CSE203 Object-Oriented Analysis and Design

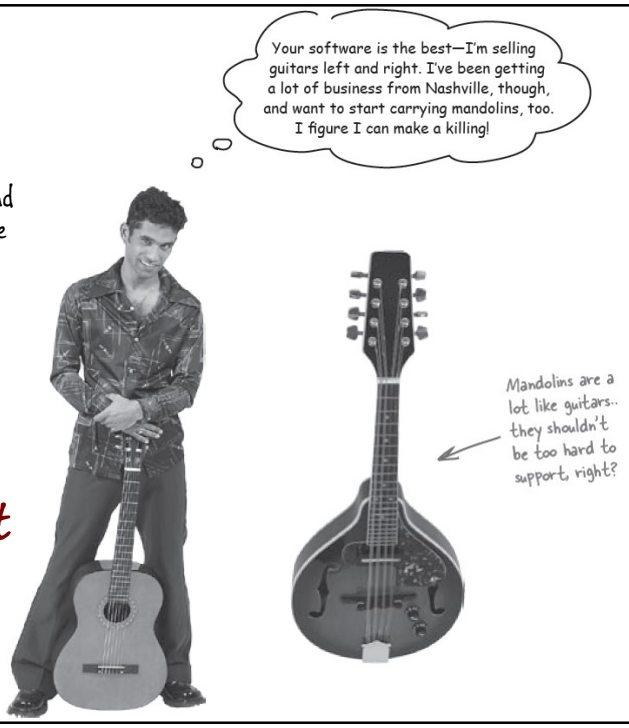
05.1 Good Design = Flexible Software

Nothing Ever Stays the Same

1

Rick's ~~Guitars~~ is expanding Stringed Instruments

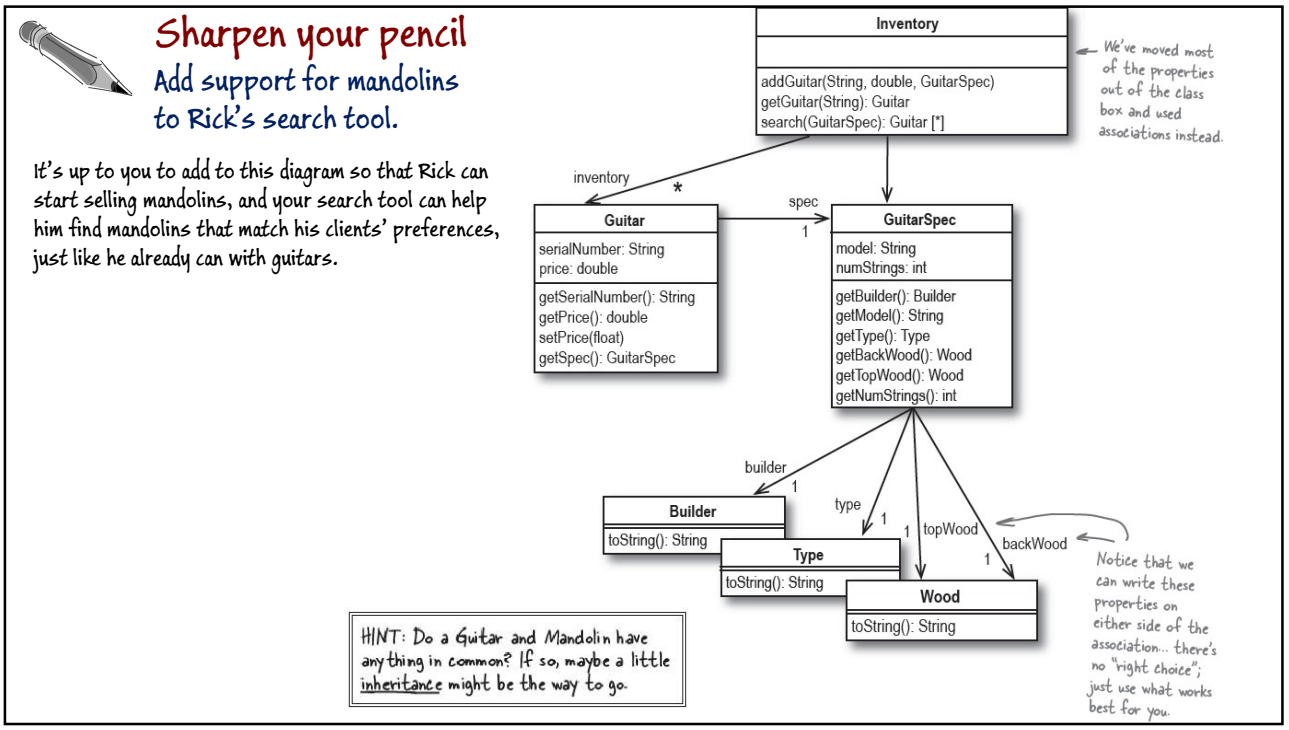
Fresh off the heels of selling three guitars to the rock group Augustana, Rick's guitar business is doing better than ever—and the search tool you built Rick back in Week 1 is the cornerstone of his business.



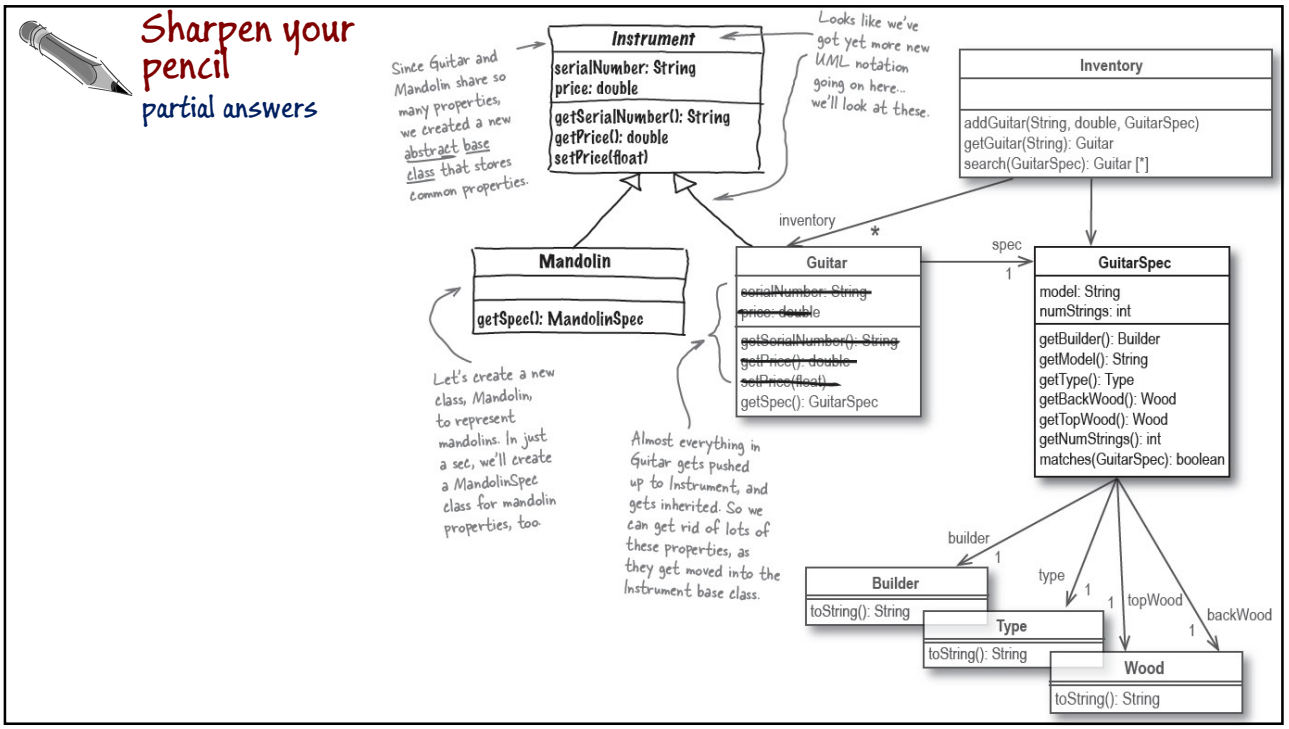
Let's put our design to the test

We've talked a lot about good analysis and design being the key to software that you can reuse and extend... and now it looks like we're going to have to prove that to Rick. Let's figure out how easy it is to restructure his application so that it supports mandolins.

