finished 1st round this whole book on 14 - July - 09 3:49 AM

decoupled - there is one class between 2 different class
.
loosecoupled - two class know each other by abstract or interface
.
tightcoupled - two classes know each other by class reference variable

here after Composite Design pattern discussed...

composite pattern might be used in UDDI's tModel tree structure implemenation

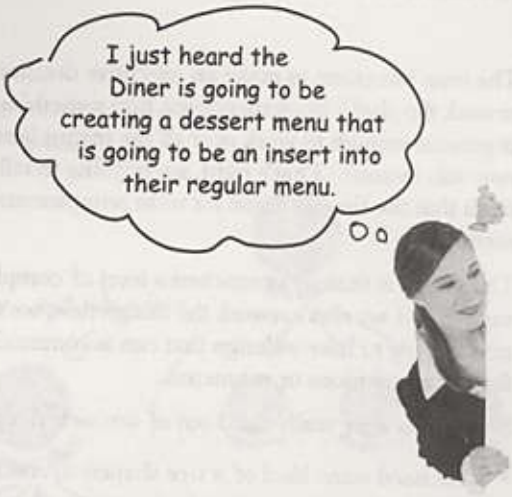
Just when we thought it was safe...

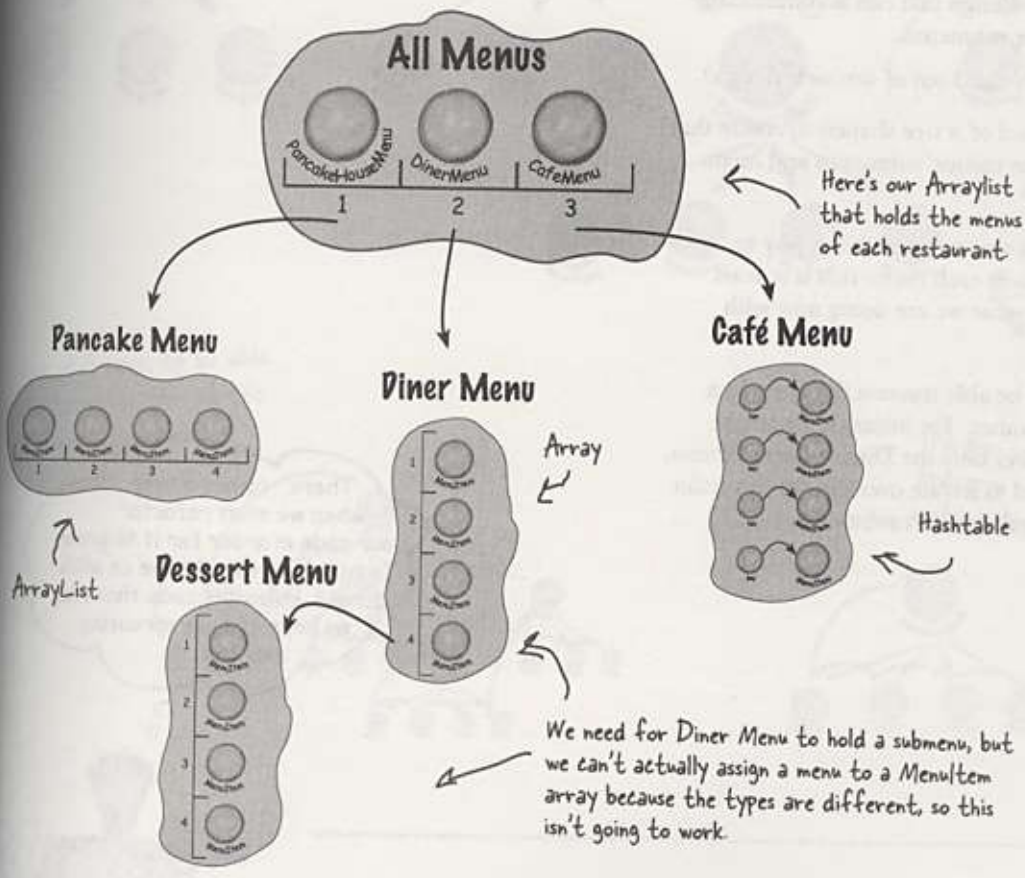
Now they want to add a dessert submenu.

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!

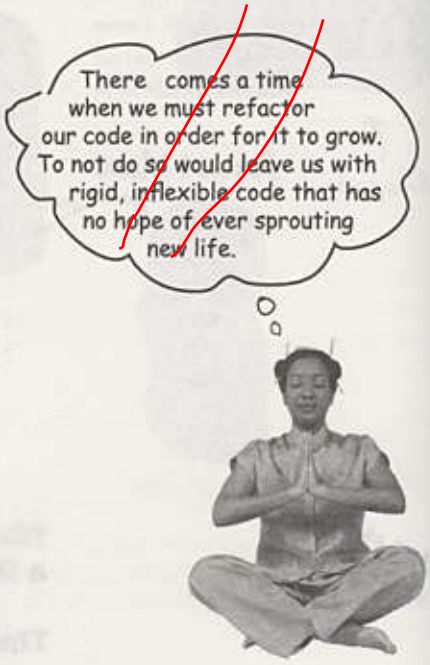
What do we need?

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.

