Lecture 11: The Composite Pattern

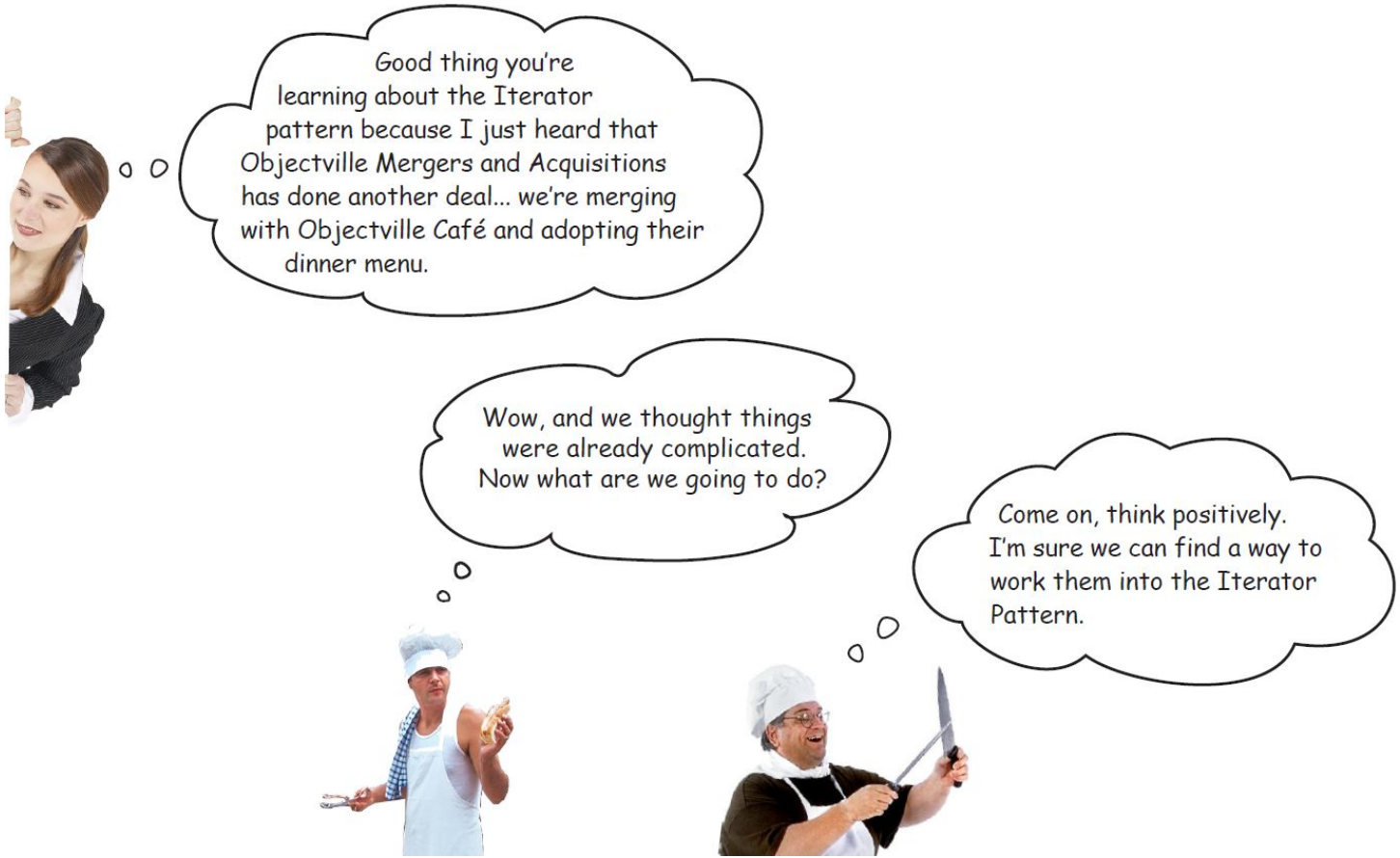
SE313, Software Design and Architecture

Damla Oguz

Chapter 9: The Composite Pattern

- In the previous lecture, we saw how we can allow our clients to iterate through our objects without showing how we store our objects.
- Let’s remember the Iterator Pattern…

A new menu


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

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

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

```
public class CafeMenu {
    HashMap<String, MenuItem> menuItems = new HashMap<String, MenuItem>();

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

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

Taking a look at the Café Menu (cont.)

```
    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 Map<String, MenuItem> getItems() {
        return menuItems;
    }
}
```

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.

Reworking the Café Menu code

- Integrating the CafeMenu into our framework is easy.
- Because HashMap is one of those Java collections that supports Iterator.
- But it's not quite the same as ArrayList.
- HashMap 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<MenuItem> 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.

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

```
public class CafeMenu implements Menu {
    HashMap<String, MenuItem> menuItems = new HashMap<String, MenuItem>();

    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 Map<String, MenuItem> getItems() {
        return menuItems;
    }

    public Iterator<MenuItem> 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 HashMap, just for the values.

Adding the Café Menu to the Waitress

```
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;
    }
```

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

```
    public void printMenu() {
        Iterator<MenuItem> pancakeIterator = pancakeHouseMenu.createIterator();
        Iterator<MenuItem> dinerIterator = dinerMenu.createIterator();
        Iterator<MenuItem> 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 = iterator.next();
            System.out.print(menuItem.getName() + ", ");
            System.out.print(menuItem.getPrice() + " -- ");
            System.out.println(menuItem.getDescription());
        }
    }
}
```

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

Nothing changes here.

Breakfast, lunch AND dinner

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

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

        waitress.printMenu();
}
```

Create a CafeMenu...

... and pass it to the waitress.

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

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

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

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

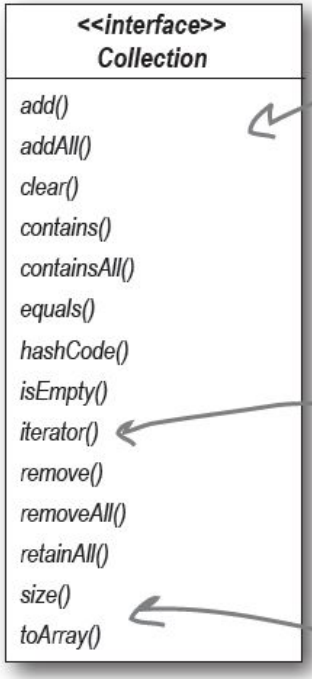
First we iterate through the pancake menu.

And then the diner menu.

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

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, such as ArrayList, 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 HashMap holds two sets of objects: keys and values. If we want to iterate over its values, we first need to retrieve them from the HashMap, and then obtain the iterator.

Iterators and Collections (cont.)

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.

Is the Waitress ready for prime time?

- 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”?

