Good Design == Flexible Software (Part 2)

Kenneth M. Anderson
University of Colorado, Boulder
CSCI 4448/5448 — Lecture 10 — 09/24/2009

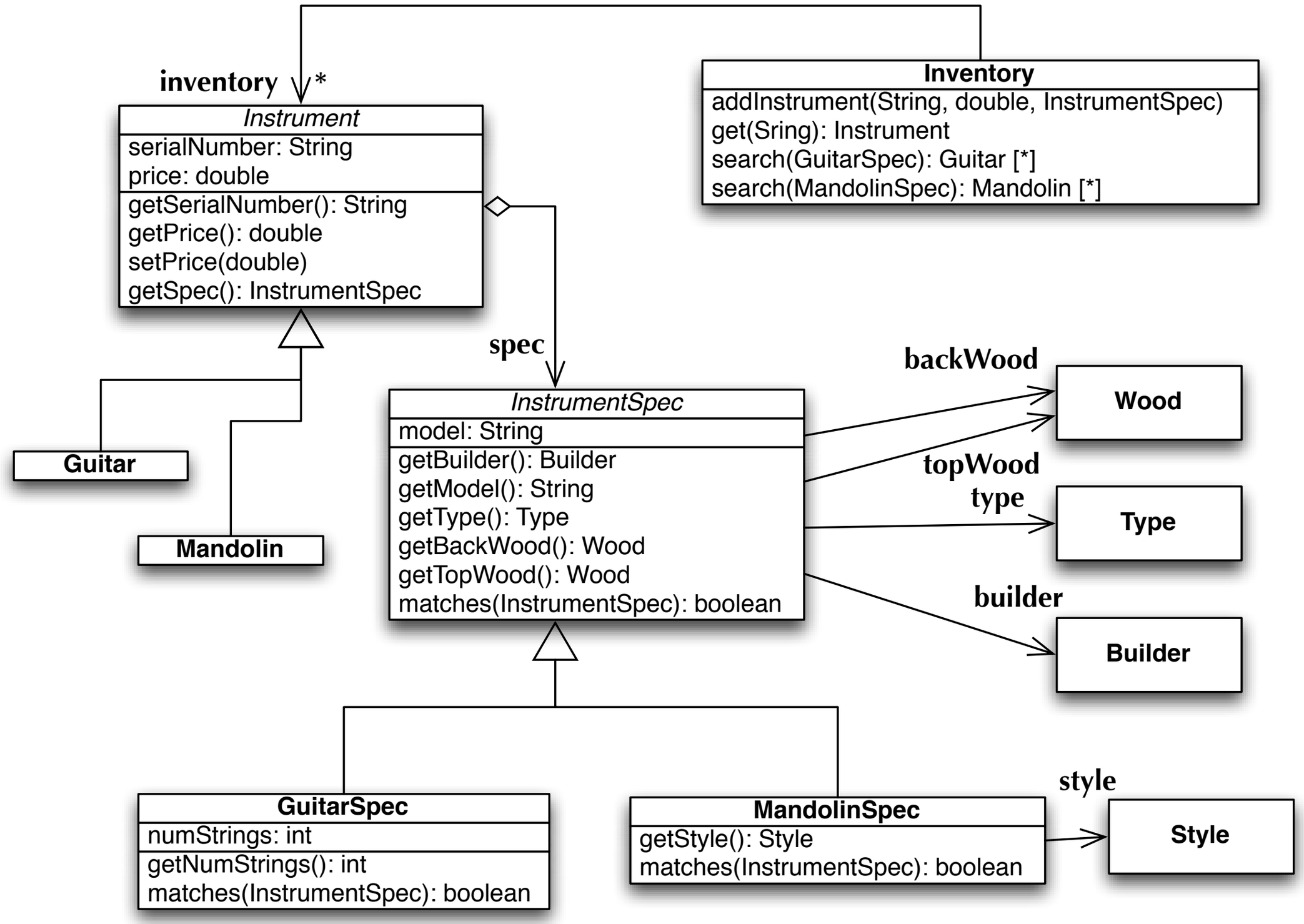
Lecture Goals

- Review material from Chapter 5, part 2, of the OO A&D textbook
    - Good Design == Flexible Software
    - How to achieve flexible software
    - The Importance of Iteration
    - The Great Ease-Of-Change Challenge
    - Cohesive Classes
    - Discuss the Chapter 5 Example: Rick’s Guitars, Revisited
    - Emphasize the OO concepts and techniques encountered in Chapter 5

Review: Our Three (Most Recent) OO Principles

- **Code to an Interface**
  - If you have a choice between coding to an interface or an abstract base class as opposed to an implementation or subclass, choose the former
  - Let polymorphism be your friend
- **Encapsulate What Varies**
  - Find the locations of your software likely to change and wrap them inside a class; hide the details of what can change behind the public interface of the class
- **Only One Reason To Change**
  - Each class should have only one reason that can cause it to change
    - (the thing it encapsulates)

Rick's Current Application



The Problems?

