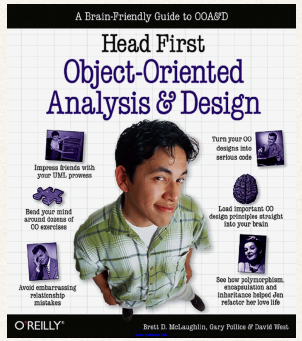
Un assaggio di OOA&D

Alberto Gianoli

Univ. Ferrara - Corso di Laurea in Informatica

Dove?



- Head first object oriented analysis & design

  qui vi espongo il primo capitolo, ma se vi dedicherete alla programmazione OO vi consiglio vivamente di leggervi tutto il libro

Lavoriamo su un caso pratico

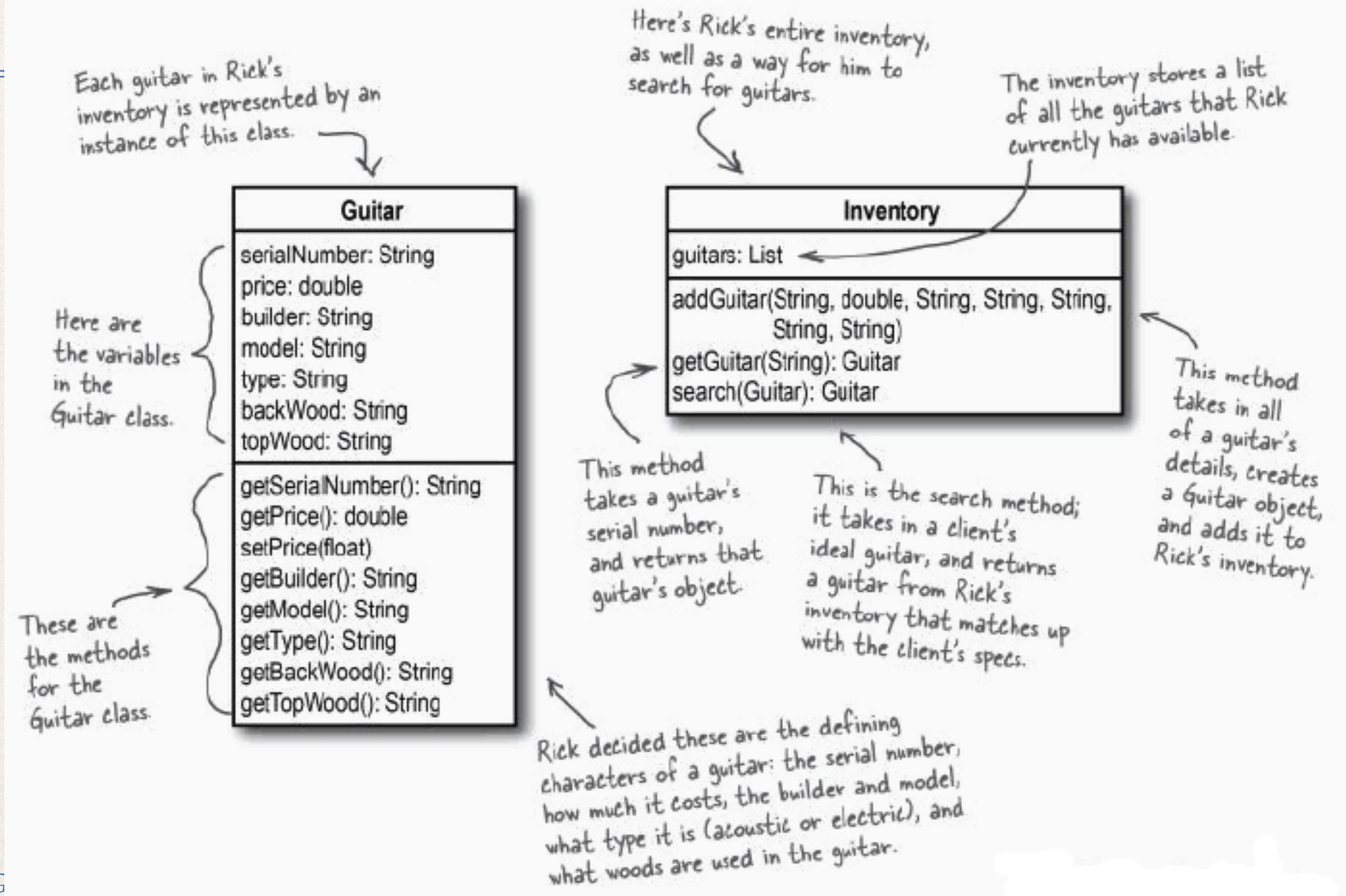
Applicazione esistente:

- problemi: non fa quello che dovrebbe fare
- migliorare

Il negozio di chitarre di Rick

- ha fatto il grande passo: da un sistema cartaceo a un sistema informatico per tenere traccia delle chitarre che ha in magazzino
- si è rivolto alla (nota) ditta Down&Dirty'R'Us
- gli hanno sviluppato una applicazione di magazzino e un search tool per trovare (se ce l'ha) qualcosa che corrisponda alle richieste dei suoi clienti

Le classi

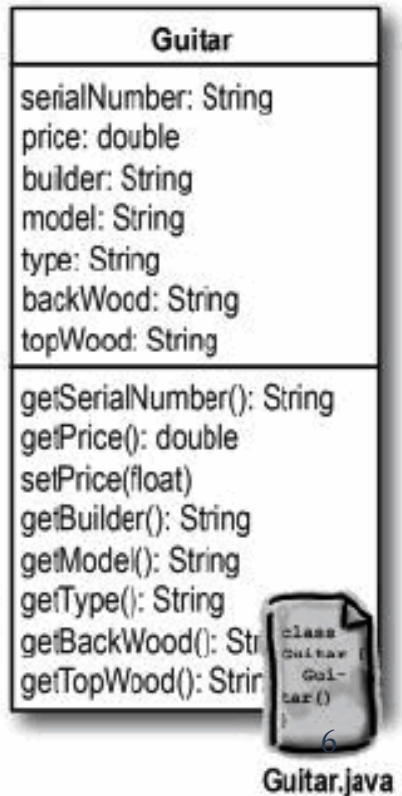

Each guitar in Rick's inventory is represented by an instance of this class.
Here's Rick's entire inventory, as well as a way for him to search for guitars.
The inventory stores a list of all the guitars that Rick currently has available.
Guitar
serialNumber: String
price: double
builder: String
model: String
type: String
backWood: String
topWood: String
getSerialNumber(): String
getPrice(): double
setPrice(float)
getBuilder(): String
getModel(): String
getType(): String
getBackWood(): String
getTopWood(): String
Inventory
guitars: List
addGuitar(String, double, String, String, String, String, String)
getGuitar(String): Guitar
search(Guitar): Guitar
Here are the variables in the Guitar class.
These are the methods for the Guitar class.
This method takes a guitar's serial number, and returns that guitar's object.
This is the search method; it takes in a client's ideal guitar, and returns a guitar from Rick's inventory that matches up with the client's specs.
This method takes in all of a guitar's details, creates a Guitar object, and adds it to Rick's inventory.
Rick decided these are the defining characters of a guitar: the serial number, how much it costs, the builder and model, what type it is (acoustic or electric), and what woods are used in the guitar.

These are all the properties we saw from the class diagram for 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;
  }
}
```

UML class diagrams don't show constructors; the Guitar constructor does just what you'd expect, though: sets all the initial properties for a new Guitar.

You can see how the class diagram matches up with the methods in the Guitar class's code.



```
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;
  }
```

Remember, we've stripped out the import statements to save some space.

addGuitar() takes in all the properties required to create a new Guitar instance, creates one, and adds it to the inventory.

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

This method is a bit of a mess... it compares each property of the Guitar object it's passed in to each Guitar object in Rick's inventory.

Si perdono clienti...

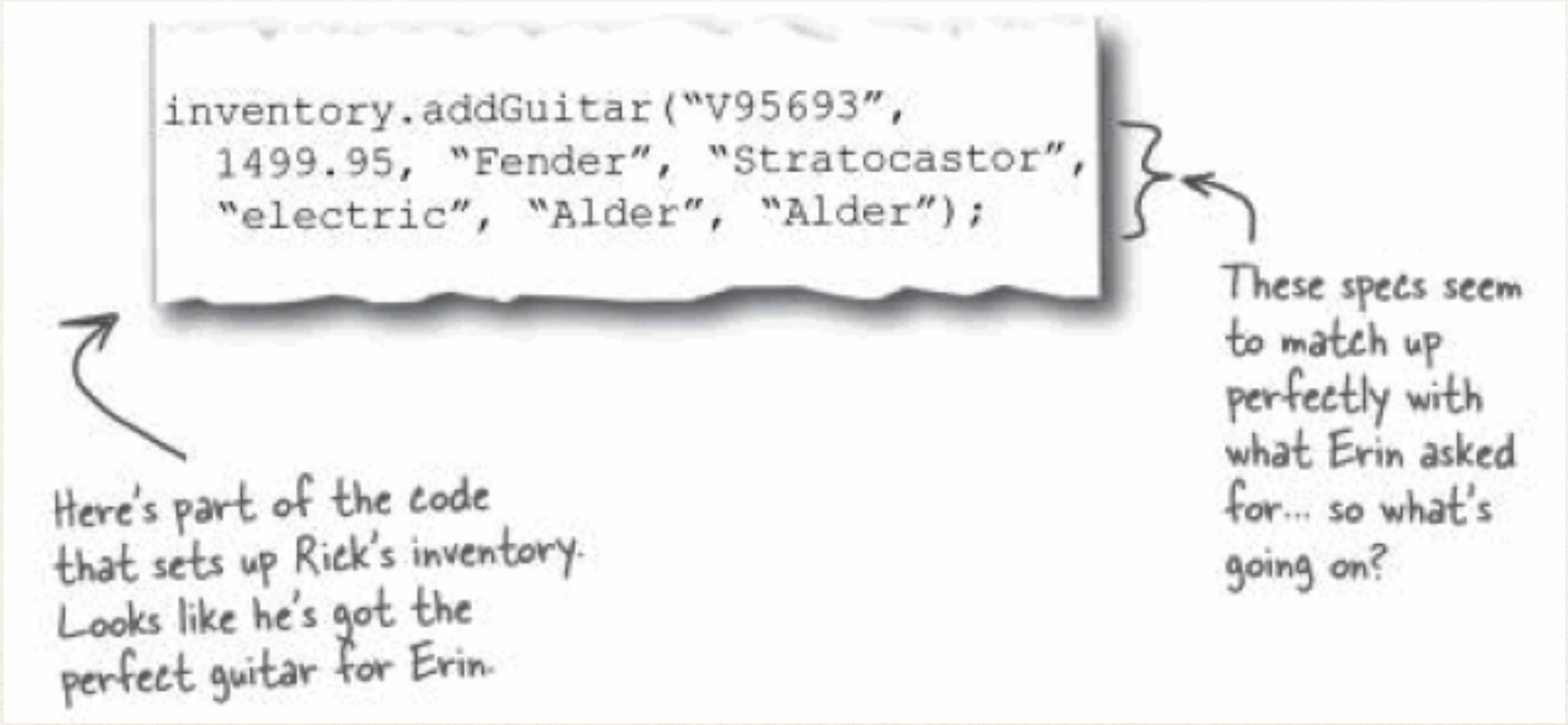
```
public class FindGuitarTester {

  public static void main(String[] args) {
    // Set up Rick's guitar inventory
    Inventory inventory = new Inventory();
    initializeInventory(inventory);

    Guitar whatErinLikes = new Guitar("", 0, "fender", "Stratocastor",
                                      "electric", "Alder", "Alder");
    Guitar guitar = inventory.search(whatErinLikes);
    if (guitar != null) {
      System.out.println("Erin, you might like this " +
        guitar.getBuilder() + " " + guitar.getModel() + " " +
        guitar.getType() + " guitar:\n   " +
        guitar.getBackWood() + " back and sides,\n   " +
        guitar.getTopWood() + " top.\nYou can have it for only $" +
        guitar.getPrice() + "!");
    } else {
      System.out.println("Sorry, Erin, we have nothing for you.");
    }
  }

  private static void initializeInventory(Inventory inventory) {
    // Add guitars to the inventory...
  }
}
```

Erin is looking for an Fender "Strat" guitar, made of Alder.

