MAC0332

Engenharia de Software

Design Orientado a Objetos (OOD)

Gustavo Ansaldi Oliva

goliva@ime.usp.br

golivax@gmail.com

Apresentação

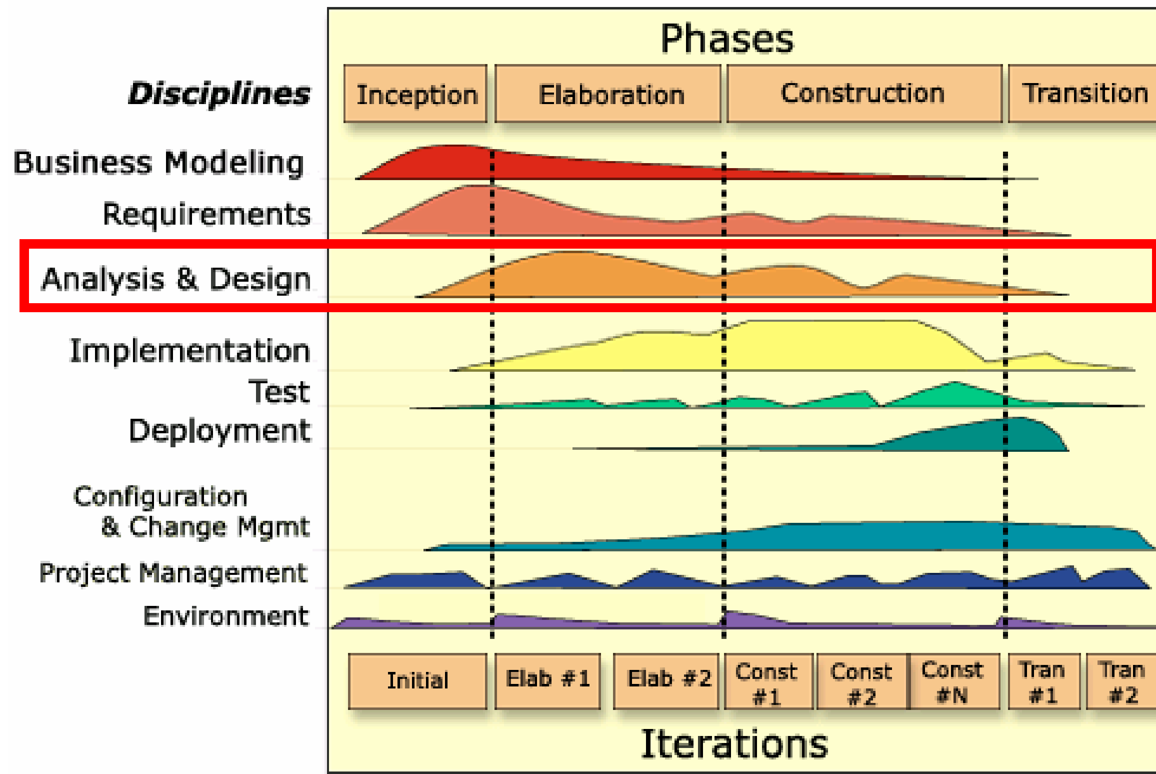
- Gustavo Ansaldi Oliva, 25 anos
- Mestrando em Ciência da Computação pelo IME-USP
- Ex IT Specialist na IBM Brasil
- Experiência de aproximadamente 7 anos com desenvolvimento de software orientado a objetos
- Áreas de Interesse
  - Análise/Design OO
  - Arquitetura de Software
  - Processos de Desenvolvimento de Software (RUP/Agile)
  - Java (J2SE/J2EE/J2ME)
- Guitarrista nas poucas horas vagas :-)

Uma breve revisão

- UML
  - Visualizar
  - Especificar
  - Construir (Forward Engineering/Reverse Engineering)
  - Documentar
- Significado de “classe” em diferentes perspectivas
  - Classe Conceitual/Conceito de Domínio
  - Classe de Software
  - Classe de Implementação
- Objetos
  - Estado (atributos)
  - Comportamento (operações)
  - Ligações com outros objetos para troca de mensagens

Design no contexto do (R)UP

- Objetivos da disciplina de “Análise & Design” (OOAD)
  - Transformar os requisitos em um design do sistema futuro
  - Evoluir uma arquitetura robusta para o sistema
  - Adaptar o design para casar com o ambiente de implementação, preocupando-se com desempenho



