Good Design == Flexible Software

Kenneth M. Anderson
University of Colorado, Boulder
CSCI 4448/5448 — Lecture 9 — 09/22/2009

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

Quiz

- If you need to indicate that a search algorithm of some type will be used to traverse an association between two classes, what notation do you use and what is it called?

- A sequence diagram shows interactions between what?

- What are the dashed lines of a sequence diagram called?

- What do the rectangles that appear above the dashed lines indicate?

Quiz

- If you need to indicate that a search algorithm of some type will be used to traverse an association between two classes, what notation do you use and what is it called?
  - **A rectangle with a list of attributes attached to the class and association; qualification**
- A sequence diagram shows interactions between what?
  - **objects (not classes)**
- What are the dashed lines of a sequence diagram called?
  - **lifelines**
- What do the rectangles that appear above the dashed lines indicate?
  - **flow of control; It indicates the object is actively doing something**

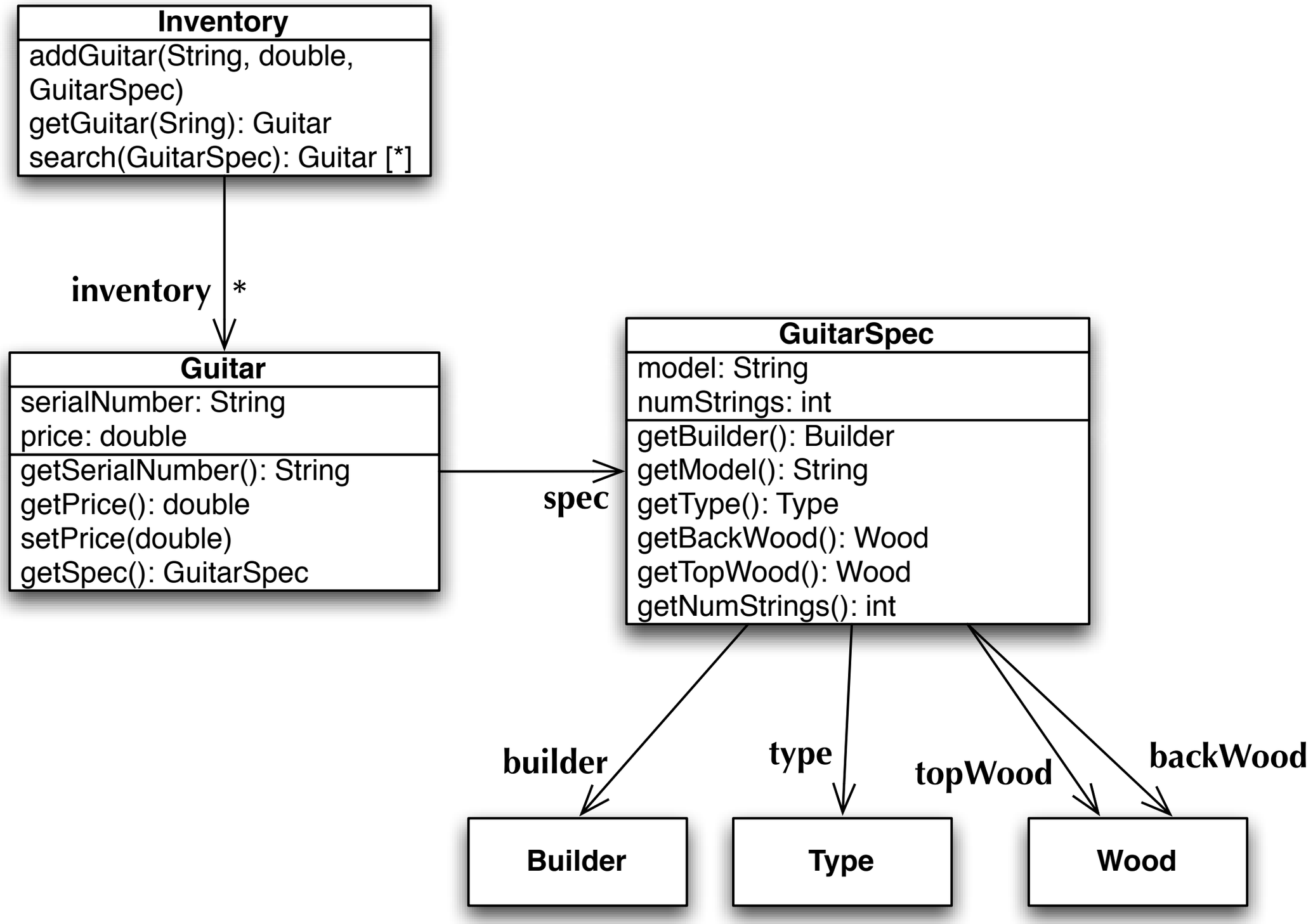
Chapter 5 Overview

- 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)



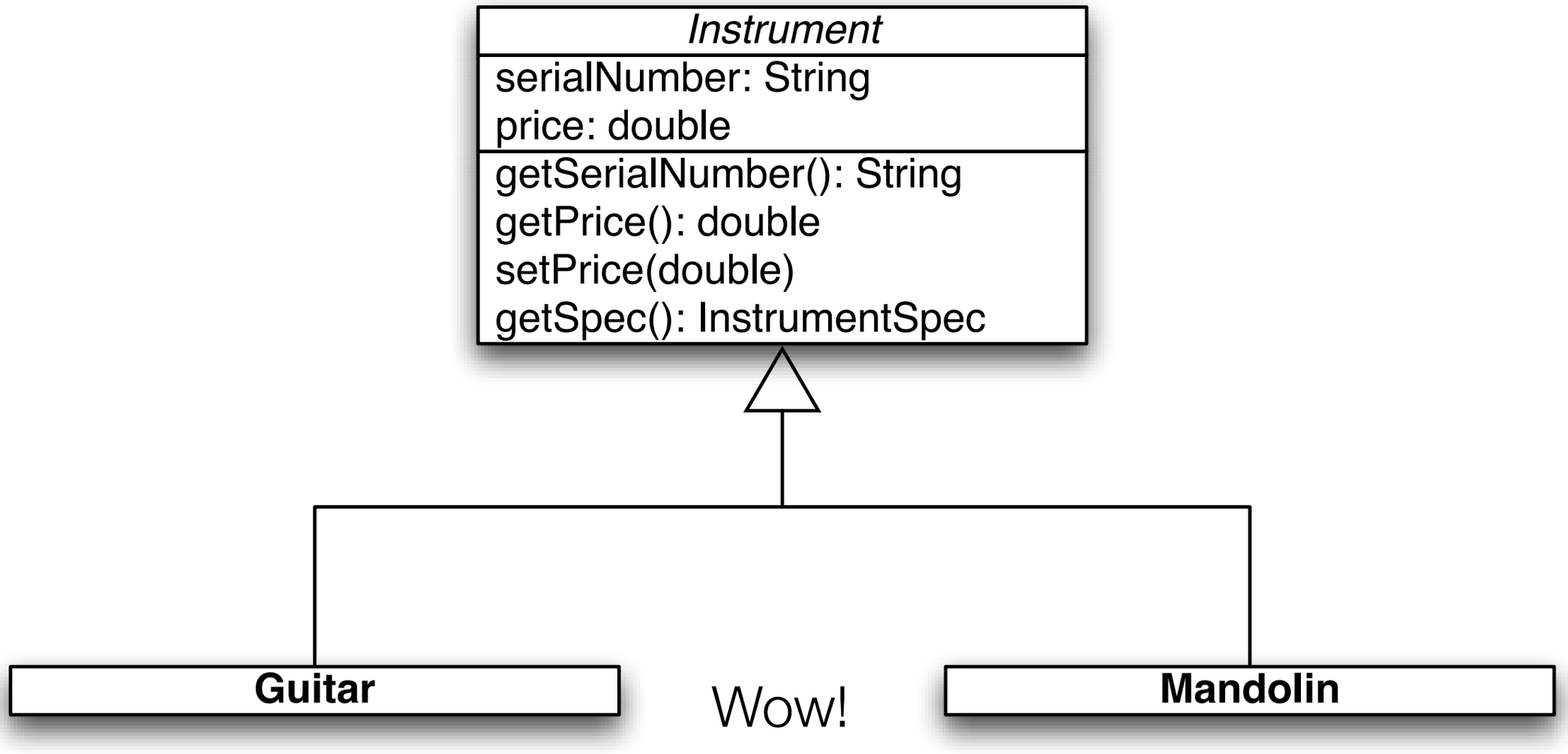
How to add a Mandolin?

