

OO Design.

Case study: Guitar store

Lecture 12

Inheritance syntax reminder: upcasting

```
class Instrument {  
    public void play() {}  
    static void tune(Instrument i) {  
        // ...  
        i.play();  
    }  
}
```

```
// Wind objects are instruments  
// because they have the same interface:  
public class Wind extends Instrument {  
    public static void main(String[] args) {  
        Wind flute = new Wind();  
        Instrument.tune(flute); // Upcasting  
    }  
} ///:~
```

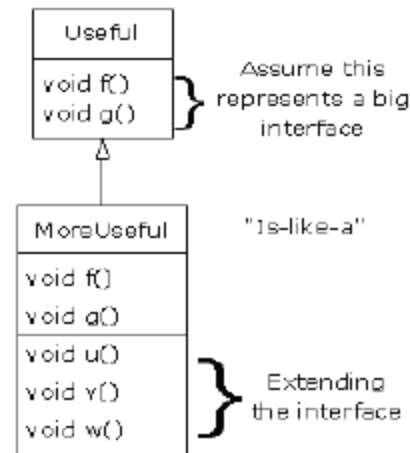
Method-call binding

- Run-time binding: late binding
- Compile-time binding: early binding

- Unless a class or a method are declared final, there is no early method binding in Java

Inheritance reminder: downcasting

- You should check if an object is really of a desired class - then you can safely downcast
- The `ClassCastException` will be thrown if trying to downcast an object of an incorrect type



```
Useful obj=new MoreUseful();  
if (obj instanceof MoreUseful)  
{  
    ((MoreUseful) obj).g();  
}
```

Reminder: interfaces

- Interfaces and abstract classes provide a more structured way to separate interface from implementation
- Abstract class provides one or more undefined methods, to be implemented by extenders
- Interface provides only method signatures
- It allows you to create classes which inherit from multiple interfaces and thus can be upcasted into different forms, according to the role

Interface syntax

```
interface Interface1 {void f();}
```

```
interface Interface2 {void g();}
```

```
class ConcreteClass implements Interface1, Inteface2{  
    void f {//implementation}  
    void g {//implementation}  
}
```

Interface example

```
interface CanFight { void fight(); }

interface CanSwim { void swim(); }

interface CanFly { void fly(); }

class ActionCharacter {
    public void fight() {}
}

class Hero extends ActionCharacter
    implements CanFight, CanSwim, CanFly
{
    public void swim() {}
    public void fly() {}
}
```

```
public class Adventure {

    public static void t(CanFight x) { x.fight(); }

    public static void u(CanSwim x) { x.swim(); }

    public static void v(CanFly x) { x.fly(); }

    public static void w(ActionCharacter x) { x.fight();
}

    public static void main(String[] args) {
        Hero h = new Hero();
        t(h); // Treat it as a CanFight
        u(h); // Treat it as a CanSwim
        v(h); // Treat it as a CanFly
        w(h); // Treat it as an ActionCharacter
    }
}
```

Example 1: Changing object behavior at run time through polymorphism

```
Interface Actor { public void act() ; }
```

```
class HappyActor implements Actor { public void act() { print("HappyActor"); } }
```

```
class SadActor implements Actor { public void act() { print("SadActor"); } }
```

```
class Stage {  
    private Actor actor = new HappyActor();  
    public void change (Actor newActor) { actor = newActor;}  
    public void performPlay() { actor.act(); }  
}
```

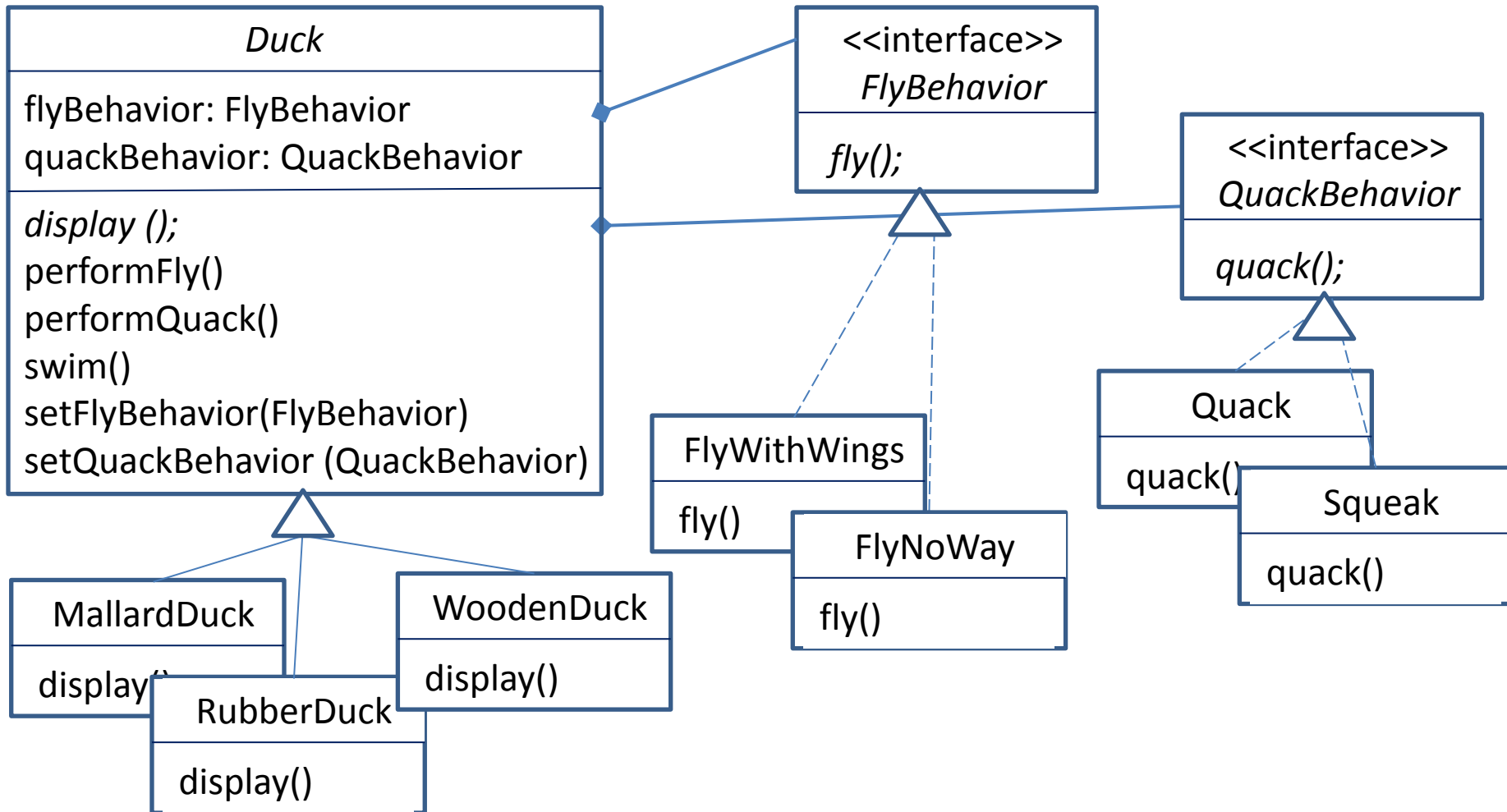
```
public class Transmogrify { public static void main(String[] args) {  
    Stage stage = new Stage();  
    stage.performPlay();  
    stage.change(new SadActor());  
    stage.performPlay();  
}
```

```
 } /* Output:
```

