Patrones de diseño “Iterator” y “Composite”

Uniendo menús

Las cadenas de comida “Objectville Pancake House”(OPH) y “Objectville Diner”(OD) decidieron unir fuerza y hacer un solo menú, acordaron usar el mismo formato para las comidas del menú pero tienen ciertas discrepancias.


Objectville Diner
Vegetarian BLT
(Fakin') Bacon with lettu
whole wheat
2.99
BLT
Bacon with lettuce & tor
Soup of the day
A bowl of the soup of the
a side of potato salad
Hot Dog
A hot dog, with saurkrau
topped with cheese
Steamed Veggies and Bro
A medley of steamed veg
Objectville Pancake House
K&B's Pancake Breakfast
Pancakes with scrambled eggs, and toast
2.99
Regular Pancake Breakfast
Pancakes with fried eggs, sausage
2.99
Blueberry Pancakes
Pancakes made with fresh blueberries,
and blueberry syrup
3.49
Waffles
Waffles, with your choice of blueberries
or strawberries
3.59

```
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;
    }

    public String getDescription() {
        return description;
    }

    public double getPrice() {
        return price;
    }

    public boolean isVegetarian() {
        return vegetarian;
    }
}
```

Diferencias creativas

El restaurante OHP ha hecho la implementación de su menú con una ArrayList, mientras que OD lo implementó usando un arreglo, y han hecho código para que funcione con su implementación, por lo que ninguno de los dos quiere cambiar su implementación.

```
public class PancakeHouseMenu {
    ArrayList menuItems;

    public PancakeHouseMenu() {
        menuItems = new ArrayList();

        addItem("K&B's Pancake Breakfast",
            "Pancakes with scrambled eggs, and toast",
            true,
            2.99);

        addItem("Regular Pancake Breakfast",
            "Pancakes with fried eggs, sausage",
            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 getMenuItems() {
        return menuItems;
    }

    // other menu methods here
}
```

```
public class DinerMenu {
    static final int MAX_ITEMS = 6;
    int numberOfItems = 0;
    MenuItem[] menuItems;

    public DinerMenu() {
        menuItems = new MenuItem[MAX_ITEMS];
        addItem("Vegetarian BLT","(Fakin') Bacon with lettuce & tomato on whole wheat", true, 2.
        addItem("BLT","Bacon with lettuce & tomato on whole wheat", false, 2.99);
        addItem("Soup of the day","Soup of the day, with a side of potato salad", false, 3.29);
    }

    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;
    }
}
```

Java-Enabled Waitress: code-name "Alice"

```
printMenu()
   - prints every item on the menu

printBreakfastMenu()
   - prints just breakfast items

printLunchMenu()
   - prints just lunch items

printVegetarianMenu()
   - prints all vegetarian menu items

isItemVegetarian(name)
   - given the name of an item, returns true
     if the items is vegetarian, otherwise,
     returns false
```

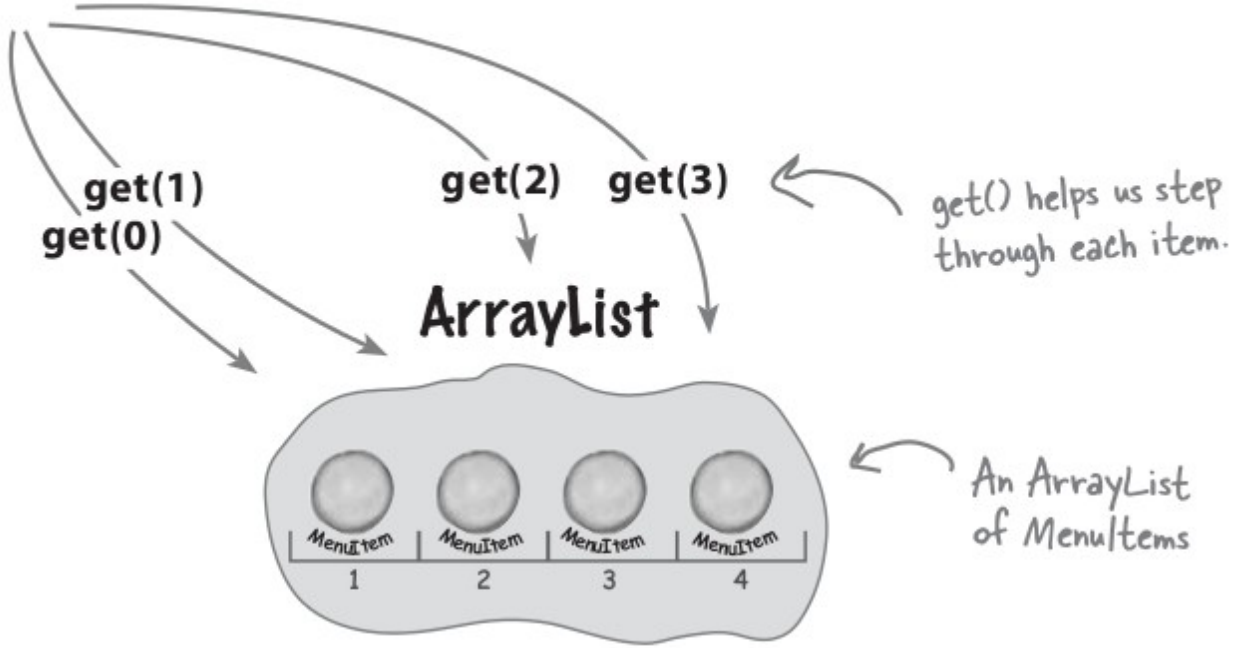
¿Que problemas hay?

Aunque ambas implementaciones tienen un método getMenuItems() cada uno devuelve objetos de distinto tipo, por lo que hay que manejarlo de distinta manera, por ejemplo para imprimir el menú.

```
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());
}

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());
}
```

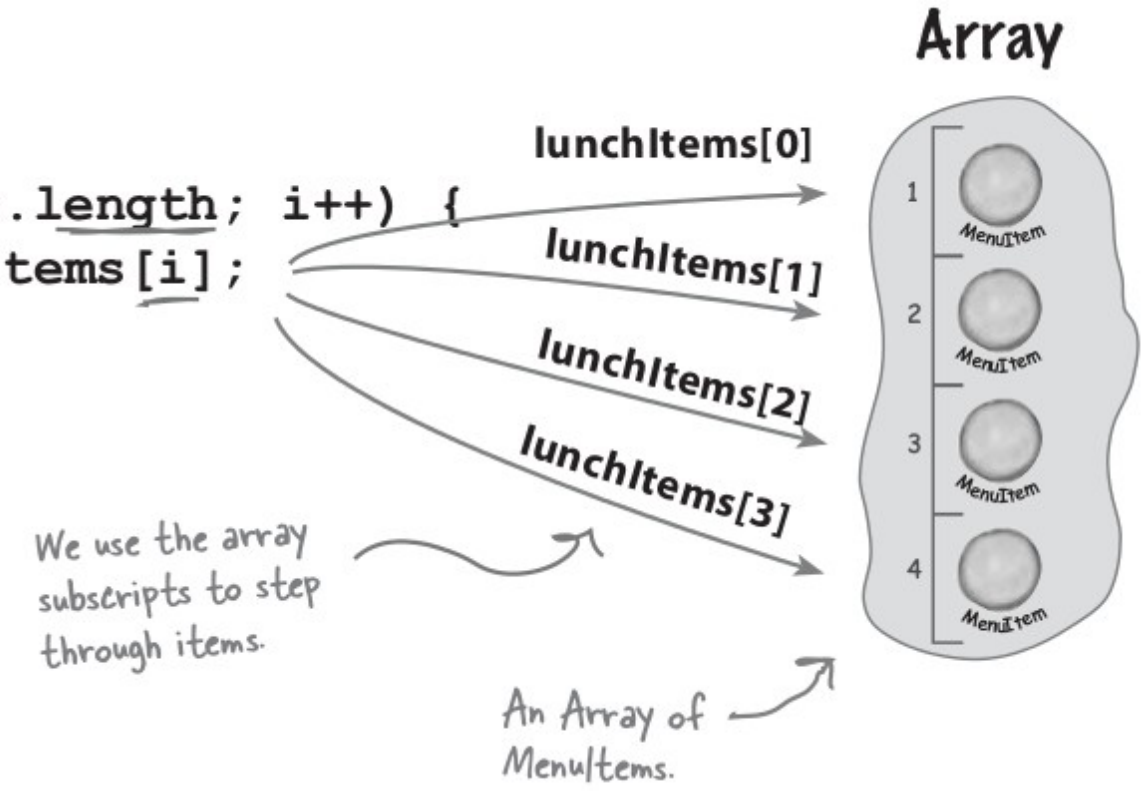
Iterando un ArrayList

```
for (int i = 0; i < breakfastItems.size(); i++) {
    MenuItem menuItem = (MenuItem)breakfastItems.get(i);
}
```



Iterando un Array

```
for (int i = 0; i < lunchItems.length; i++) {
    MenuItem menuItem = lunchItems[i];
}
```



Encapsulemos la iteración

Realmente no necesitamos tener acceso al objeto que guarda los objetos del menú, necesitamos acceso a los objetos del menú, entonces ¿Que tal si creamos un objeto que encapsule la forma de iterar a través de una colección de objetos?

