Great Software

Kenneth M. Anderson
University of Colorado, Boulder
CSCI 4448/6448 — Lecture 5 — 09/11/2007

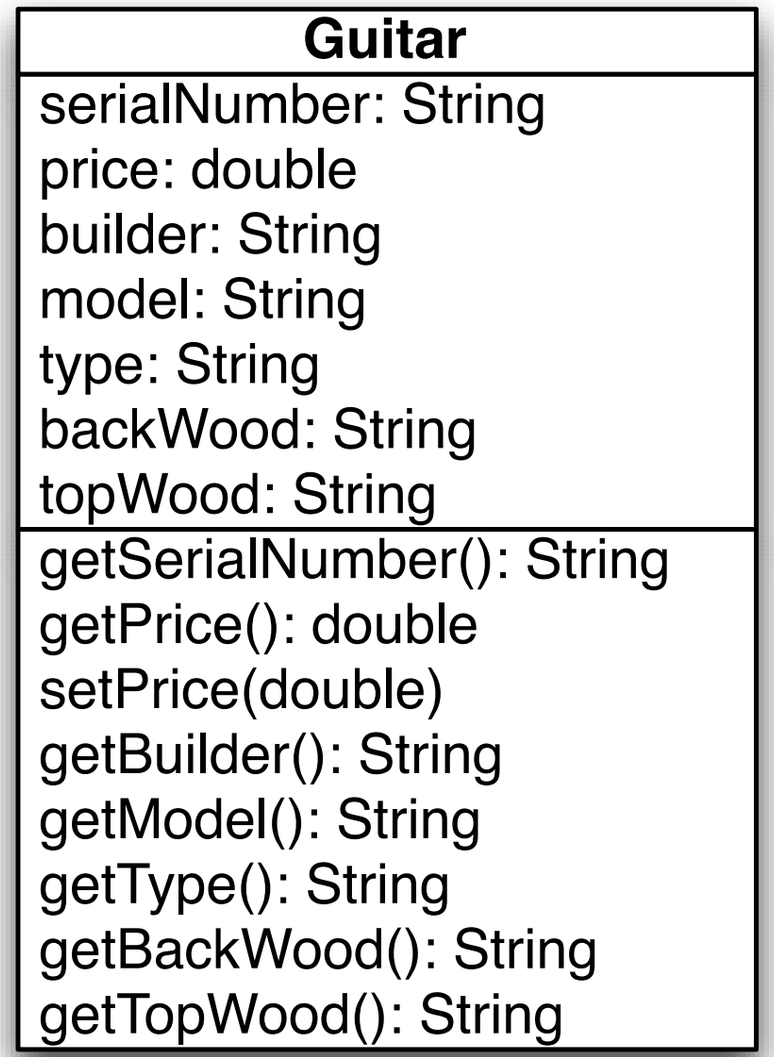
Lecture Goals

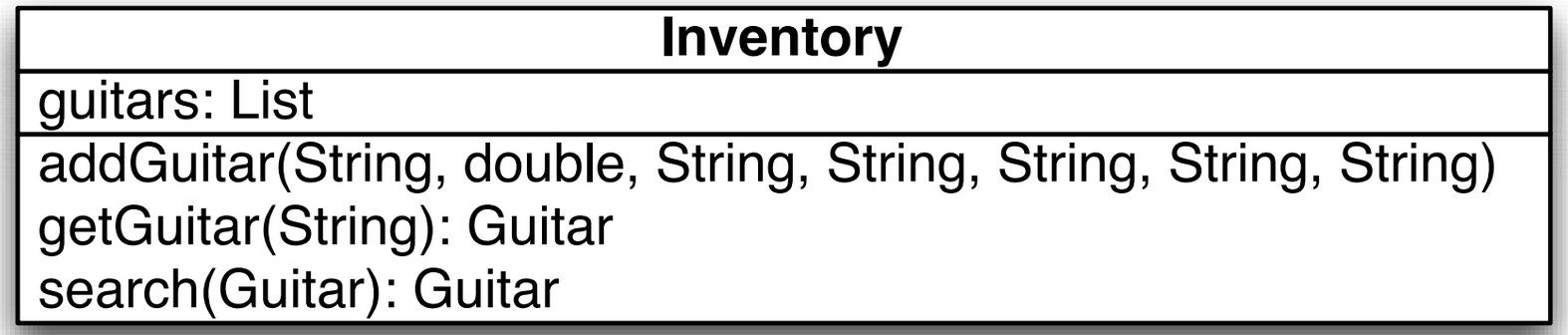
- Review material from Chapter 1 of the OO A&D textbook
  - What is Great Software?
  - How do you produce great software each time you participate in a development project
  - Discuss the Chapter 1 Example: Rick's Guitars
  - Emphasize the OO concepts and techniques encountered in Chapter 1

