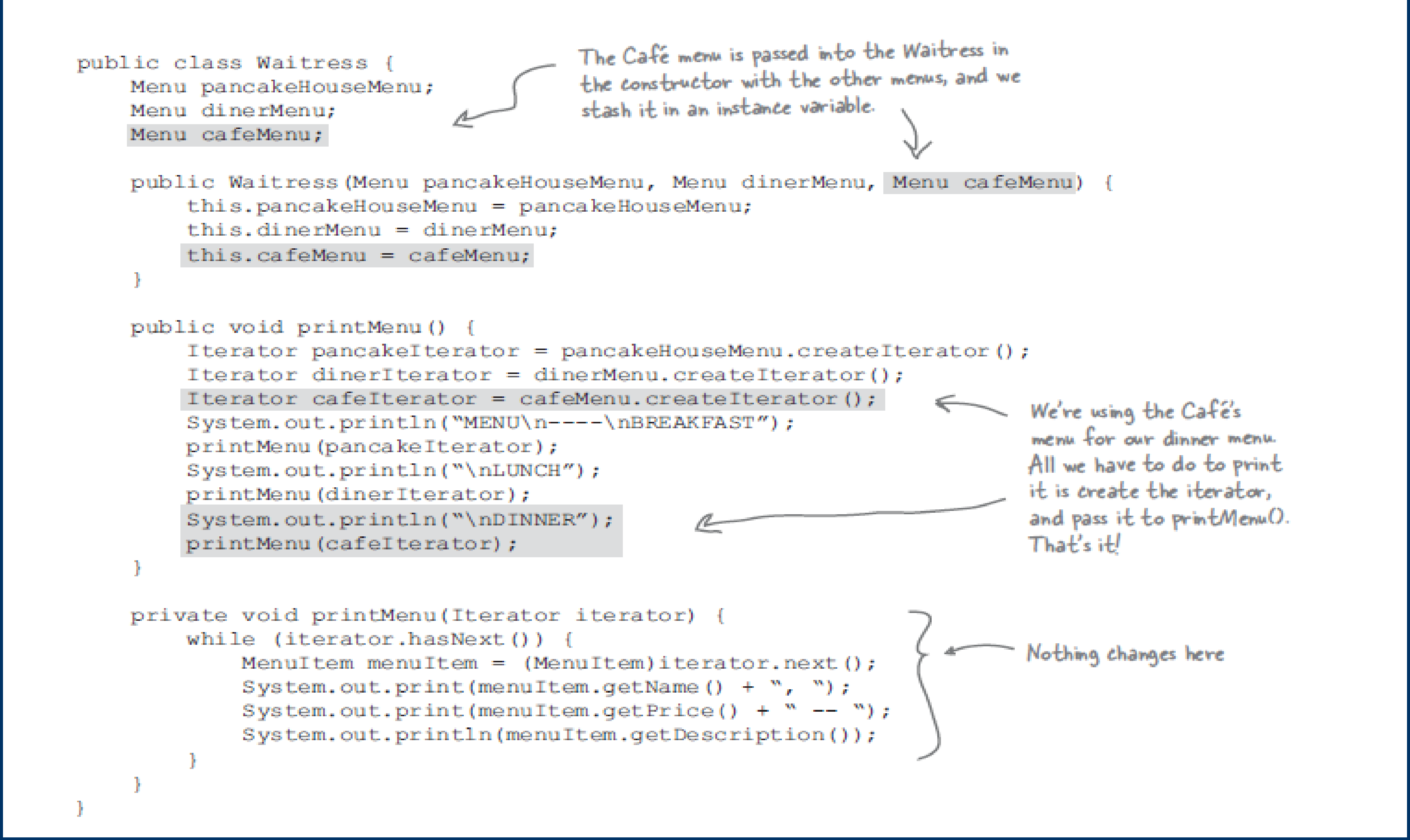
Iterator and Composite Pattern

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

Iterator and Composite Pattern



```
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

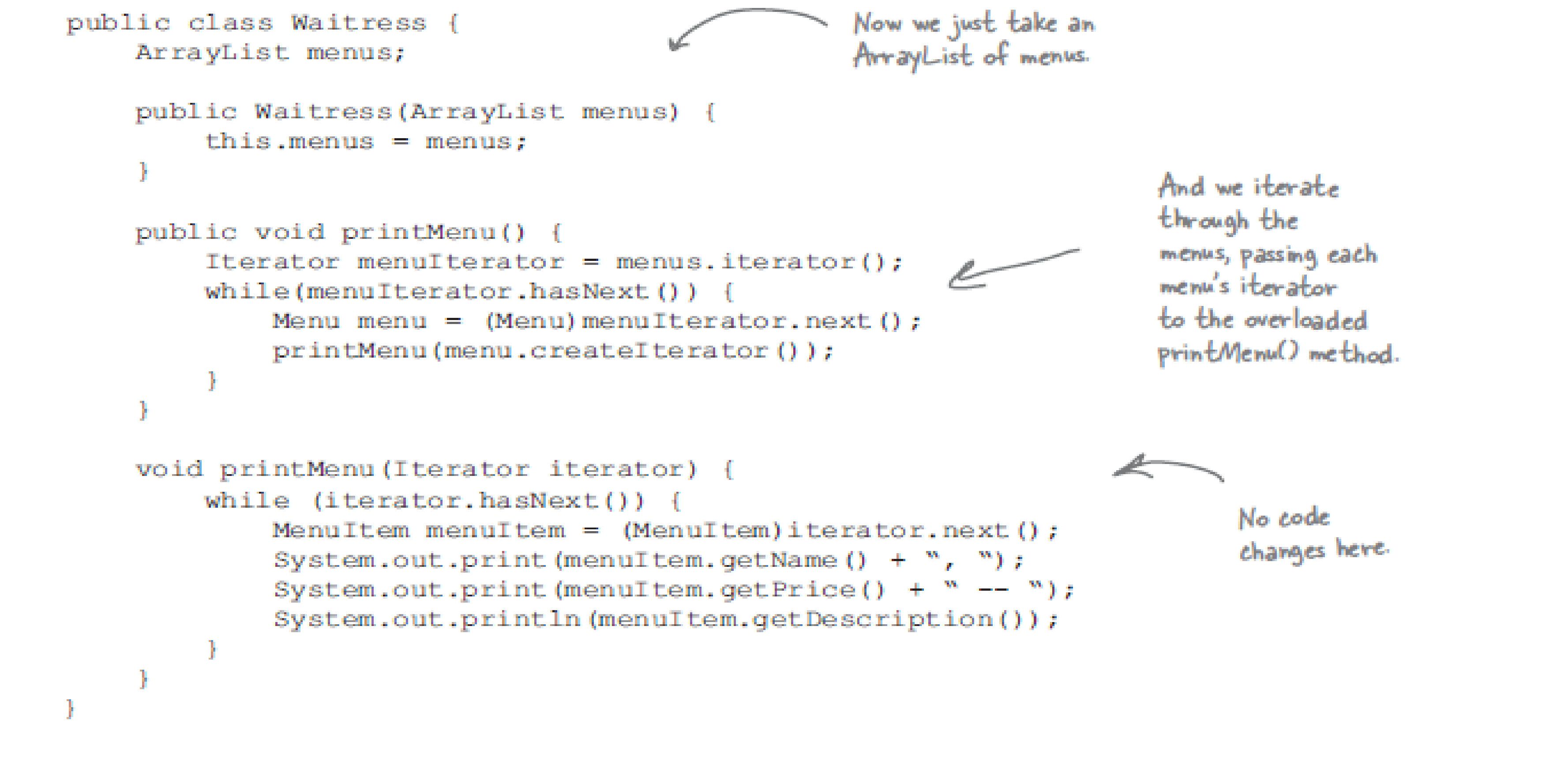
Iterator and composite Pattern

- What else can we change ?

  How do you feel with having to change printMenu every time a new menu is added ?

- What can we do to improve that ?

Iterator and Composite



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

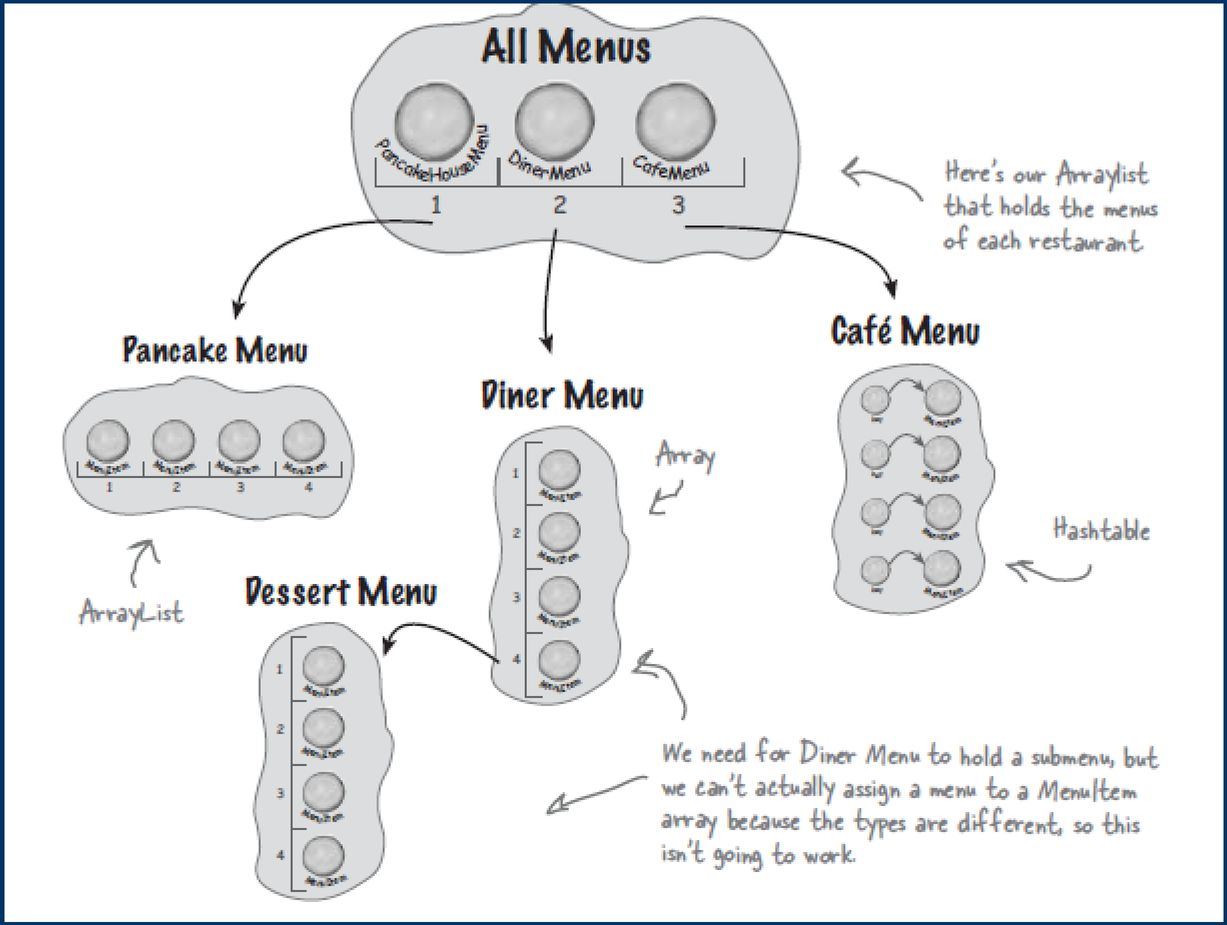
