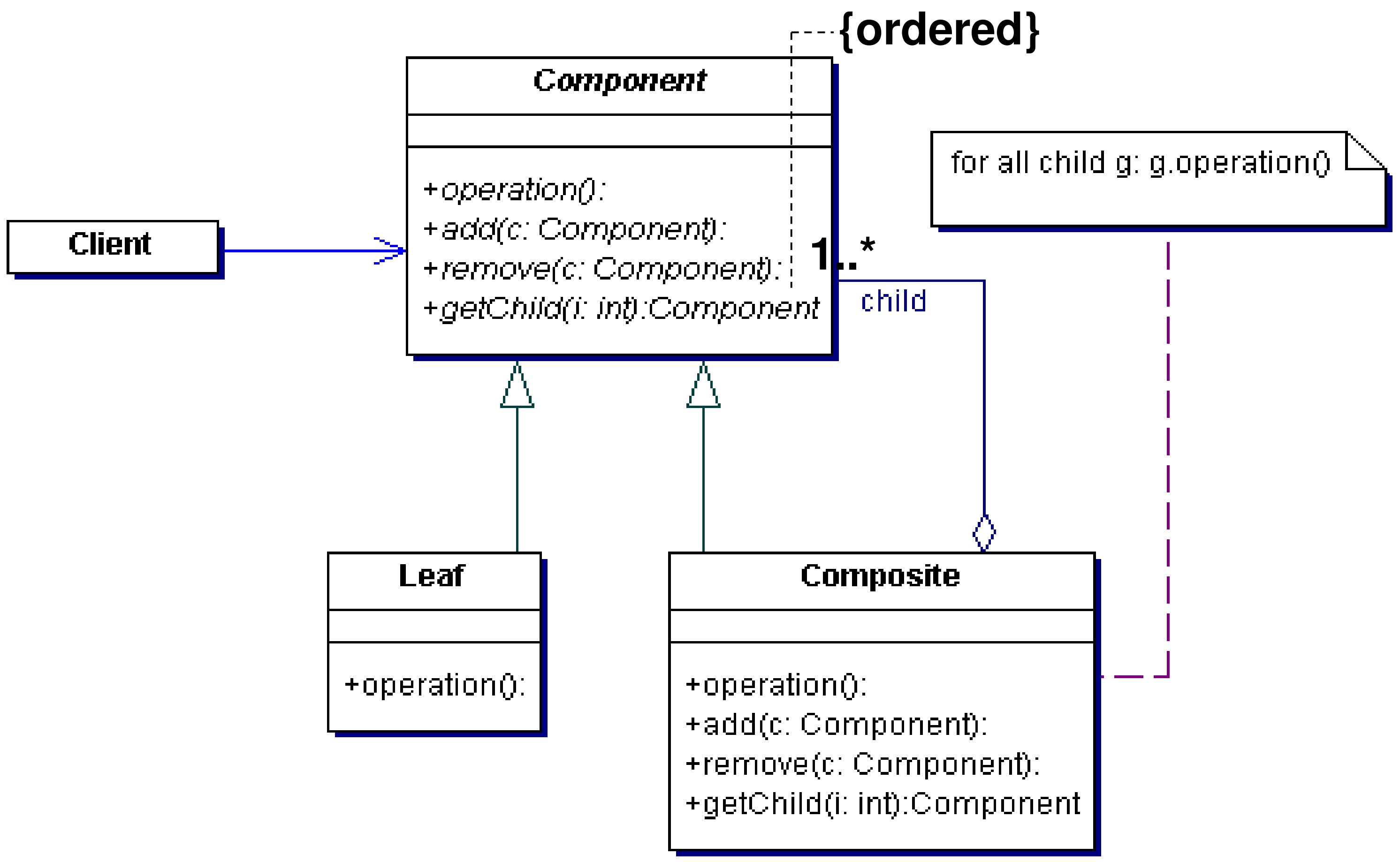
Tecniche di Progettazione: Design Patterns

GoF: Composite

Composite pattern

- Intent
  - Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly. This is called recursive composition.
- Applicability
  - Use the Composite pattern when
    - You want to represent part-whole hierarchies of objects
    - You want clients to be able to ignore the difference between compositions of objects and individual objects.

