Iterator and Composite

Kenneth M. Anderson
University of Colorado, Boulder
CSCI 4448/6448 — Lecture 27 — 11/27/2007

Lecture Goals

- Cover Material from Chapter 9 of the Design Patterns Textbook
  - Iterator Pattern
  - Composite Pattern

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

```java
public class MenuItem {

    String name;
    String description;
    boolean vegetarian;
    double price;

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

    public String getName() {
        return name;
    }

    public String getDescription() {
        return description;
    }

    public double getPrice() {
        return price;
    }

    public boolean isVegetarian() {
        return vegetarian;
    }

    public String toString() {
        return (name + ", $" + price + "\n    " + description);
    }
}
```

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! (Gasp!)

```
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

```
...

PancakeHouseMenu pancakeHouseMenu = new PancakeHouseMenu();
ArrayList breakfastItems = pancakeHouseMenu.getMenuItems();

DinerMenu dinerMenu = new DinerMenu();
MenuItem[] lunchItems = dinerMenu.getMenuItems();

...
```

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

```
...

for (int i = 0; i < breakfastItems.size(); i++) {
    MenuItem menuItem = (MenuItem)breakfastItems.get(i);
    System.out.println("" + menuItem);
}

for (int i = 0; i < lunchItems.length(); i++) {
    MenuItem menuItem = lunchItems[i]);
    System.out.println("" + menuItem);
}

