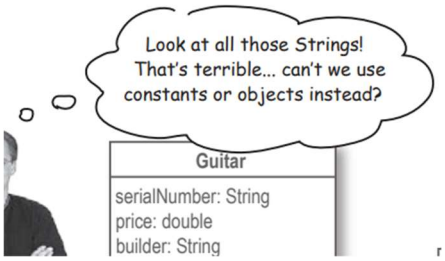
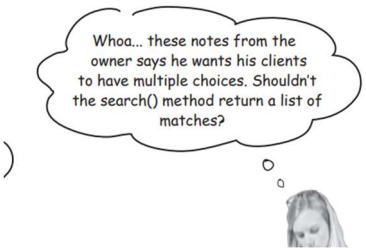
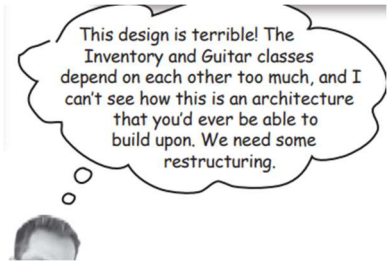
Sharpen your pencil

How would you redesign Rick's app?

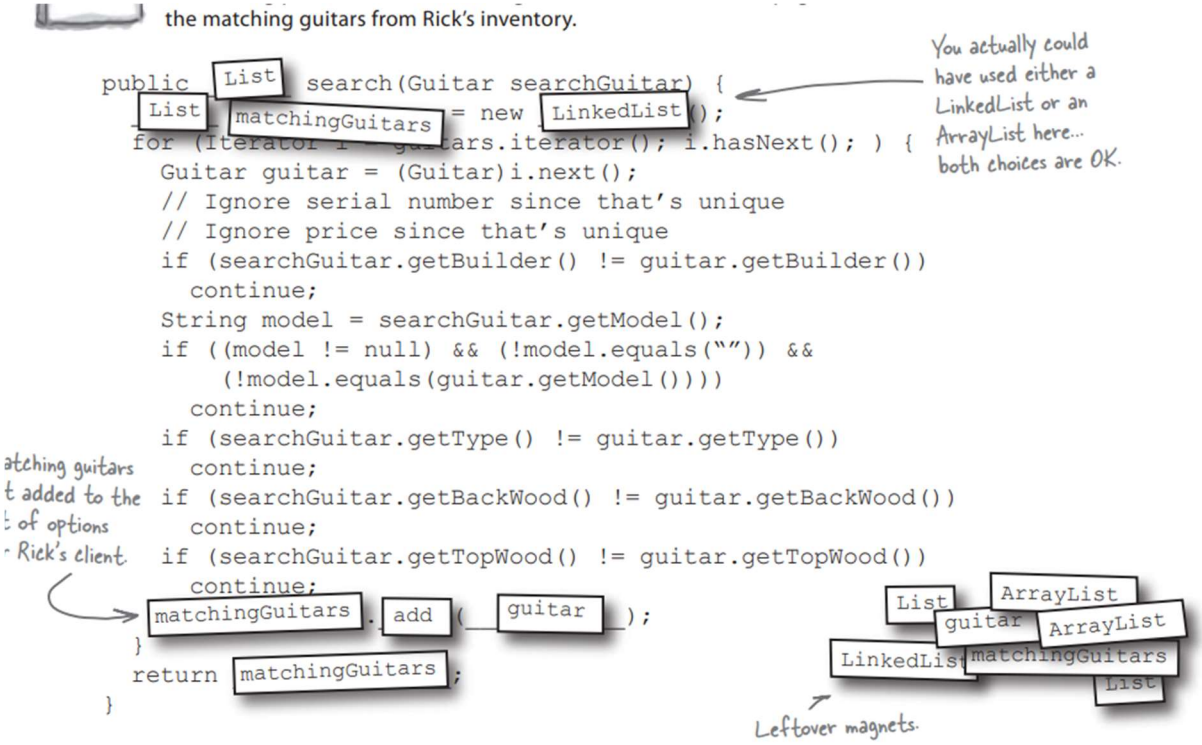
Look over the last three pages, showing the code for Rick's app, and the results of running a search. What problems do you see? What would you change? Write down the **FIRST** thing you'd do to improve Rick's app in the blanks below.







the matching guitars from Rick's inventory.



