CSE203 Object-Oriented Analysis and Design

05.2 Good Design = Flexible Software

Give Your Software a
30-minute Workout

1

previously on...

2

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

Your software is the best—I'm selling guitars left and right. I've been getting a lot of business from Nashville, though, and want to start carrying mandolins, too. I figure I can make a killing!

Mandolins are a lot like guitars... they shouldn't be too hard to support, right?

3

Did you notice that abstract base class?

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.

4

Behold: Rick's new application

it took us only 7 slides to add support for mandolins.

5

One of the best ways to see if software is well-designed is to try and CHANGE it.

Uh oh... adding new instruments is not easy

6

"What is an 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.

7

"What is an ENCAPSULATION?"

8

OO Design Principles

Encapsulate what varies

Code to an interface rather than to an implementation.

Each class should have only one reason to change.

9

THIS TIME on…

10

Back to Rick's search tool

11

A closer look at the `search()` method

We could make `InstrumentSpec` a concrete class, but would that solve all our problems?

12

The benefits of our analysis

```
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);
  }
  // get method here
  public List search(InstrumentSpec searchSpec) {
    List matchingInstruments = new LinkedList();
    for (Iterator i = inventory.iterator(); i.hasNext(); ) {
      Instrument instrument = (Instrument)i.next();
      if (instrument.getSpec().matches(searchSpec))
        matchingInstruments.add(instrument);
    }
    return matchingInstruments;
  }
}
```

We still have some issues here... this method gets bigger and more complicated every time we add a new type of instrument...

Here's the big change

...and we're coding to the implementation classes, not the Instrument base class.

search() is looking much better! Only one version, and it takes in an InstrumentSpec now.

We're coding to the Instrument base type now, not the implementation classes like Guitar and Mandolin. This is a much better design.

On top of better design, now search() can return all instruments that match, even if that list contains different types of instruments, like two guitars and one mandolin.

13

Sharpen your pencil

One of these things is not like the other... or is it?

The `search()` method isn't the only thing that makes adding new instruments to Rick's application difficult. You also have to add a new subclass of `Instrument` for each new instrument type. But why? Let's do a little more analysis.

1. Why is there a need for an `Instrument` class in Rick's application?
   Most instruments have at least a few common properties, like serial number and price. `Instrument` stores the common properties, and then each specific instrument type can extend from `Instrument`.
2. What things are common to all instruments?
   The serial number, the price, and some set of specifications (even though the details of those specs may be different for different instrument types).
3. What things are different between instruments?
   The specifications: each type of instrument has a different set of properties that it can contain. And since each instrument has a different `InstrumentSpec`, each has a different constructor.

14

A closer look at the instrument classes

Remember, we originally made **Instrument** abstract because each instrument type was represented by its own subclass:

But classes are really about behavior!

But the reason you usually create a subclass is because the **behavior** of the subclass is different than the superclass.

- In Rick's application, is the behavior of a **Guitar** different than an **Instrument**?
- Does it *function* differently than a **Mandolin** or **Banjo**?

15

Guitar and Mandolin and the other instruments don't have different behavior. But they have different **properties**... so we have to have subclasses for each instrument, right?

If we were writing a system that represented how these instruments played, we might need subclasses to handle behavior like pluck(), strum(), or fraill().

All the instruments ***behave the same***. So there are only two reasons to have subclasses for each instrument type:

1. Because the **Instrument** class represents a concept, and not an actual object, it really should be abstract. So we have to have subclasses for each instrument type.
2. Each different type of instrument has different properties, and uses a different subclass of **InstrumentSpec**, so we need an instrument-specific constructor for each type of instrument.

This is a good OO principle, but it sure is causing headaches with all the subclasses. We'll come back to this one in a moment.

This looks like another case where we're coding to an implementation instead of an interface. So this isn't a good reason to keep Instrument abstract.

Since Rick's app already does what it needs to do (Step 1), we're ready to try and make his software more flexible.

Apply basic OO principles to add flexibility.

How can we take this step and apply it to the problems we're finding in Rick's app?

16

Sharpen your pencil

Object-Oriented Principles to the rescue!

When you don't know what to do to solve a design problem, just run through the OO principles you know, and see if any of them might help improve your software's design.

Inheritance We're using inheritance already with the `Instrument` and `InstrumentSpec` classes, and their subclasses. But it does seem like the instrument-specific subclasses don't actually do anything but inherit from `Instrument`... they just have slightly different constructors.

Polymorphism We use polymorphism in the `search()` method to treat all instruments as instances of `Instrument`, instead of worrying about whether they're a `Guitar` or a `Mandolin`. So searching is a lot easier... but it would be nice to be able to use this in `addInstrument()`, too, and cut down on some repetitive code.