```
File Edit Window Help C7#5
%java FindGuitarTester

Sorry, Erin, we have nothing for you.
```

Si perdono clienti per colpa del programma...

questo capita nonostante ci sia una chitarra in inventario che può andare bene...



Quale è il problema?

Esaminiamo l'applicazione

- classe Guitar: tutte quelle stringhe, è terribile; possiamo usare oggetti o costanti?
- hmmm le note del proprietario del negozio dicono che i clienti vogliono avere possibilità di scelta: search() non dovrebbe restituire una lista di chitarre?
- il design fa un po' schifo, Inventory e Guitar dipendono troppo l'una dall'altra: dobbiamo ristrutturare

Da dove cominciamo se vogliamo del software "fatto come si deve"?

Software “fatto come si deve”... Che vuol dire? Chi ha ragione?

- Customer friendly programmer: il software fa quello che vuole il cliente; anche se il cliente pensa a un nuovo modo di usare il programma, questo non deve cedere o dare risultati inaspettati
- Object oriented programmer: il software deve essere orientato agli oggetti; bisogna ridurre la duplicazione di codice, ogni oggetto deve essere in controllo del proprio comportamento, il programma deve essere facile da modificare perchè il design è robusto e flessibile
- Design guru programmer: il software deve utilizzare design pattern e principi validi e noti; gli oggetti devono essere scarsamente accoppiati, il codice deve essere chiuso a modifiche ma aperto per estensioni, insomma semplice da riutilizzare

Software "fatto come si deve"... Che vuol dire? Chi ha ragione?

1. Assicurarsi che il software faccia ciò che il cliente vuole.

Il cliente viene prima di tutto: attenzione alla raccolta e analisi dei requisiti

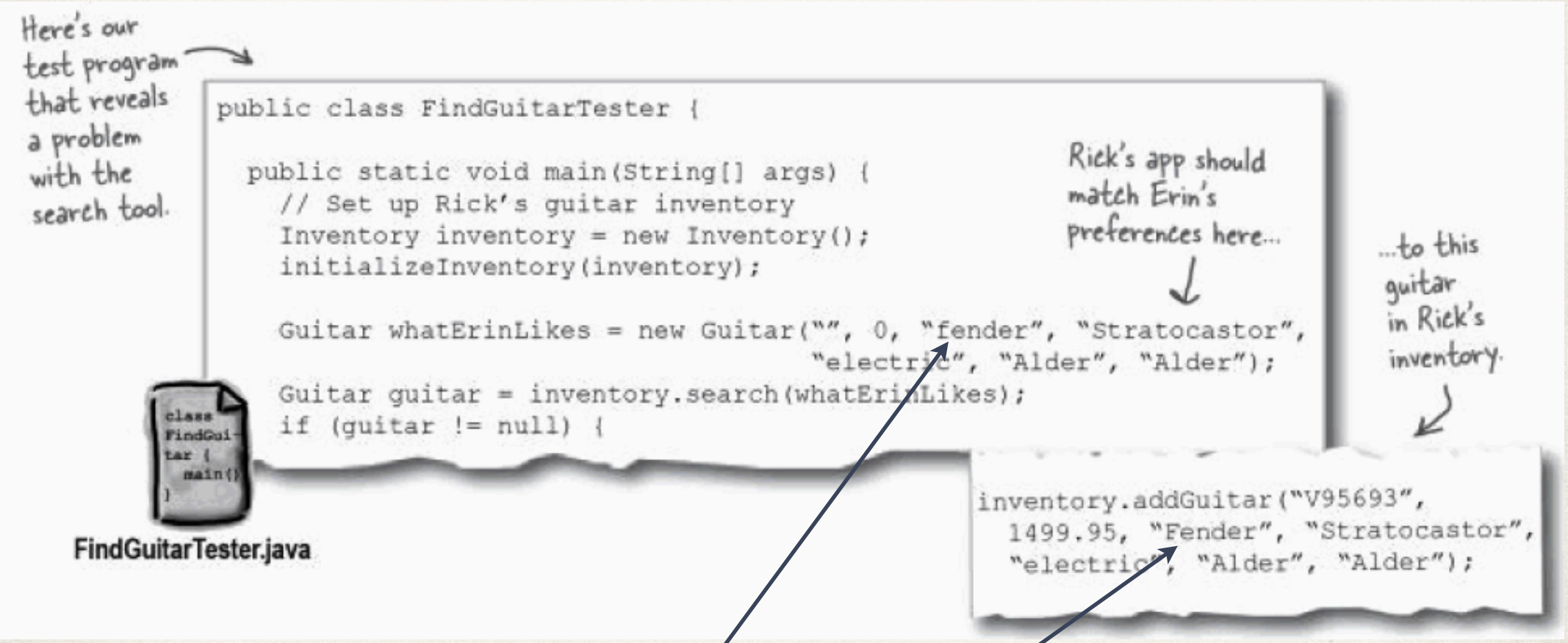
2. Applicare principi base di OO e aggiungere flessibilità

Quando il software funziona, cerca codice duplicato e applica tecniche OO

3. Punta a un design mantenibile e riusabile.

Applica principi e pattern per rendere il codice riusabile

Torniamo all'applicazione...



fender vs Fender

Non creiamo problemi per risolvere problemi...

- aggiungiamo LowerCase() e assicuriamoci che tutte le stringhe siano in minuscolo..... hmmm risolve il problema, ma è una buona idea? Ci sono alternative migliori?

- è necessario avere delle stringhe? non sono tutte costanti? Potremmo usare degli enumerated types

La prima soluzione è l'equivalente di una pezza messa in fretta e furia. La seconda soluzione non risolve solo il problema, ma rende più robusto il programma.

These are all Java enums, enumerated types that function sort of like constants.

```
public enum Type {

  ACOUSTIC, ELECTRIC;

  public String toString() {
    switch(this) {
      case ACOUSTIC: return "acoustic";
      cas
      def
    }
  }
}
```

Type.java

Each enum takes the place of one of the guitar properties that is standard across all guitars.

```
public enum Builder {

  FENDER, MARTIN, GIBSON, COLLINGS,
  OLSON, RYAN, PRS, ANY;

  public String toString() {
    switch(this) {
      case FENDER:   return "Fender";
      case
      case
      case
```

Builder.java

We can refer to these as Wood.SITKA, or Builder.GIBSON, and avoid all those string comparisons completely.

```
public enum Wood {

  INDIAN_ROSEWOOD, BRAZILIAN_ROSEWOOD, MAHOGANY,
  MAPLE, COCOBOLO, CEDAR, ADIRONDACK, ALDER, SITKA;

  public String toString() {
    switch(this) {
      case INDIAN_ROSEWOOD:
        return "Indian Rosewood";
      case BRAZILIAN_ROSEWOOD:
        return "Brazilian Rosewood";
```

Wood.java

One of the big advantages of using enums is that it limits the possible values you can supply to a method... no more misspellings or case issues.

```
public class FindGuitarTester {

  public static void main(String[] args) {
    // Set up Rick's guitar inventory
    Inventory inventory = new Inventory();
    initializeInventory(inventory);

    Guitar whatErinLikes = new Guitar("", 0, Builder.FENDER,
      "Stratocastor", Type.ELECTRIC, Wood.ALDER, Wood.ALDER);
    Guitar guitar = inventory.search(whatErinLikes);
    if (guitar != null) {
```

We can replace all those String preferences with the new enumerated type values.

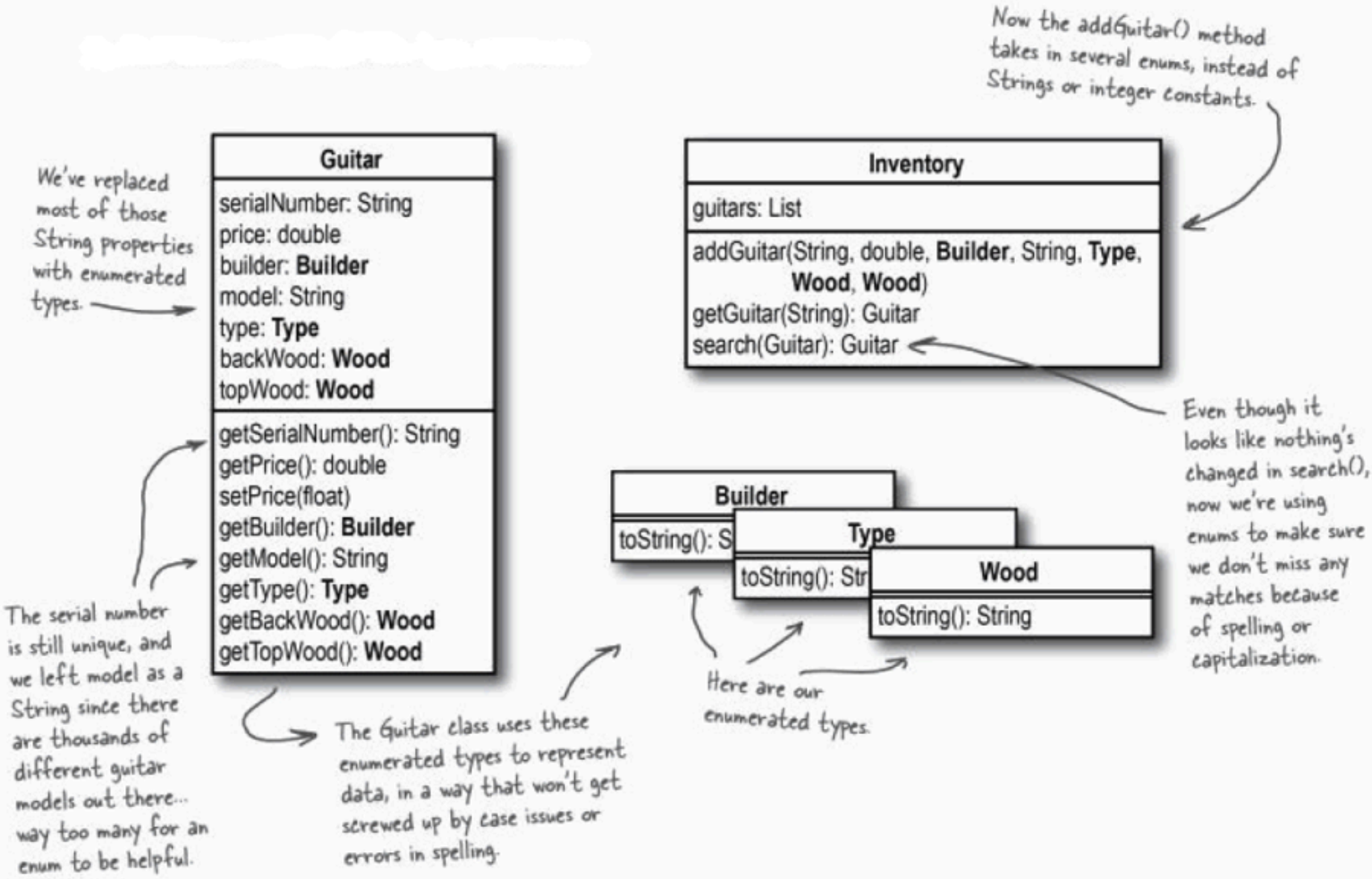
The only String left is for the model, since there really isn't a limited set of these like there is with builders and wood.


