Padrões de Projeto de Software Orientados a Objetos

Tecnologia em Análise e Desenvolvimento de Sistemas

Paulo Mauricio Gonçalves Júnior

Instituto Federal de Educação, Ciência e Tecnologia de Pernambuco

7 de maio de 2018

Parte I

Iterator e Composite

Iterator I

- Dois restaurantes se juntaram. A implementação dos itens do menu foi unificada mas a forma de armazenamento deles não.

```java
public class MenuItem {
    String name;
    String description;
    boolean vegetarian;
    double price;
    public MenuItem(String name,
            String description,
            boolean vegetarian,
            double price) {
        this.name = name;
        this.description = description;
        this.vegetarian = vegetarian;
        this.price = price;
    }
    public String getName() {
        return name;
    }
```

Iterator II

```
    public String getDescription() {
        return description;
    }
    public double getPrice() {
        return price;
    }
    public boolean isVegetarian() {
        return vegetarian;
    }
}

public class PancakeHouseMenu implements Menu {
    ArrayList<MenuItem> menuItems;
    public PancakeHouseMenu() {
        menuItems = new ArrayList<MenuItem>();
        addItem("K&B's Pancake Breakfast",
                "Pancakes with scrambled eggs, and toast",
                true,
                2.99);
        addItem("Regular Pancake Breakfast",
                "Pancakes with fried eggs, sausage",
```

Iterator III

```
                    false,
                    2.99);
    }
    public void addItem(String name, String description,
            boolean vegetarian, double price) {
        MenuItem menuItem = new MenuItem(name, description,
            vegetarian, price);
        menuItems.add(menuItem);
    }
    public ArrayList<MenuItem> getMenuItems() {
        return menuItems;
    }
    // other menu methods here
}

public class DinerMenu implements Menu {
    static final int MAX_ITEMS = 6;
    int numberOfItems = 0;
    MenuItem[] menuItems;
    public DinerMenu() {
        menuItems = new MenuItem[MAX_ITEMS];
```

Iterator IV

```
        addItem("Vegetarian BLT",
                "(Fakin') Bacon with lettuce & tomato on whole
                    wheat", true, 2.99);
        addItem("BLT",
                "Bacon with lettuce & tomato on whole wheat", false
                    , 2.99);
    }
    public void addItem(String name, String description,
            boolean vegetarian, double price) {
        MenuItem menuItem = new MenuItem(name, description,
            vegetarian, price);
        if (numberOfItems >= MAX_ITEMS) {
            System.err.println("Sorry, menu is full!  Can't add
                item to menu");
        } else {
            menuItems[numberOfItems] = menuItem;
            numberOfItems = numberOfItems + 1;
        }
    }
    public MenuItem[] getMenuItems() {
        return menuItems;
```

Iterator V

```
    }
    // other menu methods here
}
```

- Neste caso, a garçonete precisa conhecer os dois menus e como percorrê-los.
- Isto torna sua implementação difícil de manter e estender.

```
PancakeHouseMenu pancakeHouseMenu = new PancakeHouseMenu();
ArrayList breakfastItems = pancakeHouseMenu.getMenuItems();
DinerMenu dinerMenu = new DinerMenu();
MenuItem[] lunchItems = dinerMenu.getMenuItems();

for (int i = 0; i < breakfastItems.size(); i++) {
  MenuItem menuItem = (MenuItem)breakfastItems.get(i);
  System.out.print(menuItem.getName() + " ");
  System.out.println(menuItem.getPrice() + " ");
  System.out.println(menuItem.getDescription());
}
```

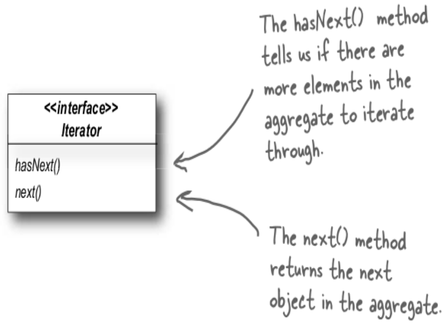
Iterator VI

```
for (int i = 0; i < lunchItems.length; i++) {
  MenuItem menuItem = lunchItems[i];
  System.out.print(menuItem.getName() + " ");
  System.out.println(menuItem.getPrice() + " ");
  System.out.println(menuItem.getDescription());
}
```

- Em ambos os casos, devemos saber o tamanho e como obter um item da coleção.
- Podemos criar um objeto Iterator que encapsula a forma de iterar em uma coleção de objetos.

```
Iterator iterator = breakfastMenu.createIterator();
while (iterator.hasNext()) {
  MenuItem menuItem = (MenuItem)iterator.next();
}
```

Iterator VII


<<interface>>
Iterator
hasNext()
next()
The hasNext() method tells us if there are more elements in the aggregate to iterate through.
The next() method returns the next object in the aggregate.

