Keeping your method calls in bounds...

Here's a Car class that demonstrates all the ways you can call methods and still adhere to the Principle of Least Knowledge:

```
public class Car {
        Engine engine;
        // other instance variables

        public Car() {
                // initialize engine, etc.
        }

        public void start(Key key) {
                Doors doors = new Doors();

                boolean authorized = key.turns();

                if (authorized) {
                        engine.start();
                        updateDashboardDisplay();
                        doors.lock();
                }
        }

        public void updateDashboardDisplay() {
                // update display
        }
}
```

Here's a component of this class. We can call its methods.

Here we're creating a new object, its methods are legal.

You can call a method on an object passed as a parameter.

You can call a method on a component of the object.

You can call a local method within the object.

You can call a method on an object you create or instantiate.

there are no Dumb Questions

Q: There is another principle called the Law of Demeter; how are they related?

A: The two are one and the same and you'll encounter these terms being intermixed. We prefer to use the Principle of Least Knowledge for a couple of reasons: (1) the name is more intuitive and (2) the use of the word "Law" implies we always have to apply this principle. In fact, no principle is a law, all principles should be used when and where they are helpful. All design involves tradeoffs (abstractions versus speed, space versus time, and so on) and while principles provide guidance, all factors should be taken into account before applying them.

Q: Are there any disadvantages to applying the Principle of Least Knowledge?

A: Yes; while the principle reduces the dependencies between objects and studies have shown this reduces software maintenance, it is also the case that applying this principle results in more "wrapper" classes being written to handle method calls to other components. This can result in increased complexity and development time as well as decreased runtime performance.

Sharpen your pencil

Do either of these classes violate the Principle of Least Knowledge? Why or why not?

```
public House {
    WeatherStation station;

    // other methods and constructor

    public float getTemp() {
         return station.getThermometer().getTemperature();
    }
}
```

```
public House {
     WeatherStation station;

    // other methods and constructor

    public float getTemp() {
         Thermometer thermometer = station.getThermometer();
         return getTempHelper(thermometer);
    }

    public float getTempHelper(Thermometer thermometer) {
        return thermometer.getTemperature();
    }
}
```

HARD HAT AREA. WATCH OUT FOR FALLING ASSUMPTIONS

BRAIN POWER

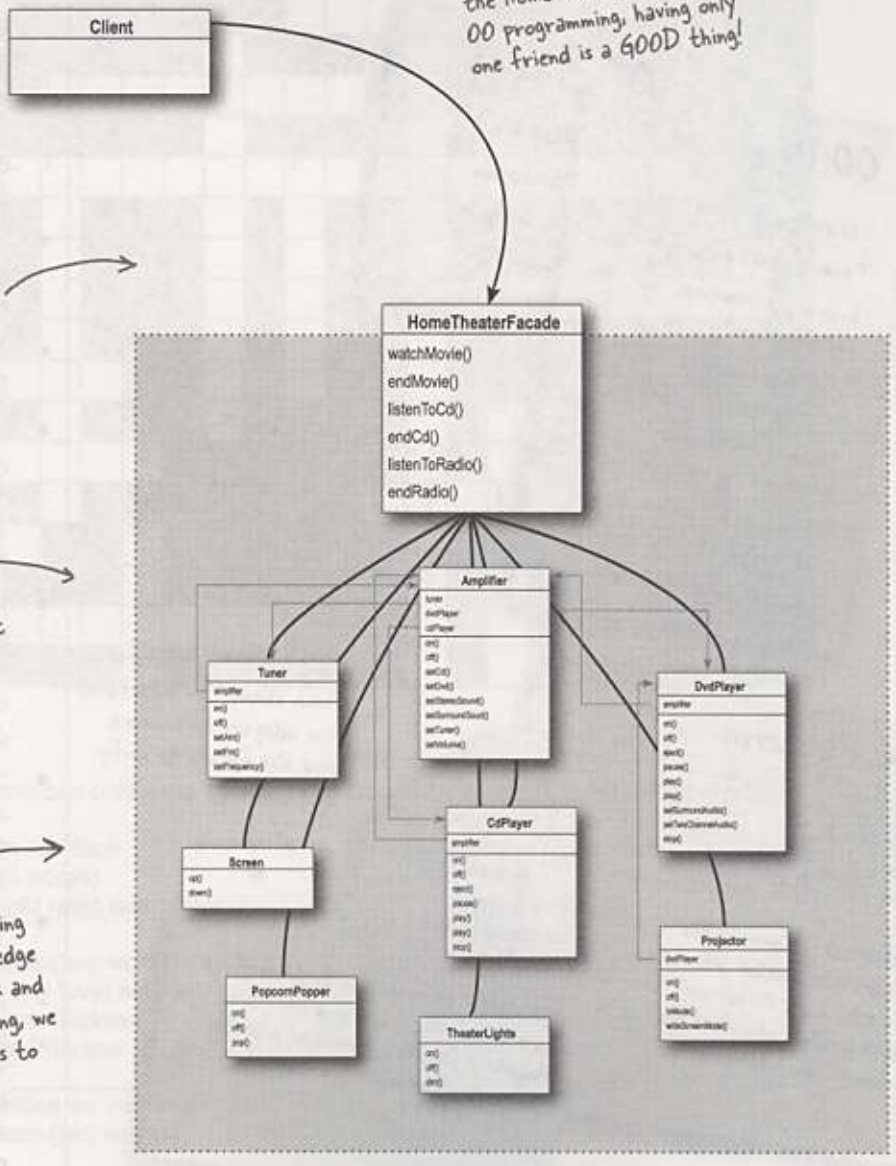
Can you think of a common use of Java that violates the Principle of Least Knowledge?

Should you care?

Answer: How about System.out.println()?

The Facade and the Principle of Least Knowledge

This client only has one friend; the HomeTheaterFacade. In OO programming, having only one friend is a GOOD thing!



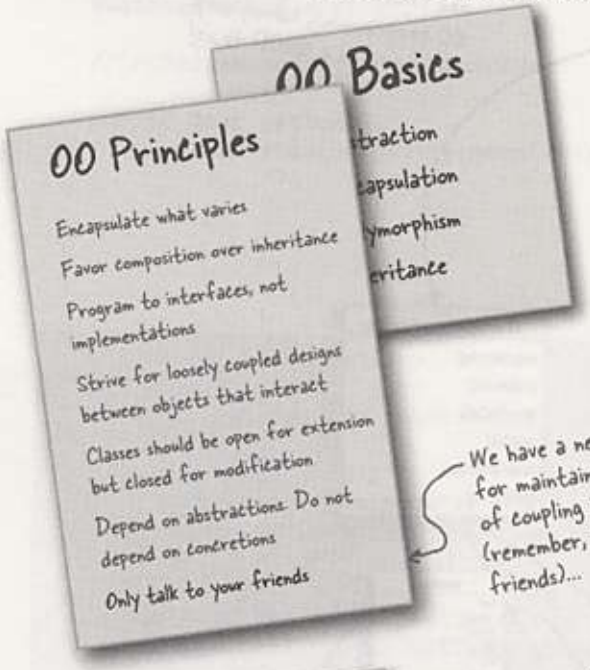
The HomeTheaterFacade manages all those subsystem components for the client. It keeps the client simple and flexible.

We can upgrade the home theater components without affecting the client.

We try to keep subsystems adhering to the Principle of Least Knowledge as well. If this gets too complex and too many friends are intermingling, we can introduce additional facades to form layers of subsystems.

Tools for your Design Toolbox

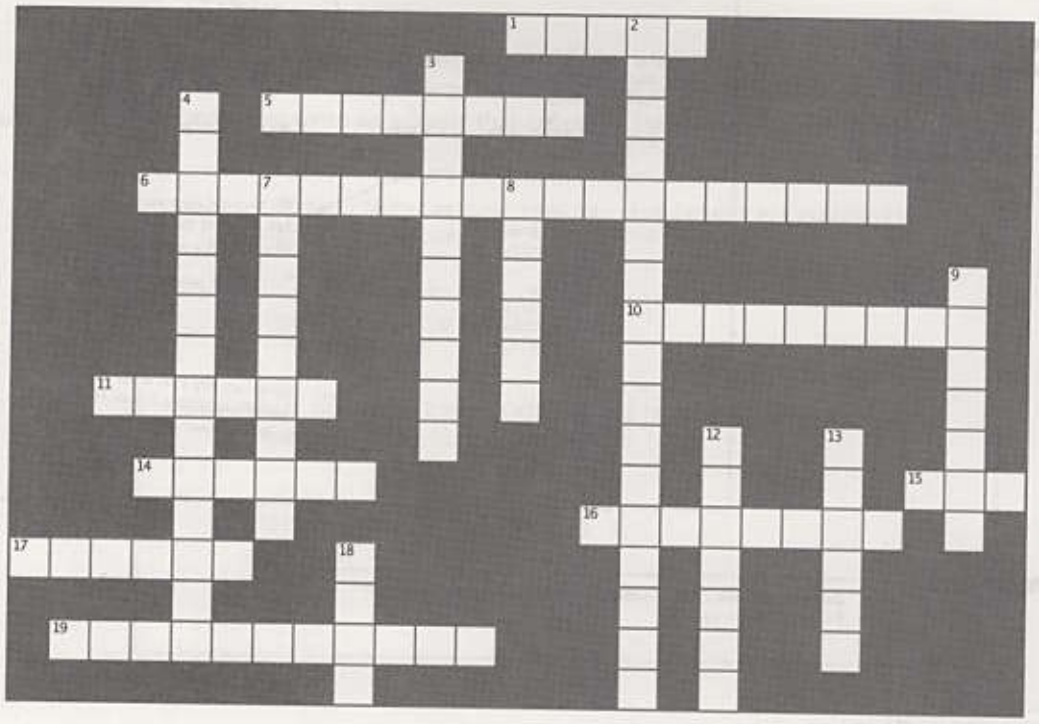
Your toolbox is starting to get heavy! In this chapter we've added a couple of patterns that allow us to alter interfaces and reduce coupling between clients and the systems they use.



BULLET POINTS

- When you need to use an existing class and its interface is not the one you need, use an adapter.
- When you need to simplify and unify a large interface or complex set of interfaces, use a facade.
- An adapter changes an interface into one a client expects.
- A facade decouples a client from a complex subsystem.
- Implementing an adapter may require little work or a great deal of work depending on the size and complexity of the target interface.
- Implementing a facade requires that we compose the facade with its subsystem and use delegation to perform the work of the facade.
- There are two forms of the Adapter Pattern: object and class adapters. Class adapters require multiple inheritance.
- You can implement more than one facade for a subsystem.
- An adapter wraps an object to change its interface, a decorator wraps an object to add new behaviors and responsibilities, and a facade "wraps" a set of objects to simplify.

Yes, it's another crossword. All of the solution words are from this chapter.



Across

1. True or false, Adapters can only wrap one object
5. An Adapter _____ an interface
6. Movie we watched (5 words)
10. If in Europe you might need one of these (two words)
11. Adapter with two roles (two words)
14. Facade still _____ low level access
15. Ducks do it better than Turkeys
16. Disadvantage of the Principle of Least Knowledge: too many _____
17. A _____ simplifies an interface
19. New American dream (two words)

Down

2. Decorator called Adapter this (3 words)
3. One advantage of Facade
4. Principle that wasn't as easy as it sounded (two words)
7. A _____ adds new behavior
8. Masquerading as a Duck
9. Example that violates the Principle of Least Knowledge: System.out._____
12. No movie is complete without this
13. Adapter client uses the _____ interface
18. An Adapter and a Decorator can be said to _____ an object

Exercise solutions

Sharpen your pencil

Let's say we also need an adapter that converts a Duck to a Turkey. Let's call it DuckAdapter. Write that class:

```
public class DuckAdapter implements Turkey {
    Duck duck;
    Random rand;

    public DuckAdapter(Duck duck) {
        this.duck = duck;
        rand = new Random();
    }

    public void gobble() {
        duck.quack();
    }

    public void fly() {
        if (rand.nextInt(5) == 0) {
            duck.fly();
        }
    }
}
```

Now we are adapting Turkeys to Ducks, so we implement the Turkey interface

We stash a reference to the Duck we are adapting

We also recreate a random object; take a look at the fly() method to see how it is used.

A gobble just becomes a quack.

Since ducks fly a lot longer than turkeys, we decided to only fly the duck on average one of five times.

Sharpen your pencil

Do either of these classes violate the Principle of Least Knowledge? For each, why or why not?

```
public House {
    WeatherStation station;

    // other methods and constructor

    public float getTemp() {
        return station.getThermometer().getTemperature();
    }
}
```

Violates the Principle of Least Knowledge! You are calling the method of an object returned from another call.

```
public House {
    WeatherStation station;

    // other methods and constructor

    public float getTemp() {
        Thermometer thermometer = station.getThermometer();
        return getTempHelper(thermometer);
    }

    public float getTempHelper(Thermometer thermometer) {
        return thermometer.getTemperature();
    }
}
```

Doesn't violate Principle of Least Knowledge! This seems like hacking our way around the principle. Has anything really changed since we just moved out the call to another method?

Exercise solutions

You've seen how to implement an adapter that adapts an Enumeration to an Iterator; now write an adapter that adapts an Iterator to an Enumaration.

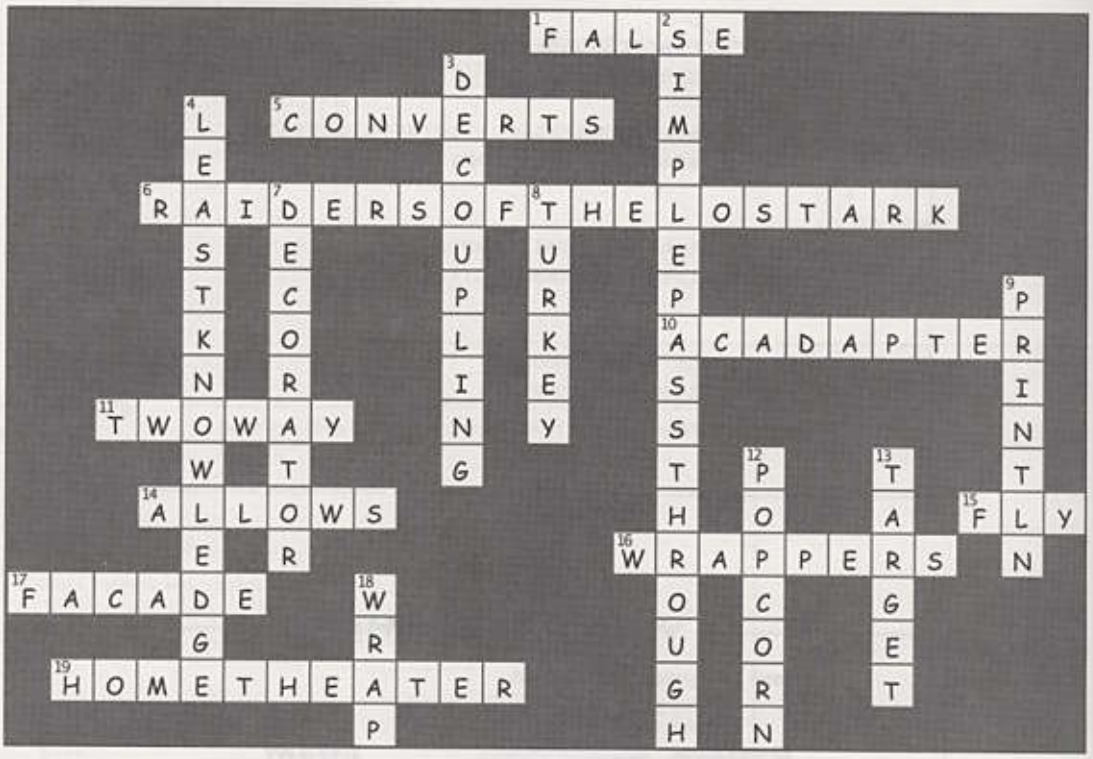
```
public class IteratorEnumeration implements Enumeration {
    Iterator iterator;

    public IteratorEnumeration(Iterator iterator) {
        this.iterator = iterator;
    }

    public boolean hasMoreElements() {
        return iterator.hasNext();
    }

    public Object nextElement() {
        return iterator.next();
    }
}
```

WHO DOES WHAT?

Match each pattern with its intent:

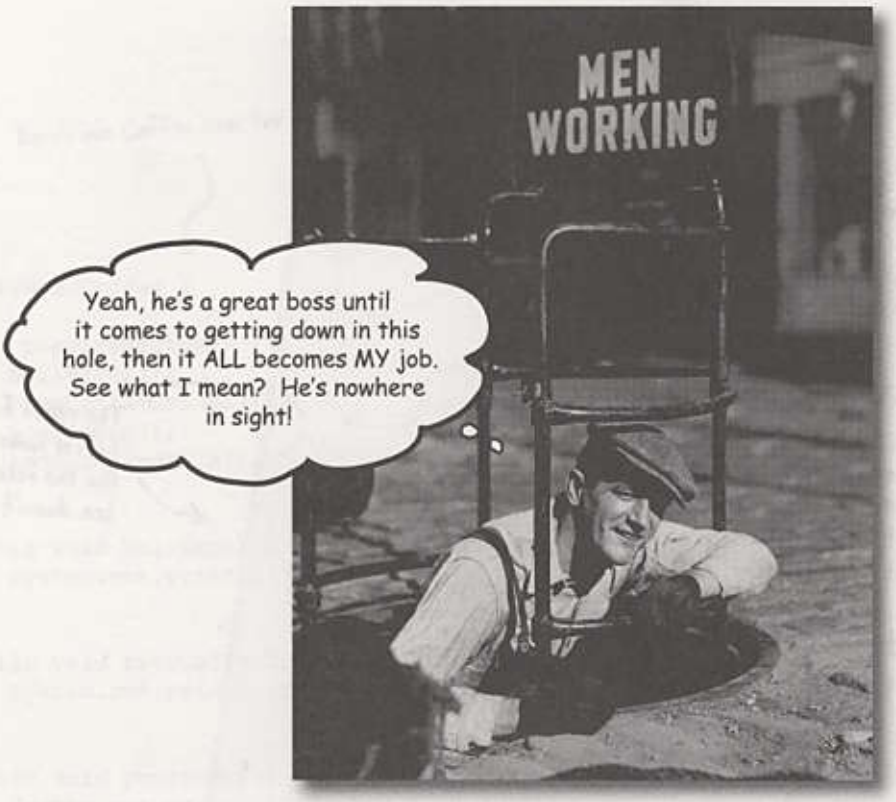
Pattern	Intent
Decorator	Convert one interface to another
Adapter	Don't alter interface, but add responsibility
Facade	Make interface simpler

Exercise solutions



8 the Template Method Pattern

Encapsulating Algorithms

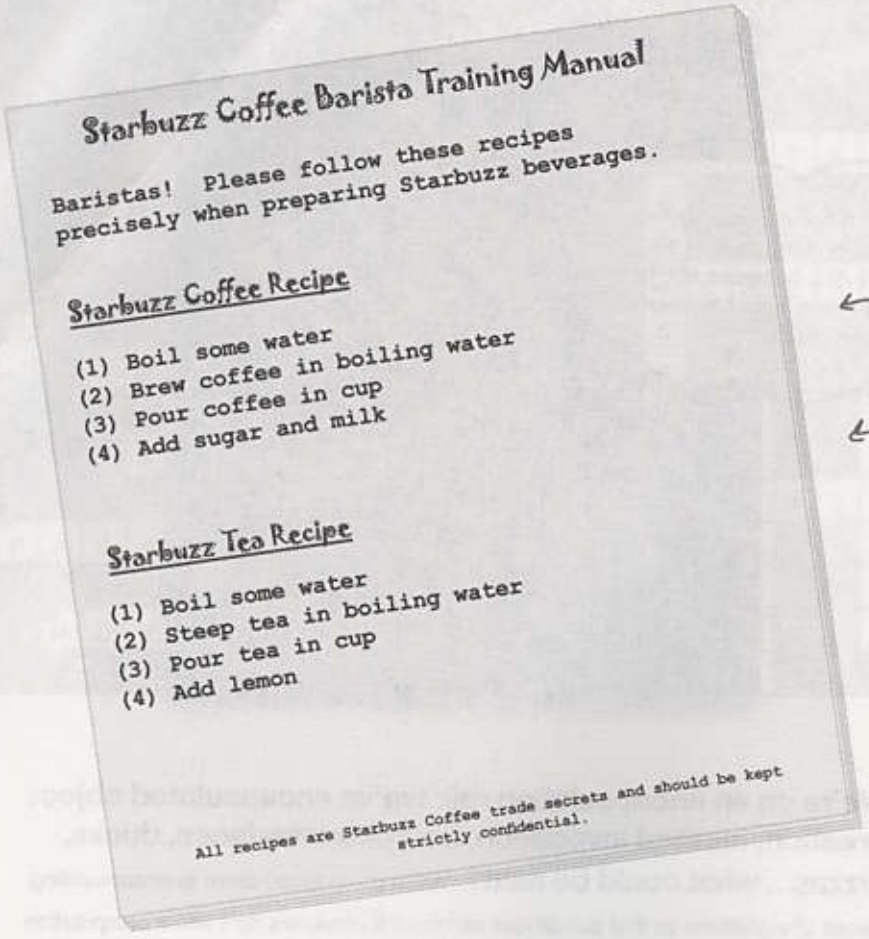


We're on an encapsulation roll; we've encapsulated object creation, method invocation, complex interfaces, ducks, pizzas... what could be next? We're going to get down to encapsulating pieces of algorithms so that subclasses can hook themselves right into a computation anytime they want. We're even going to learn about a design principle inspired by Hollywood.