```
public Guitar search(Guitar searchGuitar) {
  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;
    return guitar;
  }
  return null;
}
```

The only property that we need to worry about case on is the model, since that's still a String.

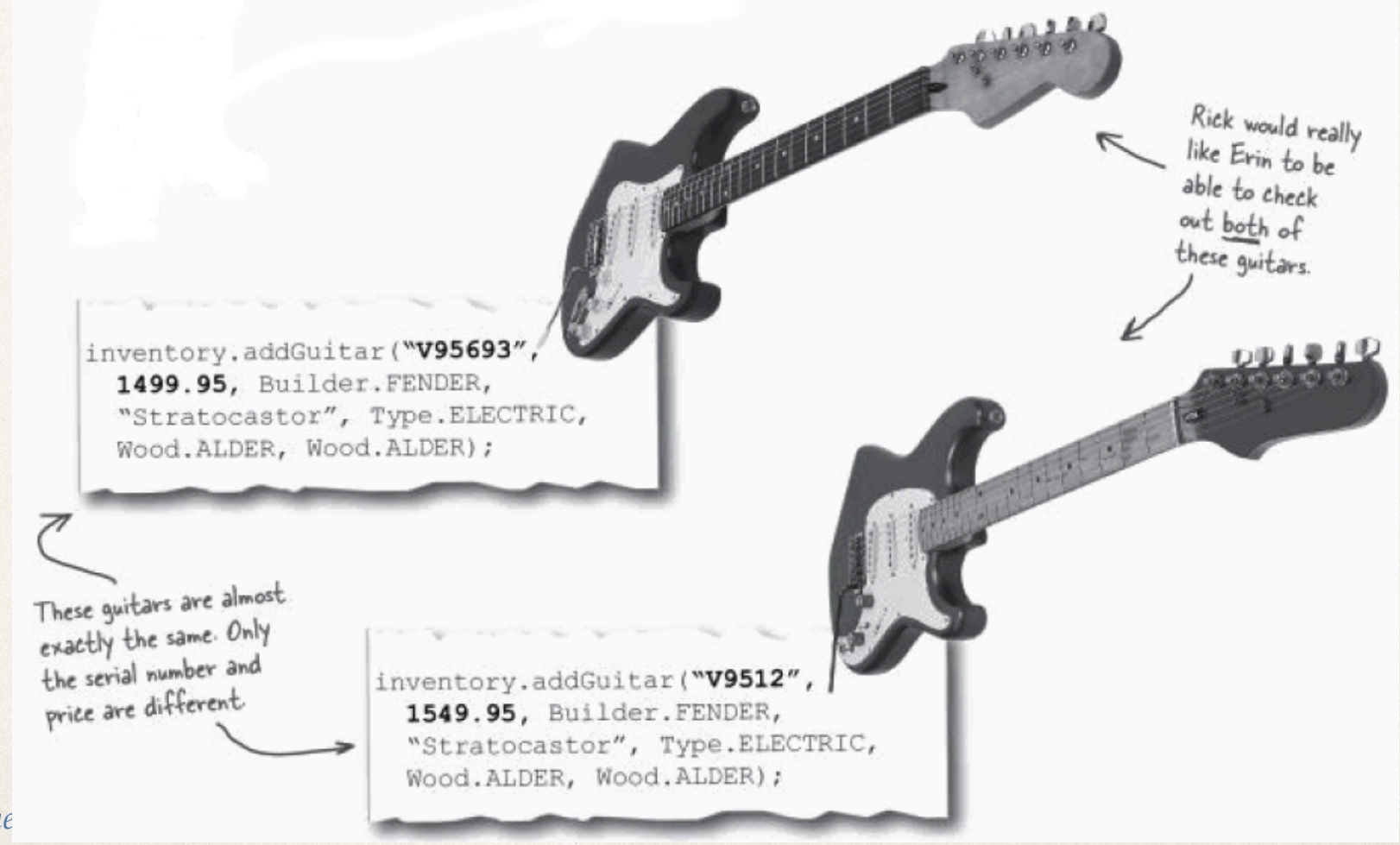
It looks like nothing has changed, but with enums, we don't have to worry about these comparisons getting screwed up by misspellings or case issues.

Inventory.java


Now the addGuitar() method takes in several enums, instead of Strings or integer constants.
We've replaced most of those String properties with enumerated types.
Guitar
serialNumber: String
price: double
builder: Builder
model: String
type: Type
backWood: Wood
topWood: Wood
getSerialNumber(): String
getPrice(): double
setPrice(float)
getBuilder(): Builder
getModel(): String
getType(): Type
getBackWood(): Wood
getTopWood(): Wood
Inventory
guitars: List
addGuitar(String, double, Builder, String, Type, Wood, Wood)
getGuitar(String): Guitar
search(Guitar): Guitar
Even though it looks like nothing's changed in search(), now we're using enums to make sure we don't miss any matches because of spelling or capitalization.
Builder
toString(): S
Type
toString(): Str
Wood
toString(): String
The serial number is still unique, and we left model as a String since there are thousands of different guitar models out there... way too many for an enum to be helpful.
The Guitar class uses these enumerated types to represent data, in a way that won't get screwed up by case issues or errors in spelling.
Here are our enumerated types.

Non abbiamo ancora finito col punto 1.

- Rick può avere più di una chitarra che risponde ai parametri di ricerca...



Here's the test program, updated to use the new version of Rick's search tool.

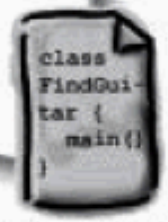
```
public class FindGuitarTester {

  public static void main(String[] args) {
    // Set up Rick's guitar inventory
    Inventory inventory = new Inventory();
    initializeInventory(inventory);

    Guitar whatErinLikes = new Guitar("", 0, Builder.FENDER,
                                      "Stratocastor", Type.ELECTRIC,
                                      Wood.ALDER, Wood.ALDER);
    List matchingGuitars = inventory.search(whatErinLikes);
    if (!matchingGuitars.isEmpty()) {
      System.out.println("Erin, you might like these guitars:");
      for (Iterator i = matchingGuitars.iterator(); i.hasNext(); ) {
        Guitar guitar = (Guitar)i.next();
        System.out.println("  We have a " +
          guitar.getBuilder() + " " + guitar.getModel() + " " +
          guitar.getType() + " guitar:\n   " +
          guitar.getBackWood() + " back and sides,\n   " +
          guitar.getTopWood() + " top.\n  You can have it for only $" +
          guitar.getPrice() + "!\n  ----");
      }
    } else {
      System.out.println("Sorry, Erin, we have nothing for you.");
    }
  }
```

We're using enumerated types in this test drive. No typing mistakes this time!

In this new version, we need to iterate over all the choices returned from the search tool.

This time we get a whole list of guitars that match the client's specs.

```
File Edit Window Help SweetSmell
%java FindGuitarTester
Erin, you might like these guitars:
  We have a Fender Stratocastor electric guitar:
     Alder back and sides,
     Alder top.
  You can have it for only $1499.95!
  ----
  We have a Fender Stratocastor electric guitar:
     Alder back and sides,
     Alder top.
  You can have it for only $1549.95!
  ----
```



FindGuitarTester.java

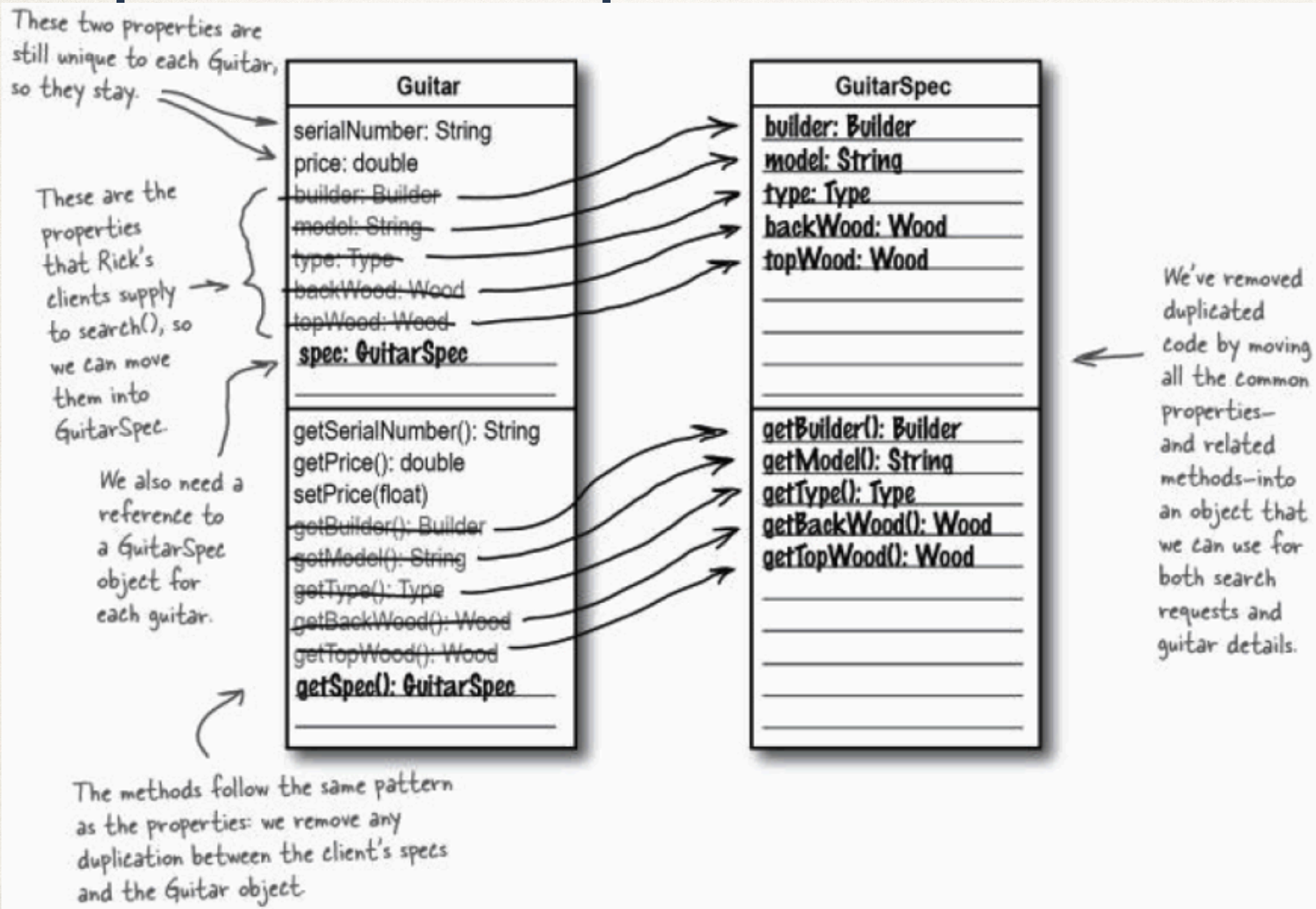
Finalmente passiamo al punto 2.: principi di OO

- ricontrolliamo come funziona il metodo search()
- il cliente vuole cercare una chitarra con alcune caratteristiche...
- ... perchè search() compara due oggetti di tipo Guitar? L'oggetto Guitar ha una serie di caratteristiche che non interessano la ricerca: ha senso "creare" comunque un oggetto Guitar per la ricerca?

- encapsulation: vi dice nulla?...

L'incapsulamento vi serve per raggruppare oggetti/attributi che sono logicamente collegati: permette di suddividere un oggetto in parti logiche

Separiamo le specs da Guitar



Now search() takes a GuitarSpec, instead of an entire Guitar object

```
public class Inventory {

  // variables, constructor, and other methods

  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;
  }
}
```

All of the information we use in comparing guitars is in GuitarSpec now, not the Guitar class.

This code is almost the same as it was before, except now we're using information in the GuitarSpec object

Even though we changed our classes a bit, this method still returns a list of guitars that match the client's specs

Inventory.java

```
public class FindGuitarTester {

  public static void main(String[] args) {
    // Set up Rick's guitar inventory
    Inventory inventory = new Inventory();
    initializeInventory(inventory);

    GuitarSpec whatErinLikes =
      new GuitarSpec(Builder.FENDER, "Stratocastor", Type.ELECTRIC,
                     Wood.ALDER, Wood.ALDER);
    List matchingGuitars = inventory.search(whatErinLikes);
    if (!matchingGuitars.isEmpty()) {
      System.out.println("Erin, you might like these guitars:");
      for (Iterator i = matchingGuitars.iterator(); i.hasNext(); ) {
        Guitar guitar = (Guitar)i.next();
        GuitarSpec spec = guitar.getSpec();
        System.out.println("  We have a " +
          spec.getBuilder() + " " + spec.getModel() + " " +
          spec.getType() + " guitar:\n     " +
          spec.getBackWood() + " back and sides,\n     " +
          spec.getTopWood() + " top.\n  You can have it for only $" +
          guitar.getPrice() + "!\n  ----");
      }
    } else {
      System.out.println("Sorry, Erin, we have nothing for you.");
    }
  }