Abstraction `InstrumentSpec` abstracts the details about each instruments specifications away from the `Instrument` class itself, so that we can add new instrument properties without affecting the basic `Instrument` class.

Encapsulation We're using encapsulation a lot, but maybe we can use it even more... remember, encapsulate what varies! Since the properties in each instrument type are what varies, can we somehow encapsulate those properties away from `Instrument` and `InstrumentSpec` completely?

17

18

Death of a design (decision)

Having separate Instrument subclasses seemed to make sense at the time, and it's HARD to change something you thought was already working!

Code once, look twice (or more!)

Keep looking over your designs when you run into problems. A decision you made earlier may be what's causing you headaches now.

Design is iterative... and you have to be willing to change your own designs, as well as those that you inherit from other programmers.

Pride kills good design

Never be afraid to examine your own design decisions, and improve on them, even if it means backtracking.

Let's kill our bad design decision to create instrument-specific subclasses once and for all, and get on with writing great software again.

19

Let's turn some bad design decisions into good ones

Let's kill all those instrument-specific subclasses:

We just cut six classes out of the design, and won't need any new classes for instruments that Rick decides to start selling!

Instrument isn't abstract anymore.

We need a new property in each instrument to know the type of instrument

We can put values in here like GUITAR, BANJO, MANDOLIN, and so on. Much better than a bunch of subclasses.

This can be another enumerated type, like Wood and Builder. So now adding a new instrument type just means adding a new value to this enumerated type.

20

One more cubicle conversation

What varies in Rick's app?

the properties inside `InstrumentSpec`

So, maybe we need to add *another* layer of encapsulation.

Write what you think varies in these blanks.

21

"Double encapsulation" in Rick's software

We realized that clients and instruments both needed to use these properties, so we created the InstrumentSpec class to abstract them away from the Instrument class.

The problem is that these properties vary across instruments, and so we're having to add a subclass for each type of instrument.

We need a way to refer to properties and their values, but not have those properties hardcoded into the **InstrumentSpec** class. Any ideas for how we could do that?

By encapsulating what varies, you make your application more flexible, and easier to change.

22

Getting dynamic with instrument properties

23

What we did: a closer look

Anytime you see something that varies, you should look for a way to encapsulate.
In the case of `InstrumentSpec`, the properties of an instrument vary.

When you have a set of properties that vary across your objects, use a collection, like a Map, to store those properties dynamically.

You'll remove lots of methods from your classes, and avoid having to change your code when new properties are added to your app.

24

Using the new `Instrument` and `InstrumentSpec` classes

Let's take one last look at how our new **`Instrument`** and **`InstrumentSpec`** classes work in practice. Here's where we are with the design right now:

If you were accessing a guitar, and wanted to know who built it, how would you do that ?

25

26

```
import java.util._______;
import java.util._______;
import java.util._______;
public class InstrumentSpec {
  private ______ properties;
  public InstrumentSpec(______ __________) {
    if (properties == ______) {
      this.properties = new _________();
    } else {
      this.properties = new _________(__________);
    }
  }
  public ________ getProperty(String ___________) {
    return properties.get(__________);
  }
  public ______ getProperties() {
    return _____________;
  }
  public boolean matches(_____________ otherSpec) {
    for (__________ i = otherSpec.____________().keySet().__________();
                 i._________(); ) {
      String _____________ = (String)i._______();
      if (!properties.get(_____________).equals(
           otherSpec.getProperty(___________))) {
        return _________;
      }
    }
    return _________;
  }
}
```

Code magnets

properties
properties
hasNext
HashMap
next
Map
propertyName
InstrumentSpec
false
iterator
propertyName
false
getProperties
false
null
InstrumentSpec
propertyName
iterator
Object
properties
true
Iterator
Iterator
List
properties
true

27

```
import java.util.Iterator
import java.util.HashMap
import java.util.__Map__;
public class InstrumentSpec {
  private __Map__ properties;
  public InstrumentSpec(__Map__ properties) {
    if (properties == __null__) {
      this.properties = new __HashMap__();
    } else {
      this.properties = new __HashMap__(properties);
    }
  }
  public __Object__ getProperty(String __propertyName__) {
    return properties.get(__propertyName__);
  }
  public __Map__ getProperties() {
    return _____________;
  }
  public boolean matches(__InstrumentSpec__ otherSpec) {
    for (__iterator__ i = otherSpec.__getProperties__().keySet().__iterator__();
                 i.__hasNext__(); ) {
      String __propertyName__ = (String)i.__next__();
      if (!properties.get(_____________).equals(
           otherSpec.getProperty(__propertyName__)) {
        return __false__;
      }
    }
    return __true__;
  }
}
```

Code magnets

28

Finishing up Rick's app: the `InstrumentType` enum

```
public enum InstrumentType {

  GUITAR, BANJO, DOBRO, FIDDLE, BASS, MANDOLIN;

  public String toString() {
    switch(this) {
      case GUITAR:   return "Guitar";
      case BANJO:    return "Banjo";
      case DOBRO:    return "Dobro";
      case FIDDLE:   return "Fiddle";
      case BASS:     return "Bass";
      case MANDOLIN: return "Mandolin";
      default:       return "Unspecified";
    }
  }
}
```

So far, these are the types of instruments that Rick sells.