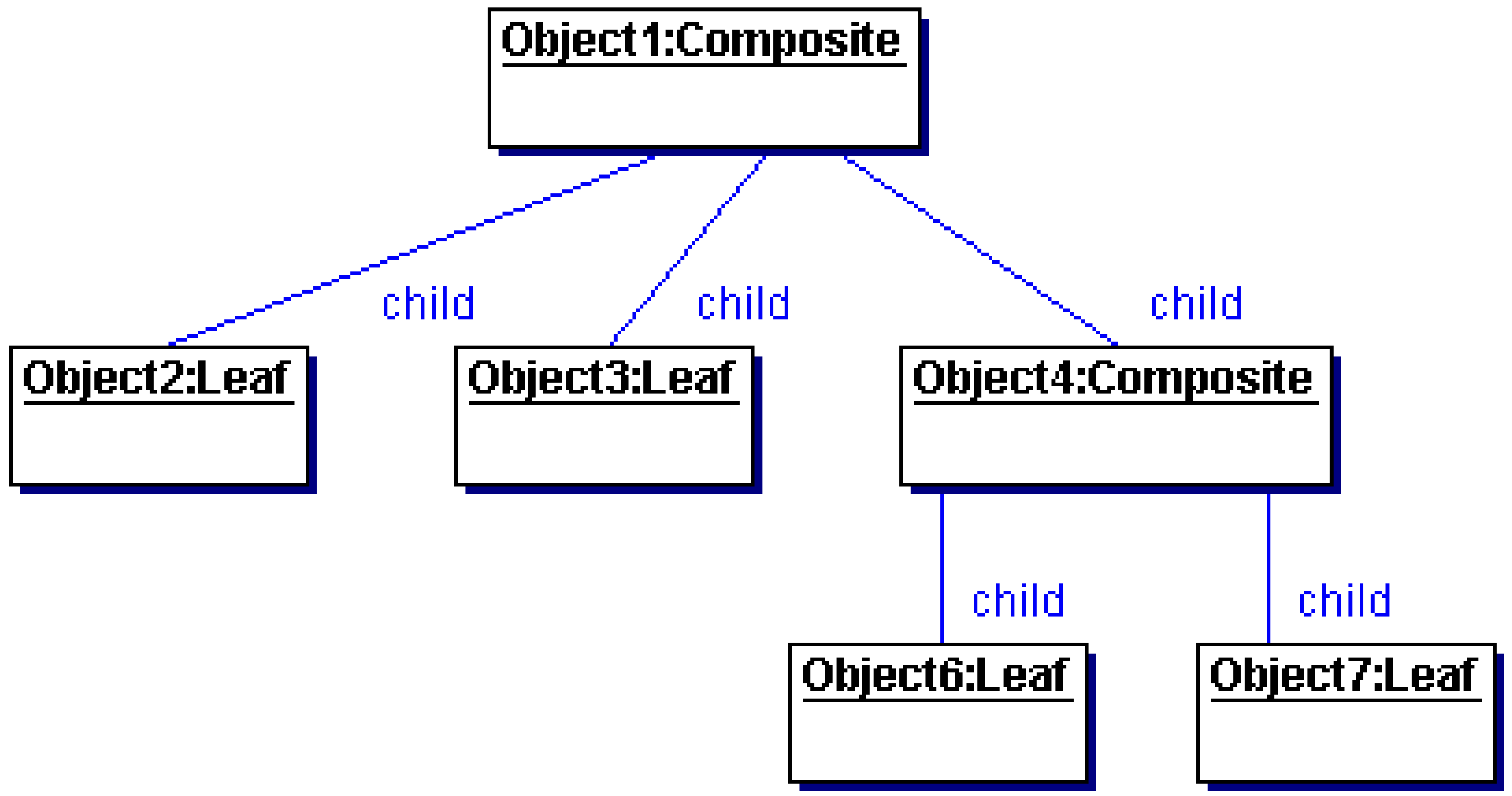
Composite: structure



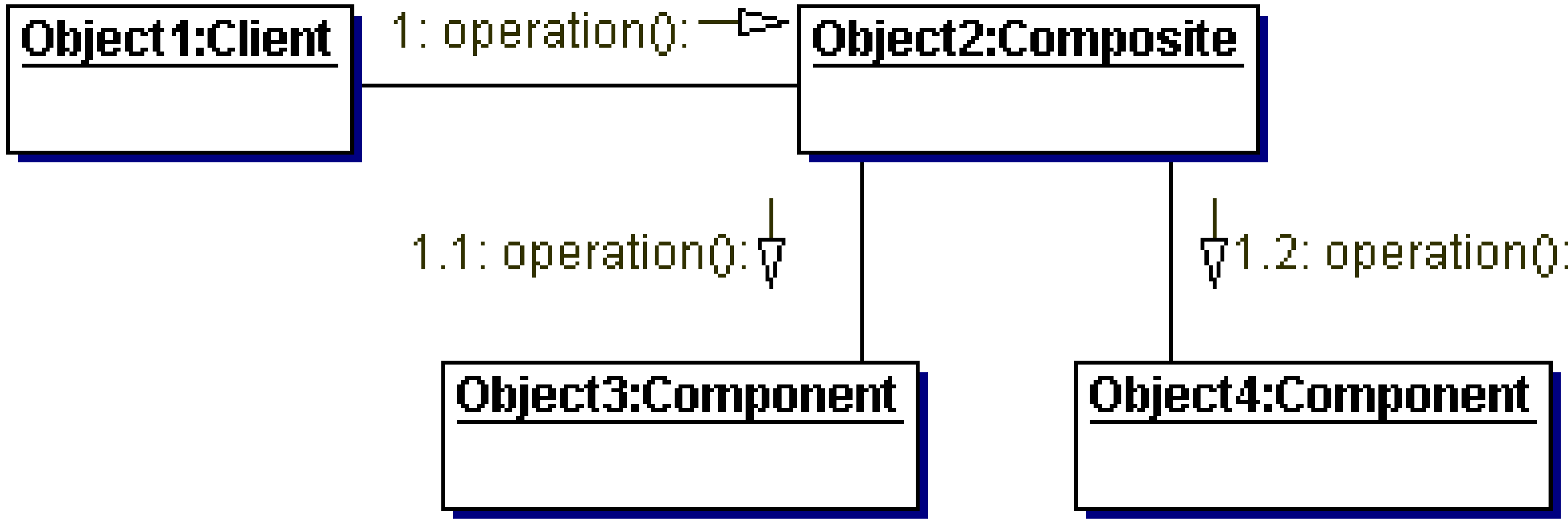
Composite: participants

- Component
  - declares the interface for object composition
  - implements default behaviour (if any)
  - declares an interface for accessing and managing the child components
- Leaf
  - Defines the behaviour of the composition primitive objects
- Composite:
  - defines behaviour for components having children
  - stores child components
  - implements operations to access childs
- Client:
  - manipulates objects in the composition through the Composite interface

