Chapter 9. The Iterator and Composite Patterns: Well-Managed Collections

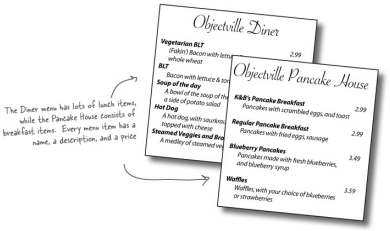
There are lots of ways to stuff objects into a collection. Put them in an Array, a Stack, a List, a Hashtable, take your pick. Each has its own advantages and tradeoffs. But at some point your client is going to want to iterate over those objects, and when he does, are you going to show him your implementation? We certainly hope not! That just wouldn't be professional. Well, you don't have to risk your career; you're going to see how you can allow your clients to iterate through your objects without ever getting a peek at how you store your objects. You're also going to learn how to create some *super collections* of objects that can leap over some impressive data structures in a single bound. And if that's not enough, you're also going to learn a thing or two about object responsibility.

Breaking News: Objectville Diner and Objectville Pancake House Merge

That's great news! Now we can get those delicious pancake breakfasts at the Pancake House and those yummy lunches at the Diner all in one place. But, there seems to be a slight problem...

Check out the Menu Items

At least Lou and Mel agree on the implementation of the MenuItems. Let's check out the items on each menu, and also take a look at the implementation.



```
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.

Lou and Mel's Menu implementations

Now let's take a look at what Lou and Mel are arguing about. They both have lots of time and code invested in the way they store their menu items in a menu, and lots of other code that depends on it.

Here's Lou's implementation of the Pancake House menu.

```
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.

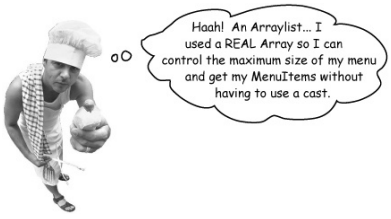
Each menu item is added to the ArrayList here, in the constructor.

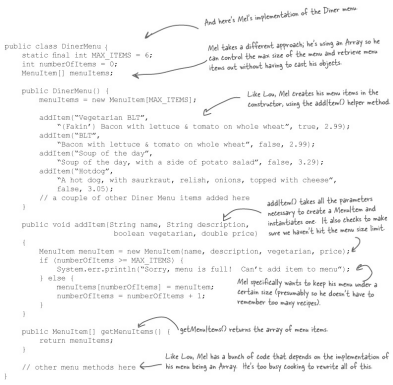
Each MenuItem has a name, a description, whether or not it's a vegetarian item, and the price.

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!





What's the problem with having two different menu representations?

To see why having two different menu representations complicates things, let's try implementing a client that uses the two menus. Imagine you have been hired by the new company formed by the merger of the Diner and the Pancake House to create a Java-enabled waitress (this *is* Objectville, after all). The spec for the Java-enabled waitress specifies that she can print a custom menu for customers on demand, and even tell you if a menu item is vegetarian without having to ask the cook – now that's an innovation!

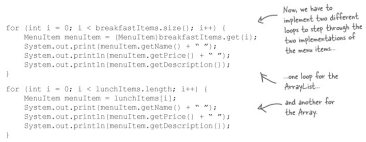
Let's check out the spec, and then step through what it might take to implement her...

The Java-Enabled Waitress Specification

Let's start by stepping through how we'd implement the printMenu() method:

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

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.



3. 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.

SHARPEN YOUR PENCIL

Based on our implementation of printMenu(), which of the following apply?

	A.	We are coding to the PancakeHouseMenu and DinerMenu concrete implementations, not to an interface.
	B.	The Waitress doesn't implement the Java Waitress API and so she isn't adhering to a standard.
	C.	If we decided to switch from using DinerMenu to another type of menu that implemented its list of menu items with a Hashtable, we'd have to modify a lot of code in the Waitress.
	D.	The Waitress needs to know how each menu represents its internal collection of menu items; this violates encapsulation.
	E.	We have duplicate code: the printMenu() method needs two separate loops to iterate over the two different kinds of menus. And if we added a third menu, we'd have yet another loop.
	F.	The implementation isn't based on MXML (Menu XML) and so isn't as interoperable as it should be.

What now?

Mel and Lou are putting us in a difficult position. They don't want to change their implementations because it would mean rewriting a lot of code that is in each respective menu class. But if one of them doesn't give in, then we're going to have the job of implementing a Waitress that is going to be hard to maintain and extend.

It would really be nice if we could find a way to allow them to implement the same interface for their menus (they're already close, except for the return type of the getMenuItems() method). That way we can minimize the concrete references in the Waitress code and also hopefully get rid of the multiple loops required to iterate over both menus.

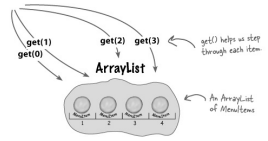
Sound good? Well, how are we going to do that?

Can we encapsulate the iteration?

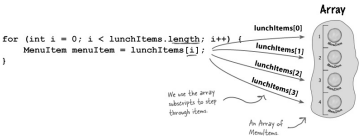
If we've learned one thing in this book, it's encapsulate what varies. It's obvious what is changing here: the iteration caused by different collections of objects being returned from the menus. But can we encapsulate this? Let's work through the idea...

1. To iterate through the breakfast items we use the size() and get() methods on the ArrayList:

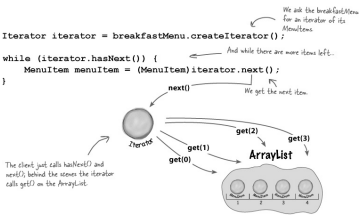
```
for (int i = 0; i < breakfastItems.size(); i++) {
    MenuItem menuItem = (MenuItem)breakfastItems.get(i);
}
```



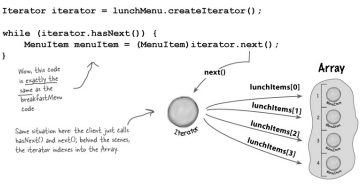
2. And to iterate through the lunch items we use the Array length field and the array subscript notation on the MenuItem Array.



3. 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



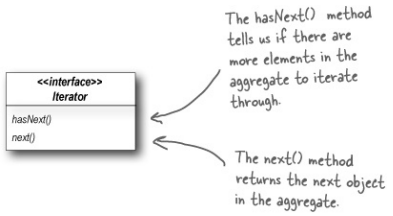
4. Let's try that on the Array too:



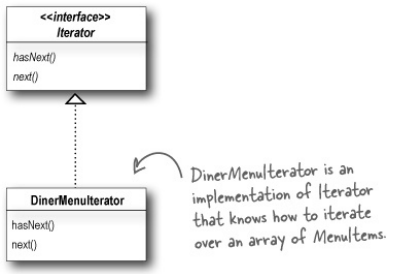
Meet the Iterator Pattern

Well, it looks like our plan of encapsulating iteration just might actually work; and as you've probably already guessed, it is a Design Pattern called the Iterator Pattern.

The first thing you need to know about the Iterator Pattern is that it relies on an interface called Iterator. Here's one possible Iterator interface:



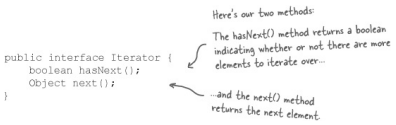
Now, once we have this interface, we can implement Iterators for any kind of collection of objects: arrays, lists, hashtables, ...pick your favorite collection of objects. Let's say we wanted to implement the Iterator for the Array used in the DinerMenu. It would look like this:



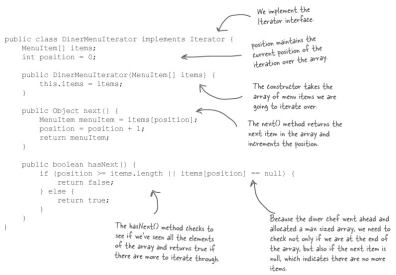
Let's go ahead and implement this Iterator and hook it into the DinerMenu to see how this works...

Adding an Iterator to DinerMenu

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

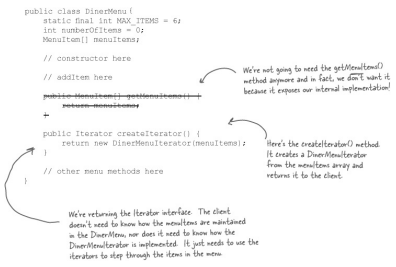


And now we need to implement a concrete Iterator that works for the Diner menu:



Reworking the Diner Menu with Iterator

Okay, we've got the iterator. Time to work it into the DinerMenu; all we need to do is add one method to create a DinerMenuIterator and return it to the client:



EXERCISE

Go ahead and implement the PancakeHouseIterator yourself and make the changes needed to incorporate it into the PancakeHouseMenu.