Análise x Design

- Análise
  - Foco no entendimento do problema (*O que?)*
  - Design idealizado
  - Comportamento
  - Estrutura do Sistema
  - Requisitos Funcionais
- Design
  - Foco no entendimento da solução (*Como?)*
  - Operações e atributos
  - Desempenho
  - Perto do código real
  - Ciclo de vida dos objetos
  - Requisitos não-funcionais

Arquitetura x Design

- Arquitetura
  - Design estratégico
  - Preocupação com aspectos globais
  - Paradigmas
- Design
  - Design táctico
  - Design Patterns
  - Refactoring

Como escrever bom software?

- A disciplina de Análise e Design Orientados a Objetos sugere três passos básicos

1. **Garanta que o seu software faz exatamente o que os clientes esperam que ele faça**
   - Análise OO, testes
2. **Aplique princípios básicos de design OO para adicionar flexibilidade**
   - Remover código duplicado e técnicas programação OO
3. **Evolua o design para que ele se torne manutenível e reutilizável**
   - Padrões e princípios OO

Um estudo de caso

- Rick's Guitars
  - Manter um inventário de guitarras
  - Localizar guitarras para clientes

Um estudo de caso

- Rick's Guitars

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

Este design faz algum sentido para você?

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

Classe Guitar

```
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;
  }
```

Classe Guitar - Continuação

```
  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;}
}
```

Classe Inventory

```
imports...

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);
  }
```

Classe Inventory (continuação)

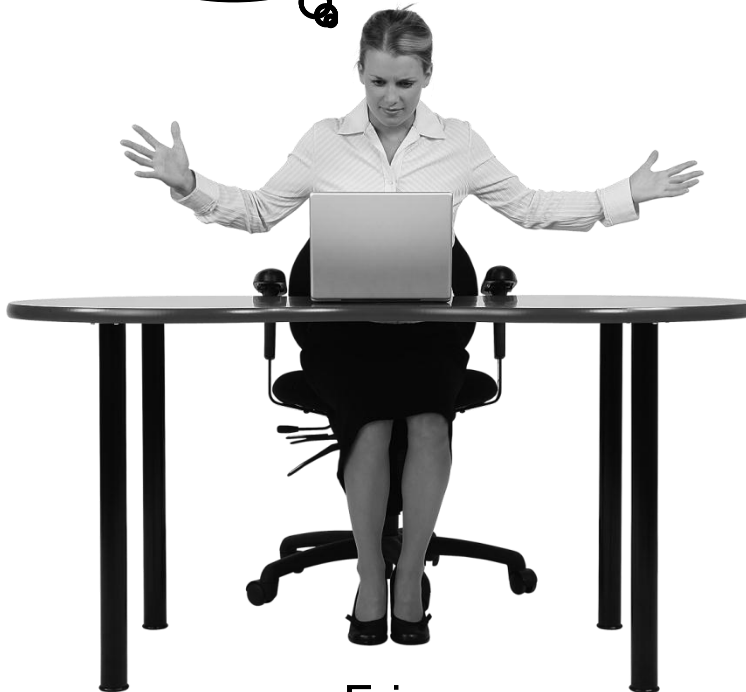
```
public Guitar getGuitar(String serialNumber) {
    for (Iterator i = guitars.iterator(); i.hasNext(); ) {
      Guitar guitar = (Guitar)i.next();
      if (guitar.getSerialNumber().equals(serialNumber)) {
        return guitar;
      }
    }
    return null;
}
```

Classe Inventory (continuação)

```
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 prince 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 guitar;
  }
  return null;
}
```

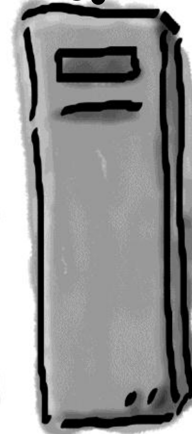
Mas existe um problema…

Estou procurando por uma fender strato…

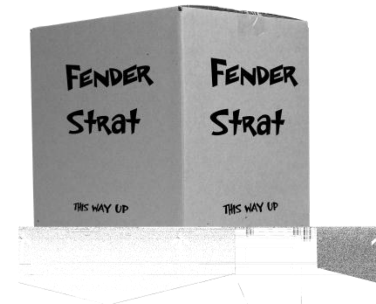


Erin

Não está no estoque



Estoque do Rick



Um teste para a busca

Here's our test program that reveals a problem with the 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, "fender", "Stratocastor",
                                "electric", "Alder", "Alder");
    Guitar guitar = inventory.search(whatErinLikes);
    if (guitar != null) {
```

Rick's app should match Erin's preferences here...

...to this guitar in Rick's inventory.