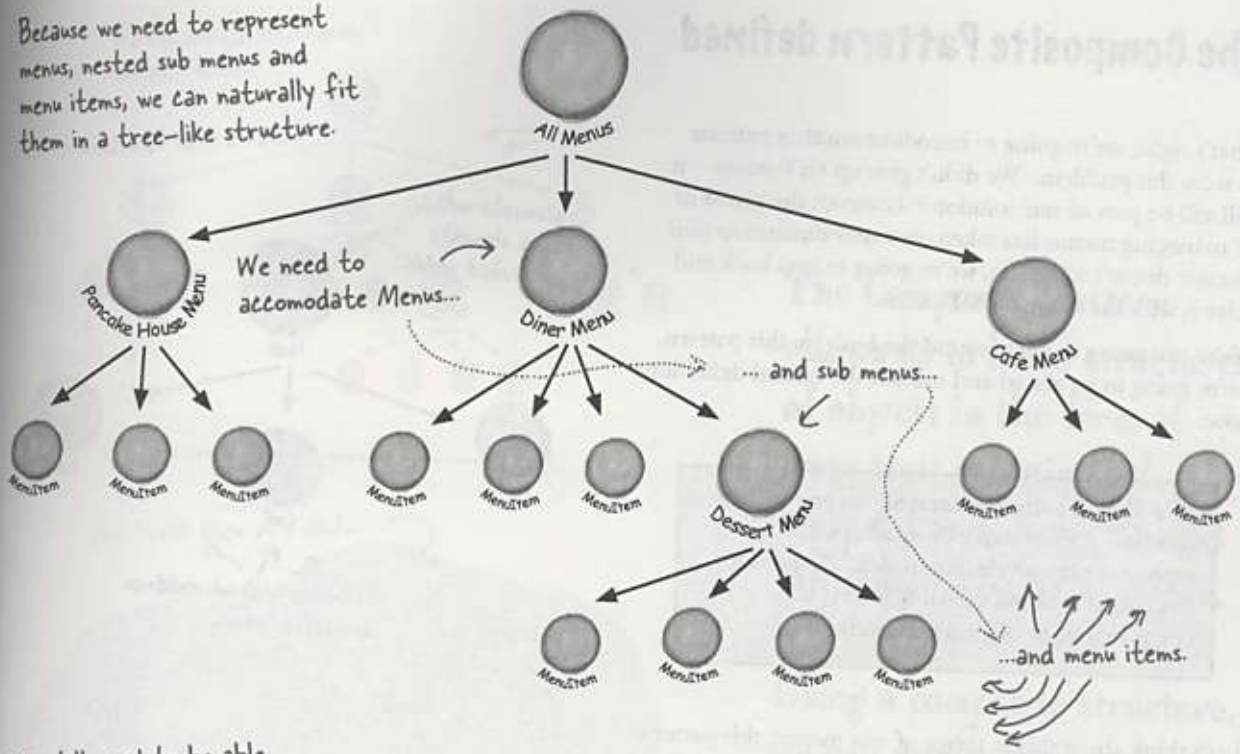
The reality is that we've reached a level of complexity such that if we don't rework the design now, we're never going to have a design that can accommodate further acquisitions or submenus.

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

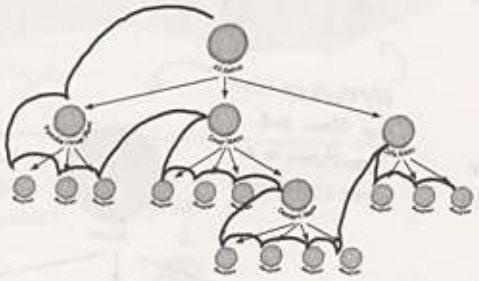
- We need some kind of a tree shaped structure that will accommodate menus, submenus and menu items.
- We need to make sure we maintain a way to traverse the items in each menu that is at least as convenient as what we are doing now with iterators.
- We may need to be able traverse the items in a more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



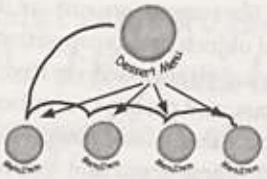
Because we need to represent menus, nested sub menus and menu items, we can naturally fit them in a tree-like structure.



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

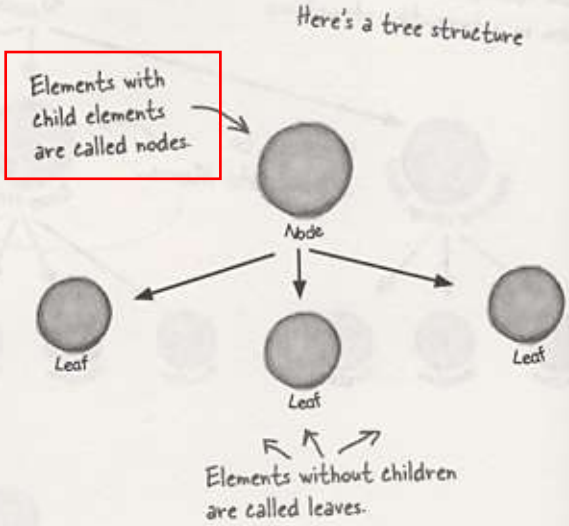
How would you handle this new wrinkle to our design requirements? Think about it before turning the page.

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.

