Good Design == Flexible Software

Kenneth M. Anderson
University of Colorado, Boulder
CSCI 4448/6448 — Lecture 9 — 09/25/2007

Lecture Goals

- Review material from Chapter 5 Part 1 of the OO A&D textbook
    - Good Design == Flexible Software
    - The problem of “It seemed like a good idea at the time”
    - Discuss the Chapter 5 Example: Rick’s Guitars, Revisited
    - Emphasize the OO concepts and techniques encountered in Chapter 5

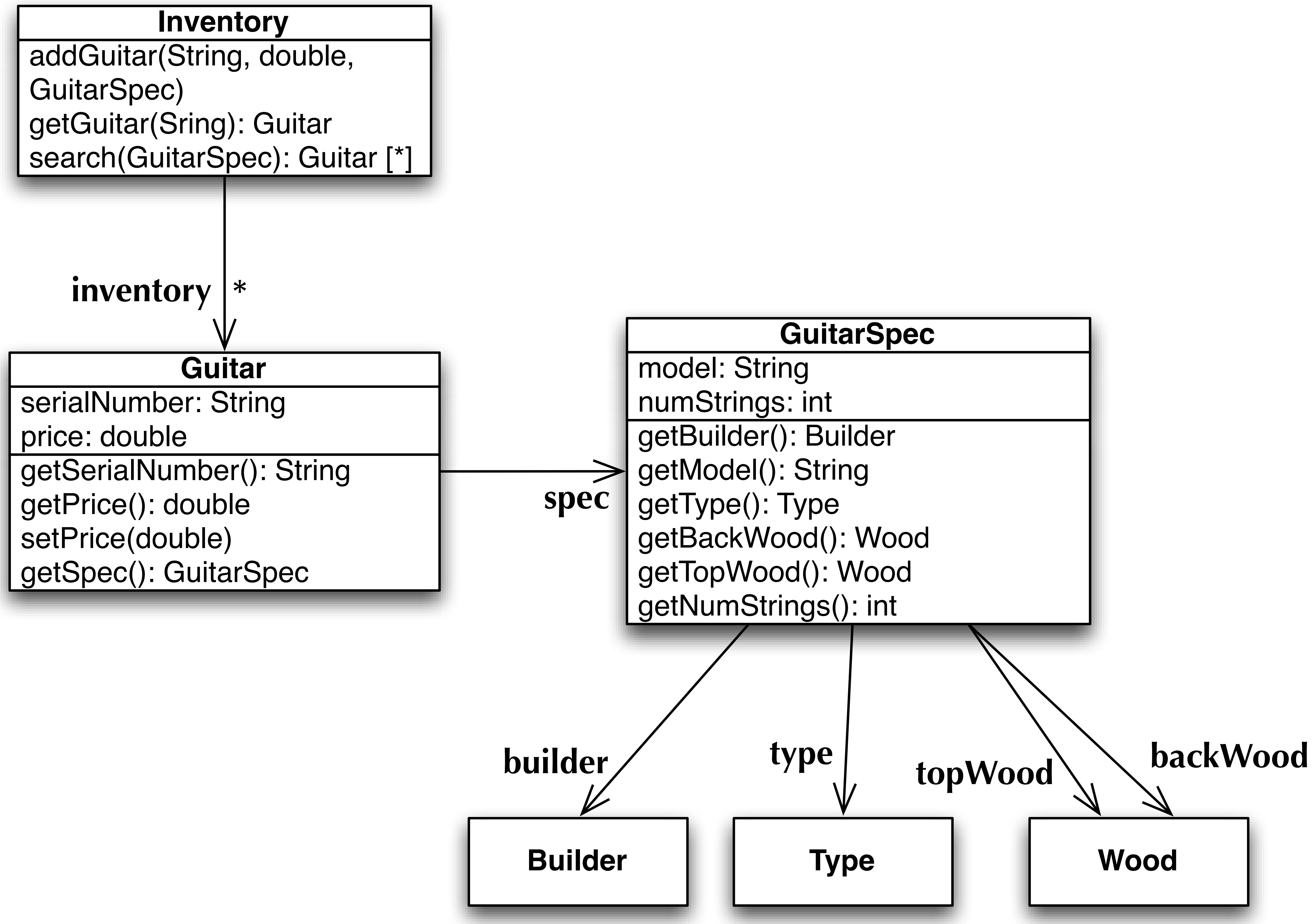
Chapter 5 Overview

- Main Points
  - Change in software development is **inevitable**
  - In order to handle change, you need flexible software
    - In particular, you need **to design** your system **to be flexible** for the most common types of change that it will encounter