```
inventory.addGuitar("V95693",
  1499.95, "Fender", "Stratocastor",
  "electric", "Alder", "Alder");
```

FindGuitarTester.java

```
File Edit Window Help C7#5
%java FindGuitarTester
Sorry, Erin, we have nothing for you.
```

Manutenção na aplicação

- **O que fazer primeiro?**
  - Você se lembra do primeiro passo da OOAD?

    Garanta que o seu software faz exatamente o que os clientes esperam que ele faça
- Vamos revisar o design e a implementação
- Alguém identifica o problema?
- Quais as possíveis soluções?

Substituindo as Strings por Enums

```
public class Guitar {

  private String serialNumber,
          model;
  private double price;
  private Builder builder;
  private Type type;
  private Wood backWood, topWood;
…
```

```
public enum Type {
  ACOUSTIC("acoustic"),
  ELECTRIC("electric"),
  UNSPECIFIED("unspecified");
  String value;
  private Type(String value)
  {
    this.value = value;
  }
  public String toString() {
    return value;
  }
}
```

O novo método de busca

```
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;
}
```

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.

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

Inventory.java

A nova classe de teste

```
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.

FindGuitarTester.java

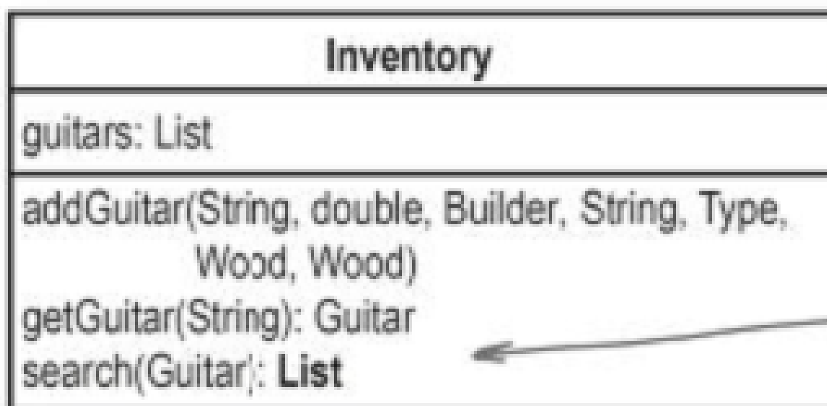
Os requisitos mudam

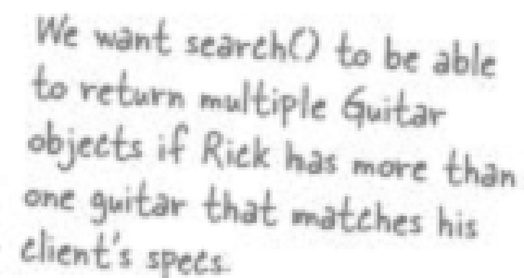
- Rick ficou feliz ao ver a aplicação funcionando e teve uma outra idéia…

Eu achei que a aplicação estava perfeita, mas aí percebi que eu tenho duas guitarras que satisfazem a busca da Erin. Você pode fazer com que a ferramenta de busca retorne as duas?

Alterando o design novamente

- Precisamos mudar a assinatura e o método da operação *search* na classe Inventory