Iterator VIII

```java
public interface Iterator {
  boolean hasNext();
  Object next();
}

public class DinerMenuIterator implements Iterator {
  MenuItem[] items;
  int position = 0;
  public DinerMenuIterator(MenuItem[] items) {
    this.items = items;
  }
  public Object next() {
    MenuItem menuItem = items[position];
    position = position + 1;
    return menuItem;
  }
  public boolean hasNext() {
    if (position >= items.length || items[position] == null) {
      return false;
    } else {
      return true;
```

Iterator IX

```
    }
  }
}

public class DinerMenu { implements Menu {
  static final int MAX_ITEMS = 6;
  int numberOfItems = 0;
  MenuItem[] menuItems;
  // constructor here
  // addItem here
  public MenuItem[] getMenuItems() {
    return menuItems;
  }
  public Iterator createIterator() {
    return new DinerMenuIterator(menuItems);
  }
  // other menu methods here
}
```

- O código da garçonete consegue ser simplificado.

Iterator X

```
public class Waitress {
  PancakeHouseMenu pancakeHouseMenu;
  DinerMenu dinerMenu;
  public Waitress(PancakeHouseMenu pancakeHouseMenu, DinerMenu
      dinerMenu) {
    this.pancakeHouseMenu = pancakeHouseMenu;
    this.dinerMenu = dinerMenu;
  }
  public void printMenu() {
    Iterator pancakeIterator = pancakeHouseMenu.createIterator();
        each menu.
    Iterator dinerIterator = dinerMenu.createIterator();
    System.out.println("MENU\n----\nBREAKFAST");
    printMenu(pancakeIterator);
    System.out.println("\nLUNCH");
    printMenu(dinerIterator);
  }
  private void printMenu(Iterator iterator) {
    while (iterator.hasNext()) {
      MenuItem menuItem = (MenuItem)iterator.next();
      System.out.print(menuItem.getName() + ", ");
```

Iterator XI

```
      System.out.print(menuItem.getPrice() + " -- ");
      System.out.println(menuItem.getDescription());
    }
  }
  // other methods here
}

public class MenuTestDrive {
  public static void main(String args[]) {
    PancakeHouseMenu pancakeHouseMenu = new PancakeHouseMenu();
    DinerMenu dinerMenu = new DinerMenu();
    Waitress waitress = new Waitress(pancakeHouseMenu, dinerMenu);
    waitress.printMenu();
  }
}
```

- Java já possui o padrão Iterator associado a coleções.
- Podemos simplificar o código acima usando uma interface para os Menus.

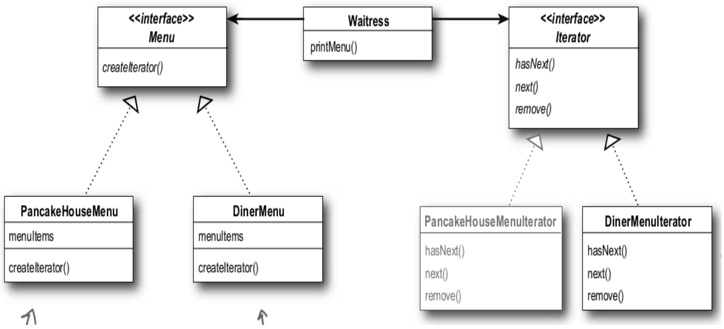
Iterator XII

```
public interface Menu {
    public Iterator<MenuItem> createIterator();
}

public class Waitress {

    Menu pancakeHouseMenu;
    Menu dinerMenu;

    public Waitress(Menu pancakeHouseMenu, Menu dinerMenu) {
        this.pancakeHouseMenu = pancakeHouseMenu;
        this.dinerMenu = dinerMenu;
    }

    public void printMenu() {
        Iterator<MenuItem> pancakeIterator = pancakeHouseMenu.
            createIterator();
        Iterator<MenuItem> dinerIterator = dinerMenu.createIterator
            ();

        System.out.println("MENU\n----\nBREAKFAST");
```

Iterator XIII

```
        printMenu(pancakeIterator);
        System.out.println("\nLUNCH");
        printMenu(dinerIterator);
    }

    private void printMenu(Iterator<MenuItem> iterator) {
        while (iterator.hasNext()) {
            MenuItem menuItem = iterator.next();
            System.out.print(menuItem.getName() + ", ");
            System.out.print(menuItem.getPrice() + " -- ");
            System.out.println(menuItem.getDescription());
        }
    }
}
```

Iterator XIV



Definição

O padrão Iterator provê uma forma de acessar os elementos de um objeto agregado sequencialmente sem expor sua representação interna.