Fixing up the Waitress code

Now we need to integrate the iterator code into the Waitress. We should be able to get rid of some of the redundancy in the process. Integration is pretty straightforward: first we create a printMenu() method that takes an Iterator, then we use the createIterator() method on each menu to retrieve the Iterator and pass it to the new method.


```
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

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

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

Note that we're down to one loop

Testing our code

It's time to put everything to a test. Let's write some test drive code and see how the Waitress works...

```
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();
    }
}
```

First we create the new menus.

Then we create a Waitress and pass her the menus.

Then we print them.

Here's the test run...

```
File Edit Window Help DinnerEquals4Now

% 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

%
```

First we iterate through the pancake menu.

And then the lunch menu, all with the same iteration code

What have we done so far?

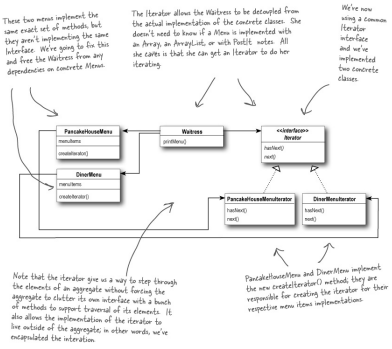
For starters, we've made our Objectville cooks very happy. They settled their differences and kept their own implementations. Once we gave them a PancakeHouseMenuIterator and a DinerMenuIterator, all they had to do was add a createIterator() method and they were finished.

We've also helped ourselves in the process. The Waitress will be much easier to maintain and extend down the road. Let's go through exactly what we did and think about the consequences:

Hard to Maintain Waitress Implementation	New, Hip Waitress Powered by Iterator
The Menus are not well encapsulated; we can see the Diner is using an ArrayList and the Pancake House an Array.	The Menu implementations are now encapsulated. The Waitress has no idea how the Menus hold their collection of menu items.
We need two loops to iterate through the MenuItems.	All we need is a loop that polymorphically handles any collection of items as long as it implements Iterator.
The Waitress is bound to concrete classes (MenuItem[] and ArrayList).	The Waitress now uses an interface (Iterator).
The Waitress is bound to two different concrete Menu classes, despite their interfaces being almost identical.	The Menu interfaces are now exactly the same and, uh oh, we still don't have a common interface, which means the Waitress is still bound to two concrete Menu classes. We'd better fix that.

What we have so far...

Before we clean things up, let's get a bird's eye view of our current design.



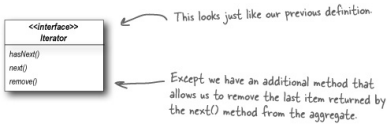
Making some improvements...

Okay, we know the interfaces of PancakeHouseMenu and DinerMenu are exactly the same and yet we haven't defined a common interface for them. So, we're going to do that and clean up the Waitress a little more.

You may be wondering why we're not using the Java Iterator interface – we did that so you could see how to build an iterator from scratch. Now that we've done that, we're going to switch to using the Java Iterator interface, because we'll get a lot of leverage by implementing that instead of our home grown

Iterator interface. What kind of leverage? You'll soon see.

First, let's check out the java.util.Iterator interface:



This is going to be a piece of cake: We just need to change the interface that both PancakeHouseMenuIterator and DinerMenuIterator extend, right? Almost... actually, it's even easier than that. Not only does java.util have its own Iterator interface, but ArrayList has an iterator() method that returns an iterator. In other words, we never needed to implement our own iterator for ArrayList. However, we'll still need our implementation for the DinerMenu because it relies on an Array, which doesn't support the iterator() method (or any other way to create an array iterator).

THERE ARE NO DUMB QUESTIONS

Q: **Q: What if I don't want to provide the ability to remove something from the underlying collection of objects?**

A: **A:** *The remove() method is considered optional. You don't have to provide remove functionality. But, obviously you do need to provide the method because it's part of the Iterator interface. If you're not going to allow remove() in your iterator you'll want to throw the runtime exception java.lang.UnsupportedOperationException. The Iterator API documentation specifies that this exception may be thrown from remove() and any client that is a good citizen will check for this exception when calling the remove() method.*

Q: **Q: How does remove() behave under multiple threads that may be using different iterators over the same collection of objects?**

A: **A:** *The behavior of the remove() is unspecified if the collection changes while you are iterating over it. So you should be careful in designing your own multithreaded code when accessing a collection concurrently.*

Cleaning things up with java.util.Iterator

Let's start with the PancakeHouseMenu, changing it over to java.util.Iterator is going to be easy. We just delete the PancakeHouseMenuIterator class, add an import java.util.Iterator to the top of PancakeHouseMenu and change one line of the PancakeHouseMenu:

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

Instead of creating our own iterator now, we just call the iterator() method on the menuItems ArrayList.

And that's it, PancakeHouseMenu is done.

Now we need to make the changes to allow the DinerMenu to work with java.util.Iterator.

```
import java.util.Iterator;

public class DinerMenuIterator implements Iterator {
    MenuItem[] list;
    int position = 0;

    public DinerMenuIterator(MenuItem[] list) {
        this.list = list;
    }

    public Object next() {
        //implementation here
    }

    public boolean hasNext() {
        //implementation here
    }

    public void remove() {
        if (position <= 0) {
            throw new IllegalStateException
                ("You can't remove an item until you've done at least one next()");
        }
        if (list[position-1] != null) {
            for (int i = position-1; i < (list.length-1); i++) {
                list[i] = list[i+1];
            }
            list[list.length-1] = null;
        }
    }
}
```

First we import java.util.Iterator, the interface we're going to implement.

None of our current implementation changes...

...but we do need to implement remove(). Here, because the chef is using a fixed sized Array, we just shift all the elements up one when remove() is called.

We are almost there...

We just need to give the Menus a common interface and rework the Waitress a

little. The Menu interface is quite simple: we might want to add a few more methods to it eventually, like addItem(), but for now we will let the chefs control their menus by keeping that method out of the public interface:

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

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

Now we need to add an `implements Menu` to both the PancakeHouseMenu and the DinerMenu class definitions and update the Waitress:

```
import java.util.Iterator;