Composite: Example

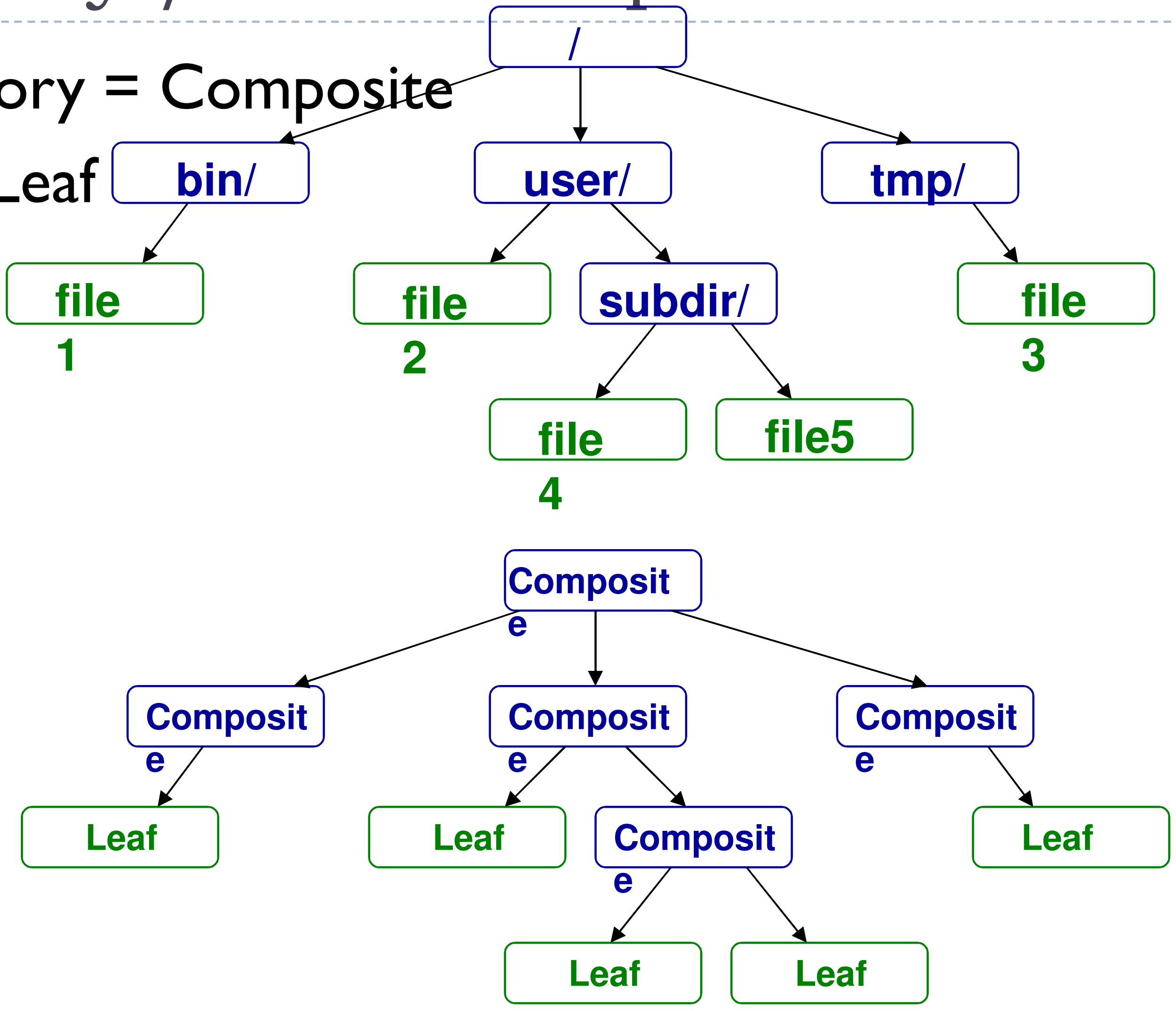


Composite: Collaboration



Directory / File Example

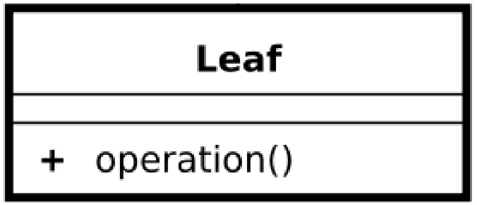
- Directory = Composite
- File = Leaf

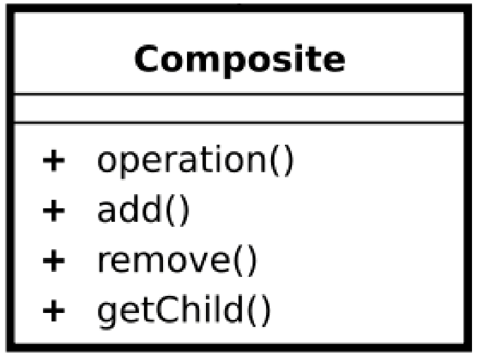


Directory / File Example – Classes

- One class for Files (Leaf nodes)
- One class for Directories (Composite nodes)
  - Collection of Directories and Files
- How do we make sure that Leaf nodes and Composite nodes can be handled uniformly?
  - Derive them from the same abstract base class