Iterator XV

Princípio de Projeto

Uma classe deve ter apenas uma razão para mudança.

- Cada classe deve ter uma única responsabilidade.
- Para cada novo menu a ser acrescentado, devemos mudar a garçonete.
- Podemos usar coleções como atributo, permitindo que a garçonete gerencie quaisquer quantidades de menus.

Iterator XVI

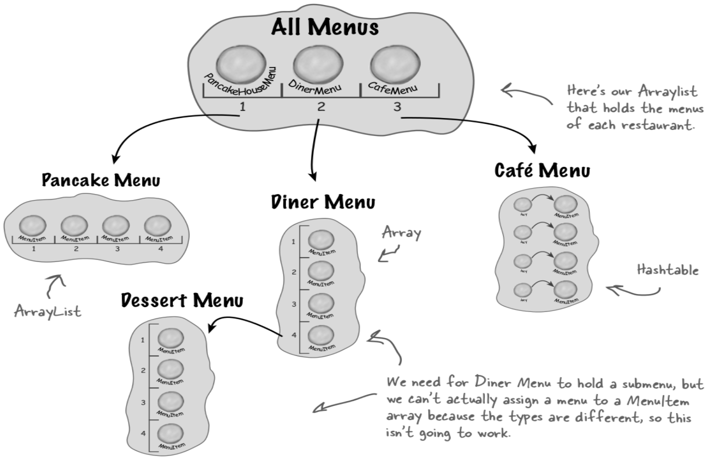
```
public class Waitress {
    ArrayList<Menu> menus;
    public Waitress(ArrayList<Menu> menus) {
        this.menus = menus;
    }
    public void printMenu() {
        Iterator<?> menuIterator = menus.iterator();
        while (menuIterator.hasNext()) {
            Menu menu = (Menu) menuIterator.next();
            printMenu(menu.createIterator());
        }
    }
    void printMenu(Iterator<?> iterator) {
        while (iterator.hasNext()) {
            MenuItem menuItem = (MenuItem) iterator.next();
            System.out.print(menuItem.getName() + ", ");
            System.out.print(menuItem.getPrice() + " -- ");
            System.out.println(menuItem.getDescription());
        }
    }
}
```

Iterator XVII

Composite I

- Agora precisamos suportar submenus, por exemplo, incluindo um menu de sobremesa como um elemento do menu de jantar.

Composite II

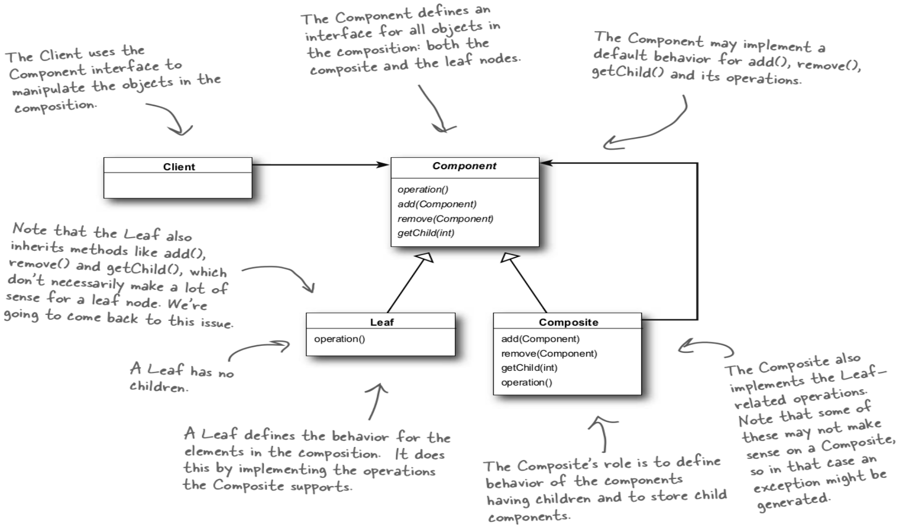

All Menus
PancakeHouseMenu
DinerMenu
CafeMenu
1
2
3
Here's our Arraylist that holds the menus of each restaurant.
Pancake Menu
MenuItem
1
2
3
4
ArrayList
Diner Menu
Array
Café Menu
key
MenuItem
Hashtable
Dessert Menu
We need for Diner Menu to hold a submenu, but we can't actually assign a menu to a MenuItem array because the types are different, so this isn't going to work.