      - Designing flexibility for infrequent change is counterproductive
  - Unfortunately, achieving flexible designs “the first time” is really hard
    - And, typically, only possible after acquiring experience with a domain
  - Without experience, small changes can turn into big problems!

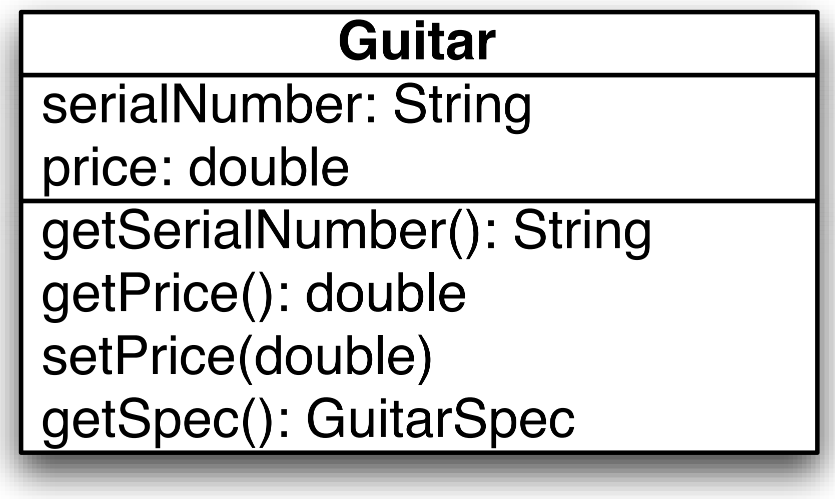
Rick is Back

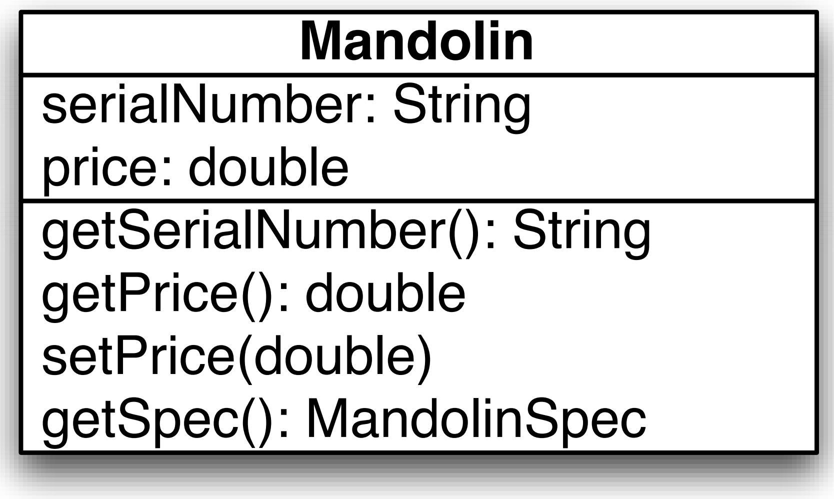
- The software application that we produced for Rick back in Chapter 1 has been working great…
  - BUT… Rick would like to start carrying mandolins alongside guitars
- Lets look at the original design and talk about how to add support for Mandolins

Original Design (circa End of Chapter 1)