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
}
```

Now the Waitress uses the java.util.Iterator as well.

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

Nothing changes here.

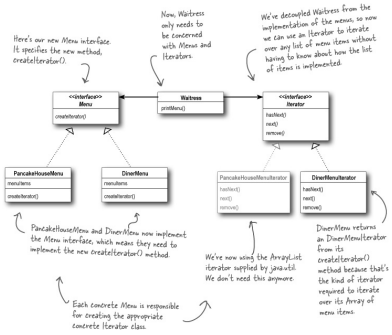
What does this get us?

The PancakeHouseMenu and DinerMenu classes implement an interface, Menu. Waitress can refer to each menu object using the interface rather than the concrete class. So, we're reducing the dependency between the Waitress and the concrete classes by "programming to an interface, not an implementation."

> NOTE
>
> This solves the problem of the Waitress depending on the concrete Menus.

The new Menu interface has one method, createIterator(), that is implemented by PancakeHouseMenu and DinerMenu. Each menu class assumes the responsibility of creating a concrete Iterator that is appropriate for its internal implementation of the menu items.

> NOTE
>
> This solves the problem of the Waitress depending on the implementation of the MenuItems.



Iterator Pattern defined

You've already seen how to implement the Iterator Pattern with your very own iterator. You've also seen how Java supports iterators in some of its collection oriented classes (the ArrayList). Now it's time to check out the official definition of the pattern:

> NOTE
>
> **The Iterator Pattern** provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

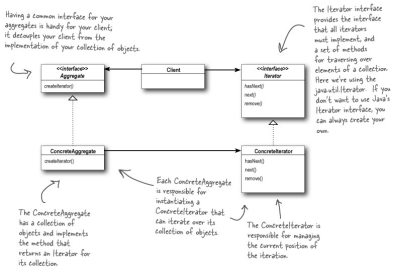
This makes a lot of sense: the pattern gives you a way to step through the elements of an aggregate without having to know how things are represented

under the covers. You've seen that with the two implementations of Menus. But the effect of using iterators in your design is just as important: once you have a uniform way of accessing the elements of all your aggregate objects, you can write polymorphic code that works with *any* of these aggregates – just like the printMenu() method, which doesn't care if the menu items are held in an Array or ArrayList (or anything else that can create an Iterator), as long as it can get hold of an Iterator.

> **The Iterator Pattern allows traversal of the elements of an aggregate without exposing the underlying implementation.**
>
> **It also places the task of traversal on the iterator object, not on the aggregate, which simplifies the aggregate interface and implementation, and places the responsibility where it should be.**

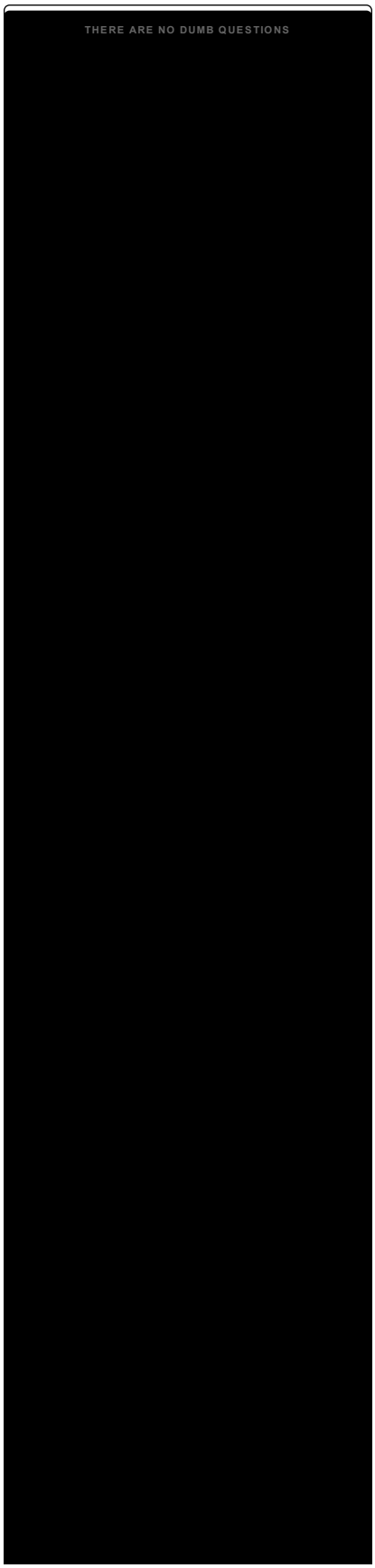
The other important impact on your design is that the Iterator Pattern takes the responsibility of traversing elements and gives that responsibility to the iterator object, not the aggregate object. This not only keeps the aggregate interface and implementation simpler, it removes the responsibility for iteration from the aggregate and keeps the aggregate focused on the things it should be focused on (managing a collection of objects), not on iteration.

Let's check out the class diagram to put all the pieces in context...



BRAIN POWER

The class diagram for the Iterator Pattern looks very similar to another Pattern you've studied; can you think of what it is? Hint: A subclass decides which object to create.


THERE ARE NO DUMB QUESTIONS

Q: **Q: I've seen other books show the Iterator class diagram with the methods first(), next(), isDone() and currentItem(). Why are these methods different?**

A: **A:** *Those are the "classic" method names that have been used. These names have changed over time and we now have next(), hasNext() and even remove() in java.util.Iterator.*

Let's look at the classic methods. The next() and currentItem() have been merged into one method in java.util. The isDone() method has obviously become hasNext(); but we have no method corresponding to first(). That's because in Java we tend to just get a new iterator whenever we need to start the traversal over. Nevertheless, you can see there is very little difference in these interfaces. In fact, there is a whole range of behaviors you can give your iterators. The remove() method is an example of an extension in java.util.Iterator.

Q: **Q: I've heard about "internal" iterators and "external" iterators. What are they? Which kind did we implement in the example?**

A: **A:** *We implemented an external iterator, which means that the client controls the iteration by calling next() to get the next element. An internal iterator is controlled by the iterator itself. In that case, because it's the iterator that's stepping through the elements, you have to tell the iterator what to do with those elements as it goes through them. That means you need a way to pass an operation to an iterator. Internal iterators are less flexible that external iterators because the client doesn't have control of the iteration. However, some might argue that they are easier to use because you just hand them an operation and tell them to iterate, and they do all the work for you.*

Q: **Q: Could I implement an Iterator that can go backwards as well as forwards?**

A: **A:** *Definitely. In that case, you'd probably want to add two methods, one to get to the previous element, and one to tell you when you're at the beginning of the collection of elements. Java's Collection Framework provides another type of iterator interface called ListIterator. This iterator adds previous() and a few other methods to the standard Iterator interface. It is supported by any Collection that implements the List interface.*

Q: **Q: Who defines the ordering of the iteration in a collection like Hashtable, which are inherently unordered?**

A: **A:** *Iterators imply no ordering. The underlying collections may be unordered as in a hashtable or in a bag; they may even contain duplicates. So ordering is related to both the properties of the underlying collection and to the implementation. In general, you should make no assumptions about ordering unless the Collection documentation indicates otherwise.*

Q: **Q: You said we can write "polymorphic code" using an iterator; can you explain that more?**

A: **A:** *When we write methods that take Iterators as parameters, we are using polymorphic iteration. That means we are creating code that can iterate over any collection as long as it supports Iterator. We don't care about how the collection is implemented, we can still write code to iterate over it.*

Q: **Q: If I'm using Java, won't I always want to use the java.util.Iterator interface so I can use my own iterator implementations with classes that are already using the Java iterators?**

A: **A:** *Probably. If you have a common Iterator interface, it will certainly make it easier for you to mix and match your*

own aggregates with Java aggregates like ArrayList and Vector. But remember, if you need to add functionality to your Iterator interface for your aggregates, you can always extend the Iterator interface.

Q: **Q: I've seen an Enumeration interface in Java; does that implement the Iterator Pattern?**

A: **A:** *We talked about this in the Adapter Chapter. Remember? The java.util.Enumeration is an older implementation of Iterator that has since been replaced by java.util.Iterator. Enumeration has two methods, hasMoreElements(), corresponding to hasNext(), and nextElement(), corresponding to next(). However, you'll probably want to use Iterator over Enumeration as more Java classes support it. If you need to convert from one to another, review the Adapter Chapter again where you implemented the adapter for Enumeration and Iterator.*

Single Responsibility

What if we allowed our aggregates to implement their internal collections and related operations AND the iteration methods? Well, we already know that would expand the number of methods in the aggregate, but so what? Why is that so bad?

> **Every responsibility of a class is an area of potential change. More than one responsibility means more than one area of change.**
>
> **This principle guides us to keep each class to a single responsibility.**

Well, to see why, you first need to recognize that when we allow a class to not only take care of its own business (managing some kind of aggregate) but also take on more responsibilities (like iteration) then we've given the class two reasons to change. Two? Yup, two: it can change if the collection changes in some way, and it can change if the way we iterate changes. So once again our friend CHANGE is at the center of another design principle:

NOTE

Design Principle

A class should have only one reason to change.

We know we want to avoid change in a class like the plague – modifying code provides all sorts of opportunities for problems to creep in. Having two ways to change increases the probability the class will change in the future, and when it does, it's going to affect two aspects of your design.

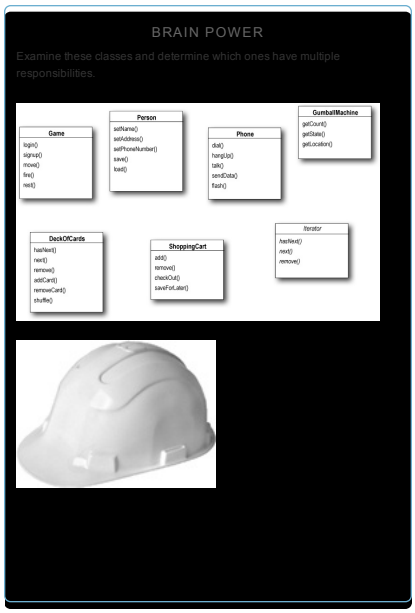
The solution? The principle guides us to assign each responsibility to one class, and only one class.

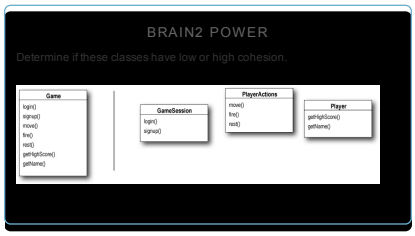
That's right, it's as easy as that, and then again it's not: separating responsibility in design is one of the most difficult things to do. Our brains are just too good at seeing a set of behaviors and grouping them together even when there are actually two or more responsibilities. The only way to succeed is to be diligent in examining your designs and to watch out for signals that a class is changing in more than one way as your system grows.

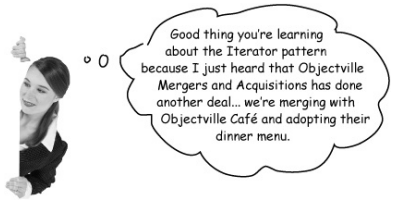
Cohesion is a term you'll hear used as a measure of how closely a class or a module supports a single purpose or responsibility.

We say that a module or class has *high cohesion* when it is designed around a set of related functions, and we say it has *low cohesion* when it is designed around a set of unrelated functions.

Cohesion is a more general concept than the Single Responsibility Principle, but the two are closely related. Classes that adhere to the principle tend to have high cohesion and are more maintainable than classes that take on multiple responsibilities and have low cohesion.


BRAIN POWER
Examine these classes and determine which ones have multiple responsibilities.
Game
login()
signup()
move()
fire()
next()
Person
setName()
setAddress()
setPhoneNumber()
save()
load()
Phone
dial()
hangUp()
talk()
sendData()
flash()
GumballMachine
getCount()
getState()
getLocation()
DeckOfCards
hasNext()
next()
remove()
addCard()
removeCard()
shuffle()
ShoppingCart
add()
remove()
checkOut()
saveForLater()
Iterator
hasNext()
next()
remove()


BRAIN2 POWER
Determine if these classes have low or high cohesion.
Game
login()
signup()
move()
fire()
next()
getHighScore()
getName()
GameSession
login()
signup()
PlayerActions
move()
fire()
next()
Player
getHighScore()
getName()


Good thing you're learning about the Iterator pattern because I just heard that Objectville Mergers and Acquisitions has done another deal... we're merging with Objectville Café and adopting their dinner menu.


Wow, and we thought things were already complicated. Now what are we going to do?

Taking a look at the Café Menu

Here's the Café Menu. It doesn't look like too much trouble to integrate the Cafe Menu into our framework... let's check it out.