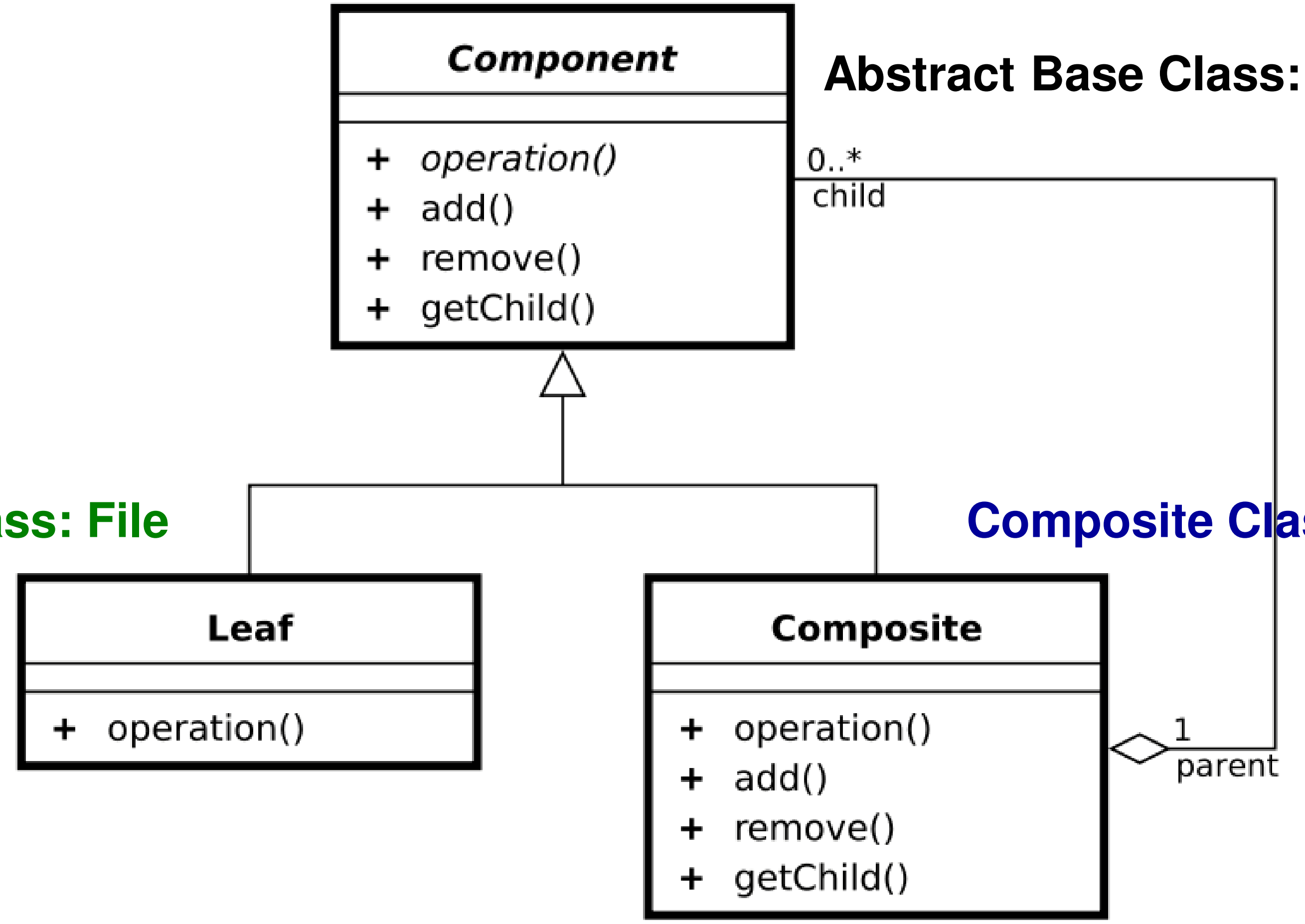
Leaf Class: File





Composite Class: Directory

Directory / File Example – Structure

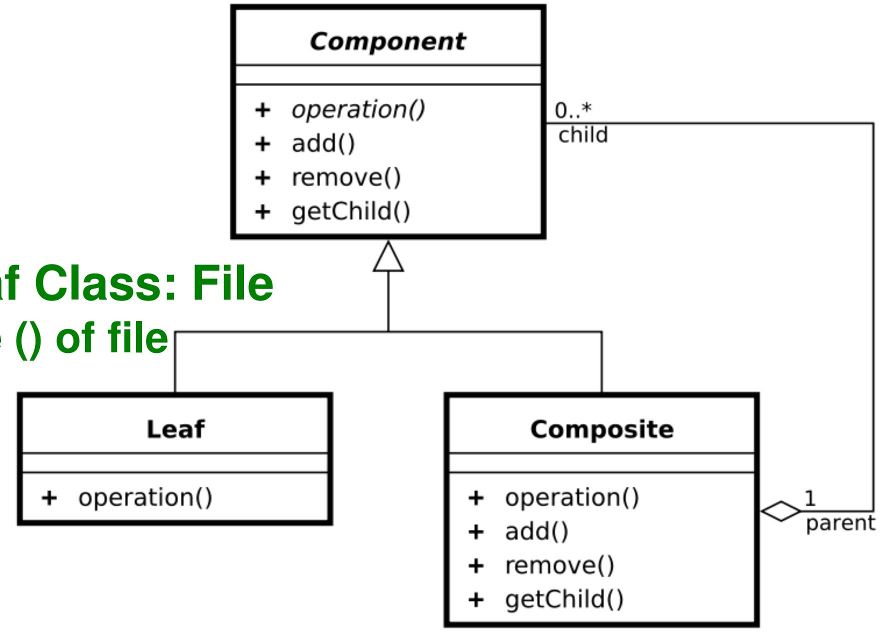


Abstract Base Class: Node

Leaf Class: File

Composite Class: Directory

Directory / File Example – Operation



Abstract Base Class: Node
size() in bytes of entire directory and sub-directories

Leaf Class: File
size () of file

Composite Class: Directory
size () Sum of file sizes in this directory and its sub-directories

```
long Directory::size () {
  long total = 0;
  Node* child;
  for (int i = 0; child = getChild(); ++i; {
    total += child->size();
  }
  return total;
}
```

Consequences

- Solves problem of how to code recursive hierarchical part-whole relationships.
- Client code is simplified.
  - Client code can treat primitive objects and composite objects uniformly.
  - Existing client code does not need changes if a new leaf or composite class is added (because client code deals with the abstract base class).
- Can make design overly general.
  - Can't rely on type system to restrict the components of a composite. Need to use run-time checks.

Implementation Issues

- Should Component maintain the list of components that will be used by a composite object? That is, should this list be an instance variable of Component rather than Composite?
  - Better to keep this part of Composite and avoid wasting the space in every leaf object.
- Where should the child management methods (add(), remove(), getChild()) be declared?
  - In the Component class: Gives transparency, since all components can be treated the same. But it's not safe, since clients can try to do meaningless things to leaf components at run-time.
  - In the Composite class: Gives safety, since any attempt to perform a child operation on a leaf component will be caught at compile-time. But we lose transparency, since now leaf and composite components have different interfaces.