Iterador de ArrayList

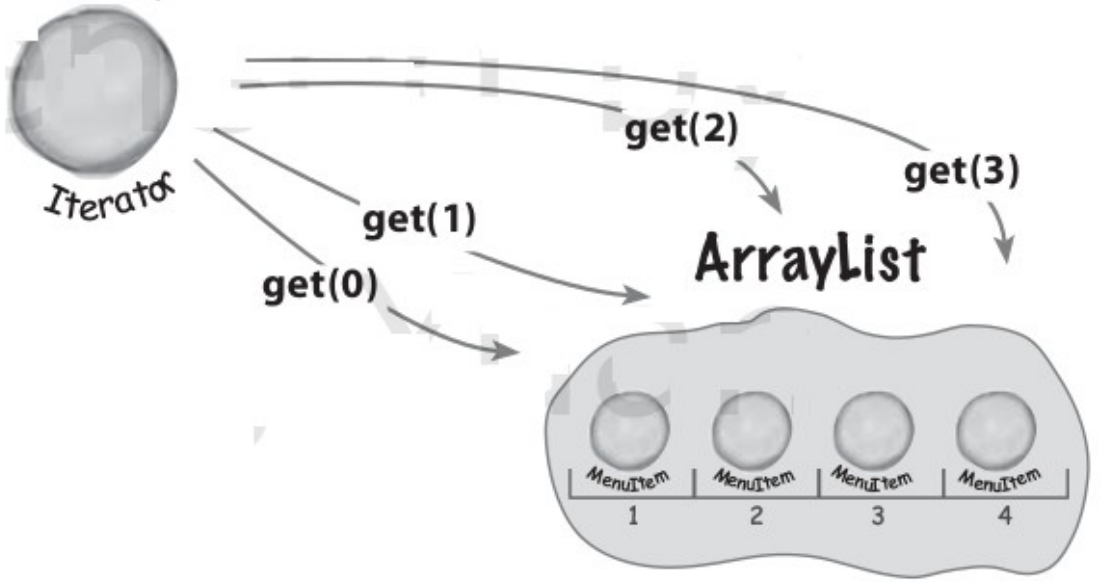
```
Iterator iterator = breakfastMenu.createIterator();

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

We ask the breakfastMenu for an iterator of its MenuItems.

And while there are more items left...

next()

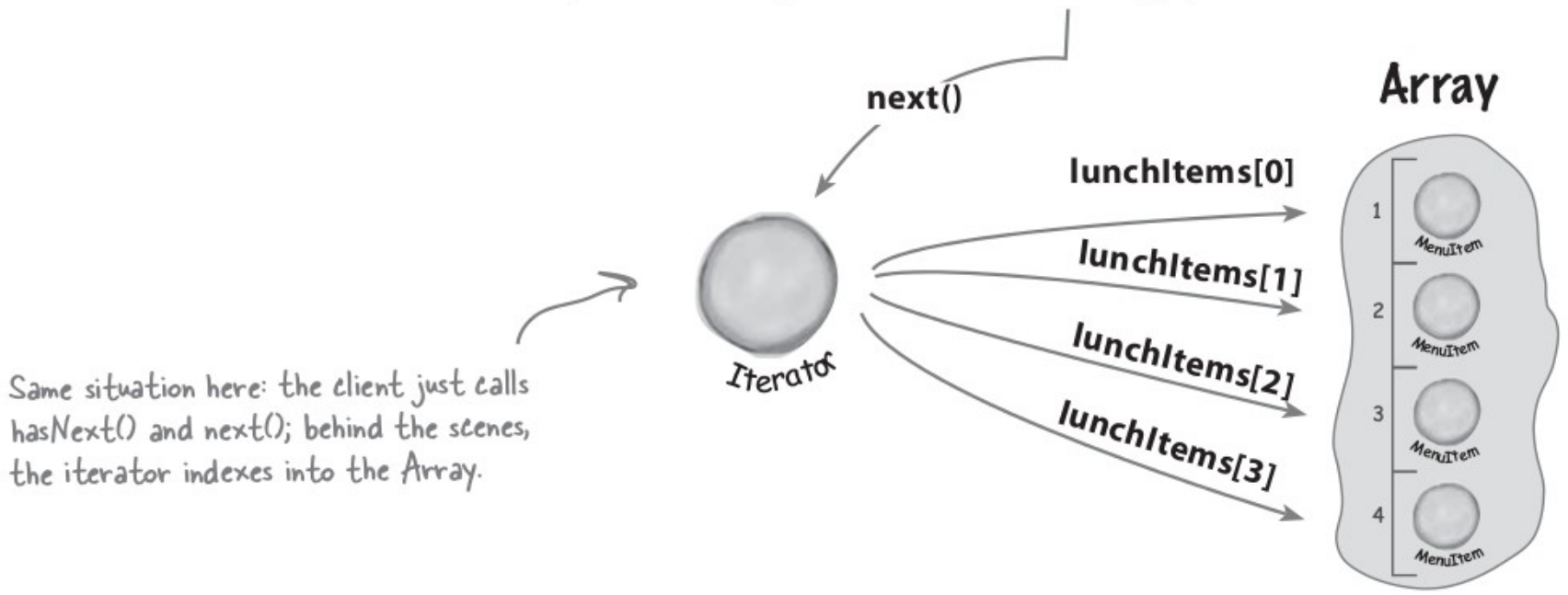
We get the next item.

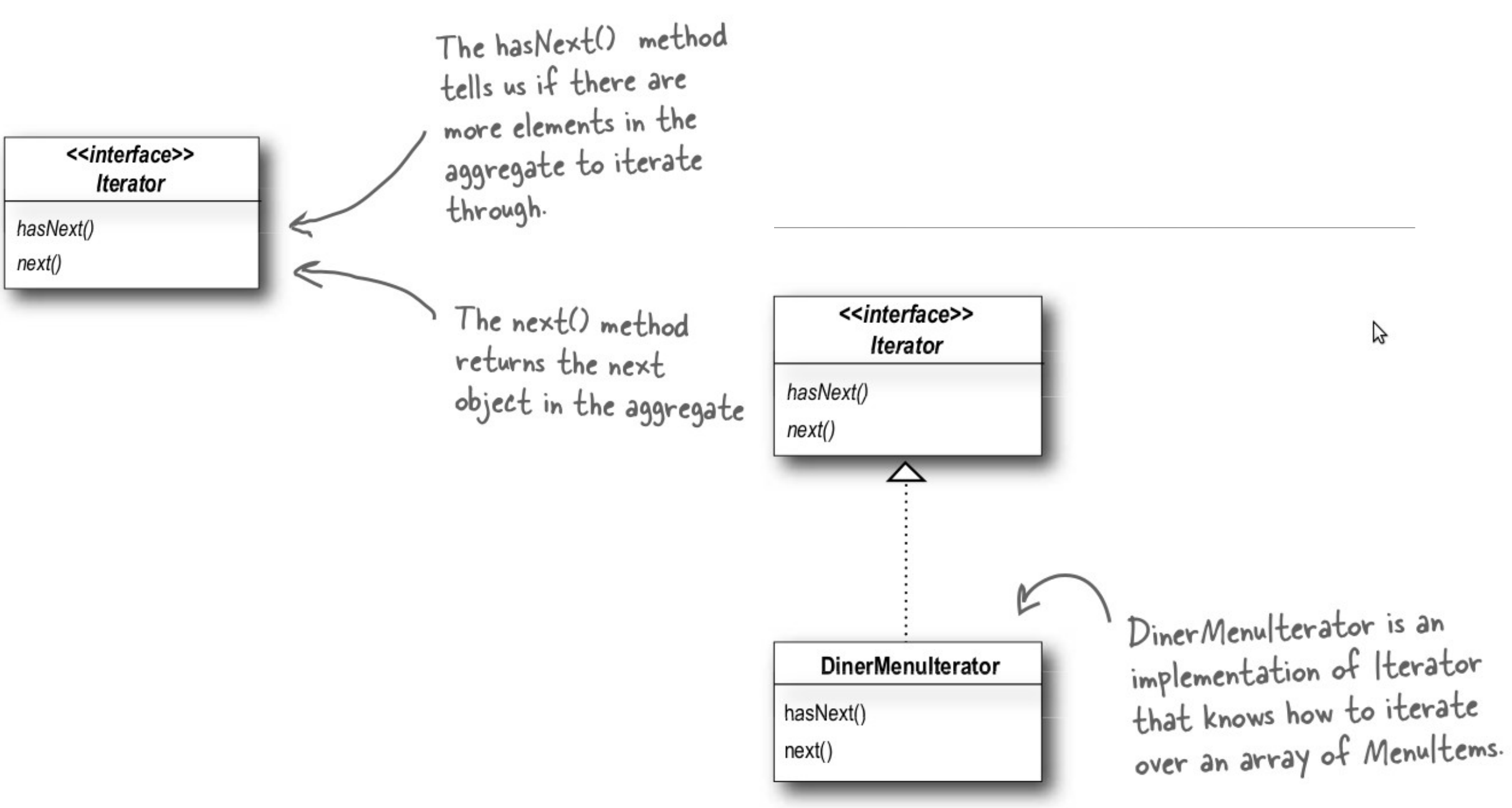


The client just calls hasNext() and next(); behind the scenes the iterator calls get() on the ArrayList.

Iterador de Array

```
Iterator iterator = lunchMenu.createIterator();

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




<<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
<<interface>>
Iterator
hasNext()
next()
DinerMenuIterator
hasNext()
next()
DinerMenuIterator is an implementation of Iterator that knows how to iterate over an array of MenuItems.

```
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;
        }
    }
}
```

Código corregido

```
public class DinerMenu {
    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
}
```

Hacemos la implementación del iterador de Pancake House(la empresa chida que uso ArrayList)

```java
public class PancakeHouseMenuIterator implements Iterator {
    ArrayList<MenuItem> items;
    int position = 0;

    public PancakeHouseMenuIterator(ArrayList<MenuItem> items) {
        this.items = items;
    }

    public MenuItem next() {
        MenuItem item = items.get(position);
        position = position + 1;
        return item;
    }

    public boolean hasNext() {
        if (position >= items.size()) {
            return false;
        } else {
            return true;
        }
    }
}
```