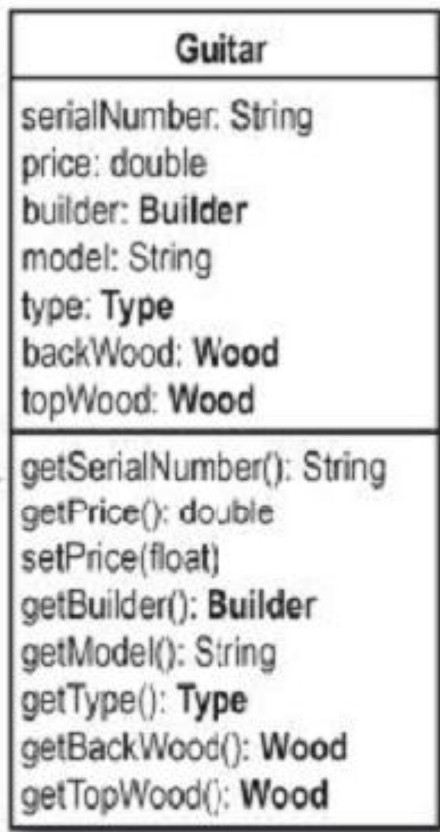


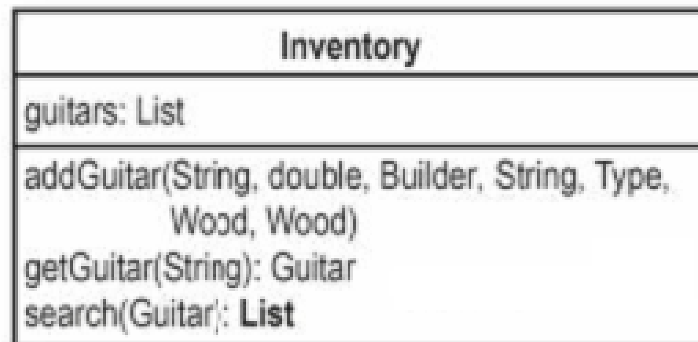


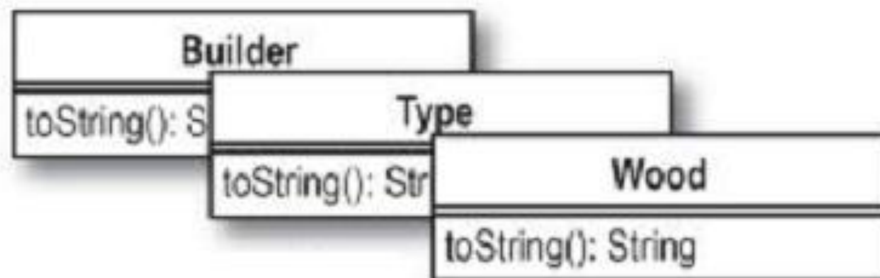
O método de busca

```
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;
}
```

O novo design da aplicação







O teste

```
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.

FindGuitarTester.java

O teste

- 1. Garanta que o seu software faz exatamente o que os clientes esperam que ele faça