Now we just take an ArrayList of menus.

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

No code changes here.

Iterator and Composite

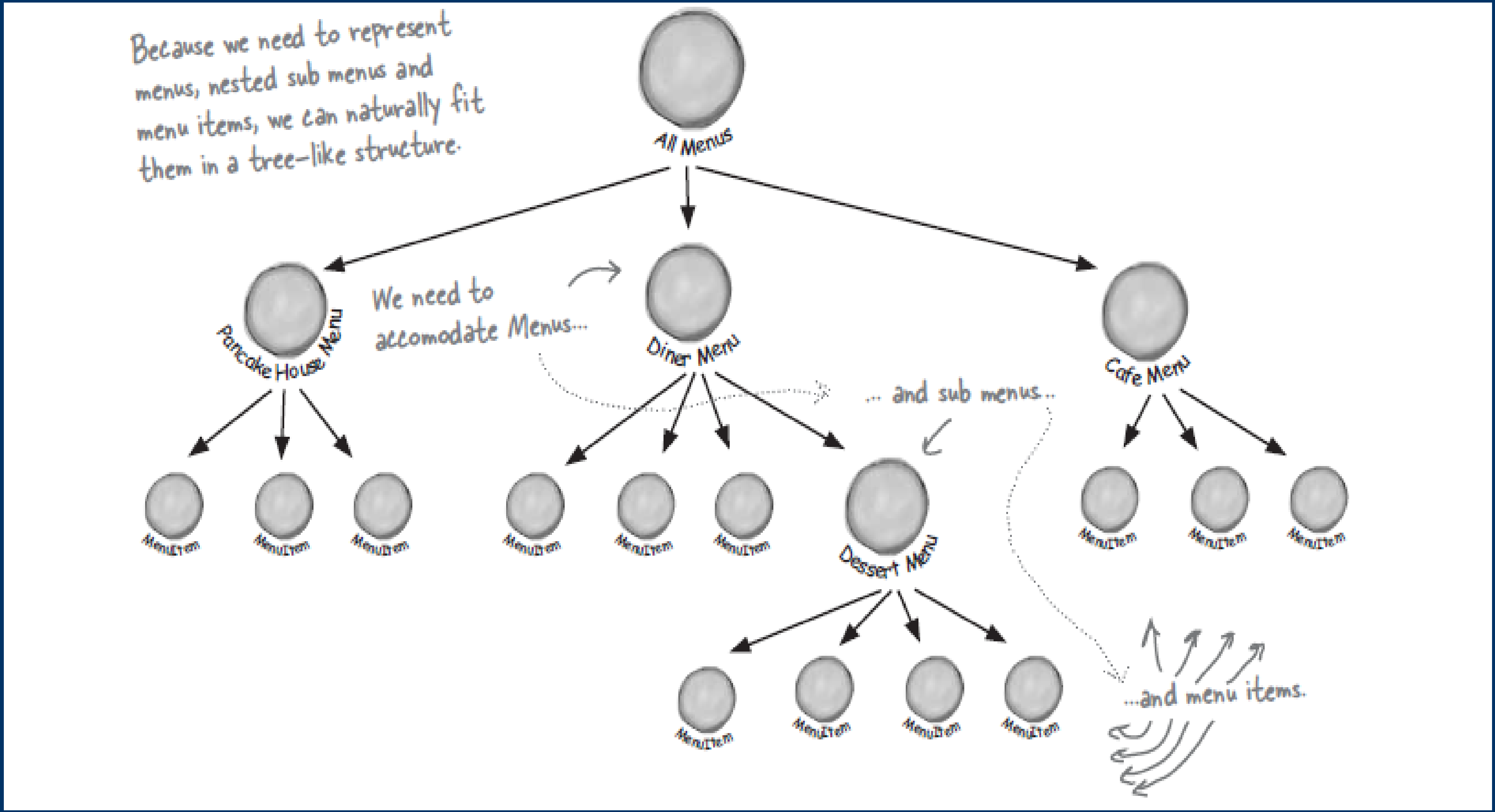
- Our new problem is that we need to add a dessert sub menu.



Here's our Arraylist that holds the menus of each restaurant.

We need for Diner Menu to hold a submenu, but we can't actually assign a menu to a MenuItem array because the types are different, so this isn't going to work.

Iterator and Composite



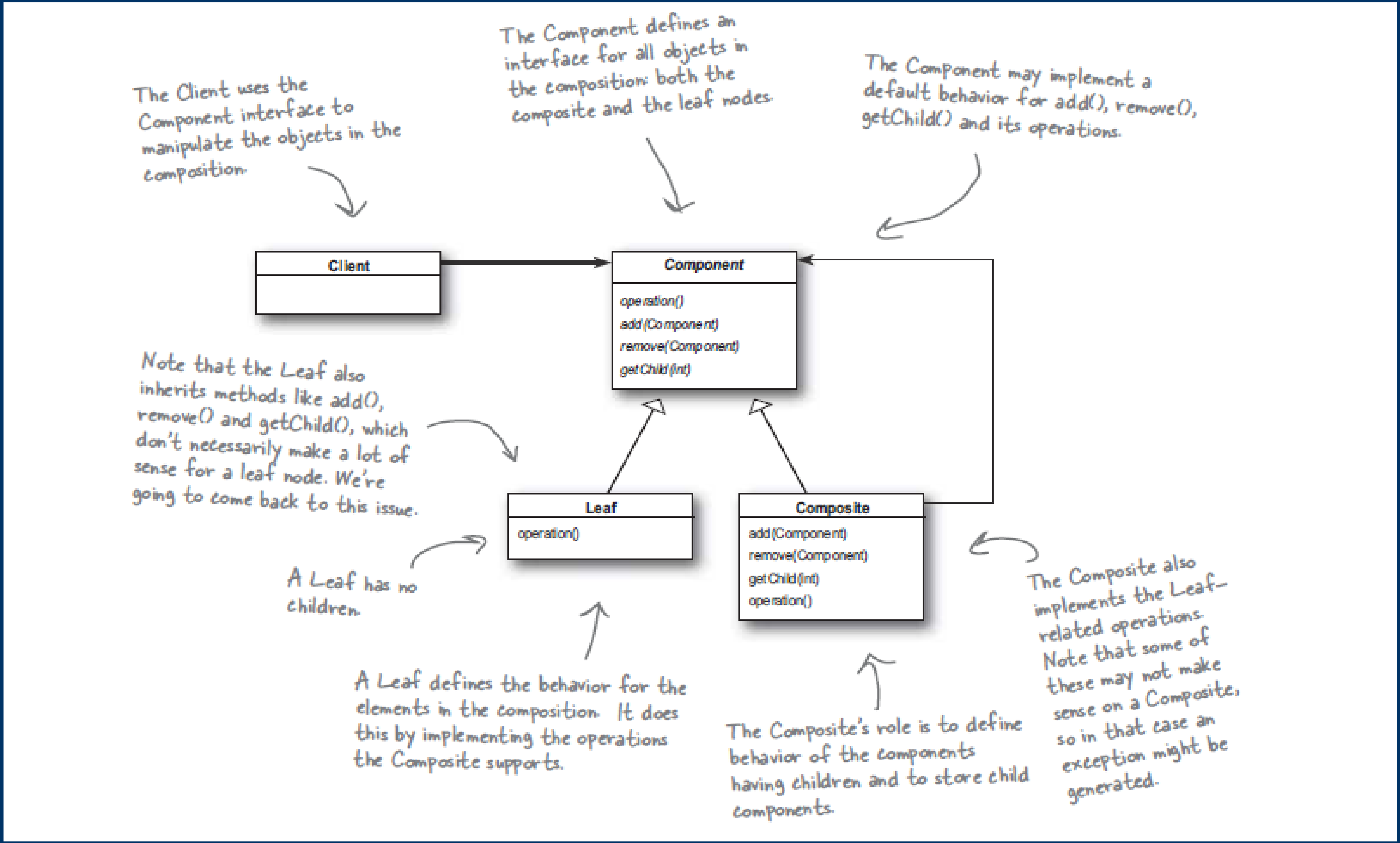
Iterator and Composite

