Table of Contents(summary)

Iterator and Composite

There is a lot of ways to stuff object into a collection. Put them in an Array, a Stack, a List, a HashTable, etc. Each has it own advantage and tradeoffs. But at some point your client is going to want to iterate over those objects, and when he does, are you going to show him your implementation ?

Iterator and Composite(example)

- We have a great news ! We are going to merge ObjectVille Diner and ObjectVille Pancake. But there seems to be a slight problem.

Iterator and Composite(example)

```
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;
    }
}
```

A MenuItem consists of a name, a description, a flag to indicate if the item is vegetarian, and a price. You pass all these values into the constructor to initialize the MenuItem.

These getter methods let you access the fields of the menu item.

Iterator and Composite(example)

```
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);

        addItem("Blueberry Pancakes",
            "Pancakes made with fresh blueberries",
            true,
            3.49);

        addItem("Waffles",
            "Waffles, with your choice of blueberries or strawberries",
            true,
            3.59);
    }
    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
}
```

Lou's using an ArrayList to store his menu items

Each menu item is added to the ArrayList here, in the constructor

Each MenuItem has a name, a description, whether or not it's a vegetarian item, and the price

To add a menu item, Lou creates a new MenuItem object, passing in each argument, and then adds it to the ArrayList

The getMenuItems() method returns the list of menu items

Lou has a bunch of other menu code that depends on the ArrayList implementation. He doesn't want to have to rewrite all that code!

Iterator and Composite(example)

```
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, 2.99);
        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);
        addItem("Hotdog",
            "A hot dog, with saurkraut, relish, onions, topped with cheese",
            false, 3.05);
        // a couple of other Diner Menu items added here
    }

    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;
    }

    // other menu methods here
}
```

Mel takes a different approach; he's using an Array so he can control the max size of the menu and retrieve menu items out without having to cast his objects.

Like Lou, Mel creates his menu items in the constructor, using the addItem() helper method.

addItem() takes all the parameters necessary to create a MenuItem and instantiates one. It also checks to make sure we haven't hit the menu size limit.

Mel specifically wants to keep his menu under a certain size (presumably so he doesn't have to remember too many recipes).

getMenuItems() returns the array of menu items.

Like Lou, Mel has a bunch of code that depends on the implementation of his menu being an Array. He's too busy cooking to rewrite all of this.

Iteration and Composite Pattern

- What is the problem with having two different menu representation ? Well let ‘s try to implement a simple client that use both implementation.
- It should:

  - printMenu() Print all every item on the menu

  - printBreakfastMenu() Print just breakfast items

  - printLunchMenu() print just lunch items

  - printVegetarianMenu() print all vegetarian menu items

  - isItemVegetarian(name) given the name of an item return true if the item is vegetarian, otherwise, return false.

Iteration and Composite Pattern

1. To print all the items on each menu, you'll need to call the getMenuItem() method on the PancakeHouseMenu and the DinerMenu to retrieve their respective menu items. Note that each returns a different type:

```
PancakeHouseMenu pancakeHouseMenu = new PancakeHouseMenu();
ArrayList breakfastItems = pancakeHouseMenu.getMenuItems();

DinerMenu dinerMenu = new DinerMenu();
MenuItem[] lunchItems = dinerMenu.getMenuItems();
```

The method looks the same, but the calls are returning different types.

The implementation is showing through, breakfast items are in an ArrayList, lunch items are in an Array.

2. Now, to print out the items from the PancakeHouseMenu, we'll loop through the items on the breakfastItems ArrayList. And to print out the Diner items we'll loop through the Array.

```
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());
}
```

Now, we have to implement two different loops to step through the two implementations of the menu items...

...one loop for the ArrayList...

and another for the Array.

Iteration and Composite Pattern

❸ Implementing every other method in the Waitress is going to be a variation of this theme. We're always going to need to get both menus and use two loops to iterate through their items. If another restaurant with a different implementation is acquired then we'll have *three* loops.

- Now the problems are that neither of both want to change their implementation since that will requiere rewritten a lot of code.
- Now if neither of both change their implementation the client will be hard to maintain and extend.
- It should be easy if both implement the same interface for theirs menu.