```
public void printMenu() {
    Iterator<MenuItem> pancakeIterator = pancakeHouseMenu.createIterator();
    Iterator<MenuItem> dinerIterator = dinerMenu.createIterator();
    Iterator<MenuItem> cafeIterator = cafeMenu.createIterator();

    System.out.println("MENU\n----\nBREAKFAST");
    printMenu(pancakeIterator);

    System.out.println("\nLUNCH");
    printMenu(dinerIterator);

    System.out.println("\nDINNER");
    printMenu(cafeIterator);
}
```

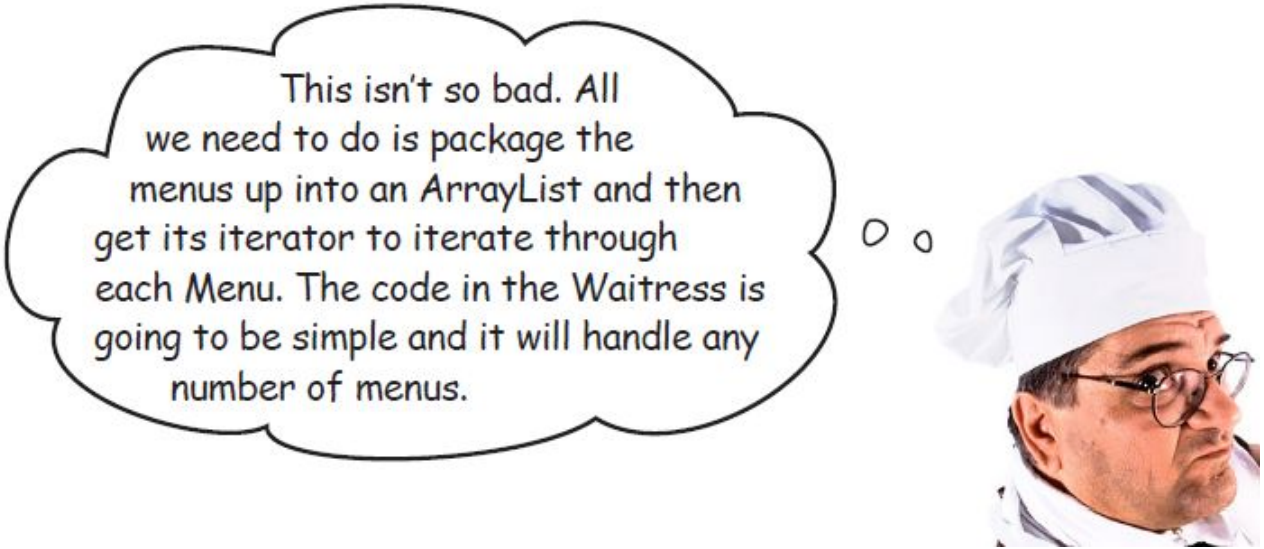
Three createIterator() calls.

Three calls to printMenu.

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

Is the Waitress ready for prime time? (cont.)

- We have done a great job of decoupling the menu implementation and extracting the iteration into an iterator.
- But we are still handling the menus with separate, independent objects—we need a way to manage them together.



```
public class Waitress {
    ArrayList<Menu> menus;

    public Waitress(ArrayList<Menu> menus) {
        this.menus = menus;
    }

    public void printMenu() {
        Iterator<Menu> menuIterator = menus.iterator();
        while(menuIterator.hasNext()) {
            Menu menu = menuIterator.next();
            printMenu(menu.createIterator());
        }
    }

    void printMenu(Iterator<Menu> iterator) {
        while (iterator.hasNext()) {
            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.

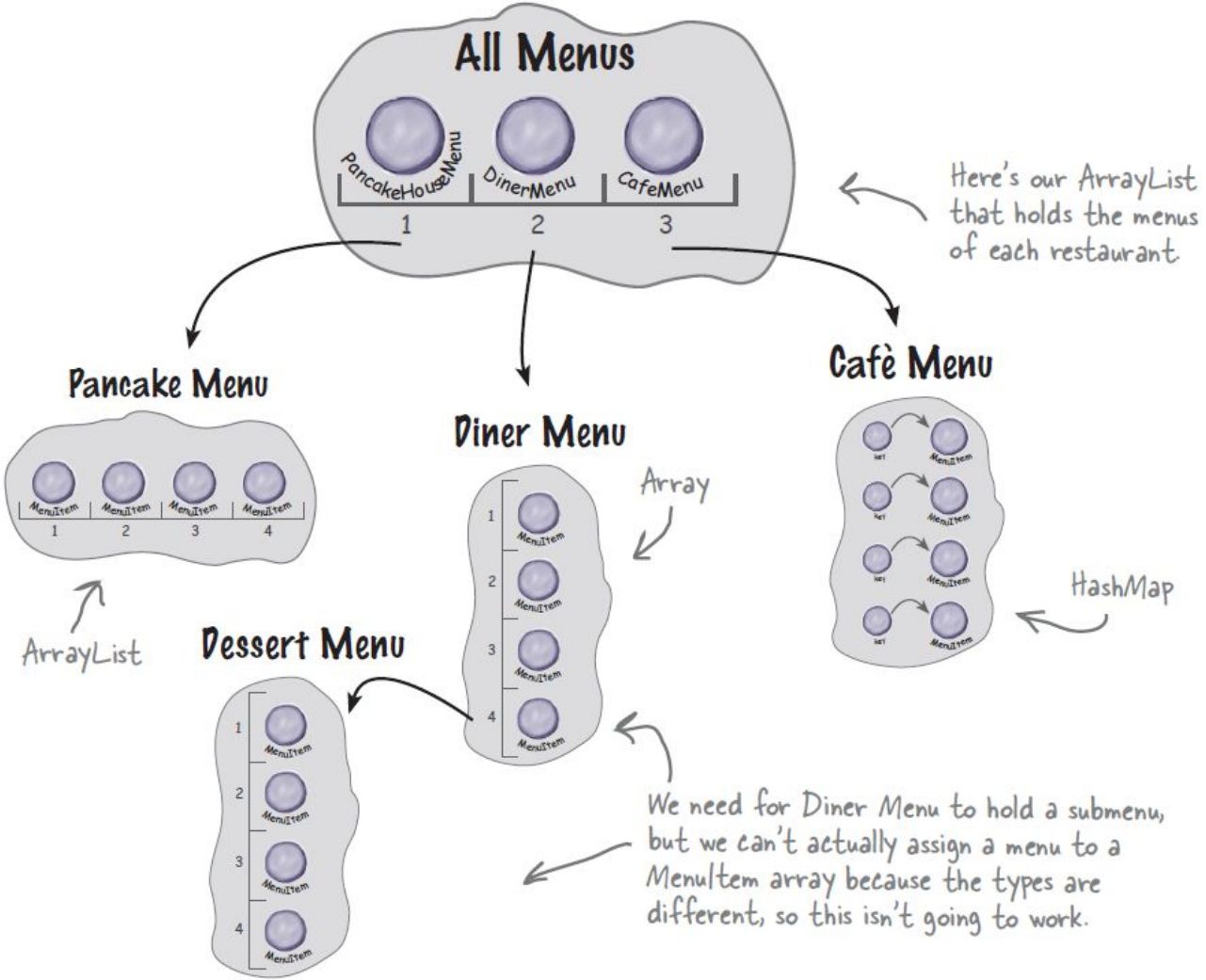
And we iterate through the menus, passing each menu's iterator to the overloaded printMenu() method.

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.