...

```

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

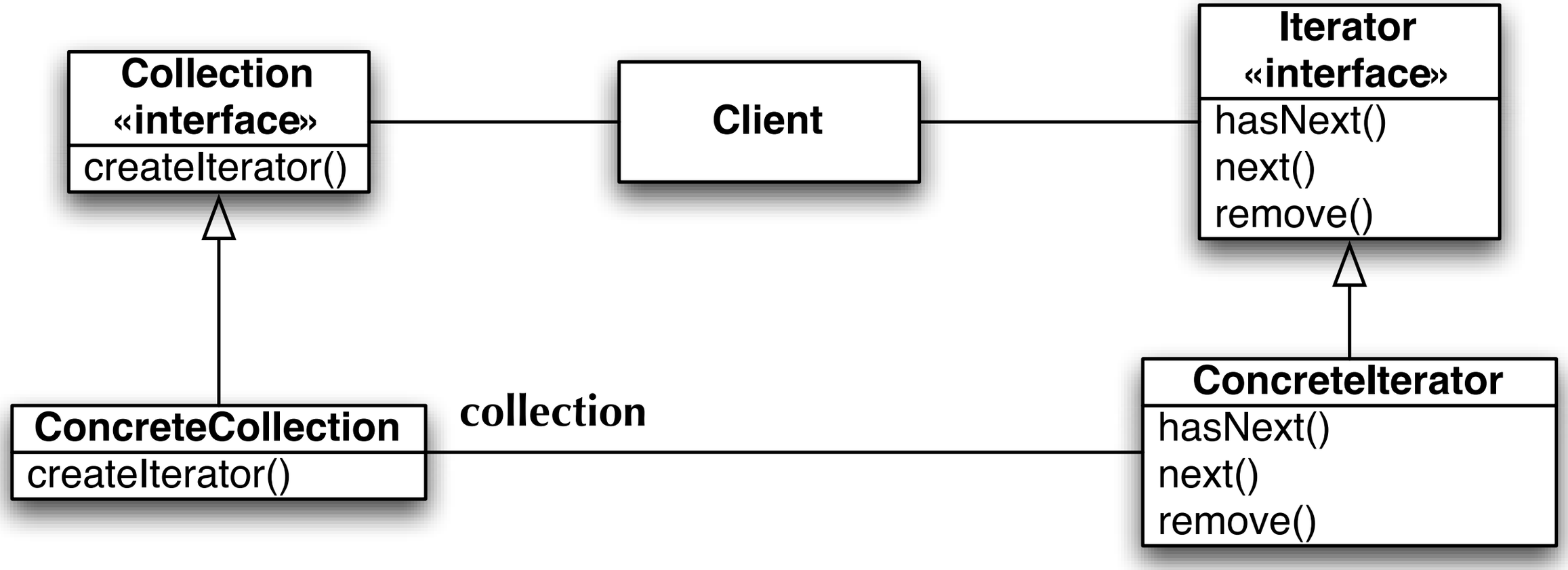
Design of Iterator

- 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 (remote iterator anyone?)

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

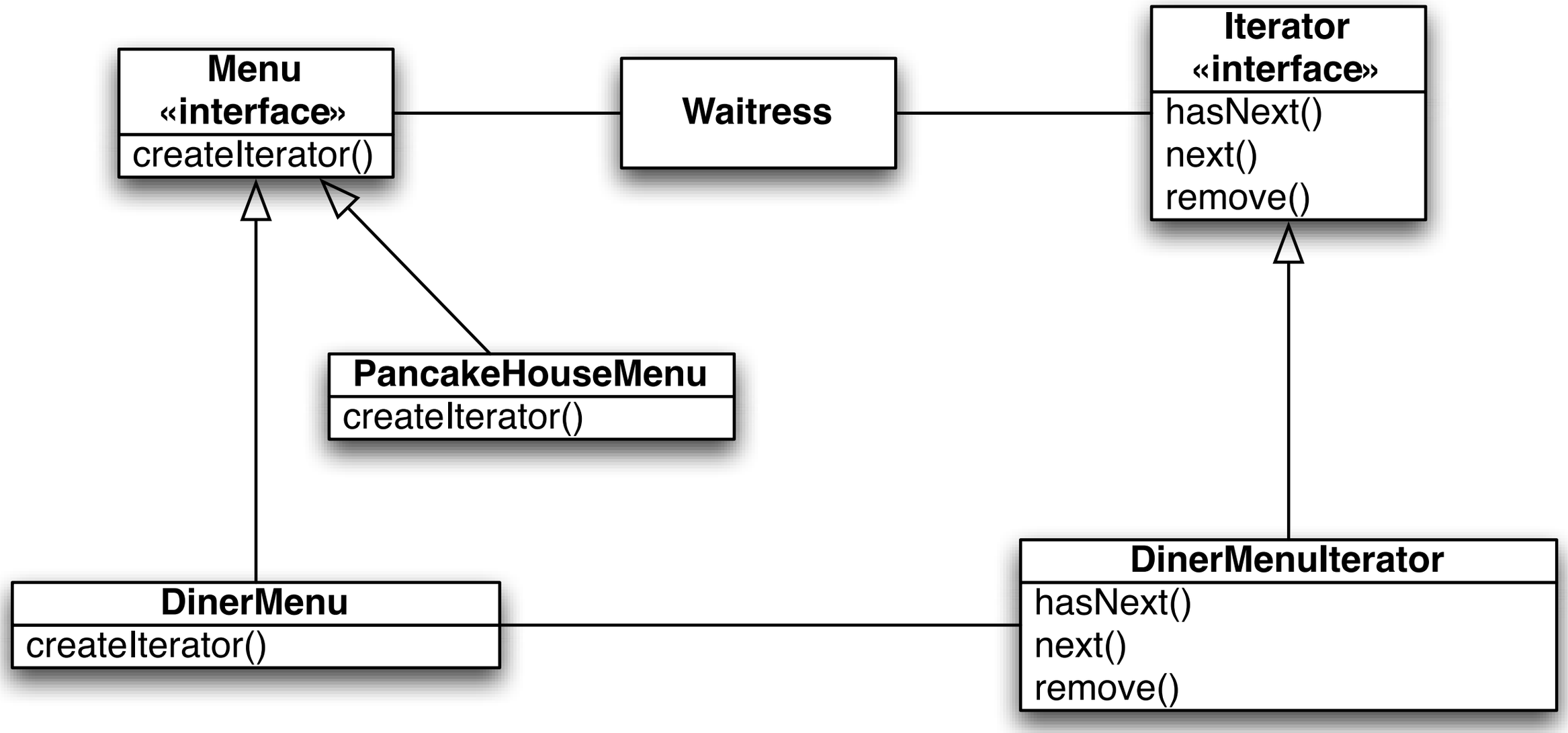
What's Missing? A common interface for Menu

- 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

Note: this design is extensible; what happens if we were to add a new menu that stores items in a hash table?

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.