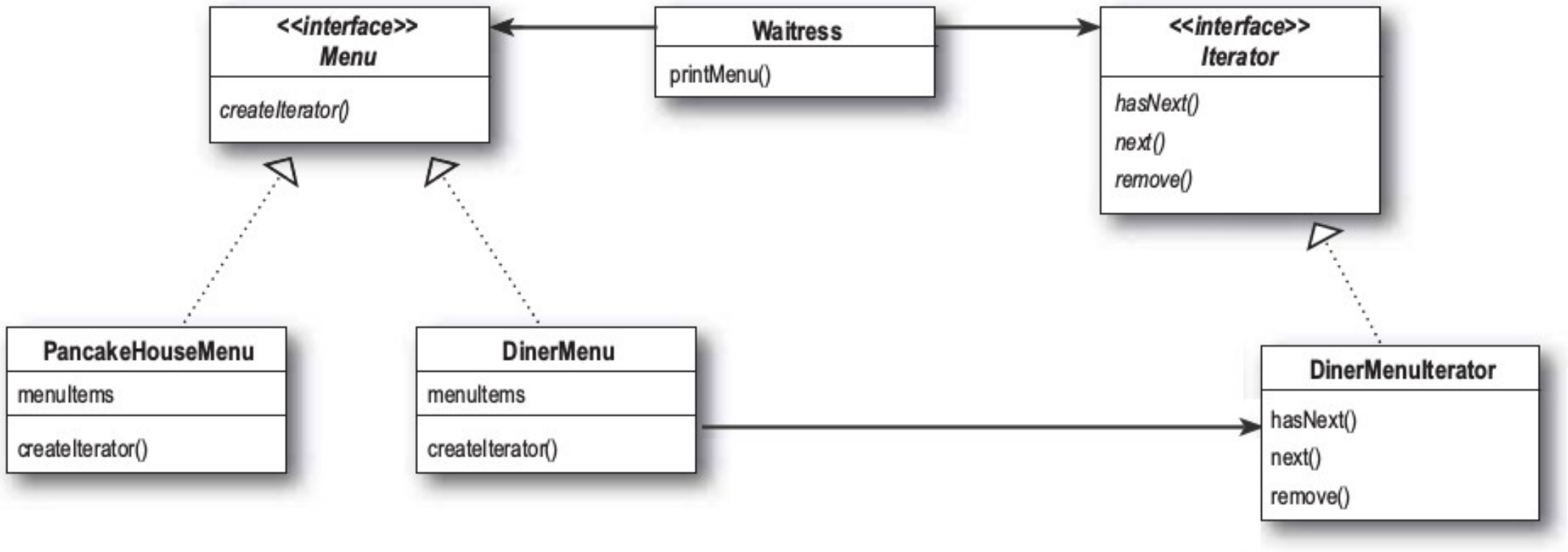
```
public class PancakeHouseMenu implements Menu {
    ArrayList<MenuItem> menuItems;

    public PancakeHouseMenu() {
        menuItems = new ArrayList<MenuItem>();

        addItem("K&B's Pancake Breakfast",
            "Pancakes with scrambled eggs, and toast",
            true,
            2.99);
    }

    public void addItem(String name, String description,
                        boolean vegetarian, double price){
        MenuItem menuItem = new MenuItem(name, description, vegetarian, price);
        menuItems.add(menuItem);
    }

    public Iterator createIterator() {
        return new PancakeHouseMenuIterator(menuItems);
    }

    // other menu methods here
}
```

```
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();
        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() + ", ");
            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();
    }
}
```

```
File Edit Window Help GreenEggs&Ham
% java DinerMenuTestDrive

MENU
----
BREAKFAST
K&B's Pancake Breakfast, 2.99 -- Pancakes with scrambled eggs, and toast
Regular Pancake Breakfast, 2.99 -- Pancakes with fried eggs, sausage
Blueberry Pancakes, 3.49 -- Pancakes made with fresh blueberries
Waffles, 3.59 -- Waffles, with your choice of blueberries or strawberries

LUNCH
Vegetarian BLT, 2.99 -- (Fakin') Bacon with lettuce & tomato on whole wheat
BLT, 2.99 -- Bacon with lettuce & tomato on whole wheat
Soup of the day, 3.29 -- Soup of the day, with a side of potato salad
Hotdog, 3.05 -- A hot dog, with saurkraut, relish, onions, topped with cheese
Steamed Veggies and Brown Rice, 3.99 -- Steamed vegetables over brown rice
Pasta, 3.89 -- Spaghetti with Marinara Sauce, and a slice of sourdough bread

%
```

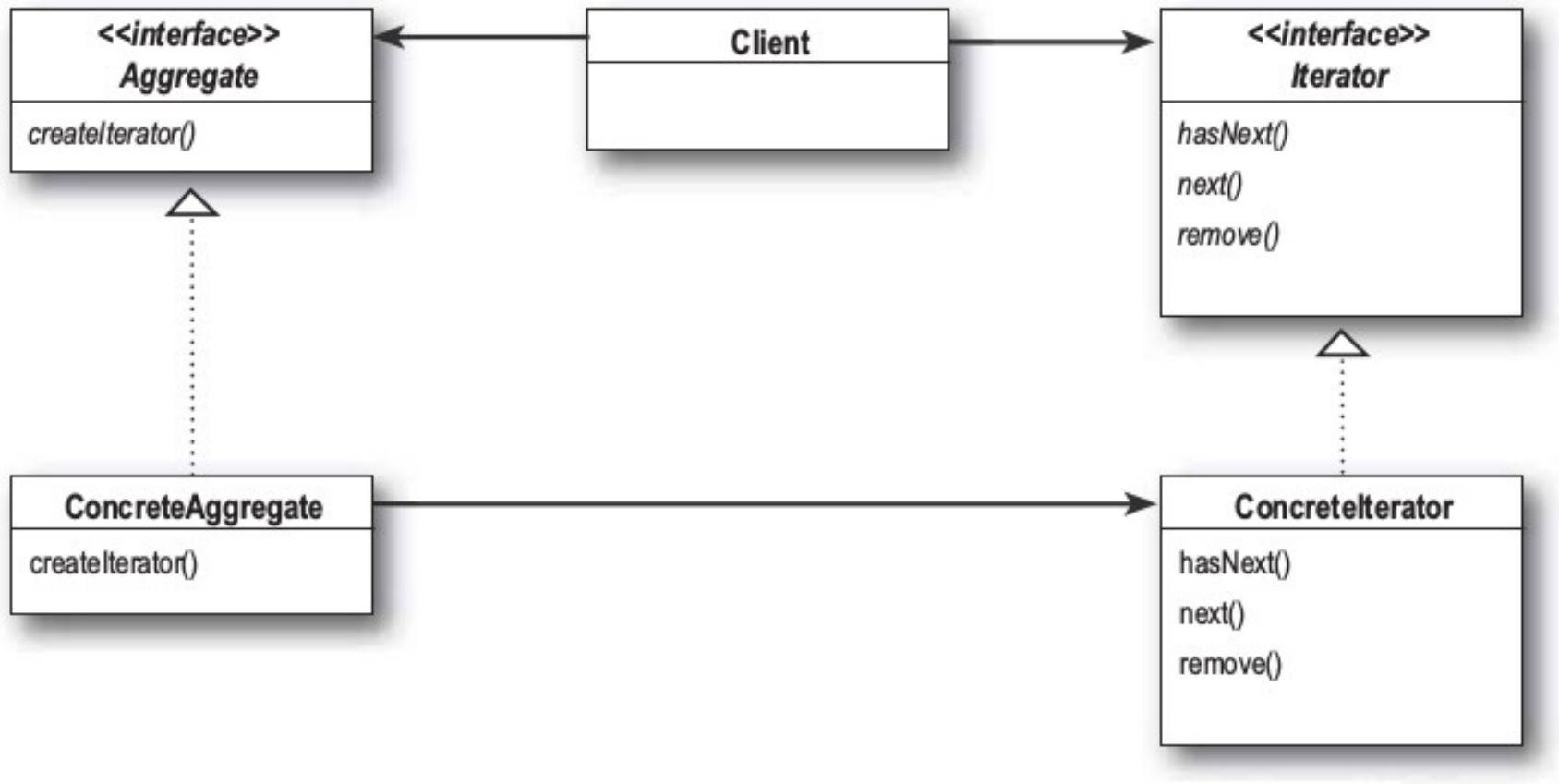
Diagrama de clases



El patrón iterator

El patrón “Iterator” provee una forma de acceder a los elementos de un objeto que los contiene de manera secuencial, sin exponer su representación interna.

Diagrama general del patrón iterator



Más, más cambios

Ahora una empresa más se unirá al monopolio,

¿Disney eres tú?

CafeMenu doesn't implement our new Menu interface, but this is easily fixed.

The Café is storing their menu items in a Hashtable. Does that support Iterator? We'll see shortly...

```
public class CafeMenu {
    Hashtable menuItems = new Hashtable();

    public CafeMenu() {
        addItem("Veggie Burger and Air Fries",
            "Veggie burger on a whole wheat bun, lettuce, tomato, and fries",
            true, 3.99);
        addItem("Soup of the day",
            "A cup of the soup of the day, with a side salad",
            false, 3.69);
        addItem("Burrito",
            "A large burrito, with whole pinto beans, salsa, guacamole",
            true, 4.29);
    }

    public void addItem(String name, String description,
                        boolean vegetarian, double price)
    {
        MenuItem menuItem = new MenuItem(name, description, vegetarian, price);
        menuItems.put(menuItem.getName(), menuItem);
    }

    public Hashtable getItems() {
        return menuItems;
    }
}
```

Like the other Menus, the menu items are initialized in the constructor.

Here's where we create a new MenuItem and add it to the menuItems hashtable.

the key is the item name.

the value is the menuItem object.

We're not going to need this anymore.

```
public class CafeMenu implements Menu {
    Hashtable menuItems = new Hashtable();

    public CafeMenu() {
        // constructor code here
    }

    public void addItem(String name, String description,
                        boolean vegetarian, double price)
    {
        MenuItem menuItem = new MenuItem(name, description, vegetarian, price);
        menuItems.put(menuItem.getName(), menuItem);
    }

    public Hashtable getItems() {
        return menuItems;
    }

    public Iterator createIterator() {
        return menuItems.values().iterator();
    }
}
```

CafeMenu implements the Menu interface, so the Waitress can use it just like the other two Menus.

We're using Hashtable because it's a common data structure for storing values; you could also use the newer HashMap.

Just like before, we can get rid of getItems() so we don't expose the implementation of menuItems to the Waitress.

And here's where we implement the createIterator() method. Notice that we're not getting an Iterator for the whole Hashtable, just for the values.