Now, they want to add a dessert submenu

- The Diner is going to be creating a dessert menu that is going to be an insert into their regular menu.
- 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)…



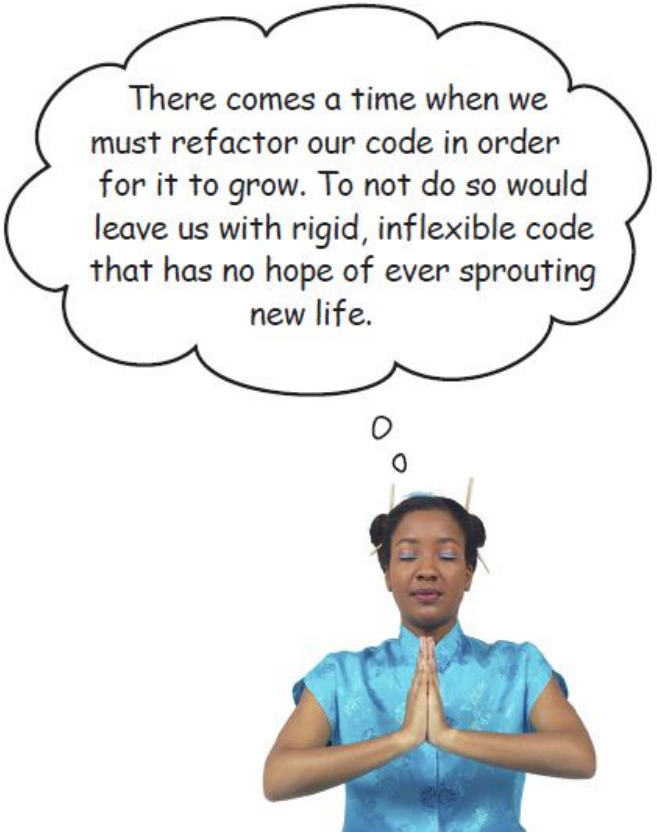
But this won't work!

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

Time for a change!

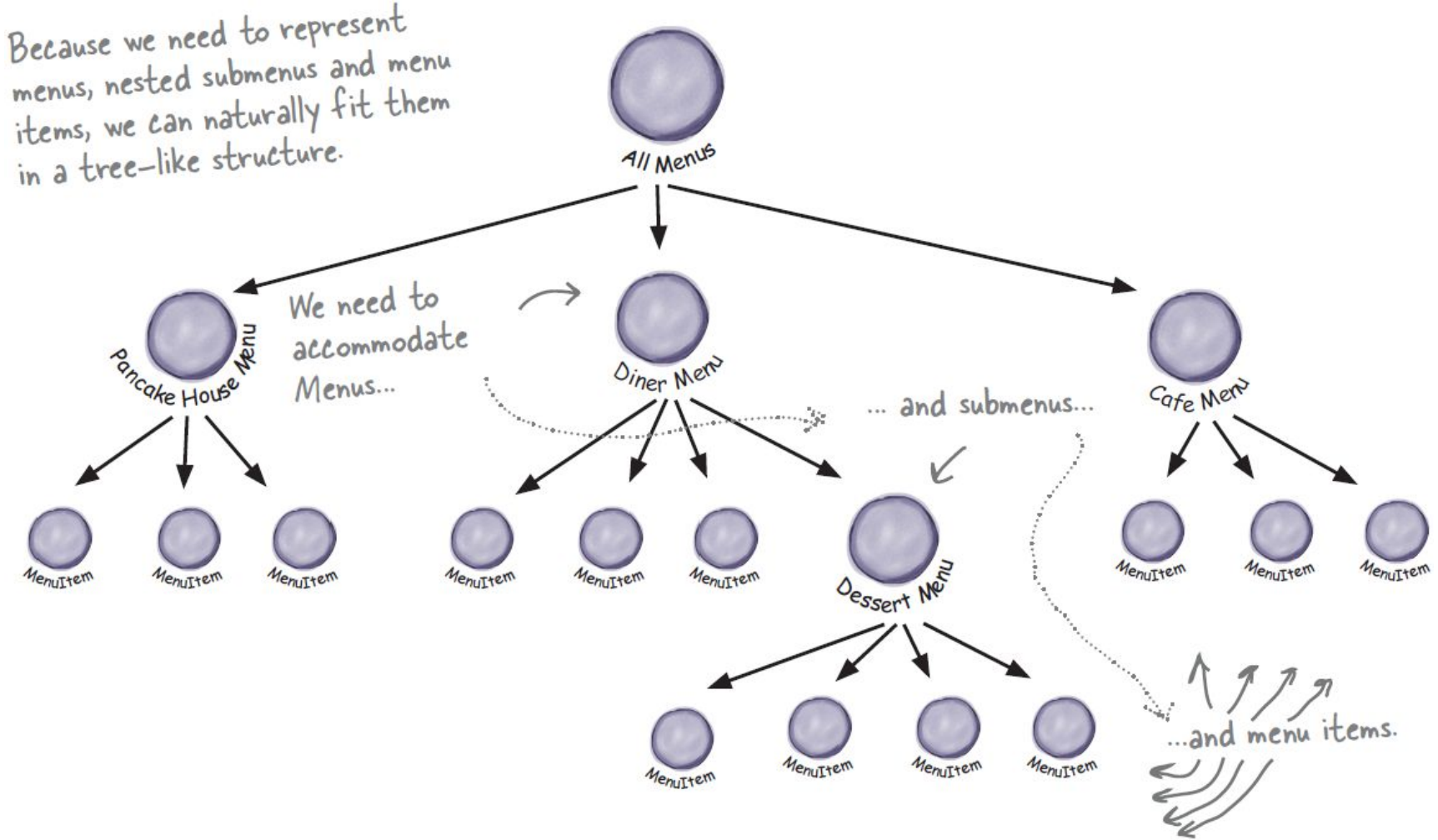
What do we need?

- We’re going to tell the chefs that the time has come for us to reimplement their menus.
- 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.

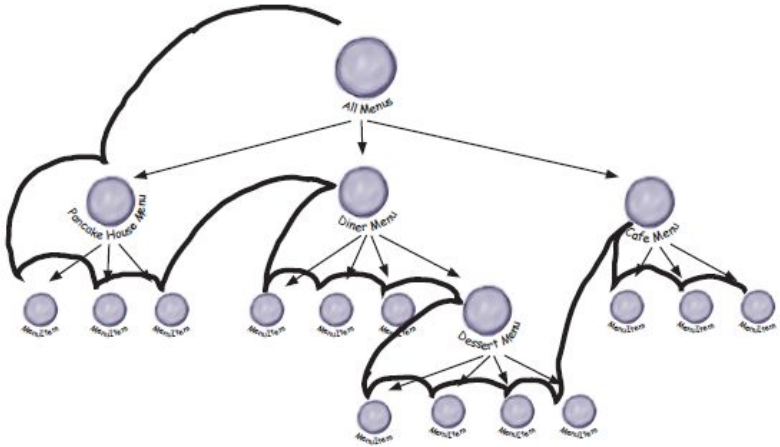


So, what is it we really need out of our new design?