It's time for some more caffeine

Some people can't live without their coffee; some people can't live without their tea. The common ingredient? Caffeine of course!

But there's more; tea and coffee are made in very similar ways. Let's check it out:



The recipe for coffee looks a lot like the recipe for tea, doesn't it?

Whipping up some coffee and tea classes (in Java)

Let's play "coding barista" and write some code for creating coffee and tea.

Here's the coffee:

Here's our Coffee class for making coffee.

Here's our recipe for coffee, straight out of the training manual.

Each of the steps is implemented as a separate method.

```
public class Coffee {

    void prepareRecipe() {
        boilWater();
        brewCoffeeGrinds();
        pourInCup();
        addSugarAndMilk();
    }

    public void boilWater() {
        System.out.println("Boiling water");
    }

    public void brewCoffeeGrinds() {
        System.out.println("Dripping Coffee through filter");
    }

    public void pourInCup() {
        System.out.println("Pouring into cup");
    }

    public void addSugarAndMilk() {
        System.out.println("Adding Sugar and Milk");
    }
}
```

Each of these methods implements one step of the algorithm. There's a method to boil water, brew the coffee, pour the coffee in a cup and add sugar and milk.

and now the Tea...

```
public class Tea {

    void prepareRecipe() {
        boilWater();
        steepTeaBag();
        pourInCup();
        addLemon();
    }

    public void boilWater() {
        System.out.println("Boiling water");
    }

    public void steepTeaBag() {
        System.out.println("Steeping the tea");
    }

    public void addLemon() {
        System.out.println("Adding Lemon");
    }

    public void pourInCup() {
        System.out.println("Pouring into cup");
    }
}
```

This looks very similar to the one we just implemented in Coffee; the second and forth steps are different, but it's basically the same recipe.

These two methods are specialized to Tea.

Notice that these two methods are exactly the same as they are in Coffee! So we definitely have some code duplication going on here.

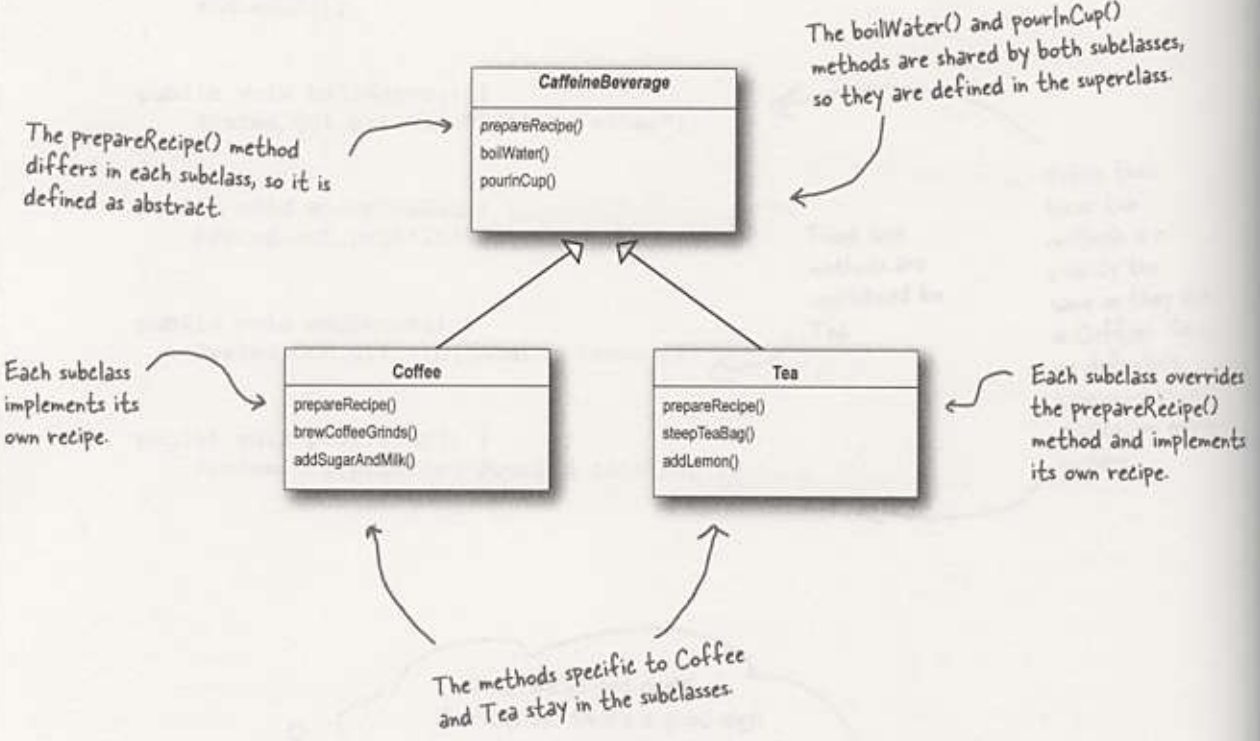
When we've got code duplication, that's a good sign we need to clean up the design. It seems like here we should abstract the commonality into a base class since coffee and tea are so similar?

Design Puzzle

You've seen that the Coffee and Tea classes have a fair bit of code duplication. Take another look at the Coffee and Tea classes and draw a class diagram showing how you'd redesign the classes to remove redundancy:

Sir, may I abstract your Coffee, Tea?

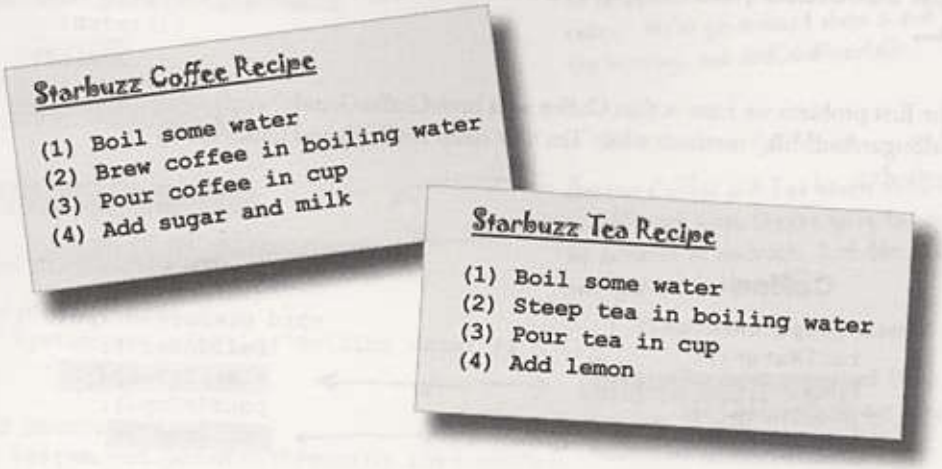
It looks like we've got a pretty straightforward design exercise on our hands with the Coffee and Tea classes. Your first cut might have looked something like this:



Did we do a good job on the redesign? Hmmmm, take another look. Are we overlooking some other commonality? What are other ways that Coffee and Tea are similar?

Taking the design further...

So what else do Coffee and Tea have in common? Let's start with the recipes.



Notice that both recipes follow the same algorithm:

1. Boil some water.
2. Use the hot water to extract the coffee or tea.
3. Pour the resulting beverage into a cup.
4. Add the appropriate condiments to the beverage.

These aren't abstracted, but are the same, they just apply to different beverages.

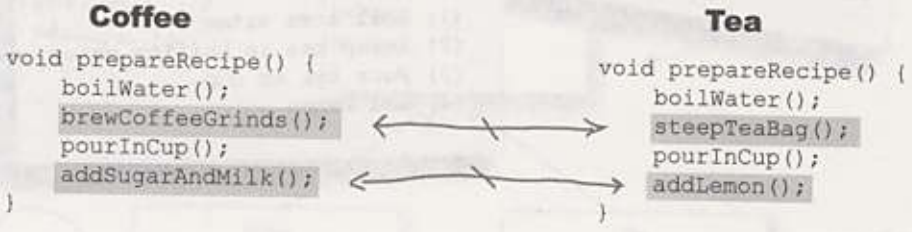
These two are already abstracted into the base class.

So, can we find a way to abstract prepareRecipe() too? Yes, let's find out...

Abstracting prepareRecipe()

Let's step through abstracting prepareRecipe() from each subclass (that is, the Coffee and Tea classes)...

1. The first problem we have is that Coffee uses brewCoffeeGrinds() and addSugarAndMilk() methods while Tea uses steepTeaBag() and addLemon() methods.



Let's think through this: steeping and brewing aren't so different; they're pretty analogous. So let's make a new method name, say, brew(), and we'll use the same name whether we're brewing coffee or steeping tea.

Likewise, adding sugar and milk is pretty much the same as adding a lemon: both are adding condiments to the beverage. Let's also make up a new method name, addCondiments(), to handle this. So, our new prepareRecipe() method will look like this:

```
void prepareRecipe() {
    boilWater();
    brew();
    pourInCup();
    addCondiments();
}
```

2. Now we have a new prepareRecipe() method, but we need to fit it into the code. To do this we are going to start with the CaffeineBeverage superclass:

CaffeineBeverage is abstract, just like in the class design.

```
public abstract class CaffeineBeverage {

    final void prepareRecipe() {
        boilWater();
        brew();
        pourInCup();
        addCondiments();
    }

    abstract void brew();

    abstract void addCondiments();

    void boilWater() {
        System.out.println("Boiling water");
    }

    void pourInCup() {
        System.out.println("Pouring into cup");
    }
}
```

Now, the same prepareRecipe() method will be used to make both Tea and Coffee. prepareRecipe() is declared final because we don't want our subclasses to be able to override this method and change the recipe! We've generalized steps 2 and 4 to brew() the beverage and addCondiments().

Because Coffee and Tea handle these methods in different ways, they're going to have to be declared as abstract. Let the subclasses worry about that stuff!

Remember, we moved these into the CaffeineBeverage class (back in our class diagram).

3 Finally we need to deal with the Coffee and Tea classes. They now rely on CaffeineBeverage to handle the recipe, so they just need to handle brewing and condiments:

As in our design, Tea and Coffee now extend CaffeineBeverage.

```
public class Tea extends CaffeineBeverage {
    public void brew() {
        System.out.println("Steeping the tea");
    }
    public void addCondiments() {
        System.out.println("Adding Lemon");
    }
}
```

Tea needs to define brew() and addCondiments() — the two abstract methods from Beverage.

Same for Coffee, except Coffee deals with coffee, and sugar and milk instead of tea bags and lemon.

```
public class Coffee extends CaffeineBeverage {
    public void brew() {
        System.out.println("Dripping Coffee through filter");
    }
    public void addCondiments() {
        System.out.println("Adding Sugar and Milk");
    }
}
```

Sharpen your pencil

Draw the new class diagram now that we've moved the implementation of prepareRecipe() into the CaffeineBeverage class:

What have we done?

We've recognized that the two recipes are essentially the same, although some of the steps require different implementations. So we've generalized the recipe and placed it in the base class.

Tea

1. Boil some water
2. Steep the teabag in the water
3. Pour tea in a cup
4. Add lemon

Coffee

1. Boil some water
2. Brew the coffee grinds
3. Pour coffee in a cup
4. Add sugar and milk

Caffeine Beverage

1. Boil some water
2. Brew
3. Pour beverage in a cup
4. Add condiments

generalize

generalize

relies on subclass for some steps

relies on subclass for some steps

Tea subclass

Coffee subclass

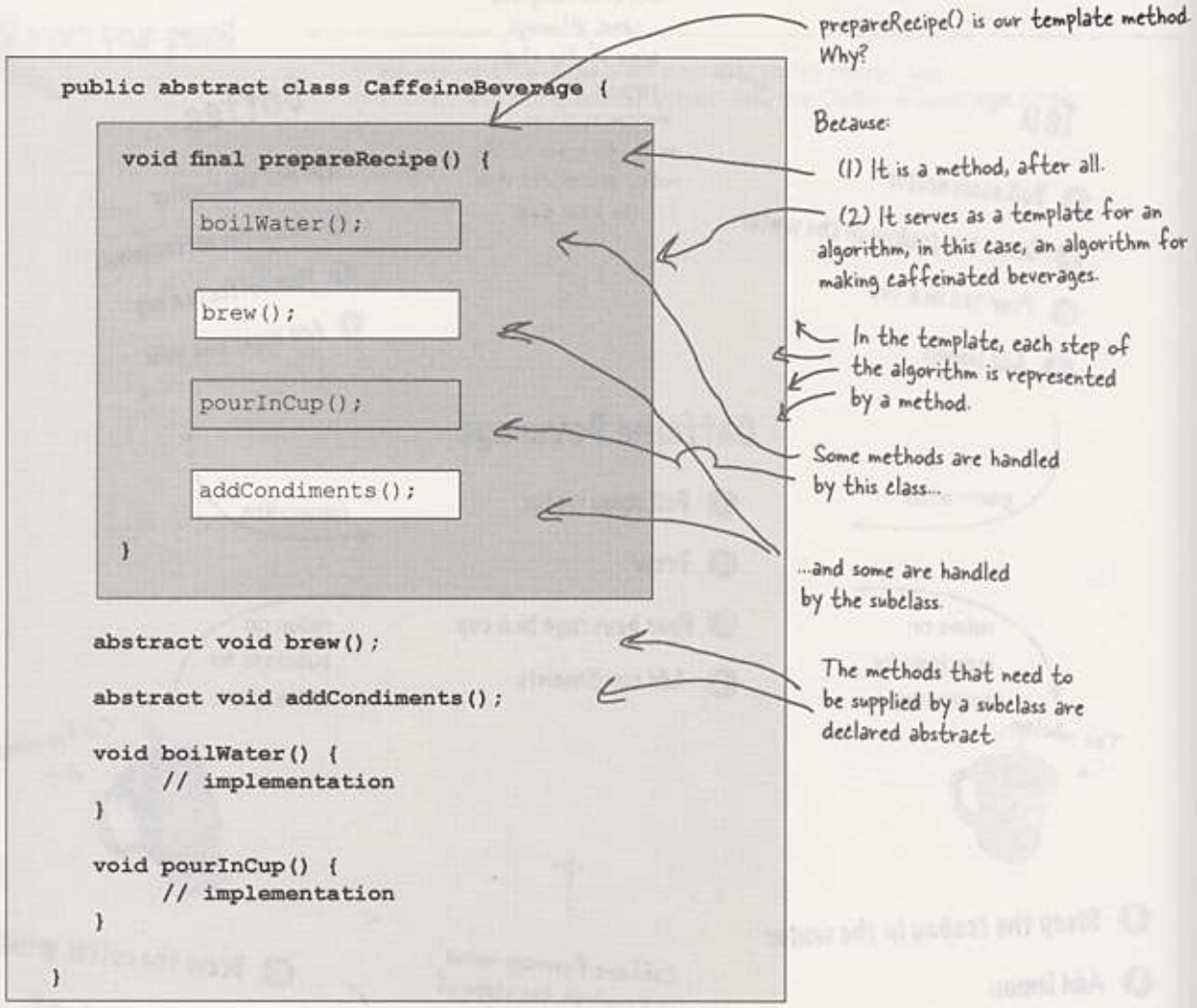
2. Steep the teabag in the water
4. Add lemon

2. Brew the coffee grinds
4. Add sugar and milk

Caffeine Beverage knows and controls the steps of the recipe, and performs steps 1 and 3 itself, but relies on Tea or Coffee to do steps 2 and 4.

Meet the Template Method

We've basically just implemented the Template Method Pattern. What's that? Let's look at the structure of the CaffeineBeverage class; it contains the actual "template method:"



The Template Method defines the steps of an algorithm and allows subclasses to provide the implementation for one or more steps.

Let's make some tea...

Behind the Scenes

Let's step through making a tea and trace through how the template method works. You'll see that the template method controls the algorithm; at certain points in the algorithm, it lets the subclass supply the implementation of the steps...

1. Okay, first we need a Tea object...

```
Tea myTea = new Tea();
```

2. Then we call the template method:

```
myTea.prepareRecipe();
```

which follows the algorithm for making caffeine beverages...

3. First we boil water:

```
boilWater();
```

which happens in CaffeineBeverage.

4. Next we need to brew the tea, which only the subclass knows how to do:

```
brew();
```

5. Now we pour the tea in the cup; this is the same for all beverages so it happens in CaffeineBeverage:

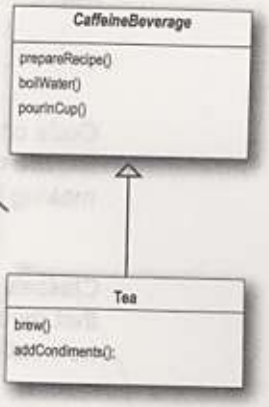
```
pourInCup();
```

6. Finally, we add the condiments, which are specific to each beverage, so the subclass implements this:

```
addCondiments();
```

```
boilWater();
brew();
pourInCup();
addCondiments();
```

The prepareRecipe() method controls the algorithm, no one can change this, and it counts on subclasses to provide some or all of the implementation.



What did the Template Method get us?

Underpowered Tea & Coffee implementation

New, hip CaffeineBeverage powered by Template Method

Coffee and Tea are running the show; they control the algorithm.	The CaffeineBeverage class runs the show; it has the algorithm, and protects it.
Code is duplicated across Coffee and Tea.	The CaffeineBeverage class maximizes reuse among the subclasses.
Code changes to the algorithm require opening the subclasses and making multiple changes.	The algorithm lives in one place and code changes only need to be made there.
Classes are organized in a structure that requires a lot of work to add a new caffeine beverage.	The Template Method version provides a framework that other caffeine beverages can be plugged into. New caffeine beverages only need to implement a couple of methods.
Knowledge of the algorithm and how to implement it is distributed over many classes.	The CaffeineBeverage class concentrates knowledge about the algorithm and relies on subclasses to provide complete implementations.