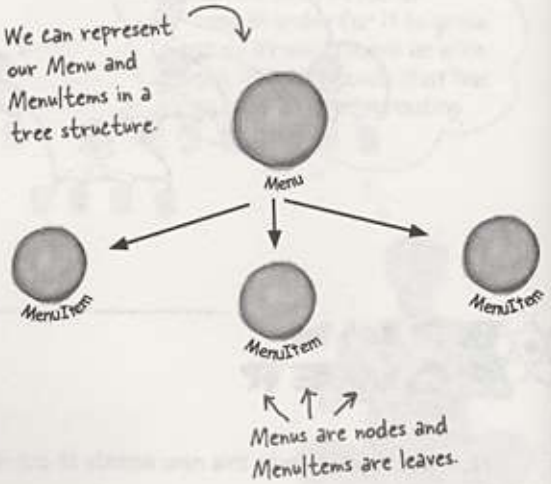
We're not going to beat around the bush on this pattern, we're going to go ahead and roll out the official definition now:

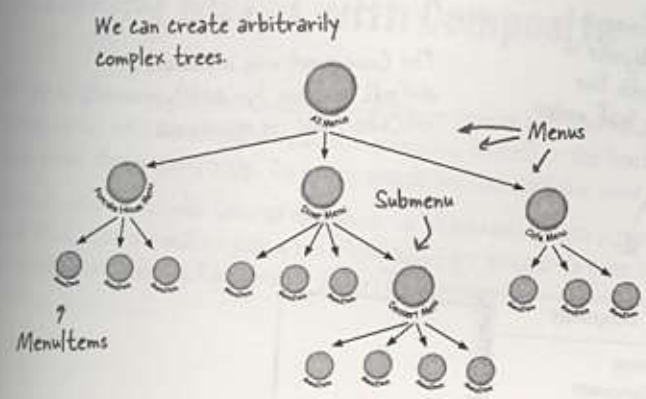
> **The Composite Pattern** allows you to compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

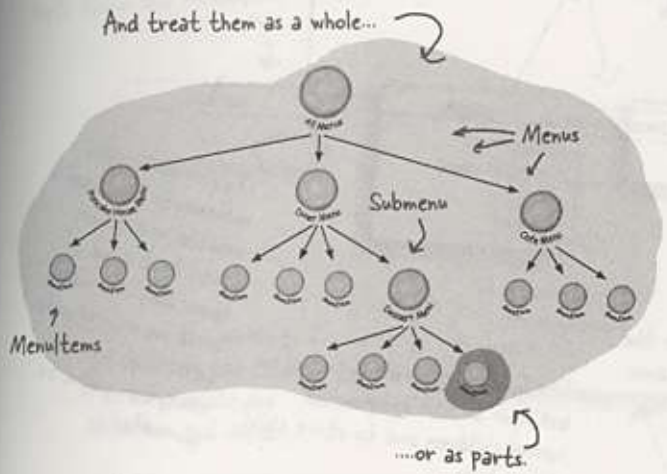


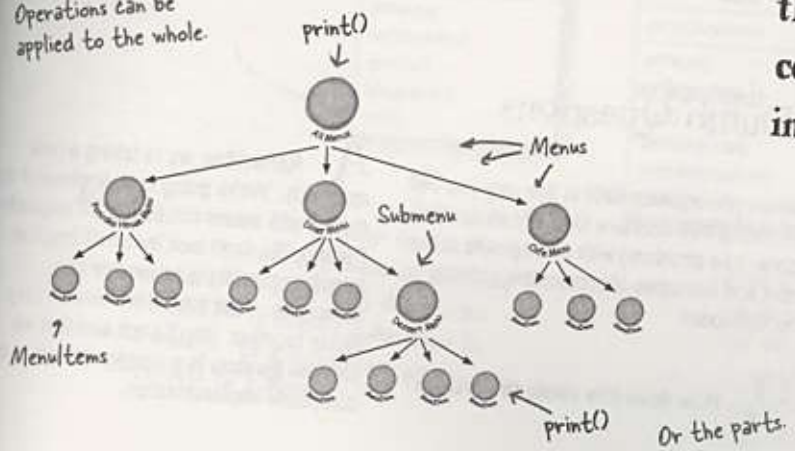
Let's think about this in terms of our menus: this pattern gives us a way to create a tree structure that can handle a nested group of menus *and* menu items in the same structure. By putting menus and items in the same structure we create a part-whole hierarchy; that is, a tree of objects that is made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