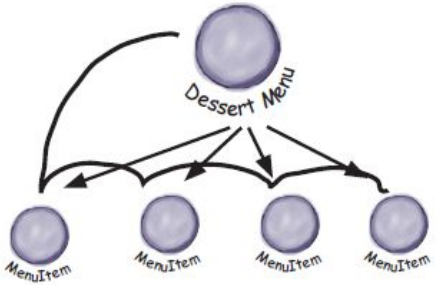
- We need some kind of a tree-shaped structure that will accommodate menus, submenus, and menu items.
- We need to make sure we maintain a way to traverse the items in each menu that is at least as convenient as what we are doing now with iterators.
- We may need to traverse the items in a more flexible 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 submenus and menu items, we can naturally fit them in a tree-like structure.
All Menus
Pancake House Menu
We need to accommodate Menus...
Diner Menu
... and submenus...
Cafe Menu
Dessert Menu
MenuItem
...and menu items.

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



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



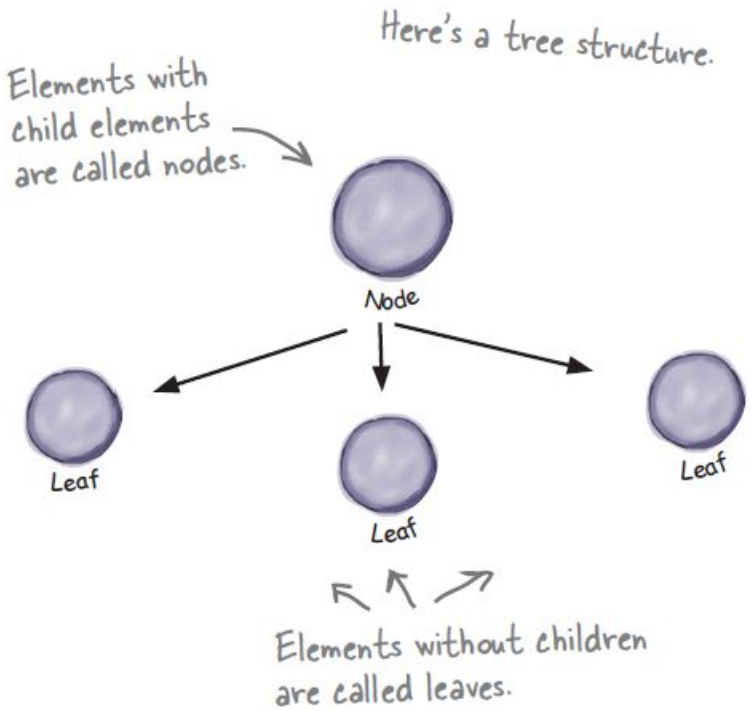
The Composite Pattern defined

- We’re going to introduce the Composite 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.

> **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.

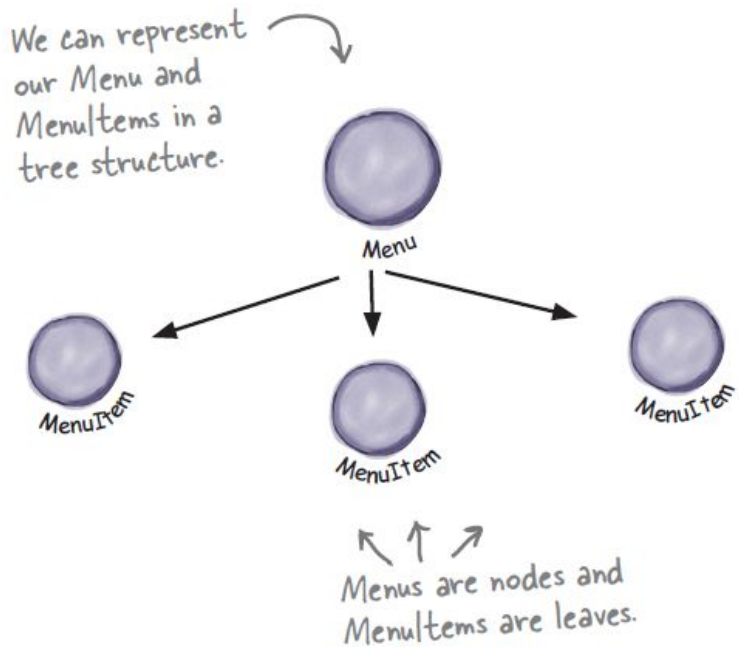
The Composite Pattern defined (cont.)

- 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 super menu.