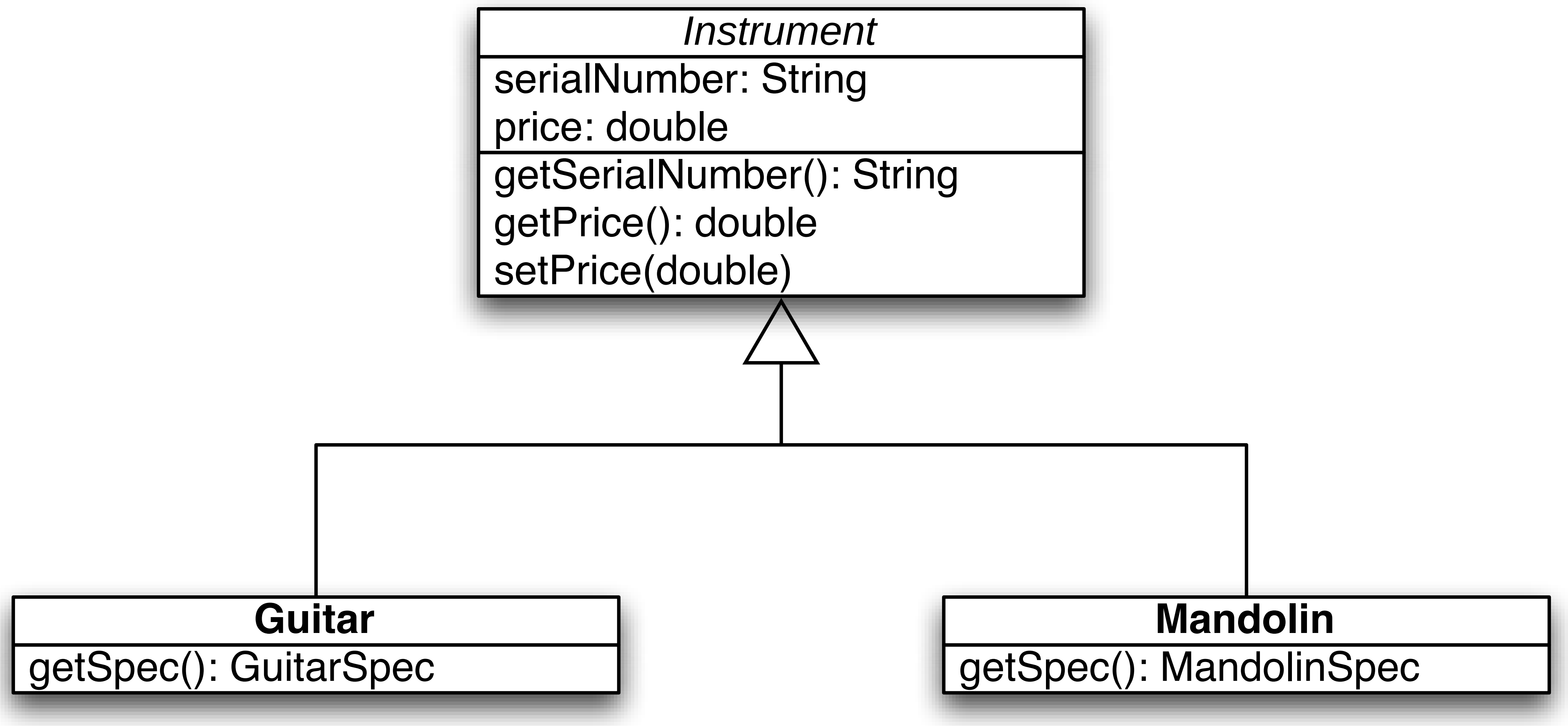
How to add a Mandolin?





These classes are very similar.
What should we do?

Remove Duplication Via Inheritance



Wow!

But why is the Instrument class name in italics?

