

Template Method, Iterator & Composite

Kenneth M. Anderson
University of Colorado, Boulder
CSCI 4448/5448 — Lecture 23 — 11/10/2008

© University of Colorado, 2009

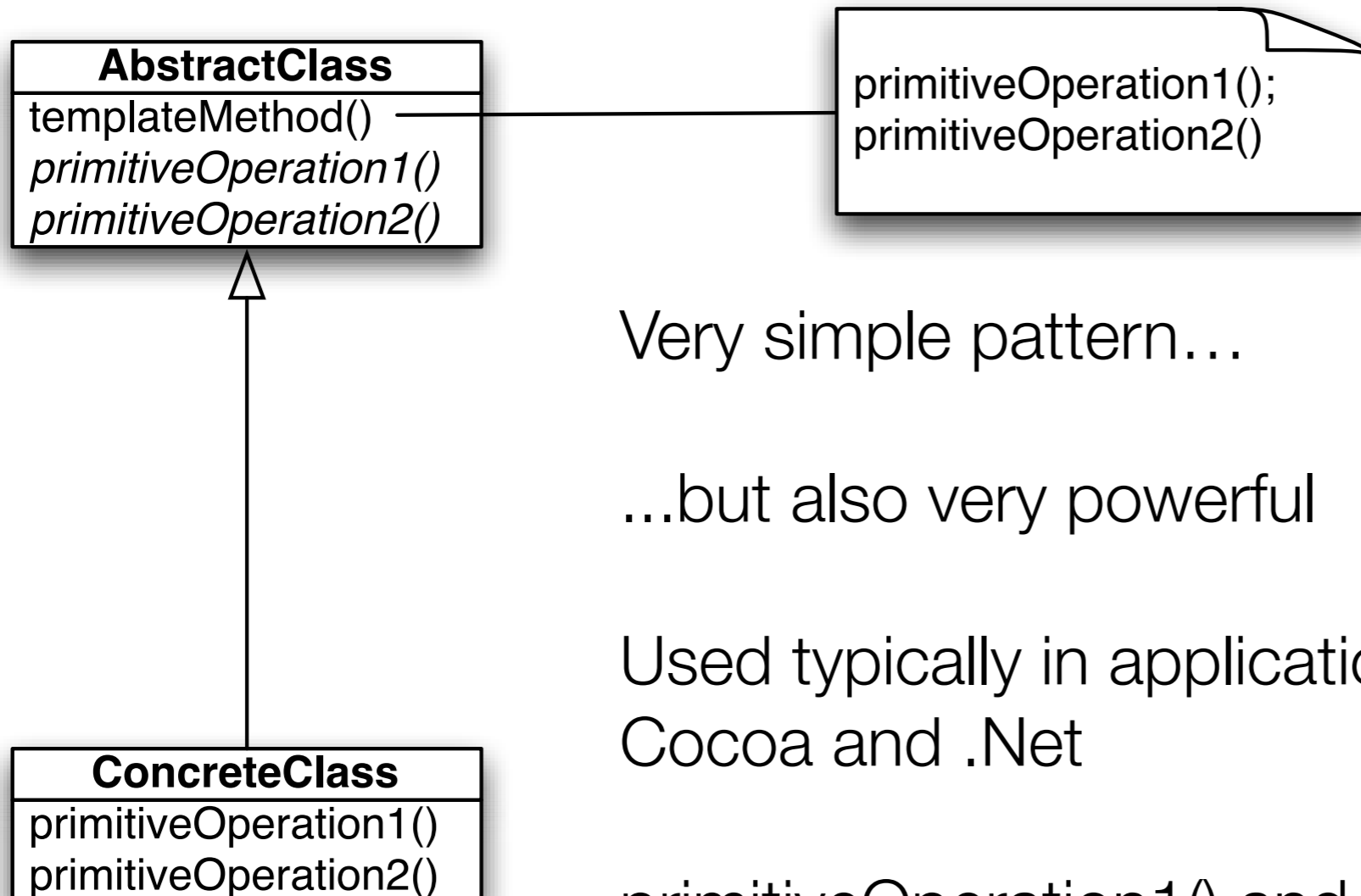
Lecture Goals

- Cover Material from Chapter 8 and 9 of the Design Patterns Textbook
 - Template Method Pattern
 - Iterator Pattern
 - Composite Pattern

Template Method: Definition

- The Template Method Pattern defines the skeleton of an algorithm in a method, deferring some steps to subclasses. Template Method lets subclasses **redefine** certain steps of an algorithm without changing the algorithm's **structure**
 - Template Method defines the steps of an algorithm and allows subclasses to provide the implementation for one or more steps
 - Makes the algorithm abstract
 - Each step of the algorithm is represented by a method
 - Encapsulates the details of most steps
 - Steps (methods) handled by subclasses are declared abstract
 - Shared steps (concrete methods) are placed in the same class that has the template method, allowing for code re-use among the various subclasses

Template Method: Structure



Very simple pattern...

...but also very powerful

Used typically in application frameworks, e.g. Cocoa and .Net

`primitiveOperation1()` and `primitiveOperation2()` are sometimes referred to as **hook methods** as they allow subclasses *to hook* their behavior *into* the service provided by `AbstractClass`

Example: Tea and Coffee

- The book returns to the Starbuzz example and shows the training guide for baristas and, in particular, the recipes for making coffee and tea
 - Coffee
 - Boil water
 - Brew coffee in boiling water
 - Pour coffee in cup
 - Add sugar and milk
 - Tea
 - Boil water
 - Steep tea in boiling water
 - Pour tea in cup
 - Add lemon