- How would you do that ?

Iterator and Composite

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

Iterator and Composite



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

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

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.

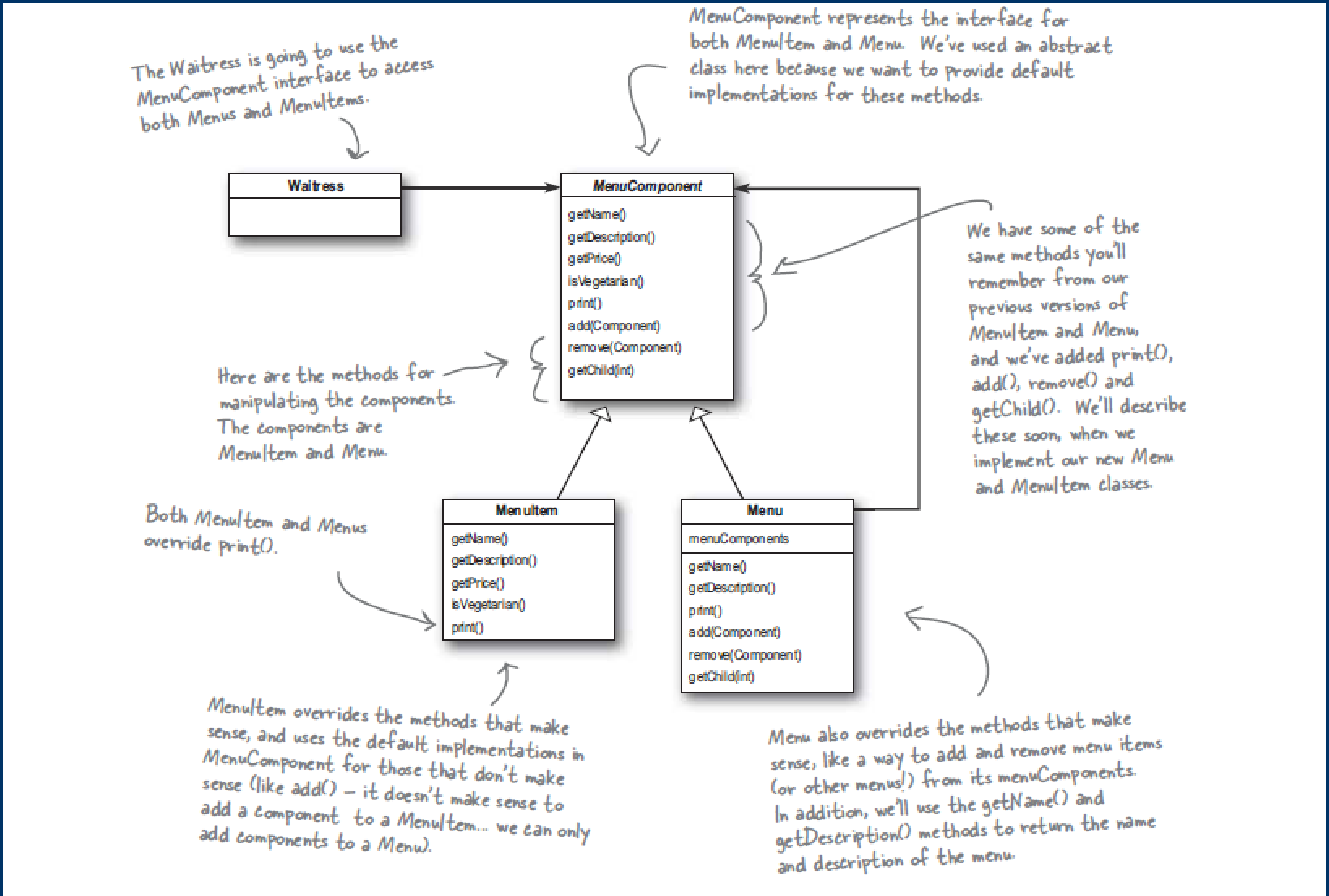
A Leaf has no children.

A Leaf defines the behavior for the elements in the composition. It does this by implementing the operations the Composite supports.

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

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.