  private static void initializeInventory(Inventory inventory) {
    // Add guitars to the inventory
  }
}
```

This time, the client sends a GuitarSpec to search().

We're using the new GuitarSpec class here as well.

The results aren't different this time, but the application is better designed, and much more flexible.

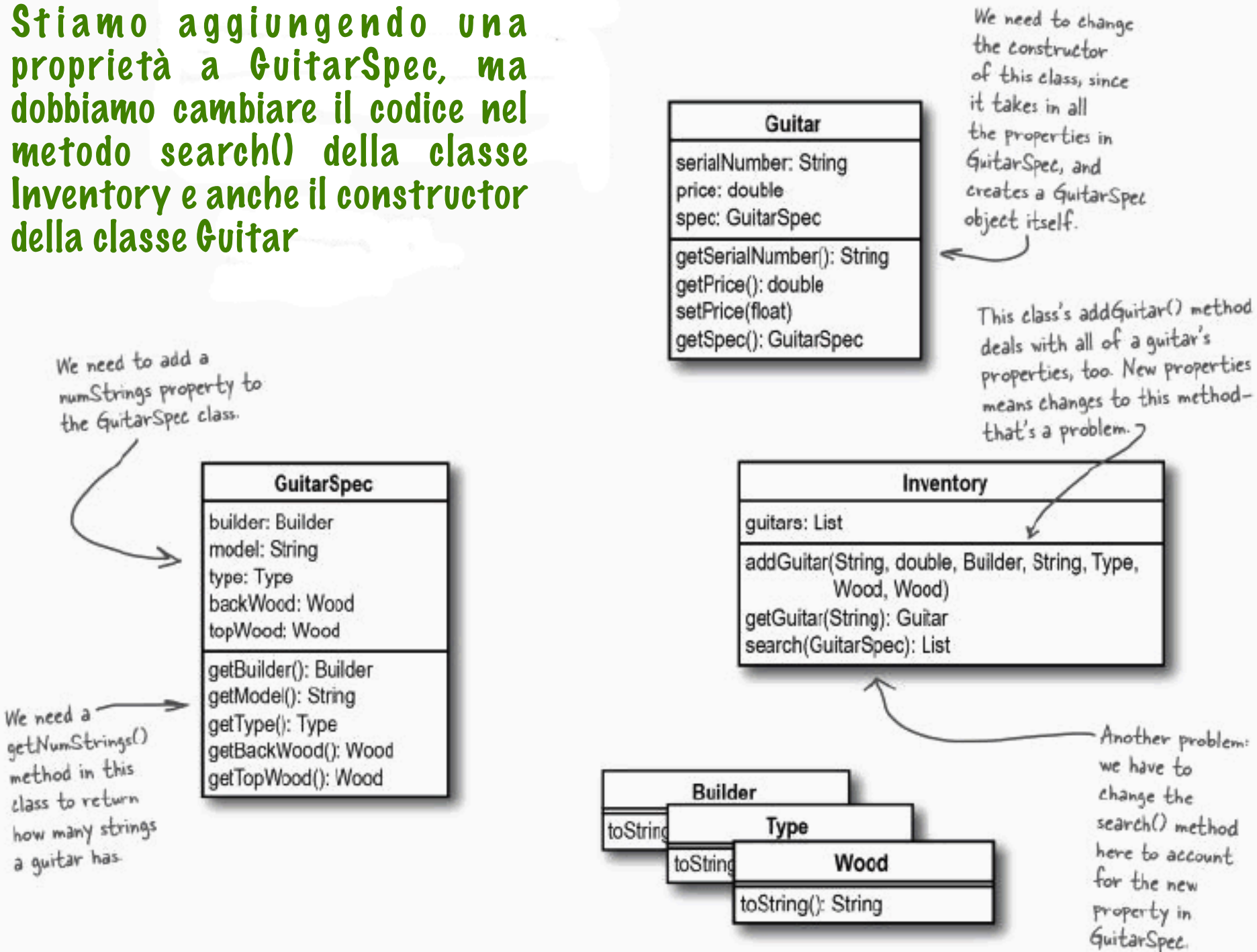
```
File Edit Window Help NotQuiteTheSame
%java FindGuitarTester
Erin, you might like these guitars:
  We have a Fender Stratocastor electric guitar:
     Alder back and sides,
     Alder top.
  You can have it for only $1499.95!
  ----
  We have a Fender Stratocastor electric guitar:
     Alder back and sides,
     Alder top.
  You can have it for only $1549.95!
  ----
```

Finalmente il punto 3.: mantenibilità e riusabilità

- Quanto è facile da mantenere il codice che abbiamo scritto? Cosa succede se dobbiamo fare una modifica?

- Esempio: Rick decide di vendere anche chitarre a 12 corde. Quanto è facile modificare il programma?

Stiamo aggiungendo una proprietà a GuitarSpec, ma dobbiamo cambiare il codice nel metodo search() della classe Inventory e anche il constructor della classe Guitar



Il codice non è debolmente accoppiato

- un cambiamento a GuitarSpec non dovrebbe coinvolgere Inventory e Guitar
- abbiamo usato l'incapsulamento creando GuitarSpec, ma non abbiamo isolato per bene gli oggetti

Come procediamo?

1. aggiungiamo numStrings come proprietà di GuitarSpec (e i metodi che servono)
2. modificare Guitar in modo che le proprietà di GuitarSpec siano incapsulate fuori dal constructor della classe
3. cambiare Inventory.search() per delegare la comparazione invece che gestirla direttamente
4. aggiornare il programma di test e verificare che tutto funzioni ancora

Come procediamo?

1. aggiungiamo numStrings co
   che servono)
2. modificare Guitar in modo che le proprietà di GuitarSpec siano incapsulate fuori dal constructor della classe
3. cambiare Inventory.search() per delegare la comparazione invece che gestirla direttamente
4. aggiornare il programma di test e verificare che tutto funzioni ancora

Delegation: when an object needs to perform a certain task, and instead of doing that task directly , it ask another object to handle the task (or sometimes just a part of

Come procediamo?

1. aggiungiamo numStrings co
   che servono)
2. modificare Guitar in modo che le proprietà di GuitarSpec siano incapsulate fuori dal constructor della classe
3. cambiare Inventory.search() per delegare la comparazione invece che gestirla direttamente
4. a

Delegation: when an object needs to perform a certain task, and instead of doing that task directly , it ask another object to handle the task (or sometimes just a part of

Delegare aiuta a mantenere il codice riusabile. Lascia anche che ogni oggetto sia responsabile delle proprie funzionalità, piuttosto che sparpagliare il codice relativo a un oggetto per tutta l'applicazione

Come procediamo?

Delegando, la classe si prende cura dell'uguaglianza tra istanze. Quindi è più indipendente dal resto (loosely coupled). Si aumenta il numero di oggetti, ma ogni oggetto ha un singolo, ben definito compito.