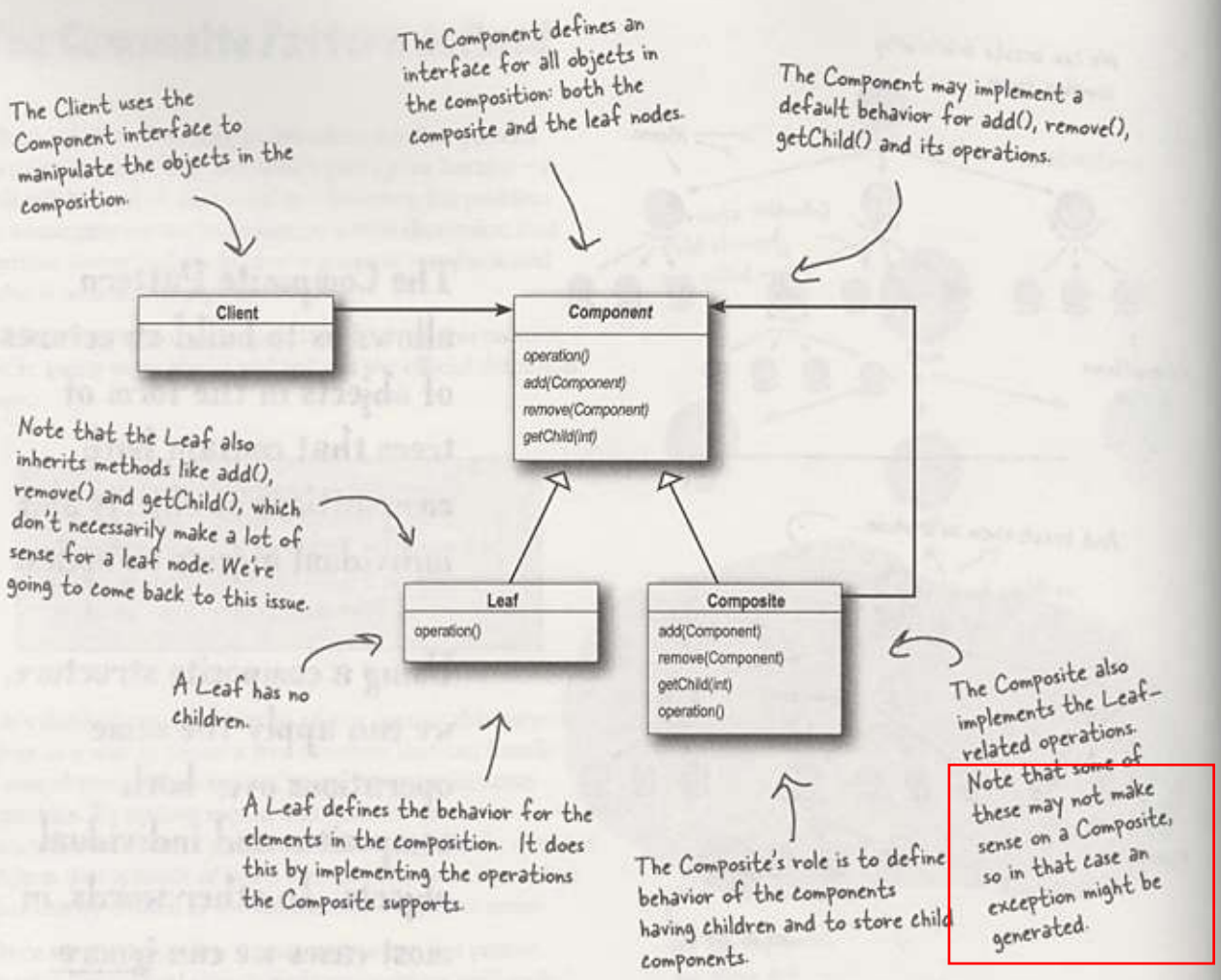






The Composite Pattern allows us to build structures of objects in the form of trees that contain both compositions of objects and individual objects as nodes.

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.



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.

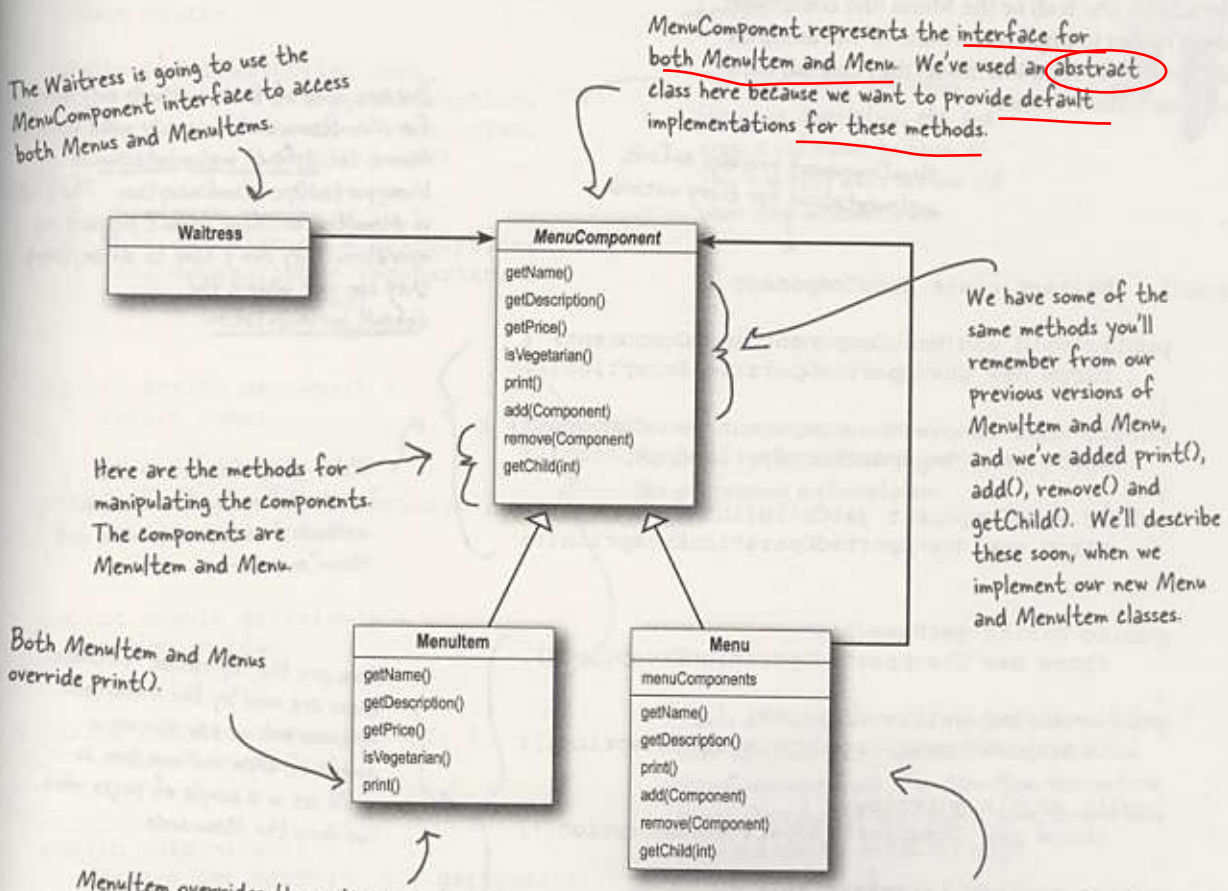
Q: How does this relate to iterators?