Guitar
serialNumber: String price: double
getSerialNumber(): String getPrice(): double setPrice(double) getSpec(): GuitarSpec

Mandolin
serialNumber: String price: double
getSerialNumber(): String getPrice(): double setPrice(double) getSpec(): MandolinSpec

These classes are very similar. What should we do?

Remove Duplication Via Inheritance



Note: Instrument class name is in italics since it is an abstract base class

Also notice: getSpec() has moved to the superclass
and its return type has changed. More on this later.

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 class name 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 defines what behavior its “category of objects” will provide and then subclasses implement that behavior
      - The abstract class can provide default behaviors (code reuse)

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 (eventually) 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

- Also, 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)

```
1  public abstract class Instrument {
2
3      private String serialNumber;
4      private double price;
5
6      public Instrument(String serialNumber, double price) {
7          this.serialNumber = serialNumber;
8          this.price = price;
9      }
10
11     public String getSerialNumber() {
12         return serialNumber;
13     }
14
15     public double getPrice() {
16         return price;
17     }
18
19     public void setPrice(double price) {
20         this.price = price;
21     }
22
23     public abstract InstrumentSpec getSpec();
24
25 }
26
```

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)
- As we will see, however, we are going down the wrong design path with these decisions

Updating Instrument

```
1  public abstract class Instrument {
2
3      private String serialNumber;
4      private double price;
5      private InstrumentSpec spec;
6
7      public Instrument(String serialNumber, double price, InstrumentSpec spec) {
8          this.serialNumber = serialNumber;
9          this.price = price;
10         this.spec  = spec;
11     }
12
13     public String getSerialNumber() {
14         return serialNumber;
15     }
16
17     public double getPrice() {
18         return price;
19     }
20
21     public void setPrice(double price) {
22         this.price = price;
23     }
24
25     public InstrumentSpec getSpec() {
26         return spec;
27     }
28
29 }
30
```

The book implements Instrument like this.

What does this imply about the Guitar and Mandolin classes?

Guitar and Mandolin as Instrument Subclasses

```
1  public class Guitar extends Instrument {
2      public Guitar(String serialNumber, double price, GuitarSpec spec) {
3          super(serialNumber, price, spec);
4      }
5  }
6
7  public class Mandolin extends Instrument {
8      public Mandolin(String serialNumber, double price, MandolinSpec spec) {
9          super(serialNumber, price, spec);
10     }
11 }
12
```

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 change was foreshadowed by the class diagram on slide 7.

Note: the need for InstrumentSpec; We have guitars creating GuitarSpecs and Mandolins creating MandolinSpecs; Since we have to store both in the superclass, we need a new (more general) type to do it.

What is the relationship to Instrument.getSpec()?

```
1  public class Guitar extends Instrument {
2
3      public Guitar(String serialNumber, double price, GuitarSpec spec) {
4          super(serialNumber, price, spec);
5      }
6
7      public GuitarSpec getSpec() {
8          return (GuitarSpec)super.getSpec();
9      }
10 }
11
12 public class Mandolin extends Instrument {
13
14     public Mandolin(String serialNumber, double price, MandolinSpec spec) {
15         super(serialNumber, price, spec);
16     }
17
18     public MandolinSpec getSpec() {
19         return (MandolinSpec)super.getSpec();
20     }
21
22 }
23
```

Overloading versus Overriding; How to override?