```
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;
    }
}
```

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...

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.

SHARPEN YOUR PENCIL

Before looking at the next page, quickly jot down the three things we have to do to this code to fit it into our framework:

1. _____

2. _____

3. _____

Reworking the Café Menu code

Integrating the Cafe Menu into our framework is easy. Why? Because Hashtable is one of those Java collections that supports Iterator. But it's not quite the same as ArrayList...

```
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.

CODE UP CLOSE

Hashtable is a little more complex than the ArrayList because it supports both keys and values, but we can still get an Iterator for the values (which are the MenuItems).

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

First we get the values of the Hashtable, which is just a collection of all the objects in the hashtable.

Luckily that collection supports the iterator() method, which returns a object of type java.util.Iterator.

Adding the Café Menu to the Waitress

That was easy; how about modifying the Waitress to support our new Menu? Now that the Waitress expects Iterators, that should be easy too.

```
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

Breakfast, lunch AND dinner

Let's update our test drive to make sure this all works.

```
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.

Here's the test run; check out the new dinner menu from the Café!

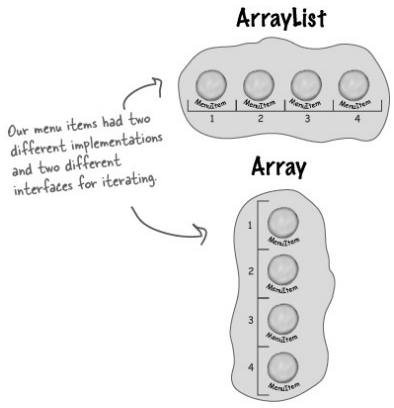
```
% java DinerMenuTestDrive
MENU
----
BREAKFAST
K&B's Pancake Breakfast, 2.99 -- Pancakes with scrambled eggs, and toast
Regular Pancake Breakfast, 2.99 -- Pancakes with fried eggs, sausage
Blueberry Pancakes, 3.49 -- Pancakes made with fresh blueberries
Waffles, 3.59 -- Waffles, with your choice of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

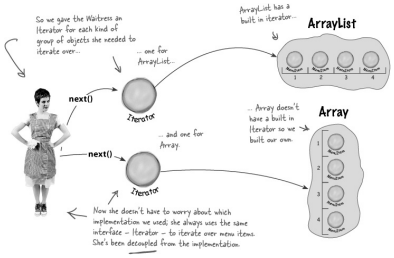
And then the diner menu

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

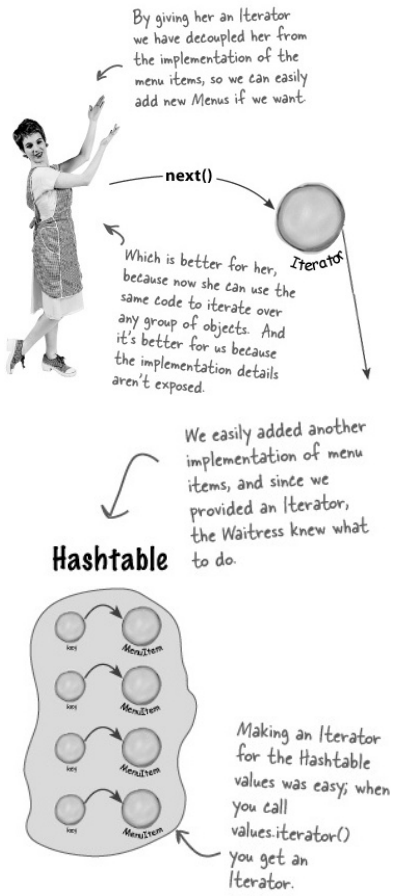
What did we do?



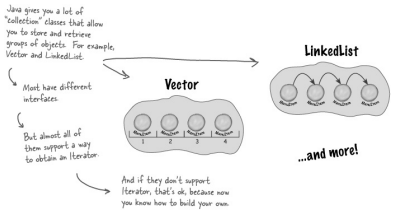
We decoupled the Waitress....



... and we made the Waitress more extensible

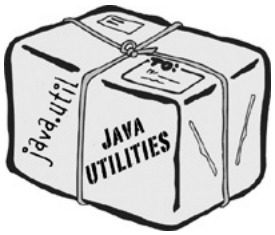


But there's more!

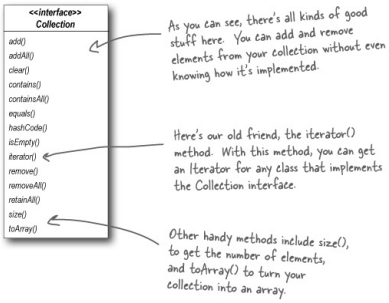


Iterators and Collections

We've been using a couple of classes that are part of the Java Collections Framework. This "framework" is just a set of classes and interfaces, including ArrayList, which we've been using, and many others like Vector, LinkedList, Stack, and PriorityQueue. Each of these classes implements the java.util.Collection interface, which contains a bunch of useful methods for manipulating groups of objects.



Let's take a quick look at the interface:



WATCH IT!

Hashtable is one of a few classes that *indirectly* supports Iterator. As you saw when we implemented the CafeMenu, you could get an Iterator from it, but only by first retrieving its Collection called values. If you think about it, this makes sense: the Hashtable holds two sets of objects: keys and values. If we want to iterate over its values, we first need to retrieve them from the Hashtable, and then obtain the iterator.

Iterators and Collections in Java 5

Java 5 includes a new form of the **for** statement, called **for/in**, that lets you iterate over a collection or an array without creating an iterator explicitly.

To use **for/in**, you use a **for** statement that looks like:

Iterates over each object in the collection.

obj is assigned to the next element in the collection each time through the loop.

```
for (Object obj: collection) {
        ...
}
```

Here's how you iterate over an ArrayList using **for/in**:

Load up an ArrayList of MenuItems.