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.

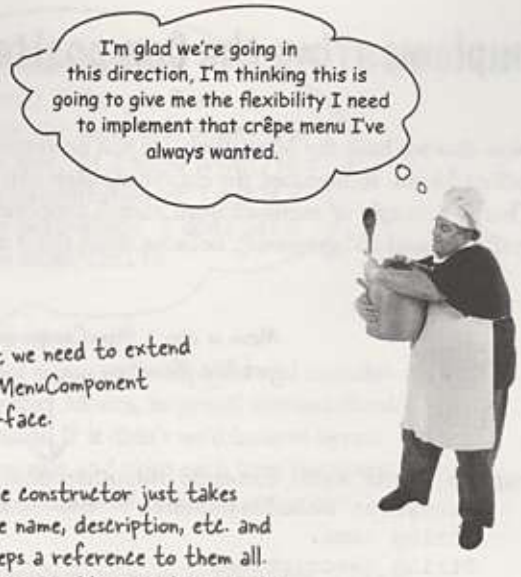
We've grouped together the "composite" methods – that is, methods to add, remove and get MenuComponents.

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:

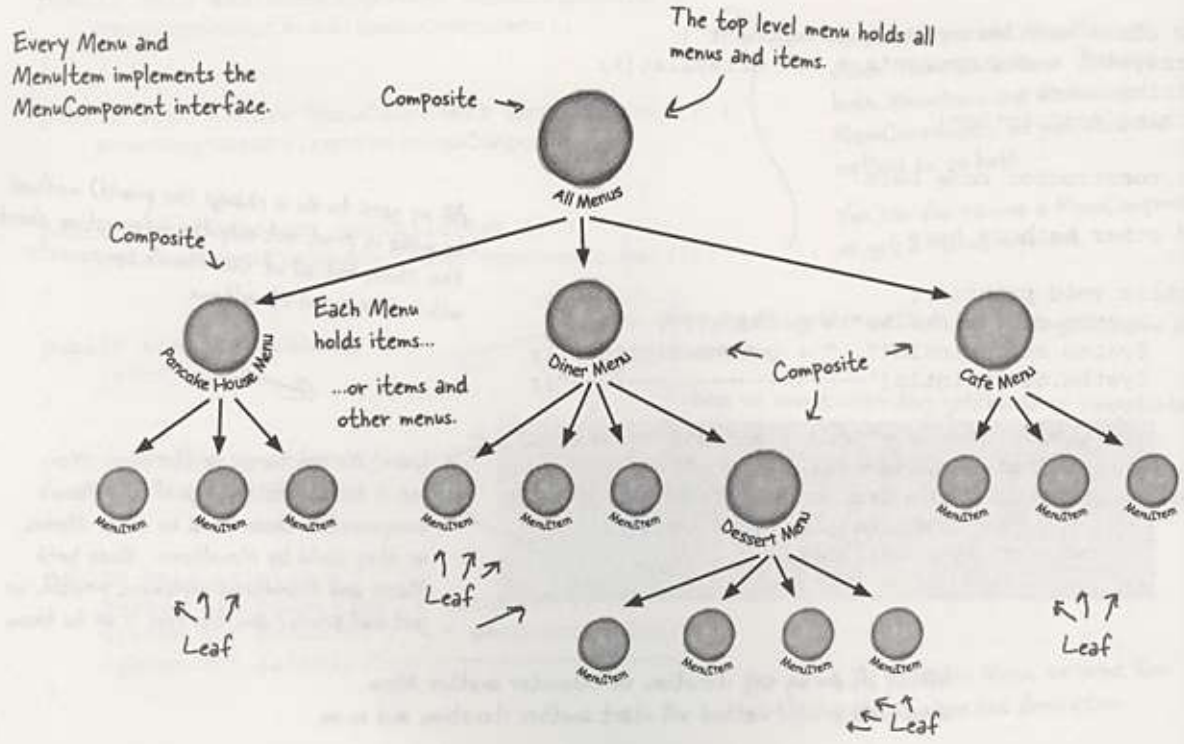
```
public class Waitress {
    MenuComponent allMenus;

    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.

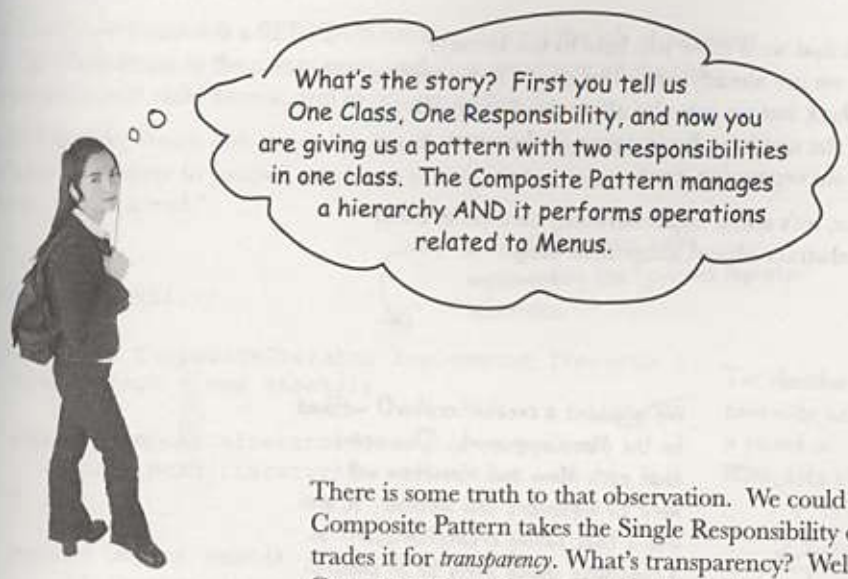
Getting ready for a test drive...

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.

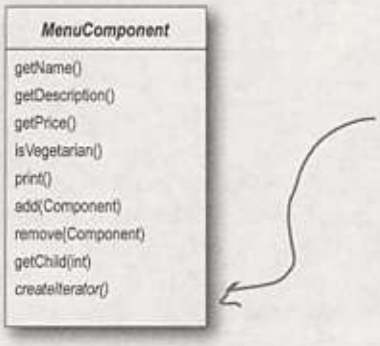
Now given we have both types of operations in the Component class, we lose a bit of *safety* because a client might try to do something inappropriate or meaningless on an element (like try to add a menu to a menu item). This is a design decision; we could take the design in the other direction and separate 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.

Flashback to Iterator

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

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



We've added a createIterator() method to the MenuComponent. This means that each Menu and MenuItem will need to implement this method. It also means that calling createIterator() on a composite should apply to all children of the composite.

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