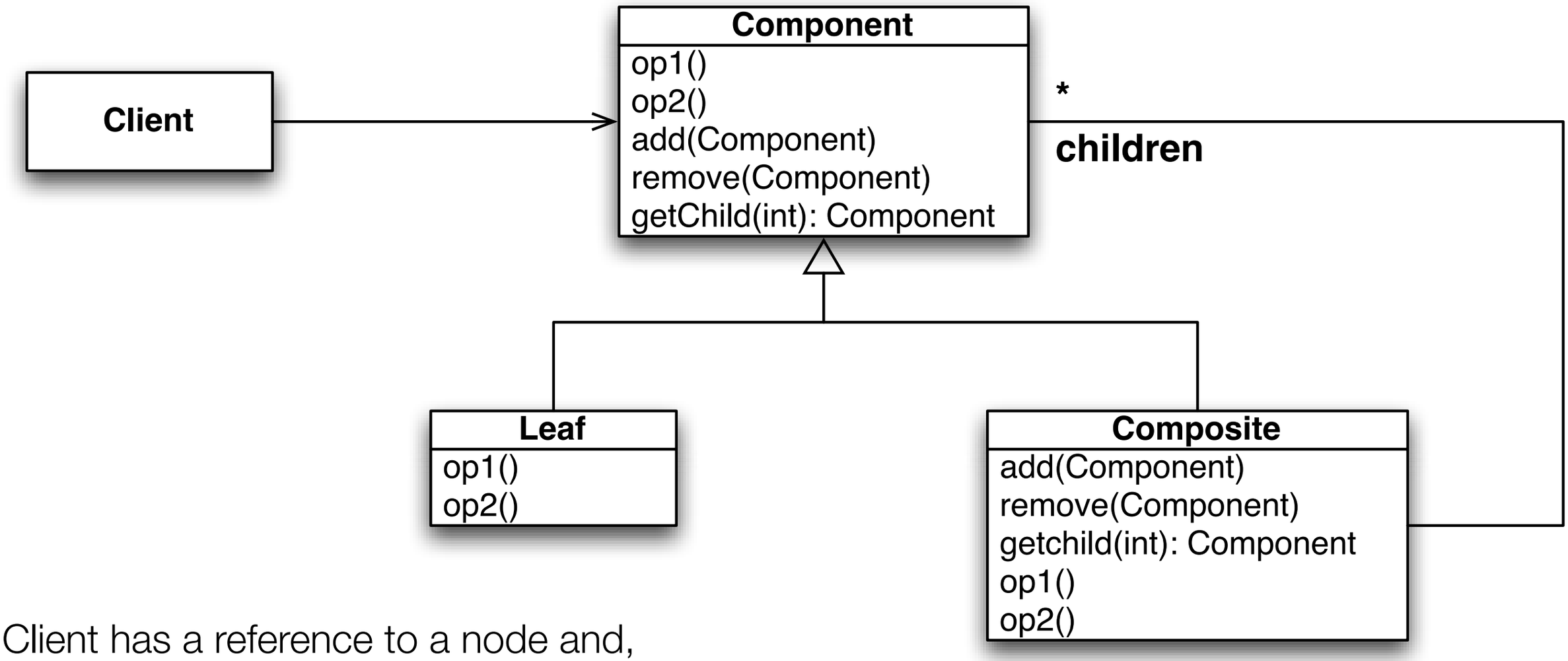
Menu Example Extended

- 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

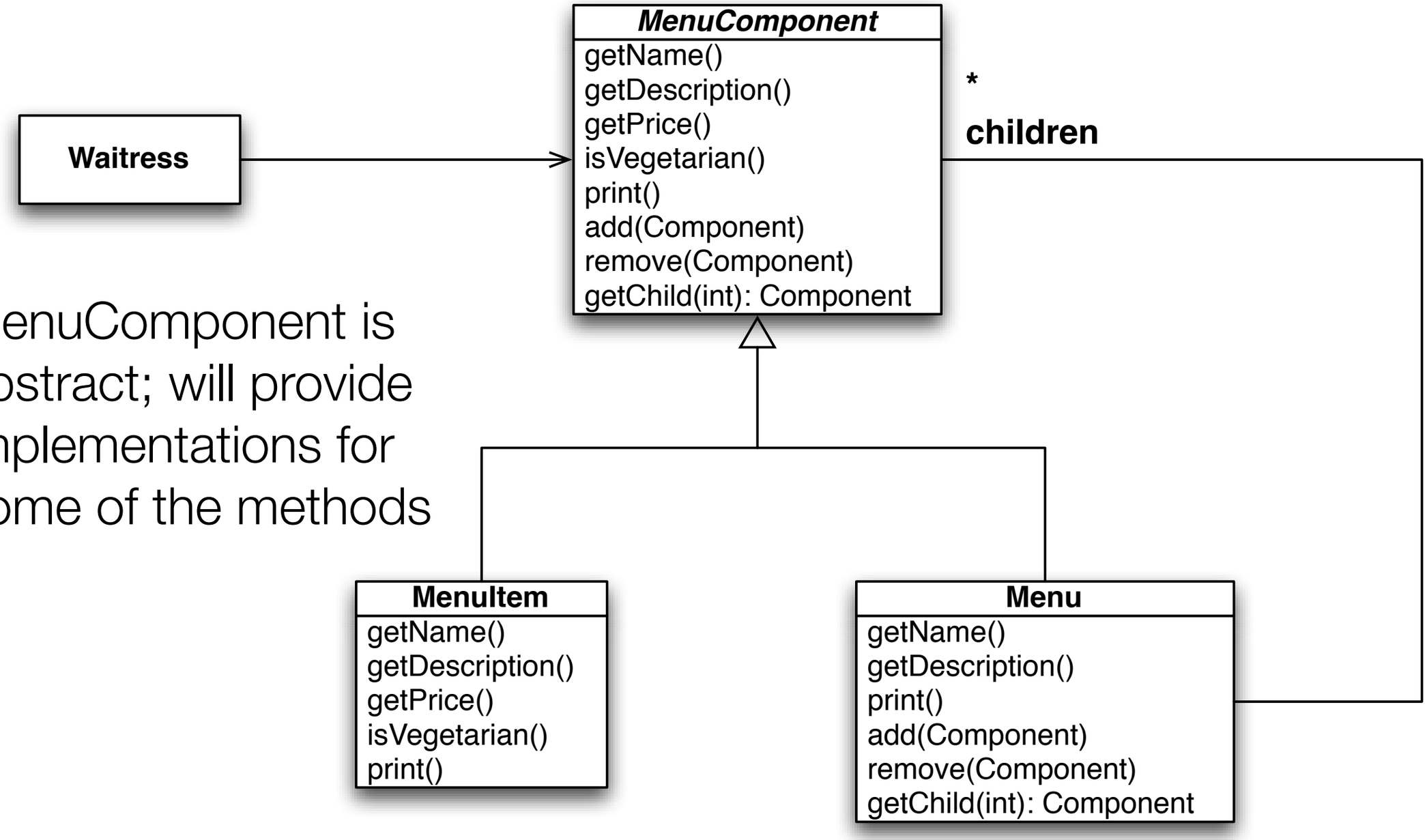
- The Composite Pattern allows you to compose objects into tree structures to represent whole-part hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly
  - Items with children are called nodes (Menus)
  - Items with no children are called leaves (Menu Items)
- 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
- 1. 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
- 2. MenuItem is exactly the same as before, except now it includes the phrase "extends MenuComponent" in its class declaration

```java
public class Menu extends MenuComponent {

    ArrayList menuComponents = new ArrayList();

    ...

    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 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();
        }
    }
}
```

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 28: State and Proxy
  - Read Chapters 10 and 11 of the Design Patterns Textbook
- Lecture 29: Patterns of Patterns
  - Read Chapter 12 of the Design Patterns Textbook