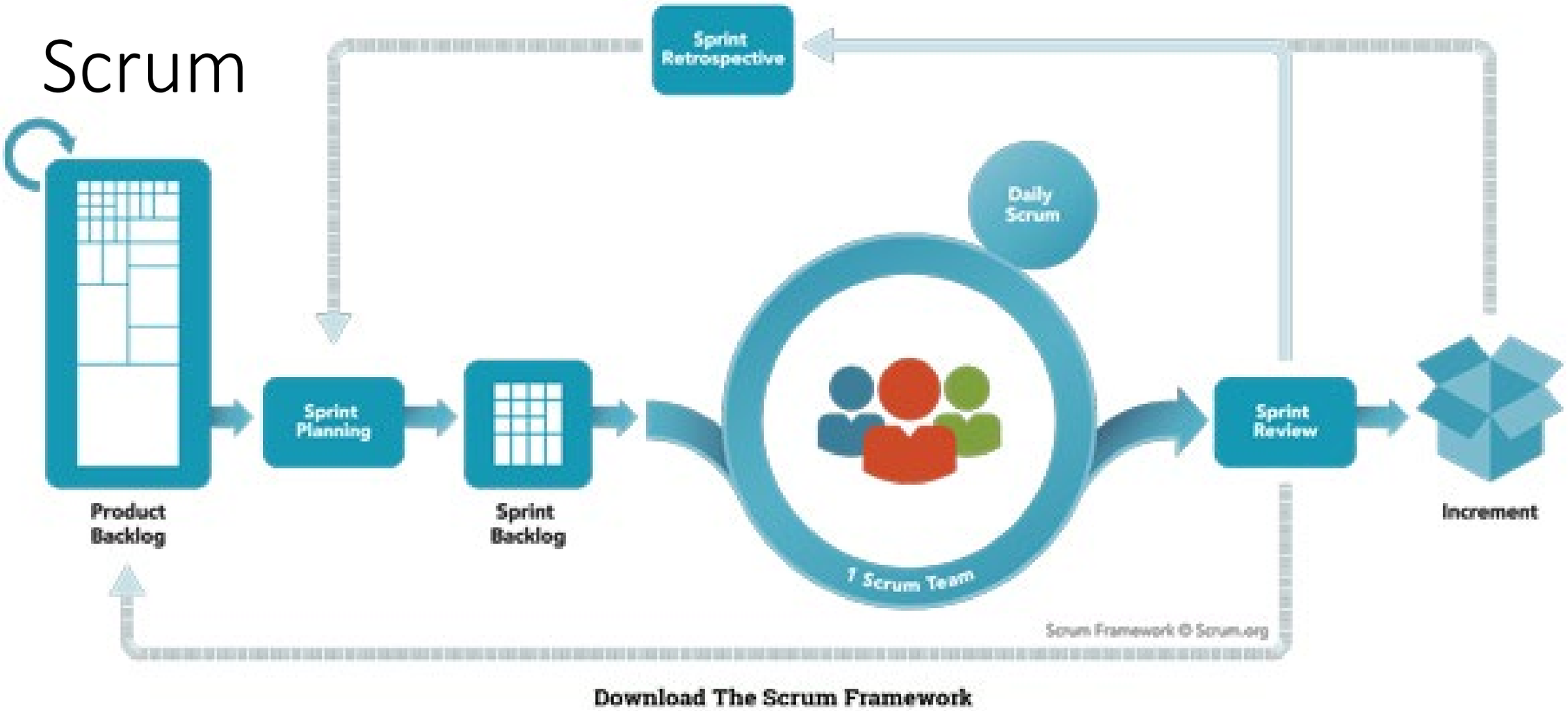
Agile Principles

We follow these principles:

- **Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.**
- **Welcome changing requirements**, even late in development. Agile processes harness change for the customer's competitive advantage.
- **Deliver working software frequently**, from a couple of weeks to a couple of months, with a preference to the shorter timescale.
- **Business people and developers must work together daily** throughout the project.
- Build projects around **motivated individuals.** Give them the environment and support they need, and **trust them to get the job done**.
- The most efficient and effective method of conveying information to and within a development team is **face-to-face conversation**.
- **Working software is the primary measure of progress**.
- Agile processes promote **sustainable development**. The sponsors, developers, and users should be able to **maintain a constant pace** indefinitely.
- Continuous **attention to technical excellence and good design** enhances agility.
- Simplicity--the art of **maximizing the amount of work not done--**is essential.
- The best architectures, requirements, and designs emerge from **self-organizing teams**.
- At regular intervals, **the team reflects** on how to become more effective, then tunes **and adjusts** its behavior accordingly.