What do you think the mismatched object type is? Write your answer in the blank below:

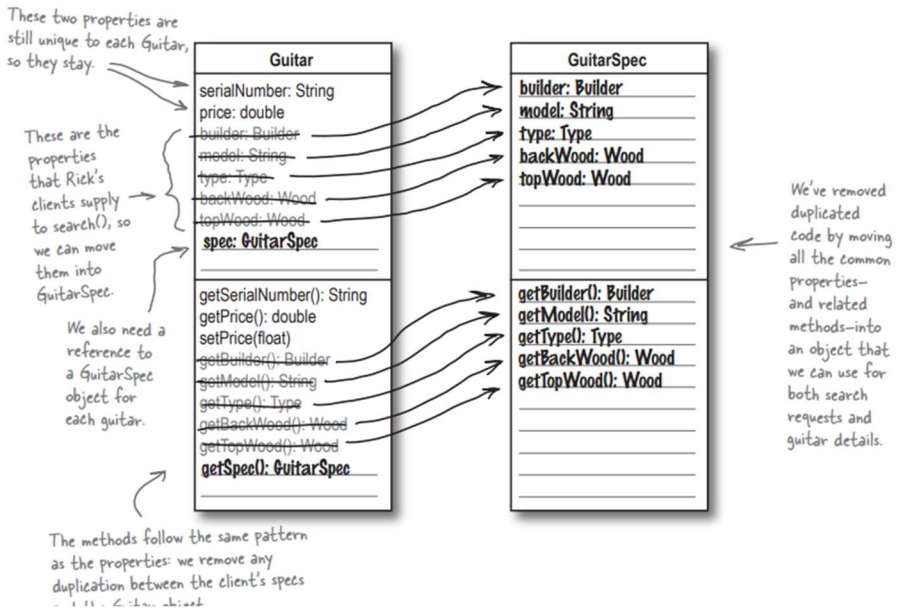
_____

What do you think you should do to fix the problem? What changes would you make?

_____

_____

_____



Sharpen your pencil answers

Annotate Rick's class diagram.

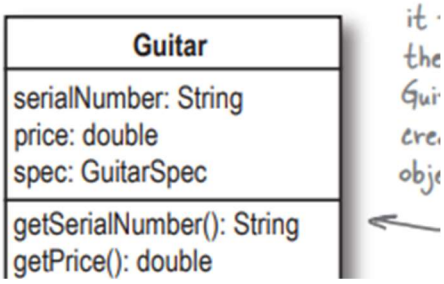
Rick wants to be able to sell 12-string guitars. Get out your pencil and add notes to the class diagram showing the following things:

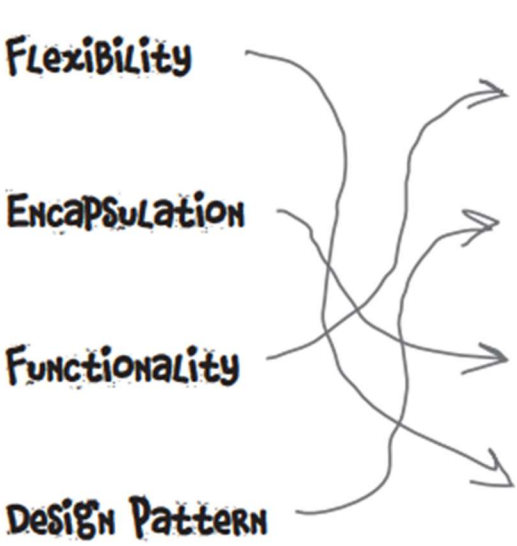
1. Where you'd add a new property, called numStrings, to store the number of strings a guitar has.
2. Where you'd add a new method, called getNumStrings(), to return the number of string a guitar has.
3. What other code you think you'd need to change so that Rick's clients can specify that they want to try out 12-string guitars.