La implementación del iterador de Hashtable se deja al lector

```
public class Waitress {
    Menu pancakeHouseMenu;
    Menu dinerMenu;
    Menu cafeMenu;

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

    public void printMenu() {
        Iterator pancakeIterator = pancakeHouseMenu.createIterator();
        Iterator dinerIterator = dinerMenu.createIterator();
        Iterator cafeIterator = cafeMenu.createIterator();
        System.out.println("MENU\n----\nBREAKFAST");
        printMenu(pancakeIterator);
        System.out.println("\nLUNCH");
        printMenu(dinerIterator);
        System.out.println("\nDINNER");
        printMenu(cafeIterator);
    }

    private 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());
        }
    }
}
```

The Café menu is passed into the Waitress in the constructor with the other menus, and we stash it in an instance variable.

We're using the Café's menu for our dinner menu. All we have to do to print it is create the iterator, and pass it to printMenu(). That's it!

Nothing changes here

```
public class MenuTestDrive {
    public static void main(String args[]) {
        PancakeHouseMenu pancakeHouseMenu = new PancakeHouseMenu();
        DinerMenu dinerMenu = new DinerMenu();
        CafeMenu cafeMenu = new CafeMenu();

        Waitress waitress = new Waitress(pancakeHouseMenu, dinerMenu, cafeMenu);

        waitress.printMenu();
}
```

Create a CafeMenu...

... and pass it to the waitress.

Now, when we print we should see all three menus.

```
File Edit Window Help Kathy&BertLikePancakes
% java DinerMenuTestDrive

MENU
----
BREAKFAST
K&B's Pancake Breakfast, 2.99 -- Pancakes with scrambled eggs, and toast
Regular Pancake Breakfast, 2.99 -- Pancakes with fried eggs, sausage
Blueberry Pancakes, 3.49 -- Pancakes made with fresh blueberries
Waffles, 3.59 -- Waffles, with your choice of blueberries or strawberries

LUNCH
Vegetarian BLT, 2.99 -- (Fakin') Bacon with lettuce & tomato on whole wheat
BLT, 2.99 -- Bacon with lettuce & tomato on whole wheat
Soup of the day, 3.29 -- Soup of the day, with a side of potato salad
Hotdog, 3.05 -- A hot dog, with saurkraut, relish, onions, topped with cheese
Steamed Veggies and Brown Rice, 3.99 -- Steamed vegetables over brown rice
Pasta, 3.89 -- Spaghetti with Marinara Sauce, and a slice of sourdough bread

DINNER
Soup of the day, 3.69 -- A cup of the soup of the day, with a side salad
Burrito, 4.29 -- A large burrito, with whole pinto beans, salsa, guacamole
Veggie Burger and Air Fries, 3.99 -- Veggie burger on a whole wheat bun,
 lettuce, tomato, and fries
%
```

First we iterate through the pancake menu.

And then the diner menu.

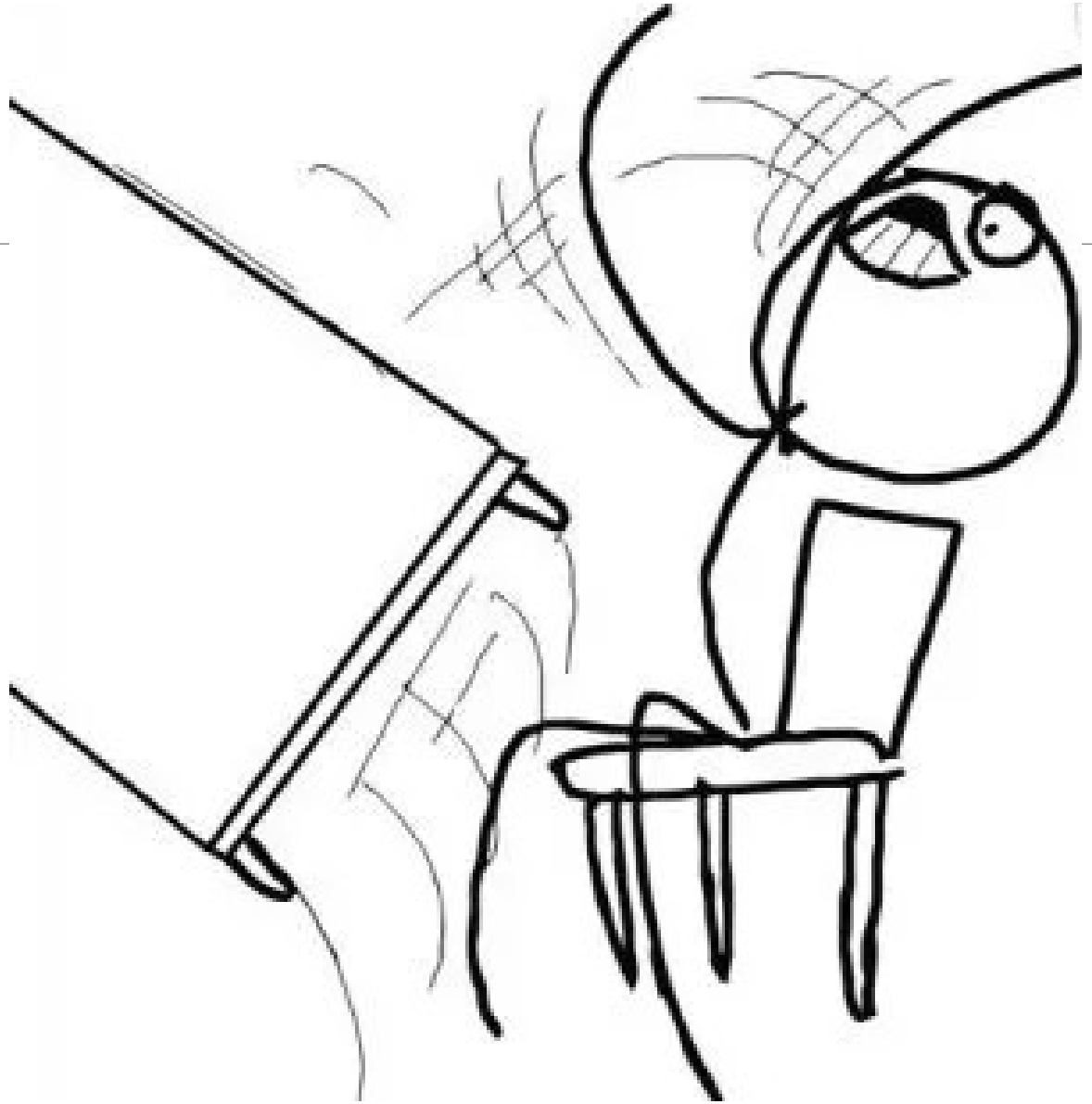
And finally the new café menu, all with the same iteration code.

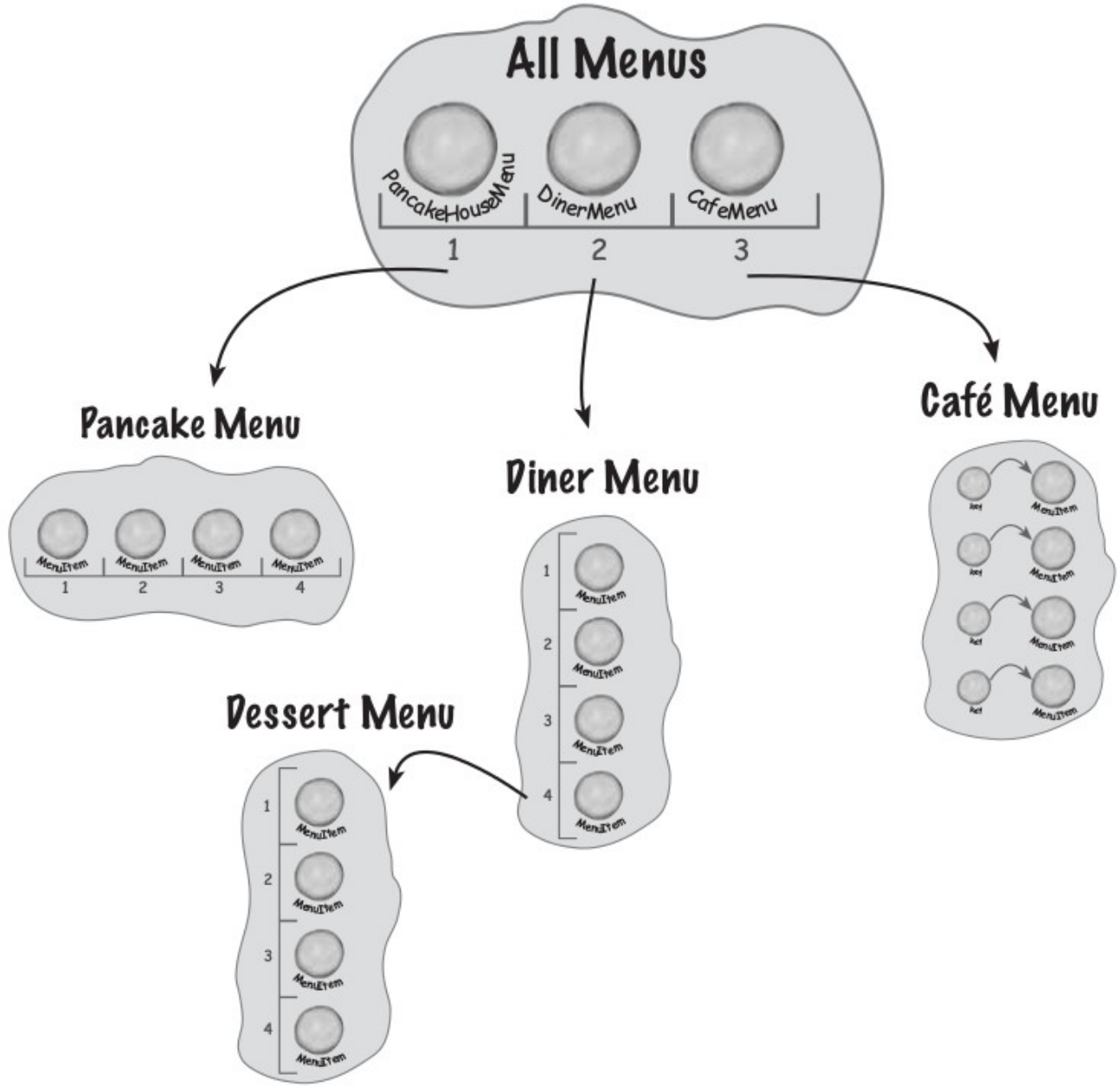
Esta historia continuara ...