Abstract Classes (I)

- Instrument is an abstract class
    - UML indicates an abstract class by setting its class name in italics
        - Too subtle for my taste… you can also add a stereotype with the value «abstract» under a bold classname to indicate the same thing
- Abstract classes are placeholders for actual implementation classes
    - You can't instantiate an abstract class directly
        - Recall that we talked about abstract classes when we discussed the concept of "design by contract"
        - The abstract class, typically, defines behavior and the subclasses implement that behavior

Abstract Classes (II)

- In this instance, Instrument provides method bodies for all of its methods, but you still don’t want to instantiate it directly
    - However, there may be differences in behavior between Guitars and Mandolins
        - Those behaviors will live in the respective subclasses
- In the previous diagram, Instrument is known as a **base class** for Mandolin and Guitar… as the book says “they **base** their behavior off of it”, and then extend it as needed to make sense for them
- Again, we make Instrument abstract because we don’t think of it as an entity that can be instantiated. In the real world, you never hold an “instrument” in your hand, you hold trumpets, trombones, flutes, triangles, etc.

Abstract Classes (III)

```java
public abstract class Instrument {

    private String serialNumber;
    private double price;

    public Instrument(String serialNumber, double price) {
        this.serialNumber = serialNumber;
        this.price = price;
    }

    public String getSerialNumber() {
        return serialNumber;
    }

    public double getPrice() {
        return price;
    }

    public void setPrice(double price) {
        this.price = price;
    }

    public abstract InstrumentSpec getSpec();

}
```

Note use of **abstract** keyword in class definition and method definition

What's an **InstrumentSpec** and where's the method body?

Abstract Classes (IV)

- The method
  - public abstract InstrumentSpec getSpec();
- is an example of an abstract class defining behavior that **MUST** be implemented by its subclasses
  - If a subclass of Instrument does not provide a method body for the getSpec() method, then it has to be declared abstract as well
  - Don’t be alarmed by the requirement of returning an InstrumentSpec, through the use of substitutability, we can return an instance of InstrumentSpec OR any of its subclasses
    - This implies that GuitarSpec is going to become a subclass of InstrumentSpec (a class we need to create)

Updating Instrument

```java
public abstract class Instrument {

    private String serialNumber;
    private double price;
    private InstrumentSpec spec;

    public Instrument(String serialNumber, double price, InstrumentSpec spec) {
        this.serialNumber = serialNumber;
        this.price = price;
        this.spec  = spec;
    }

    public String getSerialNumber() {
        return serialNumber;
    }

    public double getPrice() {
        return price;
    }

    public void setPrice(double price) {
        this.price = price;
    }

    public InstrumentSpec getSpec() {
        return spec;
    }

}

```

The book implements Instrument like this.

What does this imply about the Guitar and Mandolin classes?

Guitar and Mandolin as Instrument Subclasses

```java
public class Guitar extends Instrument {
    public Guitar(String serialNumber, double price, GuitarSpec spec) {
        super(serialNumber, price, spec);
    }
}

public class Mandolin extends Instrument {
    public Mandolin(String serialNumber, double price, MandolinSpec spec) {
        super(serialNumber, price, spec);
    }
}

```

Guitar and Mandolin are “empty” classes; all they do is define new types, no new behaviors!

All the heavy lifting is being performed by Instrument.