Iteration and Composite Pattern

- One of the thing we have already learned is encapsulate what varies. What is changing here is the iteration caused by different collections of objects being returned from the menu.

Iteration and Composite Pattern

Now what if we create an object, let's call it an Iterator, that encapsulates the way we iterate through a collection of objects? Let's try this on the ArrayList

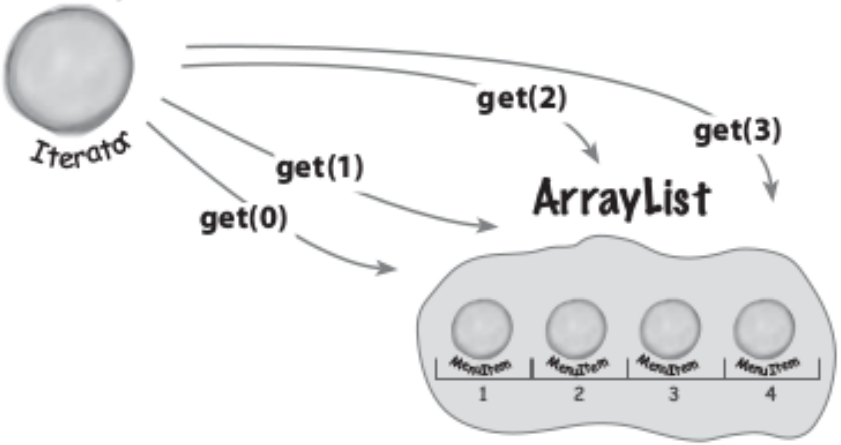
We ask the breakfastMenu for an iterator of its MenuItems.

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

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

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

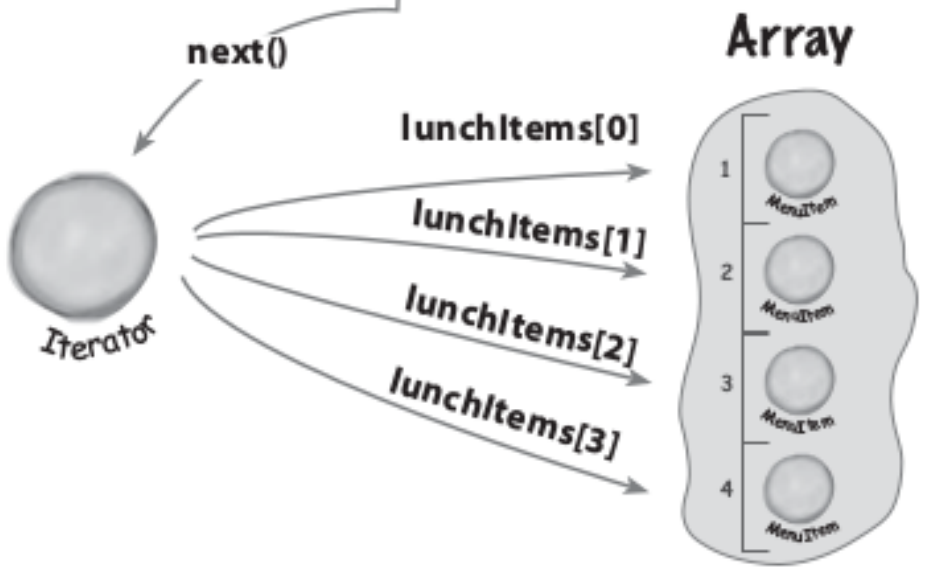
Iteration and Composite Pattern

Let's try that on the Array too:

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

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

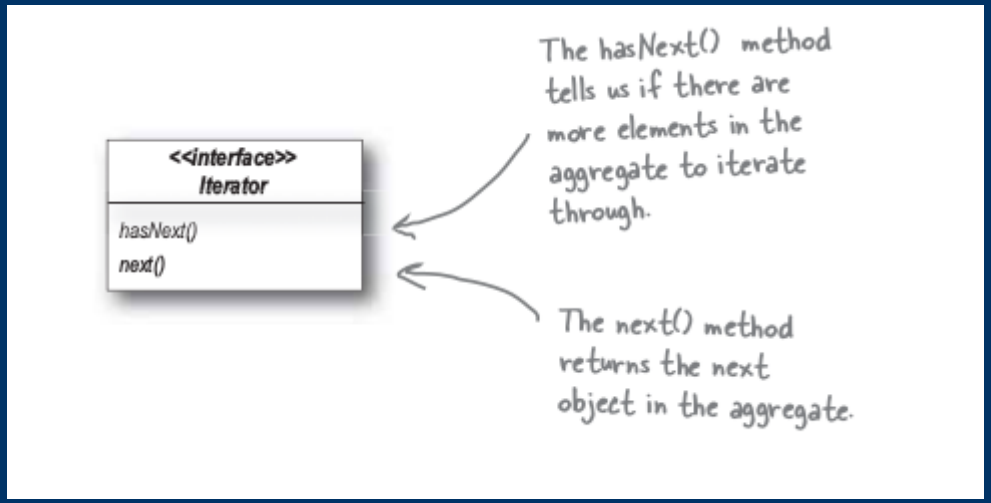
Wow, this code is exactly the same as the breakfastMenu code.

Same situation here: the client just calls hasNext() and next(); behind the scenes, the iterator indexes into the Array.



Iteration and Composite

- Well it looks like our plan is encapsulatin iteration that might actually work. And yes is Design pattern called the Iteration Pattern.
- First thing to know is that it relies on an interface called Iterator. Here is one possible Iterator interface.



Iteration (example implementation)

- Adding an iterator to the dinerMenu.

To add an Iterator to the DinerMenu we first need to define the Iterator Interface:

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

Here's our two methods:

The hasNext() method returns a boolean indicating whether or not there are more elements to iterate over...

...and the next() method returns the next element

Iteration (example implementation)

And now we need to implement a concrete Iterator that works for the Diner menu:

```
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;
        }
    }
}
```

We implement the Iterator interface.

position maintains the current position of the iteration over the array.

The constructor takes the array of menu items we are going to iterate over.

The next() method returns the next item in the array and increments the position.

The hasNext() method checks to see if we've seen all the elements of the array and returns true if there are more to iterate through.

Because the diner chef went ahead and allocated a max sized array, we need to check not only if we are at the end of the array, but also if the next item is null, which indicates there are no more items.

Iteration (example implementation)

```
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
}
```

We're not going to need the getMenuItems() method anymore and in fact, we don't want it because it exposes our internal implementation!

Here's the createIterator() method. It creates a DinerMenuIterator from the menuItems array and returns it to the client.

We're returning the Iterator interface. The client doesn't need to know how the menuItems are maintained in the DinerMenu, nor does it need to know how the DinerMenuIterator is implemented. It just needs to use the iterators to step through the items in the menu.

Iteration (example implementation)

- Fixing our client(waitress code) we need to integrate the iterator code into the client.

```
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
}
```

In the constructor the Waitress takes the two menus.

The printMenu() method now creates two iterators, one for each menu.

And then calls the overloaded printMenu() with each iterator.

Test if there are any more items.

Get the next item.

Use the item to get name, price and description and print them.

The overloaded printMenu() method uses the Iterator to step through the menu items and print them.

Note that we're down to one loop.

Iterator Patter

- What we did so far ?

  Encapsulate code. The client(waitress) has no idea how the menus hold their collection of menu items.

  Client use an interface Iterator.

  We have a loop that polymorfically handles any collection of items as long as it implements Iterator.

- What else could be improved ?