Finally, in the blanks below, write down any problems with this design that you found whe adding support for 12-string guitars.

We're adding a property to GuitarSpec, but we have to change code in the Inventory class's search() method, as well as in the constructor to the Guitar class.

Here's what we came up with... did you write down something similar?





Without me, you'll never actually make the customer happy. No matter how well-designed your application is, I'm the thing that puts a smile on the customer's face.

I'm all about reuse and making sure you're not trying to solve a problem that someone else has already figured out.

You use me to keep the parts of your code that stay the same separate from the parts that change: then it's really easy to make changes to your code without breaking everything.

Use me so that your software can change and grow without constant rework. I keep your application from being fragile.

answers

What would you change about this code?

There's a big problem with the code shown above, and it's up to you to figure it out. In the blanks below, write down what you think the problem is, and how you would fix it.

Every time a new property is added to GuitarSpec.java, or the methods in GuitarSpec change, the search() method in Inventory.java will have to change, too. We should let GuitarSpec handle comparisons, and encapsulate these properties away from Inventory.

This isn't very good design. Every time a new property is added to GuitarSpec, this code is going to have to change.

```
public List search(GuitarSpec searchSpec) {
  List matchingGuitars = new LinkedList();
  for (Iterator i = guitars.iterator(); i.hasNext(); ) {
    Guitar guitar = (Guitar)i.next();
    GuitarSpec guitarSpec = guitar.getSpec();
    if (searchSpec.getBuilder() != guitarSpec.getBuilder())
      continue;
    String model = searchSpec.getModel().toLowerCase();
    if ((model != null) && (!model.equals("")) &&
        (!model.equals(guitarSpec.getModel().toLowerCase())))
      continue;
    if (searchSpec.getType() != guitarSpec.getType())
      continue;
    if (searchSpec.getBackWood() != guitarSpec.getBackWood())
      continue;
    if (searchSpec.getTopWood() != guitarSpec.getTopWood())
      continue;
    matchingGuitars.add(guitar);
  }
  return matchingGuitars;
}
```

Across

4. These help you avoid solving problems someone else has already solved.
7. Customers focus on this part of your applications.
8. Objects in loosely coupled applications are more _____ than tightly coupled ones.
9. Flexible applications are usually easy to _____.
10. This is one type of code you don't want to write.
12. Your applications should be easy to _____.
13. You usually need some sort of process to write great software _____.
14. Encapsulate what _____.
15. These types of applications satisfy programmers.

Down

1. Once your application works correctly, focus on this.
2. Grouping your application into logical parts.
3. The goal of OOA&D is to help you write this type of software.
5. Use this to let objects focus on more specific tasks.
6. A good way to avoid duplicate code
7. An application that things can go wrong in easily.
11. This is a four letter word your mom will be proud you know.

Slayt 2

Todd and Gina's Dog Door, version 2.0
What the Door Does

1. Fido barks to be let out.	N/A
2. Todd or Gina hears Fido barking.	N/A
3. Todd or Gina presses the button on the remote control.	2
4. The dog door opens.	2
5. Fido goes outside.	1
6. Fido does his business.	N/A
6.1 The door shuts automatically.	3
6.2 Fido barks to be let back inside.	N/A
6.3 Todd or Gina hears Fido barking (again).	N/A
6.4 Todd or Gina presses the button on the remote control.	2
6.5 The dog door opens (again).	2
7. Fido goes back inside.	1
8. The door shuts automatically.	3

A lot of the things that happen to a system don't require you to do anything.

You might have put N/A here, since them pushing the button isn't something that's you have to handle... then again, 2 is OK, too, since they wouldn't push a button without a remote.

Did you get this one? Fido can't get outside if the opening isn't the right size.

The alternate path should have been easy once you figured out the requirements for the main path.