https://agilemanifesto.org/principles.html

Scrum



Download The Scrum Framework

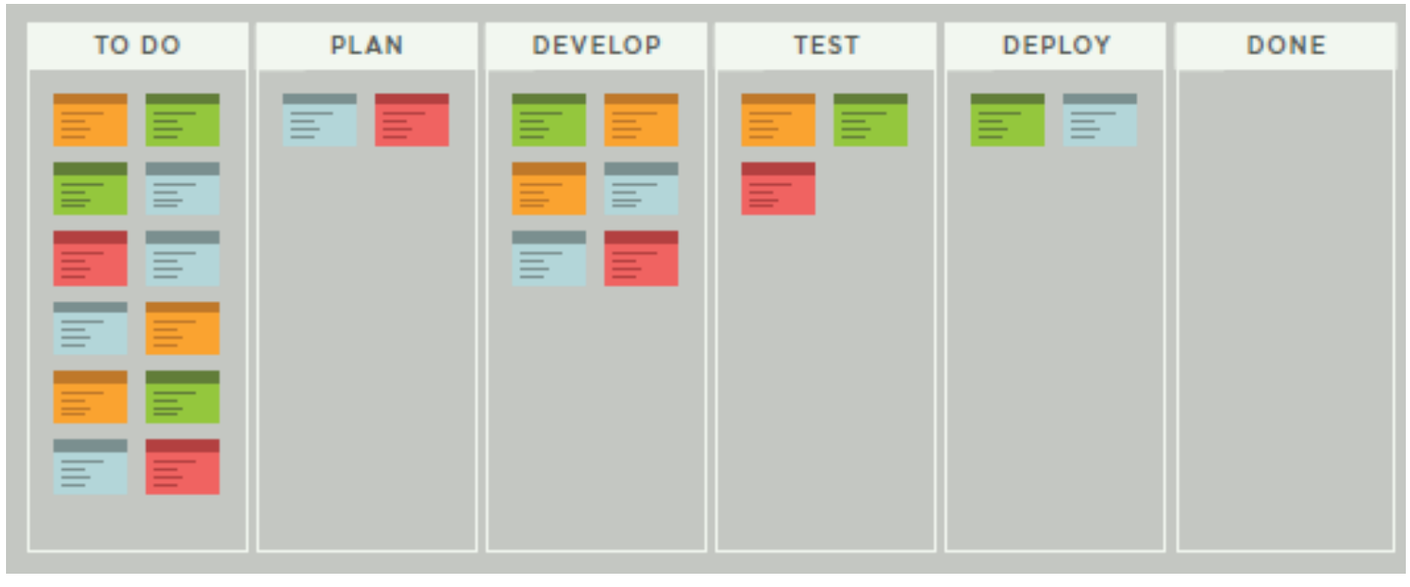
- Rugby term – (re)starting play, teamwork
- Product owner, development team, scrum master
- Sprints – time-boxed development, 1 week - 1 month
- Story points – estimates of effort for deliverables
- Daily scrum (stand-up meeting)
- Sprint reviews and demonstrations, retrospectives for continuous improvement

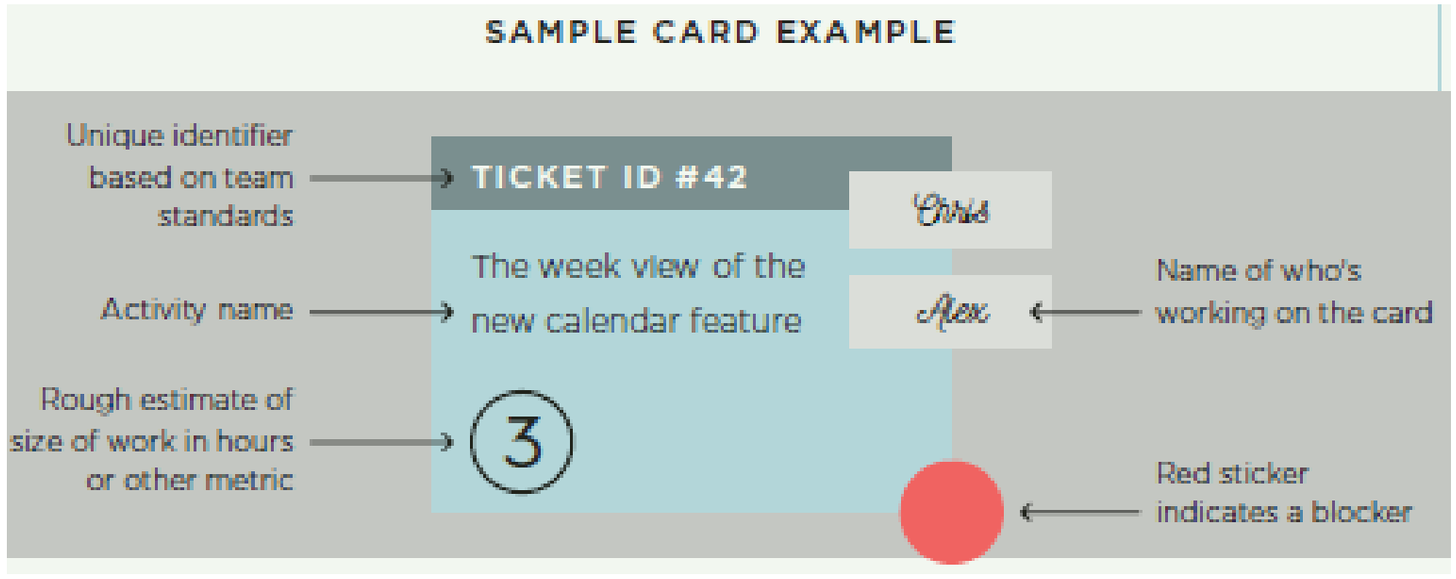
Questions for Scrum standups:

- What did you accomplish since the last meeting?
- What are you working on until the next meeting?
- What is getting in your way or keeping you from doing your job?

Kanban

- Developed initially as a lean method for a factory floor flow control system at Toyota – tasks are tracked on a Kanban board with daily reviews
- Individual work items are tracked on cards – Cards include estimated work, who's working on it, and whether the work is blocked – A key to making Kanban work is enforcing work in process limits

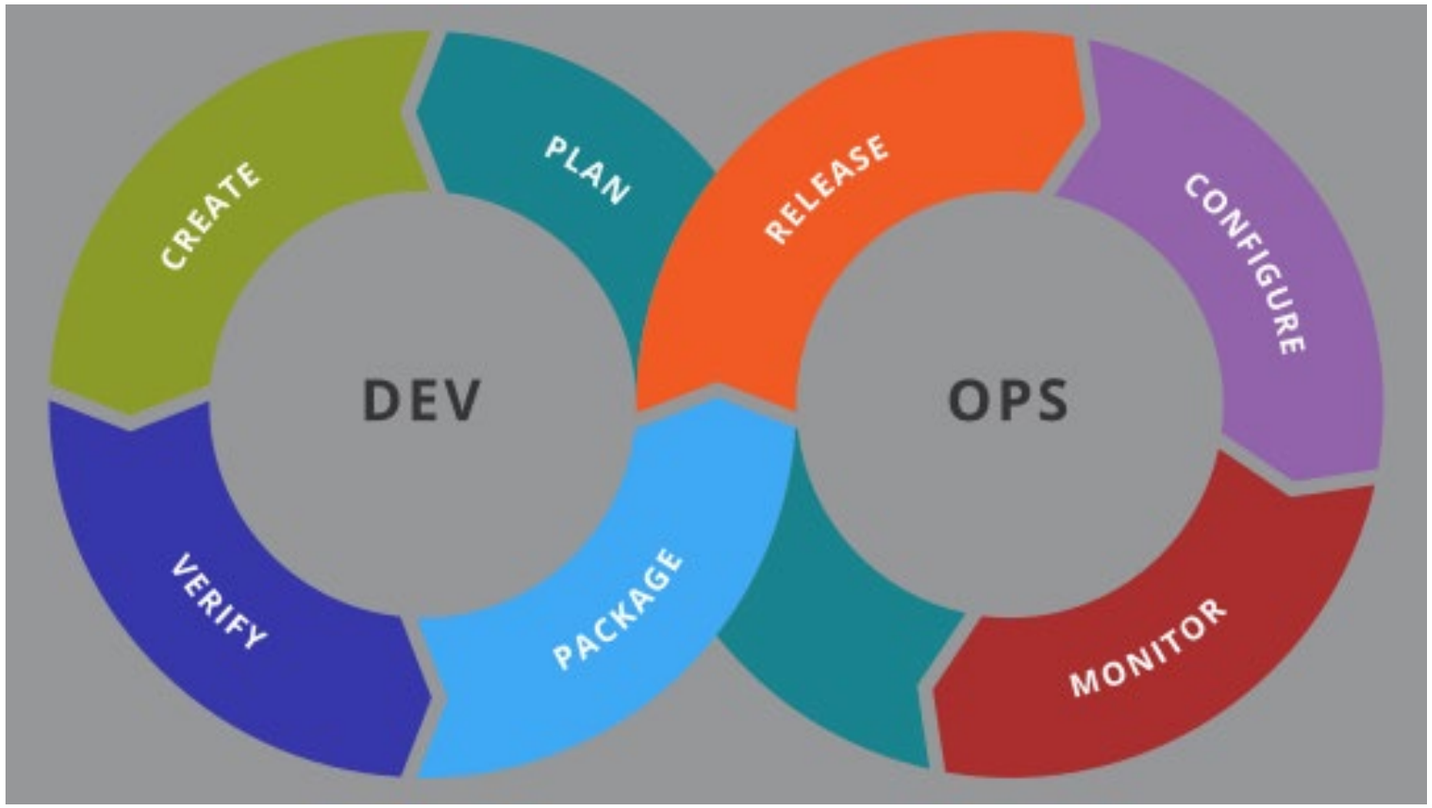




https://resources.leankit.com/guides/kanban-roadmap#

DevOps

- Process for continuous development and deployment (operations)
- Famous business novel “The Phoenix Project” on DevOps use
- https://www.redmineup.com/pages/blog/devops-in-redmine



Opposition?

- These two techniques seem to be in opposition
  - Up front design (top-down) - Waterfall
  - vs. iterative small steps (bottom-up) - Agile
- 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 value producing flexible code that supports change
  - Agile values code that can change to adapt to design cycles and discoveries

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 code quality goals here>
- 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 Rule – Kent Beck
  - 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)

- Readability is an essential quality of good code advocated in agile methods
- “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 a key goal in software development
  - The more you test the more confident you are in the software being developed
- Test-Driven Development (TDD) drives code through testing
  - Extreme Programming practice calls for tests to be written before code
- 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 – again loose coupling

Three General Goals for Good Software

- From TDD Lecture…
- Response to change
  - Readiness for change is supported by writing tests that depend on behavior in a specification
  - Automated regression testing helps keep bugs from coming back when changes are made
- Safe from bugs
  - Testing is about finding bugs in your code, and test-first programming is about finding them as early as possible, immediately when you introduced them
- Easy to understand
  - More part of documentation, code standards, and code review