```
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!
  ----
```

Os requisitos podem aumentar mais ainda

- O que o cliente quer?

Clientes nem sempre sabem todas as características da guitarra que eles querem

Eu preciso de relatórios e outras capacidades no meu inventário, mas o meu problema número um é achar a guitarra certa para o cliente

Clientes costumam procurar por uma guitarra num intervalo de preço específico

O segundo passo da OOAD

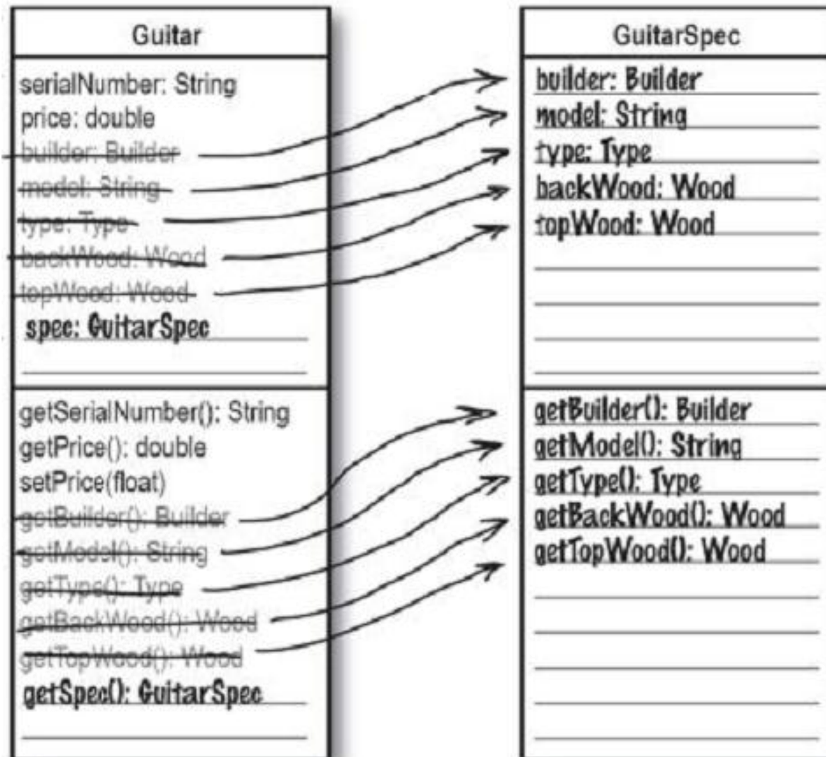
- **2. Aplique princípios básicos de design OO para adicionar flexibilidade**
- A partir de agora, o nosso foco será o design da aplicação
- Vamos examinar com mais cuidado a operação *search* da classe Inventory

Encapsulamento

- Clientes do Rick não fornecem guitarras para serem buscadas
  - Fornecem especificações gerais de guitarras
- Encapsular as especificações gerais de guitarras em uma outra classe
- **Def. Encapsulamento é um mecanismo usado para ocultar os dados, a estrutura interna e os detalhes de implementação de algum elemento, como um objeto ou subsistema.**
- Encapsule aquilo que varia: separe o que muda daquilo que se mantém
  - Permite realizar manutenção localizada
- Toda vez que se deparar com código duplicado, procure um lugar para encapsular este código e remover a duplicação

Encapsulamento

- Baseado no diagrama abaixo, devemos atualizar as classes Guitar, Inventory (método de busca) e FindGuitarTester
- **Def. Refatoração é uma técnica disciplinada para reestruturar um trecho de código existente, alterando sua estrutura interna sem alterar seu comportamento observável**



Esta alteração não muda o comportamento da aplicação. Porém, torna o design muito mais flexível!

Nova classe Guitar

```
public class Guitar {

  private String serialNumber;
  private double price;
  private GuitarSpec guitarSpec;

  public Guitar(String serialNumber, double price,
                Builder builder, Model model, Type type,
                Wood backWood, Wood topWood) {


  this.serialNumber = serialNumber;
  this.price = price;
  guitarSpec = new GuitarSpec(builder, model, type,
backWood, topWood);
  }
}
```

O Teste

- 2. Aplique princípios básicos de design OO para adicionar flexibilidade