Iterator and Composite(code example)



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.

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.

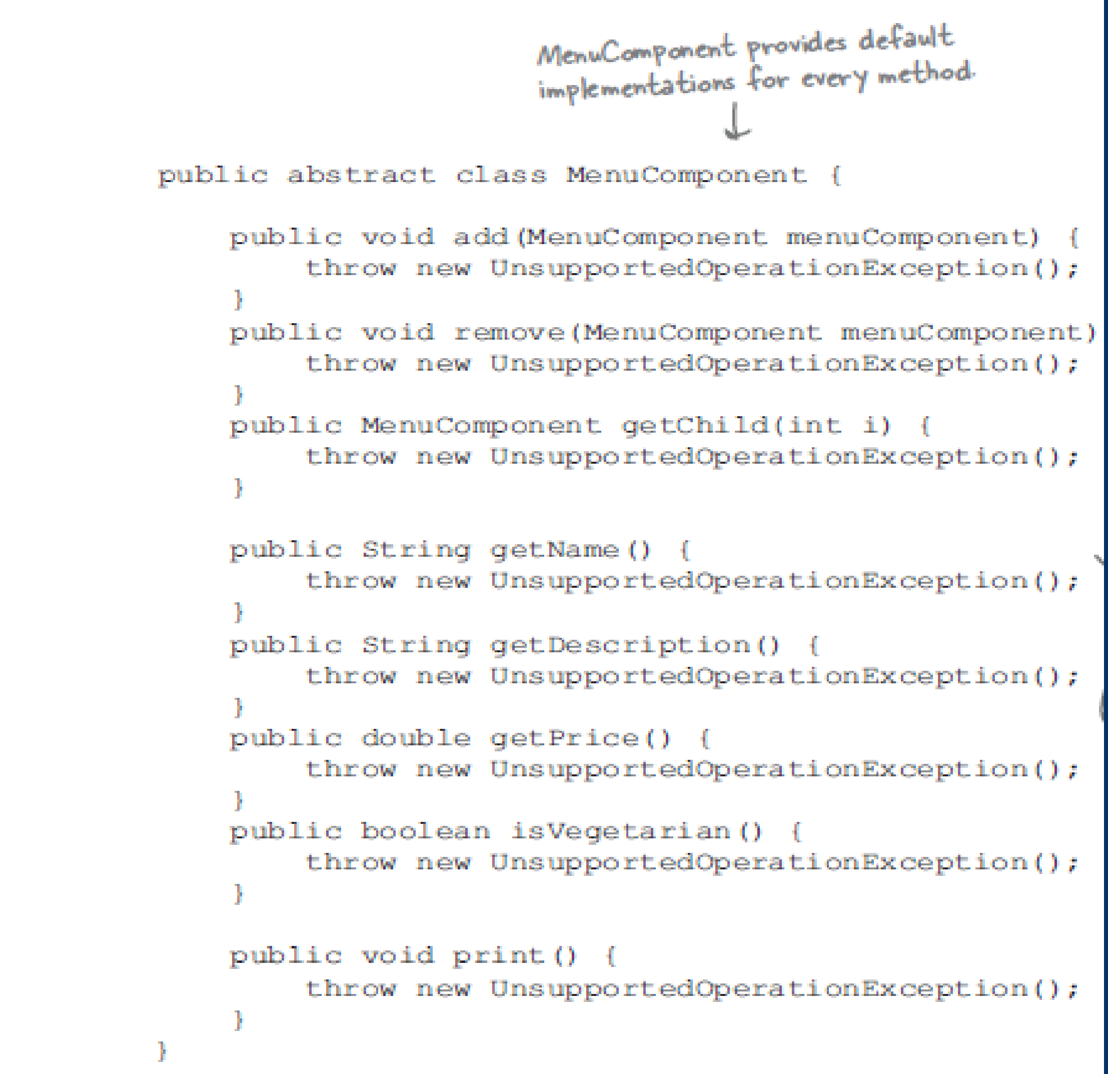
Here are the methods for manipulating the components. The components are MenuItem and Menu.

Both MenuItem and Menus override print().

MenuItem 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.

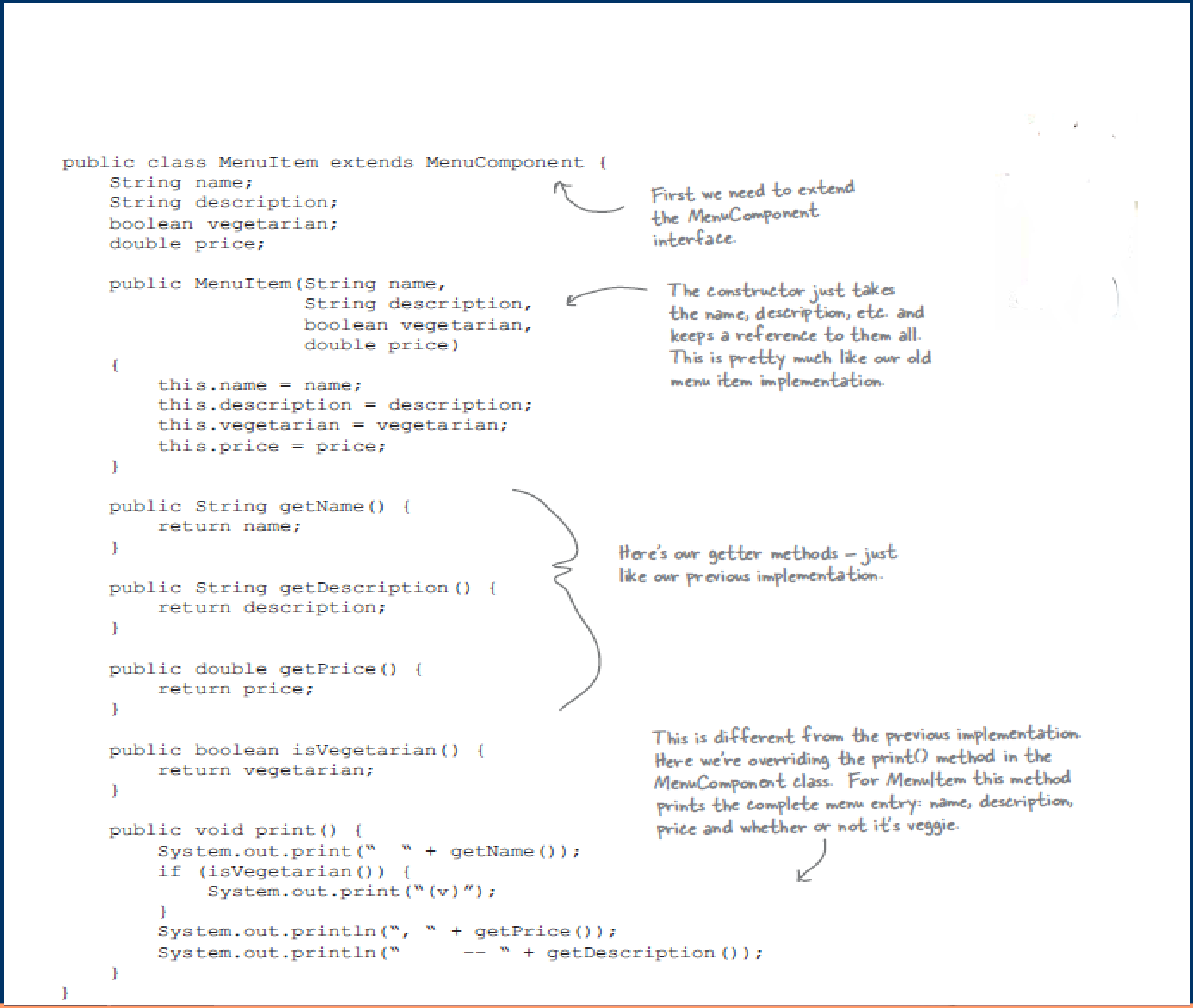
Iterator and Composite(code example)



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();
     }
}
```

