Great Software Begins Here

a cura di Paolo Foletto
paolo.foletto@jugpadova.it

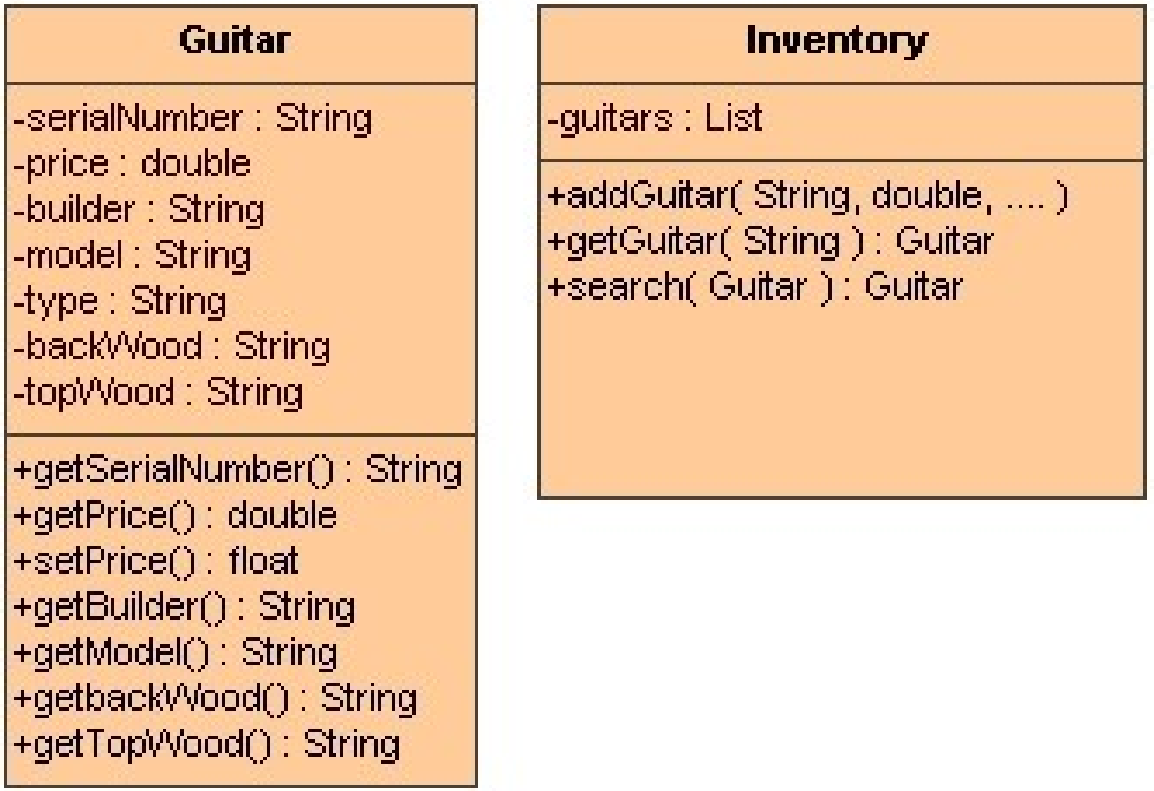
Java User Group Padova

Fantastic Guitar (Work)shop

Toni, detto “mano lenta”, appassionato di chitarre, e' il proprietario di un negozio di chitarre di fascia alta

Pochi mesi fa, Toni ha deciso di buttare via il sistema di gestione del negozio basato su carta e incominciare a usare un sistema basato su computer per la gestione del magazzino delle chitarre.
Ha fatto un contratto con una nota azienda informatica “Programma Sporco e Svelto”.
In effetti al primo contatto gli era venuto il dubbio che si trattasse di una ditta di detersivi ma date le loro referenze ha iniziato.
Stanno costruendo uno strumento di ricerca che permetta ai clienti di trovare lo strumento dei loro sogni

Fantastic Guitar (Work)Shop



Arriva in negozio un certo Ricky Portera che vuole acquistare una fender

Proviamo a eseguire la ricerca

Sembra che non ci sia nessuna fender

Ma Toni è sicuro di avere almeno una fender ...