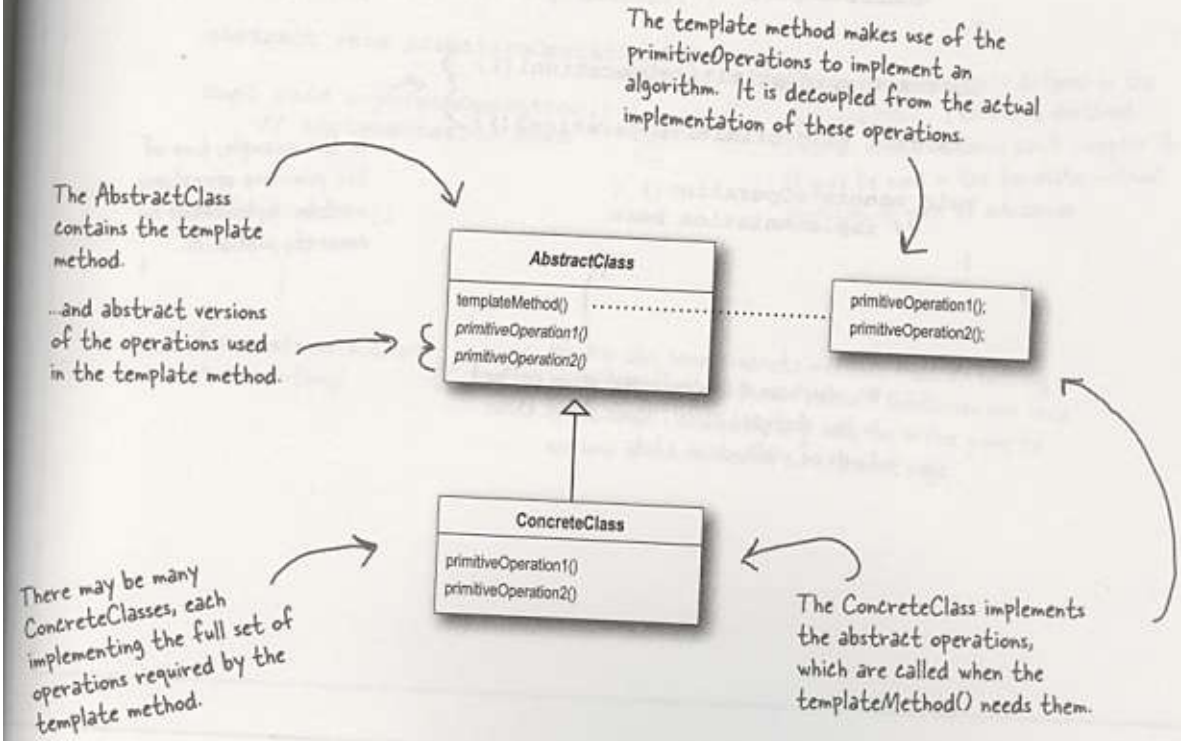
Template Method Pattern defined

You've seen how the Template Method Pattern works in our Tea and Coffee example; now, check out the official definition and nail down all the details:

> **The Template Method Pattern** defines the skeleton of an algorithm in a method, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

This pattern is all about creating a template for an algorithm. What's a template? As you've seen it's just a method; more specifically, it's a method that defines an algorithm as a set of steps. One or more of these steps is defined to be abstract and implemented by a subclass. This ensures the algorithm's structure stays unchanged, while subclasses provide some part of the implementation.

Let's check out the class diagram:



Code Up Close

Let's take a closer look at how the AbstractClass is defined, including the template method and primitive operations.

Here we have our abstract class; it is declared abstract and meant to be subclassed by classes that provide implementations of the operations.

Here's the template method. It's declared final to prevent subclasses from reworking the sequence of steps in the algorithm.

```
abstract class AbstractClass {

    final void templateMethod() {
        primitiveOperation1();
        primitiveOperation2();
        concreteOperation();
    }

    abstract void primitiveOperation1();

    abstract void primitiveOperation2();

    void concreteOperation() {
        // implementation here
    }
}
```

The template method defines the sequence of steps, each represented by a method.

In this example, two of the primitive operations must be implemented by concrete subclasses.

We also have a concrete operation defined in the abstract class. More about these kinds of methods in a bit...

Code Way Up Close

Now we're going to look even closer at the types of method that can go in the abstract class:

We've changed the templateMethod() to include a new method call.

```
abstract class AbstractClass {

    final void templateMethod() {
        primitiveOperation1();
        primitiveOperation2();
        concreteOperation();
        hook();
    }

    abstract void primitiveOperation1();

    abstract void primitiveOperation2();

    final void concreteOperation() {
        // implementation here
    }

    void hook() {}

}
```

We still have our primitive methods; these are abstract and implemented by concrete subclasses.

A concrete operation is defined in the abstract class. This one is declared final so that subclasses can't override it. It may be used in the template method directly, or used by subclasses.

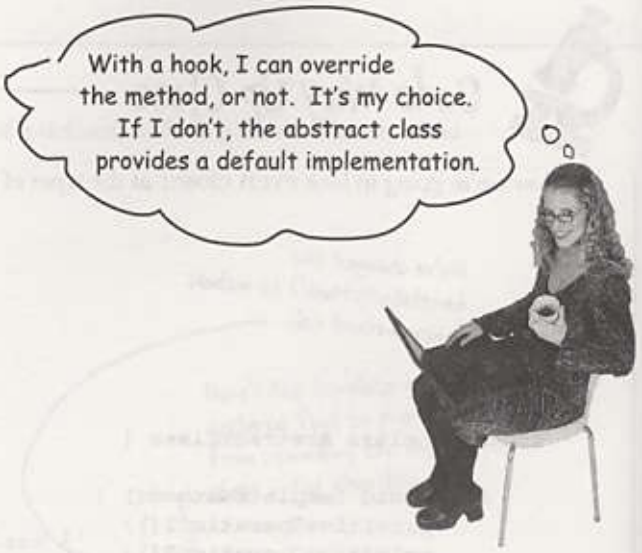
A concrete method, but it does nothing!

We can also have concrete methods that do nothing by default; we call these "hooks." Subclasses are free to override these but don't have to. We're going to see how these are useful on the next page.

Hooked on Template Method...

A hook is a method that is declared in the abstract class, but only given an empty or default implementation. This gives subclasses the ability to "hook into" the algorithm at various points, if they wish; a subclass is also free to ignore the hook.

There are several uses of hooks; let's take a look at one now. We'll talk about a few other uses later:




```
public abstract class CaffeineBeverageWithHook {

    void prepareRecipe() {
        boilWater();
        brew();
        pourInCup();
        if (customerWantsCondiments()) {
            addCondiments();
        }
    }

    abstract void brew();

    abstract void addCondiments();

    void boilWater() {
        System.out.println("Boiling water");
    }

    void pourInCup() {
        System.out.println("Pouring into cup");
    }

    boolean customerWantsCondiments() {
        return true;
    }
}
```

We've added a little conditional statement that bases its success on a concrete method, customerWantsCondiments(). If the customer WANTS a condiments, only then do we call addCondiments().

Here we've defined a method with a (mostly) empty default implementation. This method just returns true and does nothing else.

This is a **hook** because the subclass can override this method, but doesn't have to.

Using the hook

To use the hook, we override it in our subclass. Here, the hook controls whether the CaffeineBeverage evaluates a certain part of the algorithm; that is, whether it adds a condiment to the beverage.

How do we know whether the customer wants the condiment? Just ask!

```
public class CoffeeWithHook extends CaffeineBeverageWithHook {

    public void brew() {
        System.out.println("Dripping Coffee through filter");
    }

    public void addCondiments() {
        System.out.println("Adding Sugar and Milk");
    }

    public boolean customerWantsCondiments() {
        String answer = getUserInput();

        if (answer.toLowerCase().startsWith("y")) {
            return true;
        } else {
            return false;
        }
    }

    private String getUserInput() {
        String answer = null;

        System.out.print("Would you like milk and sugar with your coffee (y/n)? ");

        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
        try {
            answer = in.readLine();
        } catch (IOException ioe) {
            System.err.println("IO error trying to read your answer");
        }
        if (answer == null) {
            return "no";
        }
        return answer;
    }
}
```

Here's where you override the hook and provide your own functionality.

Get the user's input on the condiment decision and return true or false depending on the input.

This code asks the user if he'd like milk and sugar and gets his input from the command line.

Let's run the TestDrive

Okay, the water's boiling... Here's the test code where we create a hot tea and a hot coffee

```
public class BeverageTestDrive {
    public static void main(String[] args) {

        TeaWithHook teaHook = new TeaWithHook();
        CoffeeWithHook coffeeHook = new CoffeeWithHook();

        System.out.println("\nMaking tea...");
        teaHook.prepareRecipe();

        System.out.println("\nMaking coffee...");
        coffeeHook.prepareRecipe();
    }
}
```

← Create a tea

← A coffee

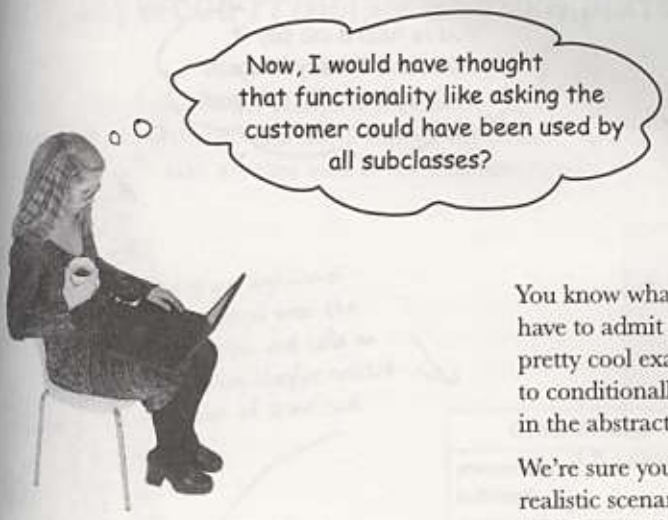
And call prepareRecipe() on both!

And let's give it a run...

```
File Edit Window Help send-more-honestlea
%java BeverageTestDrive

Making tea...
Boiling water
Steeping the tea
Pouring into cup
Would you like lemon with your tea (y/n)? y
Adding Lemon

Making coffee...
Boiling water
Dripping Coffee through filter
Pouring into cup
Would you like milk and sugar with your coffee (y/n)? n
%
```

A steaming cup of tea, and yes, of course we want that lemon!

And a nice hot cup of coffee, but we'll pass on the waistline expanding condiments.



You know what? We agree with you. But you have to admit before you thought of that it was a pretty cool example of how a hook can be used to conditionally control the flow of the algorithm in the abstract class. Right?

We're sure you can think of many other more realistic scenarios where you could use the template method and hooks in your own code.

there are no Dumb Questions

Q: When I'm creating a template method, how do I know when to use abstract methods and when to use hooks?

A: Use abstract methods when your subclass MUST provide an implementation of the method or step in the algorithm. Use hooks when that part of the algorithm is optional. With hooks, a subclass may choose to implement that hook, but it doesn't have to.

Q: What are hooks really supposed to be used for?

A: There are a few uses of hooks. As we just said, a hook may provide a way for a subclass to implement an optional part of an algorithm, or if it isn't important to the subclass' implementation, it can skip it. Another use is to give the subclass a chance to react to some step in the template method that is about to happen, or just happened. For instance, a hook method like justReOrderedList() allows the subclass to perform some activity (such as redisplaying an onscreen representation) after an internal list is reordered. As you've seen a hook can also provide a subclass with the ability to make a decision for the abstract class.

Q: Does a subclass have to implement all the abstract methods in the AbstractClass?

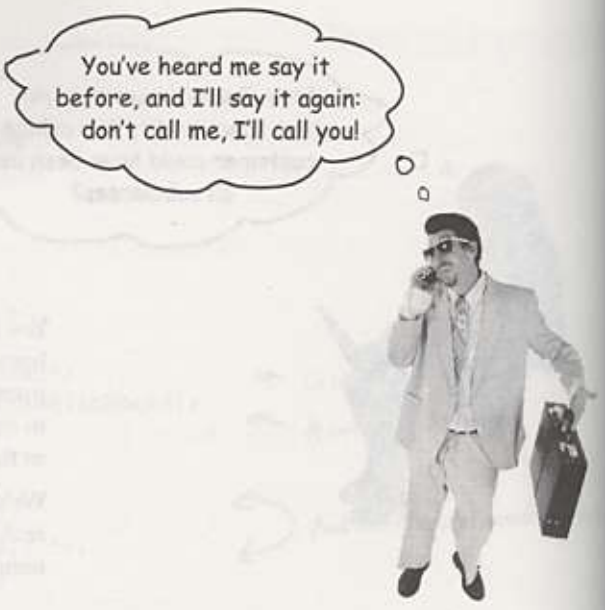
A: Yes, each concrete subclass defines the entire set of abstract methods and provides a complete implementation of the undefined steps of the template method's algorithm.

Q: It seems like I should keep my abstract methods small in number, otherwise it will be a big job to implement them in the subclass.

A: That's a good thing to keep in mind when you write template methods. Sometimes this can be done by not making the steps of your algorithm too granular. But it's obviously a trade off: the less granularity, the less flexibility.

Remember, too, that some steps will be optional; so you can implement these as hooks rather than abstract classes, easing the burden on the subclasses of your abstract class.

The Hollywood Principle



We've got another design principle for you; it's called the Hollywood Principle:

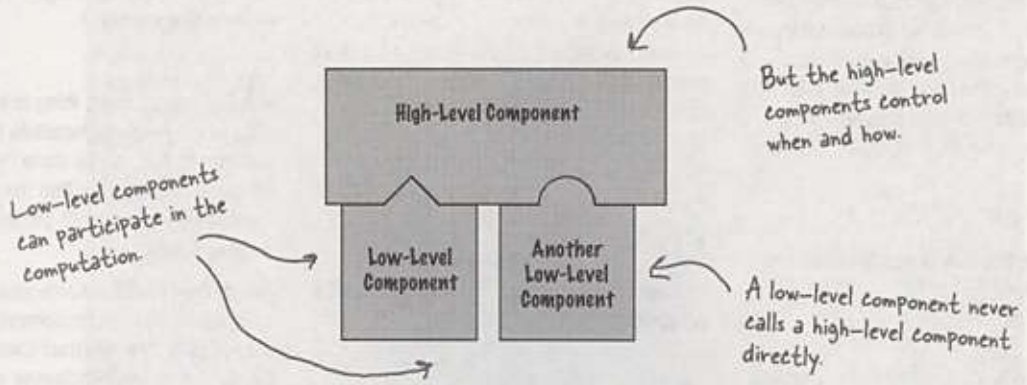
The Hollywood Principle

Don't call us, we'll call you.

Easy to remember, right? But what has it got to do with OO design?

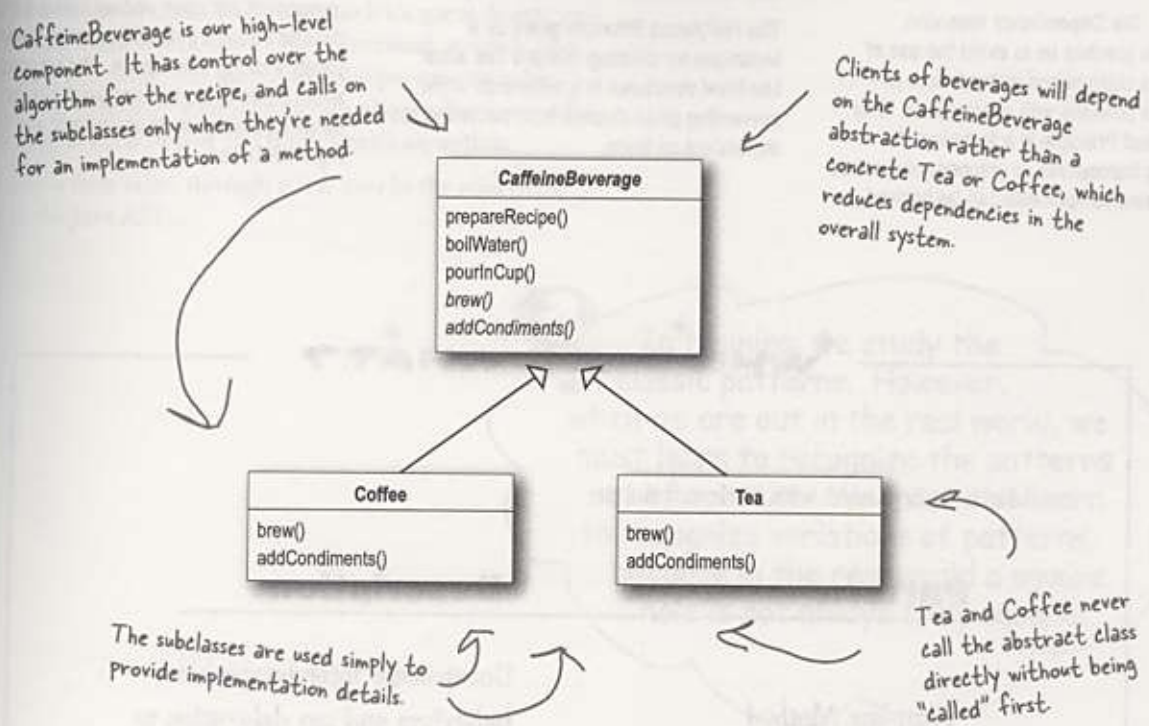
The Hollywood principle gives us a way to prevent "dependency rot." Dependency rot happens when you have high-level components depending on low-level components depending on high-level components depending on sideways components depending on low-level components, and so on. When rot sets in, no one can easily understand the way a system is designed.

With the Hollywood Principle, we allow low-level components to hook themselves into a system, but the high-level components determine when they are needed, and how. In other words, the high-level components give the low-level components a "don't call us, we'll call you" treatment.



The Hollywood Principle and Template Method

The connection between the Hollywood Principle and the Template Method Pattern is probably somewhat apparent: when we design with the Template Method Pattern, we're telling subclasses, "don't call us, we'll call you." How? Let's take another look at our CaffeineBeverage design:



BRAIN POWER

What other patterns make use of the Hollywood Principle?

The Factory Method, Observer, any others?

there are no Dumb Questions

Q: How does the Hollywood Principle relate to the Dependency Inversion Principle that we learned a few chapters back?

A: The Dependency Inversion Principle teaches us to avoid the use of concrete classes and instead work as much as possible with abstractions. The Hollywood Principle is a technique for building frameworks or components so that lower-level components can be hooked into the computation, but without creating dependencies between the lower-level components and the higher-level layers. So, they both have the goal of decoupling, but the Dependency Inversion Principle makes a much stronger and general statement about how to avoid dependencies in design.

The Hollywood Principle gives us a technique for creating designs that allow low-level structures to interoperate while preventing other classes from becoming too dependent on them.

Q: Is a low-level component disallowed from calling a method in a higher-level component?

A: Not really. In fact, a low level component will often end up calling a method defined above it in the inheritance hierarchy purely through inheritance. But we want to avoid creating explicit circular dependencies between the low-level component and the high-level ones.

WHO DOES WHAT?

Match each pattern with its description:

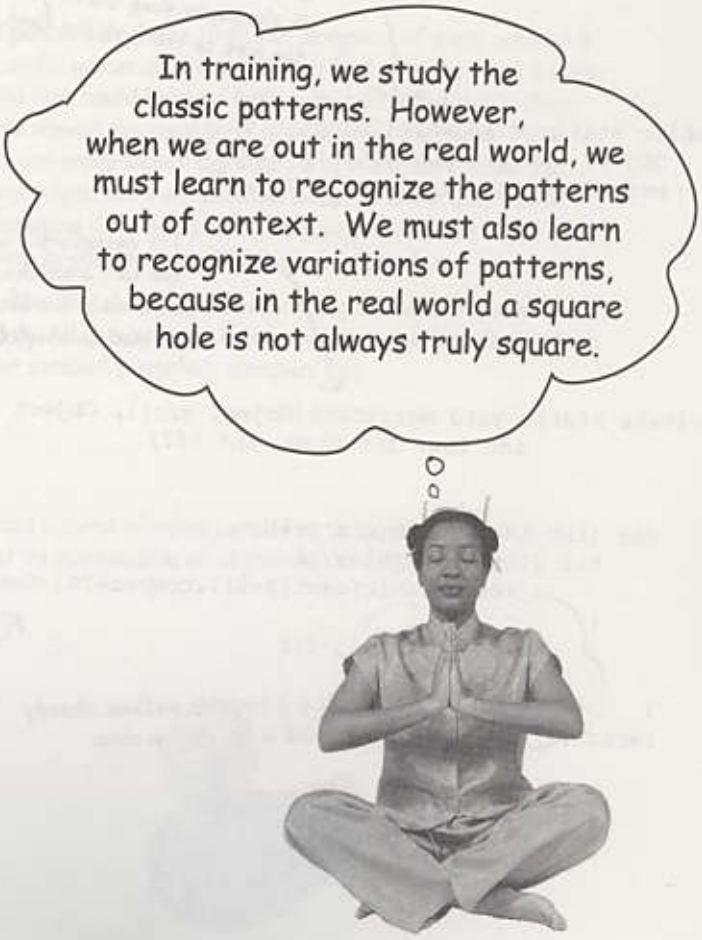
Pattern	Description
Template Method	Encapsulate interchangeable behaviors and use delegation to decide which behavior to use
Strategy	Subclasses decide how to implement steps in an algorithm
Factory Method	Subclasses decide which concrete classes to create

Template Methods in the Wild

The Template Method Pattern is a very common pattern and you're going to find lots of it in the wild. You've got to have a keen eye, though, because there are many implementations of the template methods that don't quite look like the textbook design of the pattern.