2

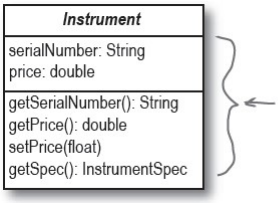

Sharpen your pencil
Add support for mandolins to Rick's search tool.
It's up to you to add to this diagram so that Rick can start selling mandolins, and your search tool can help him find mandolins that match his clients' preferences, just like he already can with guitars.
Inventory
addGuitar(String, double, GuitarSpec)
getGuitar(String): Guitar
search(GuitarSpec): Guitar [*]
We've moved most of the properties out of the class box and used associations instead.
inventory
*
spec
1
Guitar
serialNumber: String
price: double
getSerialNumber(): String
getPrice(): double
setPrice(float)
getSpec(): GuitarSpec
GuitarSpec
model: String
numStrings: int
getBuilder(): Builder
getModel(): String
getType(): Type
getBackWood(): Wood
getTopWood(): Wood
getNumStrings(): int
builder
1
type
1
topWood
1
backWood
1
Builder
toString(): String
Type
toString(): String
Wood
toString(): String
Notice that we can write these properties on either side of the association... there's no "right choice"; just use what works best for you.
HINT: Do a Guitar and Mandolin have anything in common? If so, maybe a little inheritance might be the way to go.

3


Sharpen your pencil
partial answers
Since Guitar and Mandolin share so many properties, we created a new abstract base class that stores common properties.
Instrument
serialNumber: String
price: double
getSerialNumber(): String
getPrice(): double
setPrice(float)
Looks like we've got yet more new UML notation going on here... we'll look at these.
Inventory
addGuitar(String, double, GuitarSpec)
getGuitar(String): Guitar
search(GuitarSpec): Guitar [*]
inventory
*
spec
1
Mandolin
getSpec(): MandolinSpec
Guitar
serialNumber: String
price: double
getSerialNumber(): String
getPrice(): double
setPrice(float)
getSpec(): GuitarSpec
GuitarSpec
model: String
numStrings: int
getBuilder(): Builder
getModel(): String
getType(): Type
getBackWood(): Wood
getTopWood(): Wood
getNumStrings(): int
matches(GuitarSpec): boolean
Let's create a new class, Mandolin, to represent mandolins. In just a sec, we'll create a MandolinSpec class for mandolin properties, too.
Almost everything in Guitar gets pushed up to Instrument, and gets inherited. So we can get rid of lots of these properties, as they get moved into the Instrument base class.
builder
1
type
1
topWood
1
backWood
1
Builder
toString(): String
Type
toString(): String
Wood
toString(): String

4

Did you notice that abstract base class?



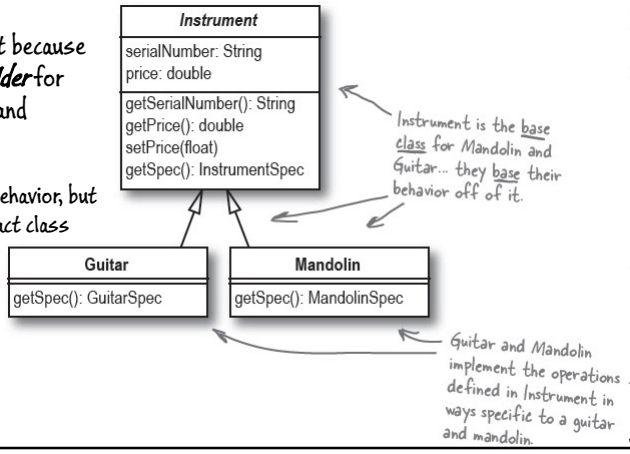
We took all the attributes and operations that are common to both Guitar and Mandolin, and put them in Instrument.

Instrument is an abstract class: that means that you can't create an instance of **Instrument**. You have to define subclasses of **Instrument**, like we did with **Mandolin** and **Guitar**:

We made **Instrument** abstract because **Instrument** is just a ***placeholder*** for actual instruments like **Guitar** and **Mandolin**.

An abstract class defines some basic behavior, but it's really the *subclasses* of the abstract class that add the implementation of those behaviors.

Instrument is just a generic class that stands in for your actual implementation classes.

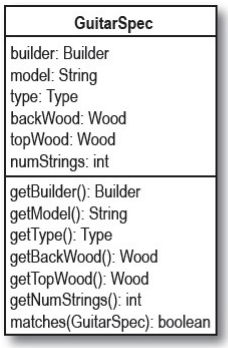


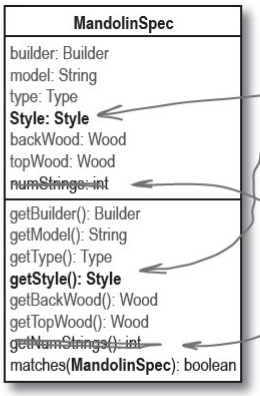
Abstract classes are placeholders for actual implementation classes. The abstract class defines behavior, and the subclasses implement that behavior.

5

We'll need a MandolinSpec class, too

Mandolins and guitars are similar, but there are just a few things different about mandolins... we can capture those differences in a **MandolinSpec** class:

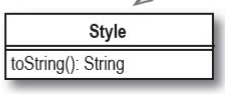


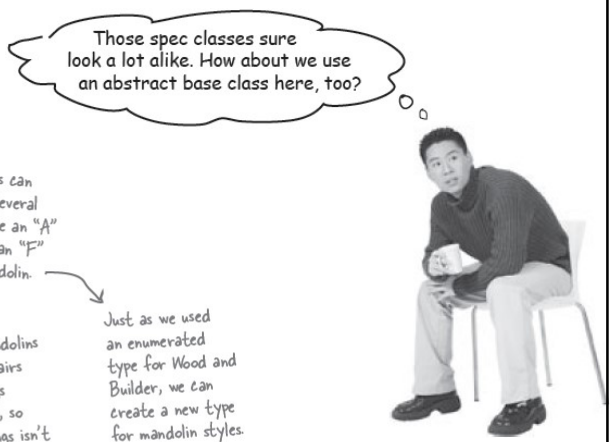


Mandolins can come in several styles, like an "A" style, or an "F" style mandolin.

Most mandolins have 4 pairs of strings (8 total), so numStrings isn't needed here.

Just as we used an enumerated type for Wood and Builder, we can create a new type for mandolin styles.





It's OK if you don't know anything about mandolins, or didn't figure out the different properties in the MandolinSpec class. The main thing is that you realized we probably need a new class for mandolins and their specs. If you came up with using an Instrument interface or abstract class, all the better!

6

there are no Dumb Questions

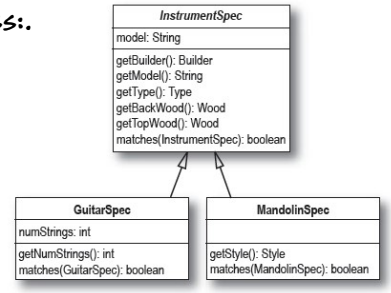
Q: We made `Instrument` abstract because we abstracted the properties common to `Guitar` and `Mandolin` into it, right?

A: No, we made `Instrument` abstract because in Rick's system right now, there's no such thing as an actual "instrument." All it does is provide a common place to store properties that exist in both the `Guitar` and `Mandolin` classes. But since an instrument currently has no behavior outside of its subclasses, it's really just defining common attributes and properties that all instruments need to implement.

So while we did abstract out the properties common to both instrument types, that doesn't necessarily mean that `Instrument` has to be abstract. In fact, we might later make `Instrument` a concrete class, if that starts to make sense in our design...

Q: Couldn't we do the same thing with `GuitarSpec` and `MandolinSpec`? It looks like they share a lot of common attributes and operations, just like `Guitar` and `Mandolin`.