Composite III

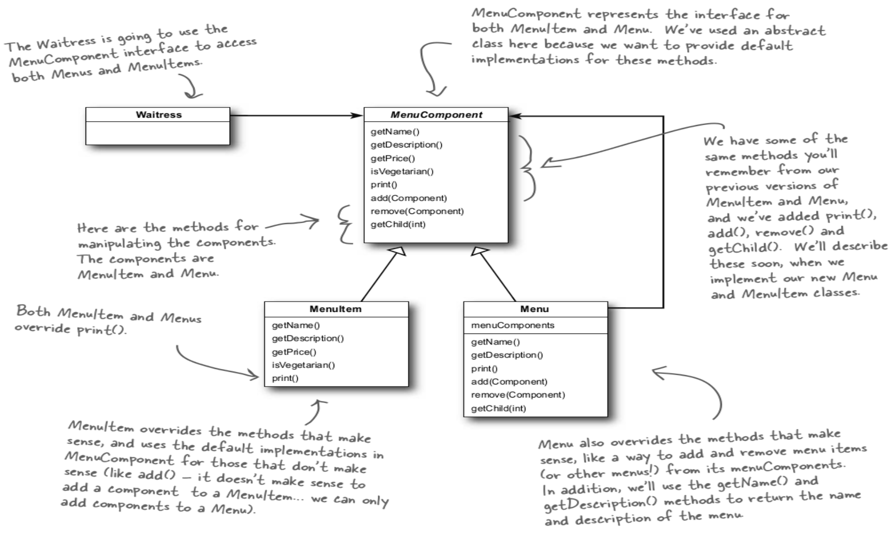
Definição

O padrão Composite permite você a compor objetos em estruturas de árvore para representar hierarquias parte-todo. Composite permite a clientes tratar objetos individuais e composições de objetos uniformemente.

Composite IV


The Client uses the Component interface to manipulate the objects in the composition.
The Component defines an interface for all objects in the composition: both the composite and the leaf nodes.
The Component may implement a default behavior for add(), remove(), getChild() and its operations.
Client
Component
operation()
add(Component)
remove(Component)
getChild(int)
Note that the Leaf also inherits methods like add(), remove() and getChild(), which don't necessarily make a lot of sense for a leaf node. We're going to come back to this issue.
Leaf
operation()
Composite
add(Component)
remove(Component)
getChild(int)
operation()
A Leaf has no children.
A Leaf defines the behavior for the elements in the composition. It does this by implementing the operations the Composite supports.
The Composite's role is to define behavior of the components having children and to store child components.
The Composite also implements the Leaf-related operations. Note that some of these may not make sense on a Composite, so in that case an exception might be generated.

Composite V


The Waitress is going to use the MenuComponent interface to access both Menus and MenuItems.
MenuComponent represents the interface for both MenuItem and Menu. We've used an abstract class here because we want to provide default implementations for these methods.
Waitress
MenuComponent
getName()
getDescription()
getPrice()
isVegetarian()
print()
add(Component)
remove(Component)
getChild(int)
We have some of the same methods you'll remember from our previous versions of MenuItem and Menu, and we've added print(), add(), remove() and getChild(). We'll describe these soon, when we implement our new Menu and MenuItem classes.
Here are the methods for manipulating the components. The components are MenuItem and Menu.
Both MenuItem and Menus override print().
MenuItem
getName()
getDescription()
getPrice()
isVegetarian()
print()
Menu
menuComponents
getName()
getDescription()
print()
add(Component)
remove(Component)
getChild(int)
MenuItem overrides the methods that make sense, and uses the default implementations in MenuComponent for those that don't make sense (like add() – it doesn't make sense to add a component to a MenuItem... we can only add components to a Menu).
Menu also overrides the methods that make sense, like a way to add and remove menu items (or other menus!) from its menuComponents. In addition, we'll use the getName() and getDescription() methods to return the name and description of the menu.