- Inventory.addInstrument() has code specific to each instrument type
    - If we add a new Instrument subclass, we have to change this method.
- The Inventory class has one search() method for each type of instrument
    - InstrumentSpec is an abstract class and we can’t instantiate it directly.
        - This means that we can’t search across instruments easily
- Instrument subclasses don’t offer much functionality
    - Instrument does all the hard work.
- Each time we add a new subclass to Instrument, we need to add a new subclass to InstrumentSpec

search() Upgrade

- We start by looking at the problem of having multiple search() methods
    - one for each subclass of InstrumentSpec
- This situation is simply not tenable… our system will never be flexible if we have to change Inventory's public interface every time we want to add a new type of Instrument to our system!
    - Having a method for each subclass of InstrumentSpec feels like we are coding to implementations, violating the "code to an interface" principle
- But, we had to do this because InstrumentSpec is abstract and we made it abstract since we wanted it to define a base class (or interface) for all of its possible subclasses
    - Recall that we created the GuitarSpec class in the first place to address the problem that Guitar was initially being asked to play two roles

```
public abstract class InstrumentSpec {

  private Builder builder;
  private String model;
  private Type type;
  private Wood backWood;
  private Wood topWood;

  public InstrumentSpec(Builder builder, String model, Type type,
                        Wood backWood, Wood topWood) {
    this.builder = builder;
    this.model = model;
    this.type = type;
    this.backWood = backWood;
    this.topWood = topWood;
  }

  public Builder getBuilder() {
    return builder;
  }

  public String getModel() {
    return model;
  }

  public Type getType() {
    return type;
  }

  public Wood getBackWood() {
    return backWood;
  }

  public Wood getTopWood() {
    return topWood;
  }

  public boolean matches(InstrumentSpec otherSpec) {
    if (builder != otherSpec.builder)
      return false;
    if ((model != null) && (!model.equals("")) &&
        (!model.equals(otherSpec.model)))
      return false;
    if (type != otherSpec.type)
      return false;
    if (backWood != otherSpec.backWood)
      return false;
    if (topWood != otherSpec.topWood)
      return false;
    return true;
  }
}
```

If you look closely you'll see that there is NOTHING abstract about InstrumentSpec!

search Upgrade

- To follow the “code to an interface” design principle (in this context), lets try making InstrumentSpec a concrete class
  - This would allow us to instantiate the class directly
  - We can eliminate all of the instrument-specific search() methods and replace it with a single method with the signature
    - public List search(InstrumentSpec searchSpec);
- Do we lose functionality with this approach?
  - Not really. If we want to do an instrument-specific search, just pass in a subclass of InstrumentSpec. This is what we had already
- Do we gain functionality with this approach?
  - Yes! We can search for more than one type of Instrument by passing in an InstrumentSpec that contains attributes shared by all instruments

search() Upgrade

```
1  public List search(GuitarSpec searchSpec) {
2      List matchingGuitars = new LinkedList();
3      for (Iterator i = inventory.iterator(); i.hasNext(); ) {
4          Guitar guitar = (Guitar)i.next();
5          if (guitar.getSpec().matches(searchSpec)) {
6              matchingGuitars.add(guitar);
7          }
8      }
9      return matchingGuitars;
10 }
11
12 public List search(MandolinSpec searchSpec) {
13     List matchingMandolins = new LinkedList();
14     for (Iterator i = inventory.iterator(); i.hasNext(); ) {
15         Mandolin mandolin = (Mandolin)i.next();
16         if (mandolin.getSpec().matches(searchSpec)) {
17             matchingMandolins.add(mandolin);
18         }
19     }
20     return matchingMandolins;
21 }
22
```

search(): the old way!

search() Upgrade

```
1  public List search(InstrumentSpec searchSpec) {
2      List results = new LinkedList();
3      for (Iterator i = inventory.iterator(); i.hasNext(); ) {
4          Instrument instrument = (Instrument)i.next();
5          if (instrument.getSpec().matches(searchSpec)) {
6              results.add(instrument);
7          }
8      }
9      return results;
10 }
11
```

search(): the new way. Can search across all instruments; can perform instrument-specific searches via substitutability and polymorphism

search() Upgrade (VI)

- The point on the last slide is key
  - Instrument-specific searches are enabled **via substitutability and polymorphism**
  - If we pass in a subclass of InstrumentSpec, the code on the previous slide still works and the behavior is the same
    - matches() works as before but since we have a subclass, it behaves in a way that is tailored to its subclass: this is fine!
- The major point?
  - Classes are really about behavior! (not data)
  - You create a new subclass when you need slightly different behavior for instances of the subclass
    - Rectangle IS-A Shape, but you need Rectangle behavior when getPerimeter() is invoked