- Reference https://ocw.mit.edu/ans7870/6/6.005/s16/classes/03-testing/

Summary

- Focusing perspective on objects and encapsulation
  - Responsibilities; hide anything (not only data)
- How to handle variation in behavior
  - Strategy pattern
- New perspective on inheritance
  - Group via behavior (vs. specialization)
- 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 OO Design Patterns, Waterfall and Agile processes
  - They all value the same code qualities
    - loose coupling, high cohesiveness, no redundancy, testability, readability, code to an interface, etc.

Summary

- Focusing perspective on objects and encapsulation
  - Responsibilities; hide anything (not only data)
- How to handle variation in behavior
  - Strategy pattern
- New perspective on inheritance
  - Group via behavior (vs. specialization)
- 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 OO Design Patterns, Waterfall and Agile processes
  - They all value the same code qualities
    - loose coupling, high cohesiveness, no redundancy, testability, readability, code to an interface, etc.

Are these OO Principles here?

- Encapsulate what varies
- Favor composition (delegation) over inheritance
- Program to interfaces not implementations
- Strive for loosely coupled designs between objects that interact
- Classes should be open for extension, but closed for modification
- Depend on abstractions, not concrete classes
- Only talk to your (immediate) friends
- Don’t call us, we’ll call you
- A class should have only one reason to change

Next Steps

- New quiz is this weekend; Grading continues…
- Assignments active
  - Semester Project Topic paper due today Fri 2/21 (not graded but required)
  - Graduate Project Topic Peer Review paper due Fri 2/28
  - Project 3 due Mon 3/2: be warned, don't underestimate the work…
- Next topics: Template, Steve McConnell, another exercise…
- Exams
  - Midterm Friday 3/6 in class – distance students see posted policies
  - Optional Canvas-based Final – Wednesday 5/6 1:30 to 4 PM in class ECCR 150
    - Final exam score does not affect Midterm grade
    - Final exam grade will be highest of Midterm grade and Final grade (if taken)
- Class Staff can help!
  - Bruce: Tue 4 - 5 PM and Wed 10:30 - 11:30 AM in ECOT 242
  - Manjunath: Monday 1 – 3 PM in DLC Lobby
  - Neethi: Tuesday 1 – 3 PM in DLC Lobby
  - Anirudh: By request
  - Ask for any other coverage you need