Todavía más cambios

Nuestros eternos inconforme clientes ahora quieren agregar un submenú de postres.

De acuerdo, ¿y ahora qué? Ahora tenemos que admitir no solo múltiples menús, sino también menús dentro de los menús.



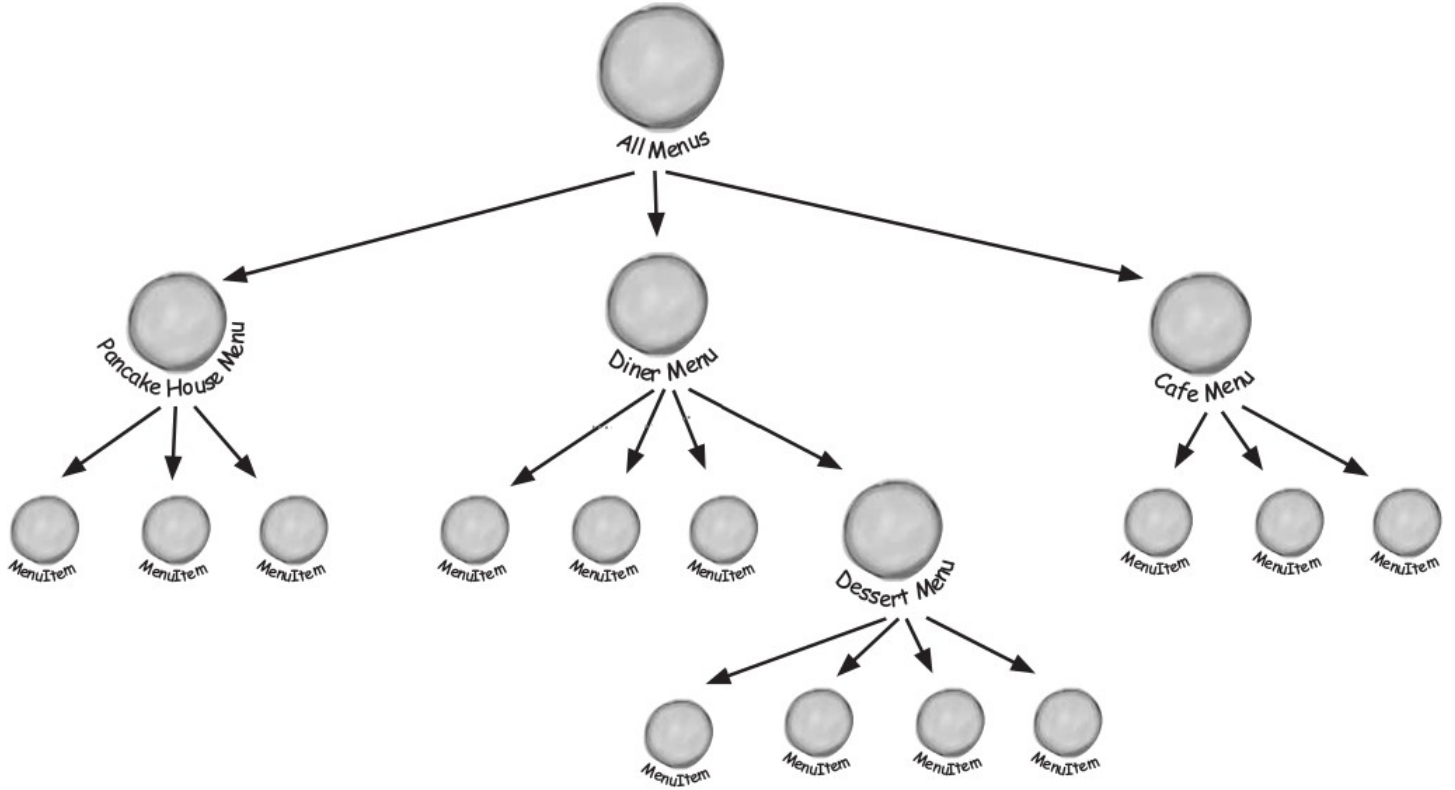

All Menus
PancakeHouseMenu
DinerMenu
CafeMenu
1
2
3
Pancake Menu
MenuItem
1
2
3
4
Diner Menu
MenuItem
1
2
3
4
Café Menu
MenuItem
Dessert Menu
MenuItem
1
2
3
4

Tiempo de re implementar sus Menús.

La realidad es que hemos alcanzado un nivel de complejidad de modo que si no rehacemos el rediseño ahora, nunca va a tener un diseño que pueda acomodar otras adquisiciones o submenús.

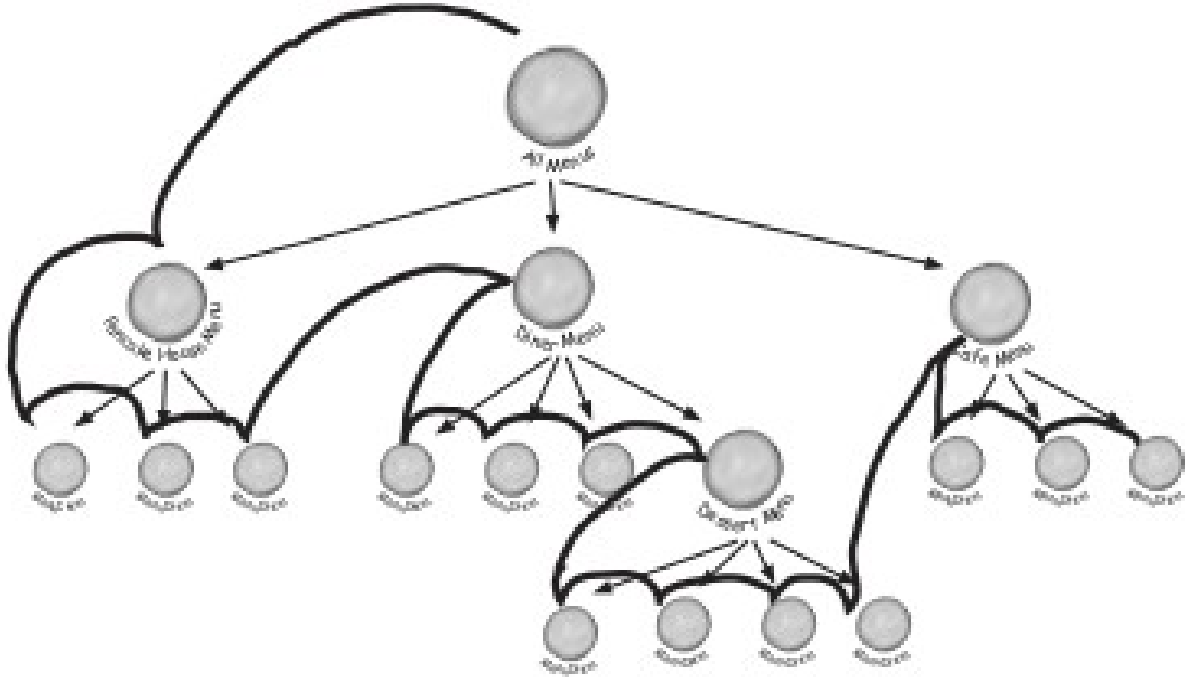
¿Qué necesitamos?

- Necesitamos un tipo de estructura en forma de árbol que nos permita acomodar menús, submenús e ítems de menús.



¿Qué necesitamos?

- Debemos asegurarnos de mantener una forma de recorrer los elementos en cada menú que sea al menos tan conveniente como lo que estamos haciendo ahora con los iteradores.



¿Qué necesitamos?

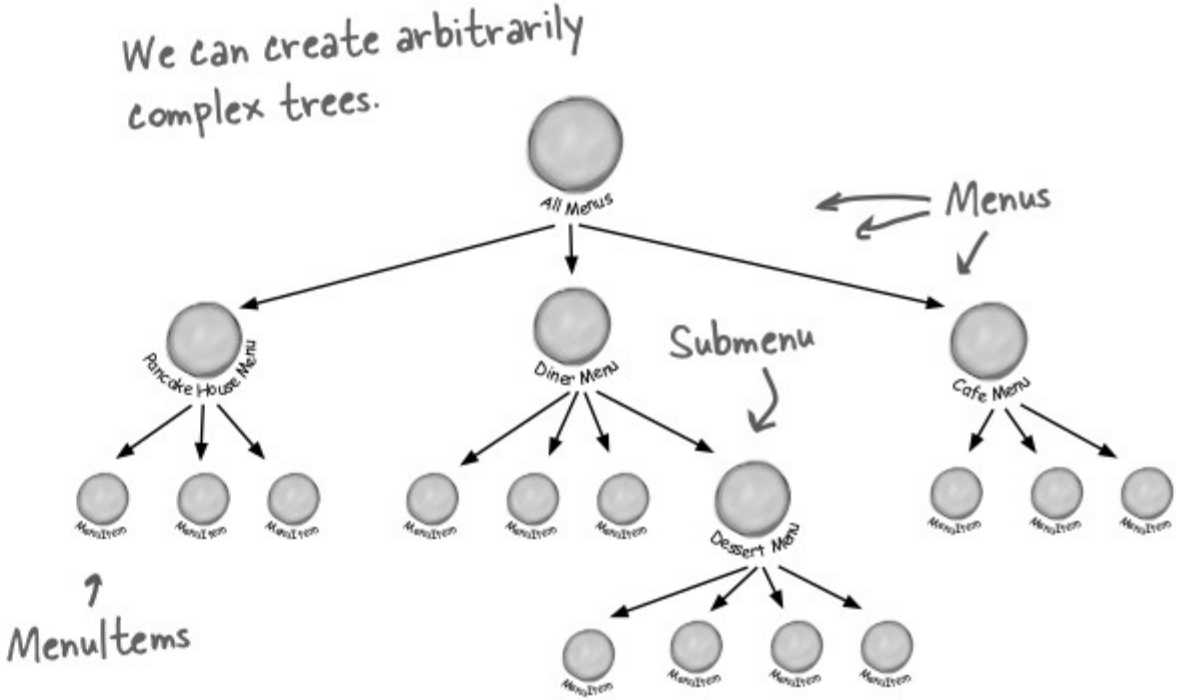
- Es posible que necesitemos poder atravesar las comidas de una manera más flexible. Por ejemplo, tal vez tengamos que iterar solo en el menú de postres del Diner, o podríamos necesitar repetir el menú completo del Diner, incluido el submenú del postre.