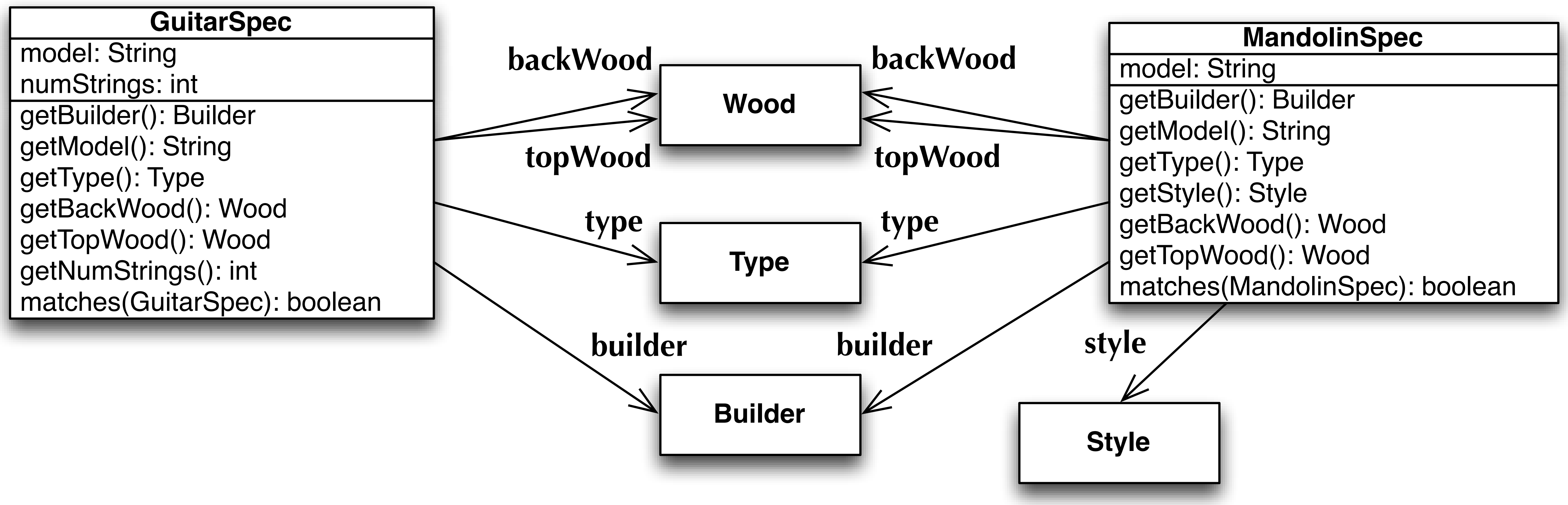
This code is slightly different than the class diagram on slide 7, however. I would implement these classes differently. See next slide.

Guitar and Mandolin Classes, Take 2

```
public class Guitar extends Instrument {

    public Guitar(String serialNumber, double price, GuitarSpec spec) {
        super(serialNumber, price, spec);
    }

    public GuitarSpec getSpec() {
        return (GuitarSpec)super.getSpec();
    }
}

public class Mandolin extends Instrument {

    public Mandolin(String serialNumber, double price, MandolinSpec spec) {
        super(serialNumber, price, spec);
    }

    public MandolinSpec getSpec() {
        return (MandolinSpec)super.getSpec();
    }

}

```

This code matches the class diagram and ensures that Guitars return GuitarSpecs and Mandolins return MandolinSpecs.