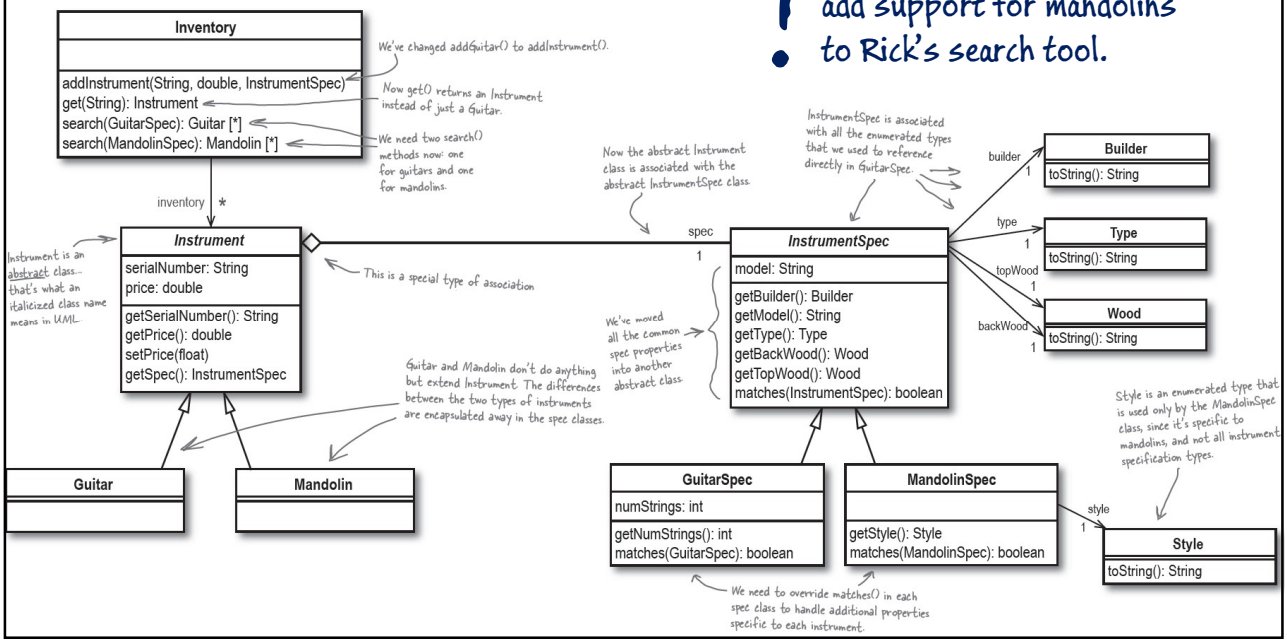
A: Good idea! We can create another abstract base class, called `InstrumentSpec`, and then have `GuitarSpec` and `MandolinSpec` inherit from that base class:.



7

Behold: Rick's new application

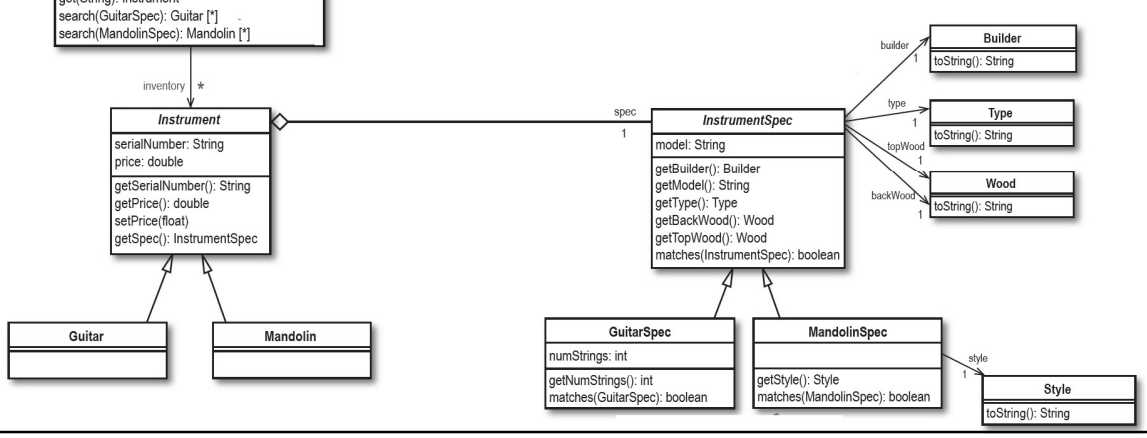
! it took us only 7 slides to add support for mandolins to Rick's search tool.



8

Here's the principle that led to us creating both the Instrument and InstrumentSpec abstract base classes.

Whenever you find common behavior in two or more places, look to abstract that behavior into a class, and then reuse that behavior in the common classes.



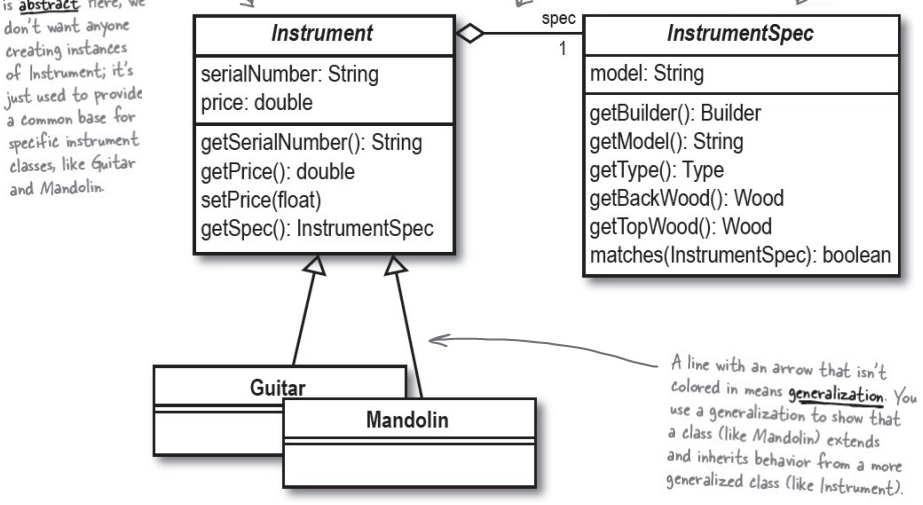
9

Class diagrams dissected (again)

This line with a diamond means **aggregation**. 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.

More italics: InstrumentSpec is also abstract

When the name of a class is in italics, the class is **abstract**. Here, we don't want anyone creating instances of Instrument; it's just used to provide a common base for specific instrument classes, like Guitar and Mandolin.



A line with an arrow that isn't colored in means **generalization**. You use a generalization to show that a class (like Mandolin) extends and inherits behavior from a more generalized class (like Instrument).

10

UML Cheat Sheet

What we call it in Java	What we call it in UML	How we show it in UML
Abstract Class	Abstract Class	Italicized Class Name
Relationship	Association	→
Inheritance	Generalization	▷
Aggregation	Aggregation	◇

there are no Dumb Questions

Q: Are there lots more types of symbols and notations that I'm going to have to keep up with to use UML?

A: There are a lot more symbols and notations in UML, but it's up to you how many of them you use, let alone memorize.

Many people use just the basics you've already learned, and are perfectly happy (as are their customers and managers).

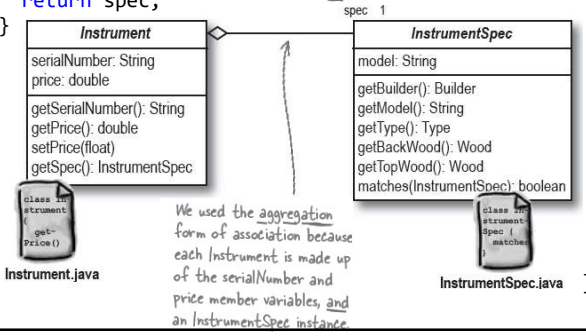
But other folks like to really get into UML, and use every trick in the UML toolbox. It's really up to you; as long as you can communicate your design, you've used UML the way it's intended.

11

```
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;
  }
  // Get and set methods for
  // serial number and price
  public InstrumentSpec getSpec() {
    return spec;
  }
}
```

Instrument is abstract... you have to instantiate subclasses of this base class, like Guitar.

Most of this is pretty simple, and looks a lot like the old Guitar class we had.



We used the aggregation form of association because each Instrument is made up of the serialNumber and price member variables, and an InstrumentSpec instance.

```
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;
  }
  // All the get methods for builder, model, type, etc.
  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;
  }
}
```

Just like Instrument, InstrumentSpec is abstract, and you'll use subclasses for each instrument type.

This is similar to our old Guitar constructor...