The Example

- Rick's Guitars
  - Purpose of Application?
    - Find Guitars for Rick's Customers
  - Design of Initial Application?
    - Worse than Bad
  - What's the Major Problem?
    - Application fails to find guitars that Rick knows exists!
- Note: Customer Focus… we could have said "The problem is due to the use of a case-sensitive string comparison" but Rick wouldn't care about that!

Initial Application



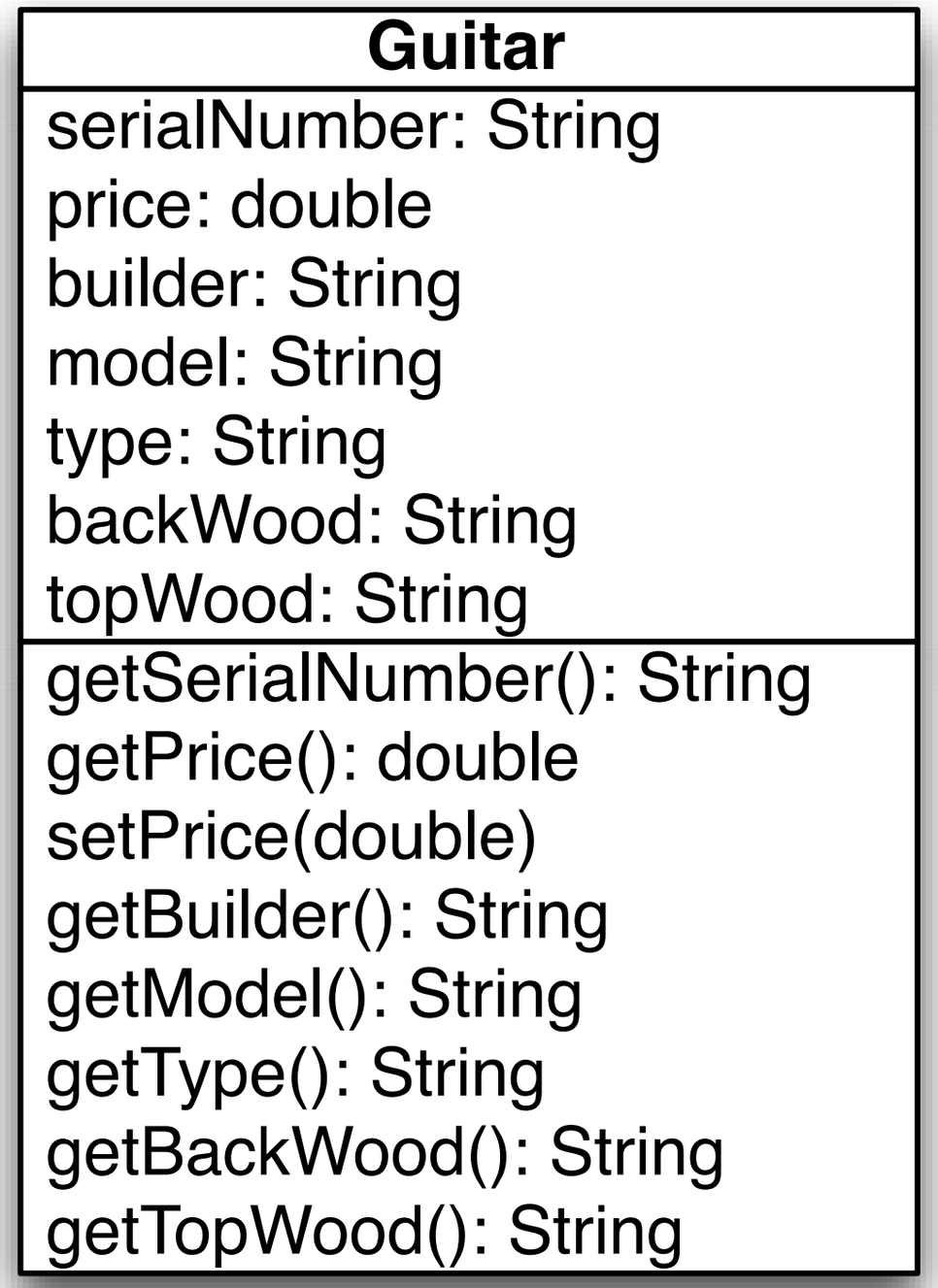


Egads!