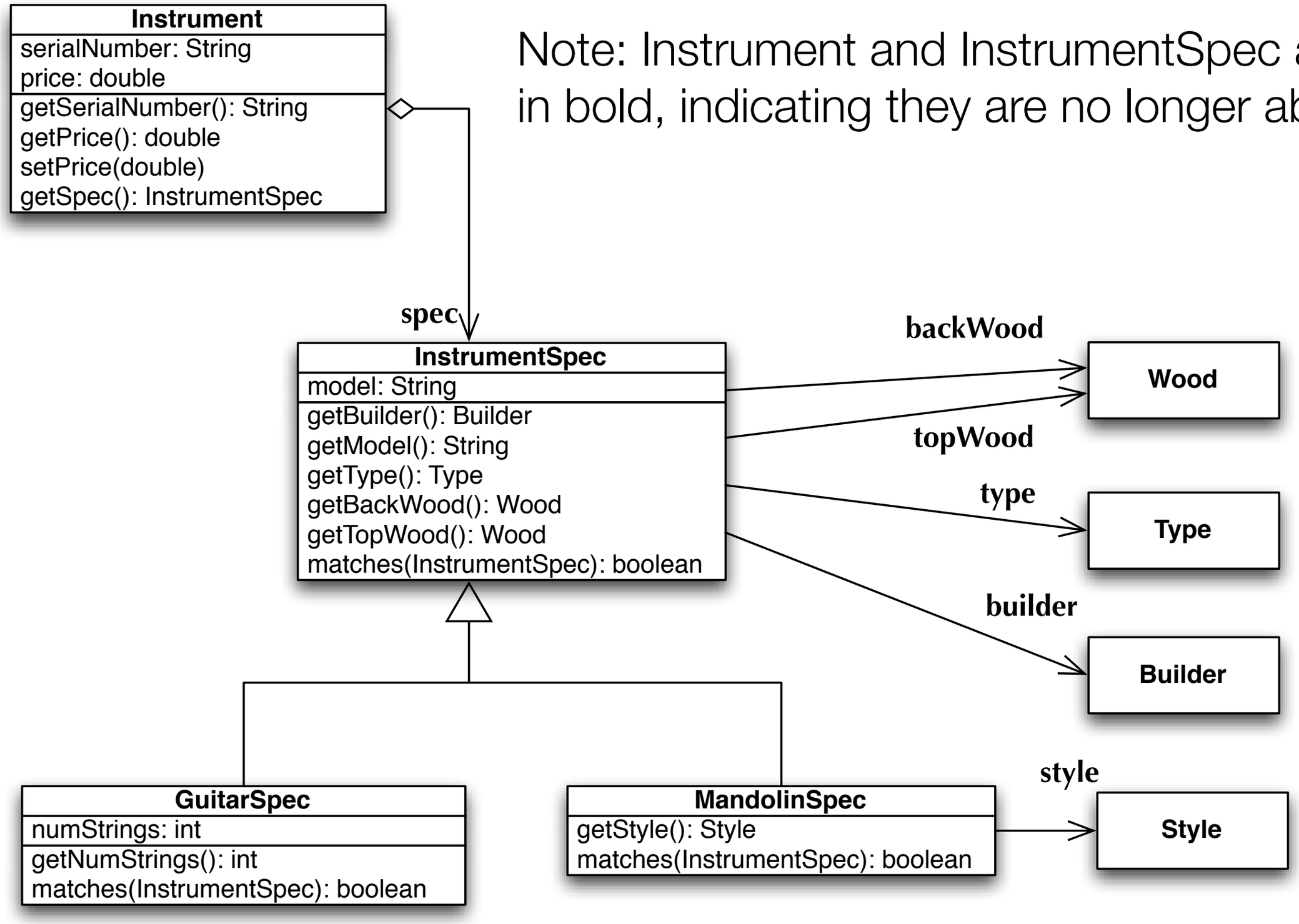
Instrument Upgrade (I)

- Lets look at the problem of Guitar and Mandolin being “empty” classes
  - They are empty because the behavior of the Instrument class does all they need; they don’t need to behave differently for any of its functionality
  - The same is not true for GuitarSpec and MandolinSpec… we really do need different behaviors: if the customer is looking for a guitar with a particular number of strings, then only a GuitarSpec will do
- Since classes are all about behavior, Guitar and Mandolin provide no real benefit to our system… lets get rid of them!
  - This may not always be true… in this situation, our software is all about maintaining an inventory and searching it. If our system was different, say it focused on composing music, then Guitar and Mandolin would definitely provide behaviors that were different from their generic superclass

Instrument Upgrade (II)

- We initially created the Instrument and InstrumentSpec classes to merge common properties between Guitars and Mandolins.
- We ended up with a design in which attributes related to “instrument as product” (serial number and price) went into the Instrument class and attributes related to the instrument itself went into InstrumentSpec.
    - As a result, we already had a design in which the “thing that varies,” i.e. properties between instruments had been encapsulated
    - So, with no differences in behavior with Instrument and with no properties stored directly in themselves, Guitar and Mandolin are truly useless classes
- We will make Instrument a concrete class and get rid of Guitar and Mandolin

Instrument Upgrade (III)



Note: Instrument and InstrumentSpec are both in bold, indicating they are no longer abstract

Death of a Design Decision

- In 8 slides, we undid most of the design work that we did in Lecture 9!
  - Instrument-specific subclasses was a decision that just didn't work out
  - But, in our defense, recall the history.
    - We STARTED with a Guitar class, then added a Mandolin class
    - Then generalized to create an Instrument class
    - Then realized that if we wanted to add new instruments that we'd have to do a lot of work each time
- One of the hardest things you'll have to do as a software developer is to let go of mistakes you made in your own design
  - Those decisions seemed to make sense at the time and it's HARD to change something you thought was already working
- Design is iterative… you have to be willing to change your own designs in order to reach a state in which your system is flexible with respect to its reqs.

Double Encapsulation (I)