Delegation: when an object needs to perform a certain task, and instead of doing that task directly , it ask another object to handle the task (or sometimes just a part of

s co

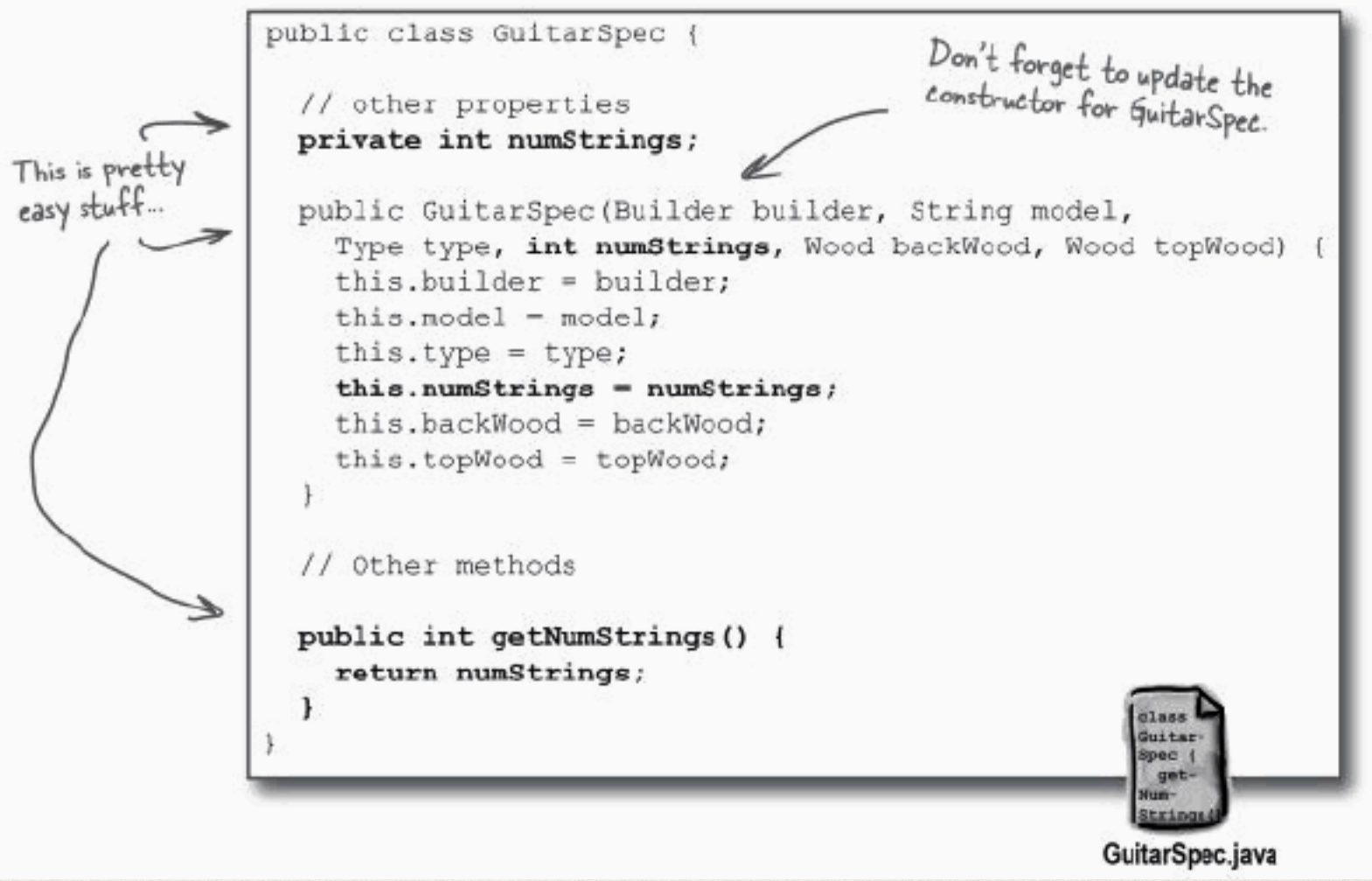
do che le proprietà di GuitarSpec siano
incapsulate fuori dal costruttore della classe

3. cambiare Inventory.search() per delegare la comparazione invece che gestirla direttamente

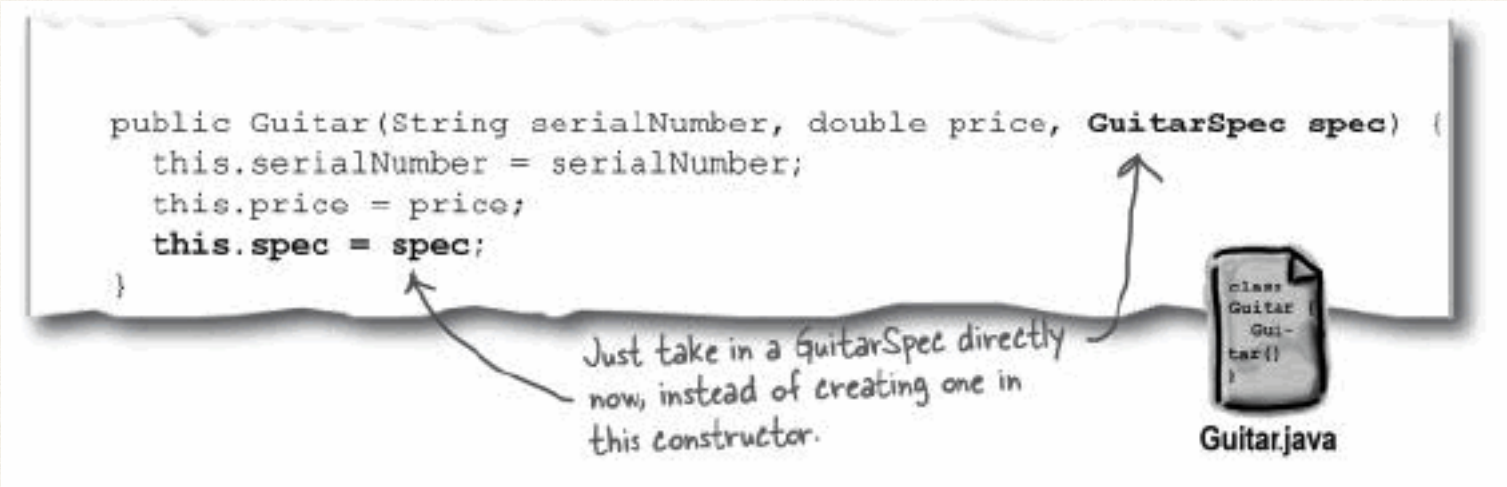
4. a

Delegare aiuta a mantenere il codice riusabile. Lascia anche che ogni oggetto sia responsabile delle proprie funzionalità, piuttosto che sparpagliare il codice relativo a un oggetto per tutta l'applicazione

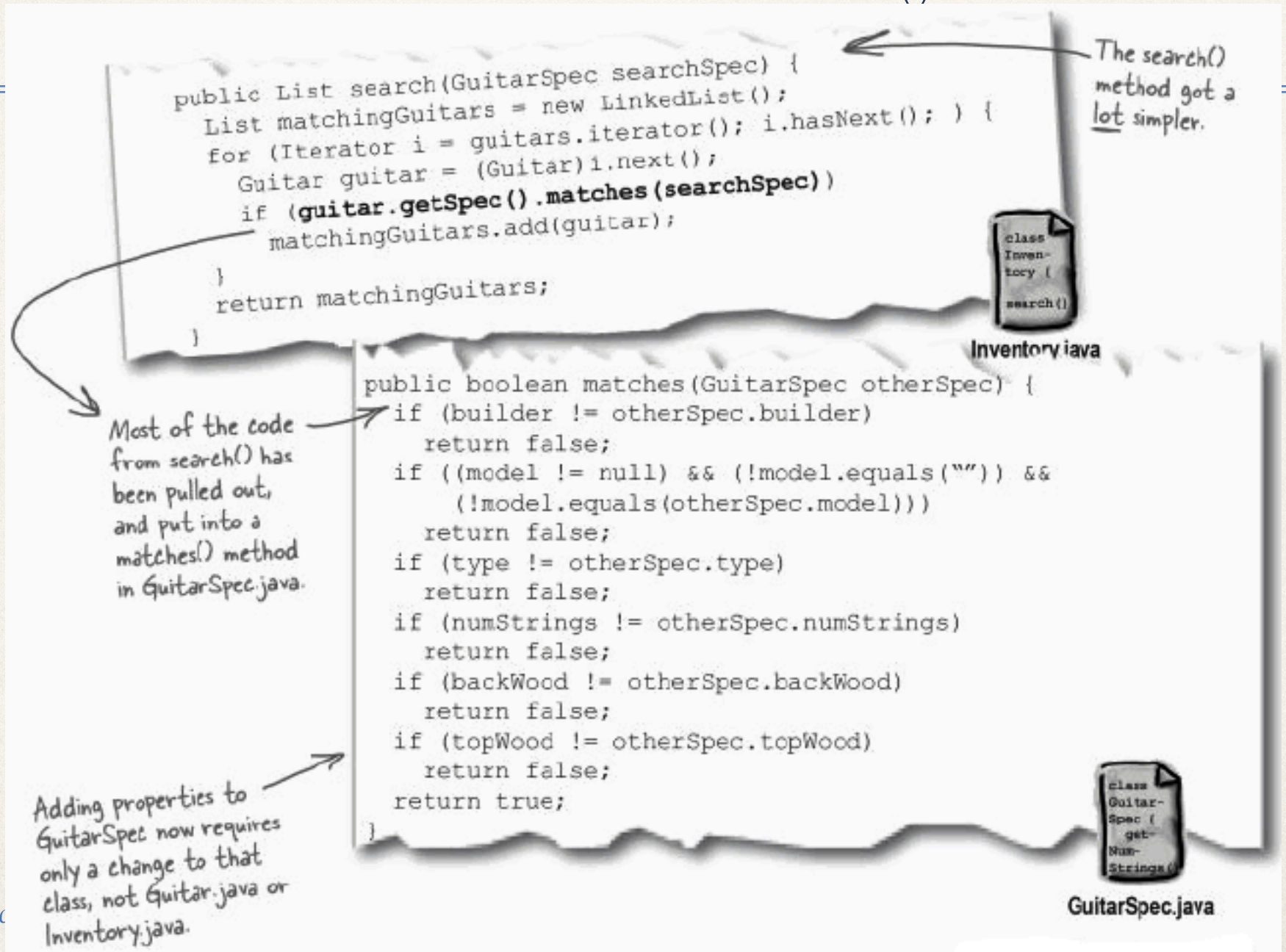
1. Aggiorniamo GuitarSpec



2. Modifichiamo Guitar



3. Modifichiamo search()

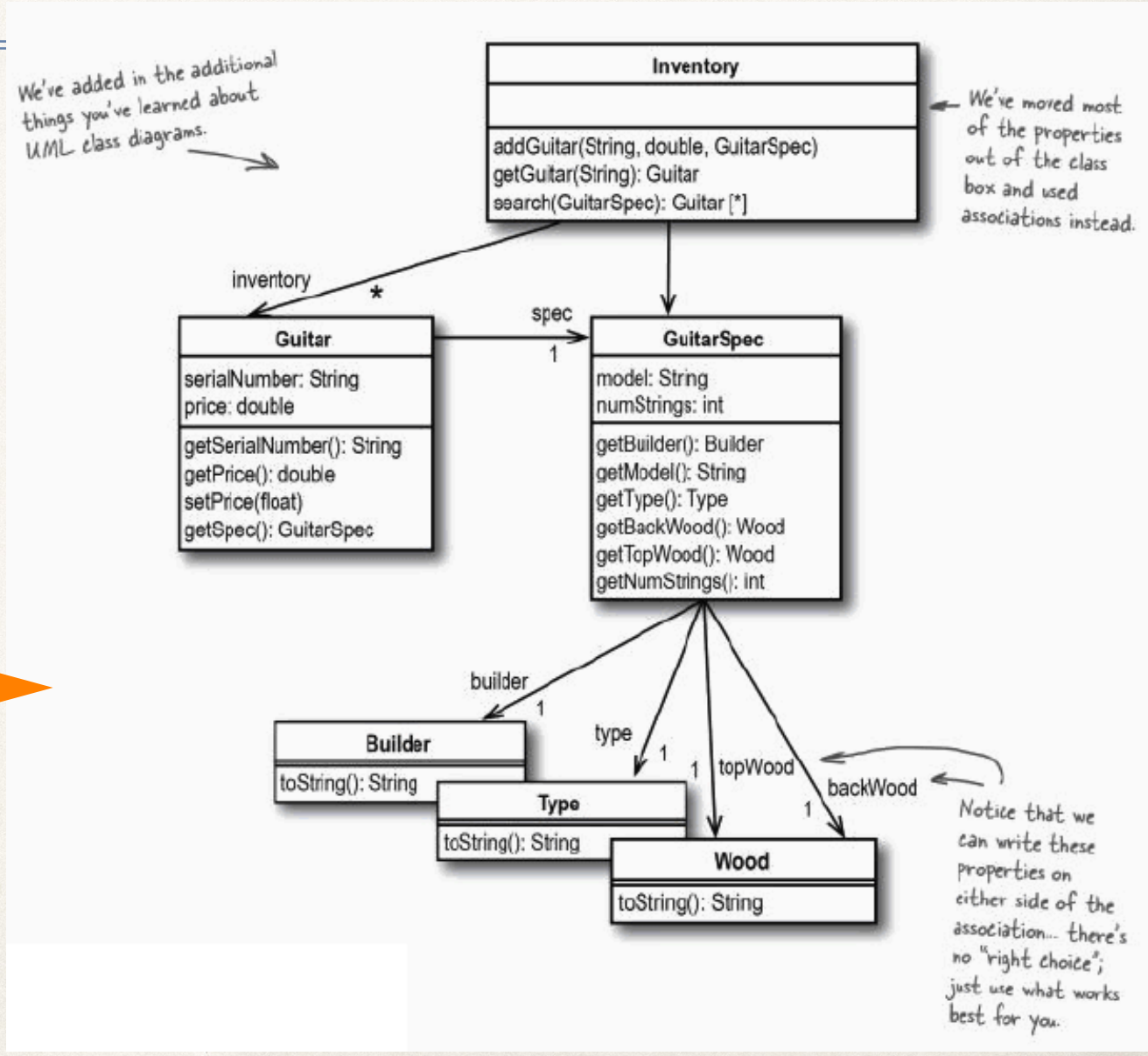


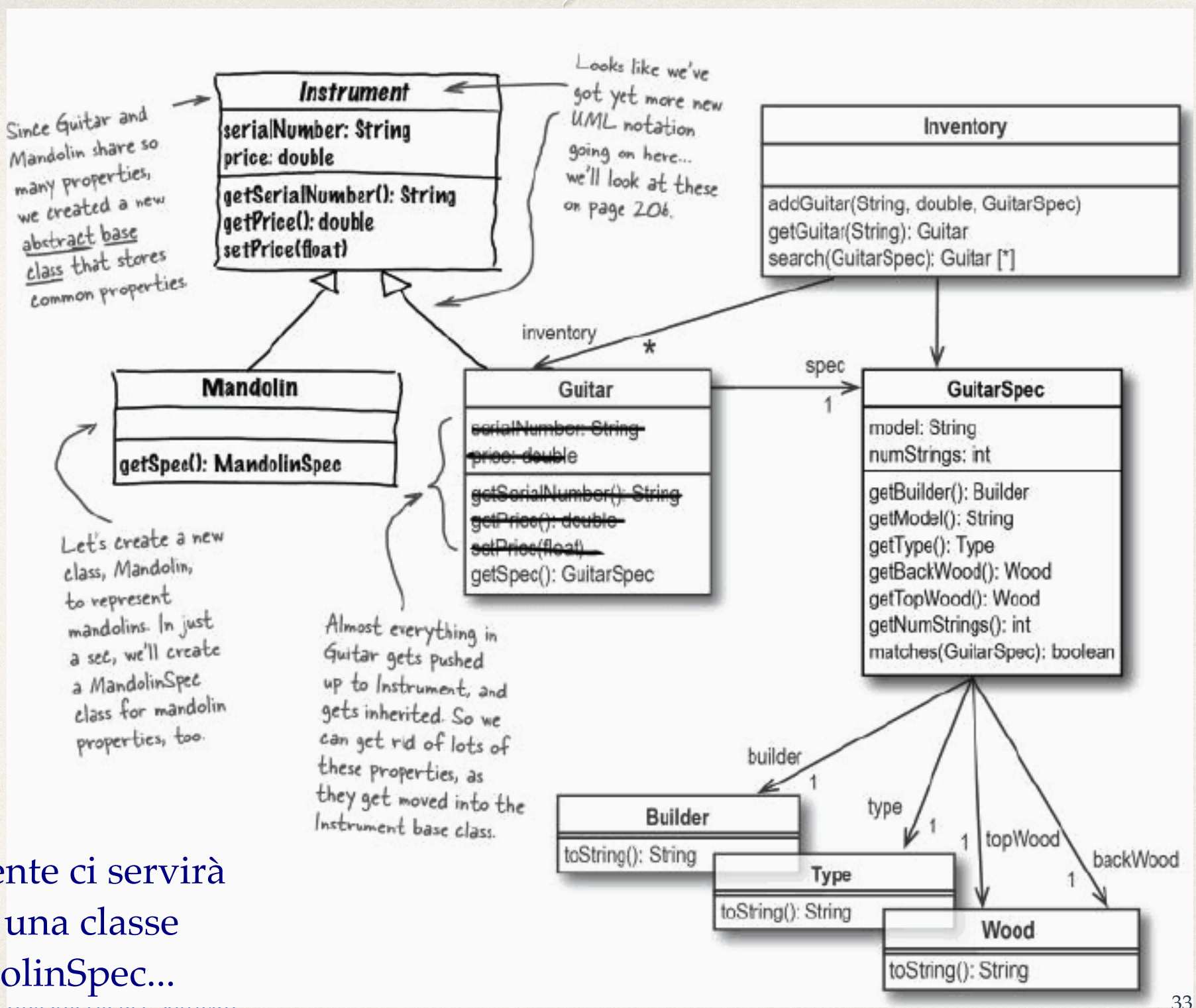
Fine della fase 1.....

Arrivano i ~~barbari~~ altre modifiche

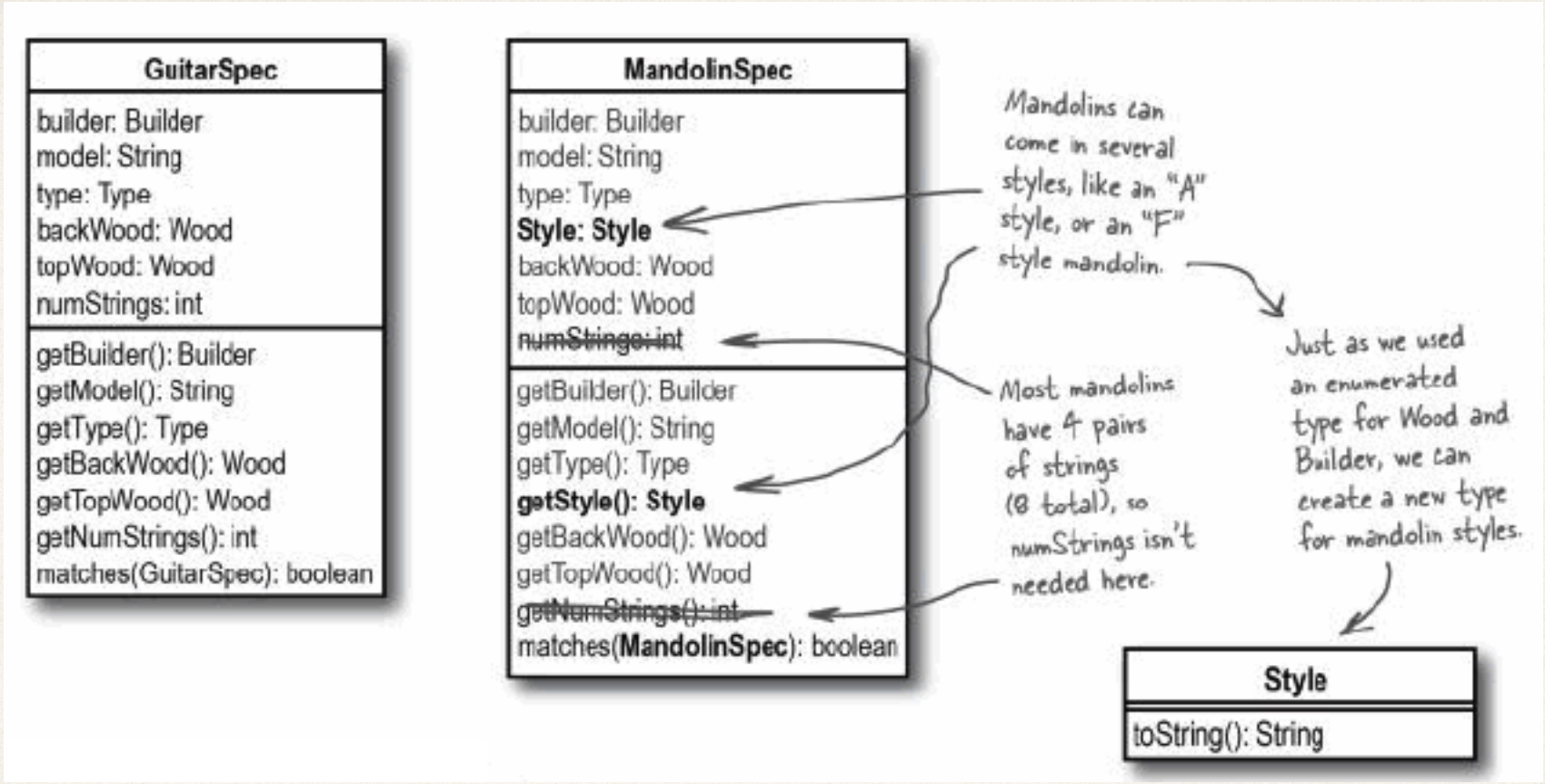
- Rick vuole vendere anche mandolini
- ...dopotutto sono molto simili alle chitarre, no?

Questo è il nostro punto di partenza



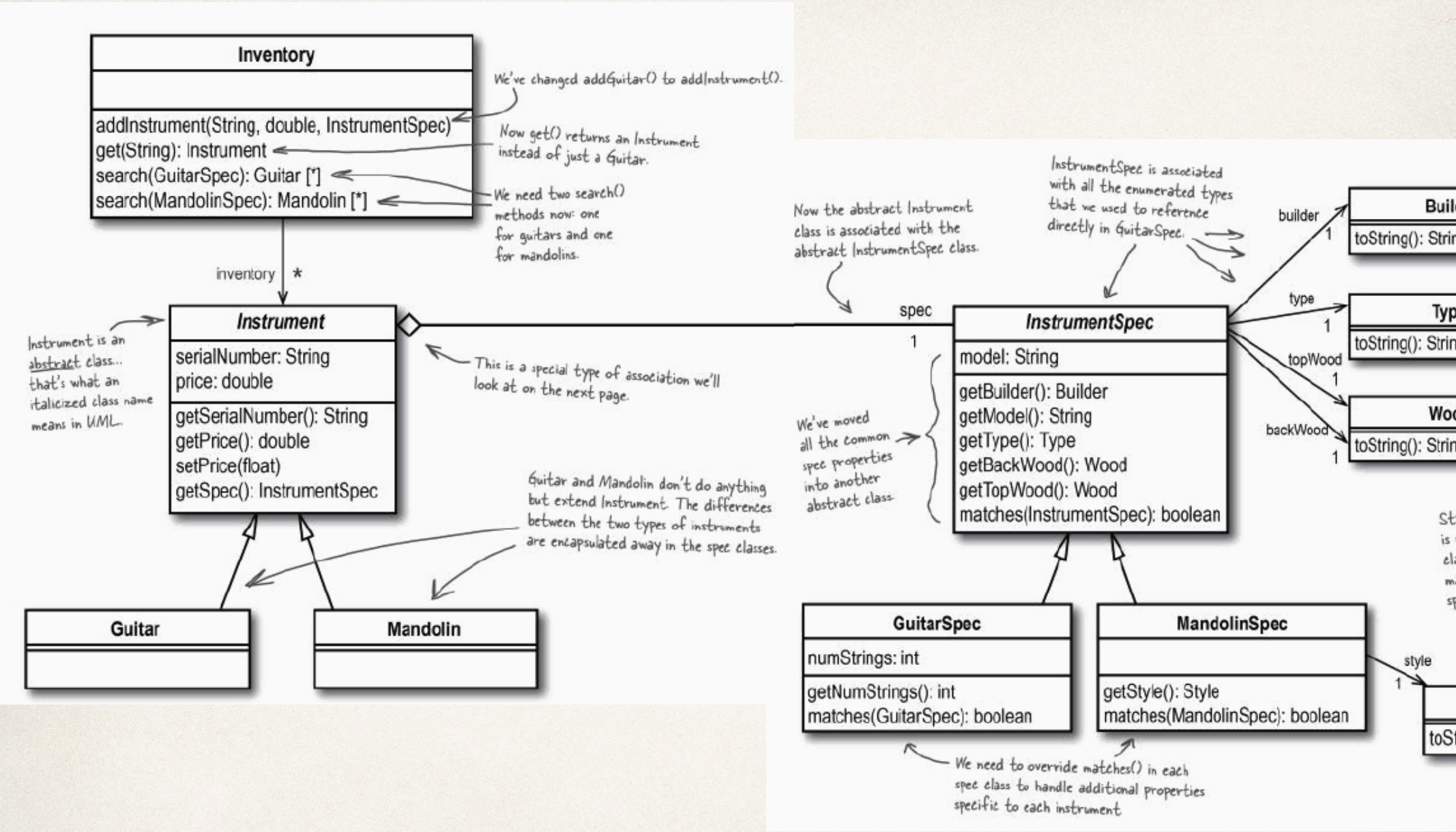


Ovviamente ci servirà anche una classe MandolinSpec...



Ogni volta che trovate un comportamento comune in due o più classi, controllate se potete astrarre il comportamento in una classe, e poi riusarlo dove vi serve, ovvero.....

Perchè due classi così separate?


Inventory
addInstrument(String, double, InstrumentSpec)
get(String): Instrument
search(GuitarSpec): Guitar [*]
search(MandolinSpec): Mandolin [*]
We've changed addGuitar() to addInstrument().
Now get() returns an Instrument instead of just a Guitar.
We need two search() methods now: one for guitars and one for mandolins.
inventory
*
Instrument
serialNumber: String
price: double
getSerialNumber(): String
getPrice(): double
setPrice(float)
getSpec(): InstrumentSpec
Instrument is an abstract class... that's what an italicized class name means in UML.
This is a special type of association we'll look at on the next page.
Now the abstract Instrument class is associated with the abstract InstrumentSpec class.
spec
1
InstrumentSpec is associated with all the enumerated types that we used to reference directly in GuitarSpec.
InstrumentSpec
model: String
getBuilder(): Builder
getModel(): String
getType(): Type
getBackWood(): Wood
getTopWood(): Wood
matches(InstrumentSpec): boolean
We've moved all the common spec properties into another abstract class.
builder
1
type
1
topWood
1
backWood
1
toString(): Strin
Guitar and Mandolin don't do anything but extend Instrument. The differences between the two types of instruments are encapsulated away in the spec classes.
Guitar
Mandolin
GuitarSpec
numStrings: int
getNumStrings(): int
matches(GuitarSpec): boolean
MandolinSpec
getStyle(): Style
matches(MandolinSpec): boolean
style
1
We need to override matches() in each spec class to handle additional properties specific to each instrument

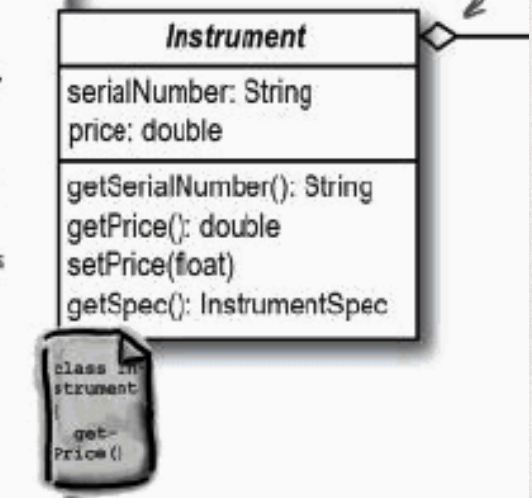
Codice?