Alarm bells in the design center of your brain should be ringing like mad!

Why?

Dumb Data Holder



- Meet the Dumb Data Holder
- A “data holder” is a class that only stores information and is dumb since it provides ZERO services
  - get() and set() don’t count!
- If this is all you are going to use an object for, you may as well go back to C and just use structs!

The Manager

- Meet the “Manager”
  - A manager is a single class that contains most of the application’s services
- For this simple application, a manager is okay, but… there are still problems

Inventory
guitars: List
addGuitar(String, double, String, String, String, String, String) getGuitar(String): Guitar search(Guitar): Guitar

1. addGuitar(): too many params

Managers must be used with care: its easy to centralize too many services within them!

2. getGuitar(): why only one param?

3. searchGuitar(): if I’m looking for a guitar, why am I passing one to the search() method?

Bad Smell?

```
1  public Guitar search(Guitar searchGuitar) {
2    for (Iterator i = guitars.iterator(); i.hasNext(); ) {
3      Guitar guitar = (Guitar)i.next();
4      // Ignore serial number since that's unique
5      // Ignore price since that's unique
6      String builder = searchGuitar.getBuilder();
7      if ((builder != null) && (!builder.equals("")) &&
8          (!builder.equals(guitar.getBuilder())))
9        continue;
10     String model = searchGuitar.getModel();
11     if ((model != null) && (!model.equals("")) &&
12         (!model.equals(guitar.getModel())))
13       continue;
14     String type = searchGuitar.getType();
15     if ((type != null) && (!searchGuitar.equals("")) &&
16         (!type.equals(guitar.getType())))
17       continue;
18     String backWood = searchGuitar.getBackWood();
19     if ((backWood != null) && (!backWood.equals("")) &&
20         (!backWood.equals(guitar.getBackWood())))
21       continue;
22     String topWood = searchGuitar.getTopWood();
23     if ((topWood != null) && (!topWood.equals("")) &&
24         (!topWood.equals(guitar.getTopWood())))
25       continue;
26     return guitar;
27   }
28   return null;
29 }
30 }
31
```

This is really horrible code.

Why?

Lots of needless duplication

Hard to Read

Some Attributes of Guitar are being ignored

Plus: It Doesn't work!

- The search relies on string comparisons and this is susceptible to problems between differences in what is stored and what is entered
  - In the case of the example
    - the database stored: "Fender"
    - the customer entered: "fender"
  - NO MATCH!

A Good Question

- A team is assembled to fix the problem in the original tool and they ask a good question
  - **How are we supposed to know where to start?**
    - One designer wants to get rid of the ubiquitous use of strings in the app
    - One wants to improve the design
    - One sees that the current app doesn't even do what the user wants
- They just want to create great software, but how?

What is Great Software?

- **Customer-Friendly Programmer**
  - Great software always does what the customer wants it to. So even if customers think of new ways to use the software, it doesn't break or give them unexpected results
- **OO Programmer**
  - Great software is code that is object-oriented. So there's not a bunch of duplicate code, and each object pretty much controls its own behavior. It's also easy to extend because your design is really solid and flexible.
- **Design-Guru Programmer**
  - Great software is when you use tried-and-true design patterns and principles. You've kept your objects loosely coupled, and your code open for extension but closed for modification. That also helps make the code more reusable, so you don't have to rework everything to use parts of your application over and over again.

What is GREAT SOFTWARE? continued

- Which programmer is correct?

All of the Above!

Great Software

- Great software must satisfy the customer
    - The software must do what the customer wants it to do!
- Great software is also
    - well-designed
    - well-coded
    - easy to maintain, reuse, and extend

- So, how do we achieve this?

Great Software in Three Easy Steps

- Your FIRST OO A&D Process!

    1. Make sure your software does what the customer wants it to do.

    2. Apply basic OO principles to add flexibility

    3. Strive for a maintainable, reusable design

Step 1: Customer First

- The first step to fixing the program is to make sure it does what the customer wants
  - Do not worry about steps 2 and 3 at this point, simply figure out how to fix the problem in a straightforward manner
  - However, be smart about how you fix things
    - **Don’t create problems to solve (other) problems**

Improvements

- Ditch String Comparisons
    - A string can have TONS of legal values
    - But Guitar's type, builder, and wood attributes DON'T
        - Solution
            - Use enumerated types to restrict the legal values
                - This provides both **type safety** and **value safety**
            - The compiler can now ensure that legal values are stored in our database and used in queries
- The code is now less fragile.