This pattern shows up so often because it's a great design tool for creating frameworks, where the framework controls how something gets done, but leaves you (the person using the framework) to specify your own details about what is actually happening at each step of the framework's algorithm.

Let's take a little safari through a few uses in the wild (well, okay, in the Java API)...



Sorting with Template Method

What's something we often need to do with arrays? Sort them!

Recognizing that, the designers of the Java Arrays class have provided us with a handy template method for sorting. Let's take a look at how this method operates:

We've paired down this code a little to make it easier to explain. If you'd like to see it all, grab the source from Sun and check it out...

We actually have two methods here and they act together to provide the sort functionality.

The first method, sort(), is just a helper method that creates a copy of the array and passes it along as the destination array to the mergeSort() method. It also passes along the length of the array and tells the sort to start at the first element.

```
public static void sort(Object[] a) {
    Object aux[] = (Object[])a.clone();
    mergeSort(aux, a, 0, a.length, 0);
}
```

The mergeSort() method contains the sort algorithm, and relies on an implementation of the compareTo() method to complete the algorithm.

Think of this as the **template method**.

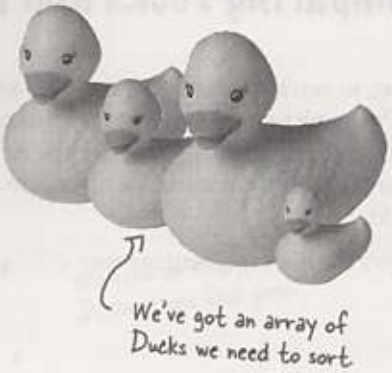
```
private static void mergeSort(Object src[], Object dest[],
                              int low, int high, int off)
{
    for (int i=low; i<high; i++){
        for (int j=i; j>low &&
             ((Comparable)dest[j-1]).compareTo((Comparable)dest[j])>0; j--)
        {
            swap(dest, j, j-1);
        }
    }
    return;
}
```

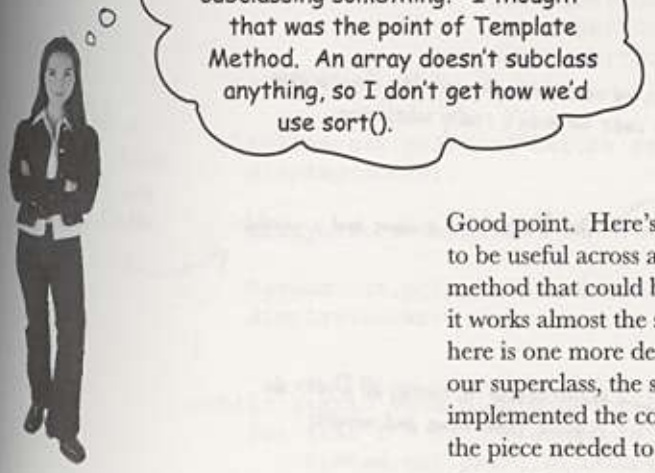
This is a concrete method, already defined in the Arrays class.

compareTo() is the method we need to implement to "fill out" the template method.

We've got some ducks to sort...

Let's say you have an array of ducks that you'd like to sort. How do you do it? Well, the sort template method in Arrays gives us the algorithm, but you need to tell it how to compare ducks, which you do by implementing the compareTo() method... Make sense?



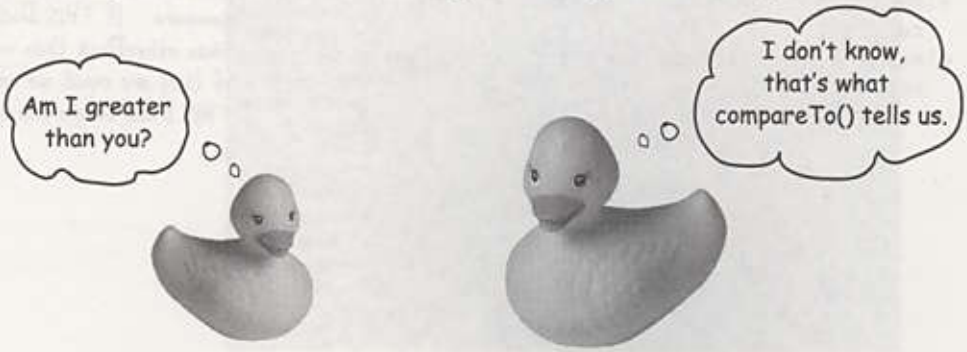


Good point. Here's the deal: the designers of sort() wanted it to be useful across all arrays, so they had to make sort() a static method that could be used from anywhere. But that's okay, it works almost the same as if it were in a superclass. Now, here is one more detail: because sort() really isn't defined in our superclass, the sort() method needs to know that you've implemented the compareTo() method, or else you don't have the piece needed to complete the sort algorithm.

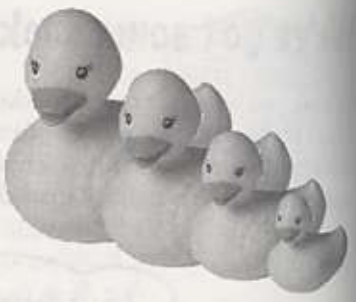
To handle this, the designers made use of the Comparable interface. All you have to do is implement this interface, which has one method (surprise): compareTo().

What is compareTo()?

The compareTo() method compares two objects and returns whether one is less than, greater than, or equal to the other. sort() uses this as the basis of its comparison of objects in the array.



Comparing Ducks and Ducks



Okay, so you know that if you want to sort Ducks, you're going to have to implement this compareTo() method; by doing that you'll give the Arrays class what it needs to complete the algorithm and sort your ducks.

Here's the duck implementation:

Remember, we need to implement the Comparable interface since we aren't really subclassing.

```
public class Duck implements Comparable {
    String name;
    int weight;

    public Duck(String name, int weight) {
        this.name = name;
        this.weight = weight;
    }

    public String toString() {
        return name + " weighs " + weight;
    }

    public int compareTo(Object object) {
        Duck otherDuck = (Duck)object;

        if (this.weight < otherDuck.weight) {
            return -1;
        } else if (this.weight == otherDuck.weight) {
            return 0;
        } else { // this.weight > otherDuck.weight
            return 1;
        }
    }
}
```

Our Ducks have a name and a weight

We're keepin' it simple; all Ducks do is print their name and weight!

Okay, here's what sort needs...

compareTo() takes another Duck to compare THIS Duck to.

Here's where we specify how Ducks compare. If THIS Duck weighs less than otherDuck then we return -1; if they are equal, we return 0; and if THIS Duck weighs more, we return 1.

Let's sort some Ducks

Here's the test drive for sorting Ducks...

```
public class DuckSortTestDrive {
    public static void main(String[] args) {
        Duck[] ducks = {
                        new Duck("Daffy", 8),
                        new Duck("Dewey", 2),
                        new Duck("Howard", 7),
                        new Duck("Louie", 2),
                        new Duck("Donald", 10),
                        new Duck("Huey", 2)
        };

        System.out.println("Before sorting:");
        display(ducks);

        Arrays.sort(ducks);

        System.out.println("\nAfter sorting:");
        display(ducks);
    }

    public static void display(Duck[] ducks) {
        for (int i = 0; i < ducks.length; i++) {
            System.out.println(ducks[i]);
        }
    }
}
```

We need an array of Ducks; these look good.

Let's print them to see their names and weights.

It's sort time!

Let's print them (again) to see their names and weights.

Notice that we call Arrays' static method sort, and pass it our Ducks.

Let the sorting commence!

```
File Edit Window Help DonaldNeedsToGoOnADiet
%java DuckSortTestDrive

Before sorting:
Daffy weighs 8
Dewey weighs 2
Howard weighs 7
Louie weighs 2
Donald weighs 10
Huey weighs 2

After sorting:
Dewey weighs 2
Louie weighs 2
Huey weighs 2
Howard weighs 7
Daffy weighs 8
Donald weighs 10
%
```

The unsorted Ducks

The sorted Ducks

The making of the sorting duck machine

Let's trace through how the Arrays sort() template method works. We'll check out how the template method controls the algorithm, and at certain points in the algorithm, how it asks our Ducks to supply the implementation of a step...

```
for (int i=low; i<high; i++){
        ... compareTo() ...
        ... swap() ...
}
```

The sort() method controls the algorithm, no class can change this. sort() counts on a Comparable class to provide the implementation of compareTo()

1. First, we need an array of Ducks:

```
Duck[] ducks = {new Duck("Daffy", 8), ... };
```

2. Then we call the sort() template method in the Array class and pass it our ducks:

```
Arrays.sort(ducks);
```

The sort() method (and its helper mergeSort()) control the sort procedure.

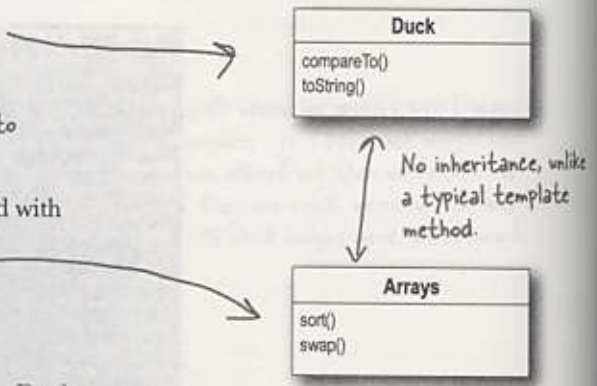
3. To sort an array, you need to compare two items one by one until the entire list is in sorted order.

When it comes to comparing two ducks, the sort method relies on the Duck's compareTo() method to know how to do this. The compareTo() method is called on the first duck and passed the duck to be compared to:

```
ducks[0].compareTo(ducks[1]);
```

First Duck

Duck to compare it to



4. If the Ducks are not in sorted order, they're swapped with the concrete swap() method in Arrays:

```
swap()
```

5. The sort method continues comparing and swapping Ducks until the array is in the correct order!

there are no Dumb Questions

Q: Is this really the Template Method Pattern, or are you trying too hard?

A: The pattern calls for implementing an algorithm and letting subclasses supply the implementation of the steps – and the Arrays sort is clearly not doing that! But, as we know, patterns in the wild aren't always just like the textbook patterns. They have to be modified to fit the context and implementation constraints.

The designers of the Arrays sort() method had a few constraints. In general, you can't subclass a Java array and they wanted the sort to be used on all arrays (and each array is a different class). So they defined a static method and deferred the comparison part of the algorithm to the items being sorted.

So, while it's not a textbook template method, this implementation is still in the spirit of the Template Method Pattern. Also, by eliminating the requirement that you have to subclass Arrays to use this algorithm, they've made sorting in some ways more flexible and useful.

Q: This implementation of sorting actually seems more like the Strategy Pattern than the Template Method Pattern. Why do we consider it Template Method?

A: You're probably thinking that because the Strategy Pattern uses object composition. You're right in a way – we're *using* the Arrays object to sort our array, so that's similar to Strategy. But remember, in Strategy, the class that you compose with implements the *entire* algorithm. The algorithm that Arrays implements for sort is incomplete; it needs a class to fill in the missing compareTo() method. So, in that way, it's more like Template Method.

Q: Are there other examples of template methods in the Java API?

A: Yes, you'll find them in a few places. For example, java.io has a read() method in InputStream that subclasses must implement and is used by the tempate method read(byte b[], int off, int len).

We know that we should favor composition over inheritance, right? Well, the implementers of the sort() template method decided not to use inheritance and instead to implement sort() as a static method that is composed with a Comparable at runtime. How is this better? How is it worse? How would you approach this problem? Do Java arrays make this particularly tricky?

Think of another pattern that is a specialization of the template method. In this specialization, primitive operations are used to create and return objects. What pattern is this?

Swingin' with Frames

Up next on our Template Method safari... keep your eye out for swinging JFrames!

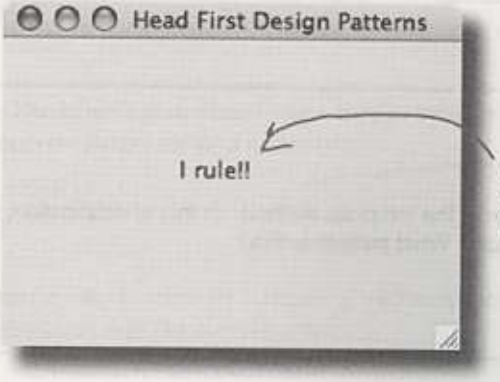
If you haven't encountered JFrame, it's the most basic Swing container and inherits a paint() method. By default, paint() does nothing because it's a *hook*! By overriding paint(), you can insert yourself into JFrame's algorithm for displaying its area of the screen and have your own graphic output incorporated into the JFrame. Here's an embarrassingly simple example of using a JFrame to override the paint() hook method:

We're extending JFrame, which contains a method update() that controls the algorithm for updating the screen. We can hook into that algorithm by overriding the paint() hook method.

Don't look behind the curtain! Just some initialization here...

JFrame's update algorithm calls paint(). By default, paint() does nothing... it's a hook. We're overriding paint(), and telling the JFrame to draw a message in the window.

```
public class MyFrame extends JFrame {

    public MyFrame(String title) {
        super(title);
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        this.setSize(300,300);
        this.setVisible(true);
    }

    public void paint(Graphics graphics) {
        super.paint(graphics);
        String msg = "I rule!!";
        graphics.drawString(msg, 100, 100);
    }

    public static void main(String[] args) {
        MyFrame myFrame = new MyFrame("Head First Design Patterns");
    }
}
```



Here's the message that gets painted in the frame because we've hooked into the paint() method.

Applets

Our final stop on the safari: the applet.

You probably know an applet is a small program that runs in a web page. Any applet must subclass Applet, and this class provides several hooks. Let's take a look at a few of them:

```
public class MyApplet extends Applet {
    String message;

    public void init() {
        message = "Hello World, I'm alive!";
        repaint();
    }

    public void start() {
        message = "Now I'm starting up...";
        repaint();
    }

    public void stop() {
        message = "Oh, now I'm being stopped...";
        repaint();
    }

    public void destroy() {
        // applet is going away...
    }

    public void paint(Graphics g) {
        g.drawString(message, 5, 15);
    }
}
```

The init hook allows the applet to do whatever it wants to initialize the applet the first time.

repaint() is a concrete method in the Applet class that lets upper-level components know the applet needs to be redrawn.

The start hook allows the applet to do something when the applet is just about to be displayed on the web page.

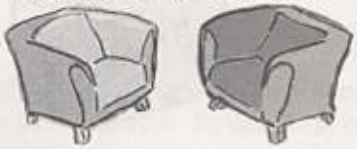
If the user goes to another page, the stop hook is used, and the applet can do whatever it needs to do to stop its actions.

And the destroy hook is used when the applet is going to be destroyed, say, when the browser pane is closed. We could try to display something here, but what would be the point?

Well looky here! Our old friend the paint() method! Applet also makes use of this method as a hook.

Concrete applets make extensive use of hooks to supply their own behaviors. Because these methods are implemented as hooks, the applet isn't required to implement them.

Fireside Chats



Tonight's talk: **Template Method and Strategy compare methods.**

Template Method

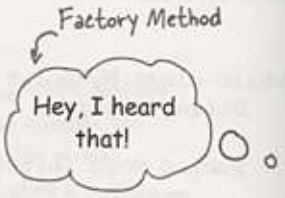
Hey Strategy, what are you doing in my chapter? I figured I'd get stuck with someone boring like Factory Method

I was just kidding! But seriously, what are you doing here? We haven't heard from you in eight chapters!

You might want to remind the reader what you're all about, since it's been so long.

Hey, that does sound a lot like what I do. But my intent's a little different from yours; my job is to define the outline of an algorithm, but let my subclasses do some of the work. That way, I can have different implementations of an algorithm's individual steps, but keep control over the algorithm's structure. Seems like you have to give up control of your algorithms.

Strategy



Nope, it's me, although be careful – you and Factory Method are related, aren't you?

I'd heard you were on the final draft of your chapter and I thought I'd swing by to see how it was going. We have a lot in common, so I thought I might be able to help...

I don't know, since Chapter 1, people have been stopping me in the street saying, "Aren't you that pattern..." So I think they know who I am. But for your sake: I define a family of algorithms and make them interchangeable. Since each algorithm is encapsulated, the client can use different algorithms easily.

I'm not sure I'd put it quite like *that*... and anyway, I'm not stuck using inheritance for algorithm implementations. I offer clients a choice of algorithm implementation through object composition.

Template Method

I remember that. But I have more control over my algorithm and I don't duplicate code. In fact, if every part of my algorithm is the same except for, say, one line, then my classes are much more efficient than yours. All my duplicated code gets put into the superclass, so all the subclasses can share it.

Yeah, well, I'm *real* happy for ya, but don't forget I'm the most used pattern around. Why? Because I provide a fundamental method for code reuse that allows subclasses to specify behavior. I'm sure you can see that this is perfect for creating frameworks.

How's that? My superclass is abstract.

Like I said Strategy, I'm *real* happy for you. Thanks for stopping by, but I've got to get the rest of this chapter done.

Got it. Don't call us, we'll call you...

Strategy

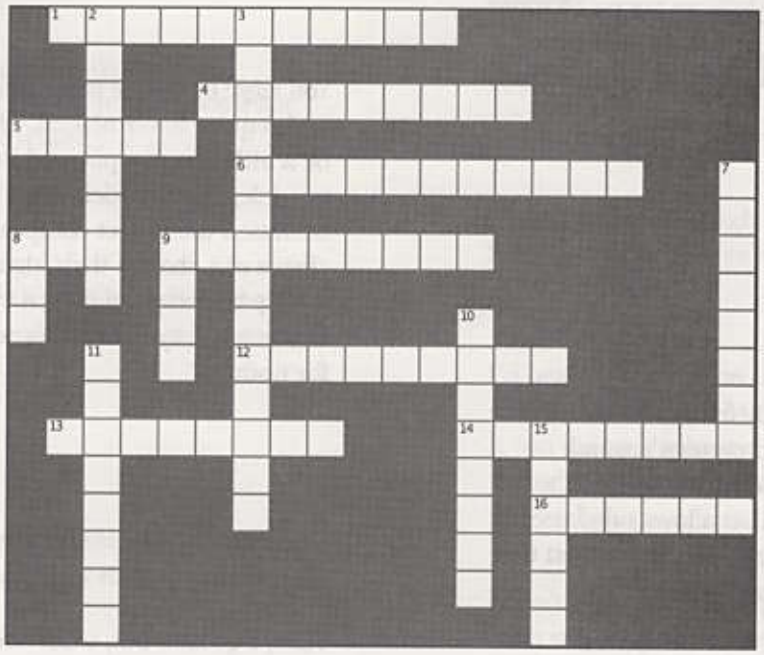
You *might* be a little more efficient (just a little) and require fewer objects. *And* you might also be a little less complicated in comparison to my delegation model, but I'm more flexible because I use object composition. With me, clients can change their algorithms at runtime simply by using a different strategy object. Come on, they didn't choose *me* for Chapter 1 for nothing!

Yeah, I guess... but, what about dependency? You're way more dependent than me.

But you have to depend on methods implemented in your superclass, which are part of your algorithm. I don't depend on anyone; I can do the entire algorithm myself!

Okay, okay, don't get touchy. I'll let you work, but let me know if you need my special techniques anyway, I'm always glad to help.

It's that time again....