...except that we've pulled out properties not common to all instruments, like numStrings and style.

Let's code Rick's new search tool

12

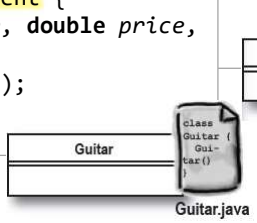
Next we need to rework **Guitar.java**, and create a class for mandolins.

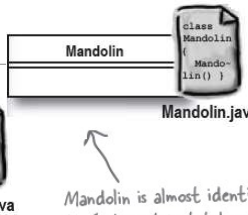
These both extend **Instrument** to get the common instrument properties, and then define their own constructors with the right type of spec class:

```
public class Mandolin extends Instrument {
  public Mandolin(String serialNumber, double price,
                  MandolinSpec spec) {
```

All each instrument class needs is to extend Instrument, and provide a constructor that takes the right kind of spec object.

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





Mandolin is almost identical to Guitar; it just takes in a MandolinSpec in the constructor, instead of a GuitarSpec.

13

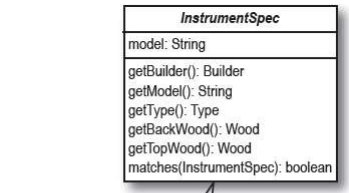
```
public class GuitarSpec extends InstrumentSpec {
  private int numStrings;
  public GuitarSpec(Builder builder, String model, Type type,
                    int numStrings, Wood backWood, Wood topWood) {
    super(builder, model, type, backWood, topWood);
    this.numStrings = numStrings;
  }

  public int getNumStrings() {
    return numStrings;
  }

  // Override the superclass matches()
  public boolean matches(InstrumentSpec otherSpec) {
    if (!super.matches(otherSpec))
      return false;
    if (!(otherSpec instanceof GuitarSpec))
      return false;
    GuitarSpec spec = (GuitarSpec)otherSpec;
    if (numStrings != spec.numStrings)
      return false;
    return true;
  }
}
```

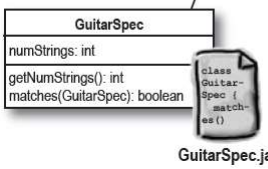
Just as Guitar extended Instrument, GuitarSpec extends InstrumentSpec.

Only a guitar has a numStrings property; it's not in the Instrument superclass.

This constructor just adds the guitar-specific properties to what's already stored in the base InstrumentSpec class.

matches() uses the superclass's matches(), and then performs additional checks to make sure the spec is the right type, and matches the guitar-specific properties.



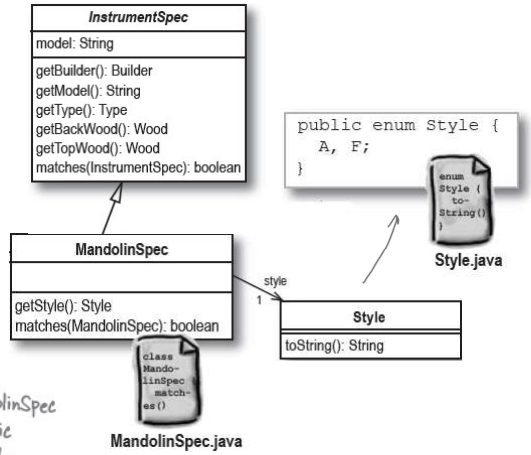


14

```
public class MandolinSpec extends InstrumentSpec {

  private Style style;

   public MandolinSpec(Builder builder, String model, Type type,
                       Style style, Wood backWood, Wood topWood) {
     super(builder, model, type, backWood, topWood);
     this.style = style;
   }
   public int getStyle() {
     return style;
   }
   // Override the superclass 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;
   }
}
```

Only mandolins have a Style, so this is not pushed up into the InstrumentSpec base class.

Just like GuitarSpec, MandolinSpec uses its superclass to do basic comparison, and then casts to MandolinSpec and compares the mandolin-specific properties.



15

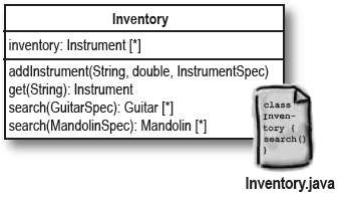
```
public class Inventory {
  private List inventory;
  public Inventory() {
    inventory = new LinkedList();
  }
  public void addInstrument(String serialNumber, double price,
                            InstrumentSpec spec) {
    Instrument instrument = null;
    if (spec instanceof GuitarSpec) {
      instrument = new Guitar(serialNumber, price, (GuitarSpec)spec);
    } else if (spec instanceof MandolinSpec) {
      instrument = new Mandolin(serialNumber, price, (MandolinSpec)spec);
    }
    inventory.add(instrument);
  }
  public Instrument get(String serialNumber) {
    for (Iterator i = inventory.iterator(); i.hasNext(); ) {
      Instrument instrument = (Instrument)i.next();
      if (instrument.getSerialNumber().equals(serialNumber)) {
        return instrument;
      }
    }
    return null;
  }
  // search(GuitarSpec) works the same as before
  public List search(MandolinSpec searchSpec) {
    List matchingMandolins = new LinkedList();
    for (Iterator i = inventory.iterator(); i.hasNext(); ) {
      Mandolin mandolin = (Mandolin)i.next();
      if (mandolin.getSpec().matches(searchSpec))
        matchingMandolins.add(mandolin);
    }
    return matchingMandolins;
  }
}
```

The inventory list now holds multiple types of instruments, not just guitars.

Hmmm... this isn't so great. Since Instrument is abstract, and we can't instantiate it directly, we have to do some extra work before creating an instrument.

Here's another spot where using an abstract base class makes our design more flexible.

We need another search() method to handle mandolins.



Finishing up Rick's search tool

All that's left is to update the `Inventory` class to work with multiple instrument types, instead of just the `Guitar` class:

16

there are no Dumb Questions

Guitar and Mandolin only have a constructor. That seems sort of silly. Do we really need a subclass for each type of instrument just for that?

These are little indicators that we may have a design problem. When things just don't seem to make sense in your application, you may want to investigate a little further... which is exactly what we're about to do.

But with Instrument as an abstract class, the addInstrument() method in Inventory.java becomes a real pain!

Why do we have two different versions of search()? Can't we combine those into a single method that takes an InstrumentSpec?

17

Wow, this is really starting to look pretty good! Using those abstract classes helped us avoid any duplicate code, and we've got instrument properties encapsulated away into our spec classes.

You've made some MAJOR improvements to Rick's app

You've done a lot more than just add support for mandolins to Rick's application. By abstracting common properties and behavior into the Instrument and InstrumentSpec classes, you've made the classes in Rick's app more independent. That's a significant improvement in his design.

I don't know... it seems like we've still got a few problems, like the almost-empty Guitar and Mandolin classes, and addInstrument() with all that nasty instrument-specific code. Are we just supposed to ignore those?

Great software isn't built in a day

Along with some major design improvements, we've uncovered a few problems with the search tool. That's OK... you're almost always going to find a few new problems when you make big changes to your design.

So now our job is to take Rick's better designed application, and see if we can improve it even further... to take it from good software to GREAT software.