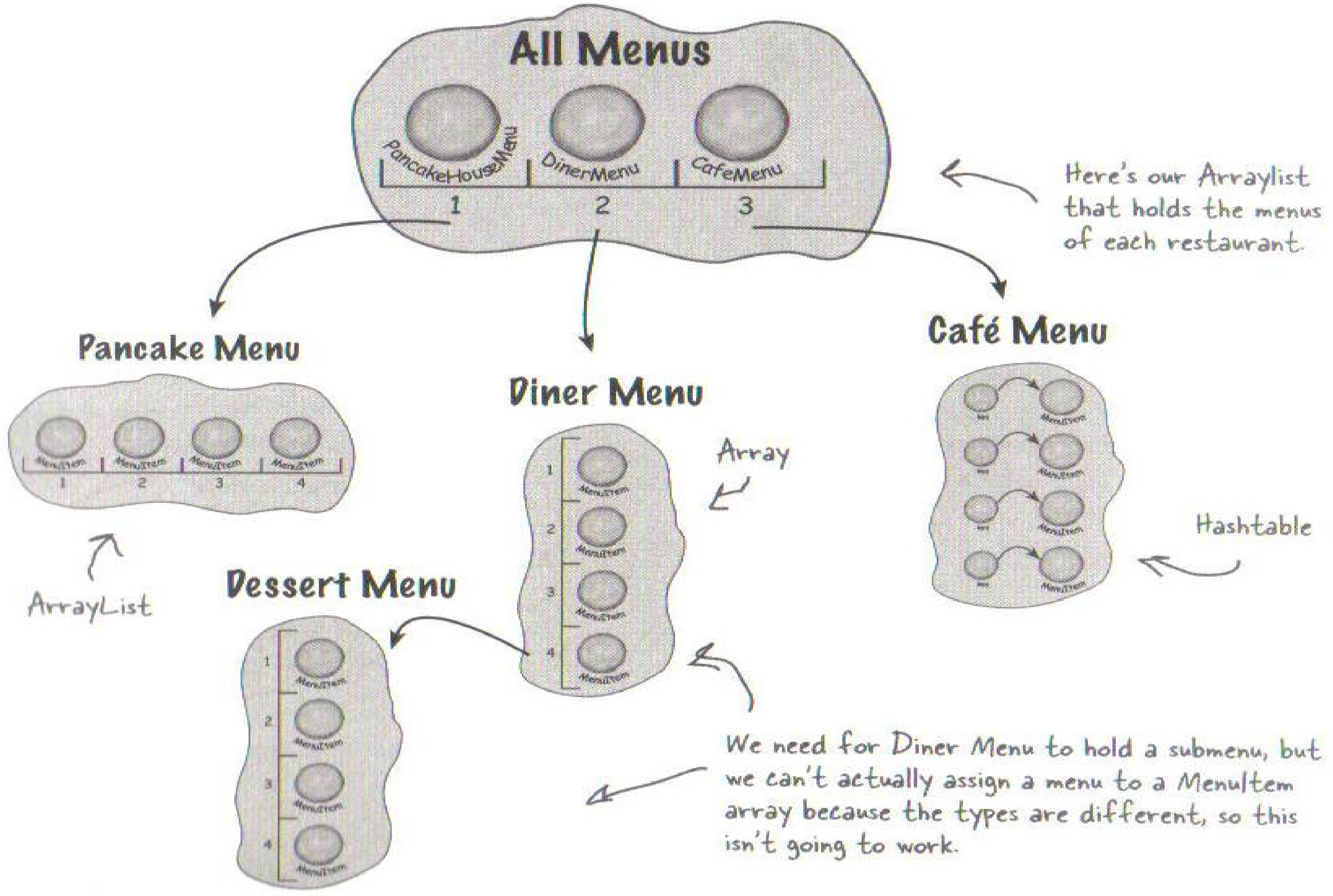
Implementation Issues cont'd

- Is child ordering important?
  - Depends on application

- What's the best data structure to store components?
  - Depends on application

- A composite object knows its contained components, that is, its children. Should components maintain a reference to their parent component?
  - Depends on application, but having these references supports the Chain of Responsibility pattern