Across

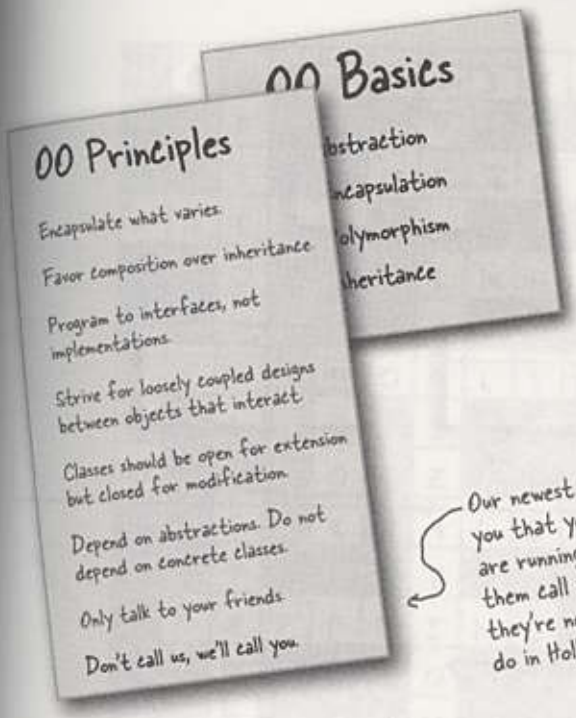
1. Strategy uses _____ rather than inheritance
4. Type of sort used in Arrays
5. The JFrame hook method that we overrode to print "I Rule"
6. The Template Method Pattern uses _____ to defer implementation to other classes
8. Coffee and _____
9. Don't call us, we'll call you is known as the _____ Principle
12. A template method defines the steps of an _____
13. In this chapter we gave you more _____
14. The template method is usually defined in an _____ class
16. Class that likes web pages

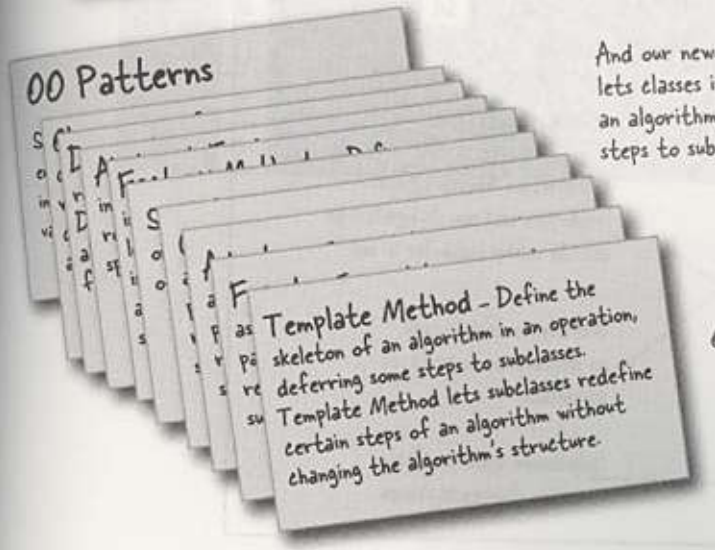
Down

2. _____ algorithm steps are implemented by hook methods
3. Factory Method is a _____ of Template Method
7. The steps in the algorithm that must be supplied by the subclasses are usually declared _____
8. Huey, Louie and Dewey all weigh _____ pounds
9. A method in the abstract superclass that does nothing or provides default behavior is called a _____ method
10. Big headed pattern
11. Our favorite coffee shop in Objectville
15. The Arrays class implements its template method as a _____ method

Tools for your Design Toolbox

We've added Template Method to your toolbox. With Template Method you can reuse code like a pro while keeping control of your algorithms.





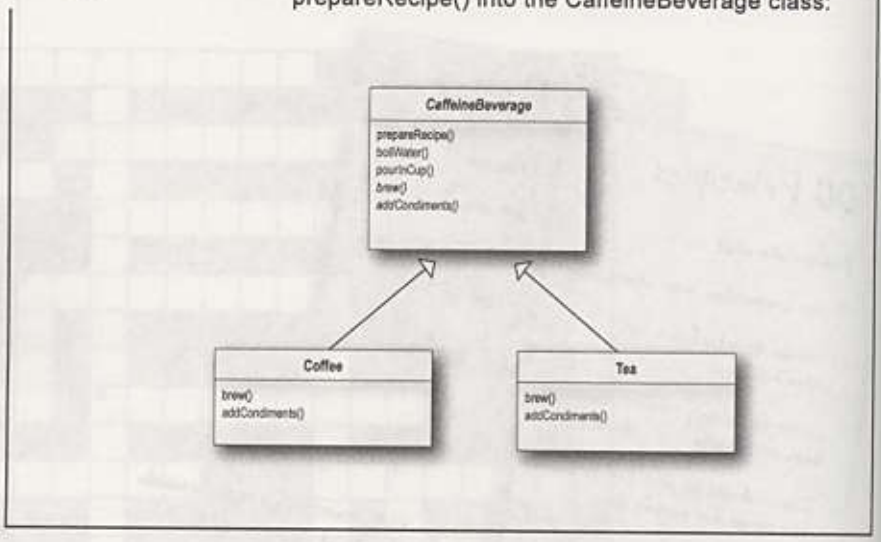
BULLET POINTS

- A "template method" defines the steps of an algorithm, deferring to subclasses for the implementation of those steps.
- The Template Method Pattern gives us an important technique for code reuse.
- The template method's abstract class may define concrete methods, abstract methods and hooks.
- Abstract methods are implemented by subclasses.
- Hooks are methods that do nothing or default behavior in the abstract class, but may be overridden in the subclass.
- To prevent subclasses from changing the algorithm in the template method, declare the template method as final.
- The Hollywood Principle guides us to put decision-making in high-level modules that can decide how and when to call low level modules.
- You'll see lots of uses of the Template Method Pattern in real world code, but don't expect it all (like any pattern) to be designed "by the book."
- The Strategy and Template Method Patterns both encapsulate algorithms, one by inheritance and one by composition.
- The Factory Method is a specialization of Template Method.

Exercise solutions

Sharpen your pencil

Draw the new class diagram now that we've moved prepareRecipe() into the CaffeineBeverage class:

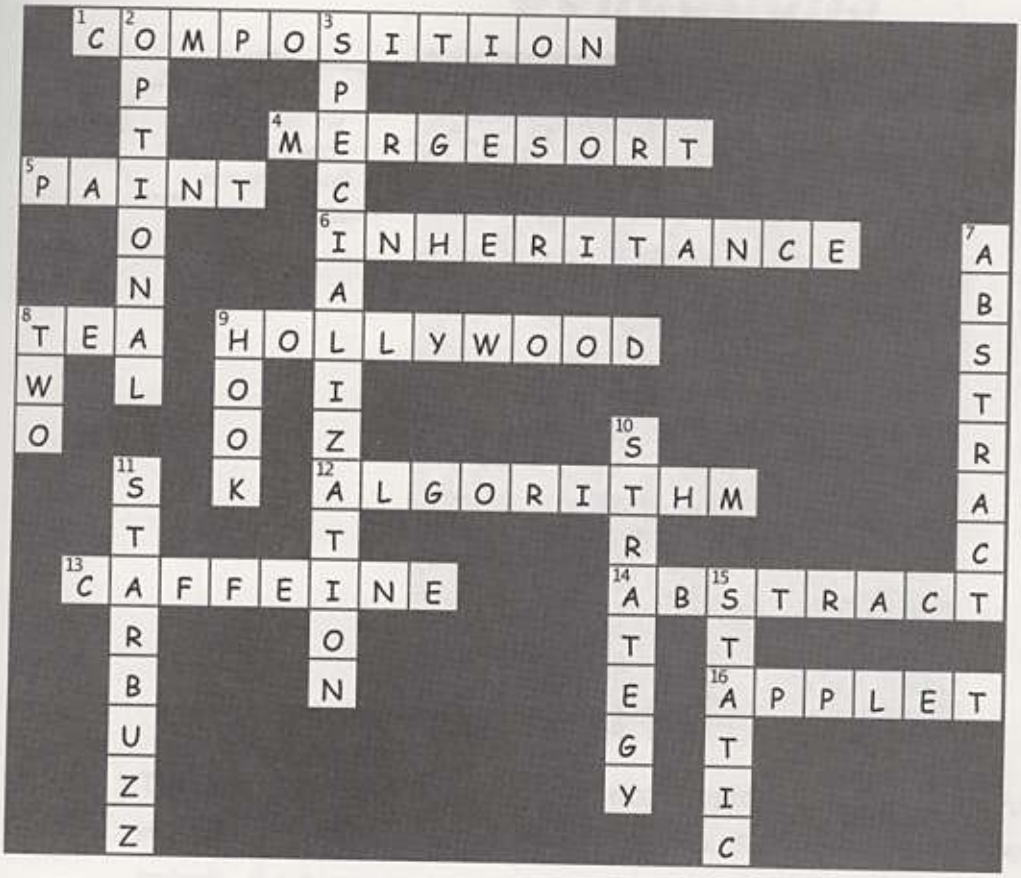


WHO DOES WHAT?

Match each pattern with its description:

Pattern	Description
Template Method	Encapsulate interchangable behaviors and use delegation to decide which behavior to use
Strategy	Subclasses decide how to implement steps in an algorithm
Factory Method	Subclasses decide which concrete classes to create

Exercise solutions



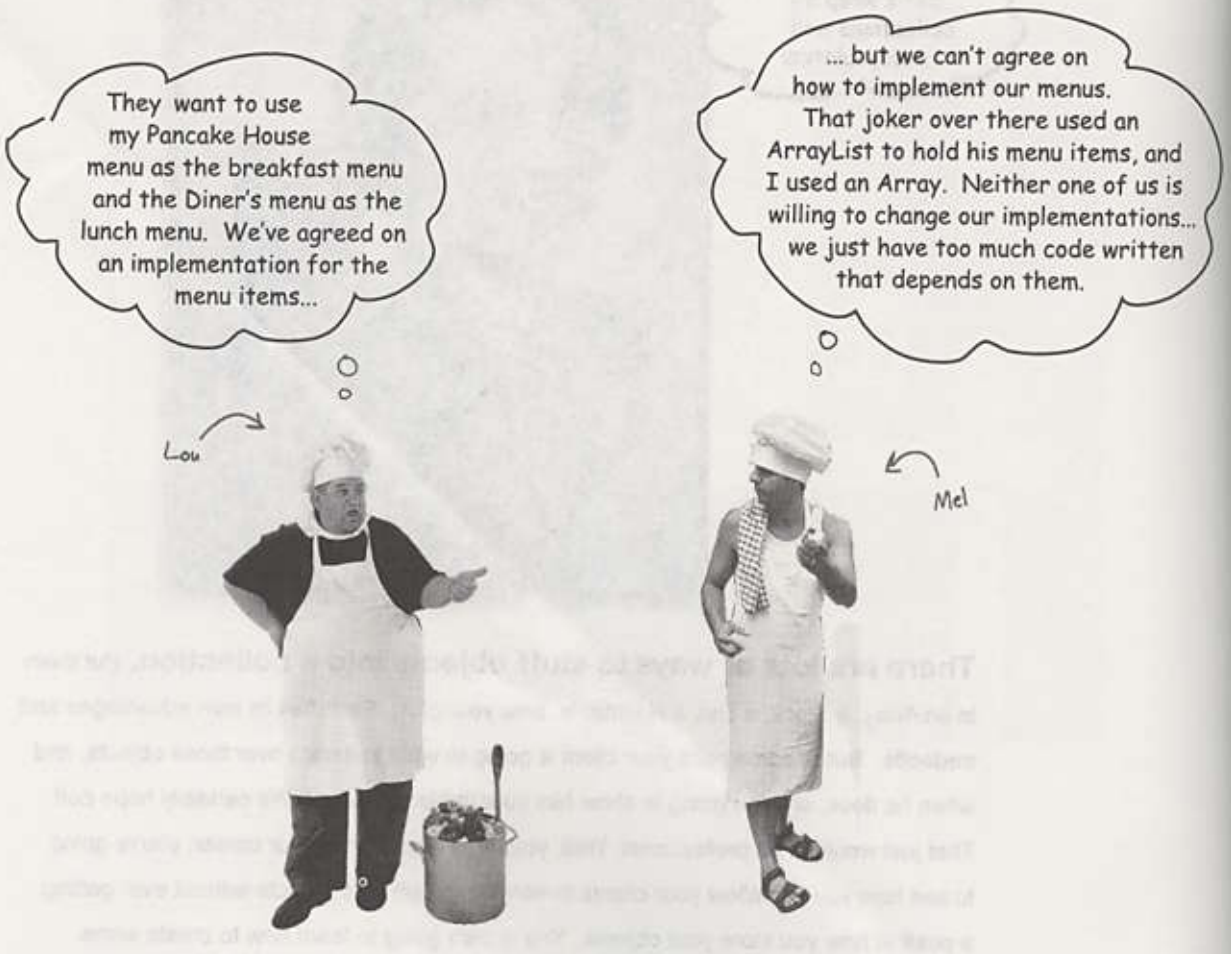
9 the Iterator and Composite Patterns

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.

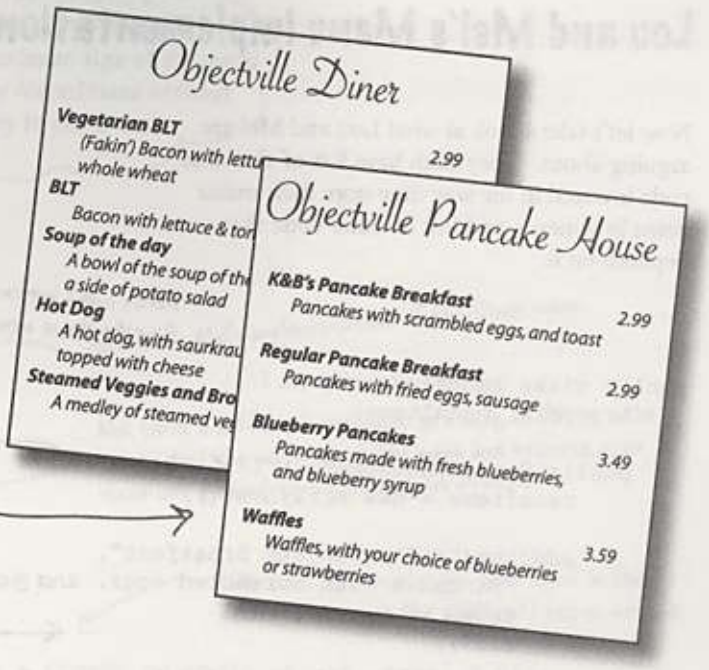
Breaking News: Objectville Diner and Objectville Pancake House Merge

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.



The Diner menu has lots of lunch items, while the Pancake House consists of breakfast items. Every menu item has a name, a description, and a price

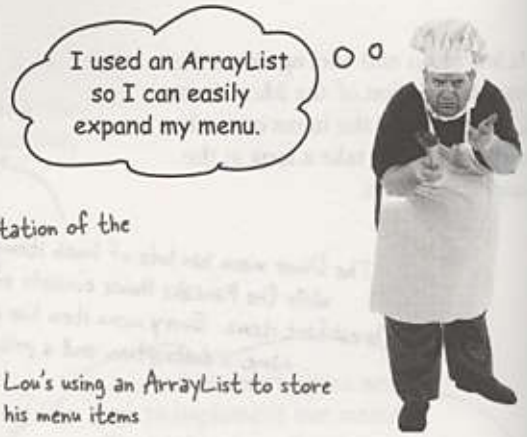
```
public class MenuItem {
    String name;
    String description;
    boolean vegetarian;
    double price;

    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!

Haah! An Arraylist... I used a REAL Array so I can control the maximum size of my menu and get my MenuItems without having to use a cast.

And here's Mel's implementation of the Diner menu.

```
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.

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

```
Java-Enabled Waitress: code-name "Alice"

printMenu()
   - prints every item on the menu

printBreakfastMenu()
   - prints just breakfast items

printLunchMenu()
   - prints just lunch items

printVegetarianMenu()
   - prints all vegetarian menu items

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

The spec for the Waitress

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 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.

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.
- ❑ C. The Waitress needs to know how each menu represents its internal collection of menu items; this violates encapsulation.
- ❑ D. 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.
- ❑ E. 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.

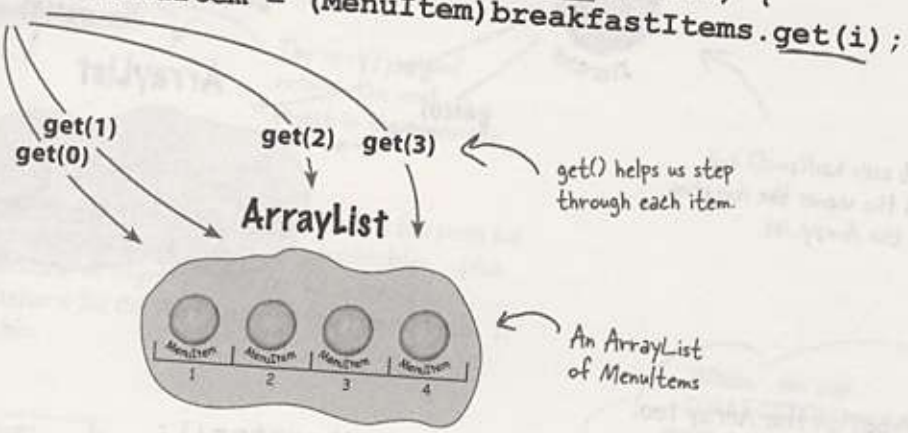
Sound good? Well, how are we going to do that?

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...

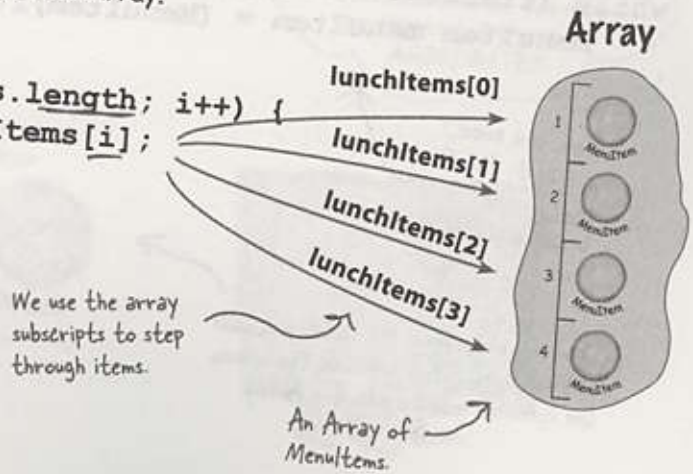
1. To iterate through the breakfast items we use the size() and get() methods on the ArrayList:

```
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.