```
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.

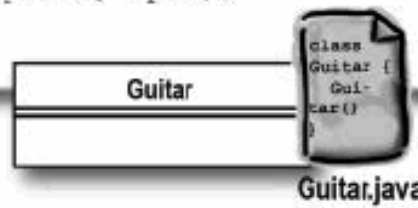


Instrument.java

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

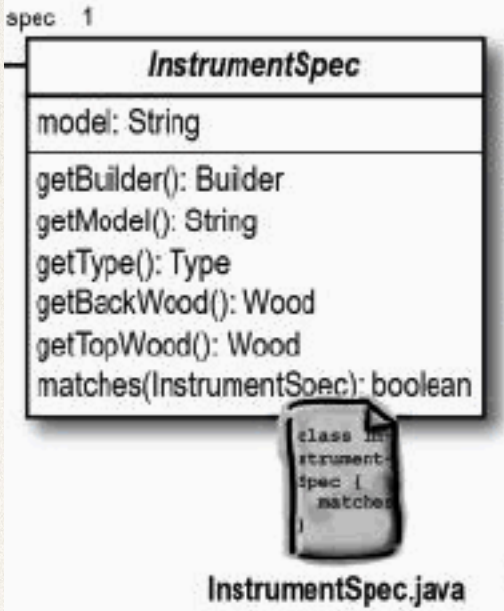
```
public class Guitar extends Instrument {

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

```
public class Mandolin extends Instrument {

Number, double price,
ec) {
spec);
```



Guitar.java

Mandolin.java




```
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.

This version of matches() does just what you'd expect: compares all properties in this class to another spec instance. We'll have to override this in subclasses, though...

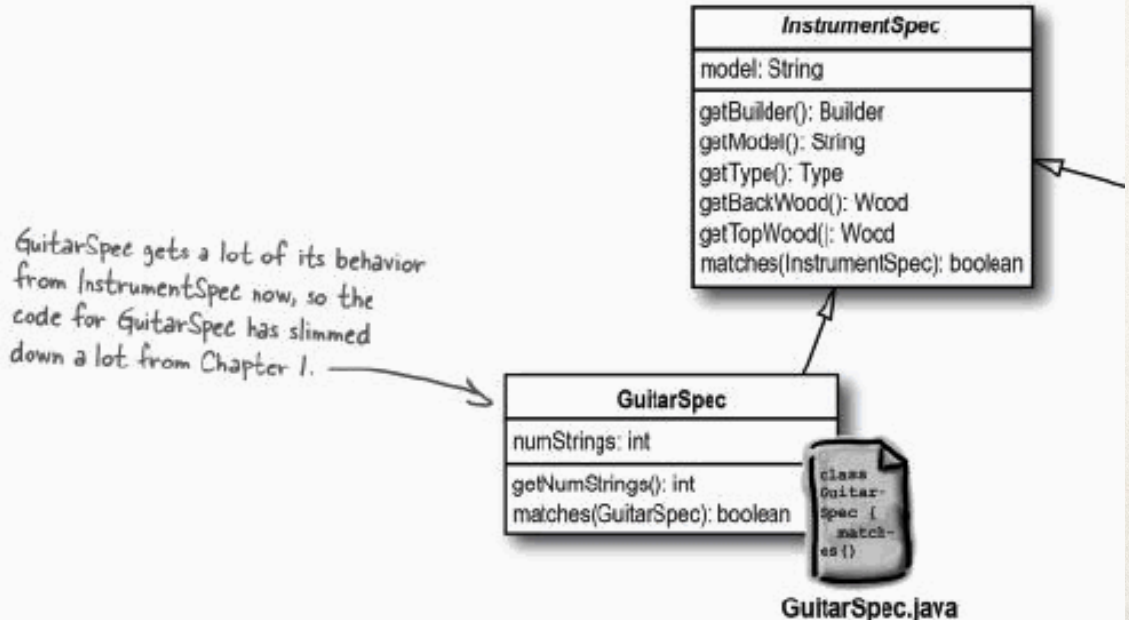
Just as Guitar extended Instrument, GuitarSpec extends InstrumentSpec.

```
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;
  }
}
```

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.