Iterator(example)

```
public interface Menu {
    public Iterator createIterator();
}
```

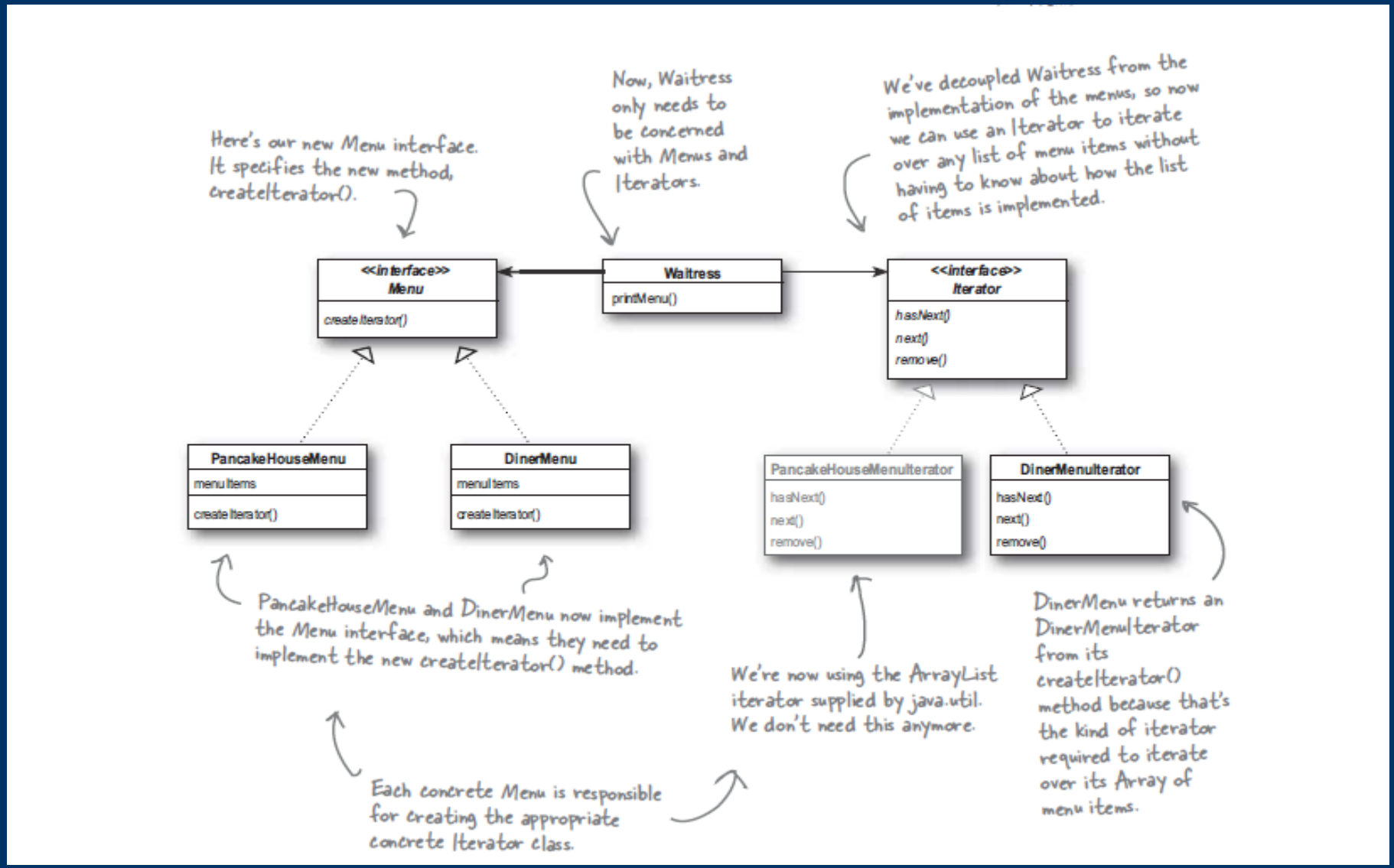
This is a simple interface that just lets clients get an iterator for the items in the menu.