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



❸ 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

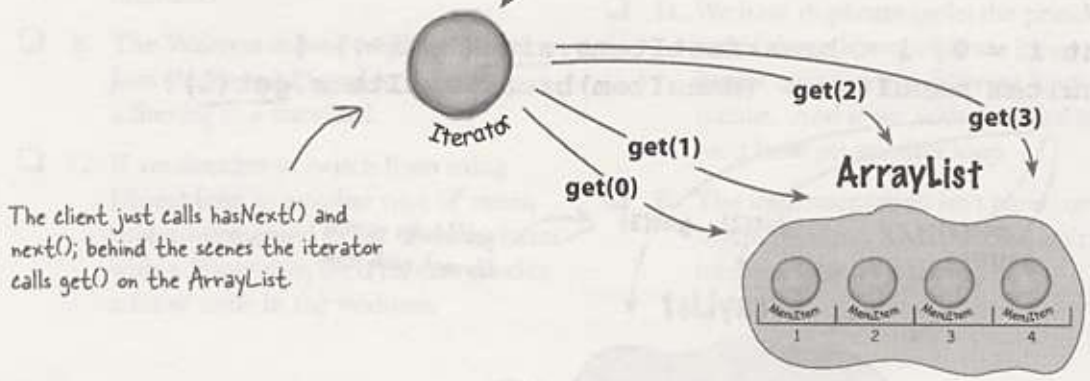
We ask the breakfastMenu for an iterator of its MenuItems.

```
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.



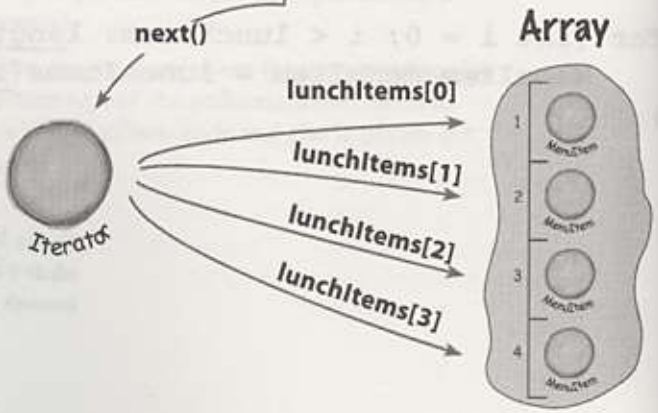
The client just calls hasNext() and next(); behind the scenes the iterator calls get() on the ArrayList.

❹ 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.

next()

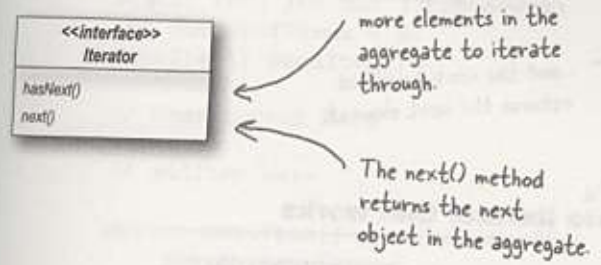


Same situation here: the client just calls hasNext() and next(); behind the scenes, the iterator indexes into the Array.

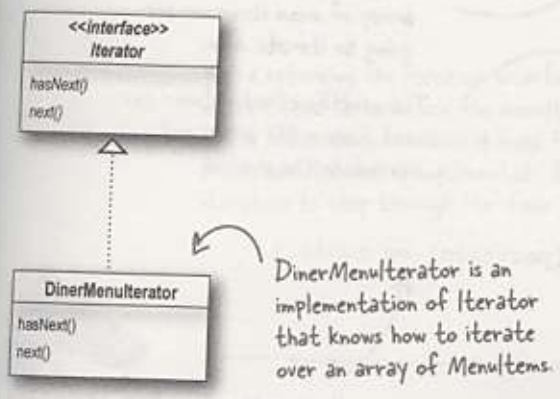
Meet the Iterator Pattern

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.

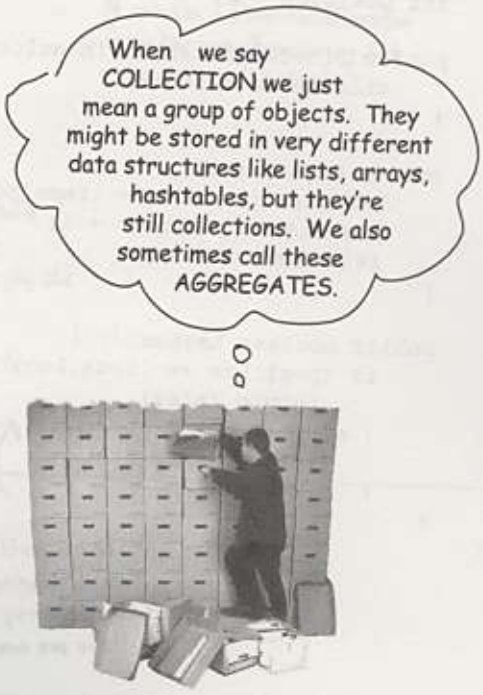
The first thing you need to know about the Iterator Pattern is that it relies on an interface called Iterator. Here's one possible Iterator interface:



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:

```
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.

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.

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:

```
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.

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

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

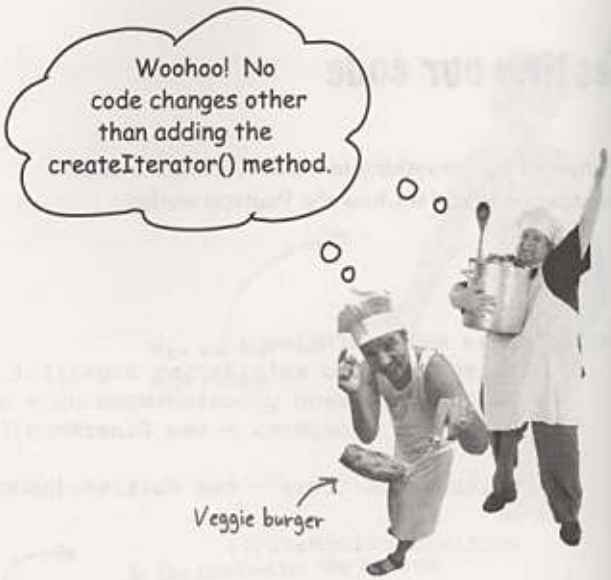
First we iterate through the pancake menu.

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 getIterator() 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.

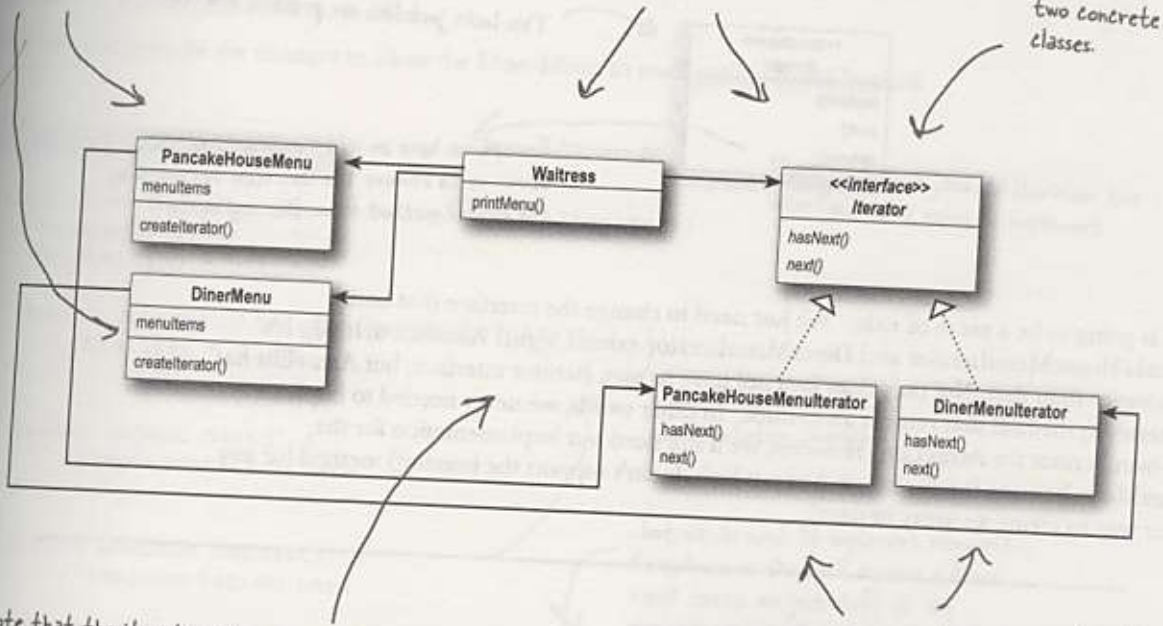
What we have so far...

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

These two menus implement the same exact set of methods, but they aren't implementing the same interface. We're going to fix this and free the Waitress from any dependencies on concrete Menus.

The Iterator allows the Waitress to be decoupled from the actual implementation of the concrete classes. She doesn't need to know if a Menu is implemented with an Array, an ArrayList, or with PostIt™ notes. All she cares is that she can get an Iterator to do her iterating.

We're now using a common Iterator interface and we've implemented two concrete classes.



Note that the iterator give us a way to step through the elements of an aggregate without forcing the aggregate to clutter its own interface with a bunch of methods to support traversal of its elements. It also allows the implementation of the iterator to live outside of the aggregate; in other words, we've encapsulated the interation.

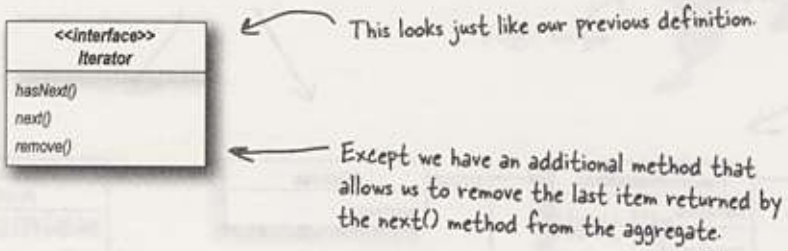
PancakeHouseMenu and DinerMenu implement the new createIterator() method; they are responsible for creating the iterator for their respective menu items implementations.

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

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

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.

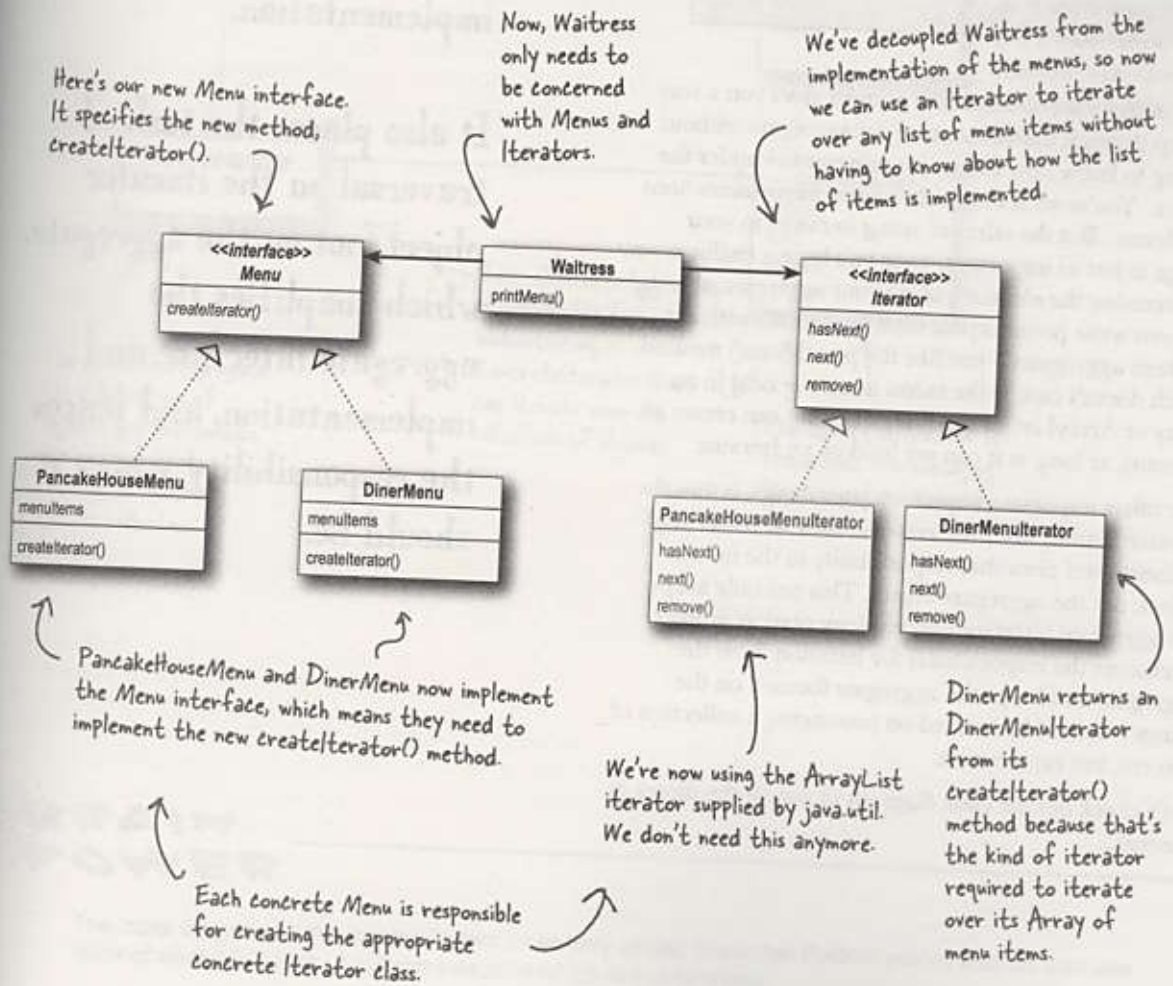
What does this get us?

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."

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.

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:

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

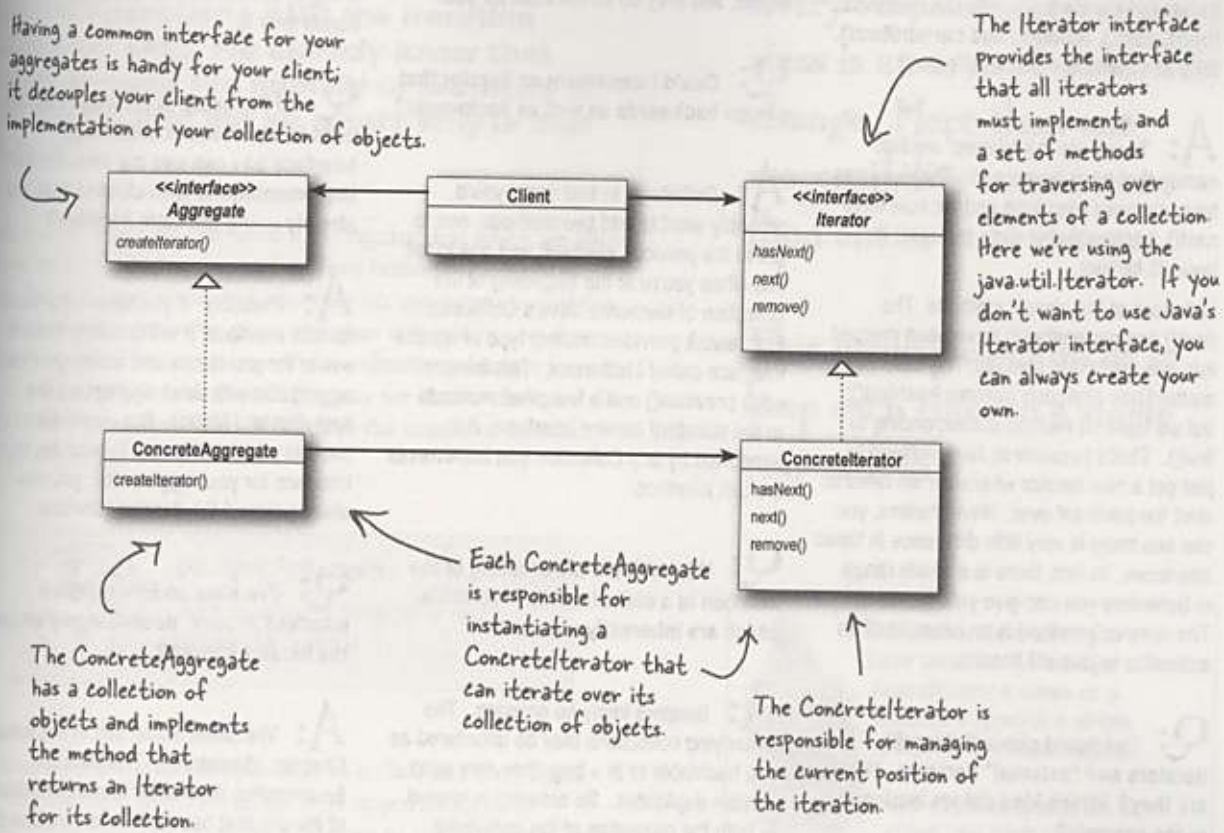
This makes a lot of sense: the pattern gives you a way to step through the elements of an aggregate without having to know how things are represented under the covers. You've seen that with the two implementations of Menus. But the effect of using iterators in your design is just as important: once you 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 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...

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

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

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

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

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

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

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?

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:

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.

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.

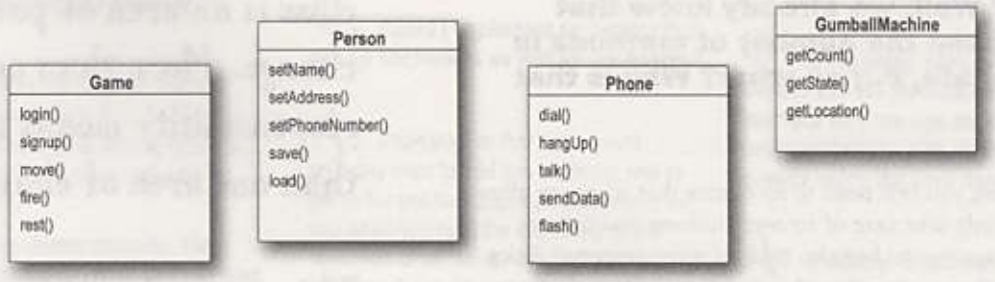
Cohesion is a term you'll hear used as a measure of how closely a class or a module supports a single purpose or responsibility.

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

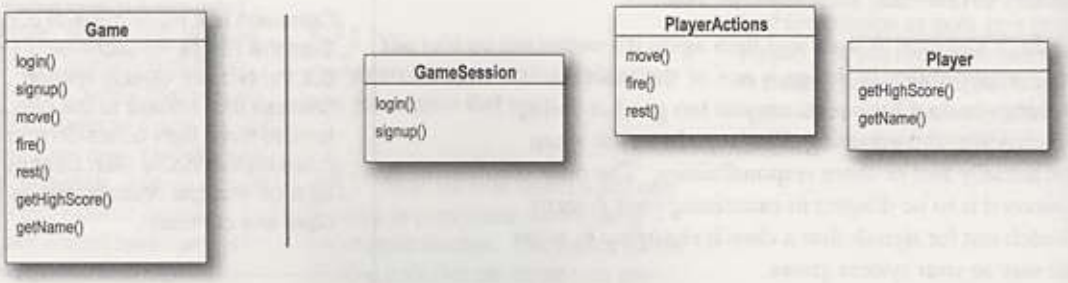
Examine these classes and determine which ones have multiple responsibilities.

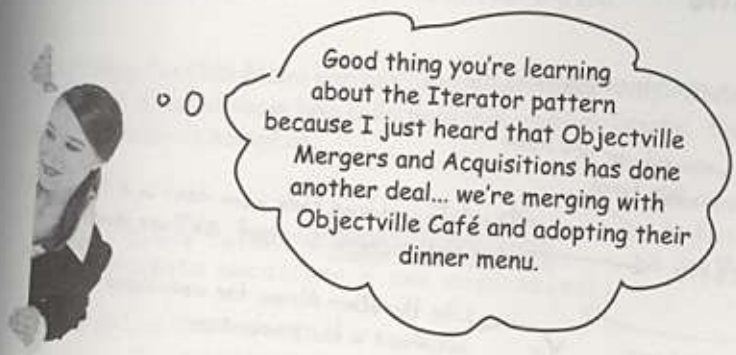


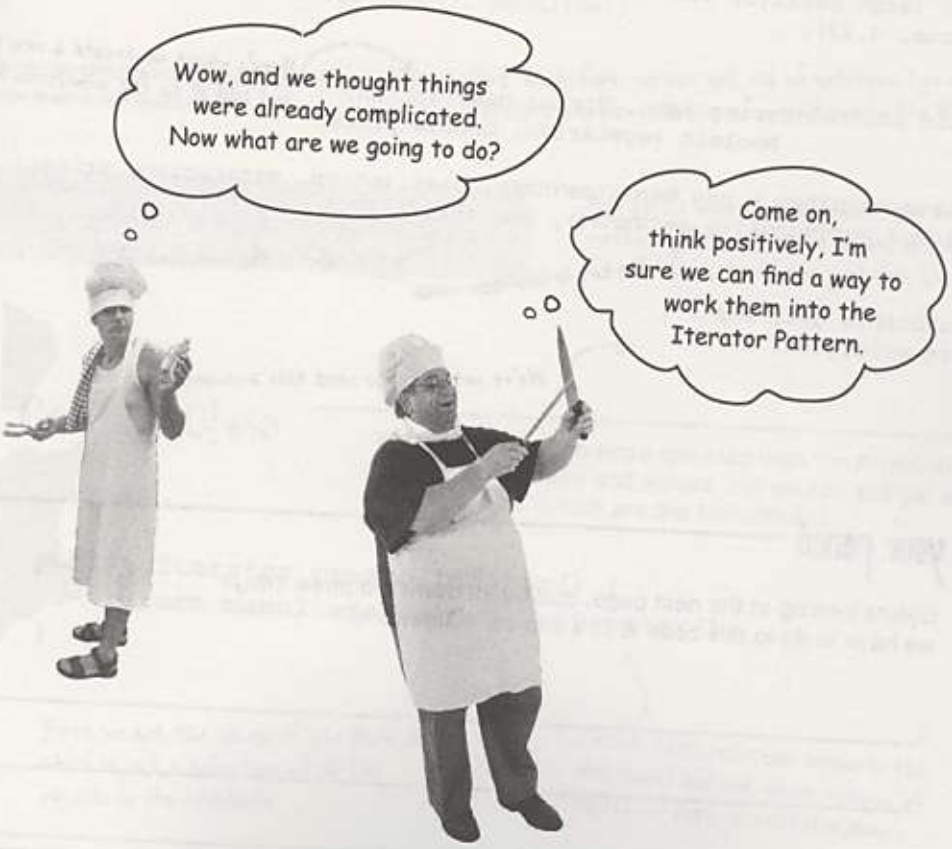
HARD HAT AREA, WATCH OUT FOR FALLING ASSUMPTIONS

BRAIN² POWER

Determine if these classes have low or high cohesion.




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?
Come on, think positively, I'm sure we can find a way to work them into the Iterator Pattern.

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...

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.