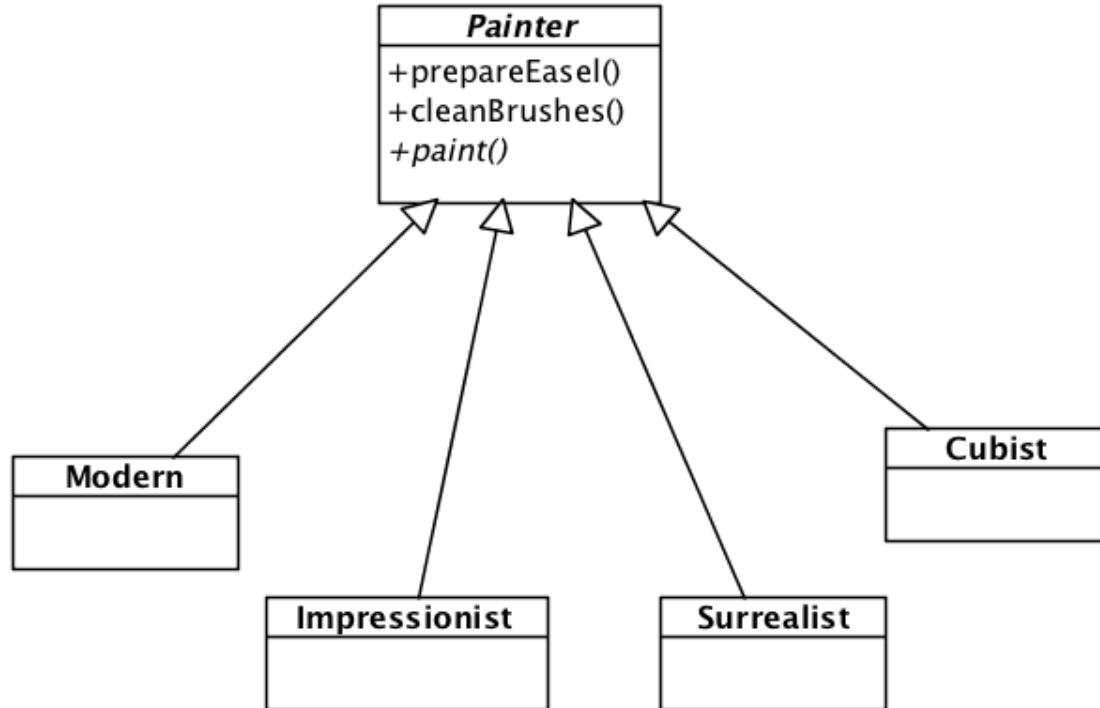
```
HappyActor
```

```
SadActor
```


Example 2: Ducks simulator design

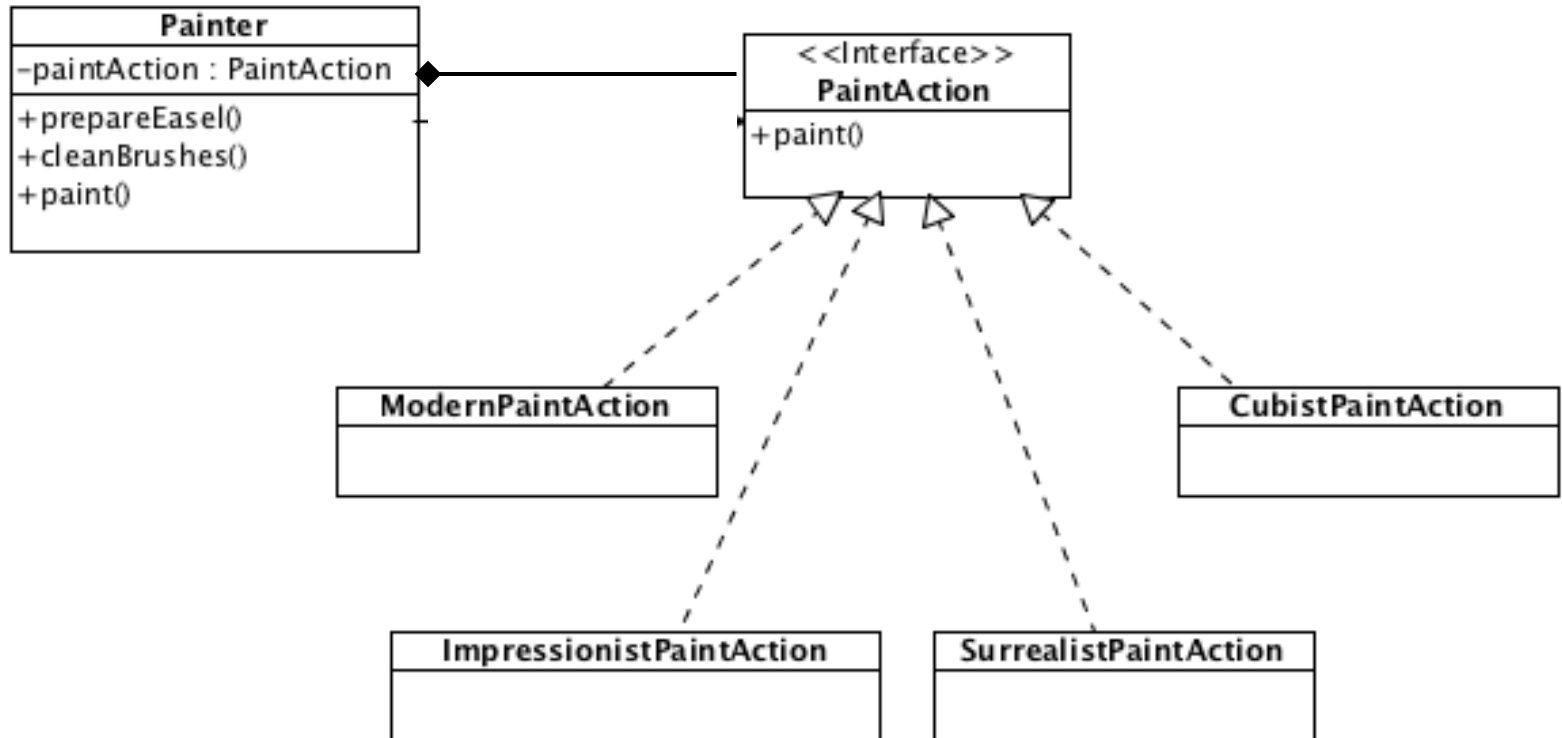


Example 3 (1/2): Painters



Every subclass has to implement *paint()*

Example 3 (2/2): Composition and encapsulation



OOP Design Principles - updated

- Encapsulate what varies and pull it away from what stays the same
- Program to an interface not to an implementation
- **Favor composition over inheritance**

Real-life application: Guitar Store

Application requirements

- Maintain a guitar inventory
- Locate guitars for customers

What your predecessor built

Guitar
serialNumber : String price : double builder : String model : String type : String backWood : String topWood : String
+getSerialNumber() : String +getPrice() : double +setPrice(newPrice : double) +getBuilder() : String +getModel() : String +getType() : String +getBackWood() : String +getTopWood() : String

Inventory
guitars : List
+addGuitar(serialNumber : String, price : double, builder : String, model : String, type : String, backWood : String)
+getGuitar(serialNumber : String) : Guitar
+searchGuitar(searchedFor : Guitar) : Guitar

A simplified view

Guitar
serialNumber : String
price : double
builder : String
model : String
type : String
backWood : String
topWood : String

Inventory
+addGuitar()
+getGuitar()
+searchGuitar()

The Guitar class

```
public class Guitar {

    private String serialNumber, builder, model, type, backWood, topWood;
    private double price;

    public Guitar(String serialNumber, double price,
                 String builder, String model, String type,
                 String backWood, String topWood) {
        this.serialNumber = serialNumber;
        this.price = price;
        this.builder = builder;
        this.model = model;
        this.type = type;
        this.backWood = backWood;
        this.topWood = topWood;
    }

    public String getSerialNumber() {return serialNumber;}
    public double getPrice() {return price;}
    public void setPrice(float newPrice) {
        this.price = newPrice;
    }

    public String getBuilder() {return builder;}
    public String getModel() {return model;}
    public String getType() {return type;}
    public String getBackWood() {return backWood;}
    public String getTopWood() {return topWood;}
}
```


The Inventory class

```
public class Inventory {
    private List guitars;
    public Inventory() { guitars = new LinkedList(); }
    public void addGuitar(String serialNumber, double price,