InstrumentType
toString(): String

InstrumentType.java

toString() just makes it easier to print things out.

Let's update `Inventory`, too

Inventory
inventory: Instrument [*]
addInstrument(String, double, InstrumentSpec)
get(String): Instrument
search(InstrumentSpec): Instrument [*]

```
public class Inventory {

  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);
    }
    Instrument instrument = new Instrument(serialNumber, price, spec);
    inventory.add(instrument);
  }

  // etc
}
```

Adding an instrument just got a lot easier.

Now we're able to instantiate Instrument directly, since it's no longer abstract.

Inventory.java

29

Behold: Rick's flexible application

We've made a ton of changes to Rick's application... and it's easy to forget what we've been working towards.
Look at the class diagram and see how much simpler Rick's application is now:

30

But does the application actually work?

```
public class FindInstrument {
  public static void main(String[] args) {
    // Set up Rick's inventory
    Inventory inventory = new Inventory();
    initializeInventory(inventory);
    Map properties = new HashMap();
    properties.put("builder", Builder.GIBSON);
    properties.put("backWood", Wood.MAPLE);
    InstrumentSpec clientSpec = new InstrumentSpec(properties);
    List matchingInstruments = inventory.search(clientSpec);
    if (!matchingInstruments.isEmpty()) {
      System.out.println("You might like these instruments:");
      for (Iterator i = matchingInstruments.iterator(); i.hasNext(); ) {
        Instrument instrument = (Instrument)i.next();
        InstrumentSpec spec = instrument.getSpec();
        System.out.println("We have a " + spec.getProperty("instrumentType") + " with the following properties:");
        for (Iterator j = spec.getProperties().keySet().iterator(); j.hasNext(); ) {
          String propertyName = (String)j.next();
          if (propertyName.equals("instrumentType"))
            continue;
          System.out.println(" " + propertyName + ": " + spec.getProperty(propertyName));
        }
        System.out.println(" You can have this " + spec.getProperty("instrumentType") +
          " for $" + instrument.getPrice() + "\n---");
        }
      } else {
      System.out.println("Sorry, we have nothing for you.");
    }
  }
  // initializeInventory() method here
}
```

FindInstrument.java

31

Inventory Roundup

Guitars

Collings CJ 6-string acoustic, Indian Rosewood back and sides, Spruce top, Serial #11277, for $3999.95

Martin D-18 6-string acoustic, Mahogany back and sides, Adirondack top, Serial #122784, for $5495.95

Fender stratocastor 6-string electric, Alder back and sides and top, Serial #V95693, for $1499.95

Fender stratocastor 6-string electric, Alder back and sides and top, Serial #V9512, for $1549.95

Gibson SG '61 Reissue 6-string electric, Mahogany back, sides, and top, Serial #82765501, for $1890.95

Gibson Les Paul 6-string electric, Maple back, sides, and top, Serial #70108276, for $2295.95

Mandolins

Gibson F5-G acoustic mandolin, Maple back, sides, and top, Serial #9019920, for $5495.99

Remember, the numStrings attribute doesn't apply to mandolins.

Banjos

Gibson RB-3 5-string acoustic banjo, Maple back and sides, Serial #8900231, for $2945.95

Banjos do not have a top wood.

```
private static void initializeInventory(Inventory inventory) {
  Map properties = new HashMap();
  properties.put("instrumentType", InstrumentType.GUITAR);
  properties.put("builder", Builder.COLLINGS);
  properties.put("model", "CJ");
  properties.put("type", Type.ACOUSTIC);
  properties.put("numStrings", 6);
  properties.put("topWood", Wood.INDIAN_ROSEWOOD);
  properties.put("backWood", Wood.SITKA);
  inventory.addInstrument("11277", 3999.95, new InstrumentSpec(properties));
  // your code goes here
}
```