Coffee Implementation

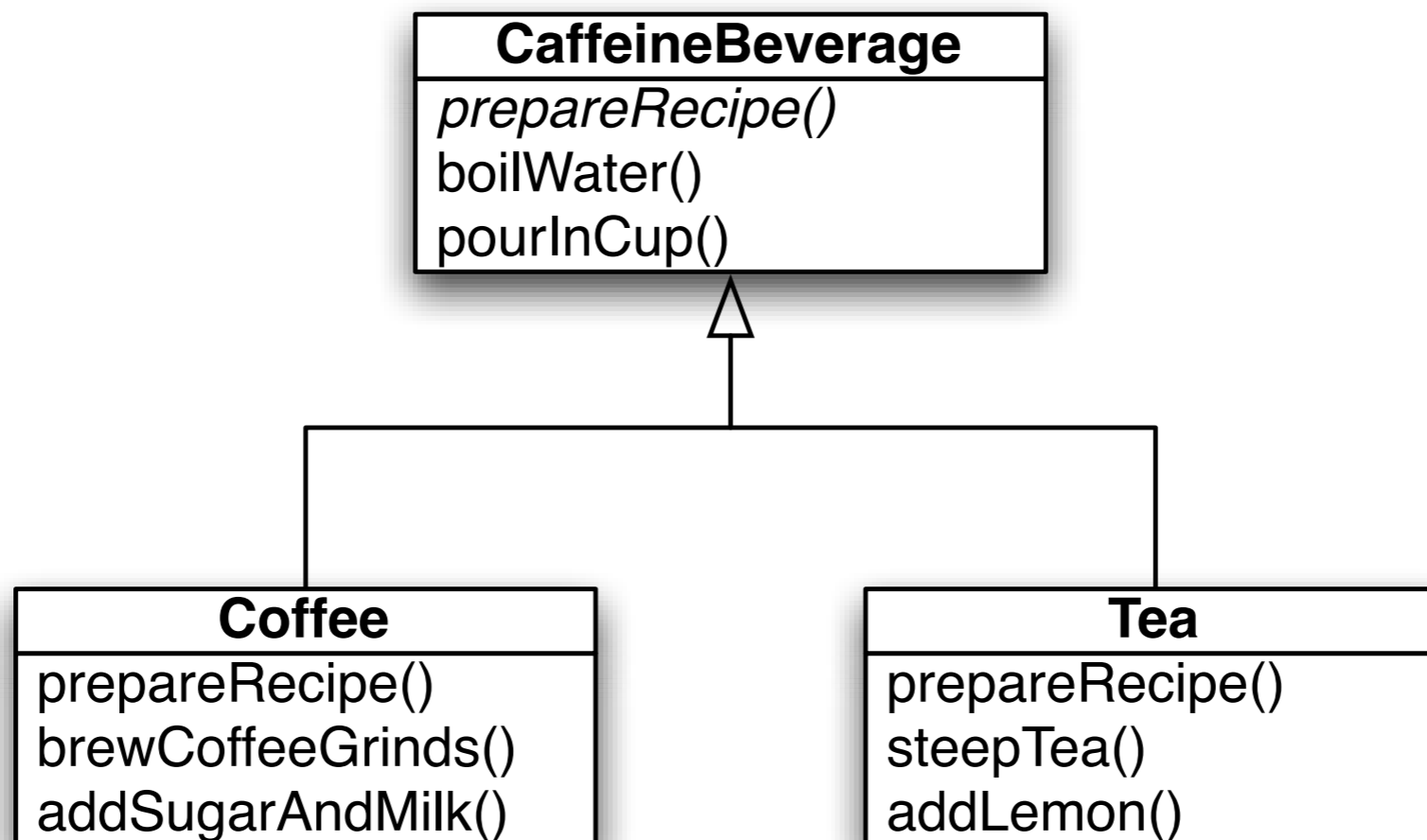
```
1 public class Coffee {
2
3     void prepareRecipe() {
4         boilWater();
5         brewCoffeeGrinds();
6         pourInCup();
7         addSugarAndMilk();
8     }
9
10    public void boilWater() {
11        System.out.println("Boiling water");
12    }
13
14    public void brewCoffeeGrinds() {
15        System.out.println("Dripping Coffee through filter");
16    }
17
18    public void pourInCup() {
19        System.out.println("Pouring into cup");
20    }
21
22    public void addSugarAndMilk() {
23        System.out.println("Adding Sugar and Milk");
24    }
25 }
26
```

Tea Implementation

```
1 public class Tea {
2
3     void prepareRecipe() {
4         boilWater();
5         steepTeaBag();
6         pourInCup();
7         addLemon();
8     }
9
10    public void boilWater() {
11        System.out.println("Boiling water");
12    }
13
14    public void steepTeaBag() {
15        System.out.println("Steeping the tea");
16    }
17
18    public void addLemon() {
19        System.out.println("Adding Lemon");
20    }
21
22    public void pourInCup() {
23        System.out.println("Pouring into cup");
24    }
25 }
26
```

Code Duplication!

- We have code duplication occurring in these two classes
 - `boilWater()` and `pourInCup()` are exactly the same
- Lets get rid of the duplication



Similar algorithms

- The structure of the algorithms in `prepareRecipe()` is similar for Tea and Coffee
 - We can improve our code further by making the code in `prepareRecipe()` more abstract
 - `brewCoffeeGrinds()` and `steepTea()` \Rightarrow `brew()`
 - `addSugarAndMilk()` and `addLemon()` \Rightarrow `addCondiments()`
- Excellent, now all we need to do is specify this structure in `CaffeineBeverage.prepareRecipe()` and make it such that subclasses can't change the structure
 - How do we do that?
 - Answer: By convention OR by using the keyword "final" in languages that support it

CaffeineBeverage Implementation

```
1 public abstract class CaffeineBeverage {
2
3     final void prepareRecipe() {
4         boilWater();
5         brew();
6         pourInCup();
7         addCondiments();
8     }
9
10    abstract void brew();
11
12    abstract void addCondiments();
13
14    void boilWater() {
15        System.out.println("Boiling water");
16    }
17
18    void pourInCup() {
19        System.out.println("Pouring into cup");
20    }
21 }
22
```

Note: use of final keyword for prepareReceipe()

brew() and addCondiments() are abstract and must be supplied by subclasses

boilWater() and pourInCup() are specified and shared across all subclasses

Coffee And Tea Implementations

```
1 public class Coffee extends CaffeineBeverage {
2     public void brew() {
3         System.out.println("Dripping Coffee through filter");
4     }
5     public void addCondiments() {
6         System.out.println("Adding Sugar and Milk");
7     }
8 }
9
10 public class Tea extends CaffeineBeverage {
11     public void brew() {
12         System.out.println("Steeping the tea");
13     }
14     public void addCondiments() {
15         System.out.println("Adding Lemon");
16     }
17 }
18
```

Nice and Simple!

What have we done?

- Took two separate classes with separate but similar algorithms
- Noticed duplication and eliminated it by introducing a superclass
- Made steps of algorithm more abstract and specified its structure in the superclass
 - Thereby eliminating another “implicit” duplication between the two classes
- Revised subclasses to implement the abstract (unspecified) portions of the algorithm... in a way that made sense for them

Comparison: Template Method (TM) vs. No TM

- **No Template Method**

- Coffee and Tea each have own copy of algorithm
- Code is duplicated across both classes
- A change in the algorithm would result in a change in both classes
- Not easy to add new caffeine beverage
- Knowledge of algorithm distributed over multiple classes

- **Template Method**

- CaffeineBeverage has the algorithm and protects it
- CaffeineBeverage shares common code with all subclasses
- A change in the algorithm likely impacts only CaffeineBeverage
- New caffeine beverages can easily be plugged in
- CaffeineBeverage centralizes knowledge of the algorithm; subclasses plug in missing pieces