18

3 steps to great software (revisited)

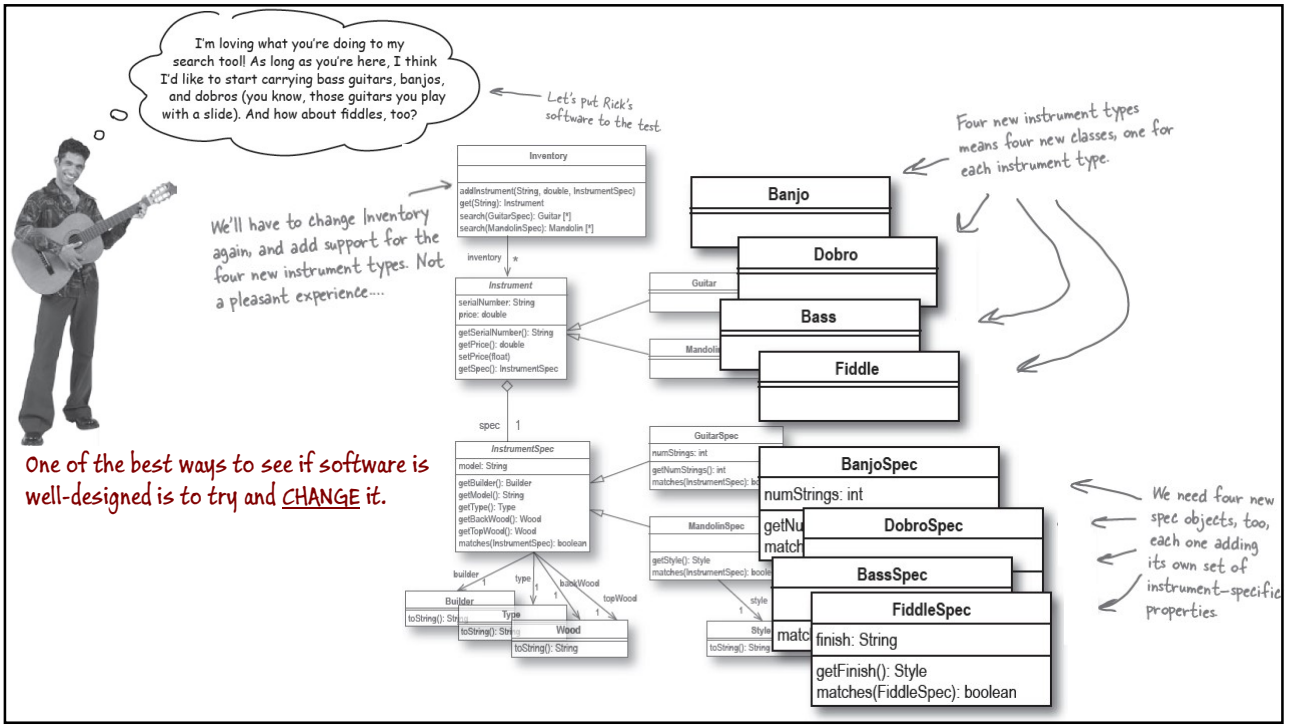
Is Rick's search tool great software?

1. Does the new search tool do what it's supposed to do?
   Absolutely. It finds guitars and mandolins, although not at the same time. So maybe it just mostly does what it's supposed to do. Better ask Rick to be sure...

2. Have you used solid OO principles, like encapsulation, to avoid duplicate code and make your software easy to extend?
   We used encapsulation when we came up with the `InstrumentSpec` classes, and inheritance when we developed an `Instrument` and `InstrumentSpec` abstract superclass. But it still takes a lot of work to add new instrument types...

3. How easy is it to reuse Rick's application? Do changes to one part of the app force you to make lots of changes in other parts of the app? Is his software loosely coupled?
   It's sort of hard to use just parts of Rick's application. Everything's pretty tightly connected, and `InstrumentSpec` is actually part of `Instrument` (remember when we talked about aggregation?).

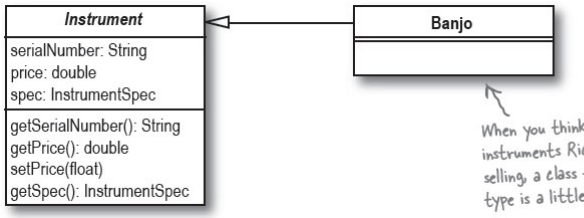
19



20

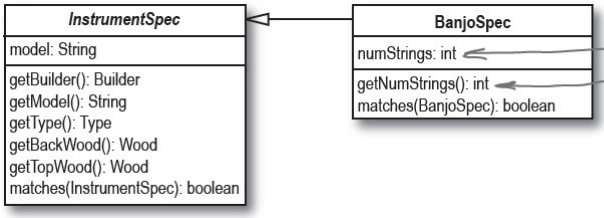
Uh oh... adding new instruments is not easy!

If ease of change is how we determine if our software is well-designed, then we've got some real issues here. Every time we need to add a new instrument, we have to add another subclass of `Instrument`:



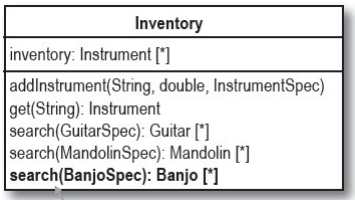
When you think about how many instruments Rick could end up selling, a class for each instrument type is a little scary.

Then, we need a new subclass of `InstrumentSpec`, too:



We're starting to have some duplicate code here.. banjos have a numStrings property like guitars, but it's not a common enough property to move into the Instrument superclass.

Then things start to really get nasty when you have to update the `Inventory` class's methods to support the new instrument type:



Remember all that instanceof and if/else stuff in addInstrument()? It gets worse with every new instrument type we support.

The search() situation is getting more annoying with every new instrument type. We need a new version that deals with banjos now.

21

So what are we supposed to do now?

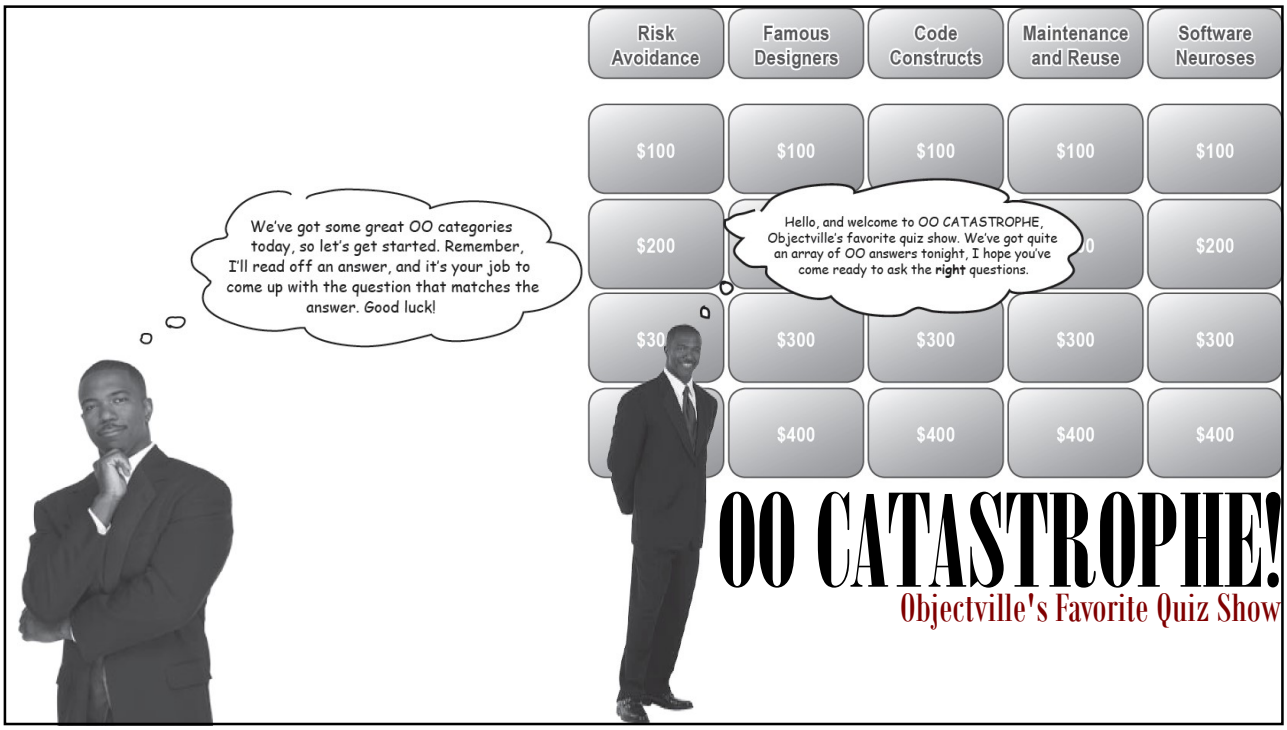
It looks like we've definitely still got some work to do to turn Rick's application into great software that's truly easy to change and extend. But that doesn't mean the work you've done isn't important... lots of times, you've got to improve your design to find some problems that weren't so apparent earlier on. Now that we've applied some of our OO principles to Rick's search tool, we've been able to locate some issues that we're going to have to resolve if we don't want to spend the next few years writing new **Banjo** and **Fiddle** classes (and who really wants to do that?).