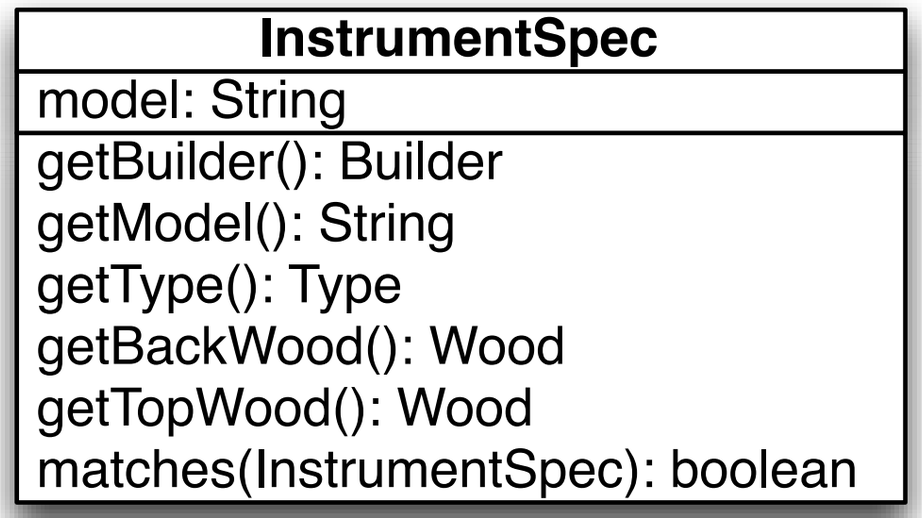
- If encapsulation is good, why not apply it again?
  - “Lets take Encapsulation and set it at 11!”: ***Obligatory Spinal Tap Reference***
- We mentioned that instrument properties are “things that can vary” and we shielded the application from this type of change with InstrumentSpec
- However, this design isn’t perfect
  - First of all, the InstrumentSpec class was created by merging two similar instruments… what happens if Rick decides to change the name of his business to “Rick’s Instruments” and starts carrying trombones?
    - Properties like “backWood” and “topWood” would not apply
- **We want to shield InstrumentSpec from this type of variation**, so we want to apply encapsulation again, hence “double encapsulation”
  - The book is right, this is not a standard OO A&D term
  - Just remove the word “double” and you’ve got the right term!

Double Encapsulation (II)

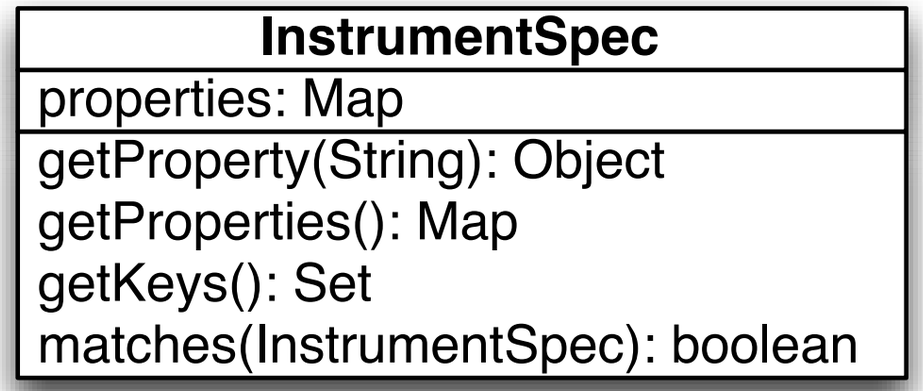
- Our goal: we want to specify a flexible set of properties for InstrumentSpec
  - By flexible we mean that the InstrumentSpec class does not have to change if we decide to change an existing property of an instrument or add a new one
  - We don’t want to hard code the properties into the class
- In order to be this flexible, the properties have to be managed dynamically in a collection class
  - A Map (hash table) is perfect for this task
  - A Map is a set of key-value pairs. Keys are typically strings and values can be an instance of ANY class (this provides tremendous flexibility)

Double Encapsulation (III)

Before



After



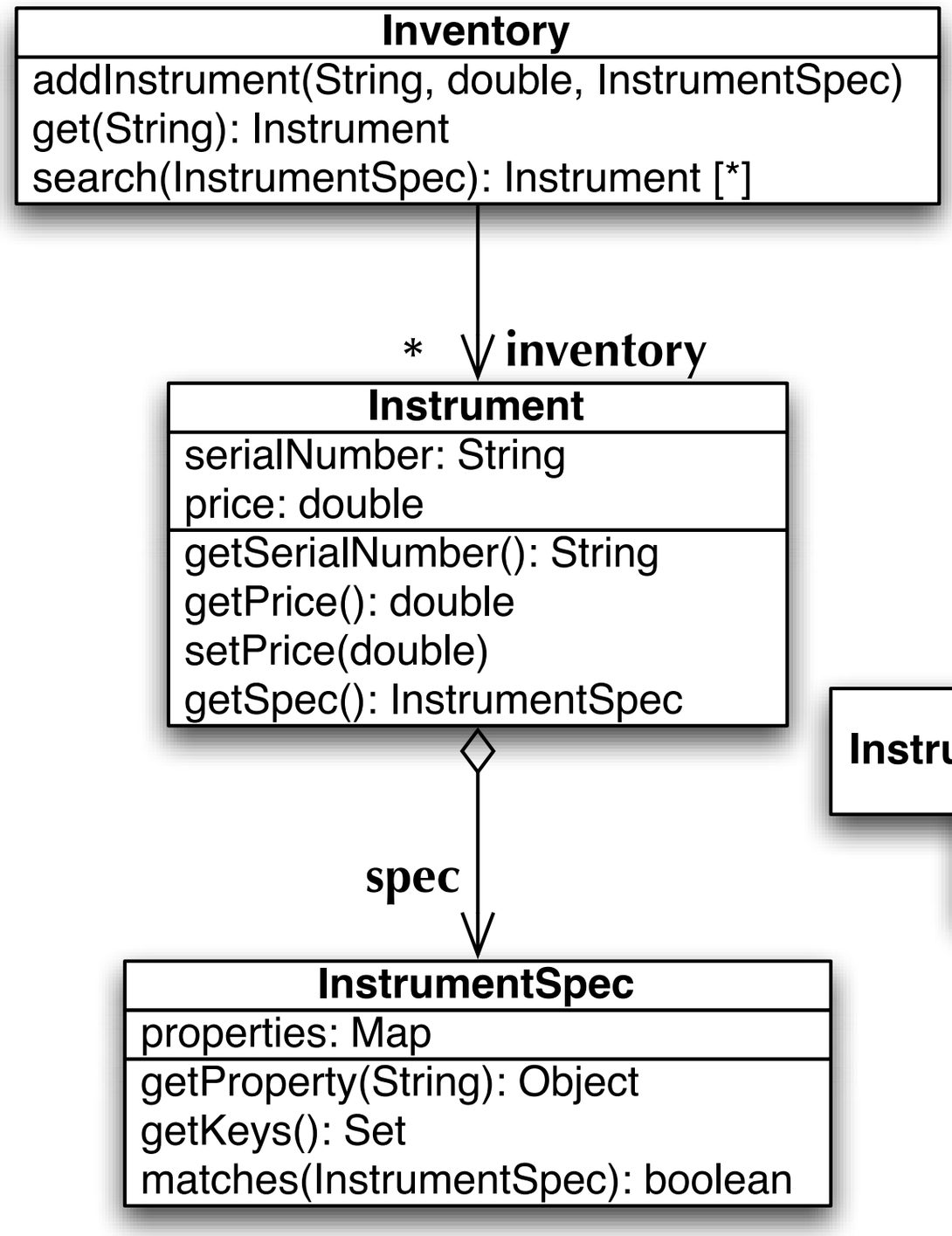
By switching to a Map to hold properties, InstrumentSpec no longer has to have hard coded methods for each property type. Instead, you pass in the name of a property and get back a value. Simple.