The Book's Hook

- Previously I called the abstract methods that appear in a template method “hook” methods
 - The book refers to hook methods as well, but they make the following distinction: a hook method is a concrete method that appears in the AbstractClass that has an empty method body (or a mostly empty method body, see example next slide), i.e.
 - `public void hook() {}`
 - Subclasses are free to override them but don't have to since they provide a method body, albeit an empty one
 - In contrast, a subclass is forced to implement abstract methods that appear in AbstractClass
- Hook methods, thus, should represent optional parts of the algorithm

Adding a Hook to CaffeineBeverage

```
1 public abstract class CaffeineBeverageWithHook {
2
3     void prepareRecipe() {
4         boilWater();
5         brew();
6         pourInCup();
7         if (customerWantsCondiments()) {
8             addCondiments();
9         }
10    }
11
12    abstract void brew();
13
14    abstract void addCondiments();
15
16    void boilWater() {
17        System.out.println("Boiling water");
18    }
19
20    void pourInCup() {
21        System.out.println("Pouring into cup");
22    }
23
24    boolean customerWantsCondiments() {
25        return true;
26    }
27 }
28
```

**prepareRecipe() altered to have a hook method:
customerWantsCondiments()**

This method provides a method body that subclasses can override

To make the distinction between hook and non-hook methods more clear, you can add the “final” keyword to all concrete methods that you don’t want subclasses to touch

```

1 import java.io.*;
2
3 public class CoffeeWithHook extends CaffeineBeverageWithHook {
4
5     public void brew() {
6         System.out.println("Dripping Coffee through filter");
7     }
8
9     public void addCondiments() {
10        System.out.println("Adding Sugar and Milk");
11    }
12
13    public boolean customerWantsCondiments() {
14
15        String answer = getUserInput();
16
17        if (answer.toLowerCase().startsWith("y")) {
18            return true;
19        } else {
20            return false;
21        }
22    }
23
24    private String getUserInput() {
25        String answer = null;
26
27        System.out.print("Would you like milk and sugar with your coffee (y/n)? ");
28
29        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
30        try {
31            answer = in.readLine();
32        } catch (IOException ioe) {
33            System.err.println("IO error trying to read your answer");
34        }
35        if (answer == null) {
36            return "no";
37        }
38        return answer;
39    }
40 }

```

Adding a Hook to Coffee

Demonstration

New Design Principle: Hollywood Principle

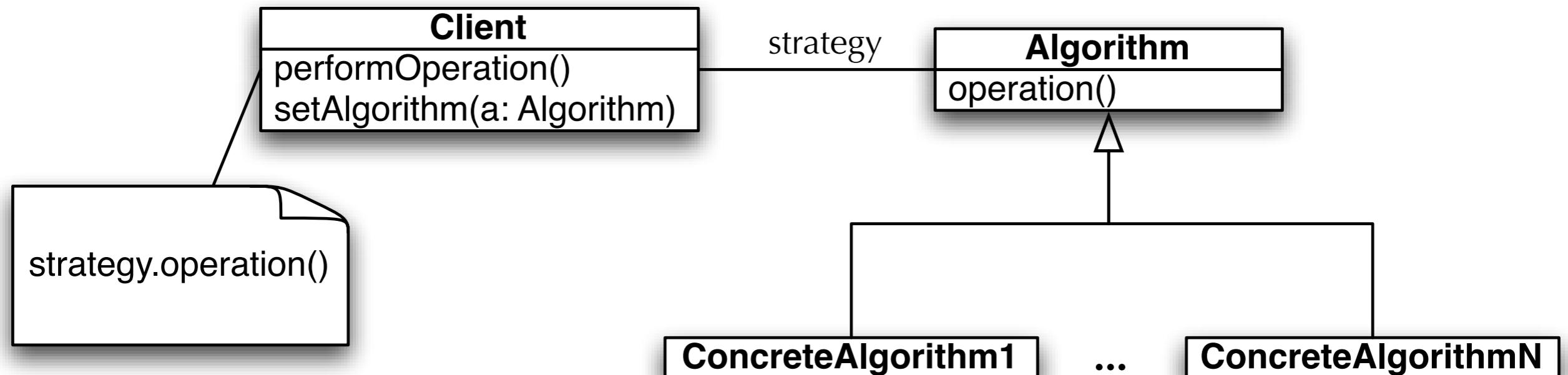
- Don't call us, we'll call you
- Or, in OO terms, high-level components call low-level components, not the other way around
 - In the context of the template method pattern, the template method lives in a high-level class and invokes methods that live in its subclasses
- This principle is similar to the dependency inversion principle we discussed back in lecture 18 (Factory pattern): “Depend upon abstractions. Do not depend upon concrete classes.”
 - Template method encourages clients to interact with the abstract class that defines template methods as much as possible; this discourages the client from depending on the template method subclasses

Template Methods in the Wild

- Template Method is used a lot since it's a great design tool for creating frameworks
 - the framework specifies how something should be done with a template method
 - that method invokes abstract and hook methods that allow client-specific subclasses to “hook into” the framework and take advantage of/influence its services
- Examples in the Java API
 - Sorting using `compareTo()` method
 - Frames in Swing
 - Applets
- Demonstration