What we want (something like this):



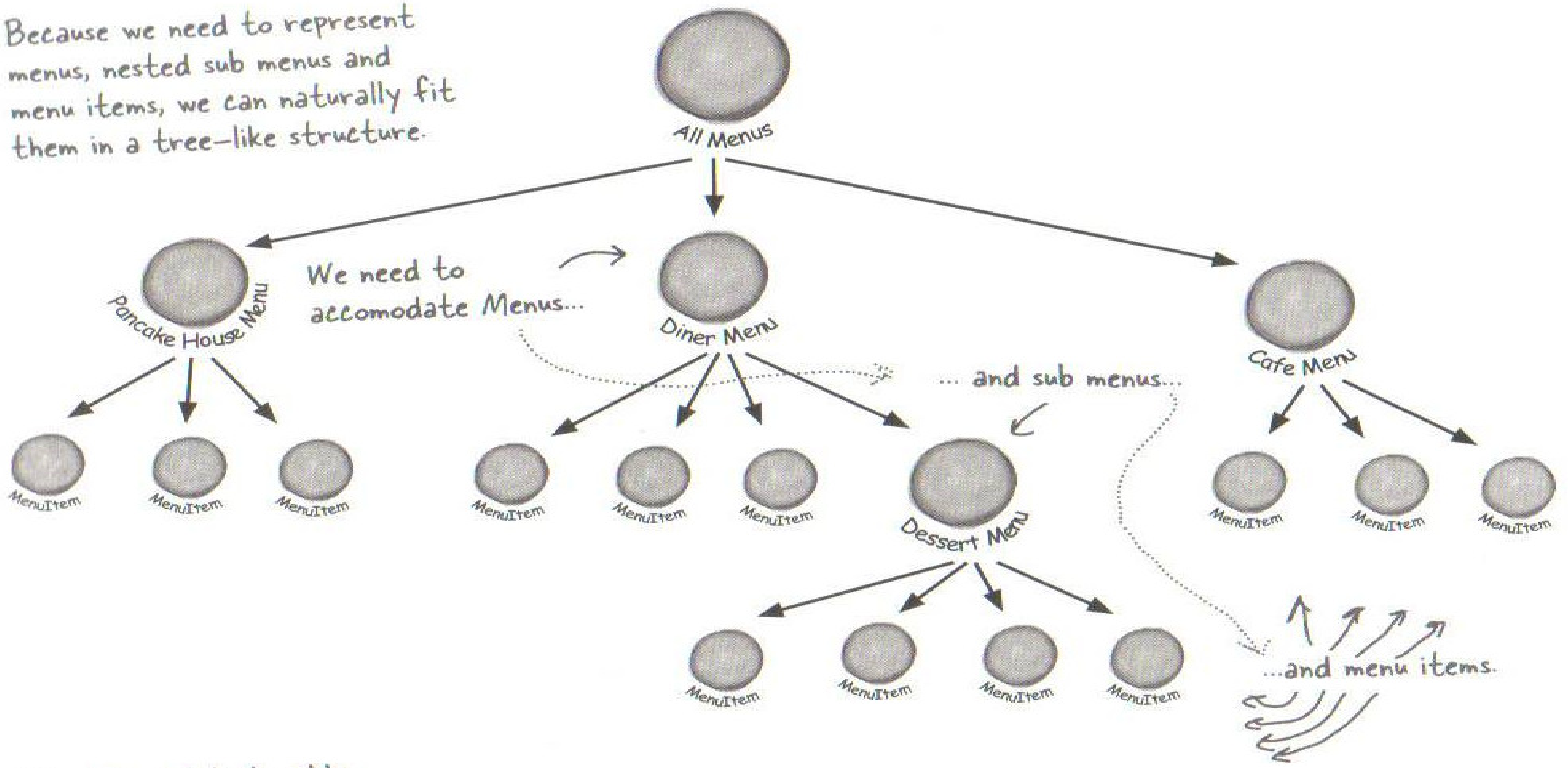
All Menus
PancakeHouseMenu
DinerMenu
CafeMenu
1
2
3
Here's our Arraylist that holds the menus of each restaurant.
Pancake Menu
MenuItem
1
2
3
4
ArrayList
Diner Menu
Array
Café Menu
Hashtable
Dessert Menu
We need for Diner Menu to hold a submenu, but we can't actually assign a menu to a MenuItem array because the types are different, so this isn't going to work.