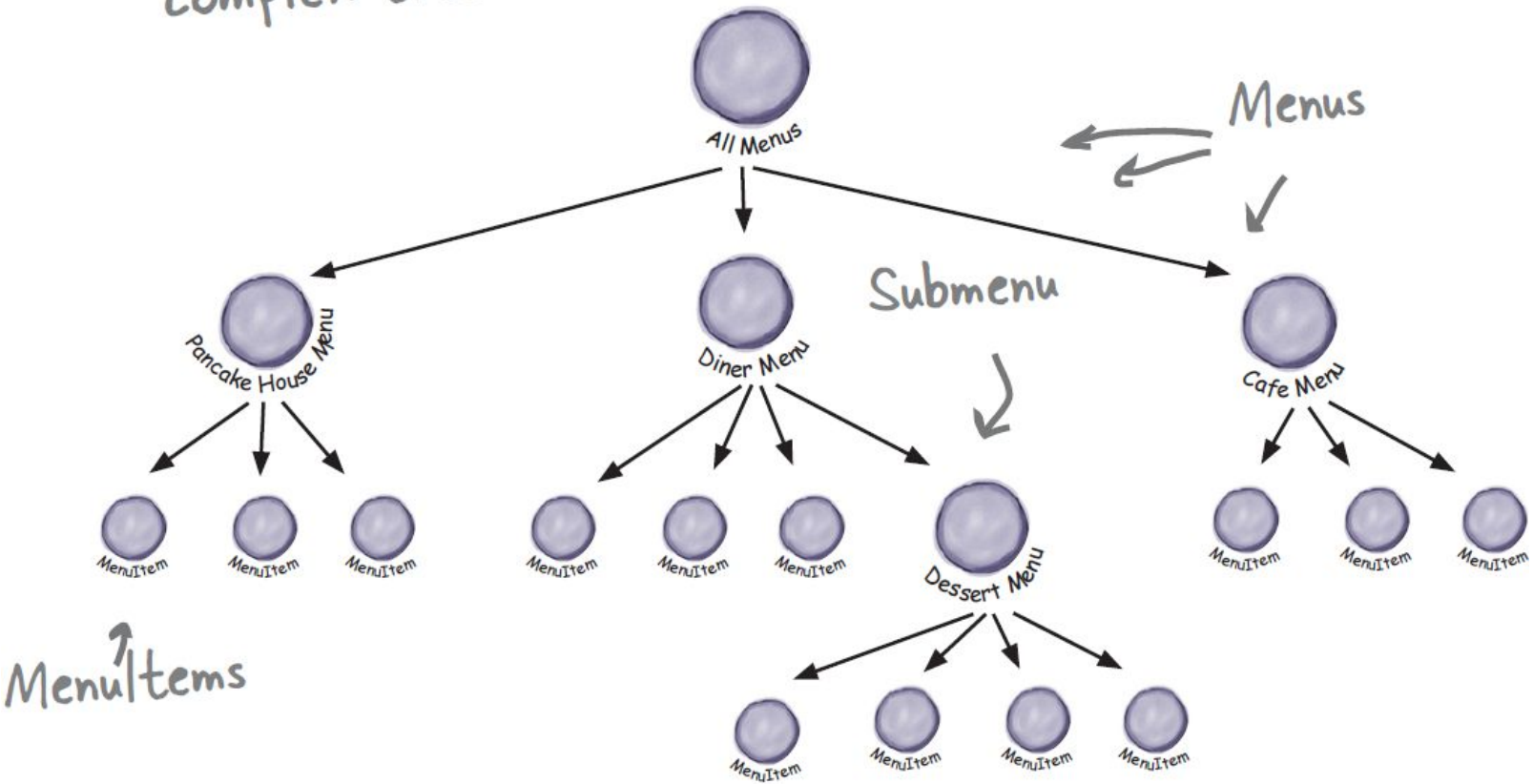


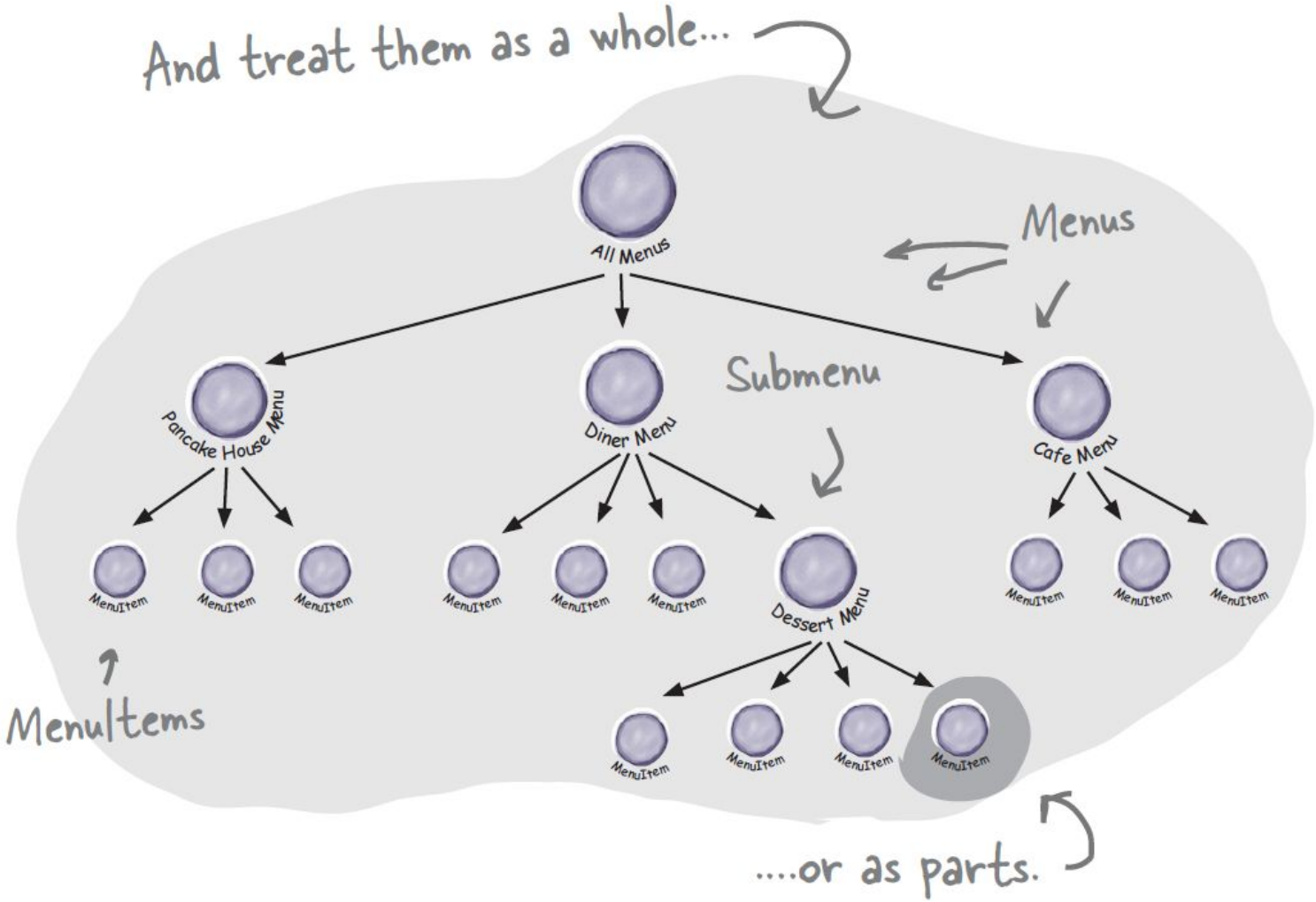
The Composite Pattern defined (cont.)

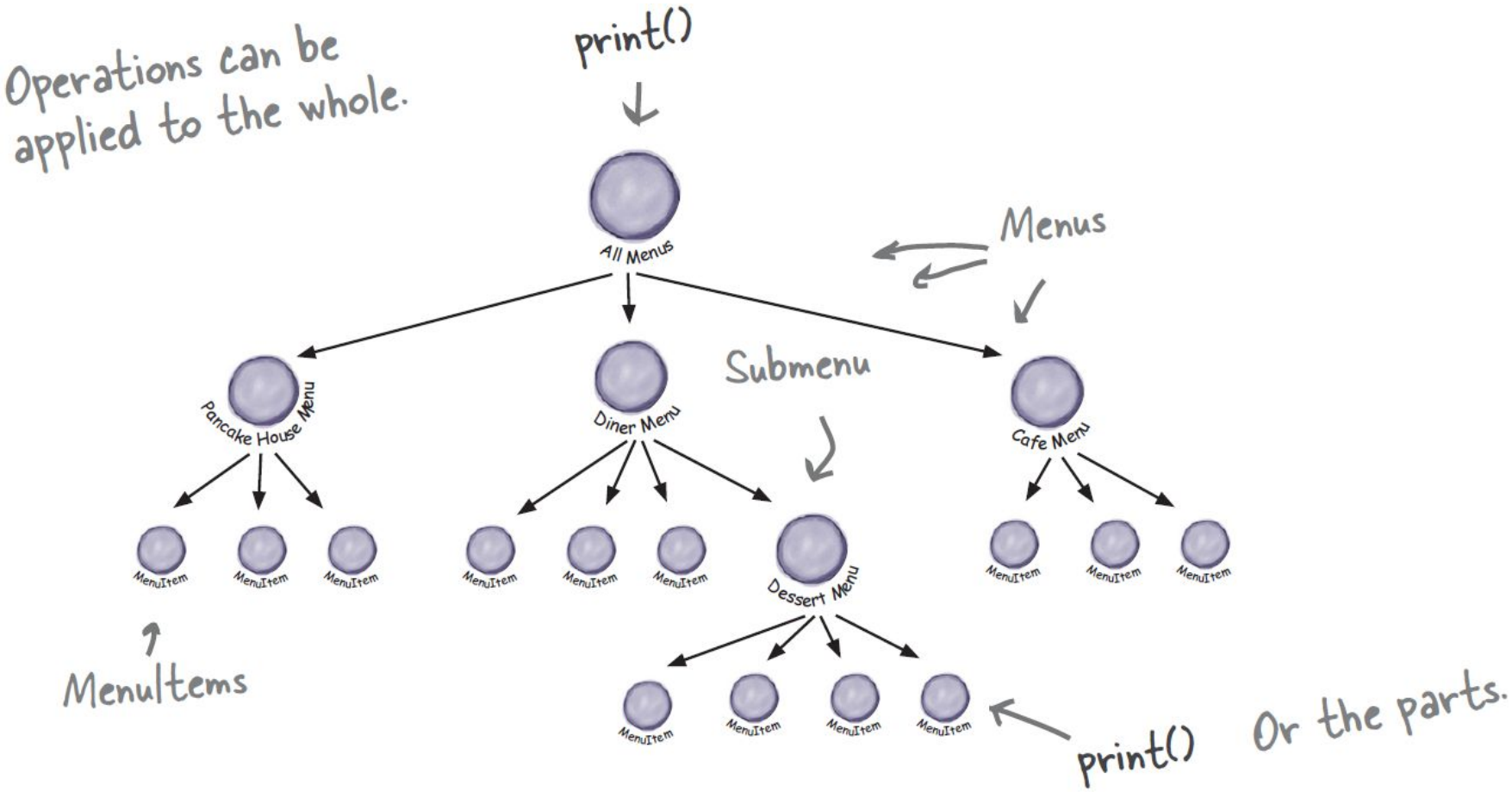
- Once we have our super menu, we can use this pattern to treat “individual objects and compositions uniformly.”
  - It means if we have a tree structure of menus, submenus, and perhaps subsubmenus along with menu items, then any menu is a “**composition**” because it can contain both other menus and menu items.
- The Composite Pattern allows us to write some simple code that can apply the same operation (like printing!) over the entire menu structure.