```
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!
  ----
```

O terceiro passo da OOAD

- **3. Evolua o design para que ele se torne manutenível e reutilizável**
- Quão fácil é estender a aplicação?
- Quão fácil é entender a aplicação?
- A aplicação é manutenível?

Muitos clientes estão procurando por guitarras de 12 cordas. Talvez nós devessemos incluir isto nas características das guitarras que gravamos

O terceiro passo da OOAD

- Em quais classes é necessário mexer para satisfazer o novo requisito?
  - Guitar (construtor que recebe todos os parâmetros do GuitarSpec e cria um GuitarSpec ele mesmo)
  - GuitarSpec (adicionar a propriedade numString)
  - Inventory (alterar assinatura da operação *addGuitar* e o método *search*)

GuitarSpec
builder: Builder model: String type: Type backWood: Wood topWood: Wood
getBuilder(): Builder getModel(): String getType(): Type getBackWood(): Wood getTopWood(): Wood

Guitar
serialNumber: String price: double spec: GuitarSpec
getSerialNumber(): String getPrice(): double setPrice(float) getSpec(): GuitarSpec

Inventory
guitars: List
addGuitar(String, double, Builder, String, Type, Wood, Wood) getGuitar(String): Guitar search(**GuitarSpec**): List

Mais encapsulamento

- Encapsular as especificações da guitarra e isolá-las do resto da ferramenta de busca
  - **Adicionar uma propriedade deveria apenas afetar a classe GuitarSpec!**
- Vamos realizar os seguintes passos
  1. Adicionar a propriedade *numStrings* e o seu respectivo *get* para GuitarSpec
  2. Modificar *Guitar* para que as propriedades de *GuitarSpec* não fiquem visíveis
  3. Mudar o método da operação *search* para **delegar** a comparação de duas instâncias de *GuitarSpec* para a classe *GuitarSpec*
  4. Atualizar a classe de teste

Tornando a aplicação manutenível

- O primeiro passo é trivial
- O segundo evolve a utilização mais inteligente de encapsulamento
  - Guitar recebe um GuitarSpec no construtor
  - Quem deve criar o GuitarSpec agora?
    - **Padrão de Projeto Factory: Em situações como esta, crie uma classe unicamente responsável por fabricar objetos (GuitarSpecs, neste caso)**
- O terceiro passo faz uso de dois importantes conceitos
  - **Def. Delegação é o ato de um objeto encaminhar uma mensagem recebida para outro objeto ao invés de tratá-la diretamente**
  - **Information Expert (GRASP): Atribua responsabilidade à classe que tenha informação necessária para satisfazê-la**
- O último passo é trivial

Classe Inventory - Delegação

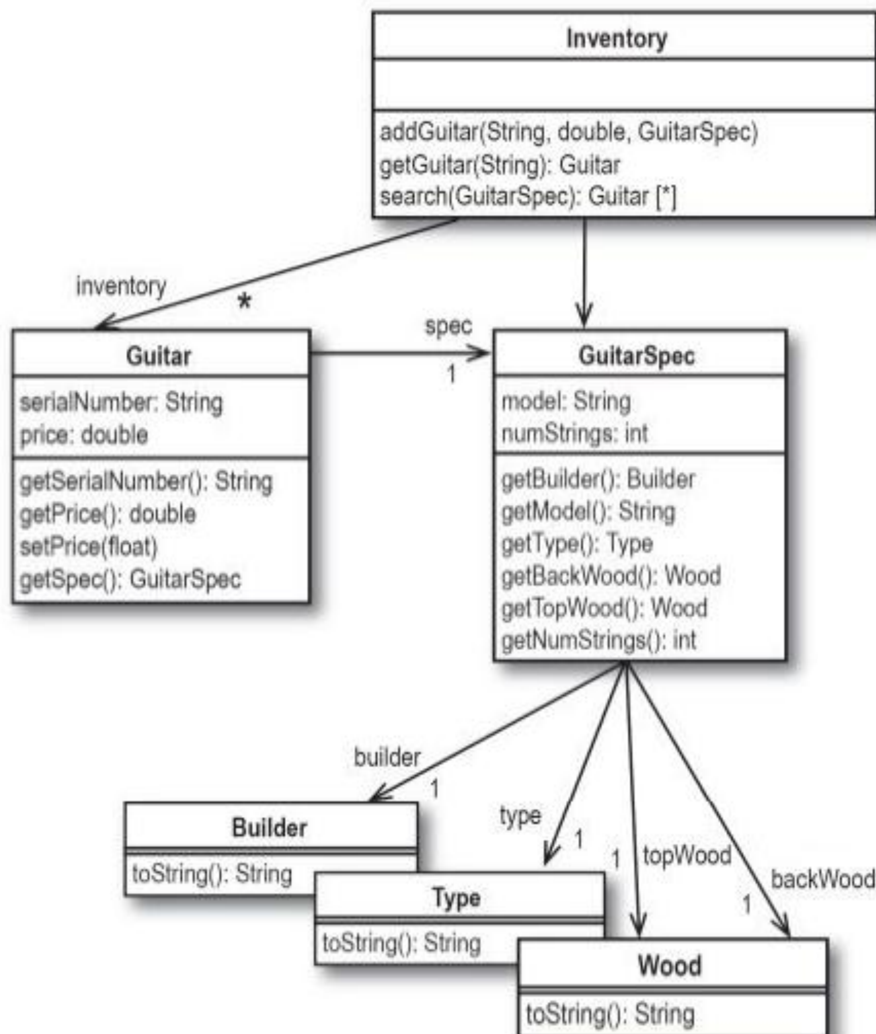
```
public class Inventory {
…
 public List search(GuitarSpec searchSpec) {
    List matchingGuitars = new LinkedList();
    for (Iterator i = guitars.iterator(); i.hasNext(); ) {
      Guitar guitar = (Guitar)i.next();
      if (guitar.getSpec().matches(searchSpec))
        matchingGuitars.add(guitar);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

Classe GuitarSpec – Information Expert

```
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;
  }
…
}
```