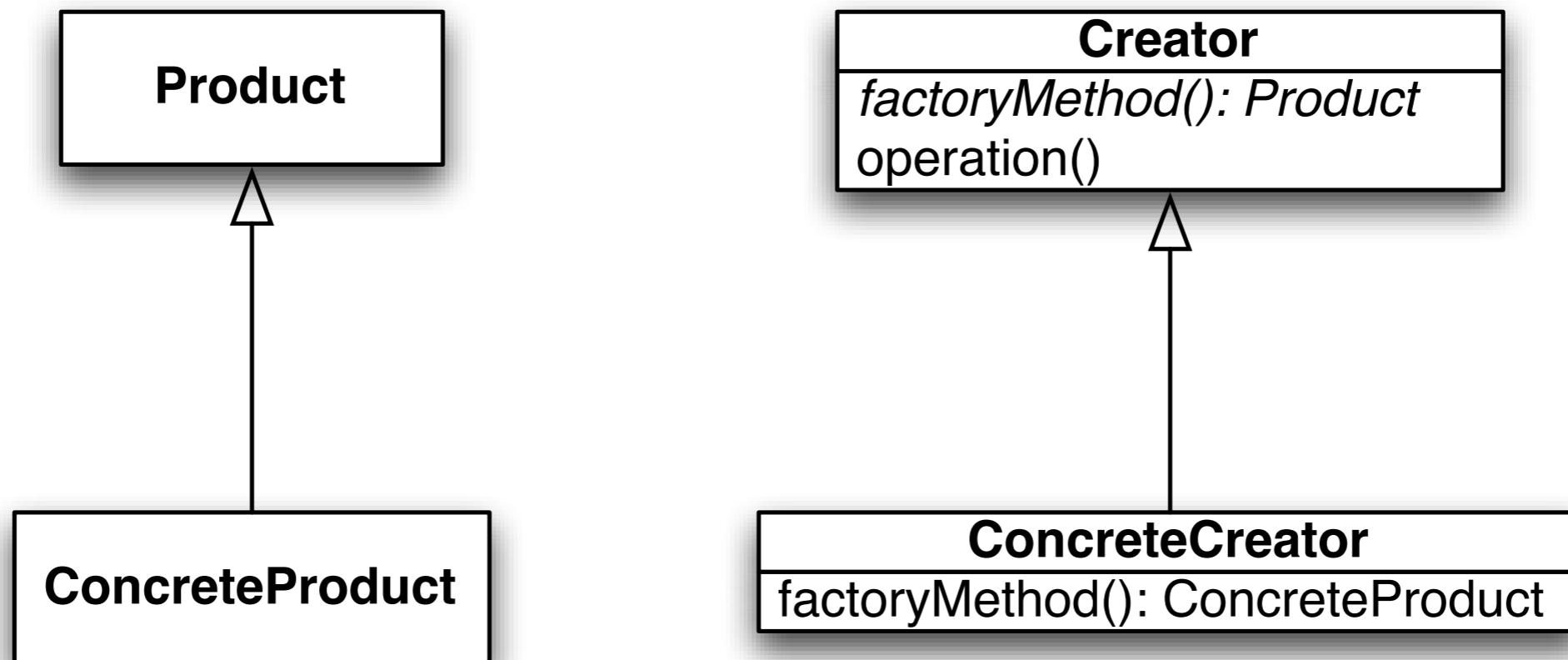
Template Method vs. Strategy (I)

- Both Template Method and Strategy deal with the encapsulation of algorithms
 - Template Method focuses encapsulation on the steps of the algorithm
 - Strategy focuses on encapsulating entire algorithms
 - You can use both patterns at the same time if you want
- Strategy Structure



Template Method vs. Strategy (II)

- Template Method encapsulate the details of algorithms using inheritance
 - Factory Method can now be seen as a specialization of the Template Method pattern



- In contrast, Strategy does a similar thing but uses composition/delegation

Template Method vs. Strategy (III)

- Because it uses inheritance, Template Method offers code reuse benefits not typically seen with the Strategy pattern
- On the other hand, Strategy provides run-time flexibility because of its use of composition/delegation
 - You can switch to an entirely different algorithm when using Strategy, something that you can't do when using Template Method

Collections-Related Design Patterns

- Collections are ubiquitous in programming
 - Lists (Array, Stack, Queue), Hash Table, Trees, ...
- When you use these collections within classes, you are making implementation decisions that need to be encapsulated
 - You don't want to expose the details of the collections that your class uses to get its job done
 - you are increasing the coupling of your system (perhaps unnecessarily) since clients are tied to your class and the class of the collection(s)
- Chapter 9 provides details on patterns you can use to encapsulate the details of internal collections; this gives you the freedom to change the collections you use as your requirements change with only minimal impact

Book Example: The merging of two diners

- Two restaurants in Objectville are merging
 - One is Lou's breakfast place, the other is Mel's diner
 - They want to merge their menus BUT
 - Lou used an `ArrayList` to store menu items
 - Mel used an array to store menu items
 - i.e. "MenuItem[] items"
 - Neither person wants to change their implementations, as they have too much existing code that depends on these data structures

Menu Item Implementation: Classic Data Holder

```
1 public class MenuItem {
2
3     String name;
4     String description;
5     boolean vegetarian;
6     double price;
7
8     public MenuItem(String name,
9                     String description,
10                    boolean vegetarian,
11                    double price)
12     {
13         this.name = name;
14         this.description = description;
15         this.vegetarian = vegetarian;
16         this.price = price;
17     }
18
19     public String getName() {
20         return name;
21     }
22
23     public String getDescription() {
24         return description;
25     }
26
27     public double getPrice() {
28         return price;
29     }
30
31     public boolean isVegetarian() {
32         return vegetarian;
33     }
34
35     public String toString() {
36         return (name + ", $" + price + "\n" + description);
37     }
38 }
```


Lou's Menu: Based on Array List

```
1 import java.util.ArrayList;
2
3 public class PancakeHouseMenu implements Menu {
4     ArrayList menuItems;
5
6     public PancakeHouseMenu() {
7         menuItems = new ArrayList();
8
9         addItem("K&B's Pancake Breakfast",
10             "Pancakes with scrambled eggs, and toast",
11             true,
12             2.99);
13
14         ...
15     }
16
17     public void addItem(String name, String description,
18         boolean vegetarian, double price)
19     {
20         MenuItem menuItem = new MenuItem(name, description, vegetarian, price);
21         menuItems.add(menuItem);
22     }
23
24     public ArrayList getMenuItems() {
25         return menuItems;
26     }
27
28     public String toString() {
29         return "Objectville Pancake House Menu";
30     }
31
32     // other menu methods here
33 }
34
```

← Yuck! Implementation Exposed!

Mel's Menu: Based on an array!

```
1 public class DinerMenu implements Menu {
2
3     static final int MAX_ITEMS = 6;
4
5     int numberOfItems = 0;
6
7     MenuItem[] menuItems;
8
9     public DinerMenu() {
10         menuItems = new MenuItem[MAX_ITEMS];
11
12         addItem("Vegetarian BLT",
13             "(Fakin') Bacon with lettuce & tomato on whole wheat", true, 2.99);
14         ...
15     }
16
17     public void addItem(String name, String description,
18         boolean vegetarian, double price)
19     {
20         MenuItem menuItem = new MenuItem(name, description, vegetarian, price);
21         if (numberOfItems >= MAX_ITEMS) {
22             System.err.println("Sorry, menu is full! Can't add item to menu");
23         } else {
24             menuItems[numberOfItems] = menuItem;
25             numberOfItems = numberOfItems + 1;
26         }
27     }
28
29     public MenuItem[] getMenuItems() { ←
30         return menuItems;
31     }
32
33     // other menu methods here
34 }
35
```

Yuck! Implementation Exposed!
(Again!)

Pros and Cons

- Use of ArrayList
 - Easy to add and remove items
 - No management of the “size” of the list
 - Can use a lot of memory if menu items allowed to grow unchecked
 - (Ignoring Java 1.5 generics) Items in ArrayList are untyped, need to cast returned objects when retrieving them from the collection
- Use of plain array
 - Fixed size of array provides control over memory usage
 - Items are stored with their types intact, no need to perform a cast on retrieval
 - Need to add code that manages the bounds of the array (kind of silly for programmers living in the 21st century!)