                          String builder, String model,
                          String type, String backWood, String
                          topWood)
    {
        Guitar guitar = new Guitar(serialNumber, price, builder,
                                   model, type, backWood, topWood);
        guitars.add(guitar);
    }
    public Guitar getGuitar(String serialNumber) {
        for (Iterator i = guitars.iterator(); i.hasNext(); ) {
            Guitar guitar = (Guitar)i.next();
            if (guitar.getSerialNumber().equals(serialNumber)) {
                return guitar;
            }
        }
        return null;
    }
}
```

The Inventory class: search

```
public Guitar search(Guitar searchGuitar) {
    for (Iterator i = guitars.iterator(); i.hasNext(); ) {
        Guitar guitar = (Guitar)i.next();
        String builder = searchGuitar.getBuilder().toLowerCase();
        if ((builder != null) && (!builder.equals("")) &&
            (!builder.equals(guitar.getBuilder().toLowerCase())))
            continue;
        String model = searchGuitar.getModel().toLowerCase();
        if ((model != null) && (!model.equals("")) &&
            (!model.equals(guitar.getModel().toLowerCase())))
            continue;
        String type = searchGuitar.getType().toLowerCase();
        if ((type != null) && (!searchGuitar.equals("")) &&
            (!type.equals(guitar.getType().toLowerCase())))
            continue;
        String backWood = searchGuitar.getBackWood().toLowerCase();
        if ((backWood != null) && (!backWood.equals("")) &&
            (!backWood.equals(guitar.getBackWood().toLowerCase())))
            continue;
        String topWood = searchGuitar.getTopWood().toLowerCase();
        if ((topWood != null) && (!topWood.equals("")) &&
            (!topWood.equals(guitar.getTopWood().toLowerCase())))
            continue;
        return guitar;
    }
    return null;
}
```

The search does not find guitars even if they are in stock. What can be done?

```
Guitar dreamGuitar=new Guitar(null, 0,"fender",  
"Stratocaster", "electric guitar", null, null);  
Guitar matchingGuitar=inventory.search(dreamGuitar);  
//returns null
```

But the matching guitar exists

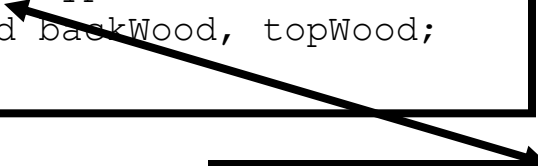
```
inventory.addGuitar("V95693", 1499, "Fender",  
"Stratocaster", "electric", "Alder","Alder");
```

To do list:

1. If there are guitars in stock that fit the customer's needs, always find them.
2. Take into account typing mistakes by the customer or make it impossible to enter erroneous data.