Iterator and Composite(code example)



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

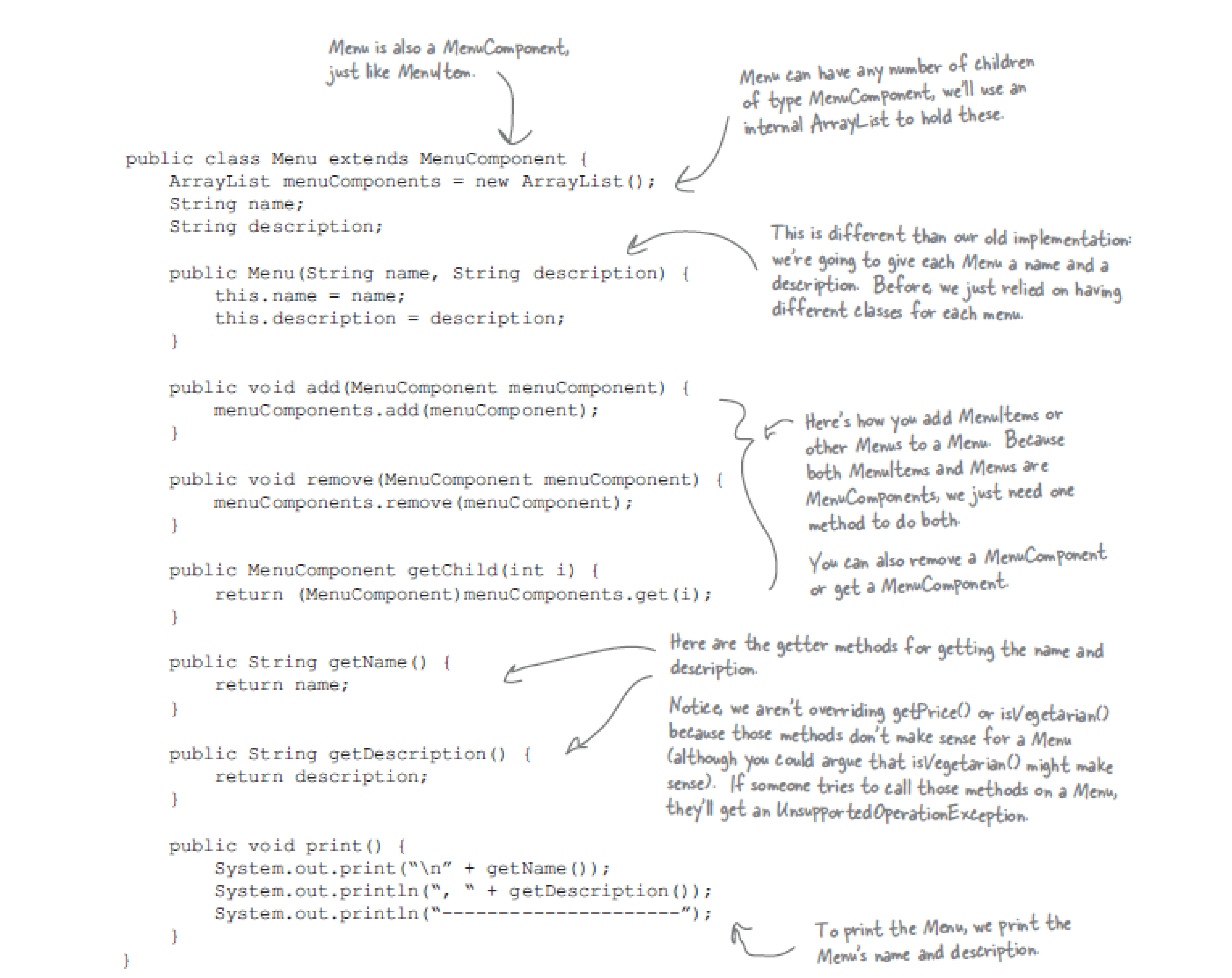
First we need to extend the MenuComponent interface.

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.

Iterator and Composite(code example)



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.