Improvements (II)

- Fix search() method to return more than one choice
  - Rick specified this, but the original coders failed to meet this requirement
  - This is a small change involving creating a list to hold search results as we find them and modifying search() to return this new list
- **Demonstration**
- Step 1 Complete!
  - The system now does what Rick wants.

Step 2: Apply Principles

- Attention turns to the fact that a Guitar object is being passed to the search() method, when that's what we are looking for in the first place!
- The problem?
  - **Weak Encapsulation**
  - The Guitar class is being asked to serve as a Guitar object AND as a search specification
    - In the latter role, some of its attributes are ignored!
      - This is always a warning sign of bad or "undercooked" design!

Part 2, Continued

- The solution?
  - **Strong Encapsulation**
  - If you need an object to act as a search specification, then create one!
  - GuitarSpec
    - It contains all of the attributes used by the original search() method
    - no ignored attributes!
- Since these attributes are also needed by the Guitar class:
  - the Guitar class can use it internally
- Thus, we also use delegation in this new design

Benefits

- Creating GuitarSpec and updating Guitar to make use of it, is an excellent example of encapsulation
  - Imagine if we didn't update the Guitar class… what would we have?
    - **DUPLICATED CODE**
      - Two builder attributes, two getBuilder() methods, etc.
- In addition, if we needed to add a property to GuitarSpec, we would also need to add it to Guitar
  - By incorporating GuitarSpec into Guitar, both problems are avoided

How Do I Know?

- How do we know which principles to apply?
  - In this situation, encapsulation was the principle that was needed, and Guitar and search() were the places that needed it applied
- The answer?
  - **Experience!**
  - It won't always be obvious and it won't always be ONE change that needs to be applied
- You will get better at spotting these things as you build more systems and gain experience

Step 3: Design Again

- Now, we look at our design and see if there are opportunities to make it more flexible, more extensible, and/or more reusable
- To set up this stage, the book asks a really good question
  - **How easy is it to make a change to Rick's application?**
    - What would be easy to change?
    - What would be not so easy to change?
- **This is an excellent question to ask of any software design**
  - You make **tradeoffs** when you create a design… it will be impossible to create a design in which everything is easy to change
    - The trick is to make things that are ***likely* to change, *easy* to change**

Robustness in the Face of Change

- When they find out from Rick that he now carries 12-string guitars, they discover that they need to
    - Add a property and method to GuitarSpec
    - Change the constructor of the Guitar class
    - Change the addGuitar() method of Inventory
    - Update the search() method of Inventory
- YIKES!
    - One change request required changes in ALL of the classes created so far

The problem?

- Even though we applied encapsulation in Step 2, we didn't go far enough
  - If we need to change Guitar and Inventory, each time we change GuitarSpec, we have a problem

The Solution

- The solution is to further encapsulate the program from the details of GuitarSpec
    - Change Guitar's constructor and Inventory's addGuitar() method to take a GuitarSpec
    - Move comparison of two GuitarSpecs into the GuitarSpec object itself
        - Yes! GuitarSpec is no longer a dumb data holder!
        - Delegation to the rescue again!
            - Inventory now delegates the responsibility of comparison to the GuitarSpec object

Program Evolution

- The final state of the example program is much improved from the poorly designed initial program
  - Moved away from “all strings all the time” design and used enumerated types to elegantly restrict the values of certain Guitar attributes
  - Applied encapsulation and delegation multiple times to create a system that guards against anticipated change:
    - if Rick wants a new attribute to track on his guitars, we just modify GuitarSpec and everything else JUST WORKS

Benefits Of Object Orientation

- **OO software (great software) does what the customer wants**
  - Our process focuses on customer needs first
- **OO software continues to work as needs change**
  - its easy to isolate information that may change
- **OO software can be upgraded**
  - OO systems are maintainable and extensible
- **OO software can be reused and is flexible**
  - Loosely coupled and highly cohesive objects can be moved to new contexts
- We'll see more examples of these points throughout the semester

Coming Up Next

- Lecture 6: Give Them What They Want
    - Read Chapter 2 of the OO A&D book
- Lecture 7: Dealing with Change
    - Read Chapter 3 of the OO A&D book