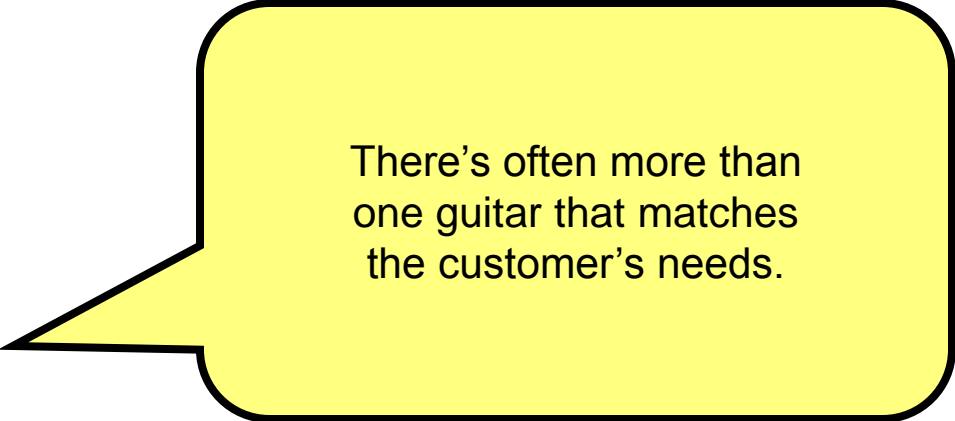
Improvement 1: Remove strings where possible

```
public class Guitar {  
  
    private String serialNumber,  
                model;  
    private double price;  
    private Builder builder;  
    private Type type;  
    private Wood backWood, topWood;  
  
    ...  
}
```

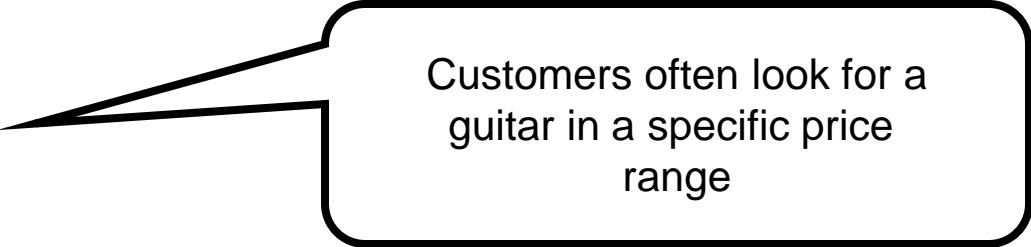


```
public enum Type {  
  
    ACOUSTIC, ELECTRIC;  
  
    public String toString() {  
        switch(this) {  
            case ACOUSTIC: return "acoustic";  
            case ELECTRIC: return "electric";  
            default:      return "unspecified";  
        }  
    }  
}
```

The owner says



There's often more than one guitar that matches the customer's needs.



Customers often look for a guitar in a specific price range

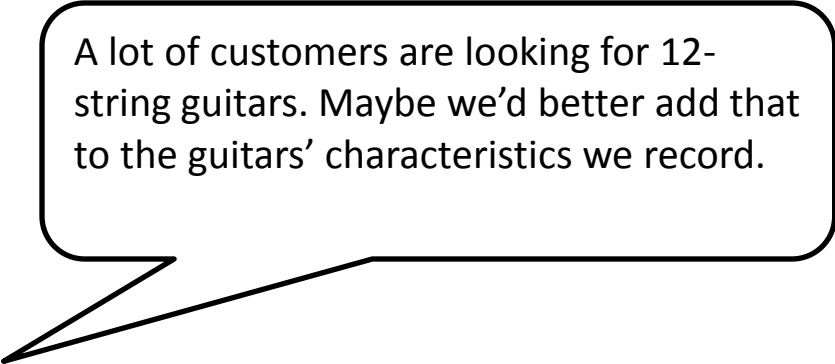
To do list:

1. If there are guitars in stock that fit the customer's needs, always find them.
2. Take into account typing mistakes by the customer or make it impossible to enter erroneous data.
3. **Find ALL matching guitars.**

Improvement 2. List of matching guitars

```
public List search(Guitar searchGuitar) {
    List matchingGuitars = new LinkedList();
    for (Iterator i = guitars.iterator(); i.hasNext(); ) {
        Guitar guitar = (Guitar)i.next();
        // Ignore serial number since that's unique
        // Ignore price since that's unique
        if (searchGuitar.getBuilder() != guitar.getBuilder())
            continue;
        String model = searchGuitar.getModel().toLowerCase();
        if ((model != null) && (!model.equals("")) &&
            (!model.equals(guitar.getModel().toLowerCase())))
            continue;
        if (searchGuitar.getType() != guitar.getType())
            continue;
        if (searchGuitar.getBackWood() != guitar.getBackWood())
            continue;
        if (searchGuitar.getTopWood() != guitar.getTopWood())
            continue;
        matchingGuitars.add(guitar);
    }
    return matchingGuitars;
}
```

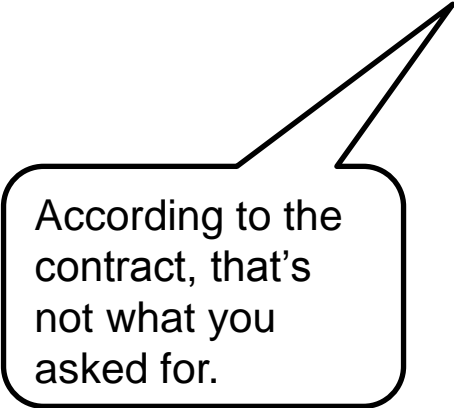

Change challenge 1



A lot of customers are looking for 12-string guitars. Maybe we'd better add that to the guitars' characteristics we record.

The owner:

The programmer



According to the contract, that's not what you asked for.

What do you need to change?

1 GuitarSpec

2 Inventory

3

4

Incremental changes

```
public class GuitarSpec {  
  
    private Builder builder;  
    private String model;  
    private Type type;  
    private Wood backWood;  
    private Wood topWood;  
  
    public GuitarSpec(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;  
    }  
  
    public Builder getBuilder() {  
        return builder;  
    }  
    }...
```

Incremental changes (2)

```
public class Guitar {

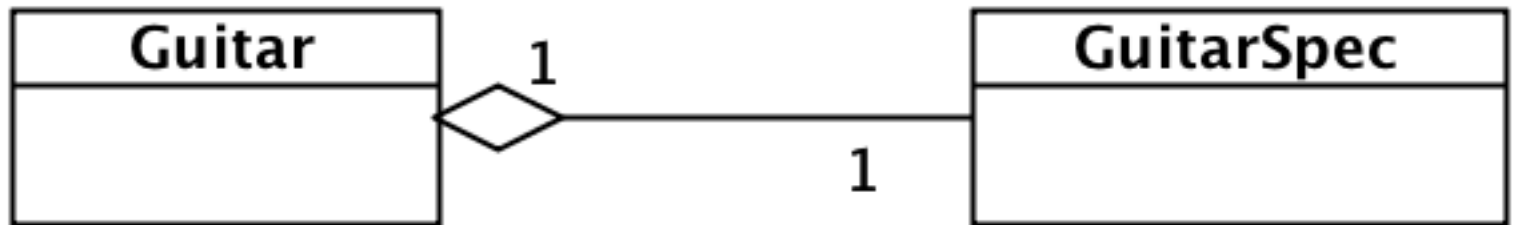
    private String serialNumber;
    private double price;
    GuitarSpec spec;

    public Guitar(String serialNumber, double price,
                 Builder builder, String model, Type type,
                 Wood backWood, Wood topWood) {
        this.serialNumber = serialNumber;
        this.price = price;
        this.spec = new GuitarSpec(builder, model, type, backWood, topWood);
    }

    ...

    public GuitarSpec getSpec() {
        return spec;
    }
}
```

Does this make sense?