But... implementation details exposed

- Both menu classes reveal the type of their collection via the `getMenuItems()` method
 - All client code is now forced to bind to the menu class and the collection class being used
 - If you needed to change the internal collection, you wouldn't be able to do it without impacting all of your clients

Example Client: Waitress

- Implement a client of these two menus with the following specification
 - `printMenu()`: print all menu items from both menus
 - `printBreakfastMenu()`: print all breakfast items
 - `printLunchMenu()`: print all lunch items
 - `printVegetarianMenu()`: print all vegetarian menu items
- All of these methods will be implemented in a similar fashion, lets look at `printMenu()`

First: retrieve the menus

```
1  ...
2
3  PancakeHouseMenu pancakeHouseMenu = new PancakeHouseMenu();
4  ArrayList breakfastItems = pancakeHouseMenu.getMenuItems();
5
6  DinerMenu dinerMenu = new DinerMenu();
7  MenuItem[] lunchItems = dinerMenu.getMenuItems();
8
9  ...
```

Simple, but its annoying to have to use two different types to store the menu items; imagine if we needed to add a third or fourth menu?

Second: loop through items and print

```
1  ...
2
3  for (int i = 0; i < breakfastItems.size(); i++) {
4      MenuItem menuItem = (MenuItem)breakfastItems.get(i);
5      System.out.println(" " + menuItem);
6  }
7
8  for (int i = 0; i < lunchItems.length(); i++) {
9      MenuItem menuItem = lunchItems[i];
10     System.out.println(" " + menuItem);
11 }
12
13 ...
14
```

Note: differences in checking size and retrieving items; having to cast items retrieved from the ArrayList

Again, think of the impact of having to add a third or fourth list!

Design-Related Problems

- Just to emphasize, the current approach has the following design-related problems
 - Coding to an implementation rather than an interface
 - Vulnerable to changes in the collections used
 - Waitress knows the internal details of each Menu class
 - We have (in essence) duplicated code, one distinct loop per menu
- Lets solve this problem with our design principle “encapsulate what varies”
 - in this case, the details of the data structures and iteration over them

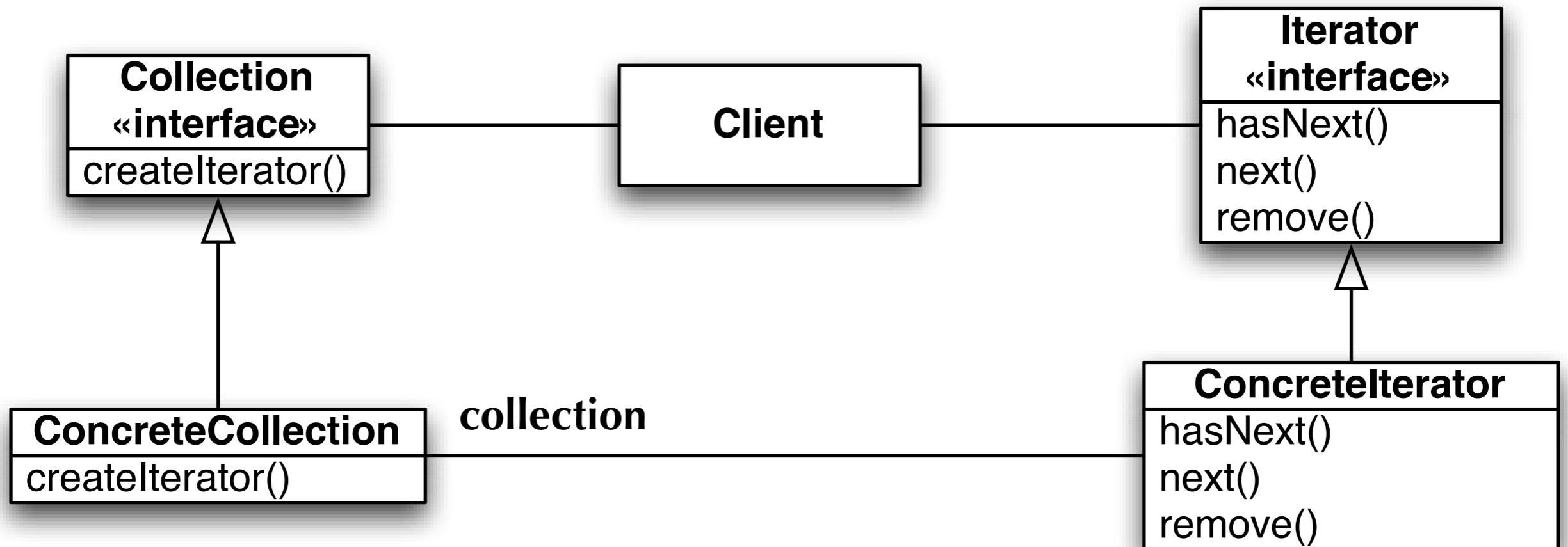
Design of Iterator

- In order to encapsulate the iteration over the data structures, we need to abstract two jobs
 - “Determine if we need to keep iterating” and “get the next item”
- Create Iterator interface with these methods
 - hasNext() (return boolean) and next() (return Object)
 - Since next() returns an Object, we’ll have to cast retrieved objects
- Create two implementations of the Iterator interface, one for each menu
- Update menu classes to return an iterator
- Update client code (Waitress) to retrieve iterators and loop over them

Iterator Pattern: Definition

- The Iterator Pattern provides a way to access the elements of a collection sequentially without exposing the collection's underlying representation
 - It also places the task of traversal on the iterator object, NOT on the collection: this simplifies the interface of the collection and places the responsibility of traversal where it should be
- Big benefit of Iterator is that it gives you a **uniform interface** for iterating over all your collections that behave polymorphically
 - Each iterator knows the details of its underlying collection and “does the right thing” as you access the hasNext() and next() methods
 - It doesn't matter if the collections are trees, lists, hash tables, or on a different machine

Iterator Pattern: Structure



Each concrete collection is responsible for creating a concrete iterator. The **ConcreteIterator** has an association back to the original collection and keeps track of its progress through that collection.

Typically, the behavior of multiple iterators over a single collection is supported as long as the collection is not modified during those traversals.

Return of Single Responsibility

- Remember the Single Responsibility Principle?
 - “A class should have only one reason to change”
- SRP is behind the idea that collections should not implement traversals
 - Collections focus on storing members and providing access to them
 - An iterator focuses on looping over the members of a collection
 - For trees, you can have multiple iterators, one each for in-order, pre-order, and post-order traversals, and perhaps others that iterate over members that match a given criteria
 - Similar to `printVegetarianMenu()` mentioned earlier

Adding Iterator: Define Iterator Interface

```
1 public interface Iterator {  
2     boolean hasNext();  
3     Object next();  
4 }  
5
```

Simple!