- In order for a method in a subclass to override a method in a superclass, the following must be true:
  - it must have the same
    - return type
    - name
    - parameter list
- it must return the same exceptions (or legal subclasses of the superclasses declared exceptions)
- The superclass method cannot be static
- The subclass method cannot have weaker access rights
  - A public method cannot be made private, for example

So what went wrong?

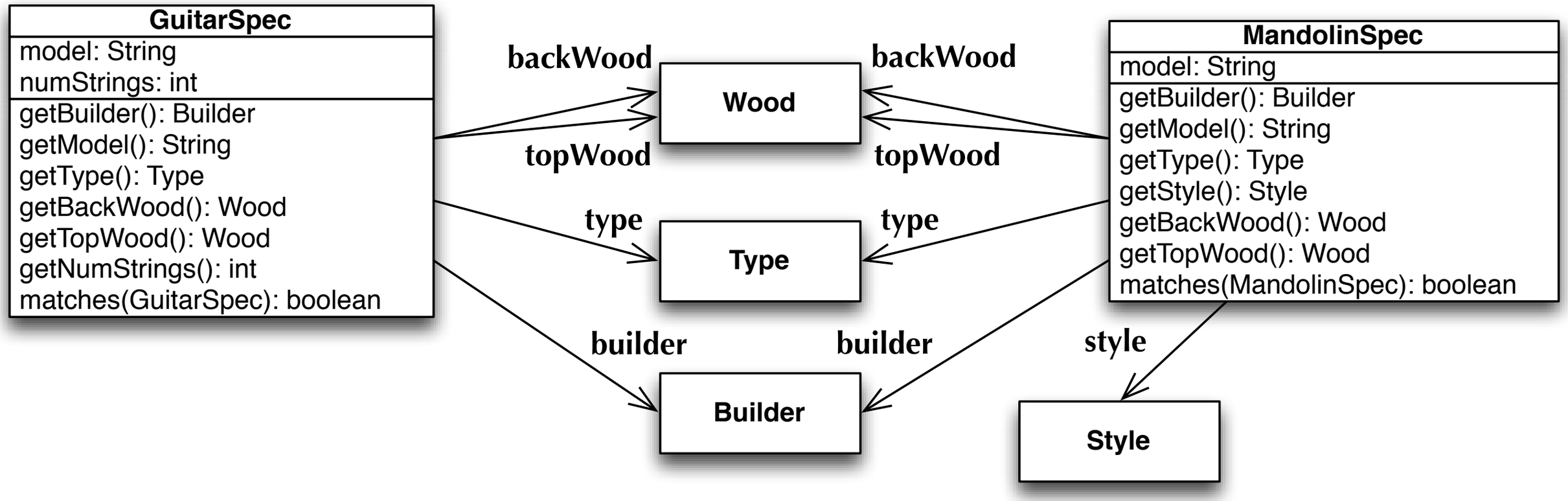
- When Instrument defined
  - InstrumentSpec getSpec()
- And Guitar defined
  - GuitarSpec getSpec()
- In versions of Java prior to J2SE 5, Guitar was **overloading** the getSpec method. In this case, it was **adding a new method with the same name but different return type**
  - Prior to J2SE5, the important thing to realize is that there is **no polymorphism occurring here**; you're just making a direct call to the method in the object you point at based on the type of your variable
- Note: from J2SE 5 and on, the subclass methods do in fact **override** the method in the superclass. This is because J2SE 5 added support for covariant return types. See <http://en.wikipedia.org/wiki/Covariance_and_contravariance_(computer_science)> for details.