We do not want to create a new instance of `GuitarSpec` inside `Guitar` constructor, because then we need to make changes in `Guitar` class each time we update spec.

It is rather



Changes to GuitarSpec

```
public class GuitarSpec {

    private Builder builder;
    private String model;
    private Type type;
    private int numStrings;
    private Wood backWood;
    private Wood topWood;

    public GuitarSpec(Builder builder, String model, Type type,
                     int numStrings, Wood backWood, Wood topWood) {
        this.builder = builder;
        this.model = model;
        this.type = type;
        this.numStrings = numStrings;
        this.backWood = backWood;
        this.topWood = topWood;
    }
    ...

    public int getNumStrings() {
        return numStrings;
    }
}
```

Changes to Inventory

```
public Iterator<Guitar> search(GuitarSpec searchSpec) {
    List<Guitar> matchingGuitars = new LinkedList<Guitar>();
    for (Iterator<Guitar> i = guitars.iterator(); i.hasNext(); ) {
        Guitar guitar = i.next();
        GuitarSpec spec = guitar.getSpec();
        if (searchSpec.getBuilder() != spec.getBuilder())
            continue;
        String model = searchSpec.getModel().toLowerCase();
        if ((model != null) && (!model.equals("")) &&
            (!model.equals(spec.getModel().toLowerCase())))
            continue;
        if (searchSpec.getType() != spec.getType())
            continue;
        if (searchSpec.getBackWood() != spec.getBackWood())
            continue;
        if (searchSpec.getTopWood() != spec.getTopWood())
            continue;
        if (searchSpec.getNumStrings() != spec.getNumStrings())
            continue;
        matchingGuitars.add(guitar);
    }
    return matchingGuitars.iterator();
}
32 }
```


OOP design problem



Why did we have to
modify two files to
make the change?



Maybe we've allocated the responsibilities incorrectly. There's a pattern called *Information Expert* that might be appropriate here.

O-O design is all about assigning responsibilities to the right objects

Information Expert

Assign responsibility to the class that has the essential information—the information expert.

Craig Larman, “Applying UML and Patterns”

Think about this?

What behavior is misplaced?

Matching the guitar to the specification

Who is the information expert?

GuitarSpec

What should we do?

Make GuitarSpec responsible for determining

if it matches a guitar.

The new GuitarSpec class

```
public class GuitarSpec {  
    ...  
    public boolean matches(GuitarSpec otherSpec) {  
        if (builder != otherSpec.builder)  
            return false;  
        if ((model != null) && (!model.equals("")) &&  
            (!model.toLowerCase().equals(otherSpec.model.toLowerCase())))  
            return false;  
        if (type != otherSpec.type)  
            return false;  
        if (numStrings != otherSpec.numStrings)  
            return false;  
        if (backWood != otherSpec.backWood)  
            return false;  
        if (topWood != otherSpec.topWood)  
            return false;  
        return true;  
    }  
    ...  
}
```

The new Inventory class

```
Public class Inventory {  
...  
    public List search(GuitarSpec searchSpec) {  
        List matchingGuitars = new LinkedList();  
        for (Iterator i = guitars.iterator(); i.hasNext(); ) {  
            Guitar guitar = (Guitar)i.next();  
            if (guitar.getSpec().matches(searchSpec))  
                matchingGuitars.add(guitar);  
        }  
        return matchingGuitars;  
    }  
...  
}
```

Software design

Software design is the process of planning how to solve a problem through software.

A software design contains enough information for a development team to implement the solution. It is the embodiment of the plan (the blueprint for the software solution).

What makes a design good?

1 Easy to understand

2 Flexible, easy to change

3 Satisfies the requirements (now and in the future)

4 ...

Change challenge 2

Business is great. I'm ready to expand and I need some changes to my application.

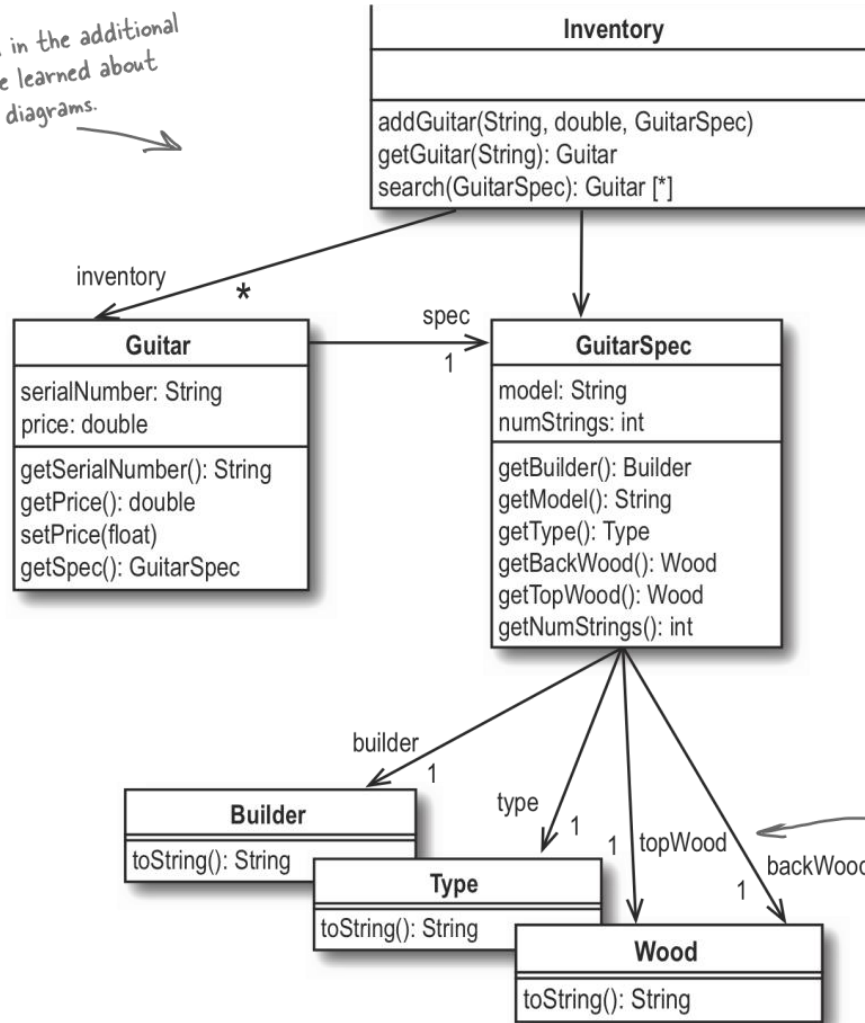
The business owner



I've added mandolins to the type of instruments I'm selling. Can you change the app to handle this?