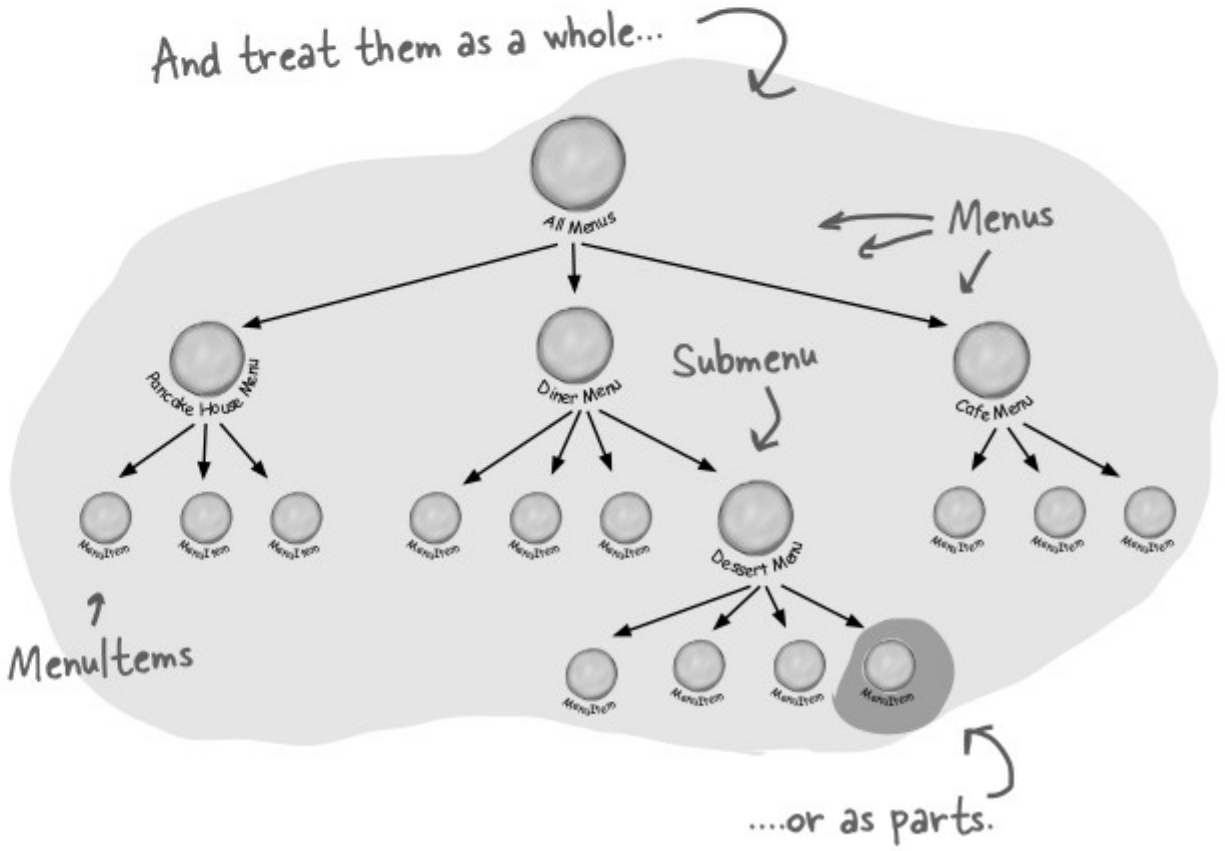
¿Y como lo traducimos al problema?

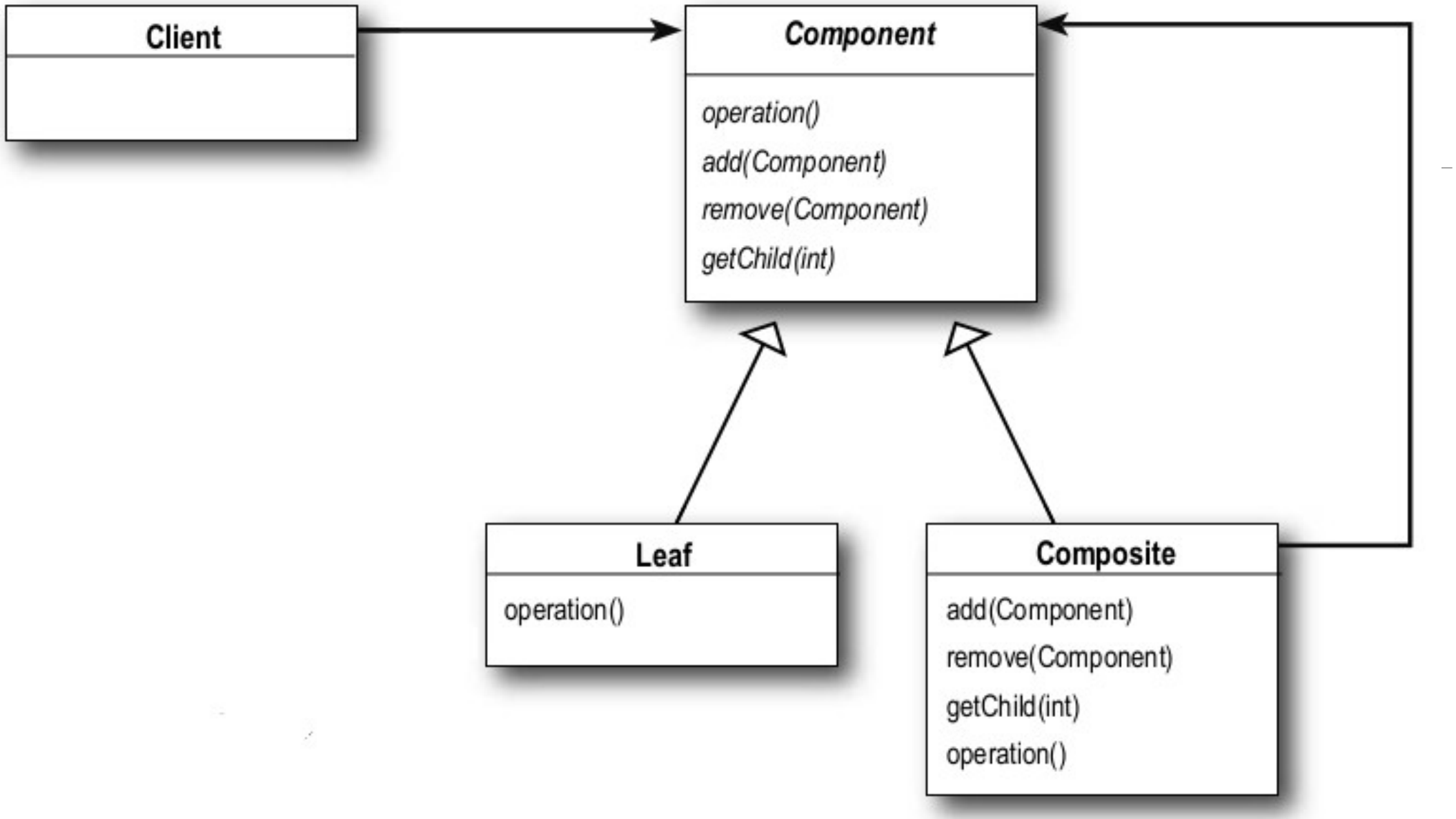
Para poder estructurar los menús como un árbol cada “nodo” del árbol debe poder tener como hijos ya sea más nodos u hojas, en este caso debe poder tener objetos del tipo Menú o del tipo MenúItem, por lo que deben heredar de la misma clase.

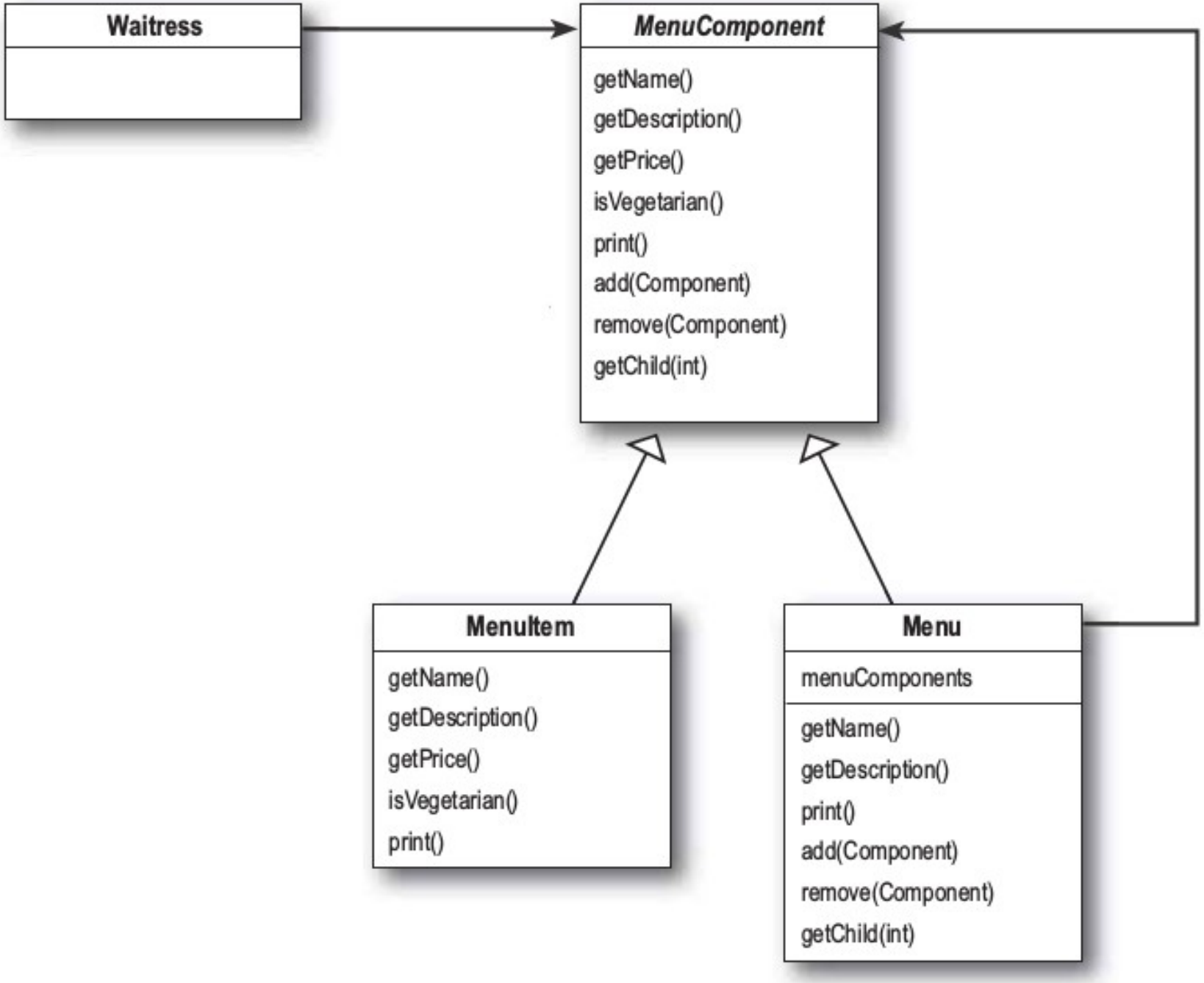
El patrón Composite.