```
public class Menu extends MenuComponent {

    // other code here doesn't change

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

}


public class MenuItem extends MenuComponent {

    // other code here doesn't change

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

}
```

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

Now for the MenuItem...

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

The Composite Iterator

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

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

WATCH OUT: RECURSION ZONE AHEAD

Like all iterators, we're implementing the java.util.Iterator interface.

```
import java.util.*;

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

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

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

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

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

The iterator of the top level composite we're going to iterate over is passed in. We throw that in a stack data structure.

Okay, when the client wants to get the next element we first make sure there is one by calling hasNext()...

If there is a next element, we get the current iterator off the stack and get its next element.

If that element is a menu, we have another composite that needs to be included in the iteration, so we throw it on the stack. In either case, we return the component.

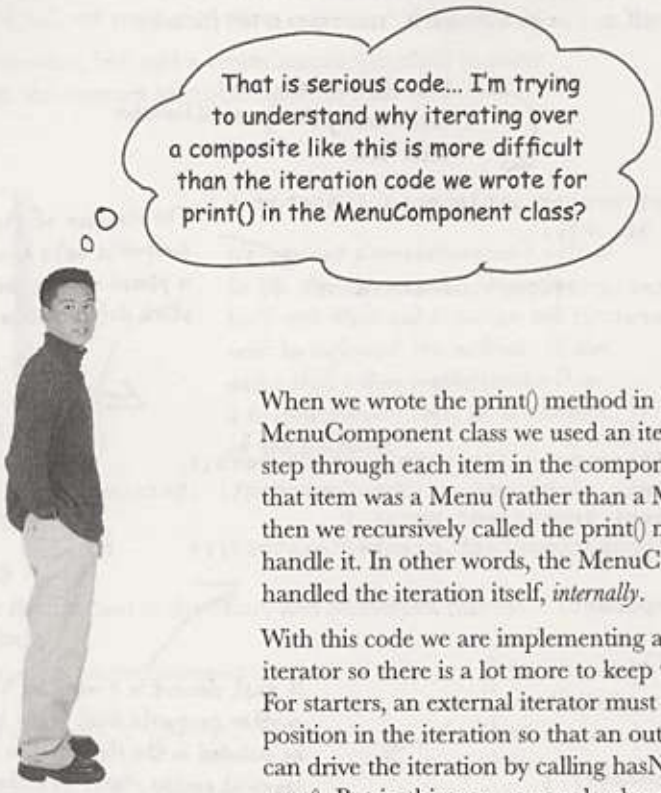
it is really good code. try to understand next round 10 / 7 / 09

To see if there is a next element, we check to see if the stack is empty; if so, there isn't.

Otherwise, we get the iterator off the top of the stack and see if it has a next element. If it doesn't we pop it off the stack and call hasNext() recursively.

Otherwise there is a next element and we return true.

We're not supporting remove, just traversal.



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

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

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

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

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

The Null Iterator

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

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

Choice one:

Return null

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

Choice two:

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

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

super...

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

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

```
import java.util.Iterator;

public class NullIterator implements Iterator {

    public Object next() {
        return null;
    }

    public boolean hasNext() {
        return false;
    }

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

When next() is called, we return null.

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

And the NullIterator wouldn't think of supporting remove.

Give me the vegetarian menu

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

```
public class Waitress {
    MenuComponent allMenus;

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

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

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

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

Iterate through every element of the composite.