Note: `getProperties()` is a method defined by the book. I prefer going the `getKeys()` route… as it doesn't expose the internal implementation of the class. I would then add a `void setProperty(String, Object)` method

Double Encapsulation (IV)

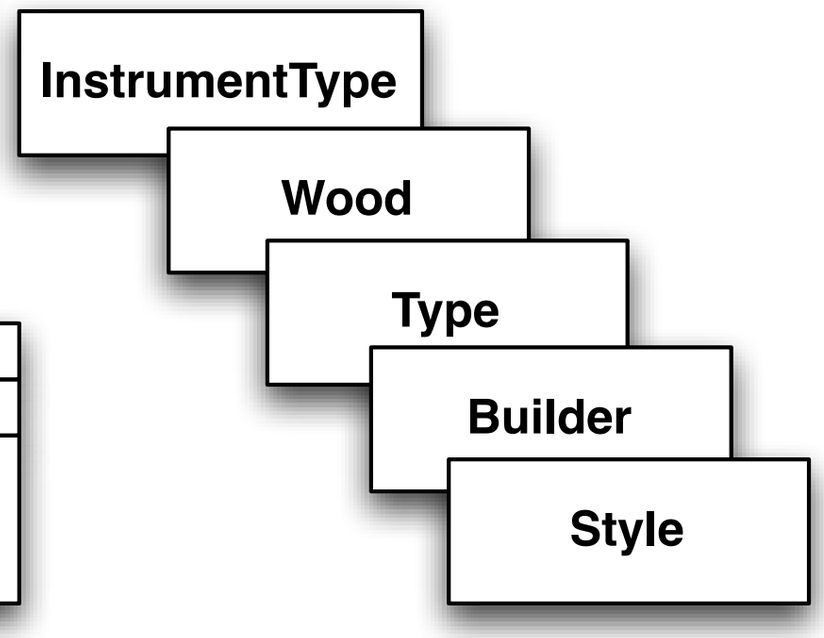
- When you have a set of properties that vary across your objects (i.e. different objects of the SAME class have different properties), use a collection, like a Map, to store those properties dynamically
  - You'll remove lots of methods from your class and avoid having to change your code when new properties are added to your application
- Wrapping up Rick's Application
  - We need to create a new enumeration class called InstrumentType, so we can search for specific instrument types if needed
  - We can also fix addInstrument() now that we no longer have multiple types of InstrumentSpec… the code essentially becomes
    - inventory.add(new Instrument(serialNumber, price, spec));
    - Nice! We no longer need to change this method if we add a new type of instrument to the application

Final Design



Inventory has zero or more Instruments, each of which has a spec that has zero or more properties of various types

Demonstration



Remember This Slide?

How should we confirm our suspicions?

- Having this many qualms about the new design is a serious problem!
- In order to confirm our suspicions, the book has an excellent suggestion
  - Test the design by changing the program once again!
- Rick stops by and says thanks for adding support for Mandolins
  - Now please add support for bass guitars, banjos, dobros and fiddles!!!
- And we discover that in order to do this, we have to make LOTS of changes to our system
  - one new Instrument subclass and one new InstrumentSpec subclass per new instrument
  - two new lines of code in addInstrument() per new instrument
  - one new search method in inventory per new instrument
  - duplication of properties across individual subclasses (numStrings in Banjo) but no easy way to merge duplication into superclasses

To test our new design, we need to try changing the program once again!

Bureau de Change

- How easy is it to change Rick's software?
- Is it well-designed?
- Is it cohesive?
    - And what does that mean again?

- Lets try adding a new instrument type: Fiddles
    - How many classes did you have to add to support Fiddles?
    - How many classes did you have to change to support Fiddles?
    - How many classes need to change to support a new property?
        - How about a property that accesses one of the existing enum types?