El patrón de diseño “Composite” te permite componer objetos en estructura de árbol para representar jerarquías, Composite deja al cliente tratar objetos individuales y composiciones de objetos uniformemente.


We can create arbitrarily complex trees.
All Menus
Menus
Pancake House Menu
Diner Menu
Submenu
Cafe Menu
Dessert Menu
MenuItem
MenuItems


And treat them as a whole...
All Menus
Menus
Pancake House Menu
Diner Menu
Submenu
Cafe Menu
Dessert Menu
MenuItem
MenuItems
....or as parts.


Client
Component
operation()
add(Component)
remove(Component)
getChild(int)
Leaf
operation()
Composite
add(Component)
remove(Component)
getChild(int)
operation()


Waitress
MenuComponent
getName()
getDescription()
getPrice()
isVegetarian()
print()
add(Component)
remove(Component)
getChild(int)
MenuItem
getName()
getDescription()
getPrice()
isVegetarian()
print()
Menu
menuComponents
getName()
getDescription()
print()
add(Component)
remove(Component)
getChild(int)

```
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();
    }

    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 String getName() {
        return name;
    }

    public String getDescription() {
        return description;
    }
```

```
    public double getPrice() {
        return price;
    }

    public boolean isVegetarian() {
        return vegetarian;
    }

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

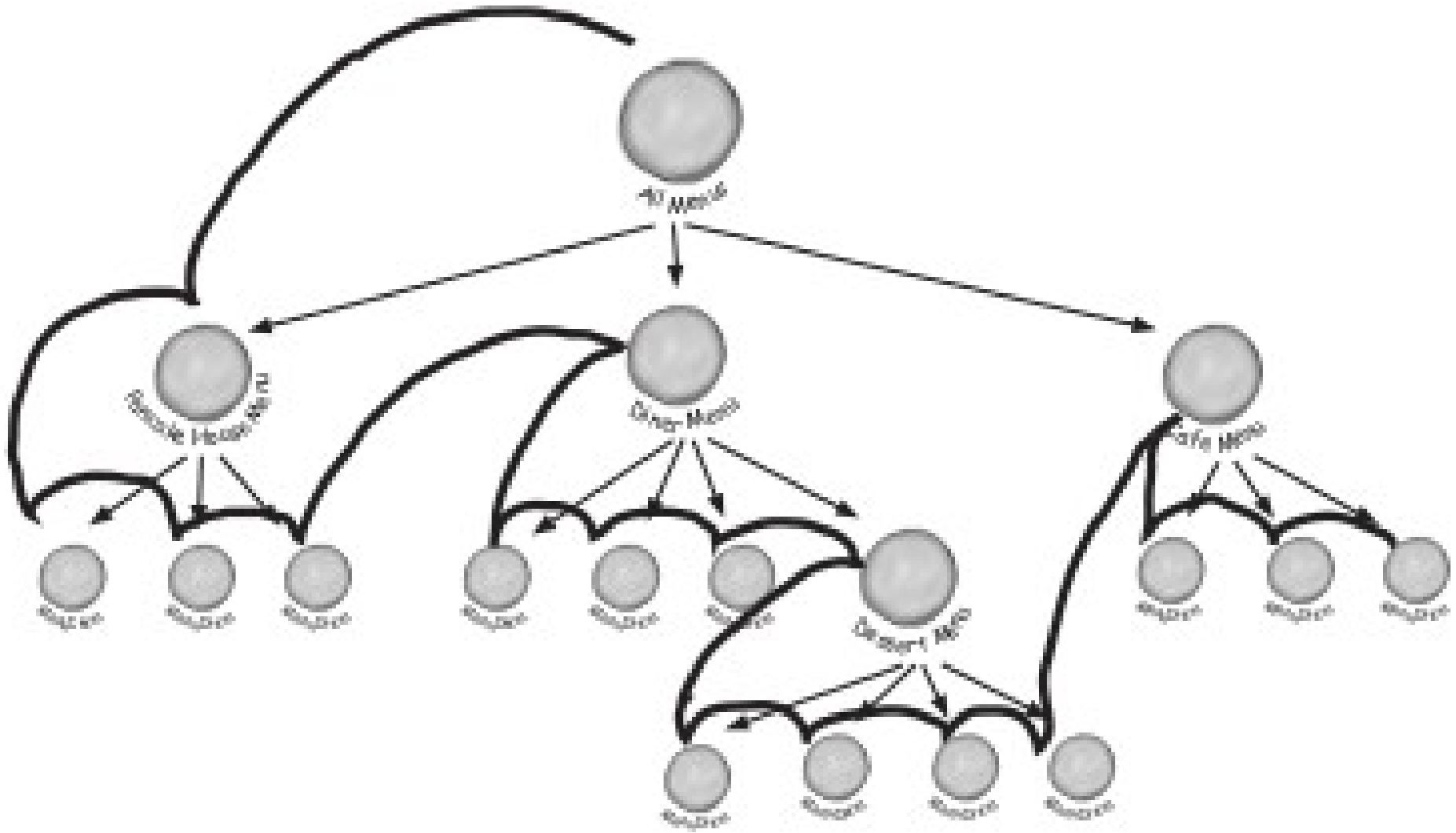
```
public class Menu extends MenuComponent {
    ArrayList menuComponents = new ArrayList();
    String name;
    String description;

    public Menu(String name, String description) {
        this.name = name;
        this.description = description;
    }

    public void add(MenuComponent menuComponent) {
        menuComponents.add(menuComponent);
    }

    public void remove(MenuComponent menuComponent) {
        menuComponents.remove(menuComponent);
    }

    public MenuComponent getChild(int i) {
        return (MenuComponent)menuComponents.get(i);
    }

    public String getName() {
        return name;
    }

    public String getDescription() {
        return description;
    }
```

```
public class Menu extends MenuComponent {
    ArrayList menuComponents = new ArrayList();
    String name;
    String description;

    // constructor code here

    // other methods here

    public void print() {
        System.out.print("\n" + getName());
        System.out.println(", " + getDescription());
        System.out.println("---------------------");

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

All we need to do is change the print() method to make it print not only the information about this Menu, but all of this Menu's components: other Menus and MenuItems.

Look! We get to use an Iterator. We use it to iterate through all the Menu's components... those could be other Menus, or they could be MenuItems. Since both Menus and MenuItems implement print(), we just call print() and the rest is up to them.

NOTE: If, during this iteration, we encounter another Menu object, its print() method will start another iteration, and so on.



```
public class Waitress {
    MenuComponent allMenus;

    public Waitress(MenuComponent allMenus) {
        this.allMenus = allMenus;
    }

    public void printMenu() {
        allMenus.print();
    }
}
```

Yup! The Waitress code really is this simple. Now we just hand her the top level menu component, the one that contains all the other menus. We've called that allMenus.

All she has to do to print the entire menu hierarchy – all the menus, and all the menu items – is call print() on the top level menu.

We're gonna have one happy Waitress.

```
public class MenuTestDrive {
    public static void main(String args[]) {
        MenuComponent pancakeHouseMenu =
            new Menu("PANCAKE HOUSE MENU", "Breakfast");
        MenuComponent dinerMenu =
            new Menu("DINER MENU", "Lunch");
        MenuComponent cafeMenu =
            new Menu("CAFE MENU", "Dinner");
        MenuComponent dessertMenu =
            new Menu("DESSERT MENU", "Dessert of course!");

        MenuComponent allMenus = new Menu("ALL MENUS", "All menus combined");

        allMenus.add(pancakeHouseMenu);
        allMenus.add(dinerMenu);
        allMenus.add(cafeMenu);

        // add menu items here

        dinerMenu.add(new MenuItem(
            "Pasta",
```

Let's first create all the menu objects.

We also need two top level menu now that we'll name allMenus.

We're using the Composite add() method to add each menu to the top level menu, allMenus.

Now we need to add all the menu items, here's one example, for the rest, look at the complete source code.

```
        dinerMenu.add(new MenuItem(
            "Pasta",
            "Spaghetti with Marinara Sauce, and a slice of sourdough bread",
            true,
            3.89));

        dinerMenu.add(dessertMenu);

        dessertMenu.add(new MenuItem(
            "Apple Pie",
            "Apple pie with a flakey crust, topped with vanilla icecream",
            true,
            1.59));

        // add more menu items here

        Waitress waitress = new Waitress(allMenus);

        waitress.printMenu();
    }
}
```

at the complete source code.