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

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

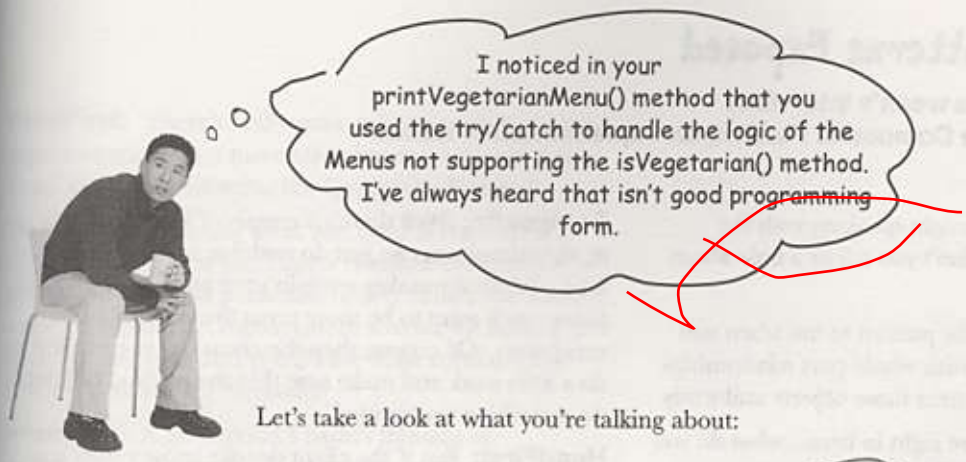
We implemented isVegetarian() on the Menus to always throw an exception. If that happens we catch the exception, but continue with our iteration.

The magic of Iterator & Composite together...

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

```
File Edit Window Help HaveUhuggedYuriteratorToday?
% java MenuTestDrive
VEGETARIAN MENU
----
  K&B's Pancake Breakfast(v), 2.99
     -- Pancakes with scrambled eggs, and toast
  Blueberry Pancakes(v), 3.49
     -- Pancakes made with fresh blueberries, and blueberry syrup
  Waffles(v), 3.59
     -- Waffles, with your choice of blueberries or strawberries
  Vegetarian BLT(v), 2.99
     -- (Fakin') Bacon with lettuce & tomato on whole wheat
  Steamed Veggies and Brown Rice(v), 3.99
     -- Steamed vegetables over brown rice
  Pasta(v), 3.89
     -- Spaghetti with Marinara Sauce, and a slice of sourdough bread
  Apple Pie(v), 1.59
     -- Apple pie with a flakey crust, topped with vanilla 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
  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
  Veggie Burger and Air Fries(v), 3.99
     -- Veggie burger on a whole wheat bun, lettuce, tomato, and fries
  Burrito(v), 4.29
     -- A large burrito, with whole pinto beans, salsa, guacamole
%
```

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



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

```
try {
    if (menuComponent.isVegetarian()) {
        menuComponent.print();
    }
} catch (UnsupportedOperationException) {}
```

We call isVegetarian() on all MenuComponents, but Menus throw an exception because they don't support the operation.

If the menu component doesn't support the operation, we just throw away the exception and ignore it.

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

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

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

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

so, there are some situation, to implement programe logic in CATCH block... wow..

Patterns Exposed

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

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

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

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

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

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

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

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

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

HeadFirst: So how do you handle that?

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

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

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

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

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

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

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

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

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

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

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

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

Composite: And did you think about caching?

HeadFirst: Caching?

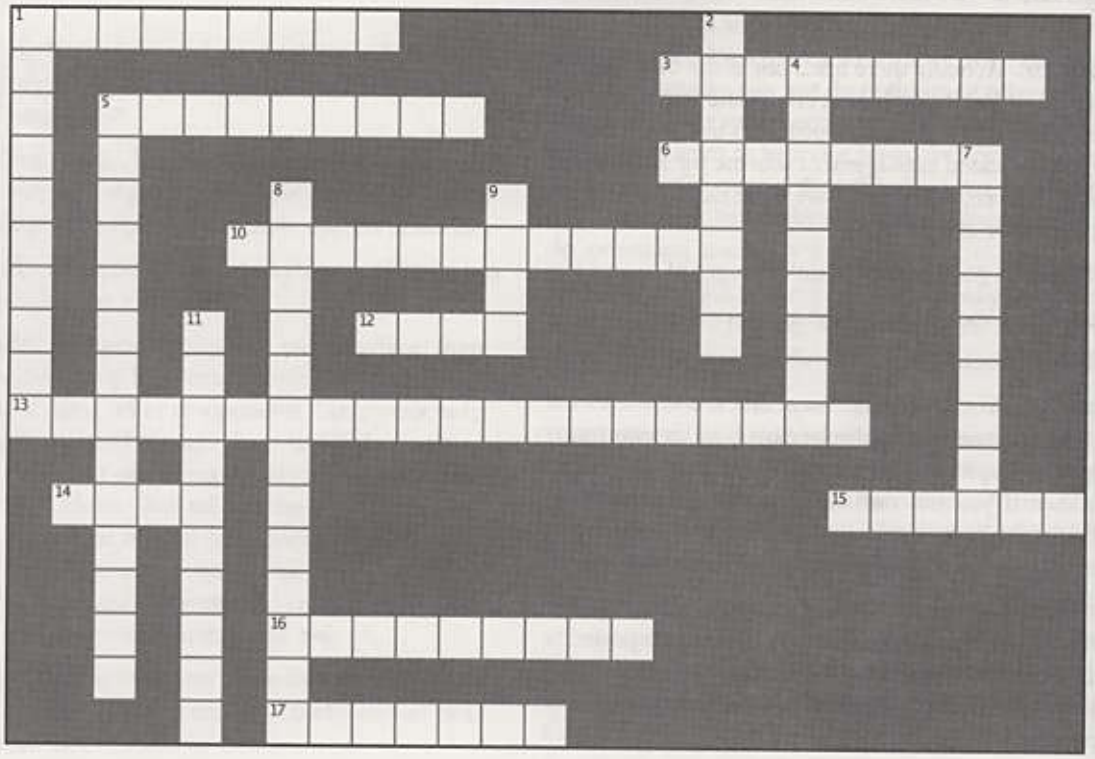
Composite: Yeah, caching. Sometimes, if the composite structure is complex or expensive to traverse, it's helpful to implement caching of the composite nodes. For instance, if you are constantly traversing a composite and all its children to compute some result, you could implement a cache that stores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

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

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

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

It's that time again....