Review of our application

We've added in the additional things you've learned about UML class diagrams.

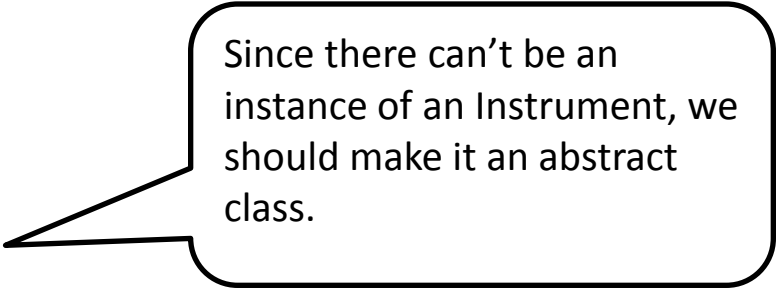


We've moved most of the properties out of the class box and used associations instead.

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.

Change challenge 2: possible solutions

Solution	Advantages	Disadvantages
Mandolin class	Simple to implement	<ul style="list-style-type: none">•Duplicate code•Hard to maintain
Instrument class with type field	No duplicate code	<ul style="list-style-type: none">•Not an O-O solution•Need to check type on objects
Instrument base class	<ul style="list-style-type: none">•O-O solution•No type field to check•No duplicate code	Does the owner have an Instrument in the inventory?

A speech bubble with a black outline and rounded corners. It has a tail pointing to the left. The text inside is black and centered.

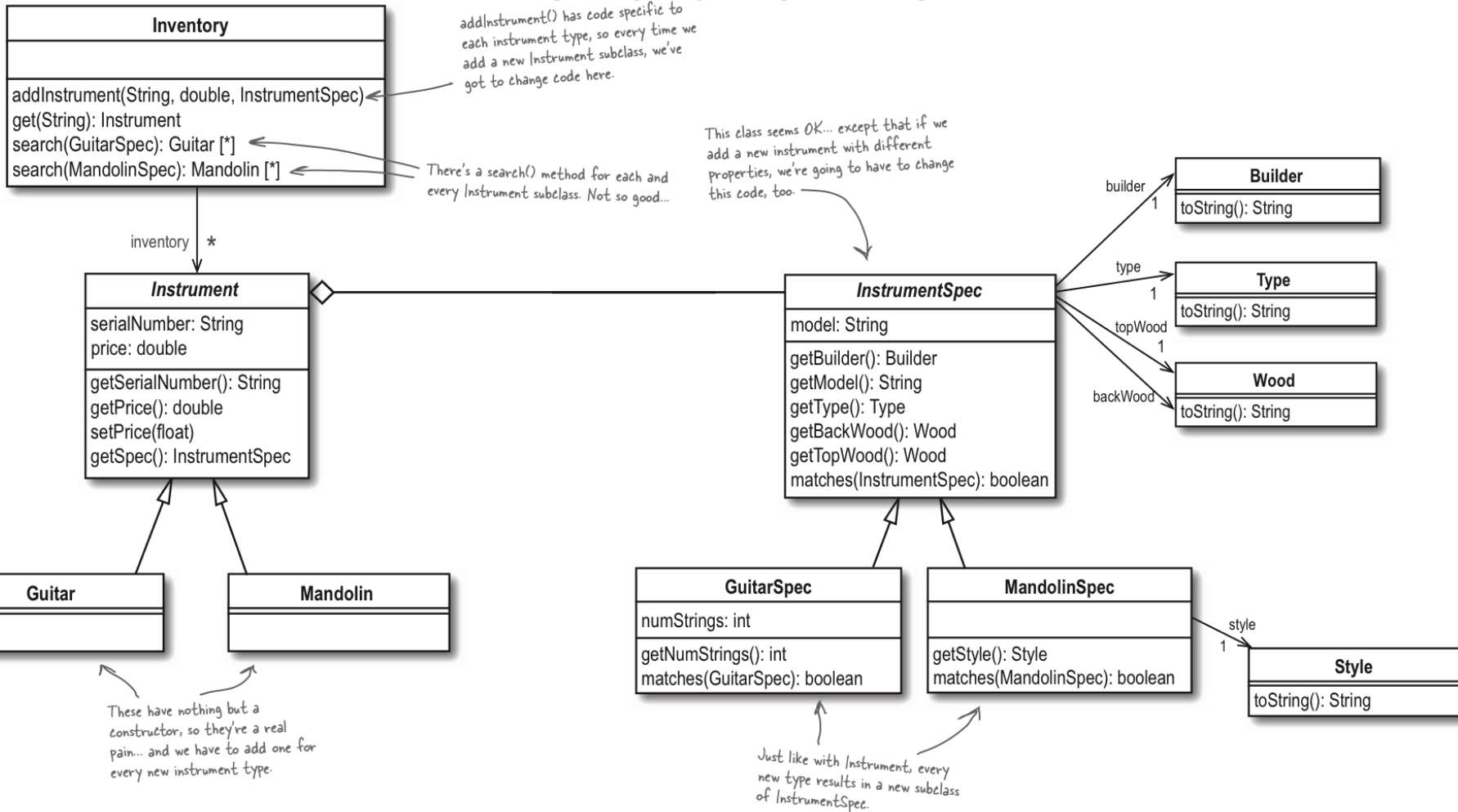
Since there can't be an instance of an Instrument, we should make it an abstract class.

Abstract classes encapsulate shared behavior and define the protocol for all subclasses

It's not quite this simple

- Things still need to be done
 - Make the Inventory class use Instrument rather than Guitar
 - Add an InstrumentSpec class
 - Abstract base class for the GuitarSpec
 - Add a MandolinSpec class derived from InstrumentSpec
 - Make the Inventory class use InstrumentSpec instead of GuitarSpec

Where are we?