To drive this point home

```
for (Foo foo : foos) {
    FooSpec s = foo.getSpec();
    System.out.println("s matches: " + s.matches(bar_spec));
}
```

- This code is taken from the Overloading/Overriding example available on the class website; here a matches() method defined in FooSpec is invoked, even though the actual object being pointed at is a subclass of FooSpec with its own matches method. Since matches() in the subclass only overloaded the matches method in the superclass, the method invoked was the one that matches the type of our variable, in this case FooSpec.

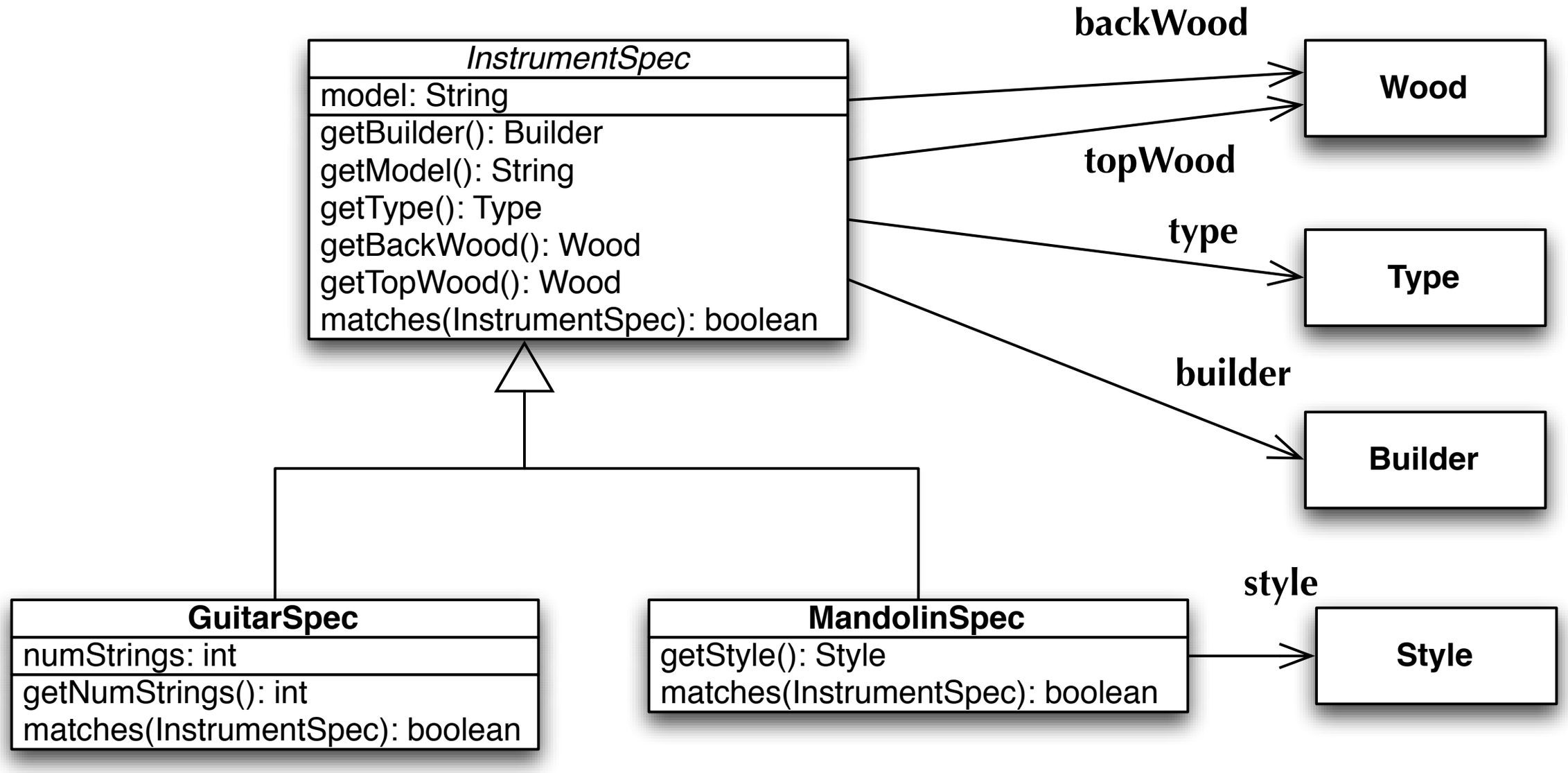
Moving On… GuitarSpec and MandolinSpec



Wow! That's a lot of duplication.
Time for another abstract base class!

Note: in initial versions, the matches methods take type-specific parameters: GuitarSpec and MandolinSpec

InstrumentSpec to the Rescue



Once again, InstrumentSpec is abstract and handles all duplication; note since we want polymorphic behavior for matches(), we need to change the type of the subclass parameters to that method to InstrumentSpec; otherwise we'd be **overloading** that method not **overriding** it