Before you're ready to really tackle the next phase of Rick's app, though, there are a few things you need to know about. So, without further ado, let's take a quick break from Rick's software, and tune in to...

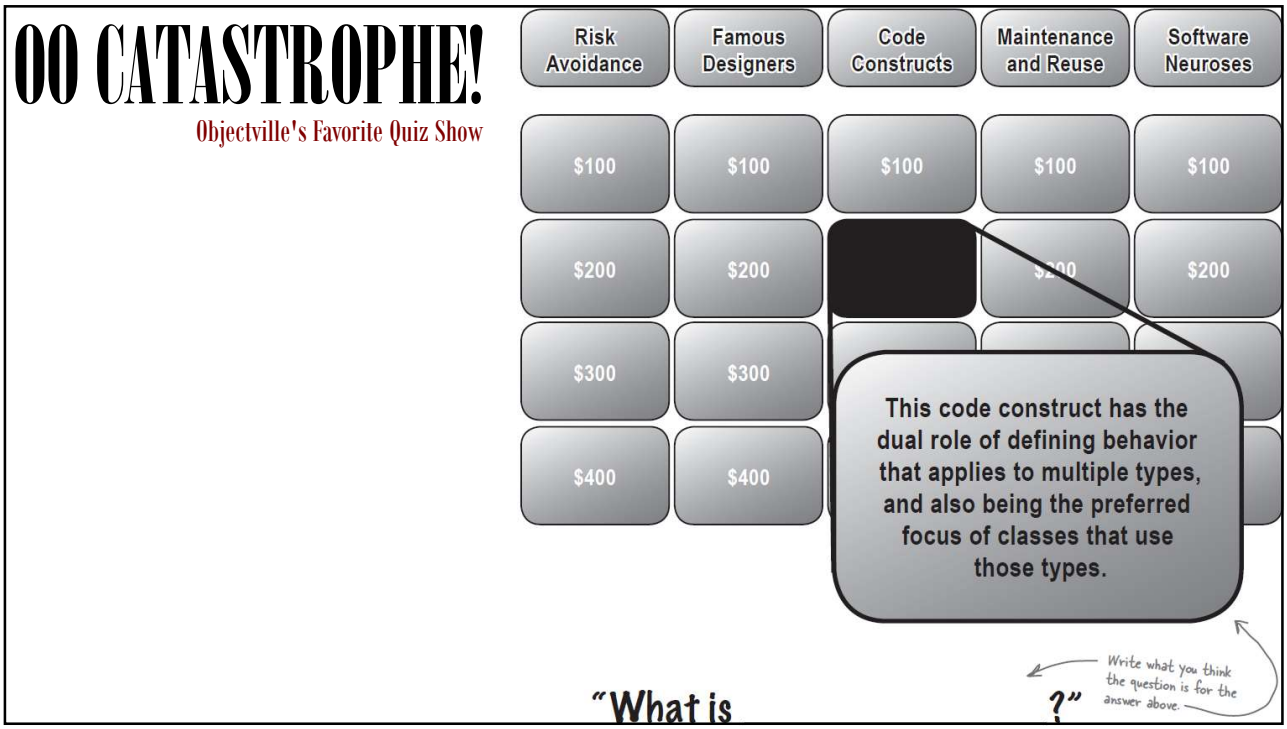
OO CATASTROPHE!

Objectville's Favorite Quiz Show

22


Risk Avoidance
Famous Designers
Code Constructs
Maintenance and Reuse
Software Neuroses
$100
$200
$300
$400
We've got some great OO categories today, so let's get started. Remember, I'll read off an answer, and it's your job to come up with the question that matches the answer. Good luck!
Hello, and welcome to OO CATASTROPHE, Objectville's favorite quiz show. We've got quite an array of OO answers tonight, I hope you've come ready to ask the right questions.
OO CATASTROPHE!
Objectville's Favorite Quiz Show

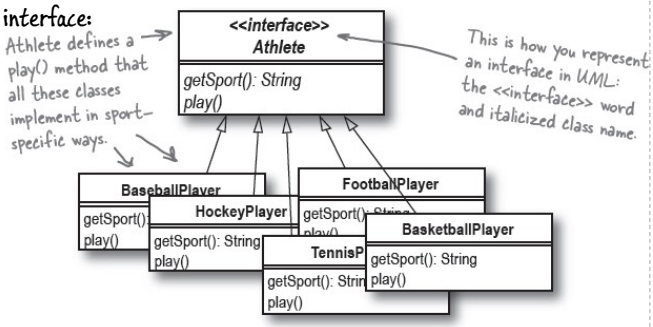
23


OO CATASTROPHE!
Objectville's Favorite Quiz Show
Risk Avoidance
Famous Designers
Code Constructs
Maintenance and Reuse
Software Neuroses
$100
$200
$300
$400
This code construct has the dual role of defining behavior that applies to multiple types, and also being the preferred focus of classes that use those types.
"What is
?"
Write what you think the question is for the answer above.

24

"What is an INTERFACE?"

Suppose you've got an application that has an interface, and then lots of subclasses that inherit common behavior from that interface:

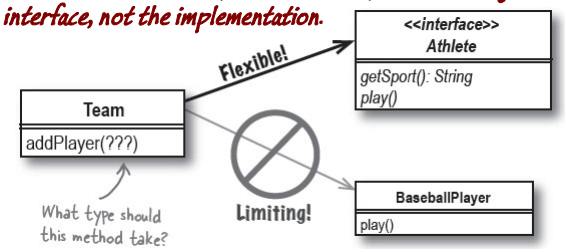


Coding to an interface, rather than to an implementation, makes your software easier to extend.

!

By coding to an interface, your code will work with all of the interface's subclasses, even ones that haven't been created yet.

Anytime you're writing code that interacts with these classes, you have two choices. You can write code that interacts directly with a subclass, like **FootballPlayer**, or you can write code that interacts with the interface, **Athlete**. When you run into a choice like this, you should *always* favor ***coding to the interface, not the implementation.***



Why is this so important? Because it adds ***flexibility*** to your app. Instead of your code being able to work with only one specific subclass—like **BaseballPlayer**—you're able to work with the more generic **Athlete**. That means that your code will work with any subclass of **Athlete**, like **HockeyPlayer** or **TennisPlayer**, and even subclasses that haven't even been designed yet (anyone for **CricketPlayer**?).

25

OO CATASTROPHE!

Objectville's Favorite Quiz Show

Risk Avoidance	Famous Designers	Code Constructs	Maintenance and Reuse	Software Neuroses
$100	$100	$100	$100	$100
$200	$200		$200	$200
$				$300
$			$400	$400

It's been responsible for preventing more maintenance problems than any other OO principle in history, by localizing the changes required for the behavior of an object to vary.

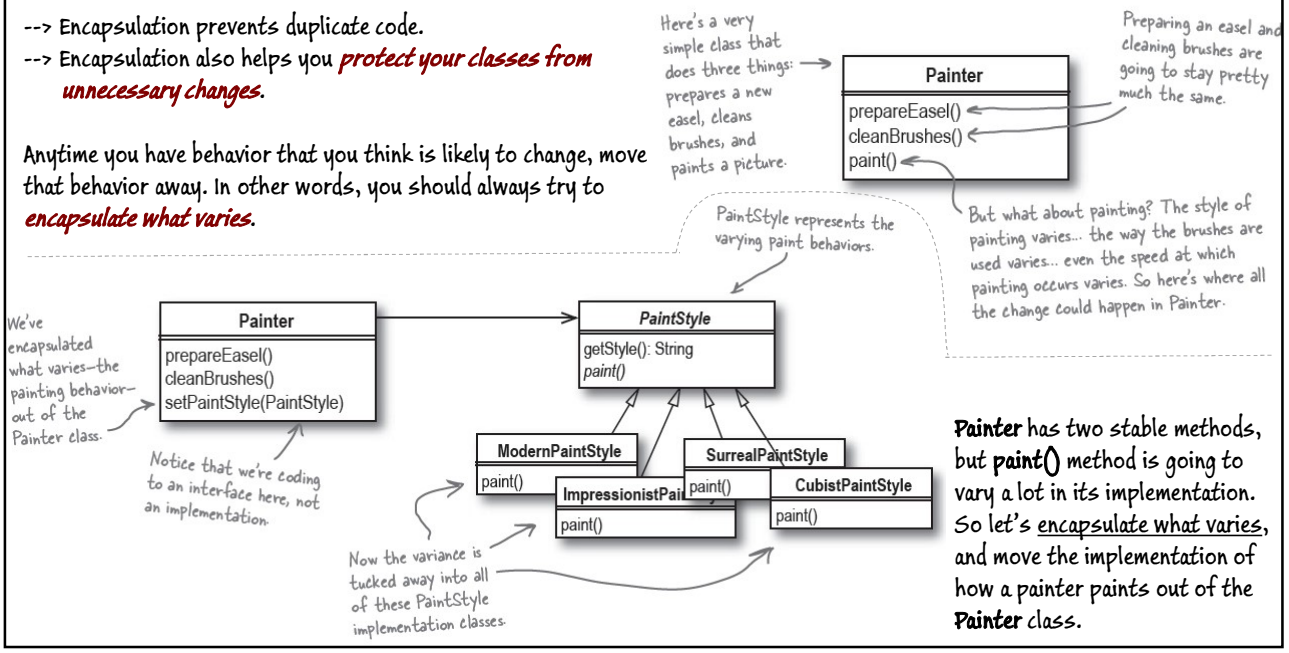
"What is _____?"