Che cosa cambiereste per PRIMA?

How do you write great software every time?

The customer – friendly programmer says

Great software always does what the customer wants it to. So even if customers think ways to use the software, it does'nt break or give them unexpected results

The object oriented programmer says

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

The design guru programmer says

Great software is when you use tried-and-true design patterns and principles. You've kept your objects loosely coupled , and you code open for extension but close for modification. That also make the code more reusable, so you don't have to rework everything to use parts of your application over and over again

Queste tre risposte sembrano molto diverse tra di loro.

Quale di queste sarà quella giusta?

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

2. Apply basic OO principles to add flexibility

3. Strive for a a maintainable, reusable design

Proviamo a confrontare le stringhe ....

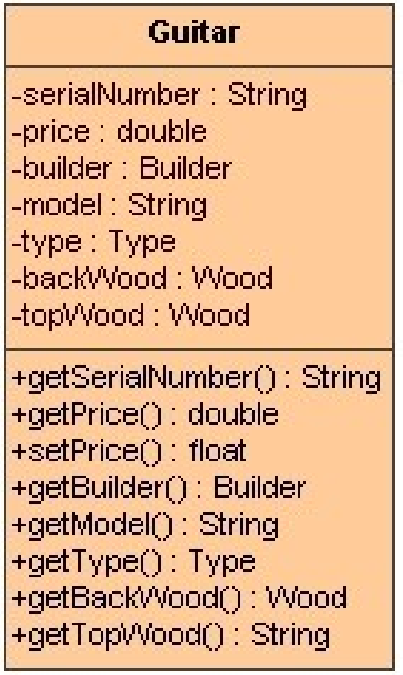
Andiamo a modificare il codice in modo che Ricky Portera possa trovare la fender

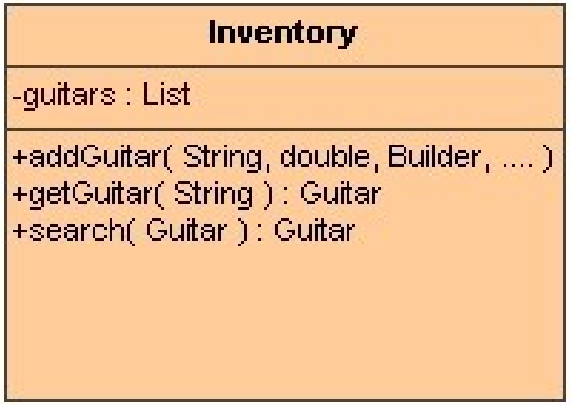
FantasticGuitarWokshop01 inventory02

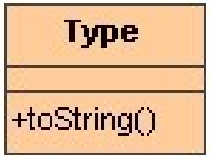
Proviamo a eliminare i confronti tra stringhe ...

Introducendo dei tipi enumerati

- Sono stati introdotti dei tipi enumerati
- Alcuni attributi sono cambiati da String a un tipo
- I valori di ritorno sono stati tipizzati







Don't create problems to solve problems

Nella precedente versione il tool di ricerca trovava la prima chitarra che soddisfava i criteri di ricerca

I REQUISITI CAMBIANO!

Toni arriva con un nuovo requisito per la sua applicazione: vuole che lo strumento di ricerca restituisca TUTTE le chitarre che corrispondono ai criteri di ricerca dei clienti e non solo la prima nel suo magazzino

Andiamo a modificare il codice in modo che Ricky Portera possa trovare tulle le fender che ci sono

Si tratta di un esercizio, dovete mettere a posto i pezzi di codice al posto giusto

```
public  ______ search(GuitarSpec searchSpec) {
  ____ _________________ = new ___________();
  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;
    _______________.____(_____);
  }
  return ___________________;
}
```

List LinkedList matchingGuitars add

Ecco la soluzione

```
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;
}
```

A questo punto Ricky Portera e' riuscito a trovare tutte le fender disponibili

I clienti di Tony hanno ricominciato a comprare chitarre

Si, è proprio quello che volevo commenta Toni “Mano lenta”