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 GuitarSpec and MandolinSpec
- The GuitarSpec matches() method invokes the superclass method and then only has to compare the numString attribute (specific to Guitars)
    - Likewise for MandolinSpec except that it only has to compare Style attributes (specific to Mandolins)
- See code next slide

matches() in subclass makes use of superclass

```
// InstrumentSpec.matches()
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;
}

// MandolinSpec.matches()
public boolean matches(InstrumentSpec otherSpec) {
  if (!super.matches(otherSpec))
    return false;
  if (!(otherSpec instanceof MandolinSpec))
    return false;
  MandolinSpec spec = (MandolinSpec)otherSpec;
  if (!style.equals(spec.style))
    return false;
  return true;
}
```

But the code in the subclass is a little messy;

since InstrumentSpec is abstract, we have to make sure that we have been handed a MandolinSpec.

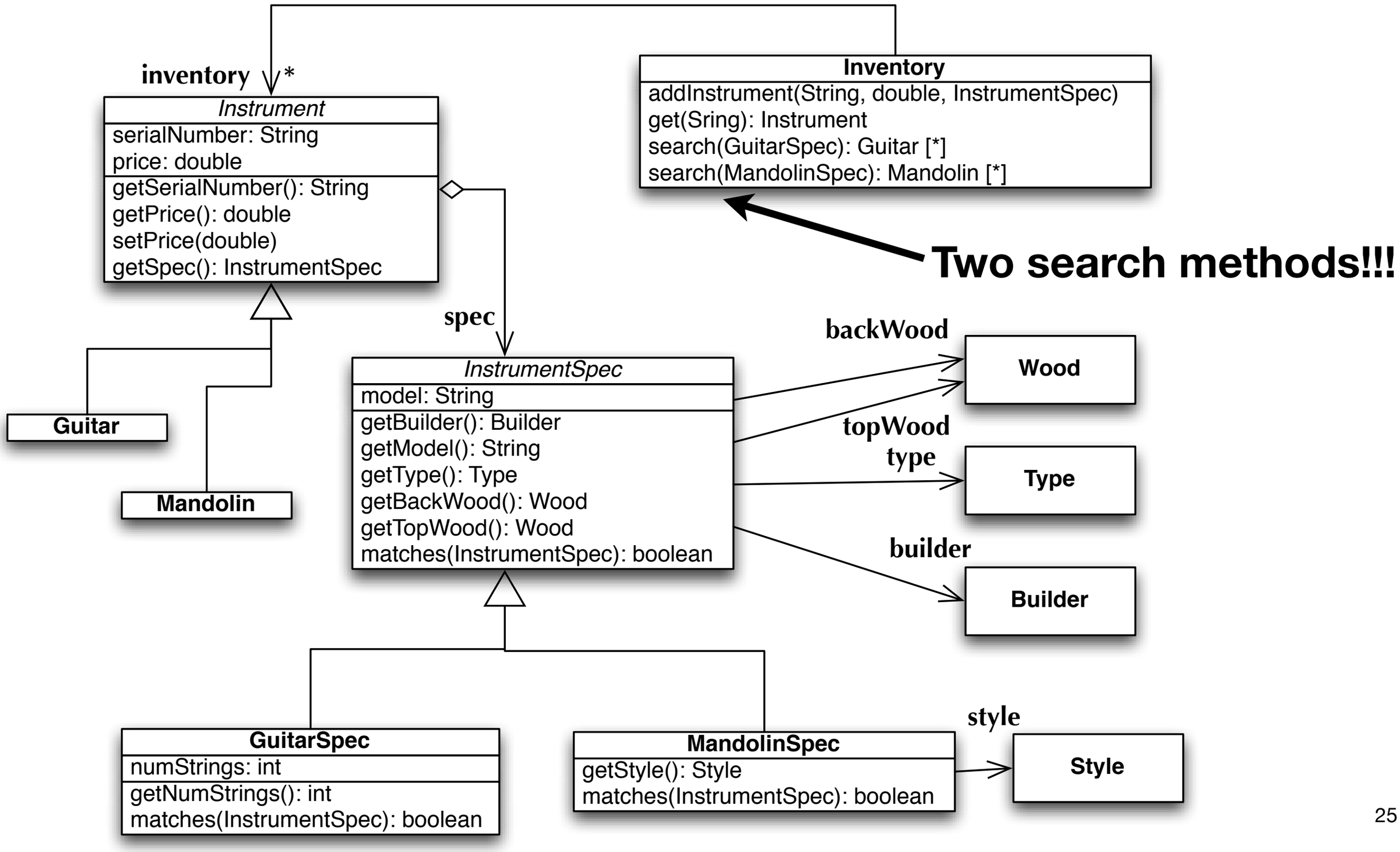
We use Java's instanceof operator to do this comparison. If we find a MandolinSpec in this fashion, we perform a cast and then check the style attribute