But this won't work!

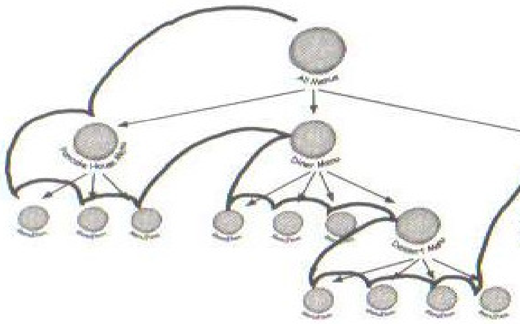
We can't assign a dessert menu to a MenuItem array.

Time for a change!

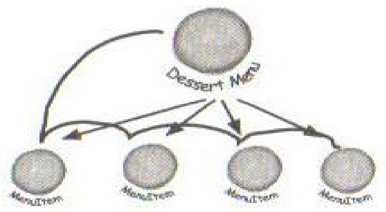
Because we need to represent menus, nested sub menus and menu items, we can naturally fit them in a tree-like structure.



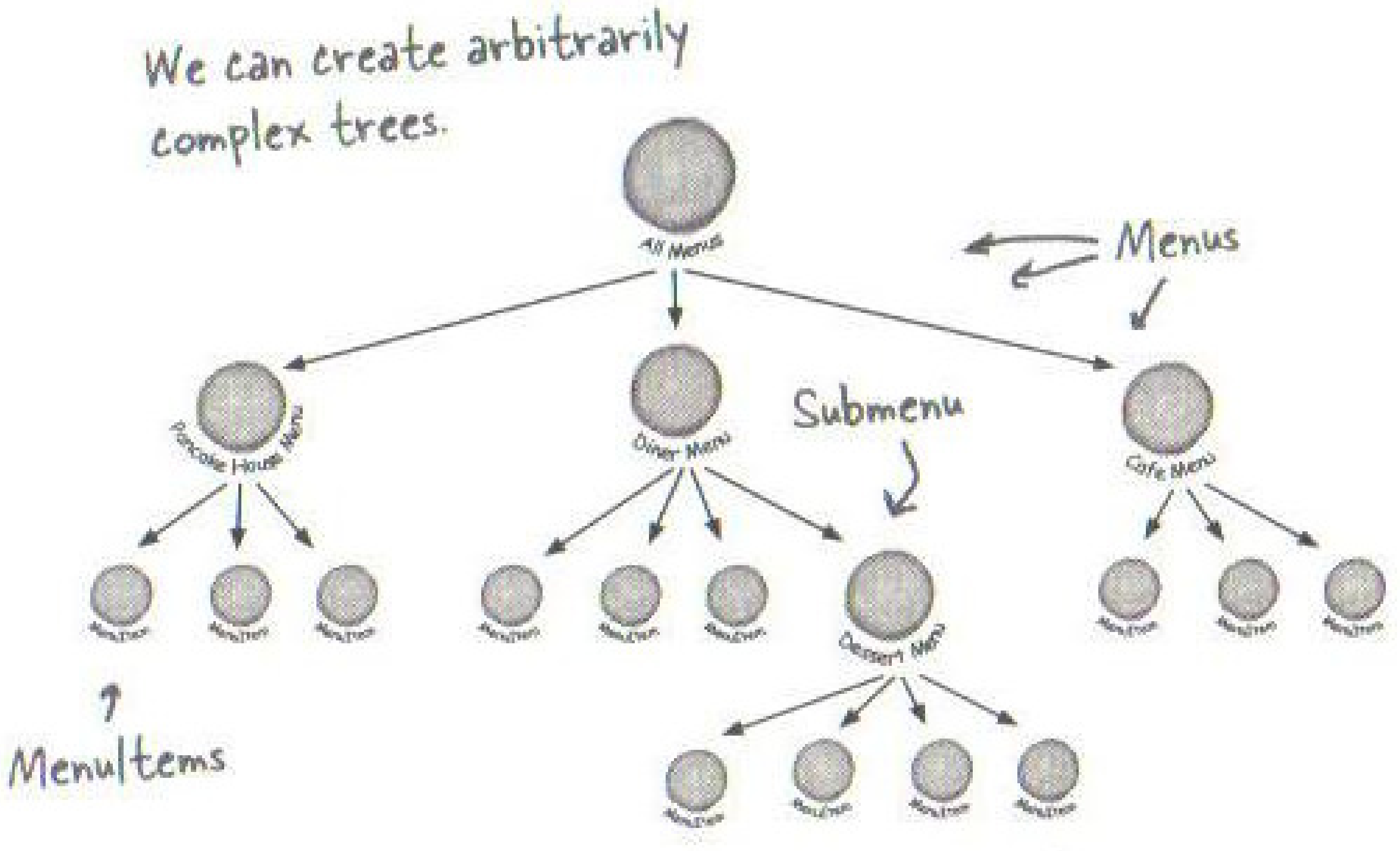
We still need to be able to traverse the all the items in the tree.