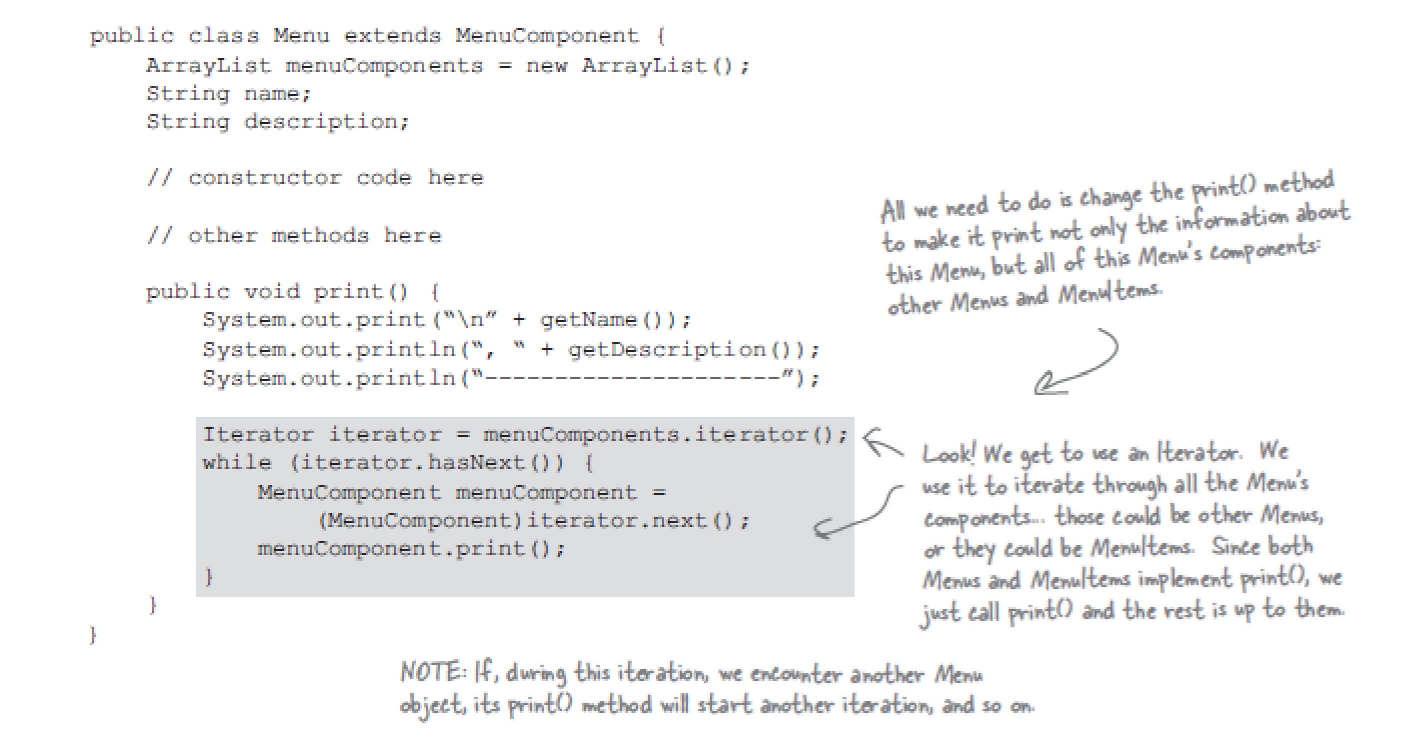
You can also remove a MenuComponent or get a MenuComponent.

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.

Iterator and Composite(code example)



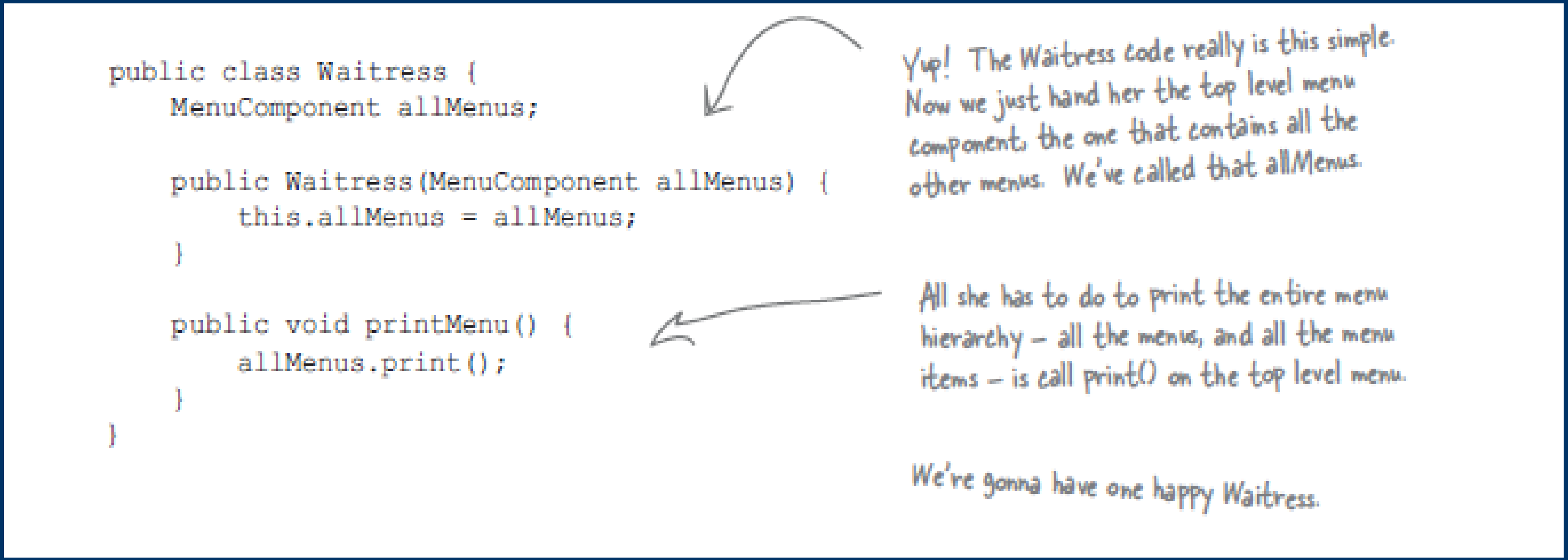
```
public class Menu extends MenuComponent {
    ArrayList menuComponents = new ArrayList();
    String name;
    String description;

    // 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.

Iterator and Composite



```
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.

Iterator(applicability)

- To access an aggregate object 's content without exposing its internal representation.
- To support multiple traversal of aggregate objects.
- To provide an uniform interface for traversing different agregate structures.

Composite(applicability)