E noi abbiamo risolto il primo punto

- Il cliente fornisce le preferenze
- Il metodo search cerca attraverso l'inventory di Toni
- Ciascuna chitarra è confrontata con le preferenze del cliente
- Al cliente di Toni viene restituita un elenco di chitarre che soddisfano le sue preferenze

- The Mistery of Mismatched Object Type

1. Gli oggetti devono fare ciò che il loro nome indica

2. Ciascun oggetto dovrebbe rappresentare un solo concetto

3. Le proprietà inutilizzate sono un segno indicativo

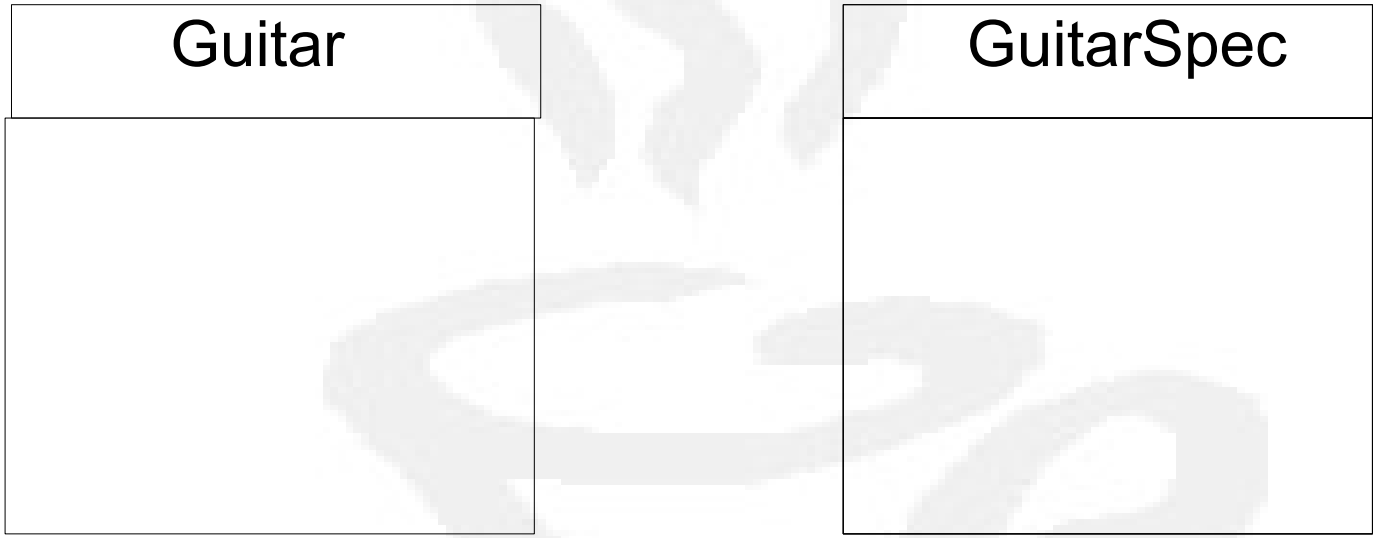
Incapsulamento

ti permette di raggruppare le tue applicazioni in parti logiche

Introduciamo la classe GuitarSpec

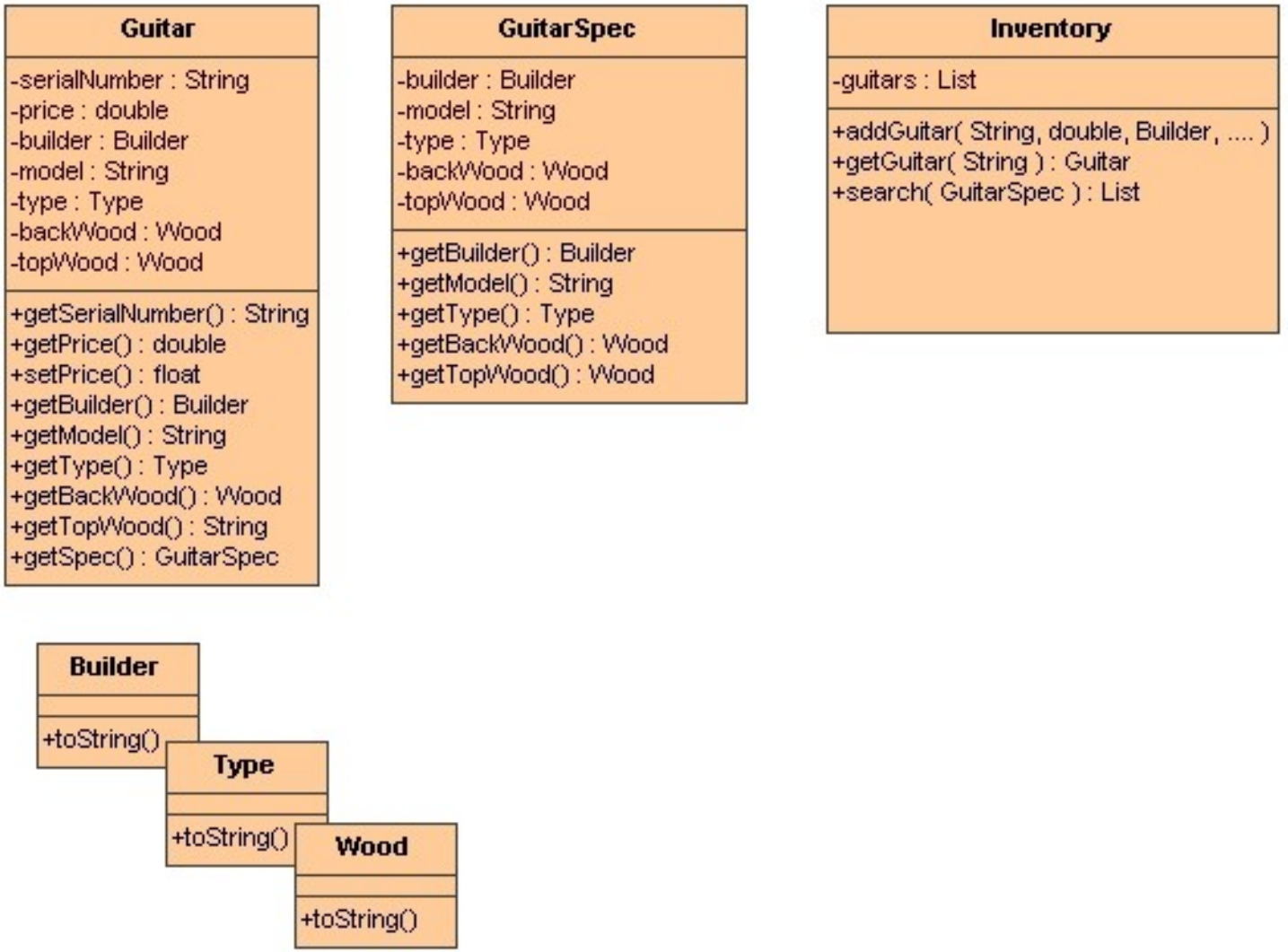
Distribuiamo le responsabilità tra Guitar e GuitarSpec

Questo è un esercizio, dovete provare con carta e matita a spostare, aggiungere, togliere metodi e proprietà da Guitar a GuitarSpec