Answers

- How many classes did you have to add to support Fiddles?
  - None!
- How many classes did you have to change to support Fiddles?
  - One: we need to change InstrumentType to have a value for Fiddle
- How many classes need to change to support a new property?
  - None, in most cases
- How about a property that accesses one of the existing enum types?
  - At most one, you may need to add a new value or values to the associated enum type. Example: A neckWood property might require changes to the Wood enumerated type
- Test Passed: Software is VERY easy to change
  - at least for the one way that it is LIKELY to change

Cohesive Classes

- A **cohesive** class **does one thing really well** and does not try to do or be something else
- Cohesive classes are focused on specific tasks
    - Inventory focuses on inventory-related tasks, not types of woods, or prices, etc.
- The more cohesive your classes are, the higher the cohesion of your software
    - **Cohesion measures the degree of connectivity among the elements of a single class.** The higher the cohesion of your software is, the more well-defined and related the responsibilities of each class in your application. Each class has a very specific set of closely related actions it performs
- “One Reason to Change” ⇒ Cohesive Class
    - If one class is made up of functionality that’s all related, then it has only one reason to change!

Are we done?

- Each time we revise our system, we want to improve its cohesion
  - This leads to flexibility and reusability and also promotes low coupling
  - But when do stop working on a design?
- Great software is usually about being good enough
  - Make sure the customer is happy
  - Make sure your design is flexible (for likely types of change)
  - And call it quits; move on to the next project!
- Otherwise, you can get stuck endlessly tweaking your software
  - if you've started to improve the design of a system
    - to handle unlikely changes or affect rarely used sections of the code
    - then it might be time to walk away

Wrapping Up (I)

- Tools for your Toolbox: Analysis and Design
  - Well-designed software is easy to change and extend
  - Use basic OO principles like encapsulation and inheritance to make your software more flexible
  - If a design isn’t flexible, change it
  - Make sure each of your classes is cohesive
  - Always strive for higher cohesion as you work on a software system

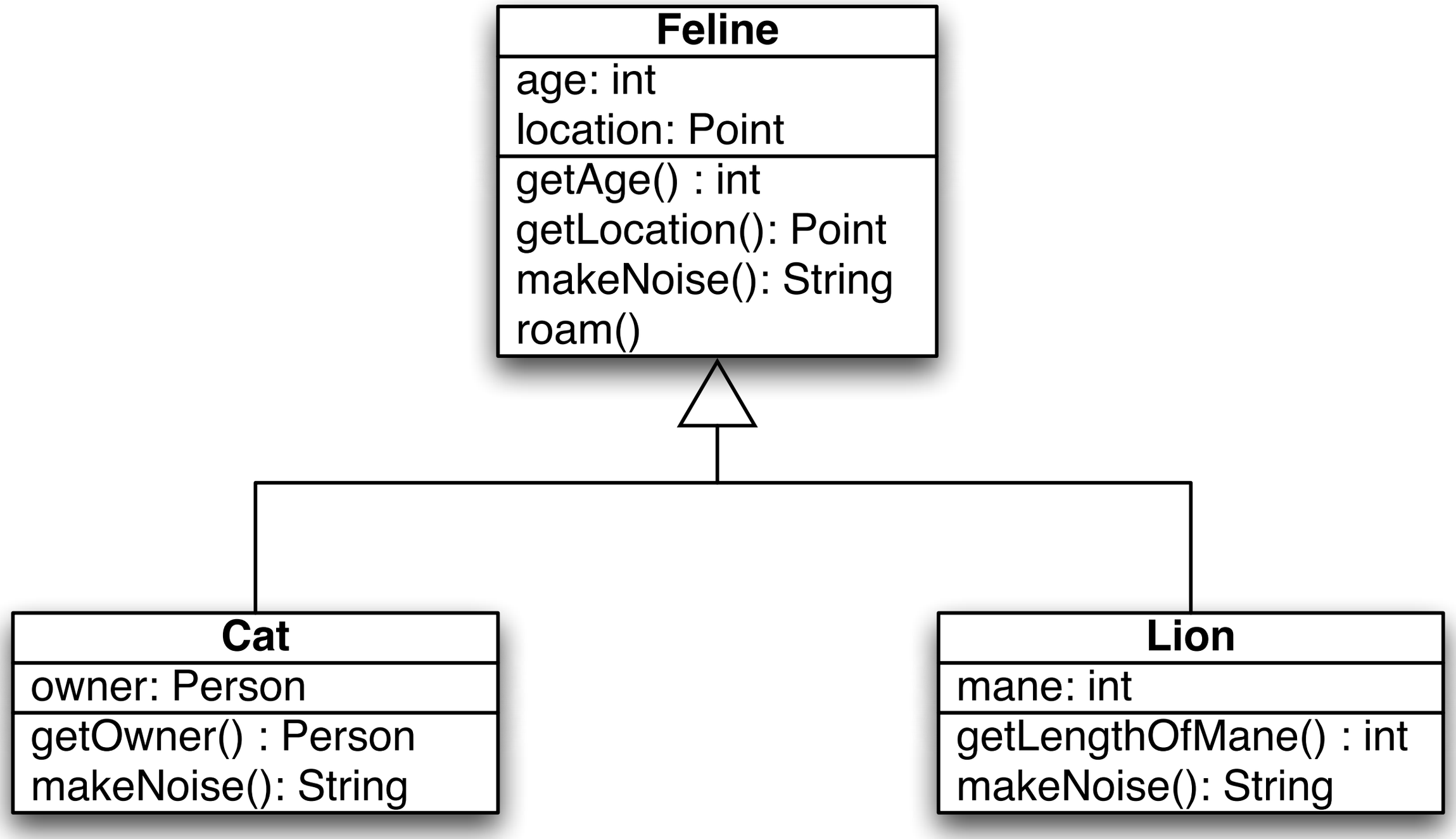
Wrapping Up (II)