26

"What is ENCAPSULATION?"

--> Encapsulation prevents duplicate code.
--> Encapsulation also helps you *protect your classes from unnecessary changes.*

Anytime you have behavior that you think is likely to change, move that behavior away. In other words, you should always try to *encapsulate what varies.*

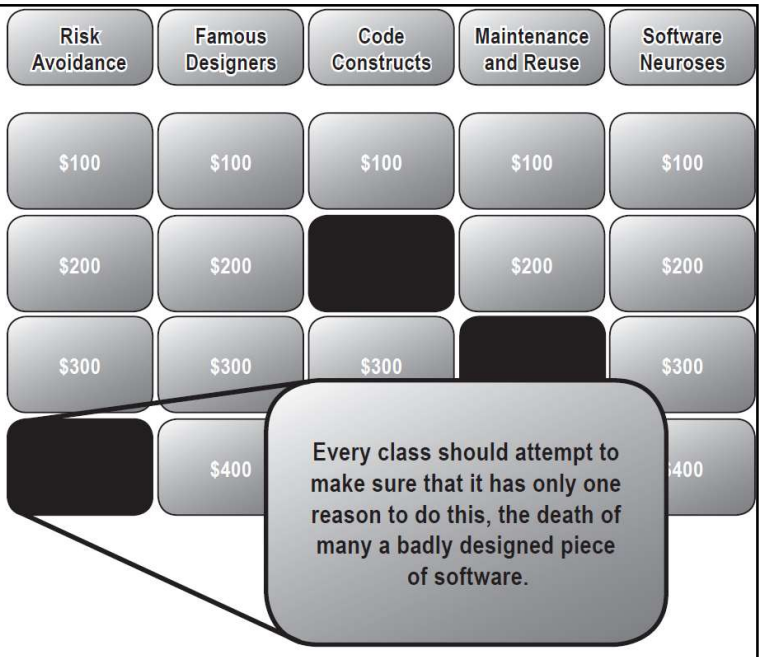


Painter has two stable methods, but **paint()** method is going to vary a lot in its implementation. So let's encapsulate what varies, and move the implementation of how a painter paints out of the **Painter** class.

27

OO CATASTROPHE!

Objectville's Favorite Quiz Show



"What is _____?"

28

"What is a CHANGE?"

Take a look at the methods in this class. They deal with starting and stopping, how tires are changed, how a driver drives the car, washing the car, and even checking and changing the oil.

Automobile
- start()
- stop()
- changeTires(Tire [*])
- drive()
- wash()
- checkOil()
- getOil(): int

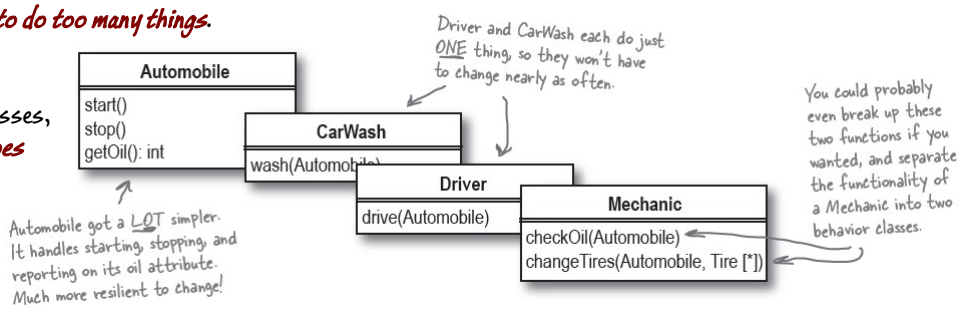
One constant in software is CHANGE.
But great software can change easily.

There are LOTS of things that could cause this class to change. If a mechanic changes how he checks the oil, or if a driver drives the car differently, or even if a car wash is upgraded, this code will need to change.

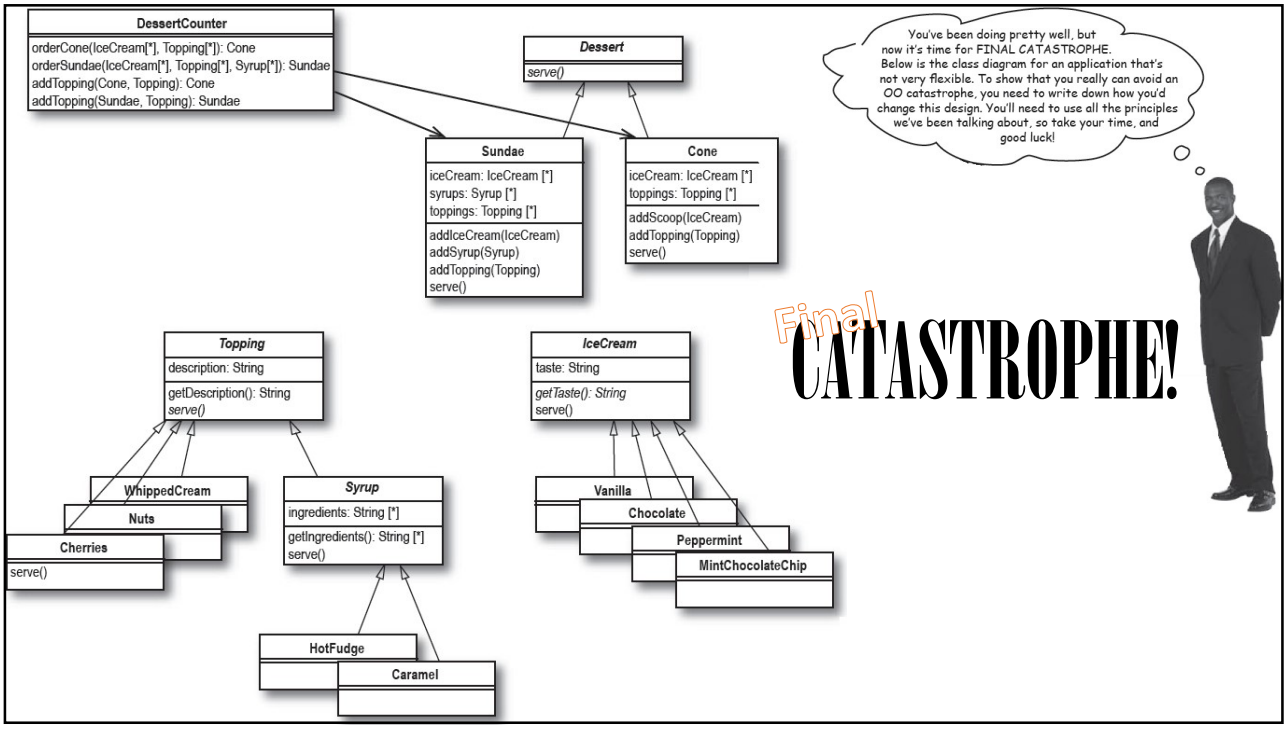
Make sure ***each class has only one reason to change.*** (minimize the chances that a class is going to have to change by reducing the number of things in that class that can ***cause*** it to change)

When you see a class that has more than one reason to change, it is probably ***trying to do too many things.***