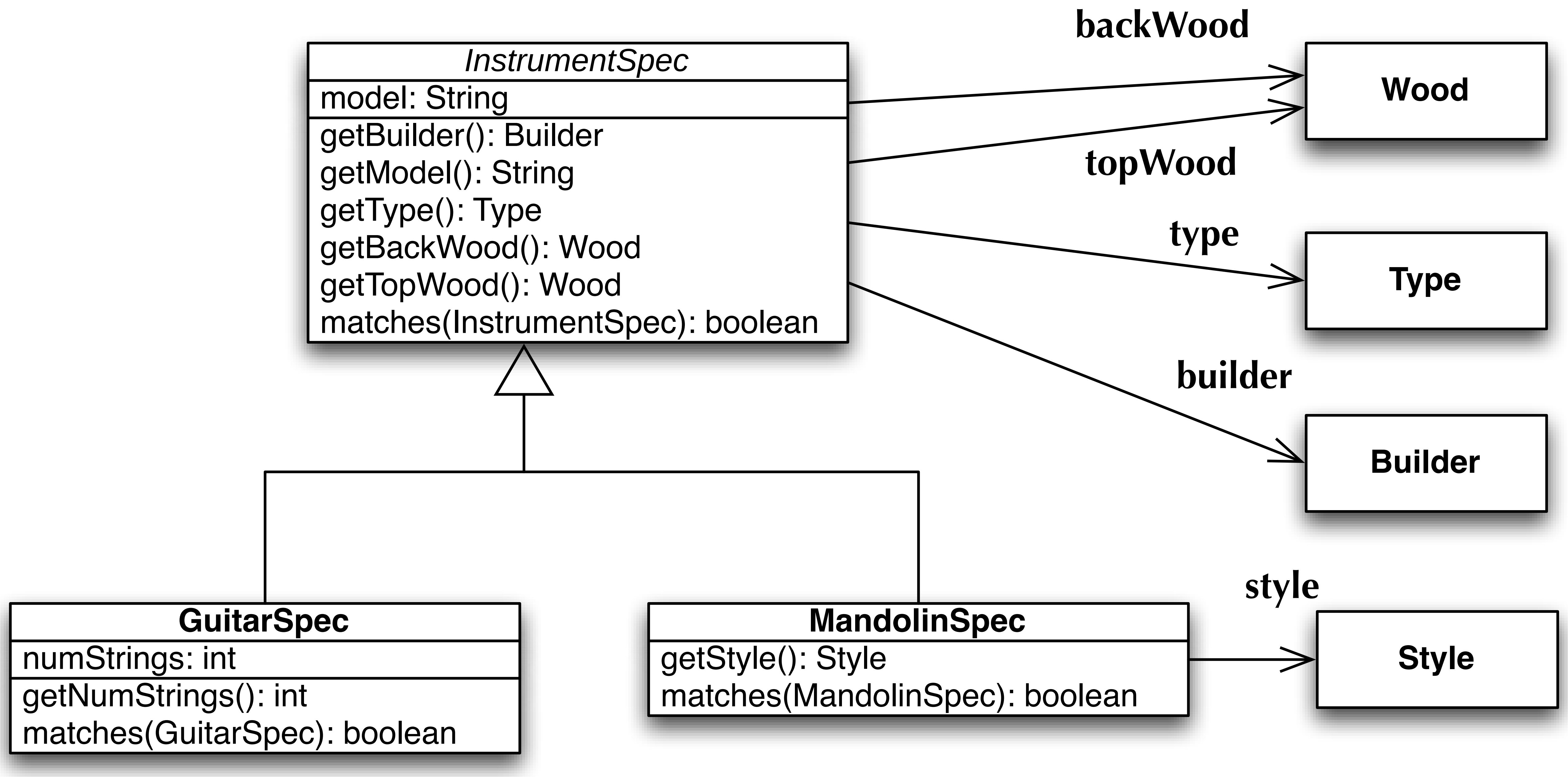
Moving On… GuitarSpec and MandolinSpec



Wow! That's a lot of duplication.

Time for another abstract base class!