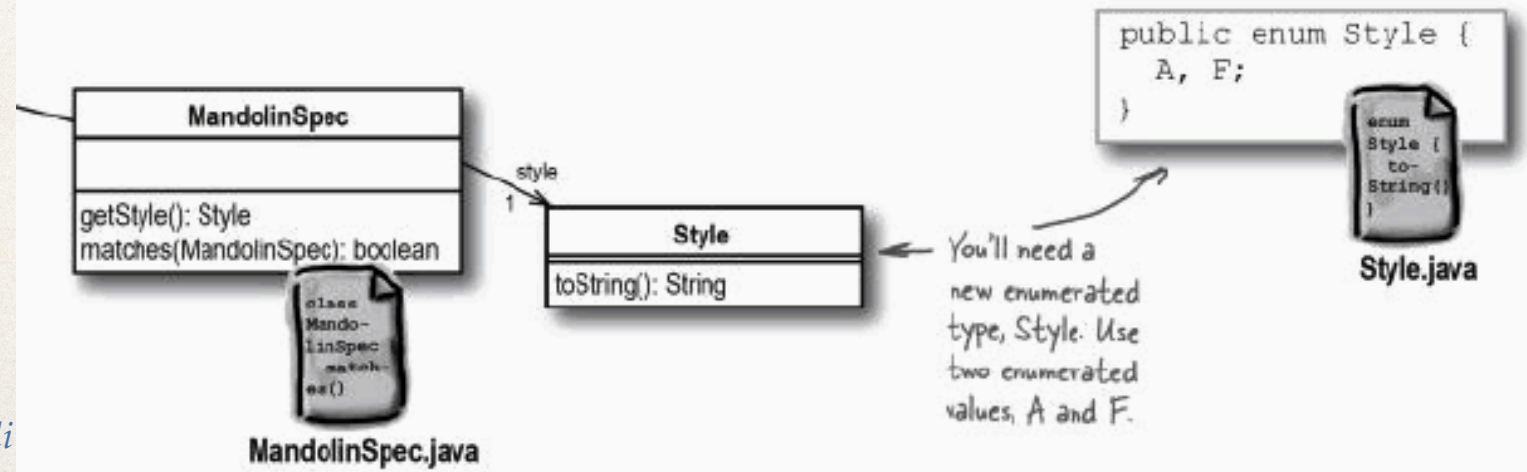
matches() uses the superclass's matches(), and then performs additional checks to make sure the spec is the right type, and matches the guitar-specific properties.



```
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 Style 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.




```
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.

By using the Instrument and InstrumentSpec classes, we can turn addGuitar() into a more generic method, and create any kind of instrument.

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.

Altri ~~barbari~~ modifiche...

- Il nostro venditore di chitarre ormai è lanciato: vuole trattare anche banjo, dobro, bassi, magari anche violini (!)

- Uno dei modi migliori per scoprire se il software è disegnato bene consiste nel provare a cambiarlo: se è difficile, probabilmente c'è spazio per miglioramenti.
- Nel nostro caso, per aggiungere un singolo strumento:
  - ✓ oggetto derivato a `Instrument`
  - ✓ specifiche derivate da `InstrumentSpec`
  - ✓ intervenire in `Inventory`, sia in `search` che in `addInstrument`

Rivedere le decisioni

- `InstrumentSpec` è una interfaccia (e c'era un perchè), ma ora la situazione è cambiata: e se non fosse una interfaccia?

- ... posso usare la classe in `search`, e non sono costretto a scrivermi N versioni del metodo

```
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);
  }
```

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

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

```
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(
  }
```