```
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();
    }
}
```

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!

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

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

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

First we iterate through the pancake menu.

And then the diner menu.

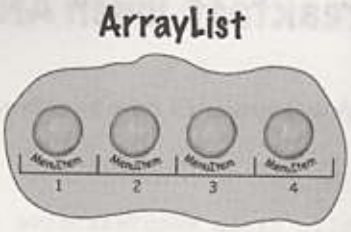
And finally the new café menu, all with the same iteration code.

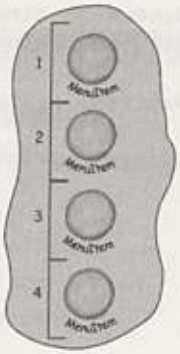
What did we do?

We wanted to give the Waitress an easy way to iterate over menu items...

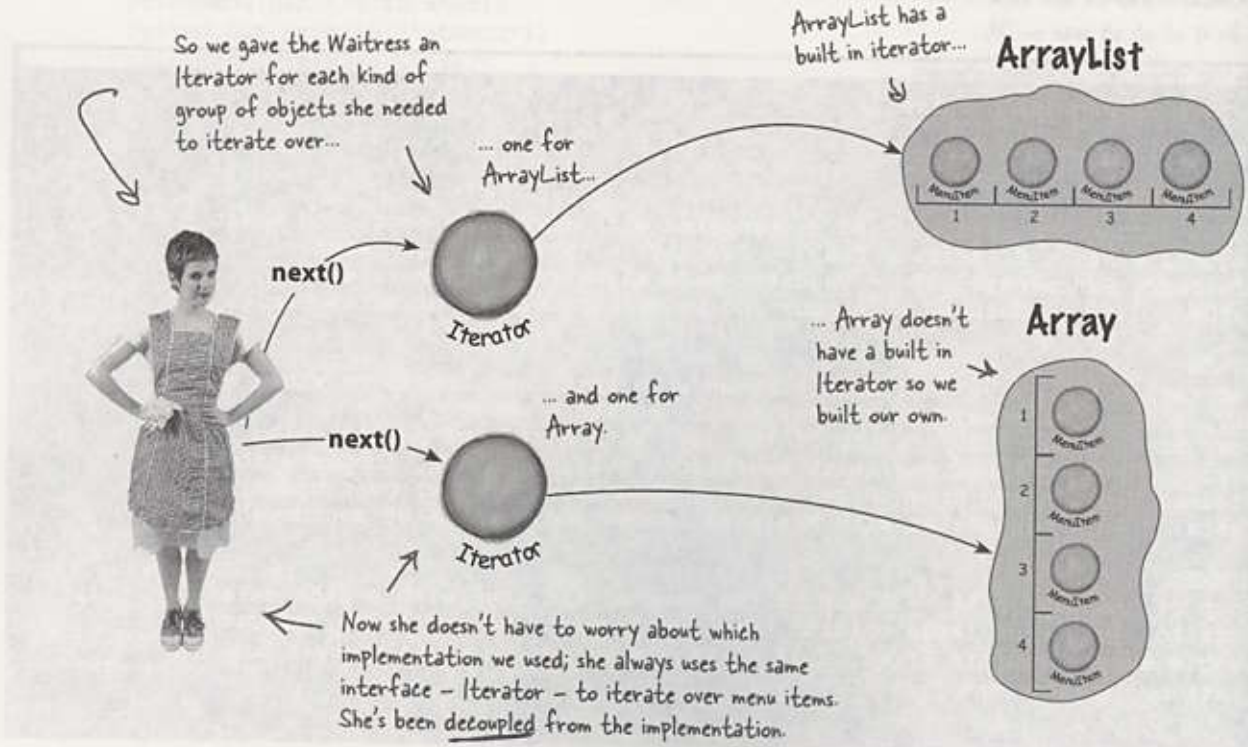
... and we didn't want her to know about how the menu items are implemented.

Our menu items had two different implementations and two different interfaces for iterating.





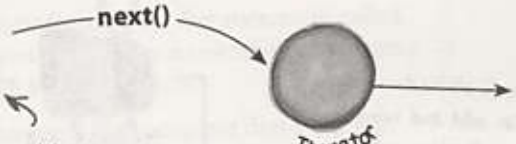
We decoupled the Waitress....



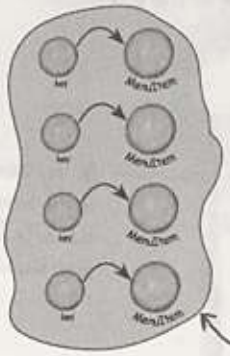
...and we made the Waitress more extensible

By giving her an Iterator we have decoupled her from the implementation of the menu items, so we can easily add new Menus if we want.

We easily added another implementation of menu items, and since we provided an Iterator, the Waitress knew what to do.



Which is better for her, because now she can use the same code to iterate over any group of objects. And it's better for us because the implementation details aren't exposed.



Making an Iterator for the Hashtable values was easy; when you call values.iterator() you get an Iterator.

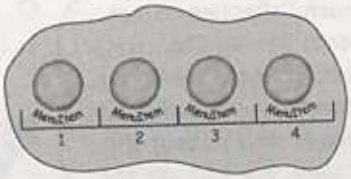
But there's more!

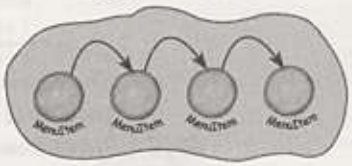
Java gives you a lot of "collection" classes that allow you to store and retrieve groups of objects. For example, Vector and LinkedList

Most have different interfaces.

But almost all of them support a way to obtain an Iterator.

And if they don't support Iterator, that's ok, because now you know how to build your own.



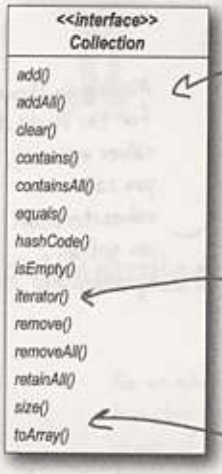


...and more!

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:



As you can see, there's all kinds of good stuff here. You can add and remove elements from your collection without even knowing how it's implemented.

Here's our old friend, the iterator() method. With this method, you can get an Iterator for any class that implements the Collection interface.

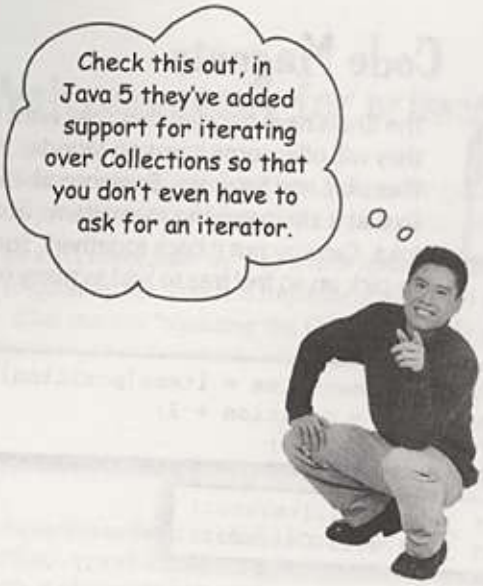
Other handy methods include size(), to get the number of elements, and toArray() to turn your collection into an array.

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.

The nice thing about Collections and Iterator is that each Collection object knows how to create its own Iterator. Calling iterator() on an ArrayList returns a concrete Iterator made for ArrayLists, but you never need to see or worry about the concrete class it uses; you just use the Iterator interface.

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.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?"*

Three createIterator() calls.

```
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);
}
```

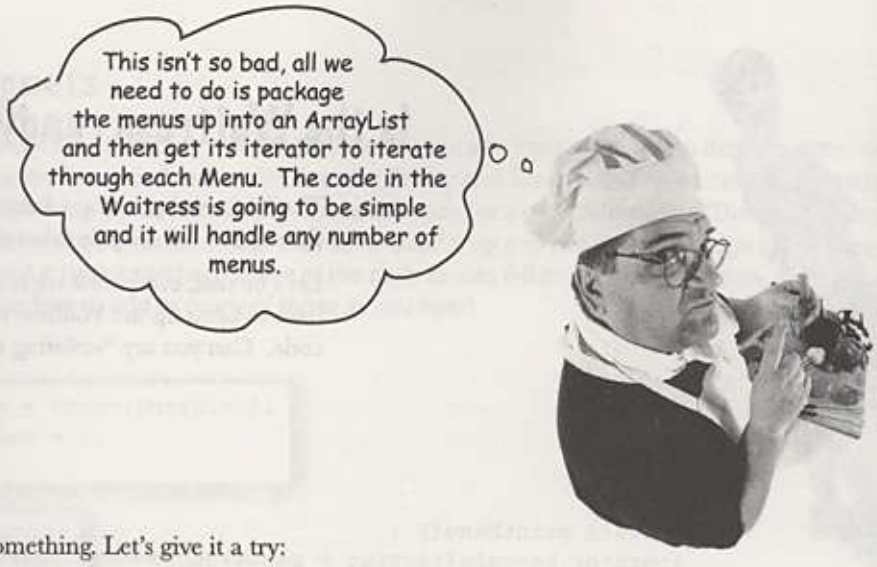
Three calls to printMenu.

Everytime we add or remove a menu we're going to have to open this code up for changes.

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.

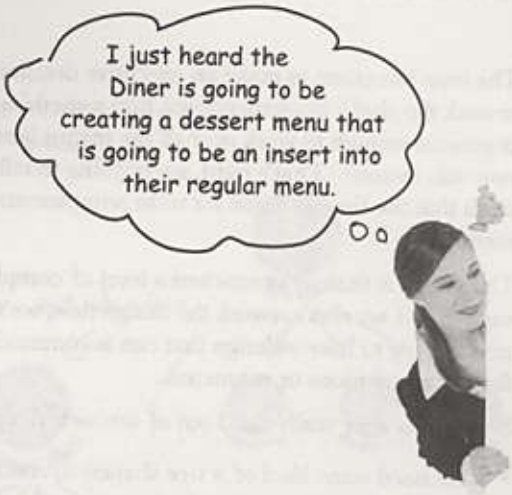
This looks pretty good, although we've lost the names of the menus, but we could add the names to each menu.

Just when we thought it was safe...

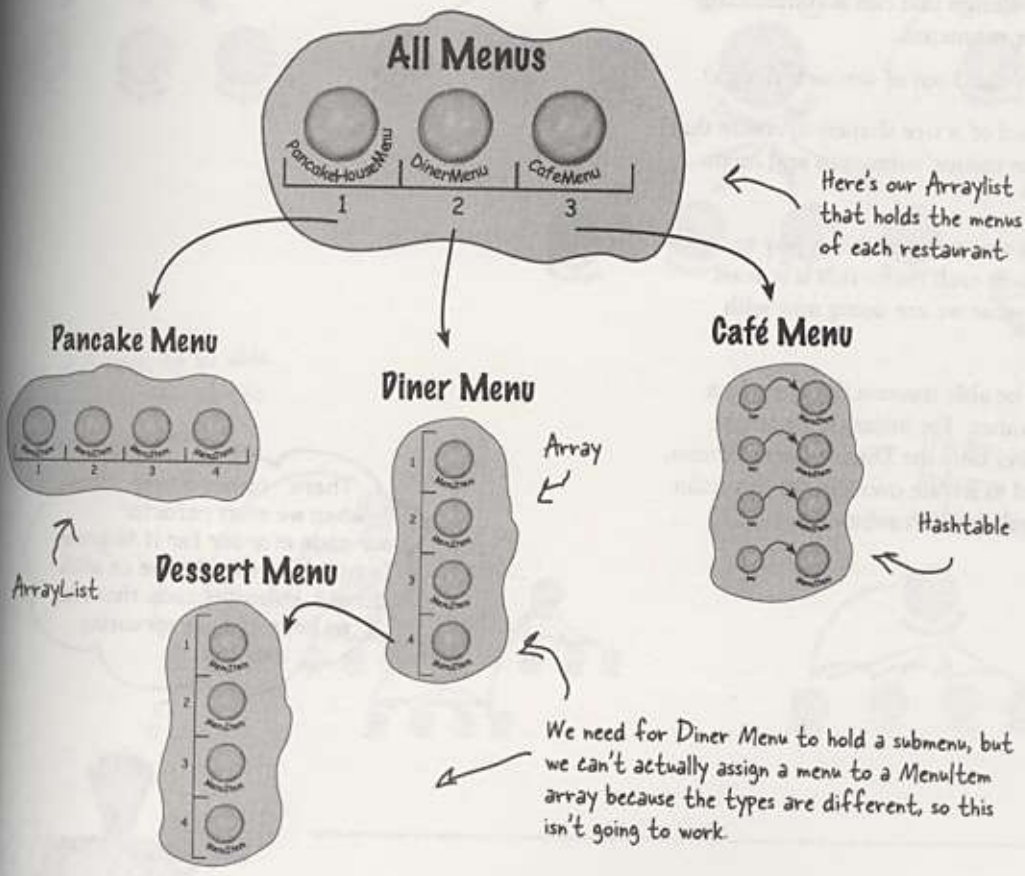
Now they want to add a dessert submenu.

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!

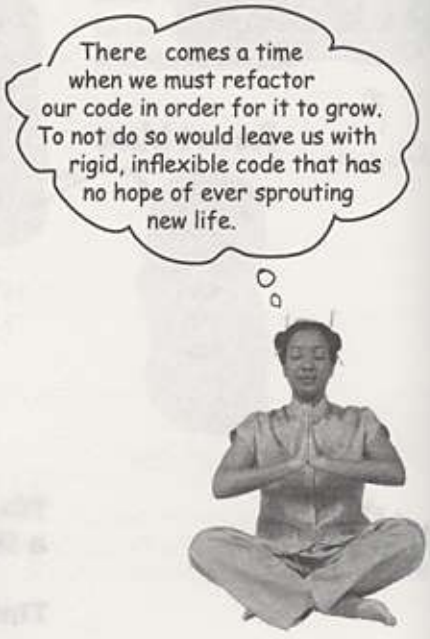
What do we need?

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 as come for us to reimplement their menus.

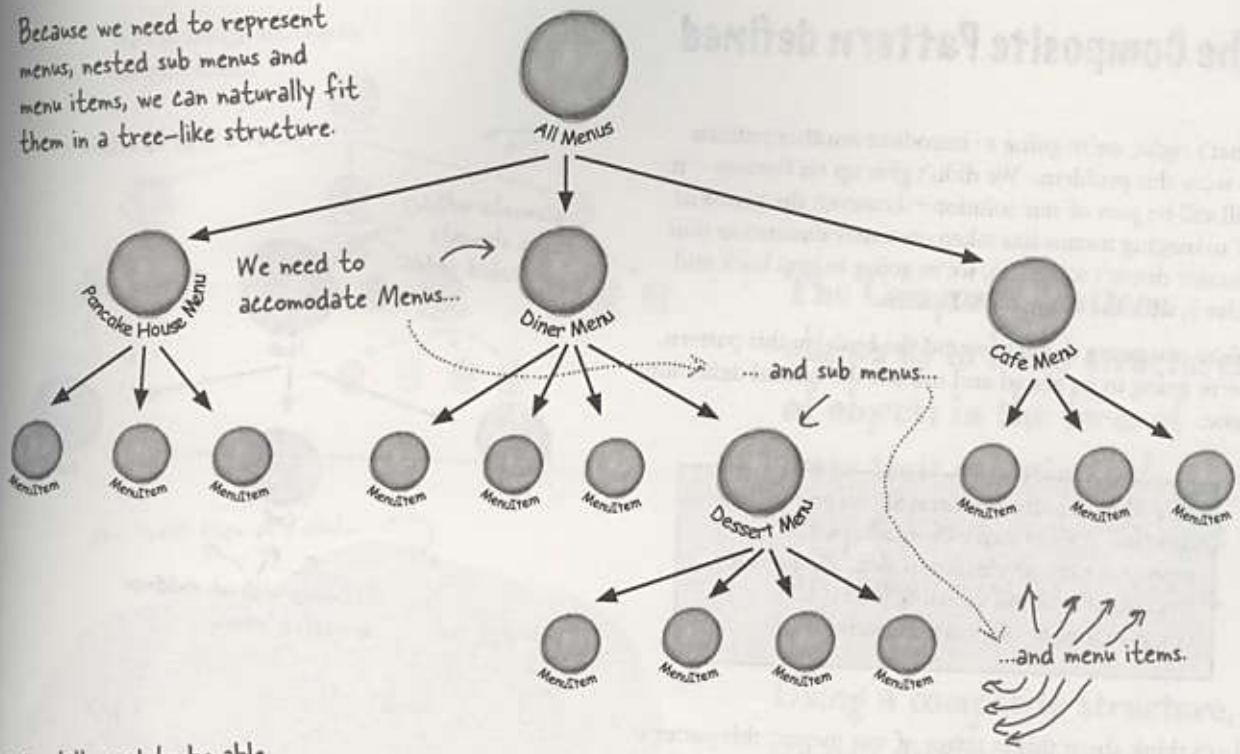
The reality is that we've reached a level of complexity such that if we don't rework the design now, we're never going to have a design that can accommodate further acquisitions or submenus.

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

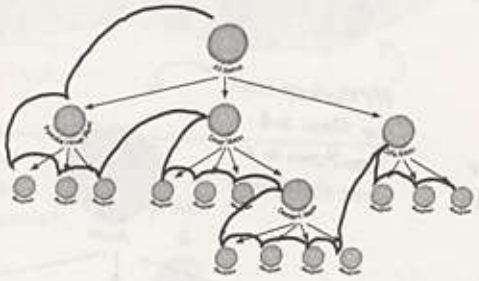
- We need some kind of a tree shaped structure that will accommodate menus, submenus and menu items.
- We need to make sure we maintain a way to traverse the items in each menu that is at least as convenient as what we are doing now with iterators.
- We may need to be able traverse the items in a more 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.



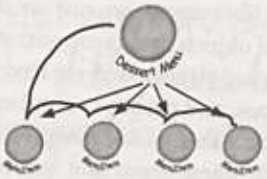
Because we need to represent menus, nested sub menus and menu items, we can naturally fit them in a tree-like structure.



We still need to be able to traverse the all the items in the tree.



We also need to be able to traverse more flexibly, for instance over one menu.



BRAIN POWER

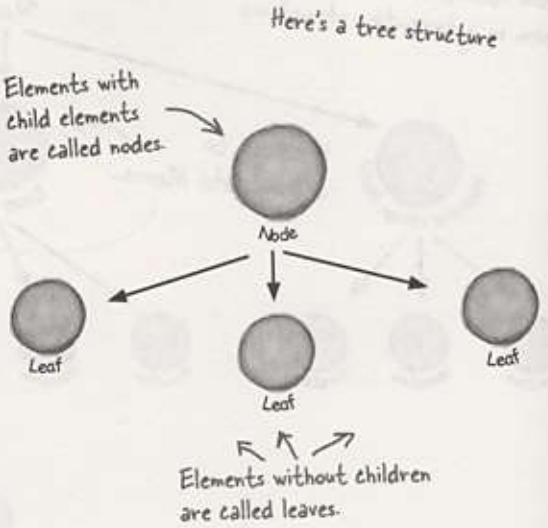
How would you handle this new wrinkle to our design requirements? Think about it before turning the page.

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.

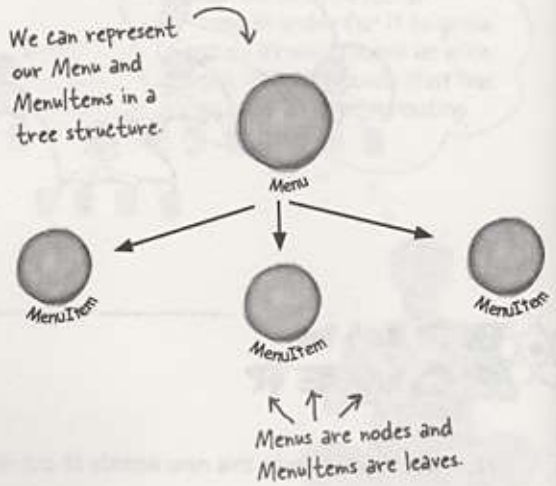
We're not going to beat around the bush on this pattern, we're going to go ahead and roll out the official definition now:

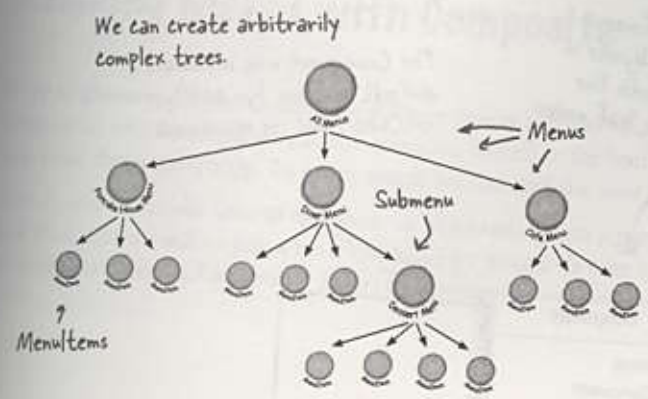
> **The Composite Pattern** allows you to compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

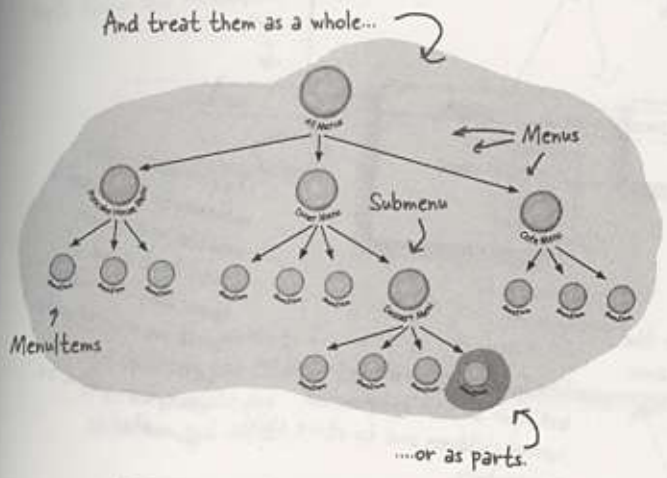


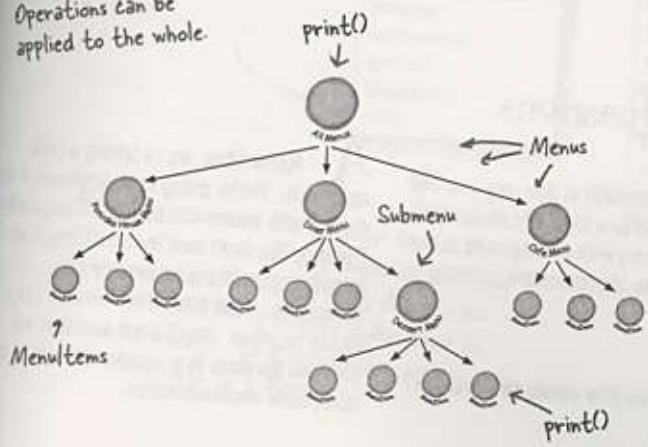
Let's think about this in terms of our menus: this pattern gives us a way to create a tree structure that can handle a nested group of menus *and* menu items in the same structure. By putting menus and items in the same structure we create a part-whole hierarchy; that is, a tree of objects that is made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





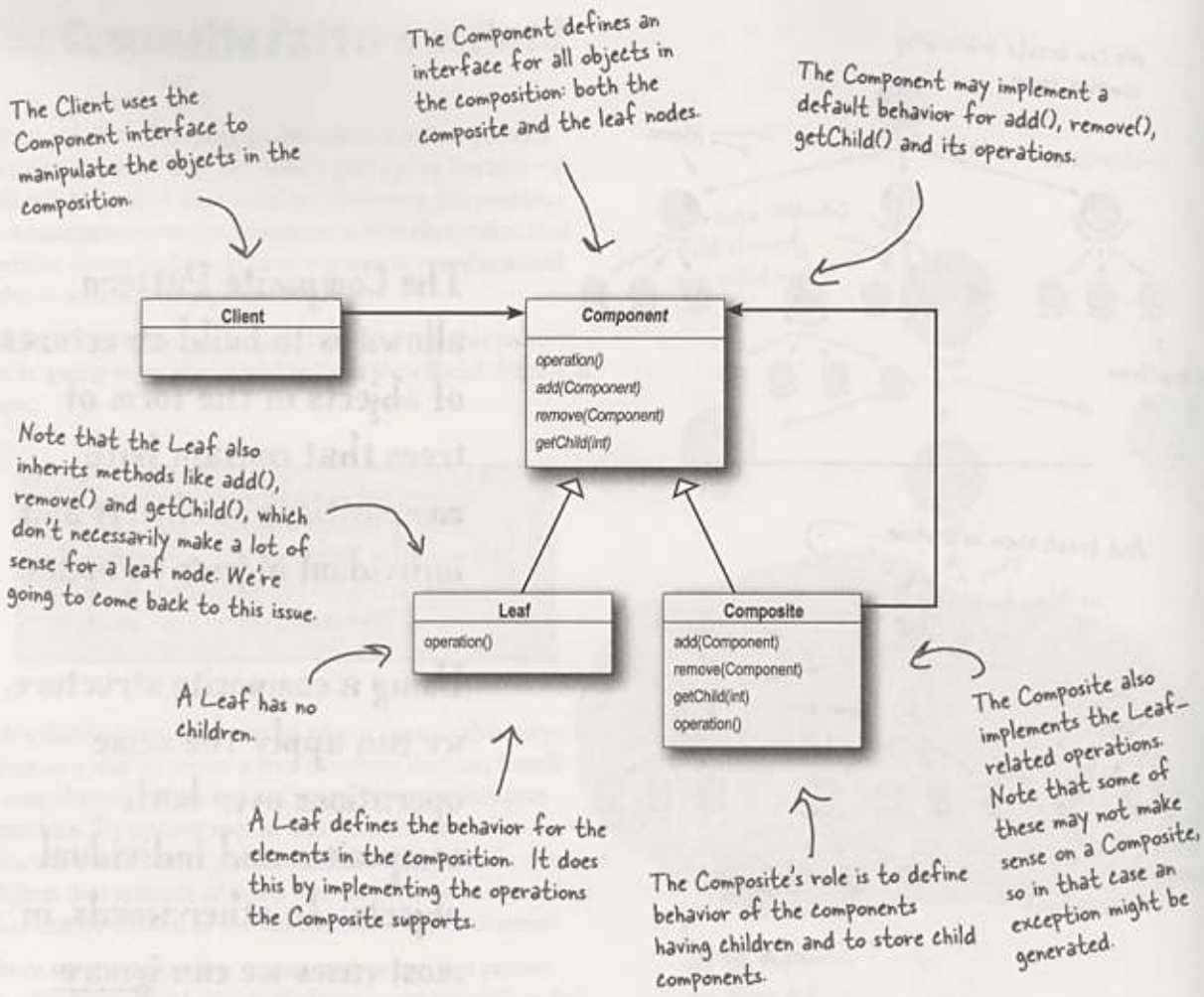




The Composite Pattern allows us to build structures of objects in the form of trees that contain both compositions of objects and individual objects as nodes.

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.

composite pattern class diagram



there are no Dumb Questions

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

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.

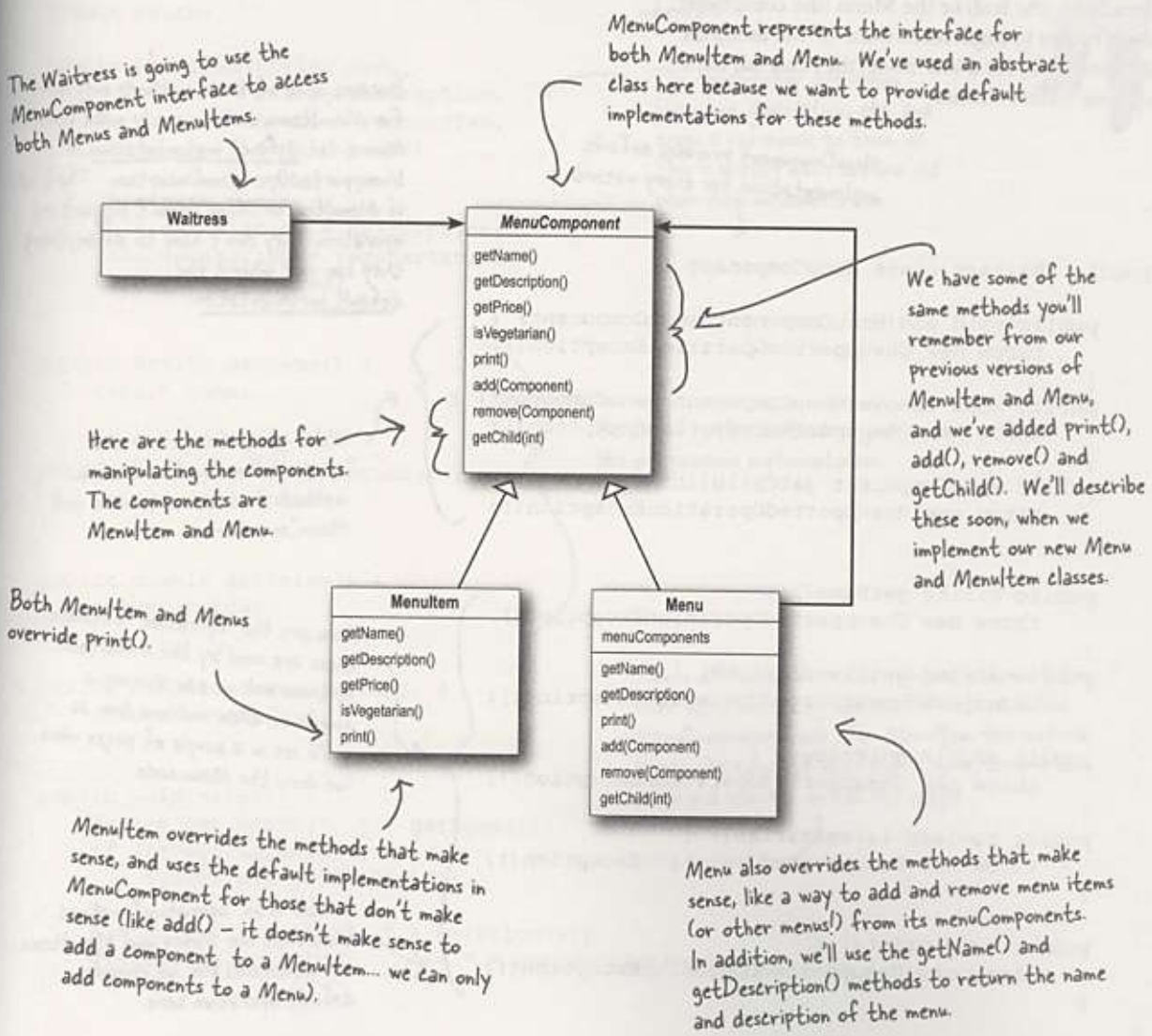
Q: How does this relate to iterators?

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:

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.

MenuComponent provides default implementations for every method.