```
ArrayList items = new ArrayList();
items.add(new MenuItem("Pancakes", "delicious pancakes", true, 1.59);
items.add(new MenuItem("Waffles", "yummy waffles", true, 1.99);
items.add(new MenuItem("Toast", "perfect toast", true, 0.59);

for (MenuItem item: items) {
    System.out.println("Breakfast item: " + item);
}
```

Iterate over the list and print each item.

WATCH IT!

You need to use Java 5's new generics feature to ensure for/in type safety. Make sure you read up on the details before using generics and for/in.

CODE MAGNETS

The Chefs have decided that they want to be able to alternate their lunch menu items; in other words, they will offer some items on Monday, Wednesday, Friday and Sunday, and other items on Tuesday, Thursday, and Saturday. Someone already wrote the code for a new "Alternating" DinerMenu Iterator so that it alternates the menu items, but they scrambled it up and put it on the fridge in the Diner as a joke. Can you put it back together? Some of the curly braces fell on the floor and they were too small to pick up, so feel free to add as many of those as you need.

```
MenuItem menuItem = items[position];
position = position + 2;
return menuItem;

import java.util.Iterator;
import java.util.Calendar;

public Object next() {

}

public AlternatingDinerMenuIterator(MenuItem[] items)

this.items = items;
Calendar rightNow = Calendar.getInstance();
position = rightNow.get(Calendar.DAY_OF_WEEK) % 2;

public void remove() {

implements Iterator

MenuItem[] items;
int position;

}

public class AlternatingDinerMenuIterator

public boolean hasNext() {

throw new UnsupportedOperationException(
    "Alternating Diner Menu Iterator does not support remove()");

if (position >= items.length || items[position] == null) {
    return false;
} else {
    return true;
}

}
```

Is the Waitress ready for prime time?

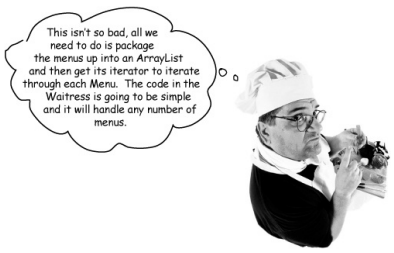
The Waitress has come a long way, but you've gotta admit those three calls to printMenu() are looking kind of ugly.

Let's be real, every time we add a new menu we are going to have to open up the Waitress implementation and add more code. Can you say "violating the Open Closed Principle?"

It's not the Waitress' fault. We have done a great job of decoupling the menu implementation and extracting the iteration into an iterator. But we still are handling the menus with separate, independent objects – we need a way to manage them together.

BRAIN POWER

The Waitress still needs to make three calls to printMenu(), one for each menu. Can you think of a way to combine the menus so that only one call needs to be made? Or perhaps so that one Iterator is passed to the Waitress to iterate over all the menus?



Sounds like the chef is on to something. Let's give it a try:

```
public class Waitress {
    ArrayList menus;