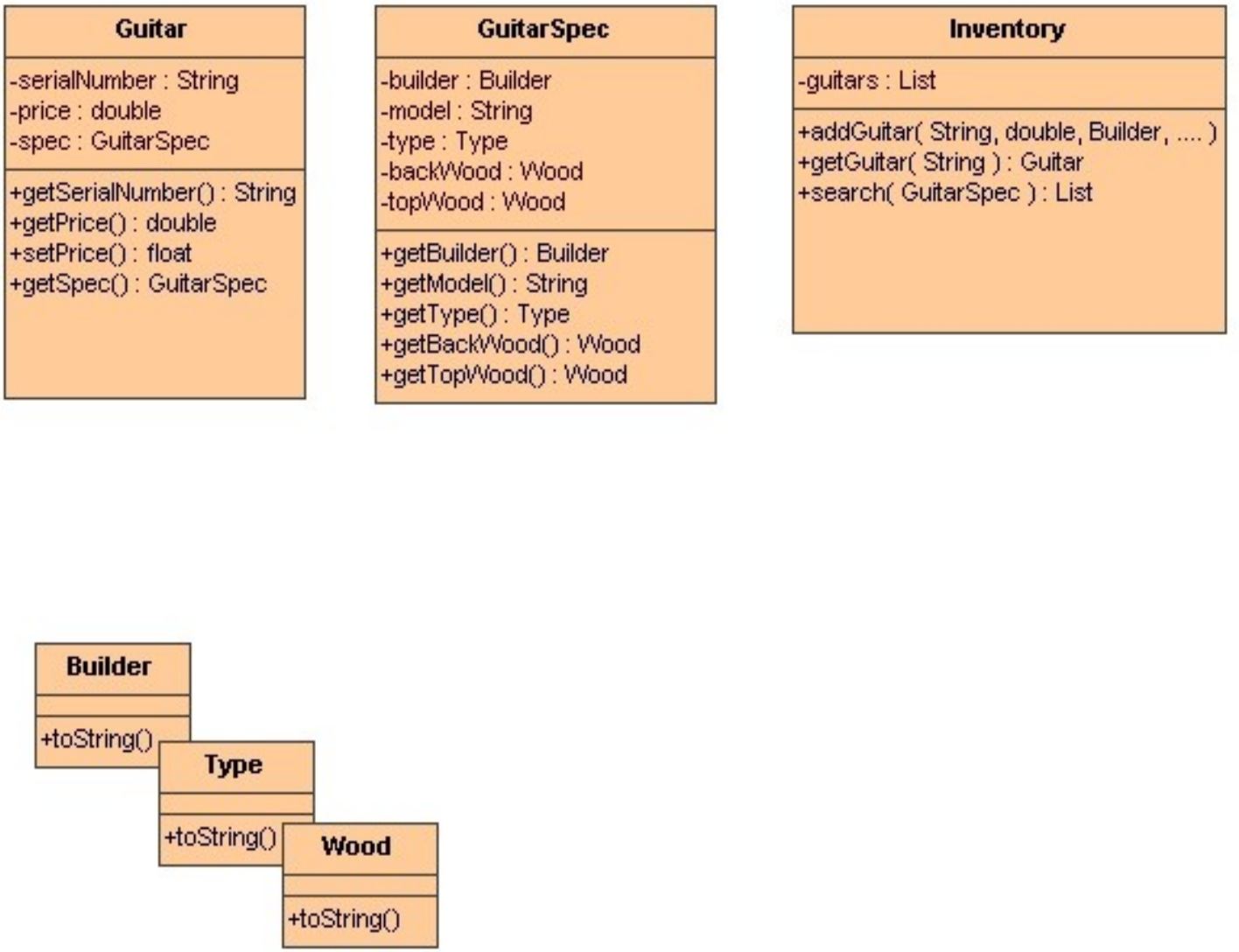


- Tutte le volte che vede del codice duplicato , cercate un posto dove incapsularlo!