We can create arbitrarily complex trees.



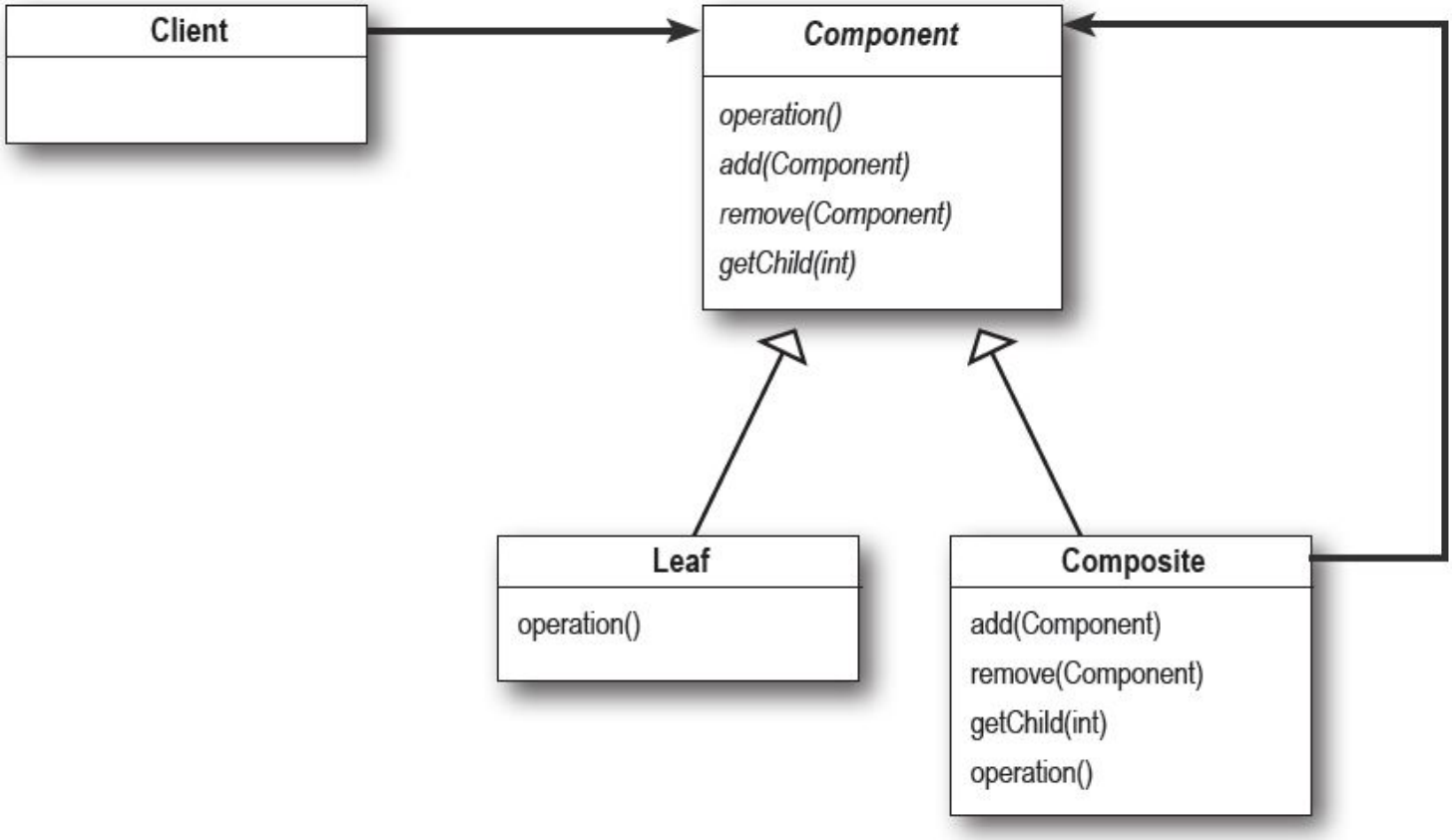

And treat them as a whole...
All Menus
Menus
Pancake House Menu
Diner Menu
Submenu
Cafe Menu
Dessert Menu
MenuItem
MenuItems
....or as parts.


Operations can be applied to the whole.
print()
All Menus
Menus
Pancake House Menu
Diner Menu
Submenu
Cafe Menu
Dessert Menu
MenuItem
MenuItems
print()
Or the parts.