Remote.java

```
import java.util.Timer;
import java.util.TimerTask;

public class Remote {

  private DogDoor door;

  public Remote(DogDoor door) {
    this.door = door;
  }

  public void pressButton() {
    System.out.println("Pressing the remote control button...");
    if (door.isOpen()) {
      door.close();
    } else {
      door.open();

      final Timer timer = new Timer();
      timer.schedule(new TimerTask() {
        public void run() {
          door.close();
          timer.cancel();
        }
      }, 5000);
    }
  }
}
```

You'll need these two import statements to use Java's timing classes.

This checks the state of the door before opening or closing it.

The remote already has code to handle closing the door if it's open.

Create a new Timer so we can schedule the dog door closing.

All the task does is close the door, and then turn off the timer.

This tells the timer how long to wait before executing the task... in this case, we're waiting 5 seconds, which is 5000 milliseconds.

DogDoorSimulator.java

```
public class DogDoorSimulator {

  public static void main(String[] args) {
    DogDoor door = new DogDoor();
    Remote remote = new Remote(door);

    System.out.println( );
    remote . pressButton ( );
    System.out.println("\nFido has gone outside...");
    System.out.println("\nFido's all done...");

    try {
      Thread.currentThread(). sleep (10000);
    } catch (InterruptedException e) { }

    System.out.println( ...but he's stuck outside!" );
    System.out.println( "\nFido starts barking..." );
    System.out.println( ...so Gina grabs the remote control." );
    remote . pressButton ( );
    System.out.println("\nFido's back inside...");
  }
}
```

You should have written in periods, semicolons, and parentheses as you needed them.

You could have chosen the message about Todd grabbing the remote, but we're trying to test for the real world, remember? We figure Gina's doing most of the work here.

External Initiator → Kicks off the list of steps described in a use case. Without this, a use case never gets going.

Use **Case** → Something a system has to do to be a success.

Start Condition → Lets you know when a use case is finished. Without this, use cases can go on forever.

Requirement → Helps you gather good requirements. Tells a story about how a system works.

Clear **Value** → How a system works when everything is going right. This is usually what customers describe when they're talking about the system.

Stop **Condition** → This is always the first step in the use case.

Main **Path** → Without this, a use case isn't worth anything to anyone. Use cases without this always fail.

Make sure you filled in all the blanks exactly like we did.

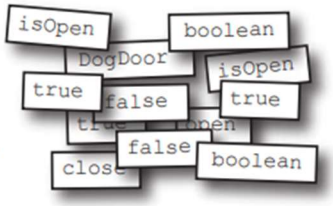
```
public class Remote {

  private DogDoor door;

  public Remote(DogDoor door) {
    this.door = door;
  }

  public void pressButton() {
    System.out.println("Pressing the remote control button...");
    if (door.isOpen()) {
      door.close();
    } else {
      door.open();
    }
  }
}
```

Here's what's leftover.



Across

4. The main path is sometimes called the _____ path.
5. A use case must have this (two words) to the user.
8. Requirements ensure that your system works _____.
10. In the dog door system, who was the external initiator?
13. Use cases focus on a _____ user goal.
14. When Fido is here, the dog door has hit its stop condition.
16. A use case is just a list of things that a system _____.
18. A use case tells a _____ about how a system works.
21. Use cases are easeist to understand when they're in this kind of language.
22. Good use cases make for _____ requirements.
24. A use case helps you understand how a system will be _____.
25. Use cases help you gather _____.

Down

1. A use case details how a system _____ with users or other systems.
2. When things go right, you're on this.
3. Use cases are all about the "_____" of your system.
6. Without use cases, you won't know if your requirements are _____.
7. If you have four goals in a system, you'll have at least this many use cases.
9. Good systems work in the _____ _____.
11. Fido does this to signal the start of the dog door system's use case.
12. When things go wrong, you end up on an _____ path.
15. Fido isn't part of this, but his dog door is.
17. This is what you should do to the customer to gather an initial set of requirements.

19. How many use cases are there in Todd and Gina's dog door?
20. Fido was chasing these when he got stuck outside.
23. A requirement documents this many needs.