- OO Principles
    - Encapsulate what varies
    - Code to an interface
    - One Reason to Change
    - Classes are about Behavior

Ken’s Corner

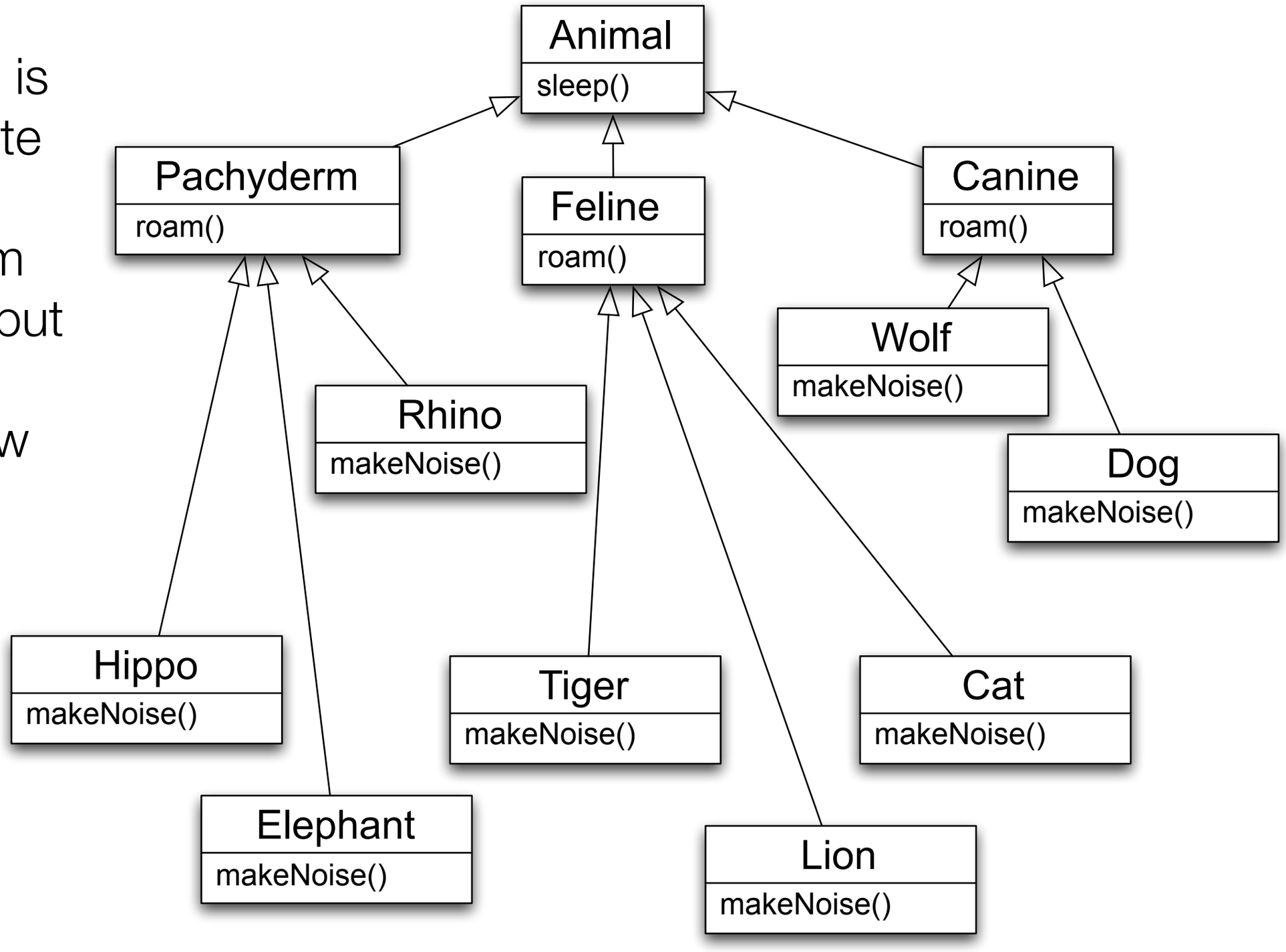
- Another Web Comic: The Order of the Stick
  - <http://www.giantitp.com/comics/oots0001.html>
  - This is really only for people into playing role-playing games…
- Homework 1 Discussion
  - Feline Class Hierarchy
  - Inheritance vs. Delegation
  - Code to an Interface (Shape, in this case)

Ken's Corner (II)