InstrumentSpec to the Rescue



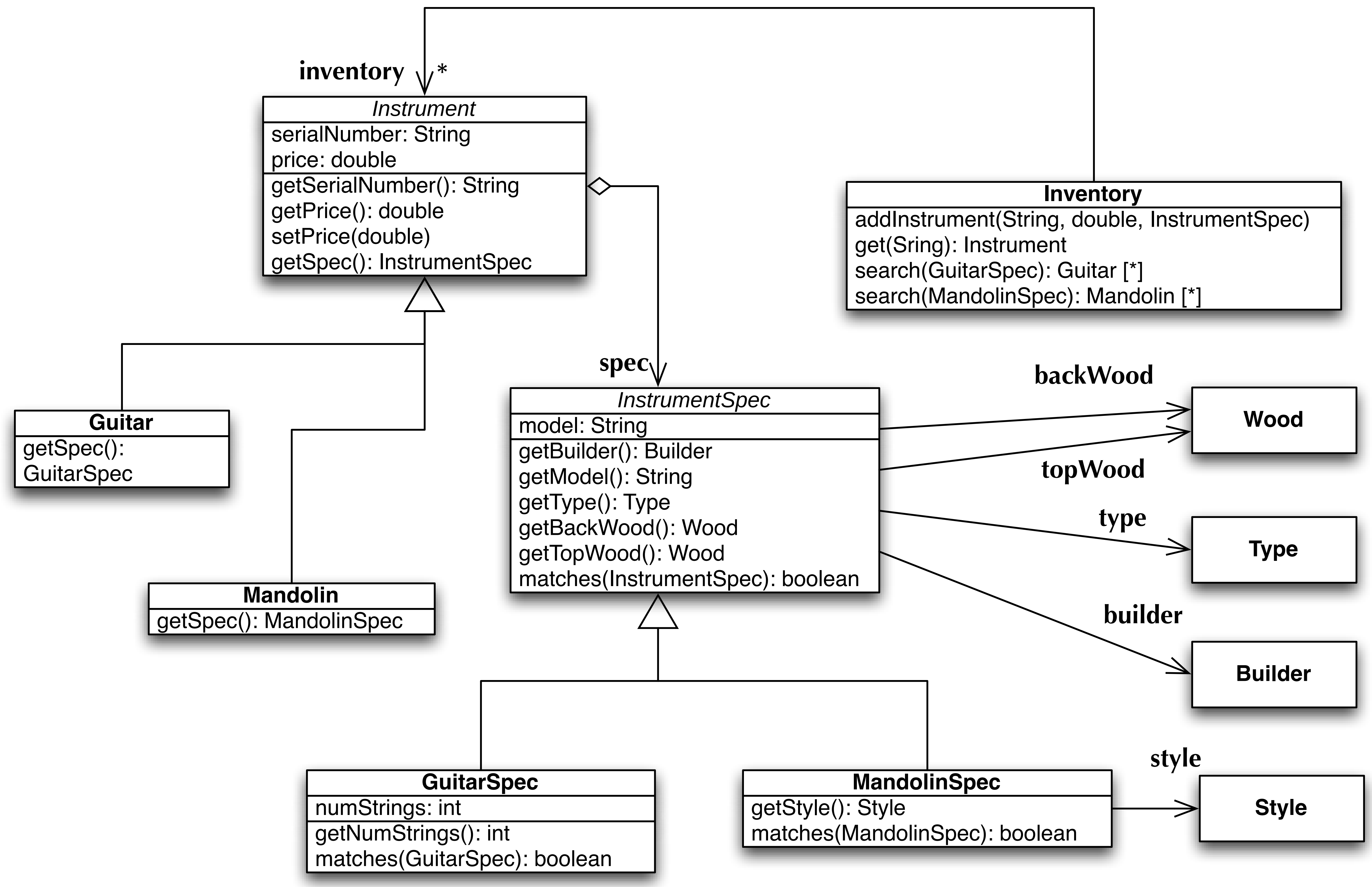
matches() in new design

- The nice thing about this new design is that the method body for matches() in InstrumentSpec takes care of comparing the builder, model, type, and wood attributes for both Guitar and Mandolin
- The GuitarSpec matches() method invokes the superclass method and then only has to compare the numString attribute
- Likewise for MandolinSpec except that it only has to compare style attributes across two MandolinSpecs
- See code on pages 209-211

Design Principle

- In both cases, with Instrument and InstrumentSpec, we did the following:
  - Whenever you find common behavior (or structure) in two or more places, look to abstract that behavior into a class and then reuse that behavior in the common classes
  - Merging shared behavior between two or more classes into a common superclass (most likely an abstract base class) is a common occurrence in OO analysis and design

Putting It All Together



Discussion

- Several changes to Inventory
  - addInstrument() not addGuitar()
  - “get(): Instrument” not “getGuitar(): Guitar”
  - two search methods, one for Guitars and one for Mandolins
- Use of aggregation between Instrument and InstrumentSpec?
  - The book introduces aggregation in a slightly different way than I did in back in our “object fundamentals” lectures
  - From the book: “Aggregation is a special form of association, and means that one thing is made up (in part) of another thing. So Instrument is partly made up of InstrumentSpec.”
    - This is compatible with my definition: here the association is telling you that one class contains another class. The latter class is considered a part of the former class.

Trouble in Paradise?