Adding Iterator: Create Iterator for DinerMenu

```
1 public class DinerMenuIterator implements Iterator {
2     MenuItem[] items;
3     int position = 0;
4
5     public DinerMenuIterator(MenuItem[] items) {
6         this.items = items;
7     }
8
9     public Object next() {
10        MenuItem menuItem = items[position];
11        position = position + 1;
12        return menuItem;
13    }
14
15    public boolean hasNext() {
16        if (position >= items.length || items[position] == null) {
17            return false;
18        } else {
19            return true;
20        }
21    }
22 }
23
```

Adding Iterator: Create Iterator for PancakeMenu

```
1 import java.util.ArrayList;
2
3 public class PancakeHouseMenuIterator implements Iterator {
4     ArrayList items;
5     int position = 0;
6
7     public PancakeHouseMenuIterator(ArrayList items) {
8         this.items = items;
9     }
10
11    public Object next() {
12        Object object = items.get(position);
13        position = position + 1;
14        return object;
15    }
16
17    public boolean hasNext() {
18        if (position >= items.size()) {
19            return false;
20        } else {
21            return true;
22        }
23    }
24 }
25
```

Adding Iterator: Update Menu Classes

- For each class
 - Delete getMenuItems() method
 - Add createIterator() method that returns the appropriate iterator
- Now, our implementation details are hidden and our client can switch to coding to an interface

Adding Iterator: Update Waitress

```
1 public class Waitress {
2
3     PancakeHouseMenu pancakeHouseMenu;
4     DinerMenu dinerMenu;
5
6     public Waitress(PancakeHouseMenu pancakeHouseMenu, DinerMenu dinerMenu) {
7         this.pancakeHouseMenu = pancakeHouseMenu;
8         this.dinerMenu = dinerMenu;
9     }
10
11     public void printMenu() {
12         Iterator pancakeIterator = pancakeHouseMenu.createIterator();
13         Iterator dinerIterator = dinerMenu.createIterator();
14
15         System.out.println("MENU\n----\nBREAKFAST");
16         printMenu(pancakeIterator);
17         System.out.println("\nLUNCH");
18         printMenu(dinerIterator);
19     }
20
21     private void printMenu(Iterator iterator) {
22         while (iterator.hasNext()) {
23             MenuItem menuItem = (MenuItem)iterator.next();
24             System.out.print(menuItem.getName() + ", ");
25             System.out.print(menuItem.getPrice() + " -- ");
26             System.out.println(menuItem.getDescription());
27         }
28     }
29
30 }
```

Only one loop; no more code duplication!

Iteration behaves polymorphically, working across multiple type of collection classes

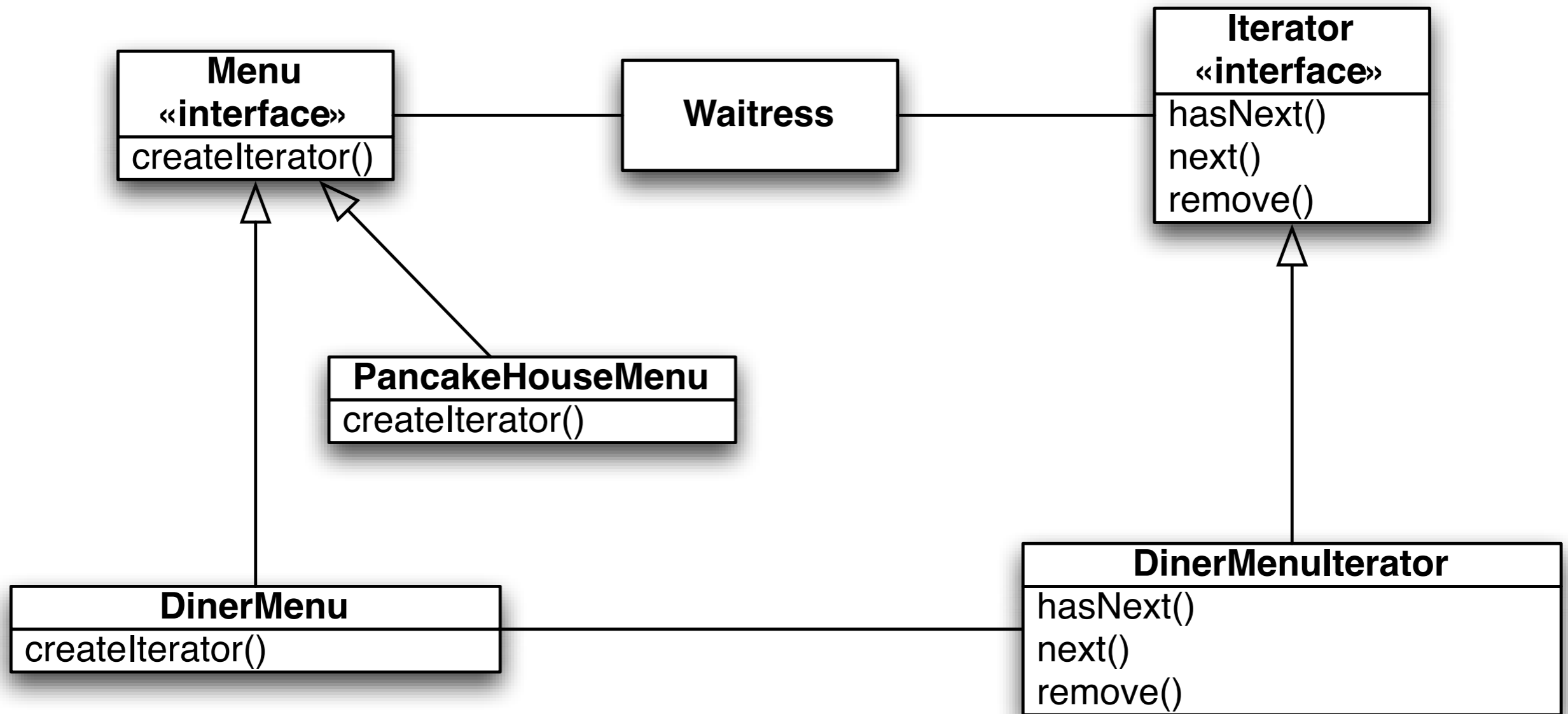
What's Missing? A common interface for Menu

- Currently Waitress is still tied to our specific Menu classes
- Create Menu interface with a single method: `createIterator()`
- Update Menu classes to implement Menu interface
- Update Waitress to hold a collection of menu objects (accessing them only via the Menu interface)
 - Now Waitress depends on two interfaces Menu and Iterator
 - Concrete Menu and Iterator classes may now come and go with ease

Why reinvent the wheel?