```
public abstract class MenuComponent {

    public void add(MenuComponent menuComponent) {
        throw new UnsupportedOperationException();
    }
    public void remove(MenuComponent menuComponent) {
        throw new UnsupportedOperationException();
    }
    public MenuComponent getChild(int i) {
        throw new UnsupportedOperationException();
    }

    public String getName() {
        throw new UnsupportedOperationException();
    }
    public String getDescription() {
        throw new UnsupportedOperationException();
    }
    public double getPrice() {
        throw new UnsupportedOperationException();
    }
    public boolean isVegetarian() {
        throw new UnsupportedOperationException();
    }

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

Because some of these methods only make sense for MenuItems, and some only make sense for Menus, the default implementation is UnsupportedOperationException. That way, if MenuItem or Menu doesn't support an operation, they don't have to do anything, they can just inherit the default implementation.

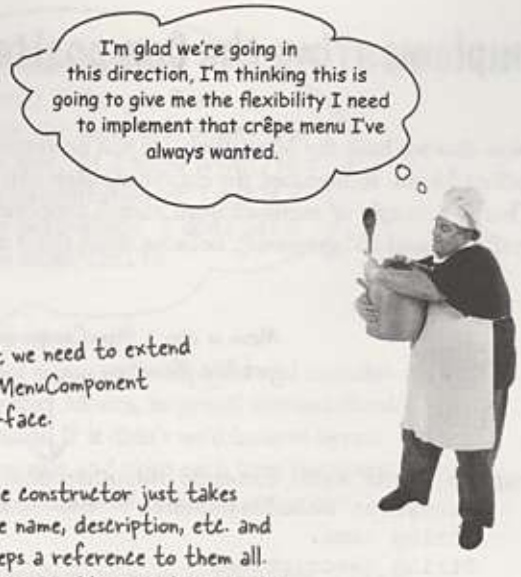
We've grouped together the "composite" methods – that is, methods to add, remove and get MenuComponents.

Here are the "operation" methods; these are used by the MenuItems. It turns out we can also use a couple of them in Menu too, as you'll see in a couple of pages when we show the Menu code.

print() is an "operation" method that both our Menus and MenuItems will implement, but we provide a default operation here.

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.




```
public class MenuItem extends MenuComponent {
    String name;
    String description;
    boolean vegetarian;
    double price;

    public MenuItem(String name,
                    String description,
                    boolean vegetarian,
                    double price)
    {
        this.name = name;
        this.description = description;
        this.vegetarian = vegetarian;
        this.price = price;
    }

    public String getName() {
        return name;
    }

    public String getDescription() {
        return description;
    }

    public double getPrice() {
        return price;
    }

    public boolean isVegetarian() {
        return vegetarian;
    }

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

First we need to extend the MenuComponent interface.

The constructor just takes the name, description, etc. and keeps a reference to them all. This is pretty much like our old menu item implementation.

Here's our getter methods – just like our previous implementation.

This is different from the previous implementation. Here we're overriding the print() method in the MenuComponent class. For MenuItem this method prints the complete menu entry: name, description, price and whether or not it's veggie.

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.

Menu is also a MenuComponent, just like MenuItem.

Menu can have any number of children of type MenuComponent, we'll use an internal ArrayList to hold these.

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

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

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

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

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

    public String getName() {
        return name;
    }

    public String getDescription() {
        return description;
    }

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

This is different than our old implementation: we're going to give each Menu a name and a description. Before, we just relied on having different classes for each menu.

Here's how you add MenuItems or other Menus to a Menu. Because both MenuItems and Menus are MenuComponents, we just need one method to do both.

You can also remove a MenuComponent or get a MenuComponent.

Here are the getter methods for getting the name and description.

Notice, we aren't overriding getPrice() or isVegetarian() because those methods don't make sense for a Menu (although you could argue that isVegetarian() might make sense). If someone tries to call those methods on a Menu, they'll get an UnsupportedOperationException.

To print the Menu, we print the Menu's name and description.

Wait a sec, I don't understand the implementation of print(). I thought I was supposed to be able to apply the same operations to a composite that I could to a leaf. If I apply print() to a composite with this implementation, all I get is a simple menu name and description. I don't get a printout of the COMPOSITE.

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

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

   // constructor code here

   // other methods here

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

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

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

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

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

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:

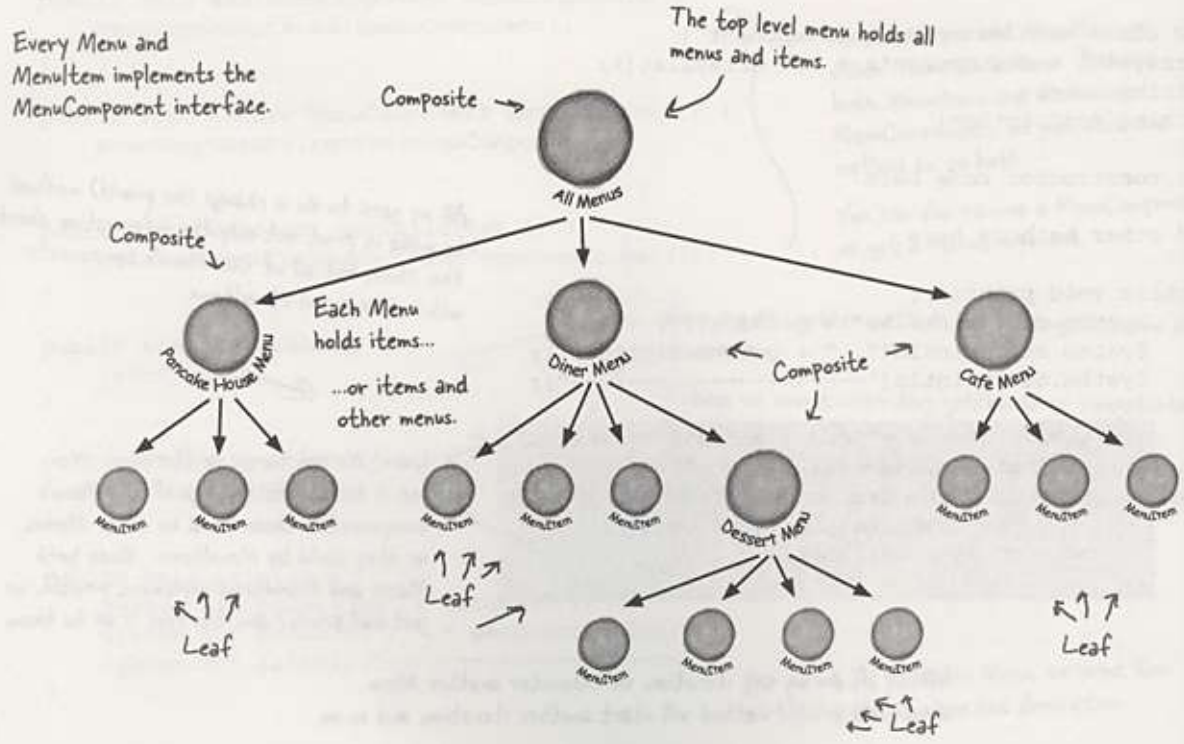
```
public class Waitress {
    MenuComponent allMenus;

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

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

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

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

We're gonna have one happy Waitress.

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 icecream",
           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 two top level menu now that we'll name allMenus.

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

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

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

Add some apple pie to the dessert menu...

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

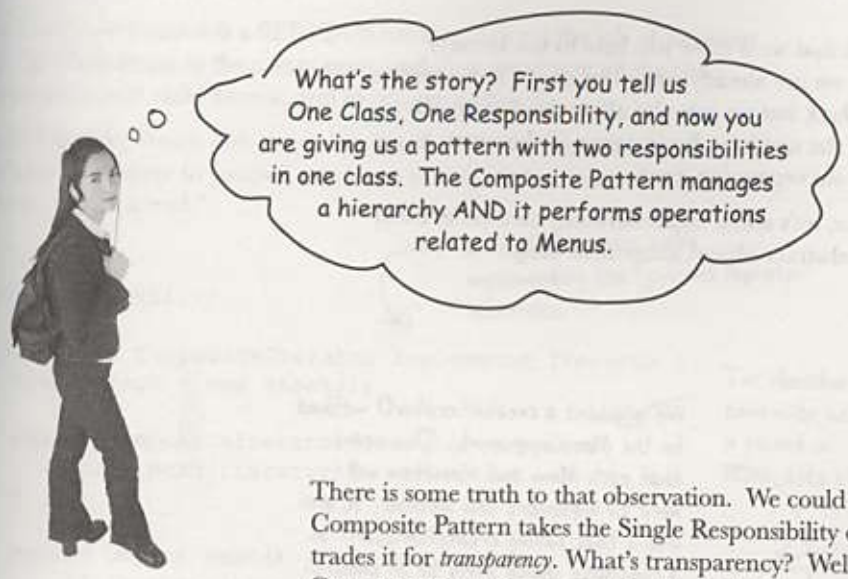
Getting ready for a test drive...

NOTE: this output is based on the complete source.

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

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

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

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

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

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

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

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



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.