- After testing the new application, the book indicates a couple of weird characteristics of the new design
  - Guitar and Mandolin don’t seem to be pulling their weight. They are mainly empty classes content to let Instrument do most of the work
  - addInstrument() in the Inventory class now contains ugly code
    - see next slide
  - multiple search methods: Since Instrument and InstrumentSpec are abstract classes, you can’t instantiate them directly and thus you are forced to create either a GuitarSpec or a MandolinSpec in order to do a search
    - This means you can’t do a search across both types of instruments
  - Plus the design seems to be more tightly coupled than before

addInstrument() code

- Because the third parameter to addInstrument() is an InstrumentSpec, we need to check at run-time what type of specification has been passed in order to add the correct type of Instrument to the Inventory. Like this:

```
public void addInstrument(… InstrumentSpec spec) {
   Instrument instrument = null;
   if (spec instanceof GuitarSpec) {
      instrument = new Guitar(…);
   } else if (spec instanceof MandolinSpec) {
      instrument = new Mandolin();
   }
   inventory.add(instrument);
}
```

- instanceof allows us to determine an object’s type at run-time. Unfortunately, this means that each time we add a subclass to Instrument, this code has to change. Yuck!

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

Design Heuristic Violated

- All of this work to add a new Instrument to our design indicates that something is not right
  - When working on Mandolin, each decision that we made “seemed right at the time”
    - but when we finished the design, we could see that similar changes would just exacerbate the problems we noticed with the new design
- When you find yourself in a situation like this, you need to reexamine the decisions that you made closely
  - You may see new ways in which “things can vary” and thus need to be encapsulated. We need to develop a new design in which adding a new instrument to the system is easy to do!
- After all, Rick has demonstrated that he likes to add new instruments all the time!

OO Catastrophe!

- The book now reviews concepts that will play a critical role in helping to solve the problems we encountered in the new design of Rick's application
- These concepts are
  - Code to an Interface
  - Encapsulate What Varies
  - Each Class has only One Reason to Change

Code to an Interface

- Coding to an interface, rather than to an implementation makes your software easier to extend
  - Example of an Athlete interface with lots of different implementations
    - FootballPlayer, BaseballPlayer, HockeyPlayer, …
  - When dealing with these classes, you can either choose to code to one of the subclasses directly or to the common interface
    - The latter choice leads to more flexible code that is easier to extend
      - This is similar to my argument about coding to the root of a class hierarchy. That code continues to work no matter how many subclasses you add to the hierarchy
    - In this case, interface-specific code can handle all of the implementing classes uniformly. And does not need to change if another class decides to implement the interface

Encapsulate What Varies

- Encapsulation, aka information hiding, has several benefits
  - It can reduce duplicated code (as seen with Instrument and its subclasses)
  - but, more importantly, it can protect classes from unnecessary changes
- Anytime you have behavior in your application that is likely to change, move it away from parts of your application that is unlikely to change
  - In other words, encapsulate what varies
- Painter class with prepareEasel(), cleanBrushes(), and paint() methods
  - The first two are unlikely to change, the latter might change frequently
- As a result, create a PaintStyle() base class with an abstract paint() method
  - Allow subclasses to specify particular styles: Modern, Cubist, Surreal, …
  - Associate the Painter class with a particular style and change it as needed
  - This is our first encounter with the **Strategy** design pattern

Each Class has only One Reason to Change

- Since change in a software system is inevitable:
  - take steps to **minimize** the **impact** of change
- The best way to do this is to make sure that each class has only one reason to change
  - consider the reverse situation
    - if a particular class has five ways in which it can change, it has a greater chance of needing to change when any given change request comes in than classes that have only one reason to change
- Automobile example with methods start(), stop(), changeTires(), drive(), wash(), checkOil(), etc.
  - Class can be split up to isolate potential changes
    - such as different driving styles, washing styles, approaches to changing the oil and tires, etc.

Wrapping Up

- These three heuristics will aid us in our goal of addressing the problems discovered in the new design
  - The key point of this chapter is to demonstrate how easy it is to go down the wrong design path
  - how looking at one change in isolation, our decisions can seem sensible and “right at the time”
  - reflecting on whether we would want to make similar changes using the same process can help to identify problems early
    - “Do I really want to change addInstrument() each time I add a new type of instrument to the system?”
- As we will see, the three heuristics will lead us to a design in which classes are more cohesive and less coupled than the current design

Coming Up Next

- Lecture 10: Flexible Software
  - Read Chapter 5 (part 2) of the OO A&D book
- Lecture 11: Solving Really Big Problems
  - Read Chapter 6 of the OO A&D book