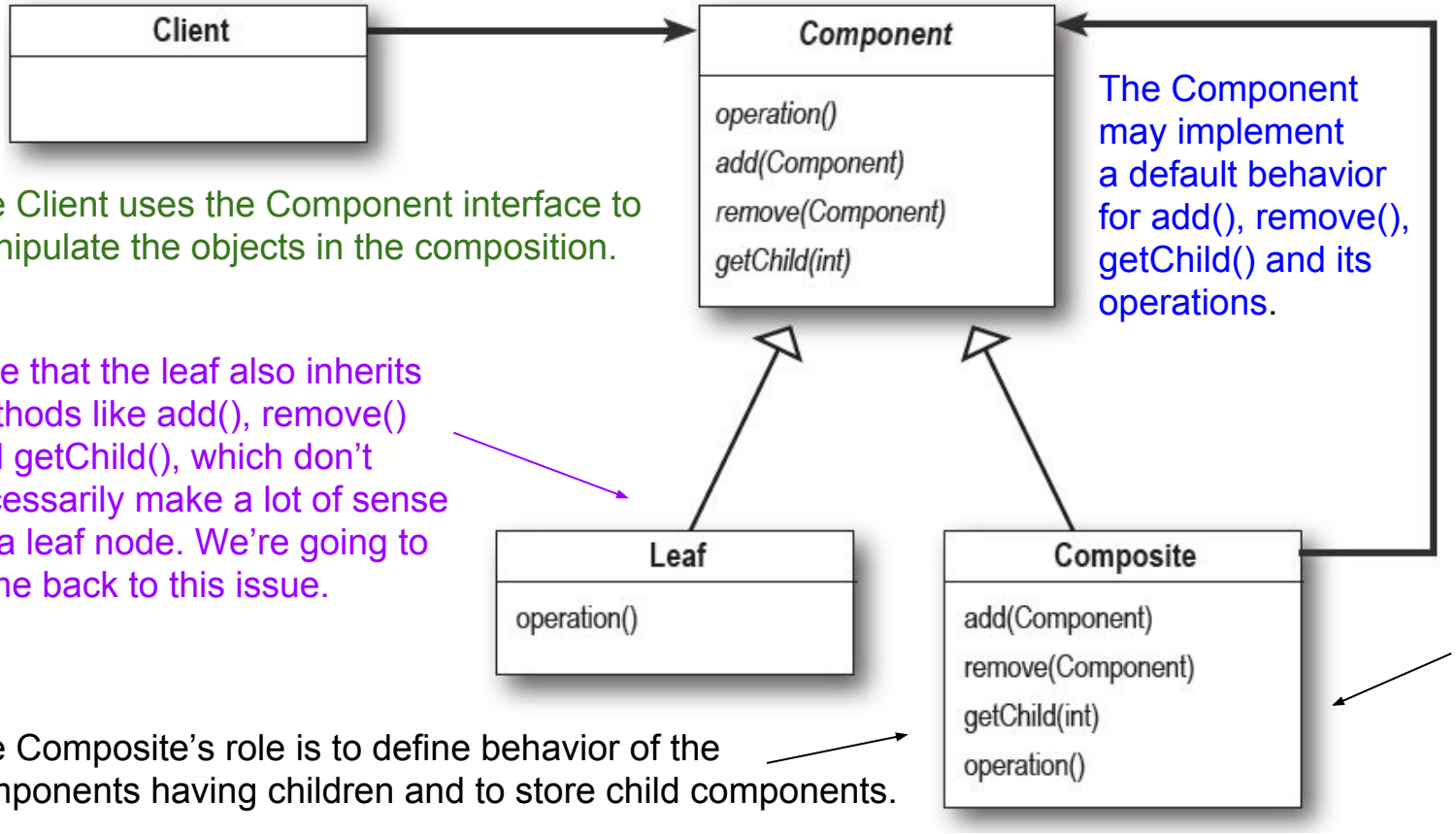
The Composite Pattern defined (cont.)

- 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.

Class diagram



The Component defines an interface for all objects in the composition: both the composite and the leaf nodes.



The Client uses the Component interface to manipulate the objects in the composition.

The Component may implement a default behavior for add(), remove(), getChild() and its operations.

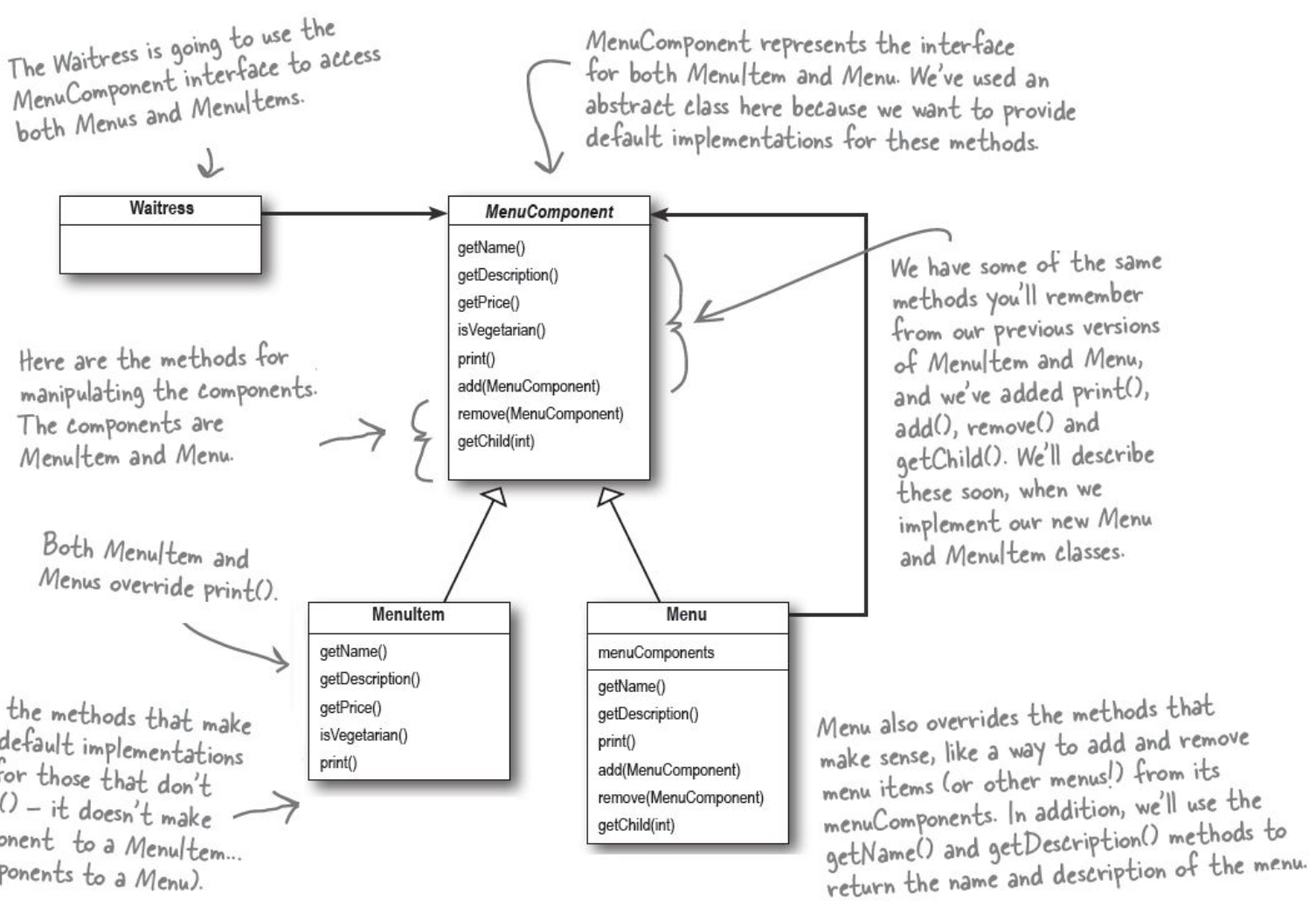
Note that the leaf also inherits methods like add(), remove() and getChild(), which don't necessarily make a lot of sense for a leaf node. We're going to come back to this issue.