- Java provides an Iterator interface in `java.util`
 - It has one extra method than our homegrown iterator: `remove()`
- Lets switch our code to make use of this interface
 - Delete `PancakeHouseMenuIterator` class: `ArrayList` provides its own implementation of `java.util.Iterator`
 - Update `DinerMenuIterator` to implement `remove()` method
- With these changes, we have the following structure...

Menu Example's Structure



Demonstration

Moving On: Composite Pattern

- The Composite Pattern allows us to build structures of objects in the form of trees that contain both objects and other composites
 - Simple example: Grouping objects in a vector drawing tool
 - You can create an individual shape and apply operations to it: `move()`, `scale()`, `rotate()`, etc.
 - You can create a group of objects and apply the SAME operations to it: `move()`, `scale()`, `rotate()`, etc.
 - Client view: individual objects and groups behave in a similar fashion
- The composite pattern lets us take individual objects, group them into a composite and then deal with that group as if it was an individual object
- Using a composite structure, we can apply the same operations over both composites and individual objects allowing us to ignore their differences
 - ... for the most part; there will still be a need for code that knows the diff.

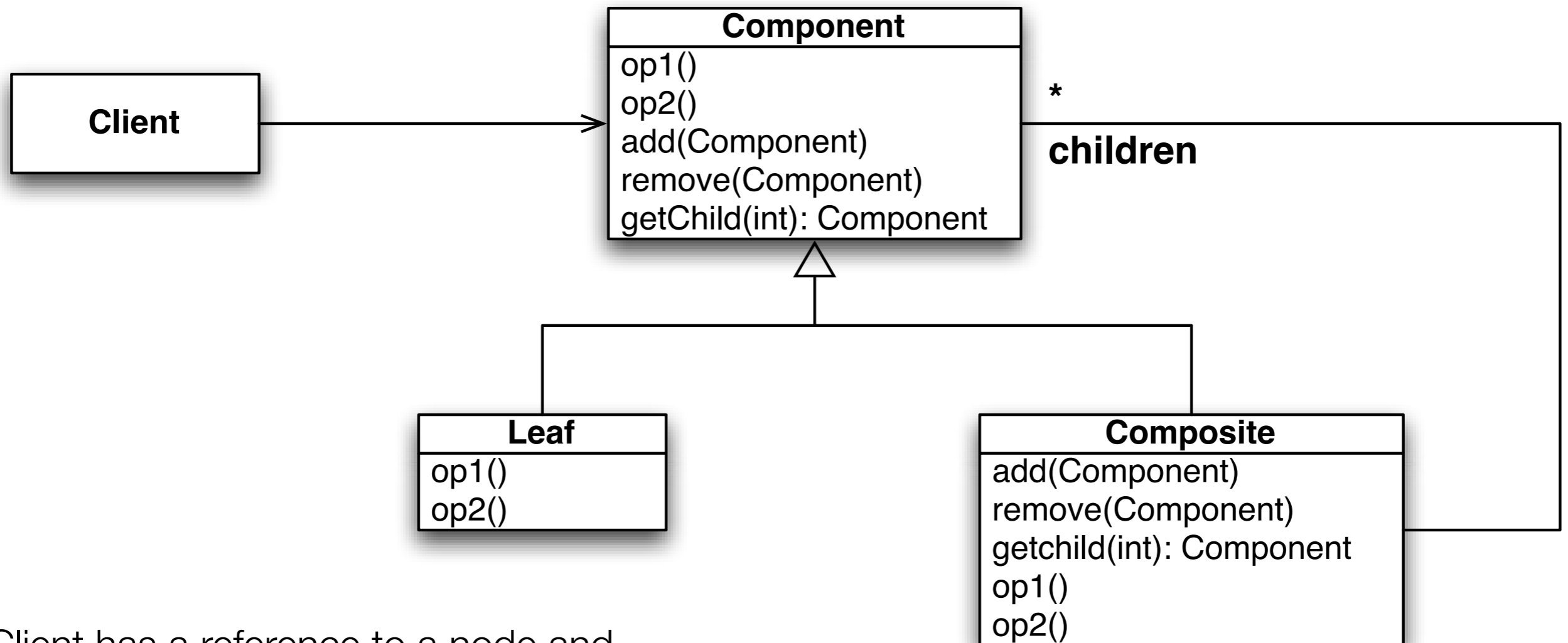
Menu Example Extended

- To explore the composite pattern, we are going to add a requirement to our menu program such that it has to allow for menus to have sub-menus
 - In particular, the diner menu is now going to feature a dessert menu
- We can view our concepts in a hierarchical fashion
 - All Menus
 - Menu Objects
 - Menu Items and Sub-Menus
 - Menu Items
- Once we have this (new) composite structure, we still need to meet all our previous requirements, such as being able to iterate over all menu items