Across

1. User interface packages often use this pattern for their components.
3. Collection and Iterator are in this package
5. We encapsulated this.
6. A separate object that can traverse a collection.
10. Merged with the Diner.
12. Has no children.
13. Name of principle that states only one responsibility per class.
14. Third company acquired.
15. A class should have only one reason to do this.
16. This class indirectly supports Iterator.
17. This menu caused us to change our entire implementation.

Down

1. A composite holds this.
2. We java-enabled her.
4. We deleted PancakeHouseMenuIterator because this class already provides an iterator.
5. The Iterator Pattern decouples the client from the aggregates _____.
7. CompositeIterator used a lot of this.
8. Iterators are usually created using this pattern.
9. A component can be a composite or this.
11. Hashtable and ArrayList both implement this interface.

WHO DOES WHAT?

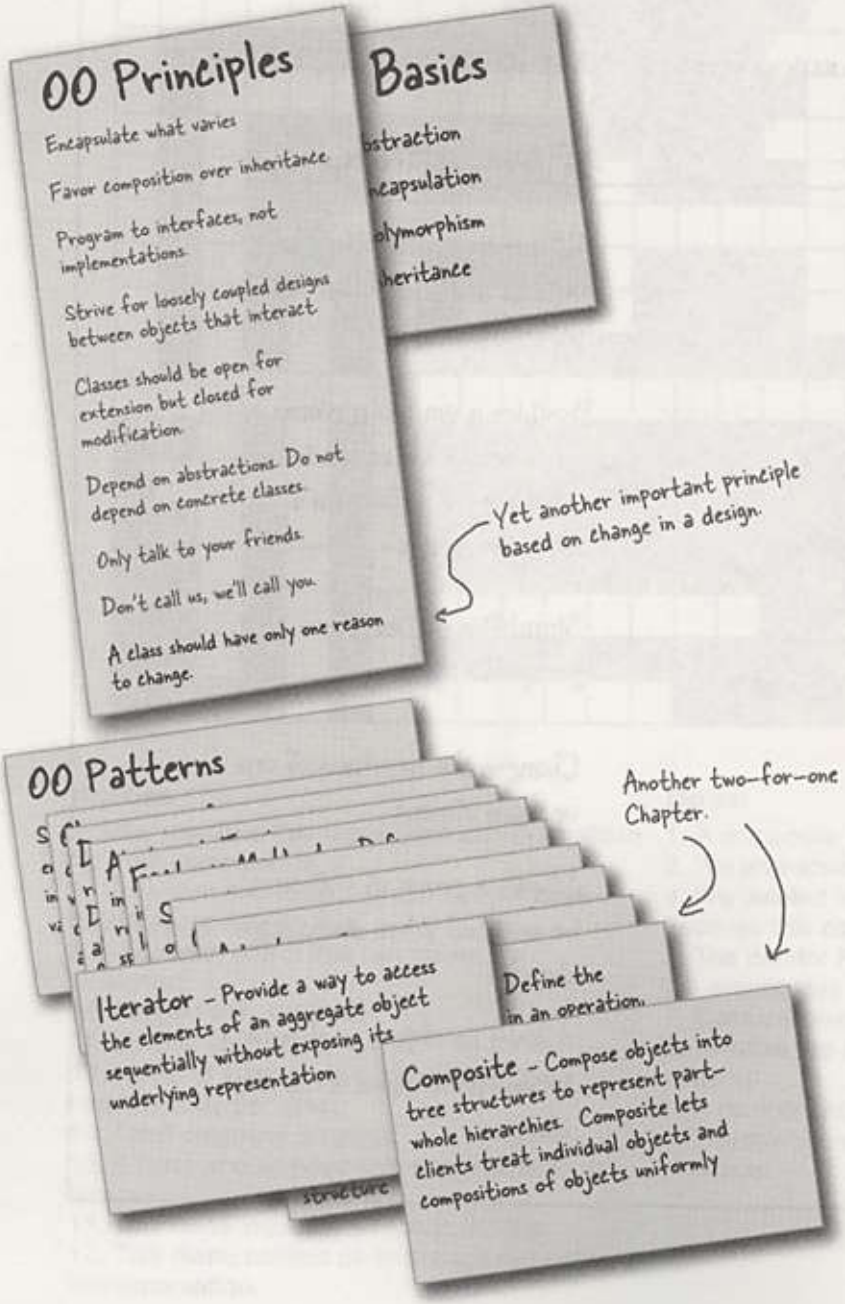
Match each pattern with its description:

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

composite pattern might be used in UDDI's tModel tree structure implemenation

Tools for your Design Toolbox

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



BULLET POINTS

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

Exercise solutions

Sharpen your pencil

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

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

Sharpen your pencil

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

1. implement the Menu interface
2. get rid of getItems()
3. add createIterator() and return an Iterator that can step through the Hashtable values

Code Magnets Solution

The unscrambled "Alternating" DinerMenu Iterator

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

public class AlternatingDinerMenuIterator implements Iterator {

    MenuItem[] items;
    int position;

    public AlternatingDinerMenuIterator(MenuItem[] items) {

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

    }

    public boolean hasNext() {

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

    }

    public Object next() {

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

    }

    public void remove() {

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

    }

}
```

Notice that this Iterator implementation does not support remove()

WHO DOES WHAT?

Match each pattern with its description:

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

Exercise solutions