The Composite also implements the Leaf related operations. Note that some of these may not make sense on a Composite, so in that case an exception might be generated.

The Composite's role is to define behavior of the components having children and to store child components.

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…


The Waitress is going to use the MenuComponent interface to access both Menus and MenuItems.
MenuComponent represents the interface for both MenuItem and Menu. We've used an abstract class here because we want to provide default implementations for these methods.
Waitress
MenuComponent
getName()
getDescription()
getPrice()
isVegetarian()
print()
add(MenuComponent)
remove(MenuComponent)
getChild(int)
We have some of the same methods you'll remember from our previous versions of MenuItem and Menu, and we've added print(), add(), remove() and getChild(). We'll describe these soon, when we implement our new Menu and MenuItem classes.
Here are the methods for manipulating the components. The components are MenuItem and Menu.
Both MenuItem and Menus override print().
MenuItem
getName()
getDescription()
getPrice()
isVegetarian()
print()
Menu
menuComponents
getName()
getDescription()
print()
add(MenuComponent)
remove(MenuComponent)
getChild(int)
MenuItem overrides the methods that make sense, and uses the default implementations in MenuComponent for those that don't make sense (like add() – it doesn't make sense to add a component to a MenuItem... we can only add components to a Menu).
Menu also overrides the methods that make sense, like a way to add and remove menu items (or other menus!) from its menuComponents. In addition, we'll use the getName() and getDescription() methods to return the name and description of the menu.

Implementing the Menu Component

- 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.
- 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.

```
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

- 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;
    }
```

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.

```
    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());
    }
}
```

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

```
public class Menu extends MenuComponent {
    ArrayList<MenuComponent> menuComponents = new ArrayList<MenuComponent>();
    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) {
        returnmenuComponents.get(i);
    }
```

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

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.

Implementing the Composite Menu (cont.)

```
    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("---------------------");
    }
}
```

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.

Fixing the print() method

- Since menu is a composite and contains both MenuItems 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...

```
public class Menu extends MenuComponent {
    ArrayList<MenuComponent> menuComponents = new ArrayList<MenuComponent>();
    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<MenuComponent> iterator = menuComponents.iterator();
        while (iterator.hasNext()) {
            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...

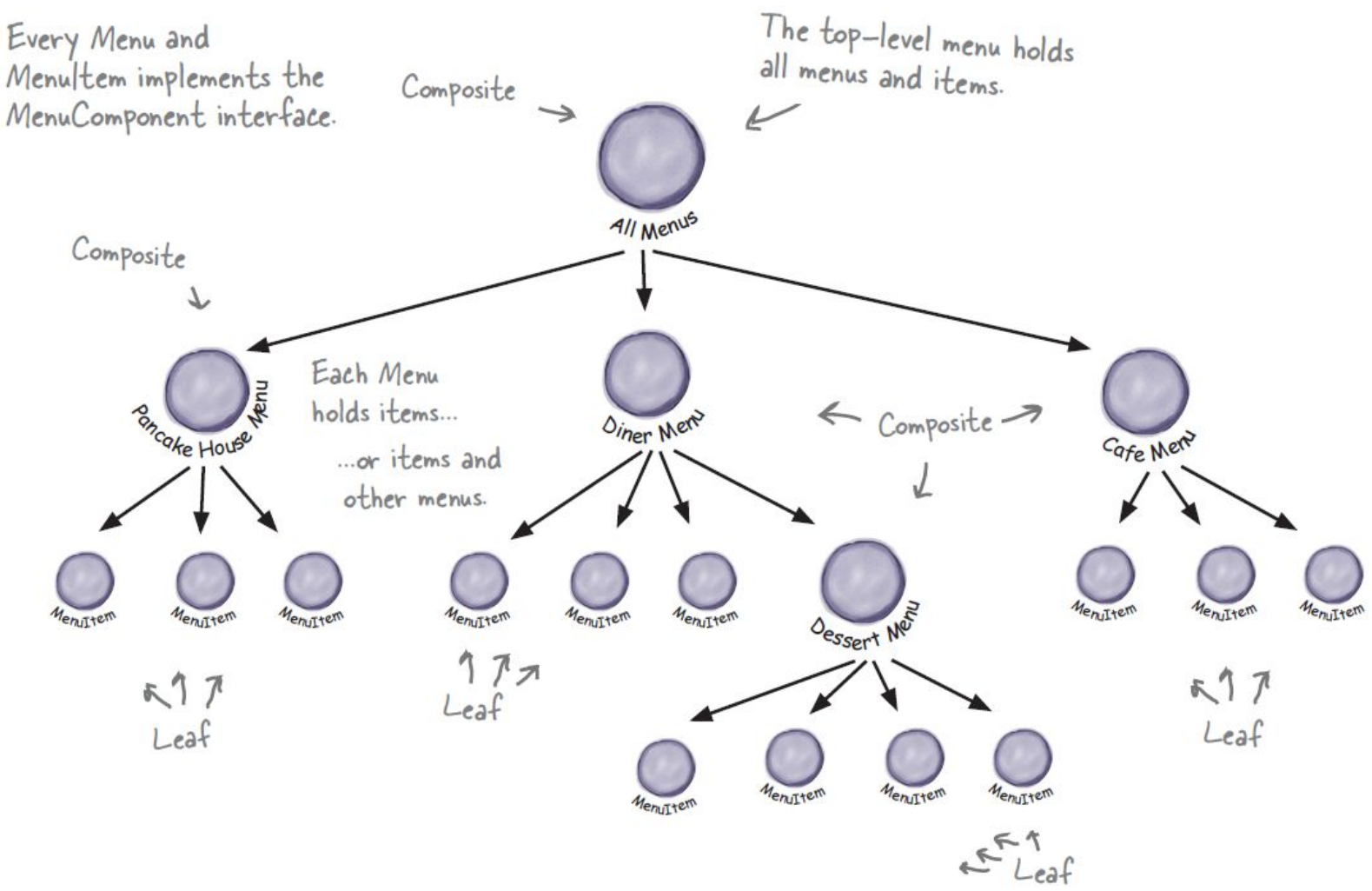
- We need to update the Waitress code.
- After all she's the main client of this code:

```
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.


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

Now for the test drive...

```
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);
```

Let's first create all the menu objects.

We also need a top-level menu that we'll name allMenus.

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

```
    // 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();
}
```

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 as easy as apple pie for her to print it out.

What about the Single Responsibility?

- The Composite Pattern manages a hierarchy AND it performs operations related to Menus. Two responsibilities in a class?
- The Composite Pattern takes the Single Responsibility design principle and trades it for transparency.
  - 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.

What about the Single Responsibility? (cont.)

- 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.

What about the Single Responsibility? (cont.)

- 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.

Some Bullet Points

- 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.

References

Material in this lecture is taken from Freeman, E., Robson, E., Bates, B., & Sierra, K., *Head First Design Patterns: A Brain-Friendly Guide*, O'Reilly Media, Inc., 2014.