The current search code in Inventory class: method overloading

```
public List search(GuitarSpec searchSpec) {
    List matchingGuitars = new LinkedList();
    for (Iterator i = inventory.iterator(); i.hasNext(); ) {
        Guitar guitar = (Guitar)i.next();
        if (guitar.getSpec().matches(searchSpec))
            matchingGuitars.add(guitar);
    }
    return matchingGuitars;
}
```

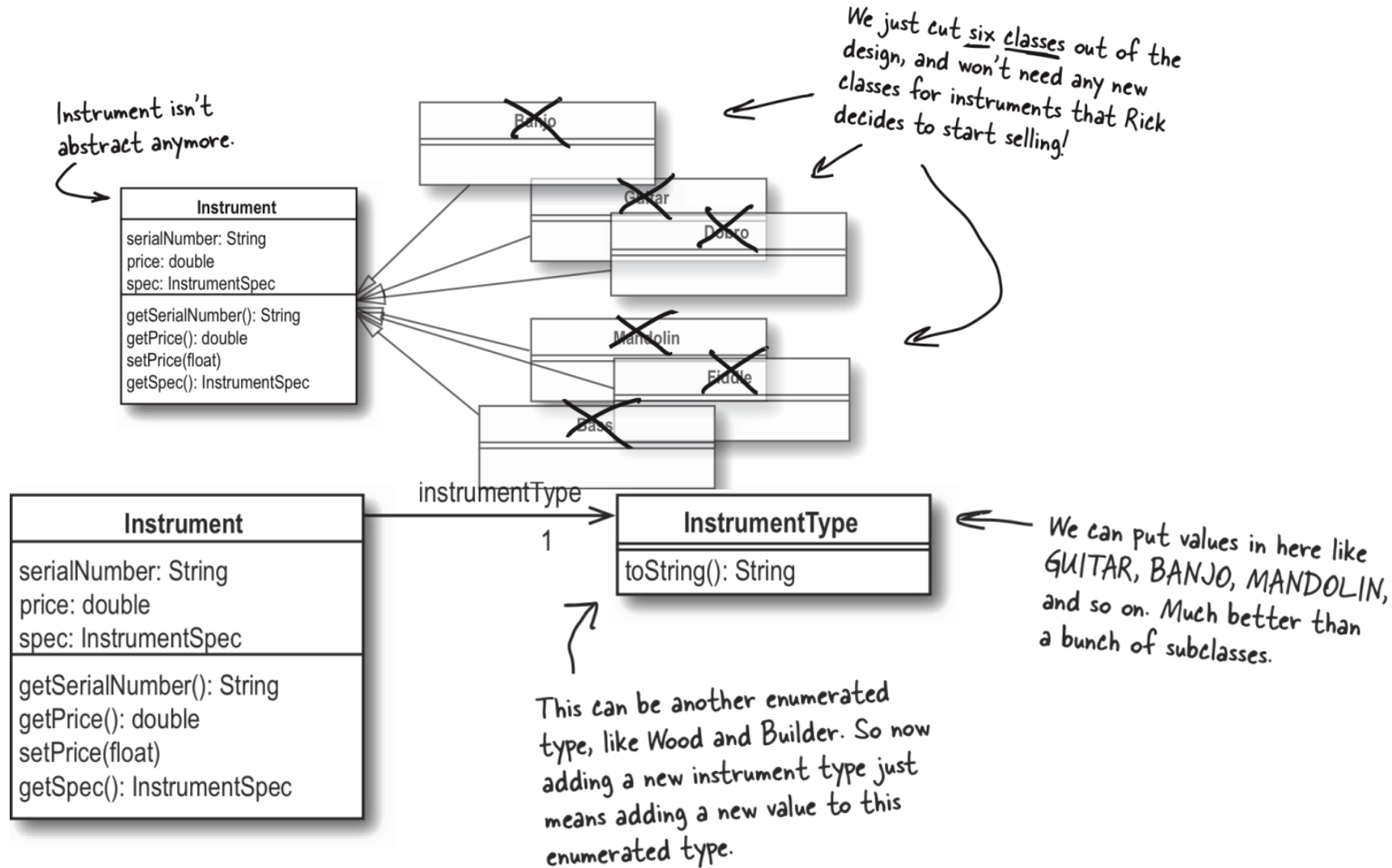
```
public List search(MandolinSpec searchSpec) {
    List matchingMandolins = new LinkedList();
    for (Iterator i = inventory.iterator(); i.hasNext(); ) {
        Mandolin mandolin = (Mandolin)i.next();
        if (mandolin.getSpec().matches(searchSpec))
            matchingMandolins.add(mandolin);
    }
    return matchingMandolins;
}
```

Code to an interface

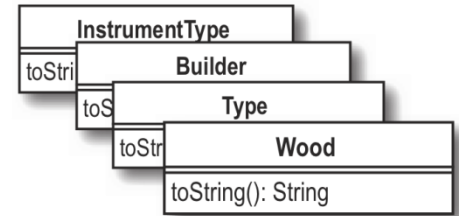
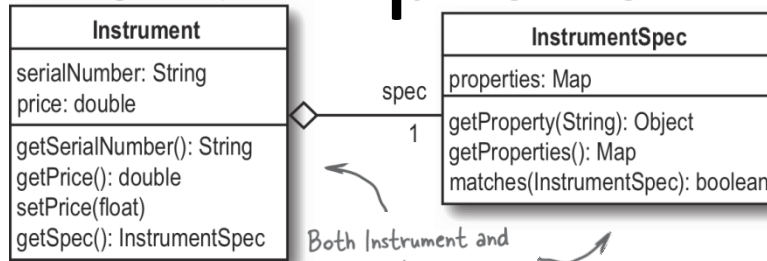
(or an abstract class)