And we're also adding a menu to a menu. All dinerMenu cares about is that everything it holds, whether it's a menu item or a menu, is a MenuComponent.

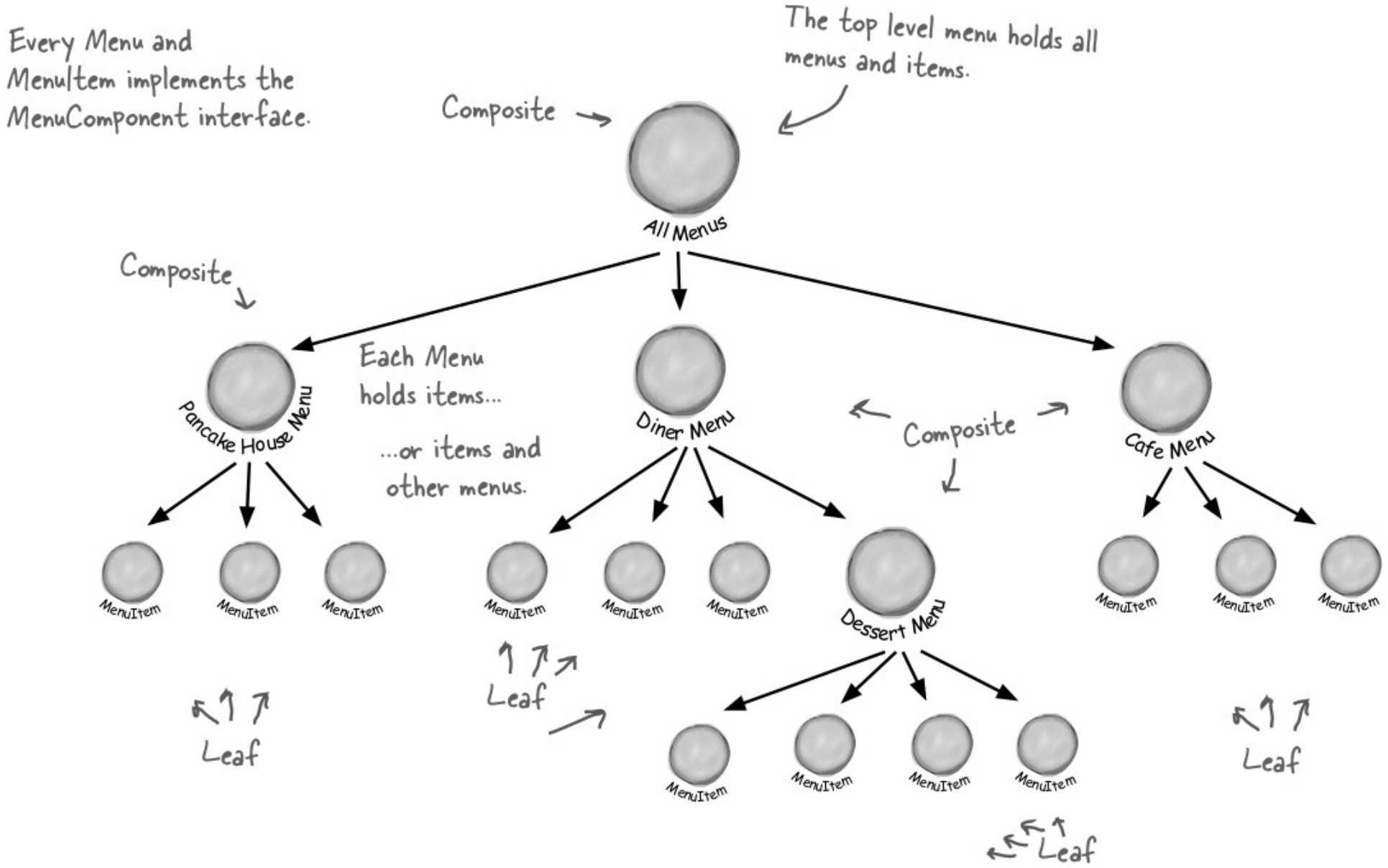
Add some apple pie to the dessert menu...

Once we've constructed our entire menu hierarchy, we hand the whole thing to the Waitress, and as you've seen, it's easy as apple pie for her to print it out.

```
File Edit Window Help GreenEggs&Spam
% java MenuTestDrive

ALL MENUS, All menus combined
---------------------

PANCAKE HOUSE MENU, Breakfast
---------------------
  K&B's Pancake Breakfast(v), 2.99
     -- Pancakes with scrambled eggs, and toast
  Regular Pancake Breakfast, 2.99
     -- Pancakes with fried eggs, sausage
  Blueberry Pancakes(v), 3.49
     -- Pancakes made with fresh blueberries, and blueberry syrup
  Waffles(v), 3.59
     -- Waffles, with your choice of blueberries or strawberries

DINER MENU, Lunch
---------------------
  Vegetarian BLT(v), 2.99
     -- (Fakin') Bacon with lettuce & tomato on whole wheat
  BLT, 2.99
     -- Bacon with lettuce & tomato on whole wheat
  Soup of the day, 3.29
     -- A bowl of the soup of the day, with a side of potato salad
  Hotdog, 3.05
     -- A hot dog, with saurkraut, relish, onions, topped with cheese
  Steamed Veggies and Brown Rice(v), 3.99
     -- Steamed vegetables over brown rice
  Pasta(v), 3.89
     -- Spaghetti with Marinara Sauce, and a slice of sourdough bread

DESSERT MENU, Dessert of course!
---------------------
  Apple Pie(v), 1.59
     -- Apple pie with a flakey crust, topped with vanilla icecream
  Cheesecake(v), 1.99
     -- Creamy New York cheesecake, with a chocolate graham crust
  Sorbet(v), 1.89
     -- A scoop of raspberry and a scoop of lime

CAFE MENU, Dinner
---------------------
  Veggie Burger and Air Fries(v), 3.99
     -- Veggie burger on a whole wheat bun, lettuce, tomato, and fries
  Soup of the day, 3.69
     -- A cup of the soup of the day, with a side salad
  Burrito(v), 4.29
     -- A large burrito, with whole pinto beans, salsa, guacamole
%
```

Here's all our menus... we printed all this just by calling print() on the top level menu

The new dessert menu is printed when we are printing all the Diner menu components


Every Menu and MenuItem implements the MenuComponent interface.
The top level menu holds all menus and items.
Composite
All Menus
Composite
Each Menu holds items...
...or items and other menus.
Pancake House Menu
Diner Menu
Composite
Cafe Menu
MenuItem
MenuItem
MenuItem
MenuItem
MenuItem
MenuItem
Dessert Menu
MenuItem
MenuItem
MenuItem
MenuItem
MenuItem
MenuItem
MenuItem
Leaf
Leaf
Leaf
Leaf

¿Con esto basta?

Con este código podemos organizar los menús en una estructura en forma de árbol, pero aun no se puede iterar, necesitamos agregar el iterador.

```
public class Menu extends MenuComponent {

    // other code here doesn't change

    public Iterator createIterator() {
        return new CompositeIterator(menuComponents.iterator());
    }

}
```

```
public class MenuItem extends MenuComponent {

    // other code here doesn't change

    public Iterator createIterator() {
        return new NullIterator();
    }

}
```

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

    public CompositeIterator(Iterator iterator) {
        stack.push(iterator);
    }

    public Object next() {
        if (hasNext()) {
            Iterator iterator = (Iterator) stack.peek();
            MenuComponent component = (MenuComponent) iterator.next();
            if (component instanceof Menu) {
                stack.push(component.createIterator());
            }
            return component;
        } else {
            return null;
        }
    }
```

```
    public boolean hasNext() {
        if (stack.empty()) {
            return false;
        } else {
            Iterator iterator = (Iterator) stack.peek();
            if (!iterator.hasNext()) {
                stack.pop();
                return hasNext();
            } else {
                return true;
            }
        }
    }

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

El iterador nulo

Bien, ¿ahora de qué se trata este Iterador Nulo? Piénsalo de esta manera: un MenuItem no tiene nada que repetir, ¿verdad? Entonces, ¿cómo manejamos la implementación de su método createIterator()? Bueno, tenemos dos opciones:

Primera elección

Return null

Podríamos devolver null desde createIterator(), pero luego necesita código condicional en el cliente para ver si nulo era devuelto o no

Segunda elección

Devuelve un iterador que siempre devuelve falso cuando se llama a hasNext() Esto parece ser un mejor plan. Todavía podemos devolver un iterador, pero el cliente no tiene que preocuparse por si se devuelve o no el valor nulo

```
import java.util.Iterator;

public class NullIterator implements Iterator {

    public Object next() {
        return null;
    }

    public boolean hasNext() {
        return false;
    }

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

¿Y si solo quiero ve el menú vegetariano?

```
public class Waitress {
    MenuComponent allMenus;

    public Waitress(MenuComponent allMenus) {
        this.allMenus = allMenus;
    }

    public void printMenu() {
        allMenus.print();
    }

    public void printVegetarianMenu() {
        Iterator iterator = allMenus.createIterator();
        System.out.println("\nVEGETARIAN MENU\n----");
        while (iterator.hasNext()) {
            MenuComponent menuComponent =
                    (MenuComponent)iterator.next();
            try {
                if (menuComponent.isVegetarian()) {
                    menuComponent.print();
                }
            } catch (UnsupportedOperationException e) {}
        }
    }
}
```

The printVegetarianMenu() method takes the allMenu's composite and gets its iterator. That will be our CompositeIterator.

Iterate through every element of the composite.

Call each element's isVegetarian() method and if true, we call its print() method.

print() is only called on MenuItems, never composites. Can you see why?

We implemented isVegetarian() on the Menus to always throw an exception. If that happens we catch the exception, but continue with our iteration.

El patrón Composite nos permite construir estructuras de objetos en forma de árboles que contienen tanto composiciones de objetos como objetos individuales como nodos.

Al usar una estructura compuesta, podemos aplicar las mismas operaciones sobre los compuestos y los objetos individuales. En otras palabras, en la mayoría de los casos podemos ignorar las diferencias entre composiciones de objetos y objetos individuales.