32

Test driving Rick's well-designed software

Rick's client didn't specify an instrument type, but he wants something from Gibson with a maple back.

```
Map properties = new HashMap();
properties.put("builder", Builder.GIBSON);
properties.put("backWood", Wood.MAPLE);
InstrumentSpec clientSpec =
  new InstrumentSpec(properties);
```

class FindInstrument { main() }

FindInstrument.java

Rick's client ends up with three instruments to choose from: a guitar, a mandolin, and a banjo.

```
File Edit Window Help SatisfyTheCustomer
%java FindInstrument
You might like these instruments:
We have a Guitar with the following properties:
    topWood: Maple
    backWood: Maple
    builder: Gibson
    type: electric
    model: Les Paul
    numStrings: 6
  You can have this Guitar for $2295.95
---
We have a Mandolin with the following properties:
    topWood: Maple
    backWood: Maple
    builder: Gibson
    type: acoustic
    model: F-5G
  You can have this Mandolin for $5495.99
---
We have a Banjo with the following properties:
    backWood: Maple
    builder: Gibson
    type: acoustic
    model: RB-3 Wreath
    numStrings: 5
  You can have this Banjo for $2945.95
---
```

The guitar meets Bryan's specs because it has a maple back and sides, and is made by Gibson.

Here's a Gibson mandolin with a maple back... this also meets Bryan's specs.

One more maple instrument by Gibson... this one's a banjo. No top wood on banjos, but that doesn't matter.

there are no Dumb Questions

Is this really a good test since we only have one banjo and one mandolin?

33

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

34

The Great Ease-of-Change Challenge

Here's what our design looks like right now.

How easy is it to change Rick's search tool?

Let's add support for dobros and fiddles back into Rick's application.

Apply the ***ease-of-change test*** to our software:

1. How many classes did you have to ***add*** to support Rick's new instrument types?
   None!
2. How many classes did you have to ***change*** to support Rick's new instrument types?
   One: to the `InstrumentType` enumerated type.
3. Suppose that Rick decided that he wanted to start keeping up with what year an instrument was made in. How many classes would you need to change to support keeping up with this new information?
   None!
4. Rick also wants to add a new property, neckWood, that tracks what wood is used in the neck of an instrument. How many classes would you need to change to support this property?
   One in the worst case, and maybe none! `neckWood` is just another property we can store in the `InstrumentSpec` map... but we might need to add new wood enumerated values to the `Wood` enum.

35

A cohesive class does **one thing** really well and does not try to **do** or be **something else.**

The more cohesive your classes are, the higher the cohesion of your software.

Cohesive classes are focused on specific tasks. Our Inventory class worries about just Rick's inventory, not what woods can be used in a guitar, or how to compare two instrument specs.

Look through the methods of your classes—do they all relate to the name of your class? If you have a method that looks out of place, it might belong in another class.

Instrument doesn't try to handle searches, or keep up with what woods are available. It is focused on describing an instrument—and nothing else.

36

Cohesion, and one reason for a class to change

Every class should attempt to make sure that it has only one reason to do this, the death of many a badly designed piece of software.

This was one of the answers from OO CATASTROPHE! Do you remember what the question was?

CHANGE

the Scholar's Corner

cohesion. Cohesion measures the degree of connectivity among the elements of a single module, class, or object. The higher the cohesion of your software is, the more well-defined and related the responsibilities of each individual class in your application. Each class has a very specific set of closely related actions it performs.

37

Rick's software, in review

38

Knowing when to say "It's good enough!"

I love it! I can finally sell any kind of instrument I want, and help my clients find just what they're looking for.

Make sure the customer is happy

Before you ever leave a project, you always want to make sure your software does what it's supposed to do.

Really nice design here. High cohesion, the classes are loosely coupled... I'll bet the next time Rick needs something changed, we won't have too much trouble at all.

Make sure your design is flexible

Once you've got functionality down, move on to making good design decisions, using solid OO principles to add flexibility.

IF YOU'VE DONE BOTH OF THESE THINGS, IT MAY JUST BE TIME TO MOVE ON...

39

Tools for your toolbox

Wow, you've really come a long way since we started working with Rick way back in Week 1.

You've learned a ton about design, so let's do a quick review of what you've added to your toolbox.

Requirements

Good requirements ensure you works like your customers exp

Make sure your requirements by developing use cases for y

Use your use cases to find o things your customers forgot

Your use cases will reveal any or missing requirements that have.

Your requirements will always grow) over time.

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, then CHANGE IT! Never settle on bad design, even if it's your bad design that has to change.

Make sure each of your classes is cohesive: each of your classes should focus on doing ONE THING really well.

Always strive for higher cohesion as you move through your software's design life cycle.

The goal of good design is highly cohesive, loosely coupled software.

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.

Classes are about behavior and functionality.

Between OO CATASTROPHE! we've added quite a few new OO principles to our toolbox.

40