Composite Pattern: Definition

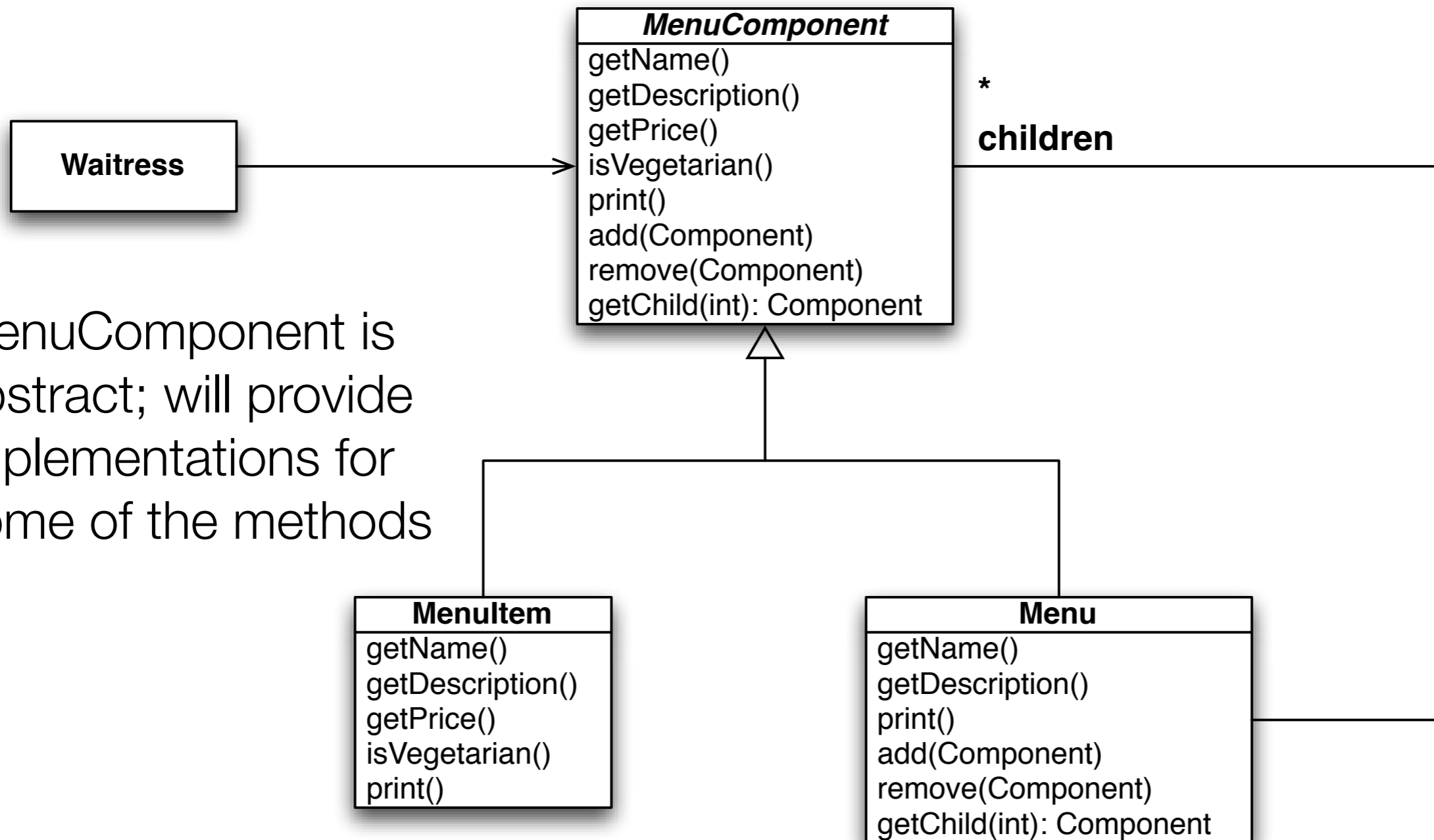
- The Composite Pattern allows you to compose objects into tree structures to represent whole-part hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly
 - Items with children are called nodes (Menus)
 - Items with no children are called leaves (Menu Items)
- We can create arbitrarily complex trees (see page 356 and 357 in textbook)
 - And treat them as groups or individuals (that is individual nodes within the tree are accessible, if needed)
 - And, we can apply an operation to the root of the tree and it will make sure that the operation is applied to all nodes within the tree
 - That is, if you apply `print()` to the root, an internal traversal makes sure that `print()` is applied to all child nodes

Composite Pattern: Structure



Client has a reference to a node and, typically, invokes shared operations on it (op1(), op2()). Component defines the **shared interface** between Composite and Leaf and adds signatures for **tree-related methods** (add(), remove(), getChild()). Leaf implements just the shared interface and ignores the tree-related methods. Composite implements the tree-related methods and implements the shared interface methods in a way that causes them to be invoked on its children. In some cases, a shared method may not make sense when applied to a Composite...

Implementing Menus as a Composite (I)



MenuComponent is abstract; will provide implementations for some of the methods

Menuitem is pretty much the same, it ignores the tree-based methods; Menu is different, it implements the tree-based methods and three of the shared operations.

Implementing Menus as a Composite (II)

- Initial steps are easy
- MenuComponent is an abstract class that implements all methods with the same line of code
 - throw new UnsupportedOperationException();
- This is a run-time exception that indicates that the object doesn't respond to this method
 - Since Menu and MenuItem are subclasses they need to override each method that they support
 - This means that both of these classes will behave the same when an unsupported method is invoked on them
- MenuItem is exactly the same as before, except now it includes the phrase "extends MenuComponent" in its class declaration

```

1 public class Menu extends MenuComponent {
2
3     ArrayList menuComponents = new ArrayList();
4
5     ...
6
7     public Menu(String name, String description) {
8         this.name = name;
9         this.description = description;
10    }
11
12    public void add(MenuComponent menuComponent) {
13        menuComponents.add(menuComponent);
14    }
15
16    public void remove(MenuComponent menuComponent) {
17        menuComponents.remove(menuComponent);
18    }
19
20    public MenuComponent getChild(int i) {
21        return (MenuComponent)menuComponents.get(i);
22    }
23
24    ...
25
26    public void print() {
27        System.out.print("\n" + getName());
28        System.out.println(", " + getDescription());
29        System.out.println("-----");
30
31        Iterator iterator = menuComponents.iterator();
32        while (iterator.hasNext()) {
33            MenuComponent menuComponent =
34                (MenuComponent)iterator.next();
35            menuComponent.print();
36        }
37    }
38 }
39

```

Menu uses an ArrayList to store its children; making the implementation of add(), remove(), and get() trivial

Menus have names and descriptions. Getter methods for these attributes are not shown.

The print() operation displays the menu's title and description and then uses ArrayList's iterator to loop over its children; it invokes print() on each of them, thus (eventually) displaying information for the entire menu.

Demonstration

Design Trade-Offs

- The Composite Pattern violates one of our design principles
 - The Single Responsibility Principle
- In particular, the design of Composite is handling two responsibilities, tree-related methods and component-related methods
 - Menu IS-A Menu AND Menu IS-A Node
 - MenuItem IS-A MenuItem AND MenuItem IS-A Leaf
- Even worse, both Menu and MenuItem inherit methods that they don't use!
- BUT, we gain transparency! Our client code can treat nodes and leaves in the same way... it doesn't care which one its pointing at!
 - And sometimes that characteristic is worth violating other principles
 - As with all trade-offs, you have to evaluate the benefits you are receiving and decide if they are worth the cost

Adding Iterator to Composite

- Producing your own iterator for a composite is straightforward
 - Add a `createIterator()` to `MenuComponent`
 - Have `Leaf` return a `NullIterator`
 - a `NullIterator`'s `hasNext()` method always returns `false`
 - Implement the traversal semantics that you want for your `Composite`'s iterator
 - The code will be different depending on whether you want an in-order, pre-order, or post-order traversal of the tree
 - The book shows code for a pre-order traversal of the `Menu` tree
- Demonstration

Wrapping Up: Two Collection-Oriented Patterns

- Iterator: separate the management of a collection from the traversal of a collection
- Composite: allow individual objects and groups of objects to be treated uniformly. Side Note: Caching
 - if the purpose of a shared operation is to calculate some value based on information stored in the node's children
 - then a composite pattern can add a field to each node that ensures that the value is only calculated once.
 - the first time the operation is called, we traverse the children, compute the value, and store it in the root
 - thereafter, we return the cached value.
 - this technique requires monitoring changes to the tree to clear out cached values that are stale.
- Iterator can be used within the context of the Composite Pattern

Coming Up Next

- Lecture 24: State and Proxy
 - Read Chapters 10 and 11 of the Design Patterns Textbook
- Lecture 25: Patterns of Patterns
 - Read Chapter 12 of the Design Patterns Textbook