    public Waitress(ArrayList 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());
        }
    }
}
```

Now we just take an ArrayList of menus.

And we iterate through the menus, passing each menu's iterator to the overloaded printMenu() method.

No code changes here.

This looks pretty good, although we've lost the names of the menus, but we could add the names to each menu.

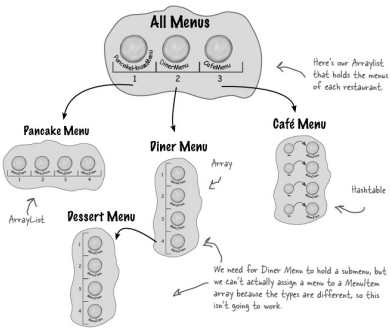
Just when we thought it was safe...

Now they want to add a dessert submenu.

Okay, now what? Now we have to support not only multiple menus, but menus within menus.

It would be nice if we could just make the dessert menu an element of the DinerMenu collection, but that won't work as it is now implemented.

What we want (something like this):



But this won't work!

We can't assign a dessert menu to a MenuItem array.

Time for a change!

What do we need?

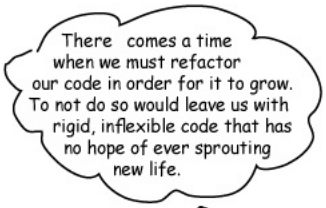
The time has come to make an executive decision to rework the chef's implementation into something that is general enough to work over all the menus (and now sub menus). That's right, we're going to tell the chefs that the time has come for us to reimplement their menus.

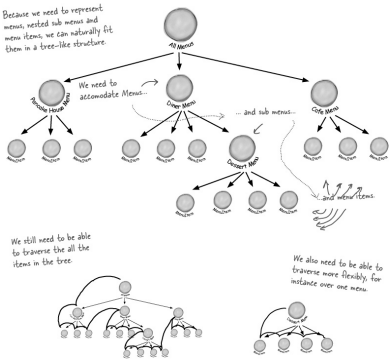
The reality is that we've reached a level of complexity such that if we don't rework the design now, we're never going to have a design that can accommodate further acquisitions or submenus.

So, what is it we really need out of our new design?

- We need some kind of a tree shaped structure that will accommodate menus, submenus and menu items.

- We need to make sure we maintain a way to traverse the items in each menu that is at least as convenient as what we are doing now with iterators.
- We may need to be able traverse the items in a more flexible manner. For instance, we might need to iterate over only the Diner's dessert menu, or we might need to iterate over the Diner's entire menu, including the dessert submenu.



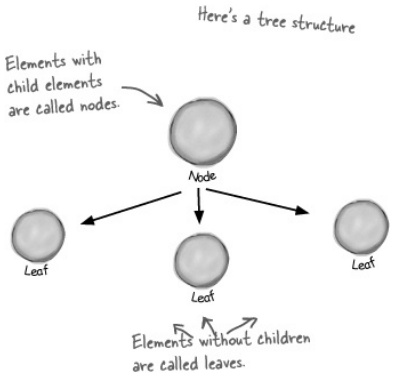


BRAIN POWER

How would you handle this new wrinkle to our design requirements? Think about it before turning the page.

The Composite Pattern defined

That's right, we're going to introduce another pattern to solve this problem. We didn't give up on Iterator – it will still be part of our solution – however, the problem of managing menus has taken on a new dimension that Iterator doesn't solve. So, we're going to step back and solve it with the Composite Pattern.



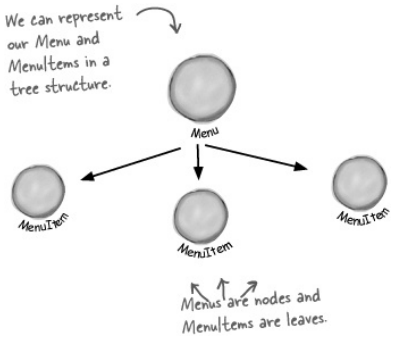
We're not going to beat around the bush on this pattern, we're going to go ahead and roll out the official definition now:

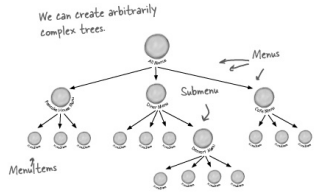
NOTE

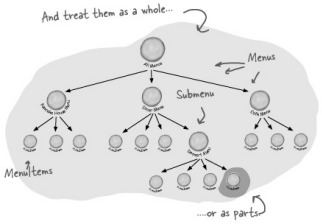
The Composite Pattern allows you to compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

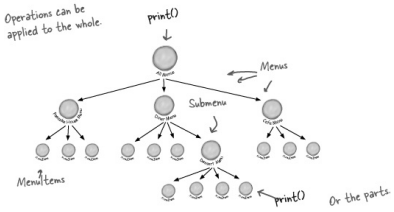
Let's think about this in terms of our menus: this pattern gives us a way to create a tree structure that can handle a nested group of menus *and* menu items in the same structure. By putting menus and items in the same structure we create a part-whole hierarchy; that is, a tree of objects that is made of parts (menus and menu items) but that can be treated as a whole, like one big über menu.

Once we have our über menu, we can use this pattern to treat "individual objects and compositions uniformly." What does that mean? It means if we have a tree structure of menus, submenus, and perhaps subsubmenus along with menu items, then any menu is a "composition" because it can contain both other menus and menu items. The individual objects are just the menu items – they don't hold other objects. As you'll see, using a design that follows the Composite Pattern is going to allow us to write some simple code that can apply the same operation (like printing!) over the entire menu structure.



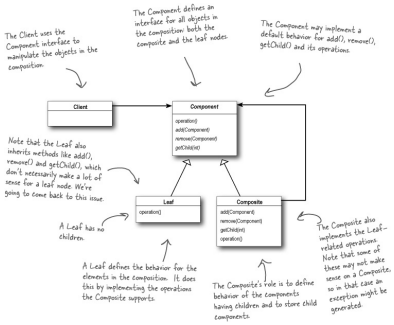






The Composite Pattern allows us to build structures of objects in the form of trees that contain both compositions of objects and individual objects as nodes.

Using a composite structure, we can apply the same operations over both composites and individual objects. In other words, in most cases we can ignore the differences between compositions of objects and individual objects.



THERE ARE NO DUMB QUESTIONS

Q: **Q: Component, Composite, Trees? I'm confused.**

A: **A:** *A composite contains components. Components come in two flavors: composites and leaf elements. Sound recursive? It is. A composite holds a set of children, those children may be other composites or leaf elements.*

When you organize data in this way you end up with a tree structure (actually an upside down tree structure) with a composite at the root and branches of composites growing up to leaf nodes.

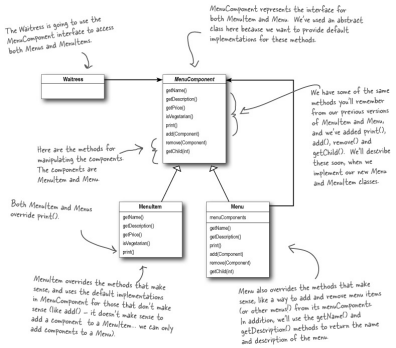
Q: **Q: How does this relate to iterators?**

A: **A:** *Remember, we're taking a new approach. We're going to re-implement the menus with a new solution: the Composite Pattern. So don't look for some magical transformation from an iterator to a composite. That said, the two work very nicely together. You'll soon see that we can use iterators in a couple of ways in the composite implementation.*

Designing Menus with Composite

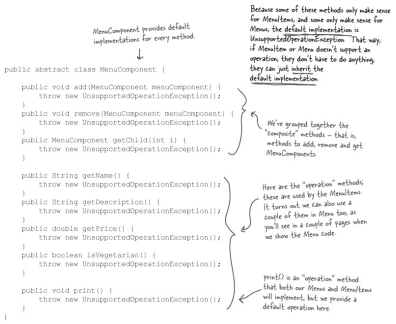
So, how do we apply the Composite Pattern to our menus? To start with, we need to create a component interface; this acts as the common interface for both menus and menu items and allows us to treat them uniformly. In other words we can call the *same* method on menus or menu items.

Now, it may not make *sense* to call some of the methods on a menu item or a menu, but we can deal with that, and we will in just a moment. But for now, let's take a look at a sketch of how the menus are going to fit into a Composite Pattern structure:



Implementing the Menu Component

Okay, we're going to start with the MenuComponent abstract class; remember, the role of the menu component is to provide an interface for the leaf nodes and the composite nodes. Now you might be asking, "Isn't the MenuComponent playing two roles?" It might well be and we'll come back to that point. However, for now we're going to provide a default implementation of the methods so that if the MenuItem (the leaf) or the Menu (the composite) doesn't want to implement some of the methods (like getChild() for a leaf node) they can fall back on some basic behavior:

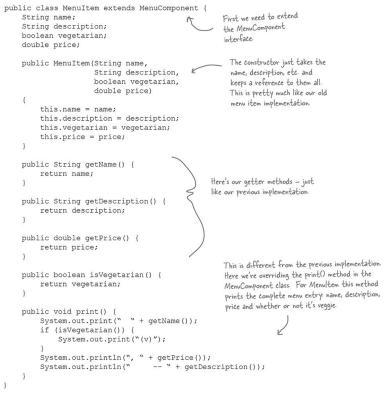


NOTE

All components must implement the MenuComponent interface; however, because leaves and nodes have different roles we can't always define a default implementation for each method that makes sense. Sometimes the best you can do is throw a runtime exception.

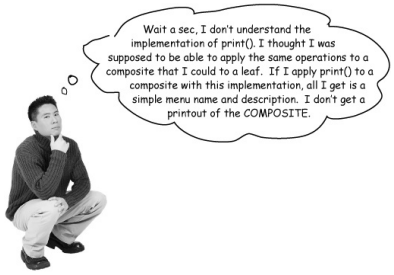
Implementing the Menu Item

Okay, let's give the MenuItem class a shot. Remember, this is the leaf class in the Composite diagram and it implements the behavior of the elements of the composite.



Implementing the Composite Menu

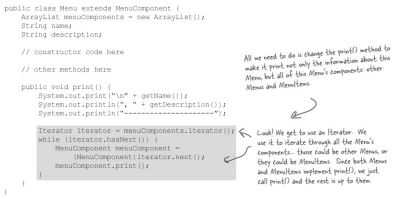
Now that we have the MenuItem, we just need the composite class, which we're calling Menu. Remember, the composite class can hold MenuItems *or* other Menus. There's a couple of methods from MenuComponent this class doesn't implement: getPrice() and isVegetarian(), because those don't make a lot of sense for a Menu.



Excellent catch. Because menu is a composite and contains both Menu Items and other Menus, its print() method should print everything it contains. If it didn't we'd have to iterate through the entire composite and print each item ourselves. That kind of defeats the purpose of having a composite structure.

As you're going to see, implementing print() correctly is easy because we can rely on each component to be able to print itself. It's all wonderfully recursive and groovy. Check it out:

Fixing the print() method

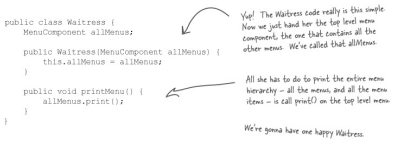


NOTE

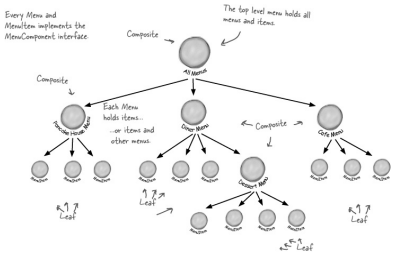
NOTE: If, during this iteration, we encounter another Menu object, its print() method will start another iteration, and so on.

Getting ready for a test drive...

It's about time we took this code for a test drive, but we need to update the Waitress code before we do – after all she's the main client of this code:



Okay, one last thing before we write our test drive. Let's get an idea of what the menu composite is going to look like at runtime:



Now for the test drive...

Okay, now we just need a test drive. Unlike our previous version, we're going to handle all the menu creation in the test drive. We could ask each chef to give us his new menu, but let's get it all tested first. Here's the code:

```
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",
            "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 ice cream",
            true,
            1.59));

        // add more menu items here

        Waitress waitress = new Waitress(allMenus);

        waitress.printMenu();
    }
}
```

Let's first create all the menu objects.

We also need a top level menu 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.

And we've 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.

Getting ready for a test drive...

> **NOTE**
>
> NOTE: this output is based on the complete source.

```
% 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 sauerkraut, 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

There is some truth to that observation. We could say that the Composite Pattern takes the Single Responsibility design principle and trades it for

transparency. What's transparency? Well, by allowing the Component interface to contain the child management operations *and* the leaf operations, a client can treat both composites and leaf nodes uniformly; so whether an element is a composite or leaf node becomes transparent to the client.

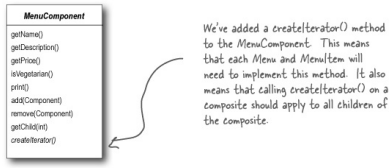
Now given we have both types of operations in the Component class, we lose a bit of *safety* because a client might try to do something inappropriate or meaningless on an element (like try to add a menu to a menu item). This is a design decision; we could take the design in the other direction and separate out the responsibilities into interfaces. This would make our design safe, in the sense that any inappropriate calls on elements would be caught at compile time or runtime, but we'd lose transparency and our code would have to use conditionals and the `instanceof` operator.

So, to return to your question, this is a classic case of tradeoff. We are guided by design principles, but we always need to observe the effect they have on our designs. Sometimes we purposely do things in a way that seems to violate the principle. In some cases, however, this is a matter of perspective; for instance, it might seem incorrect to have child management operations in the leaf nodes (like add(), remove() and getChild()), but then again you can always shift your perspective and see a leaf as a node with zero children.

Flashback to Iterator

We promised you a few pages back that we'd show you how to use Iterator with a Composite. You know that we are already using Iterator in our internal implementation of the print() method, but we can also allow the Waitress to iterate over an entire composite if she needs to, for instance, if she wants to go through the entire menu and pull out vegetarian items.