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

  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;
  }
}
```

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.

- Abbiamo sistemato `search`, ma c'è ancora `addInstruments` da sistemare
- `Instruments` è abstract perchè ogni strumento è rappresentato dalla sua classe derivata
- **ma le classi sono legate al comportamento!**
  - io creo sottoclassi perchè il comportamento della sottoclasse è diverso
- **...in questo programma, una chitarra si comporta diversamente da un banjo?**

- Be', una chitarra e un qualsiasi strumento non hanno un comportamento diverso, ma hanno attributi diversi; è giusto fare sottoclassi per questo motivo?

- Ci sono due possibili ragioni per avere sottoclassi:

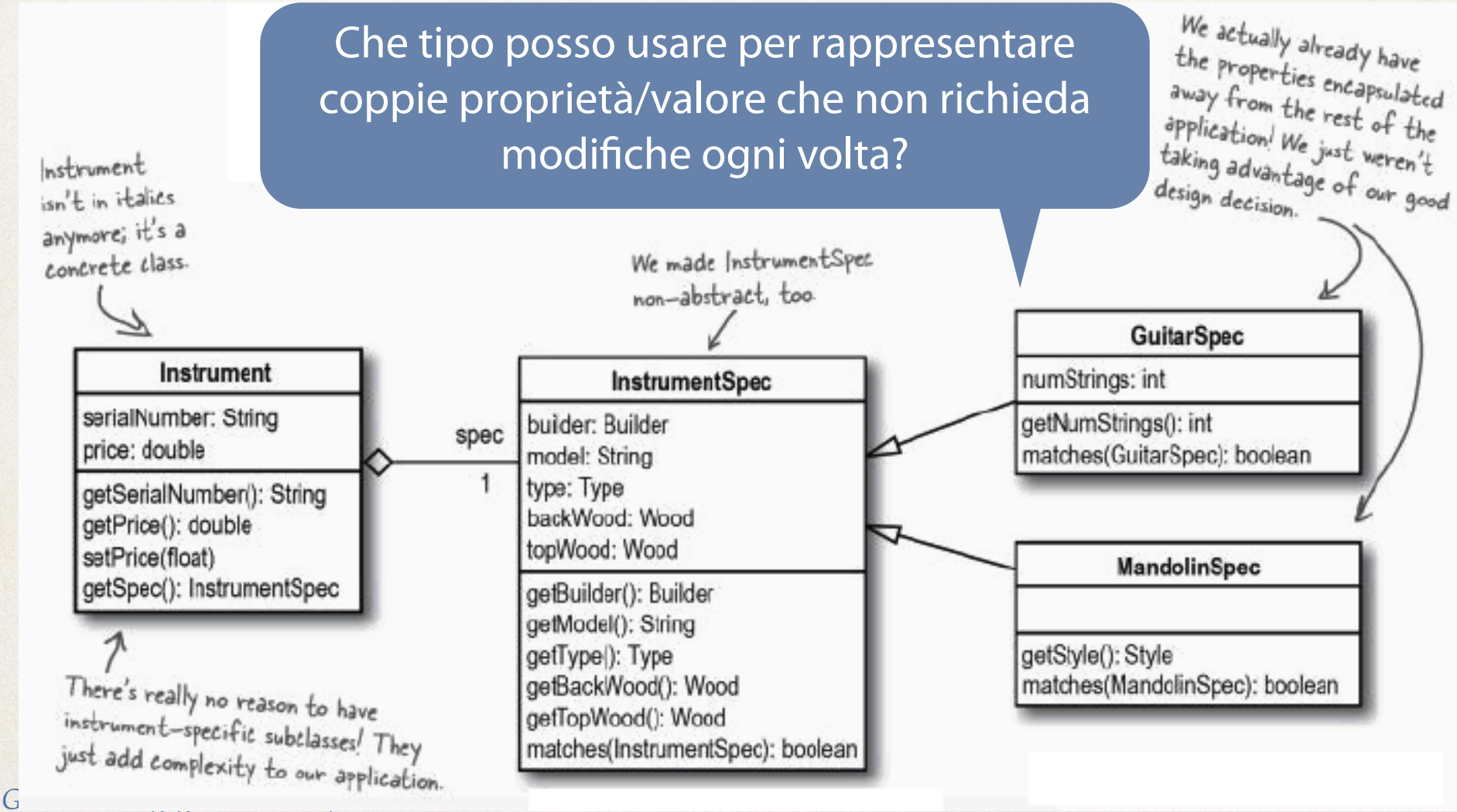
  1. `Instrument` rappresenta un concetto, e non un oggetto reale; per questo è abstract
  2. ogni strumento ha proprietà sue, per questo uso sottoclassi diverse

- Le ragioni sembrano buone, ma poi il software non è flessibile...

- Noi abbiamo usato ereditarietà, polimorfismo, astrazione...

  e se la soluzione invece fosse nell'incapsulare?

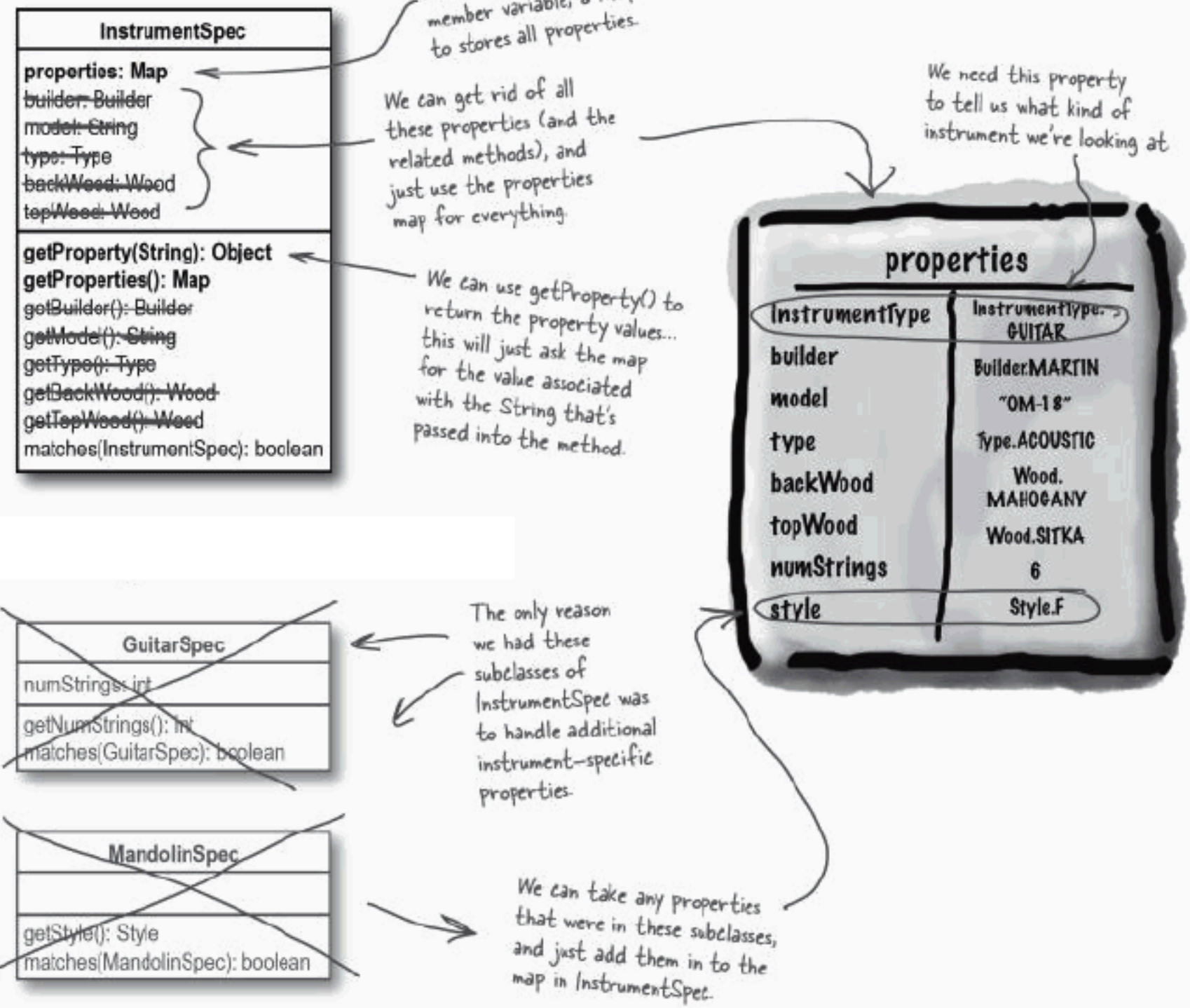
- ... separare ciò che cambia da ciò che non cambia...

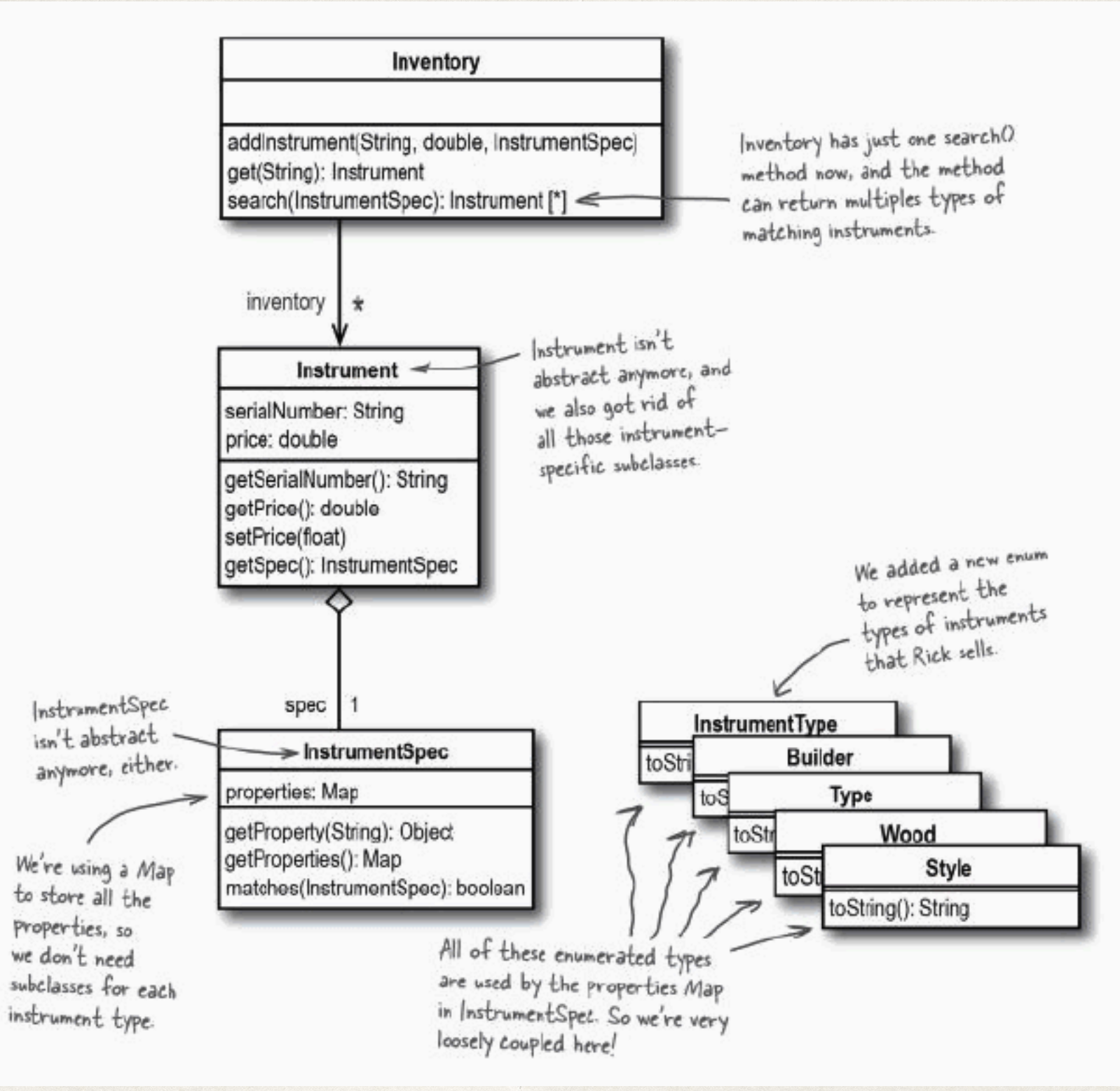


- Noi abbiamo usato ereditarietà, polimorfismo, astrazione...

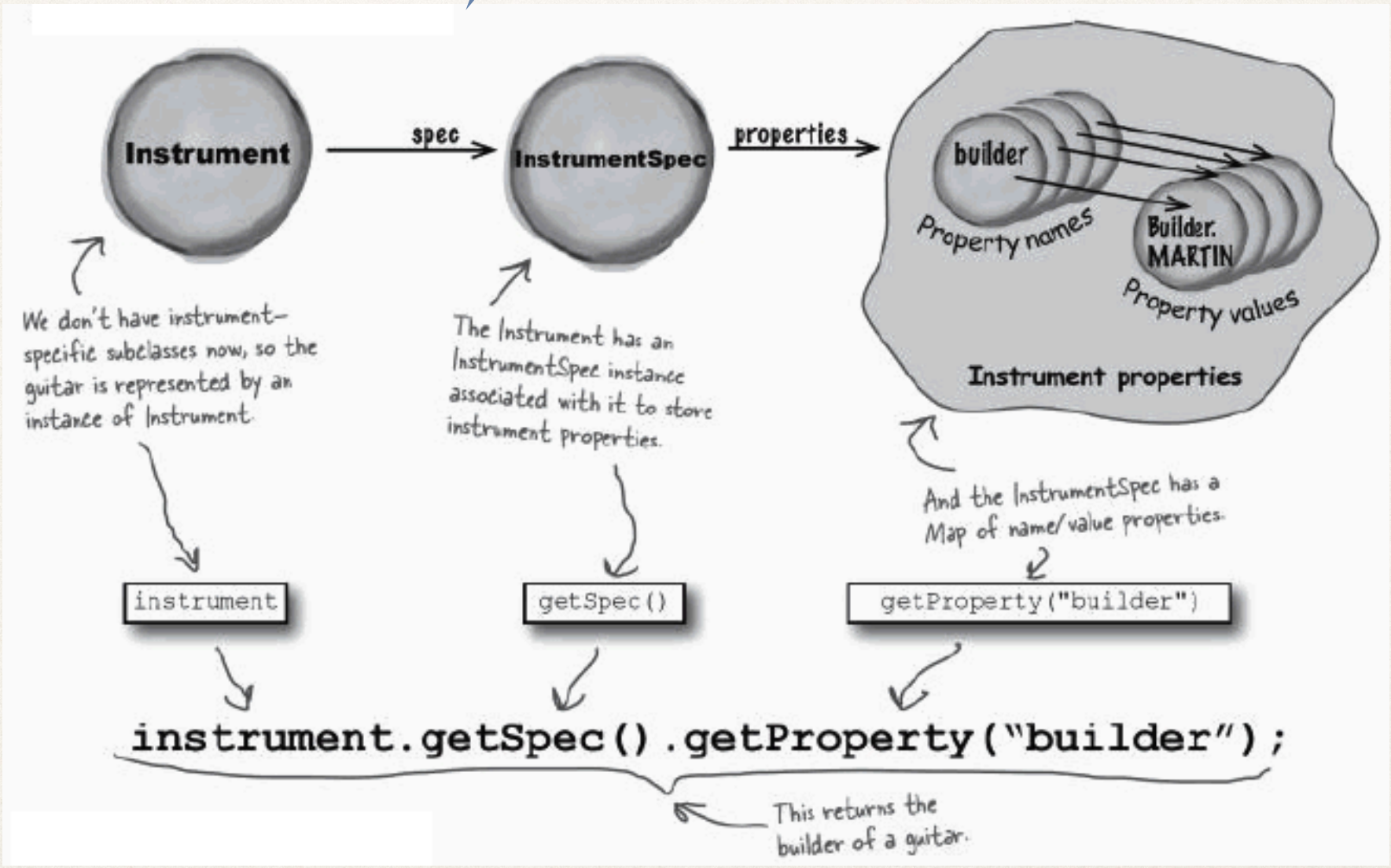
  e se la soluzione invece fosse nell'incapsulare?

- ... separare ciò che cambia da ciò che non cambia...


InstrumentSpec
properties: Map
builder: Builder
model: String
type: Type
backWood: Wood
topWood: Wood
getProperty(String): Object
getProperties(): Map
getBuilder(): Builder
getModel(): String
getType(): Type
getBackWood(): Wood
getTopWood(): Wood
matches(InstrumentSpec): boolean
Now we just have one member variable, a Map to stores all properties.
We can get rid of all these properties (and the related methods), and just use the properties map for everything.
We can use getProperty() to return the property values... this will just ask the map for the value associated with the String that's passed into the method.
We need this property to tell us what kind of instrument we're looking at
properties
InstrumentType
InstrumentType.GUITAR
builder
Builder.MARTIN
model
"OM-18"
type
Type.ACOUSTIC
backWood
Wood.MAHOGANY
topWood
Wood.SITKA
numStrings
6
style
Style.F
GuitarSpec
numStrings: int
getNumStrings(): int
matches(GuitarSpec): boolean
The only reason we had these subclasses of InstrumentSpec was to handle additional instrument-specific properties.
MandolinSpec
getStyle(): Style
matches(MandolinSpec): boolean
We can take any properties that were in these subclasses, and just add them in to the map in InstrumentSpec.


Inventory
addInstrument(String, double, InstrumentSpec)
get(String): Instrument
search(InstrumentSpec): Instrument [*]
Inventory has just one search() method now, and the method can return multiples types of matching instruments.
inventory
*
Instrument
serialNumber: String
price: double
getSerialNumber(): String
getPrice(): double
setPrice(float)
getSpec(): InstrumentSpec
Instrument isn't abstract anymore, and we also got rid of all those instrument-specific subclasses.
We added a new enum to represent the types of instruments that Rick sells.
spec
1
InstrumentSpec isn't abstract anymore, either.
InstrumentSpec
properties: Map
getProperty(String): Object
getProperties(): Map
matches(InstrumentSpec): boolean
We're using a Map to store all the properties, so we don't need subclasses for each instrument type.
InstrumentType
toStri
Builder
toS
Type
toStr
Wood
toSt
Style
toString(): String
All of these enumerated types are used by the properties Map in InstrumentSpec. So we're very loosely coupled here!

Se voglio sapere la marca di uno strumento, ora procedo in questo modo

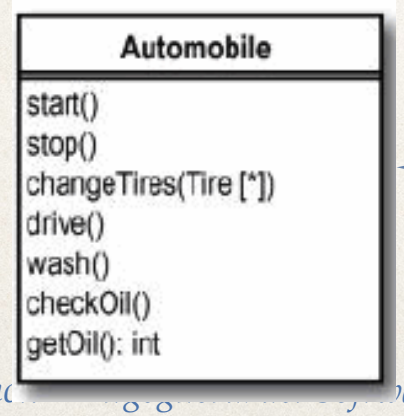


Alcune considerazioni

- **INCAPSULAMENTO**: perchè?
  - ➡ prevenire duplicazione di codice
  - ➡ isolare ciò che varia
- **CAMBIAMENTI**: come gestirli?
  - fanno parte del gioco: codice fatto male cade a pezzi al primo cambiamento, codice fatto bene è facile da cambiare
  - per essere resilienti ai cambiamenti: essere sicuri che ogni classe abbia una sola ragione per cambiare

Alcune considerazioni

- **INCAPSULAMENTO**: perchè?
  - ➡ prevenire duplicazione di codice
  - ➡ isolare ciò che varia
- **CAMBIAMENTI**: come gestirli?
  - fanno parte del gioco: codice fatto male cade a pezzi al primo cambiamento, codice fatto bene è facile da cambiare
  - per essere resilienti ai cambiamenti: essere sicuri che ogni classe abbia una sola ragione per cambiare

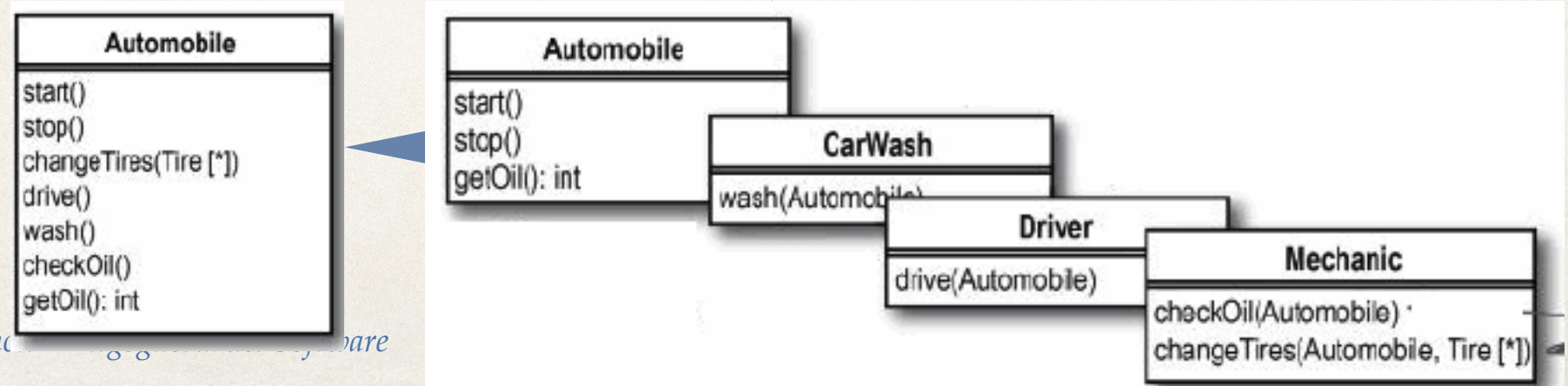


Questa classe fa troppe cose, può cambiare per troppe ragioni

Alcune considerazioni

- **INCAPSULAMENTO**: perchè?
  - ➡ prevenire duplicazione di codice
  - ➡ isolare ciò che varia
- **CAMBIAMENTI**: come gestir
  - fanno parte del gioco: codic cambiamento, codice fatto ben è facile da cambiare
  - per essere resilienti ai cambiam enti: essere sicuri che ogni classe abbia una sola ragione per cambiare

Ora abbiamo più classi, ma ogni classe fa una cosa sola e quindi ha una sola ragione per cambiare



- **Cambiare il design**
  - c'è voluto un po' per capire che non ha senso avere sottoclassi diverse per ogni strumento: perchè?
    - sembrava avesse senso, ed è difficile rivedere qualcosa che si pensa sia già a posto
  - **code once, look twice or more**: se cambiando finisci nei guai, controlla attentamente il tuo design. Una decisione precedente può essere la causa dei tuoi problemi attuali