Questa è una prima fase di lavorazione



E questa potrebbe essere una soluzione

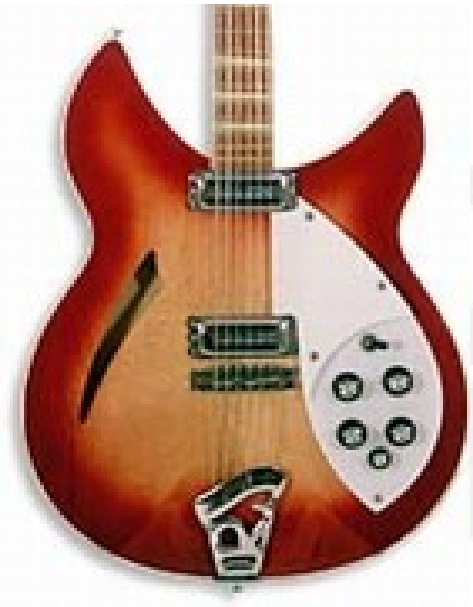


Il codice java dopo l'introduzione di GuitarSpec

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

  GuitarSpec whatRickyPorteraLikes =
    new GuitarSpec(Builder.FENDER, "Stratocastor",
                   Type.ELECTRIC, Wood.ALDER, Wood.ALDER);
  List matchingGuitars = inventory.search(whatRickyPorteraLikes);
  if (!matchingGuitars.isEmpty()) {
    System.out.println("RickyPortera, 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, RickyPortera, we have nothing for you.");
  }
}
```

Adesso Tony vuole vendere chitarre con 12 corde e quindi permettere ai suoi clienti di ricercarle



Quanto facile è applicare questo cambiamento alla applicazione di Toni?

- Stiamo aggiungendo un attributo alla classe GuitarSpec ma dobbiamo cambiare il codice nella classe Inventory nel metodo search() così come nel costruttore della classe Guitar

Collegare correttamente i titoli a dx con le definizioni a sn

Flessibilità

Incapsulamento

Funzionalità

Design pattern

Senza di me non riuscirai a rendere veramente felice il cliente

Riguardo il riuso e essere sicuro che non tu non stai tentando di risolvere un problema che qualcun altro ha già risolto

Usami per tenere le parti del tuo codice che non cambiano separate dalle parti che cambiano; quindi sarà facile cambiare il codice senza rompere tutto

Usami quando il tuo software può cambiare e crescere senza continuo rilavoro. Prevengo la tua applicazione dall'essere fragile

Il problema:

Aggiungere una proprietà a **GuitarSpec.java** comporta dei cambiamenti nel codice a **Guitar.java** e **Inventory.java**.

La applicazione deve essere ristrutturata in modo tale che aggiungere proprietà a GuitarSpec non impatti il codice nel resto della applicazione

- Delega: l'atto di un oggetto che trasferisce una operazione a un'altro oggetto , per essere eseguita al posto del primo oggetto

1 aggiungiamo numStrings a GuitarSpec

2 modifichiamo Guitar.java

3 cambiamo il metodo search() in Inventory per delegare il confronto tra i due GuitarSpec alla classe stessa invece di eseguire direttamente il confronto

L'ultimo test

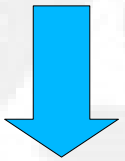
(e una applicazione pronta per il riuso)

Congratulazioni!

Avete trasformato lo strumento di ricerca sull'inventory di Tony, che inizialmente non funzionava, in un pezzo ben progettato di gran software

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

2. Apply basic OO principles to add flexibility



3. Strive for a a maintainable, reusable design

- I clienti sono soddisfatti quando le loro applicazioni **FUNZIONANO**
- I clienti sono soddisfatti quando le loro applicazioni **CONTINUANO A FUNZIONARE**
- I clienti sono soddisfatti quando le loro applicazioni possono **ESSERE AGGIORNATE**
- I programmatori sono soddisfatti quando le loro applicazioni possono essere **RIUTILIZZATE**
- I programmatori sono soddisfatti quando le loro applicazioni sono **FLESSIBILI**

Questa presentazione è stata liberamente ispirata dal libro

Object-Oriented Analysis & Design

Brett D. McLaughlin, Gaty Pollice & David West
della collana **Head First** della O'Reilly.
Vi consiglio tutti i libri di questa collana

- Sito Web:
  - `http://www.jugpadova.it`
- Mailing List:
  - !Attenzione nuova mailing list su googlegroups
  - `http://groups.google.com/group/jugpadova`
- Persone di riferimento
  - Dario Santamaria (dario.santamaria@jugpadova.it)
  - Lucio Benfante (lucio.benfante@jugpadova.it)
  - Paolo Donà (paolo.dona@jugpadova.it)