To implement a Composite iterator, let's add a createIterator() method in every component. We'll start with the abstract MenuComponent class:



Now we need to implement this method in the Menu and MenuItem classes:

```
public class Menu extends MenuComponent {
    // other code here doesn't change
    public Iterator createIterator() {
        return new CompositeIterator(menuComponents.iterator());
    }
}
```

Here we're using a new iterator called CompositeIterator. It knows how to iterate over any composite. We pass it the current composite's iterator.

```
public class MenuItem extends MenuComponent {
    // other code here doesn't change
    public Iterator createIterator() {
        return new NullIterator();
    }
}
```

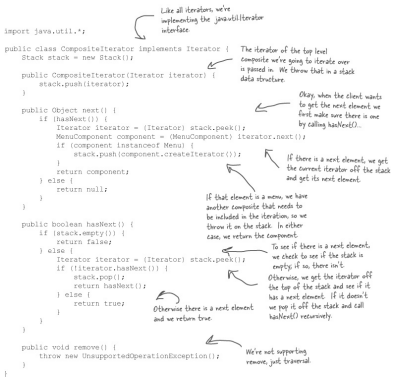
Now for the MenuItem...

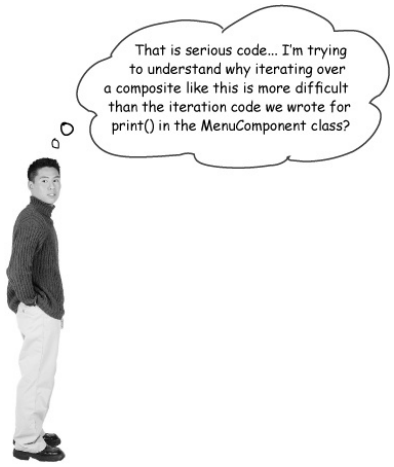
Whoa! What's this NullIterator? You'll see in two pages.

The Composite Iterator

The CompositeIterator is a SERIOUS iterator. It's got the job of iterating over the MenuItems in the component, and of making sure all the child Menus (and child child Menus, and so on) are included.

Here's the code. Watch out, this isn't a lot of code, but it can be a little mind bending. Just repeat to yourself as you go through it "recursion is my friend, recursion is my friend."





When we wrote the print() method in the MenuComponent class we used an iterator to step through each item in the component and if that item was a Menu (rather than a MenuItem), then we recursively called the print() method to handle it. In other words, the MenuComponent handled the iteration itself, *internally*.

With this code we are implementing an *external* iterator so there is a lot more to keep track of. For starters, an external iterator must maintain its position in the iteration so that an outside client can drive the iteration by calling hasNext() and next(). But in this case, our code also needs to maintain that position over a composite, recursive structure. That's why we use stacks to maintain our position as we move up and down the composite hierarchy.

BRAIN POWER

Draw a diagram of the Menus and MenuItems. Then pretend you are the CompositeIterator, and your job is to handle calls to hasNext() and next(). Trace the way the CompositeIterator traverses the structure as this code is executed:

```
public void testCompositeIterator(MenuComponent component) {
       CompositeIterator iterator = new CompositeIterator(component

       while(iterator.hasNext()) {
              MenuComponent component = iterator.next();
       }
}
```

The Null Iterator

Okay, now what is this Null Iterator all about? Think about it this way: a MenuItem has nothing to iterate over, right? So how do we handle the implementation of its createIterator() method? Well, we have two choices:

NOTE

NOTE: Another example of the Null Object "Design Pattern."

- **Choice one:**

- — Return null

  — We could return null from createIterator(), but then we'd need conditional code in the client to see if null was returned or not.

- **Choice two:**

- — Return an iterator that always returns false when hasNext() is called

  — This seems like a better plan. We can still return an iterator, but the client doesn't have to worry about whether or not null is ever returned. In effect, we're creating an iterator that is a "no op".

The second choice certainly seems better. Let's call it NullIterator and implement it.

```
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();
    }
}
```

This is the laziest Iterator you've ever seen, at every step of the way it punts.

When next() is called, we return null.

Most importantly when hasNext() is called we always return false.

And the NullIterator wouldn't think of supporting remove.

Give me the vegetarian menu

Now we've got a way to iterate over every item of the Menu. Let's take that and give our Waitress a method that can tell us exactly which items are vegetarian.

```
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?

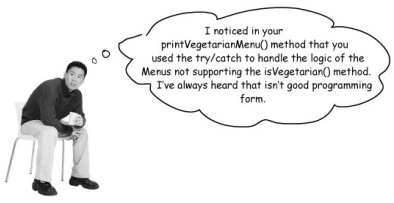
We implemented isVegetarian() on the Menus to always throw an exception. If that happens we catch the exception, but continue with our iteration.

The magic of Iterator & Composite together...

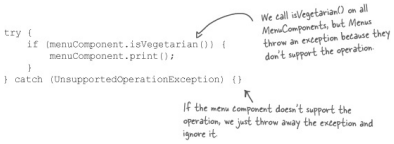
Whooo! It's been quite a development effort to get our code to this point. Now we've got a general menu structure that should last the growing Diner empire for some time. Now it's time to sit back and order up some veggie food:

```
% java MenuTestDrive

VEGETARIAN MENU
----
  K&B's Pancake Breakfast(v), 2.99
    -- Pancakes with scrambled eggs, and toast
  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
  Vegetarian BLT(v), 2.99
    -- (Fakin') Bacon with lettuce & tomato on whole wheat
  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
  Apple Pie(v), 1.59
    -- Apple pie with a flakey crust, topped with vanilla ice cream
  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
  Apple Pie(v), 1.59
    -- Apple pie with a flakey crust, topped with vanilla ice cream
  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
  Veggie Burger and Air Fries(v), 3.99
    -- Veggie burger on a whole wheat bun, lettuce, tomato, and fries
  Burrito(v), 4.29
    -- A large burrito, with whole pinto beans, salsa, guacamole
%
```

The Vegetarian Menu consists of the vegetarian items from every menu.



Let's take a look at what you're talking about:



In general we agree; try/catch is meant for error handling, not program logic. What are our other options? We could have checked the runtime type of the menu component with instanceof to make sure it's a MenuItem before making the call to isVegetarian(). But in the process we'd lose *transparency* because we wouldn't be treating Menus and MenuItems uniformly.

We could also change isVegetarian() in the Menus so that it returns false. This provides a simple solution and we keep our transparency.

In our solution we are going for clarity: we really want to communicate that this is an unsupported operation on the Menu (which is different than saying isVegetarian() is false). It also allows for someone to come along and actually implement a reasonable isVegetarian() method for Menu and have it work with the existing code.

That's our story and we're stickin' to it.

PATTERNS EXPOSED

This week's interview: The Composite Pattern, on Implementation issues

HeadFirst: We're here tonight speaking with the Composite Pattern. Why don't you tell us a little about yourself, Composite?

Composite: Sure... I'm the pattern to use when you have collections of objects with whole-part relationships and you want to be able to treat those objects uniformly.

HeadFirst: Okay, let's dive right in here... what do you mean by whole-part relationships?

Composite: Imagine a graphical user interface; there you'll often find a top level component like a Frame or a Panel, containing other components, like menus, text panes, scrollbars and buttons. So your GUI consists of several parts, but when you display it, you generally think of it as a whole. You tell the top level component to display, and count on that component to display all its parts. We call the components that contain other components, composite objects, and components that don't contain other components, leaf objects.

HeadFirst: Is that what you mean by treating the objects uniformly? Having common methods you can call on composites and leaves?

Composite: Right. I can tell a composite object to display or a leaf object to display and they will do the right thing. The composite object will display by telling all its components to display.

HeadFirst: That implies that every object has the same interface. What if you have objects in your composite that do different things?

Composite: Well, in order for the composite to work transparently to the client, you must implement the same interface for all objects in the composite, otherwise, the client has to worry about which interface each object is implementing, which kind of defeats the purpose. Obviously that means that at times you'll have objects for which some of the method calls don't make sense.

HeadFirst: So how do you handle that?

Composite: Well there's a couple of ways to handle it; sometimes you can just do nothing, or return null or false – whatever makes sense in your application. Other times you'll want to be more proactive and throw an exception. Of course, then the client has to be willing to do a little work and make sure that the method call didn't do something unexpected.

HeadFirst: But if the client doesn't know which kind of object they're dealing with, how would they ever know which calls to make without checking the type?

Composite: If you're a little creative you can structure your methods so that the default implementations do something that does make sense. For instance, if the client is calling getChild(), on the composite this makes sense. And it makes sense on a leaf too, if you think of the leaf as an object with no children.

HeadFirst: Ah... smart. But, I've heard some clients are so worried about this issue, that they require separate interfaces for different objects so they aren't allowed to make nonsensical method calls. Is that still the Composite Pattern?

Composite: Yes. It's a much safer version of the Composite Pattern, but it requires the client to check the type of every object before making a call so the object can be cast correctly.

HeadFirst: Tell us a little more about how these composite and leaf objects are structured.

Composite: Usually it's a tree structure, some kind of hierarchy. The root is the top level composite, and all its children are either composites or leaf nodes.

HeadFirst: Do children ever point back up to their parents?

Composite: Yes, a component can have a pointer to a parent to make traversal of the structure easier. And, if you have a reference to a child, and you need to delete it, you'll need to get the parent to remove the child. Having the parent reference makes that easier too.

HeadFirst: There's really quite a lot to consider in your implementation. Are there other issues we should think about when implementing the Composite Pattern?

Composite: Actually there are... one is the ordering of children. What if you have a composite that needs to keep its children in a particular order? Then you'll need a more sophisticated management scheme for adding and removing children, and you'll have to be careful about how you traverse the hierarchy.

HeadFirst: A good point I hadn't thought of.

Composite: And did you think about caching?

HeadFirst: Caching?

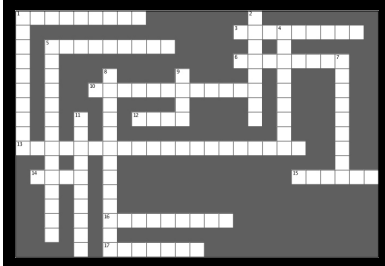
Composite: Yeah, caching. Sometimes, if the composite structure is complex or expensive to traverse, it's helpful to implement caching of the composite nodes. For instance, if you are constantly traversing a composite and all its children to compute some result, you could implement a cache that stores the result temporarily to save traversals.

HeadFirst: Well, there's a lot more to the Composite Patterns than I ever would have guessed. Before we wrap this up, one more question: What do you consider your greatest strength?

Composite: I think I'd definitely have to say simplifying life for my clients. My clients don't have to worry about whether they're dealing with a composite object or a leaf object, so they don't have to write if statements everywhere to make sure they're calling the right methods on the right objects. Often, they can make one method call and execute an operation over an entire structure.

HeadFirst: That does sound like an important benefit. There's no doubt you're a useful pattern to have around for collecting and managing objects. And, with that, we're out of time... Thanks so much for joining us and come back soon for another Patterns Exposed.

It's that time again....



Across	Down
1. User interface packages often use this pattern for their components. 3. Collection and Iterator are in this package 5. We encapsulated this. 6. A separate object that can traverse a collection. 10. Merged with the Diner. 12. Has no children. 13. Name of principle that states only one responsibility per class. 14. Third company acquired. 15. A class should have only one reason to do this. 16. This class indirectly supports Iterator. 17. This menu caused us to change our entire implementation.	1. A composite holds this. 2. We java-enabled her. 4. We deleted PancakeHouseMenuIterator because this class already provides an iterator. 5. The Iterator Pattern decouples the client from the aggregates _____. 7. CompositeIterator used a lot of this. 8. Iterators are usually created using this pattern. 9. A component can be a composite or this. 11. Hashtable and ArrayList both implement this interface.

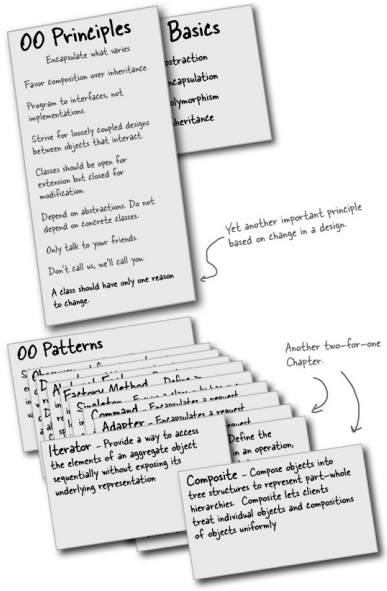
WHO DOES WHAT?

Match each pattern with its description:

Strategy	Clients treat collections of objects and individual objects uniformly
Adapter	Provides a way to traverse a collection of objects without exposing the collection's implementation
Iterator	Simplifies the interface of a group of classes
Facade	Changes the interface of one or more classes
Composite	Allows a group of objects to be notified when some state changes
Observer	Encapsulates interchangeable behaviors and uses delegation to decide which one to use

Tools for your Design Toolbox

Two new patterns for your toolbox – two great ways to deal with collections of objects.



BULLET POINTS

- An Iterator allows access to an aggregate's elements without exposing its internal structure.
- An Iterator takes the job of iterating over an aggregate and encapsulates it in another object.
- When using an Iterator, we relieve the aggregate of the responsibility of supporting operations for traversing its data.
- An Iterator provides a common interface for traversing the items of an aggregate, allowing you to use polymorphism when writing code that makes use of the items of the aggregate.
- We should strive to assign only one responsibility to each class.
- The Composite Pattern provides a structure to hold both individual objects and composites.
- The Composite Pattern allows clients to treat composites and individual objects uniformly.
- A Component is any object in a Composite structure. Components may be other composites or leaf nodes.
- There are many design tradeoffs in implementing Composite. You need to balance transparency and safety with your needs.

Exercise Solutions

SHARPEN YOUR PENCIL

Based on our implementation of printMenu(), which of the following apply?

	A.	We are coding to the PancakeHouseMenu and DinerMenu concrete implementations, not to an interface.
	B.	The Waitress doesn't implement the Java Waitress API and so isn't adhering to a standard.
	C.	If we decided to switch from using DinerMenu to another type of menu that implemented its list of menu items with a Hashtable, we'd have to modify a lot of code in the Waitress.
	D.	The Waitress needs to know how each menu represents its internal collection of menu items is implemented, this violates encapsulation.
	E.	We have duplicate code: the printMenu() method needs two separate loop implementations to iterate over the two different kinds of menus. And if we added a third menu, we might have to add yet another loop.
	F.	The implementation isn't based on MXML (Menu XML) and so isn't as interoperable as it should be.

SHARPEN YOUR PENCIL

Before turning the page, quickly jot down the three things we have to do to this code to fit it into our framework:

1. implement the Menu interface

2. get rid of getItems()

3. add createIterator() and return an Iterator that can step through the Hashtable values

CODE MAGNETS SOLUTION

The unscrambled "Alternating" DinerMenu Iterator

```
import java.util.Iterator;
import java.util.Calendar;

public class AlternatingDinerMenuIterator implements Iterator {
    MenuItem[] items;
    int position;

    public AlternatingDinerMenuIterator(MenuItem[] items) {
        this.items = items;
        Calendar rightNow = Calendar.getInstance();
        position = rightNow.get(Calendar.DAY_OF_WEEK) % 2;
    }

    public boolean hasNext() {
        if (position >= items.length || items[position] == null) {
            return false;
        } else {
            return true;
        }
    }

    public Object next() {
        MenuItem menuItem = items[position];
        position = position + 2;
        return menuItem;
    }

    public void remove() {
        throw new UnsupportedOperationException(
            "Alternating Diner Menu Iterator does not support remove()");
    }
}
```

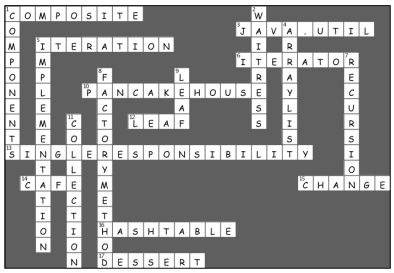
Notice that this Iterator implementation does not support remove()

WHO DOES WHAT?

Match each pattern with its description:

Pattern	Description
State	Clients treat collections of objects and individual objects uniformly
Adapter	Provides a way to traverse a collection of objects without exposing the collection's implementation
Iterator	Simplifies the interface of a group of classes
Facade	Changes the interface of one or more classes
Composite	Allows a group of objects to be notified when some state changes
Observer	Allows an object to change its behavior when some state changes

Exercise solutions



PREV
8. The Template Method Pattern: Encapsulating Algorithms

NEXT
10. The State Pattern: The State of Things

Terms of Service / Privacy Policy