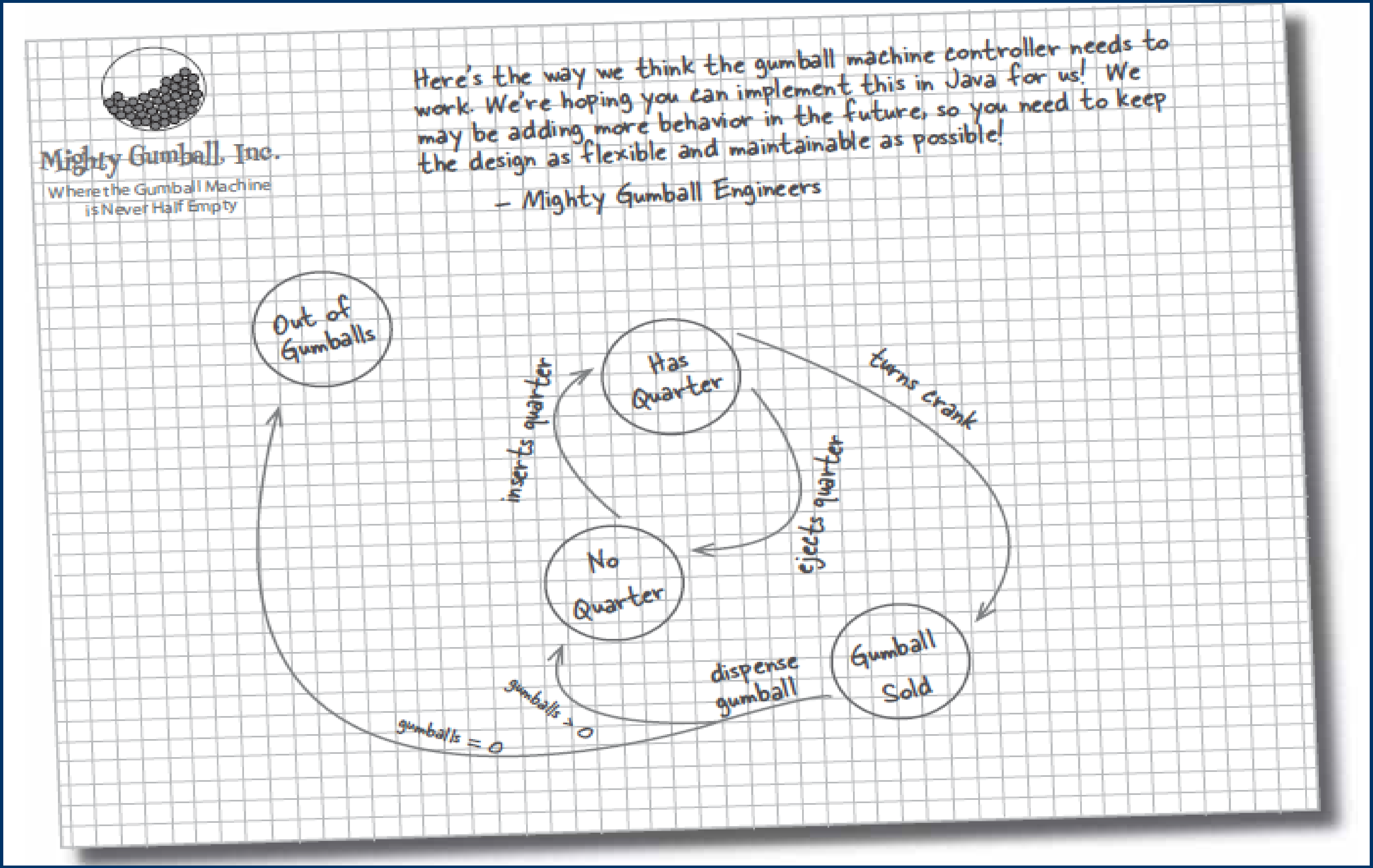
- You want to represent part-whole hierarchies of objects.
- You want clients to ignore the difference between composition of objects and individual objects.

State Pattern

- Allow an object to alter its internal behavior when its internal state changes. The object will appear to change its class.

State Pattern(exersice)

- Gumball machine. How do you think it works ?

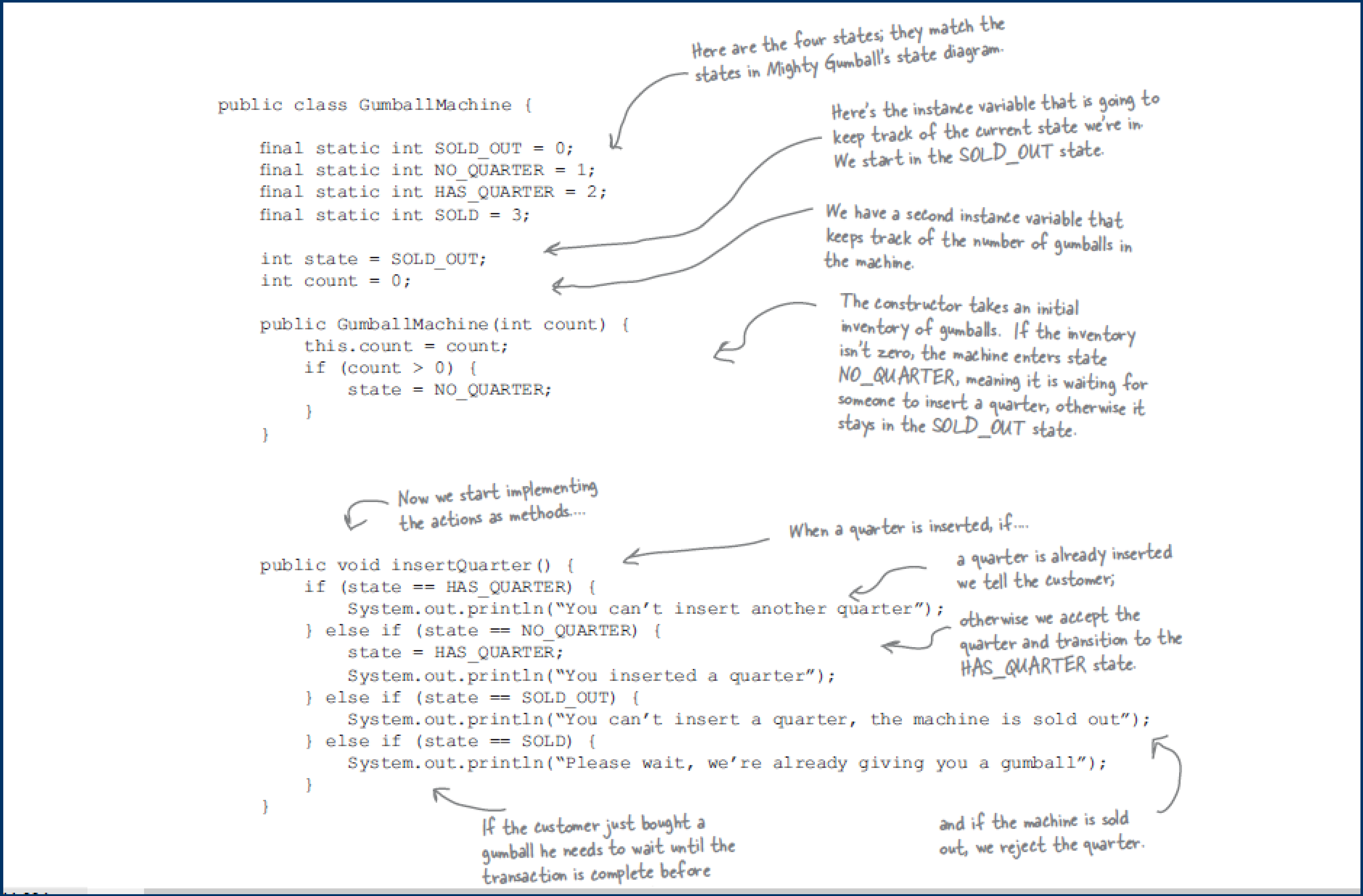


Mighty Gumball, Inc.
Where the Gumball Machine is Never Half Empty

Here's the way we think the gumball machine controller needs to work. We're hoping you can implement this in Java for us! We may be adding more behavior in the future, so you need to keep the design as flexible and maintainable as possible!

– Mighty Gumball Engineers

State Pattern(exersice)



Here are the four states; they match the states in Mighty Gumball's state diagram.

Here's the instance variable that is going to keep track of the current state we're in. We start in the SOLD_OUT state.