Composite VI

```
public abstract class MenuComponent {
    public void add(MenuComponent menuComponent) {
        throw new UnsupportedOperationException();
    }
    public void remove(MenuComponent menuComponent) {
        throw new UnsupportedOperationException();
    }
    public MenuComponent getChild(int i) {
        throw new UnsupportedOperationException();
    }
    public String getName() {
        throw new UnsupportedOperationException();
    }
    public String getDescription() {
        throw new UnsupportedOperationException();
    }
    public double getPrice() {
        throw new UnsupportedOperationException();
    }
    public boolean isVegetarian() {
        throw new UnsupportedOperationException();
```

Composite VII

```
    }
    public void print() {
        throw new UnsupportedOperationException();
    }
}

public class MenuItem extends MenuComponent {
    String name;
    String description;
    boolean vegetarian;
    double price;
    public MenuItem(String name,
            String description,
            boolean vegetarian,
            double price) {
        this.name = name;
        this.description = description;
        this.vegetarian = vegetarian;
        this.price = price;
    }
    public void print() {
```

Composite VIII

```
        System.out.print("  " + getName());
        if (isVegetarian()) {
            System.out.print("(v)");
        }
        System.out.println(", " + getPrice());
        System.out.println("     -- " + getDescription());
    }
}

public class Menu extends MenuComponent {
    ArrayList<MenuComponent> menuComponents = new ArrayList<
        MenuComponent>();
    String name;
    String description;
    public Menu(String name, String description) {
        this.name = name;
        this.description = description;
    }
    public void add(MenuComponent menuComponent) {
        menuComponents.add(menuComponent);
    }
```

Composite IX

```
    public void remove(MenuComponent menuComponent) {
        menuComponents.remove(menuComponent);
    }
    public MenuComponent getChild(int i) {
        return (MenuComponent) menuComponents.get(i);
    }
    public void print() {
        System.out.print("\n" + getName());
        System.out.println(", " + getDescription());
        System.out.println("---------------------");

        Iterator<MenuComponent> iterator = menuComponents.iterator
            ();
        while (iterator.hasNext()) {
            MenuComponent menuComponent
                    = (MenuComponent) iterator.next();
            menuComponent.print();
        }
    }
}
```

Composite X

```
public class Waitress {
  MenuComponent allMenus;
  public Waitress(MenuComponent allMenus) {
    this.allMenus = allMenus;
  }
  public void printMenu() {
    allMenus.print();
  }
}
```

- Podemos usar o Composite em conjunto com Iterator. Vejamos como percorrer os itens do menu e obter os pratos vegetarianos.

Composite XI

```
public class Menu extends MenuComponent {
  Iterator iterator = null;
  // other code here does not change
  public Iterator createIterator() {
    if (iterator == null) {
      iterator = new CompositeIterator(menuComponents.iterator());
    }
    return iterator;
  }
}

public class MenuItem extends MenuComponent {
  // other code here does not change
  public Iterator createIterator() {
    return new NullIterator();
  }
}

public class CompositeIterator implements Iterator<MenuComponent> {
    Stack<Iterator<MenuComponent>> stack = new Stack<Iterator<
        MenuComponent>>();
```

Composite XII

```
public CompositeIterator(Iterator<MenuComponent> iterator) {
    stack.push(iterator);
}
public MenuComponent next() {
    if (hasNext()) {
        Iterator<MenuComponent> iterator = stack.peek();
        MenuComponent component = iterator.next();
        stack.push(component.createIterator());
        return component;
    } else {
        return null;
    }
}
public boolean hasNext() {
    if (stack.empty()) {
        return false;
    } else {
        Iterator<MenuComponent> iterator = stack.peek();
        if (!iterator.hasNext()) {
            stack.pop();
            return hasNext();
```

Composite XIII

```
            } else {
                return true;
            }
        }
    }
}

public class NullIterator implements Iterator {
  public Object next() {
    return null;
  }
  public boolean hasNext() {
    return false;
  }
  public void remove() {
    throw new UnsupportedOperationException();
  }
}

public class Waitress {
    MenuComponent allMenus;
```

Composite XIV

```
    public Waitress(MenuComponent allMenus) {
        this.allMenus = allMenus;
    }
    public void printMenu() {
        allMenus.print();
    }
    public void printVegetarianMenu() {
        Iterator<MenuComponent> iterator = allMenus.createIterator
            ();
        while (iterator.hasNext()) {
            MenuComponent menuComponent = iterator.next();
            try {
                if (menuComponent.isVegetarian()) {
                    menuComponent.print();
                }
            } catch (UnsupportedOperationException e) {
            }
        }
    }
}
```