This seems logical but its going to break one of Rick's requirements; which one?

The ability to search across all types of instruments

- We can’t instantiate an InstrumentSpec directly
    - So, when we call Inventories search method, we have to choose:
        - We pass in a MandolinSpec and search all Mandolins
    - or
        - We pass in a GuitarSpec and search all Guitars
- A customer won’t be able to say “show me all instruments that have Alder wood in them”
    - We’d have to perform the search twice and then union the results!
- Indeed, take a look at how the book changes the Inventory class

Final Class Diagram



No Test Program!

- This new design is such a mess with respect to Rick's reasonable expectation that a customer be able to search across all instruments that:
    - THEY DIDN'T EVEN BOTHER CREATING A TEST PROGRAM!!!
        - (I created one and made it work by modifying the search method for guitars. You can see that program in ricksGuitars-mandolins)
- Indeed, with the way we have the matches code set-up, you can't do a search on just one attribute. When you pass a spec into the search routine, you have to specify values for all attributes.
    - We could emulate the original program, by having the test program first perform a search for guitars for one customer and then perform a search for mandolins for a second customer, but the searches have to be VERY specific;

Showing the Problem

- I created a directory called ricksGuitars-crash in the example code to show what happens when I add a Mandolin to the inventory and run the test program from Chapter 1 that only searches for guitars

```
Exception in thread "main" java.lang.ClassCastException:
Mandolin cannot be cast to Guitar
  at Inventory.search(Inventory.java:37)
  at FindGuitarTester.main(FindGuitarTester.java:14)
```

Additional Discussion

- Several changes to Inventory
  - addInstrument() not addGuitar()
  - “get(): Instrument” not “getGuitar(): Guitar”
- 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
- (As noted previously) 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 are 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.

Also, its not all bad… New 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
- This approach is not intrinsically bad: it often leads to less code and greater re-use of behavior and data across subclasses
- Its just that in this particular context with these requirements, it turns out to be the WRONG approach

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