O Design Atualizado



O Teste

- 3. Evolua o design para que ele se torne manutenível e reutilizável

```
File Edit Window Help ReuseRules
%java FindGuitarTester
Erin, you might like these guitars:
  We have a Fender Stratocastor 6-string electric guitar:
     Alder back and sides,
     Alder top.
  You can have it for only $1499.95!
  ----
  We have a Fender Stratocastor 6-string electric guitar:
     Alder back and sides,
     Alder top.
  You can have it for only $1549.95!
  ----
```

Design não é fácil e nem simples

Projetar software orientado a objetos é difícil, e projetar software orientado a objetos reutilizável é ainda mais difícil. Você deve encontrar objetos pertinentes, fatorá-los em classes na granularidade correta, definir interfaces de classes e hierarquias de herança, e estabelecer relações chave entre eles. Seu design deve ser específico para o problema em mãos, mas também deve ser genérico o suficiente para tratar problemas futuros e requisitos.

– Design Patterns: Elements of Reusable Object-Oriented Software, Gang of Four (GoF)

Mais conceitos e princípios

- Acoplamento
  - É uma medida de quão fortemente um elemento está conectado a, tem conhecimento de, ou depende de outros elementos
  - Princípio do baixo acoplamento (low coupling)
- Coesão
  - Coesão mede o grau de conectividade entre os elementos de um mesmo módulo, classe ou objeto. Cada classe deve executar um conjunto bastante específico de ações intimamente relacionadas
  - Princípio da alta coesão (high cohesion)

Mais conceitos e princípios

- Interfaces
  - Codificar para uma interface ao invés de codificar para uma implementação torna o seu software **mais fácil de ser estendido e facilita a testabilidade**
  - Código funcionará com todas as implementações da interface – inclusive aquelas que ainda não foram criadas
  - OCP: Open/Closed Principle
- Mudanças
  - São inevitáveis e frequentes
  - Bons softwares são fáceis de serem mudados
  - Métodos ágeis aceitam “com prazer” mudanças nos requisitos mesmo quando aparecem tardiamente
  - Cada classe deve ter uma única razão para mudar, uma única responsabilidade
  - SRP: Single Responsibility Principle

Mais conceitos e princípios

- DRY (Don’t Repeat Yourself)
  - Evite código duplicado abstraindo coisas que são comuns e as colocando em um local único
  - Um requisito, um lugar
- Liskov Substitution Principle
  - Subtipos devem ser substituíveis por seu tipo base
  - Uso correto de herança: deve ser possível substituir a subclasse pela classe base sem que haja efeitos colaterais
- Composição
  - Um objeto composto de outros objetos torna-se “dono” deles
  - Quando o objeto é destruido, todos as suas partes também são (controle do ciclo de vida)
- Herança
  - Deve ser utilizada para explorar polimorfismo
  - Favoreça delegação, agregação e composição em vez de herança

“Maus cheiros” de design

- **Rigidez**
  - Difícil de mudar
  - Efeito dominó
- **Fragilidade**
  - Quebra em muitos lugares na presença de mudanças
  - Efeito borboleta-maremoto
- **Imobilidade**
  - Baixo grau de reúso
  - Efeito banana-gorila
- **Viscosidade**
  - Design não preparado para mudanças
  - Fácil fazer a coisa errada e difícil fazer a coisa certa
- **Complexidade desnecessária**
- **Repetição desnecessária**

Design no OpenUP – Práticas e Conceitos

- Design Evolutivo
  - Prática para design que assume que o design vai evoluir ao longo do tempo e a documentação será minimizada porém ainda efetiva
  - A cada revisão do design, a solução será ampliada, refinada e refatorada
- Test Driven Development
  - Abordagem em que os casos de teste são definidos primeiro e em seguida é escrito o código que faz os testes passarem
  - Criado originalmente como prática no processo ágil XP (eXtreme Programming)
  - Design “emerge”
    - Não é Silver Bullet! Dominar OOAD é essencial
- Mecanismos de Design
  - Padrões de design e frameworks
  - Relativamente independente de implementação (ex: ORM)

Bibliografia

- UML Essencial 3ed, Martin Fowler
- Head First Object Oriented Analysis and Design, Brett D. McLaughlin, Gary Pollice & David West
- Applying UML and Patterns 3ed, Craig Larman
- Agile Principles, Patterns and Practices in C#, Robert C. Martin (Uncle Bob)
- Design Patterns: Elements of Reusable Object-Oriented Software, Gang of Four (GoF)