We have a second instance variable that keeps track of the number of gumballs in the machine.

The constructor takes an initial inventory of gumballs. If the inventory isn't zero, the machine enters state NO_QUARTER, meaning it is waiting for someone to insert a quarter, otherwise it stays in the SOLD_OUT state.

```
public class GumballMachine {

    final static int SOLD_OUT = 0;
    final static int NO_QUARTER = 1;
    final static int HAS_QUARTER = 2;
    final static int SOLD = 3;

    int state = SOLD_OUT;
    int count = 0;

    public GumballMachine(int count) {
        this.count = count;
        if (count > 0) {
            state = NO_QUARTER;
        }
    }
```

Now we start implementing the actions as methods....

When a quarter is inserted, if....

a quarter is already inserted we tell the customer;

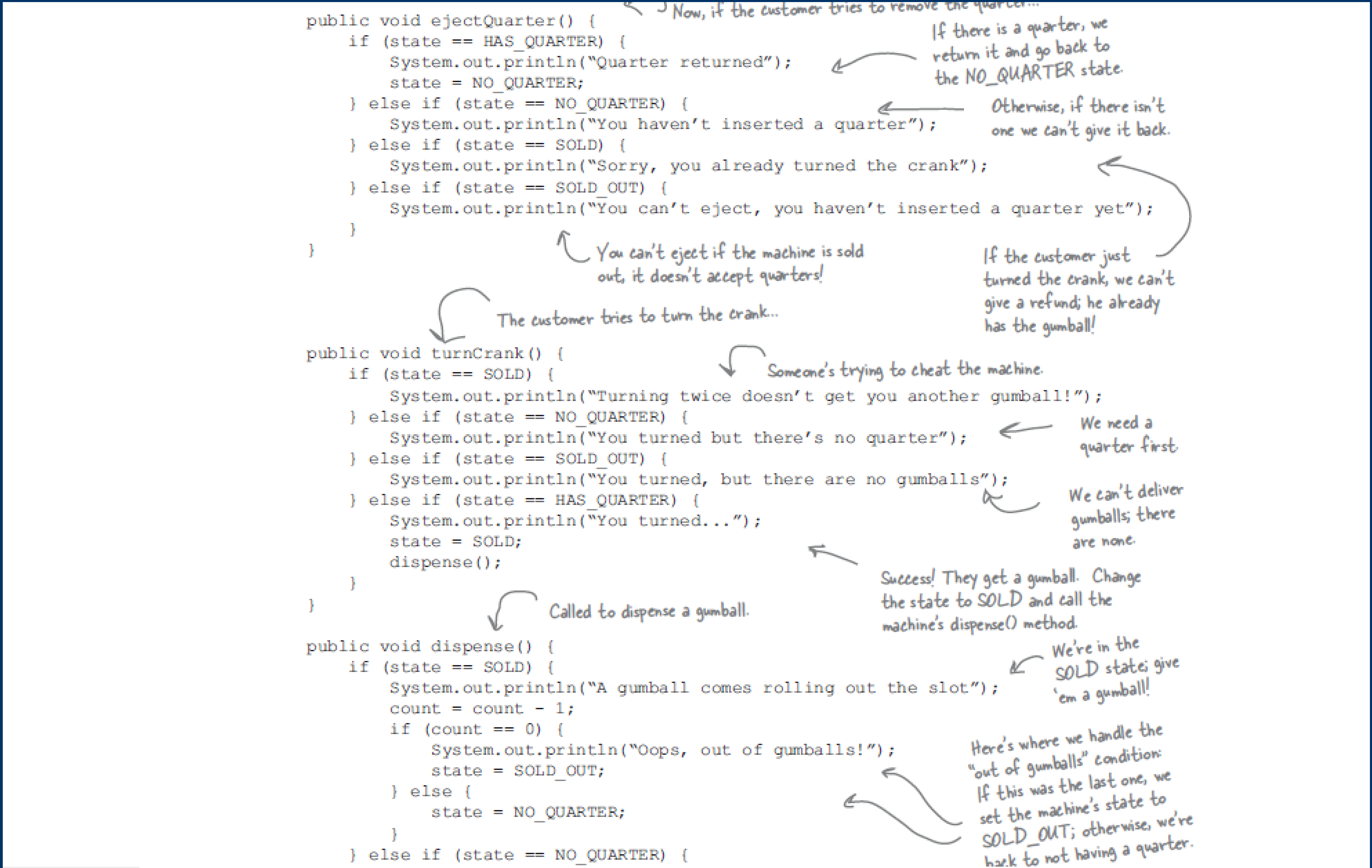
otherwise we accept the quarter and transition to the HAS_QUARTER state.

```
    public void insertQuarter() {
        if (state == HAS_QUARTER) {
            System.out.println("You can't insert another quarter");
        } else if (state == NO_QUARTER) {
            state = HAS_QUARTER;
            System.out.println("You inserted a quarter");
        } else if (state == SOLD_OUT) {
            System.out.println("You can't insert a quarter, the machine is sold out");
        } else if (state == SOLD) {
            System.out.println("Please wait, we're already giving you a gumball");
        }
    }
```

If the customer just bought a gumball he needs to wait until the transaction is complete before

and if the machine is sold out, we reject the quarter.

State Pattern(exersice)



Now, if the customer tries to remove the quarter...

```
public void ejectQuarter() {
    if (state == HAS_QUARTER) {
        System.out.println("Quarter returned");
        state = NO_QUARTER;
    } else if (state == NO_QUARTER) {
        System.out.println("You haven't inserted a quarter");
    } else if (state == SOLD) {
        System.out.println("Sorry, you already turned the crank");
    } else if (state == SOLD_OUT) {
        System.out.println("You can't eject, you haven't inserted a quarter yet");
    }
}
```

If there is a quarter, we return it and go back to the NO_QUARTER state.

Otherwise, if there isn't one we can't give it back.

If the customer just turned the crank, we can't give a refund; he already has the gumball!

You can't eject if the machine is sold out, it doesn't accept quarters!

The customer tries to turn the crank...

```
public void turnCrank() {
    if (state == SOLD) {
        System.out.println("Turning twice doesn't get you another gumball!");
    } else if (state == NO_QUARTER) {
        System.out.println("You turned but there's no quarter");
    } else if (state == SOLD_OUT) {
        System.out.println("You turned, but there are no gumballs");
    } else if (state == HAS_QUARTER) {
        System.out.println("You turned...");
        state = SOLD;
        dispense();
    }
}
```

Someone's trying to cheat the machine.

We need a quarter first.

We can't deliver gumballs; there are none.

Success! They get a gumball. Change the state to SOLD and call the machine's dispense() method.

Called to dispense a gumball.

```
public void dispense() {
    if (state == SOLD) {
        System.out.println("A gumball comes rolling out the slot");
        count = count - 1;
        if (count == 0) {
            System.out.println("Oops, out of gumballs!");
            state = SOLD_OUT;
        } else {
            state = NO_QUARTER;
        }
    } else if (state == NO_QUARTER) {
```

We're in the SOLD state; give 'em a gumball!

Here's where we handle the "out of gumballs" condition: If this was the last one, we set the machine's state to SOLD_OUT; otherwise, we're back to not having a quarter.

State Pattern(exersice)

- 10% of the time, when the crunck is turned the customer get two gumballs instead of one.