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

    public Waitress(Menu pancakeHouseMenu, Menu 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
}
```

We need to replace the concrete Menu classes with the Menu Interface.

Nothing changes here.

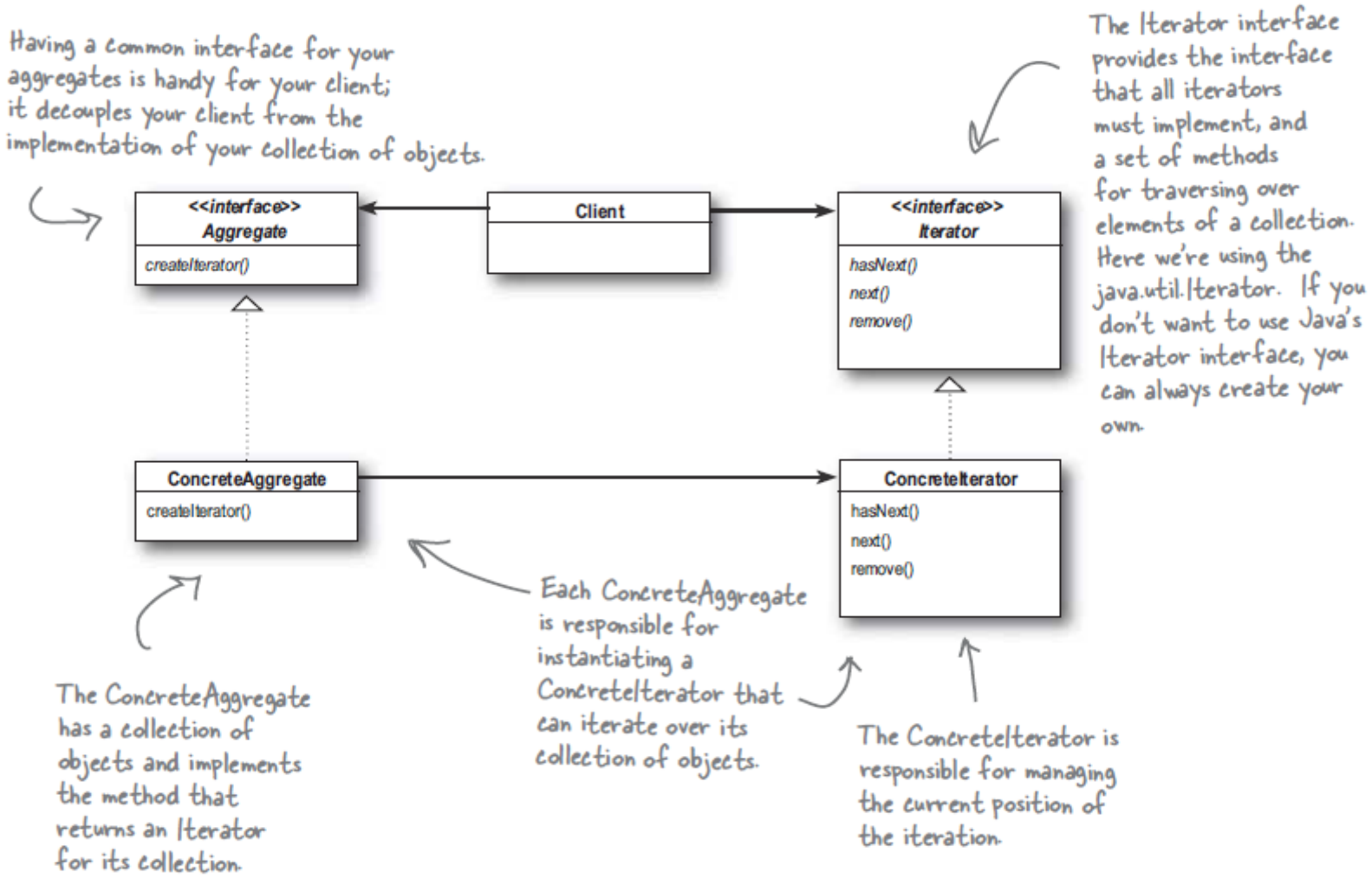
Iterator(diagram)



Iterator Pattern(definition)

- (Definition)Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.
- Allow traversal of the elements without exposing the underlying implementation.

Iterator(class diagram)



Iterator and Composite Pattern

- Things go better and ObjectVille has another adquisition Objectville Cafe and adopting their dinner menu.

Iterator and Composite Pattern

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

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

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

```
    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);
    }
```

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

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

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.

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

We're not going to need this anymore.

Iterator and Composite Pattern

- Is your time to integrate Cafe menu in our code.

Iterator and Composite Pattern

```
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.

Iterator and Composite Pattern

- Is your time to integrate Cafe menu to the client(Waitress).