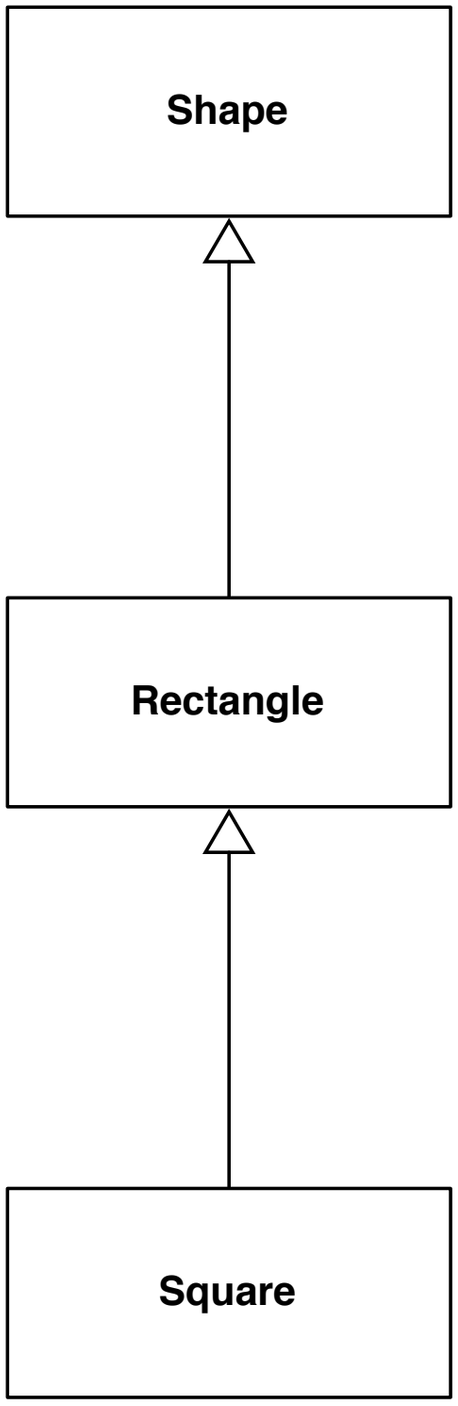


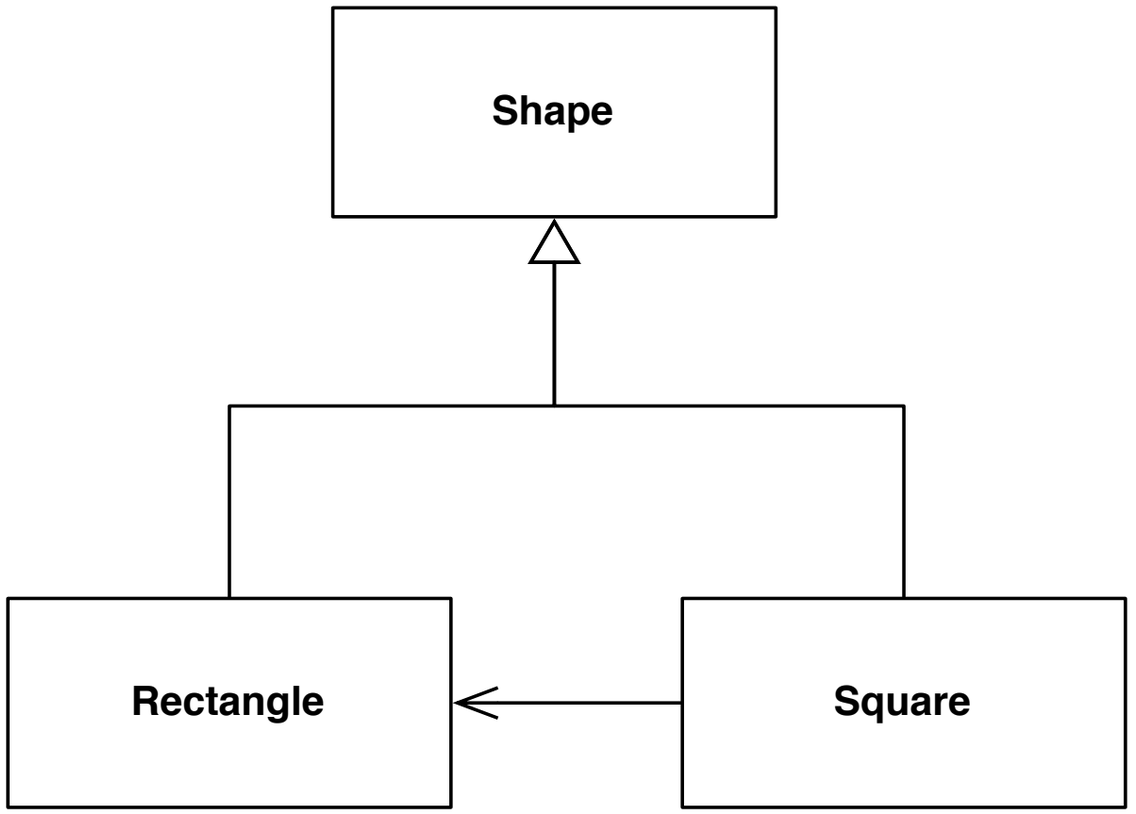
Ken's Corner (III)

This diagram from Lecture 4 is NOT a complete answer to question 3 from Homework 1; but it serves as an example of how to start.



Ken’s Corner (IV)





Ken’s Corner (V)

- Imagine the following code with respect to the inheritance-based approach
  - for r in rectangles:
    - r.setWidth(5)
    - r.setHeight(10)
    - We would now expect area to be 50 for each rectangle
      - but someone got sneaky and stuck a few squares into the rectangles collection class and now some of the shapes have an area of 100
- Conclusion: Squares do not behave the same way as Rectangles
  - As a result, Square IS-A Rectangle doesn’t make sense for this situation
- In the model that uses composition/delegation, there is no expectation that rectangles and squares will behave in the same way; plus we can still gain benefits of code reuse via composition that we saw with the first approach

Coming Up Next

- Lecture 11: Solving Really Big Problems and Bringing Order to Chaos
  - Read Chapter 6 & 7 of the OO A&D book
- Lecture 12: Originality is Overrated
  - Read Chapter 8 of the OO A&D book