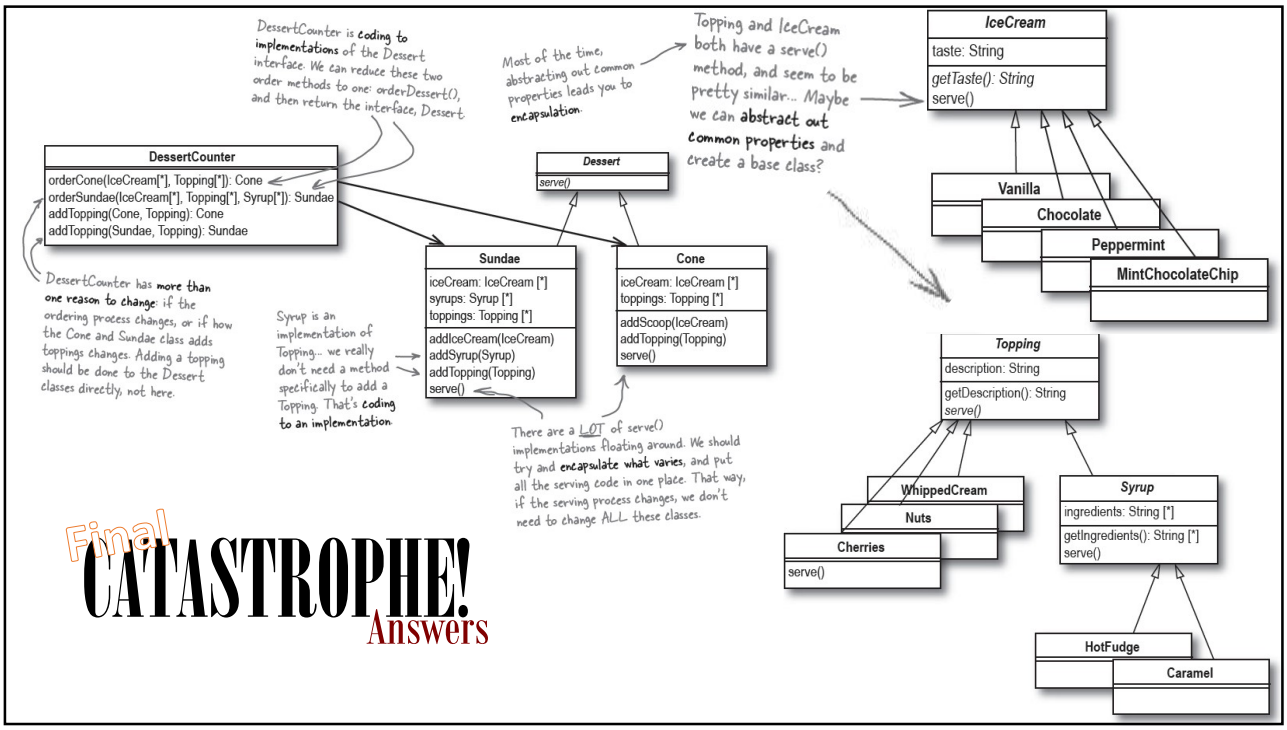
See if you can break up the functionality into multiple classes, where ***each individual class does only one thing***—and therefore has only **one** reason to change.



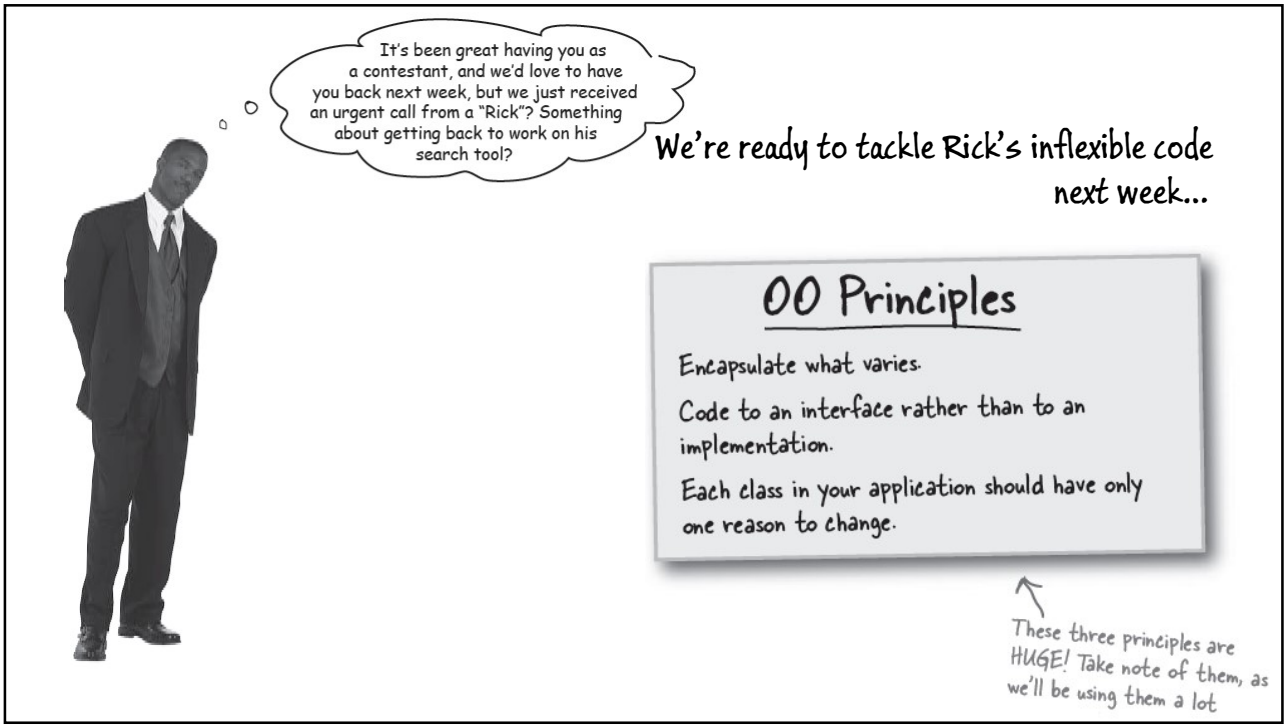
29



30


DessertCounter is coding to implementations of the Dessert interface. We can reduce these two order methods to one: orderDessert(), and then return the interface, Dessert.
Most of the time, abstracting out common properties leads you to encapsulation.
Topping and IceCream both have a serve() method, and seem to be pretty similar... Maybe we can abstract out common properties and create a base class?
DessertCounter
orderCone(IceCream[*], Topping[*]): Cone
orderSundae(IceCream[*], Topping[*], Syrup[*]): Sundae
addTopping(Cone, Topping): Cone
addTopping(Sundae, Topping): Sundae
Dessert
serve()
IceCream
taste: String
getTaste(): String
serve()
Vanilla
Chocolate
Peppermint
MintChocolateChip
Sundae
iceCream: IceCream [*]
syrups: Syrup [*]
toppings: Topping [*]
addIceCream(IceCream)
addSyrup(Syrup)
addTopping(Topping)
serve()
Cone
iceCream: IceCream [*]
toppings: Topping [*]
addScoop(IceCream)
addTopping(Topping)
serve()
DessertCounter has more than one reason to change: if the ordering process changes, or if how the Cone and Sundae class adds toppings changes. Adding a topping should be done to the Dessert classes directly, not here.
Syrup is an implementation of Topping... we really don't need a method specifically to add a Topping. That's coding to an implementation.
There are a LOT of serve() implementations floating around. We should try and encapsulate what varies, and put all the serving code in one place. That way, if the serving process changes, we don't need to change ALL these classes.
Topping
description: String
getDescription(): String
serve()
WhippedCream
Nuts
Cherries
serve()
Syrup
ingredients: String [*]
getIngredients(): String [*]
serve()
HotFudge
Caramel
Final
CATASTROPHE!
Answers

31


It's been great having you as a contestant, and we'd love to have you back next week, but we just received an urgent call from a "Rick"? Something about getting back to work on his search tool?
We're ready to tackle Rick's inflexible code next week...
OO Principles
Encapsulate what varies.
Code to an interface rather than to an implementation.
Each class in your application should have only one reason to change.
These three principles are HUGE! Take note of them, as we'll be using them a lot

32