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


Our improved design

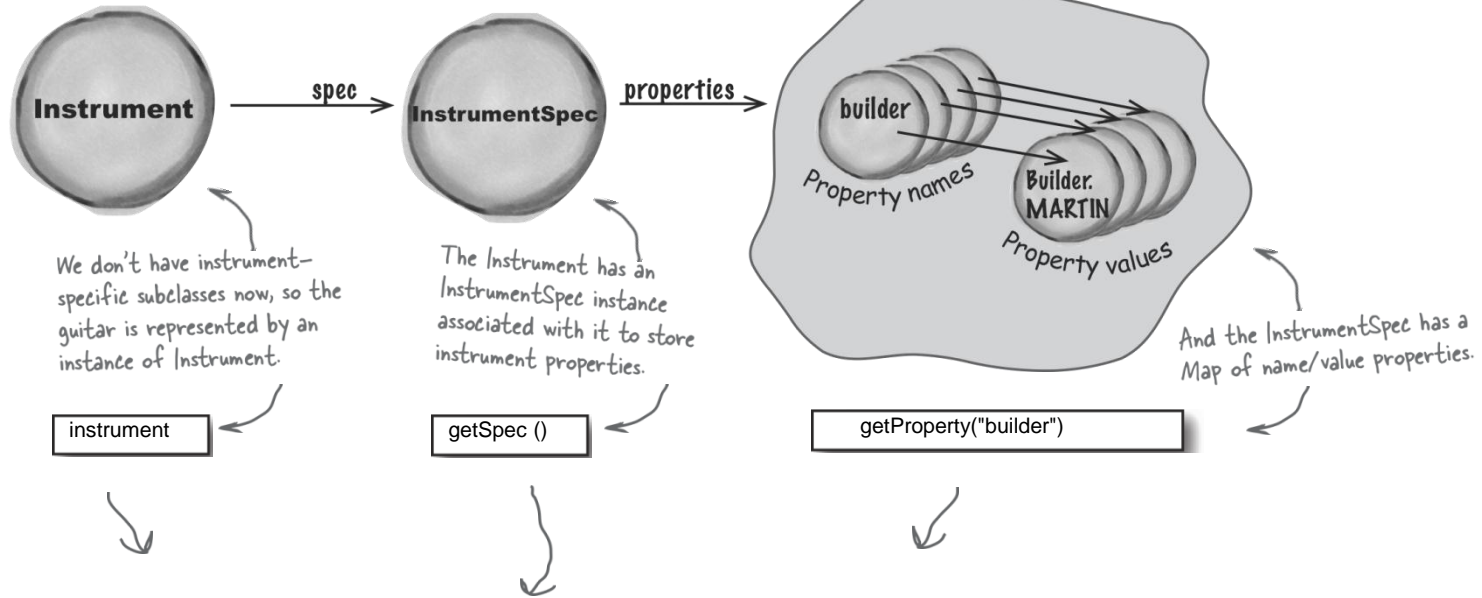


Our next improvement



Both Instrument and InstrumentSpec are no longer abstract.

InstrumentSpec's Map uses these enumerated types.

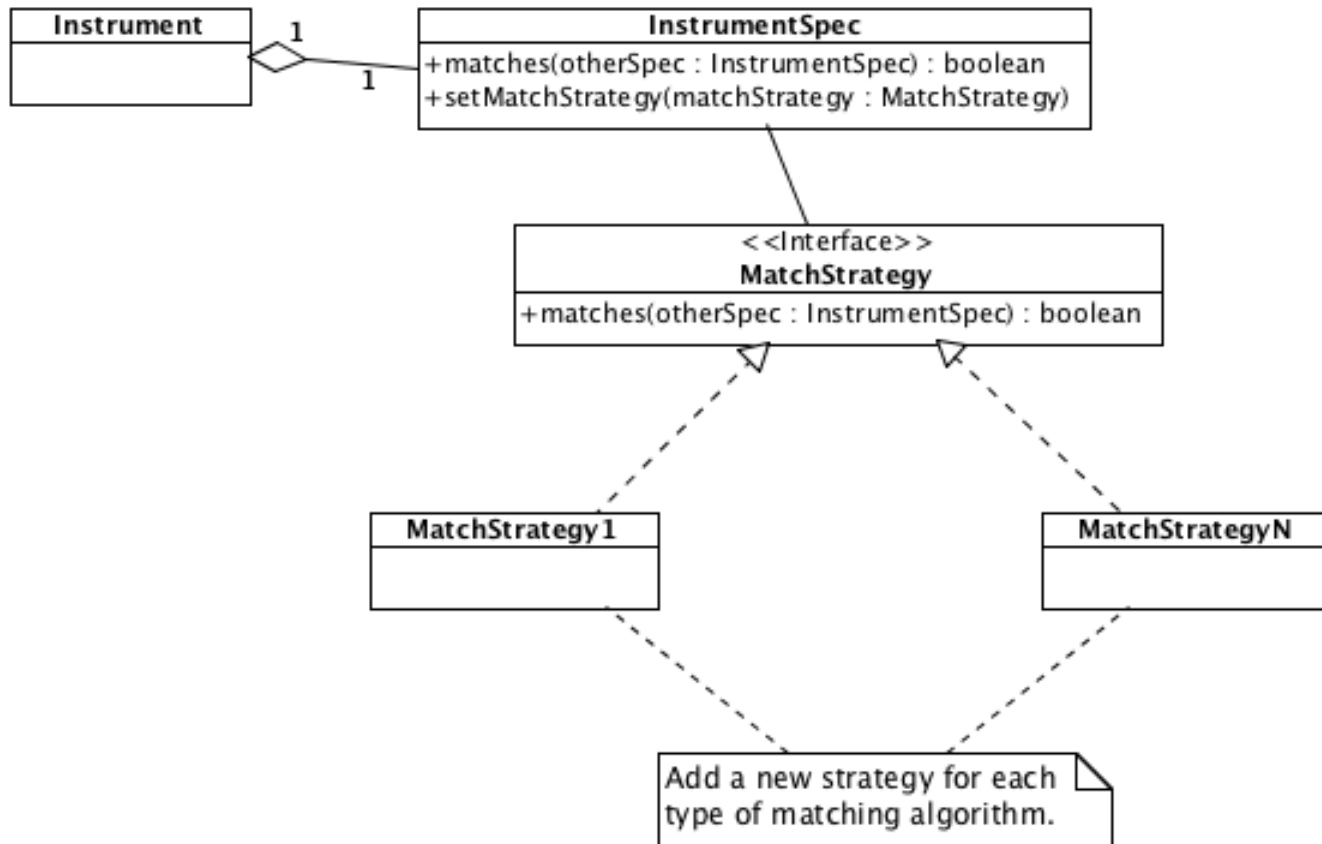


```
instrument.getSpec().getProperty("builder");
```

It depends upon what “matches” is!

```
public boolean matches(InstrumentSpec otherSpec) {
    for (Iterator i = otherSpec.getProperties().keySet().iterator();
         i.hasNext(); ) {
        String propertyName = (String)i.next();
        if (!properties.get(propertyName).equals(
            otherSpec.getProperty(propertyName))) {
            return false;
        }
    }
    return true;
}
```

The Strategy design pattern



Design home quiz

- Design an application for home books inventory. The requirements: maintain a book inventory and book search
- Draw a UML diagram to show your design
 - Provide any additional documentation you think is appropriate
- Summarize which design principles and OOP concepts have been applied