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

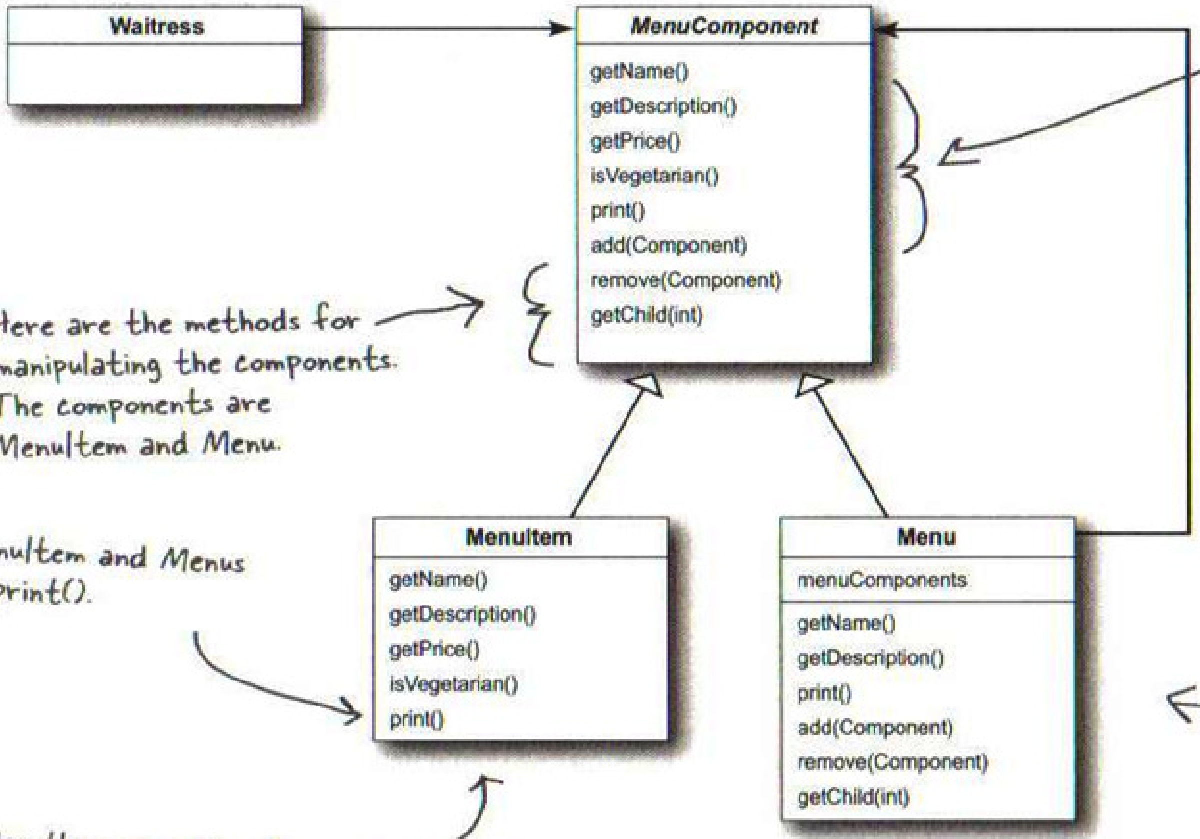


Complex hierarchy of menu items



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.

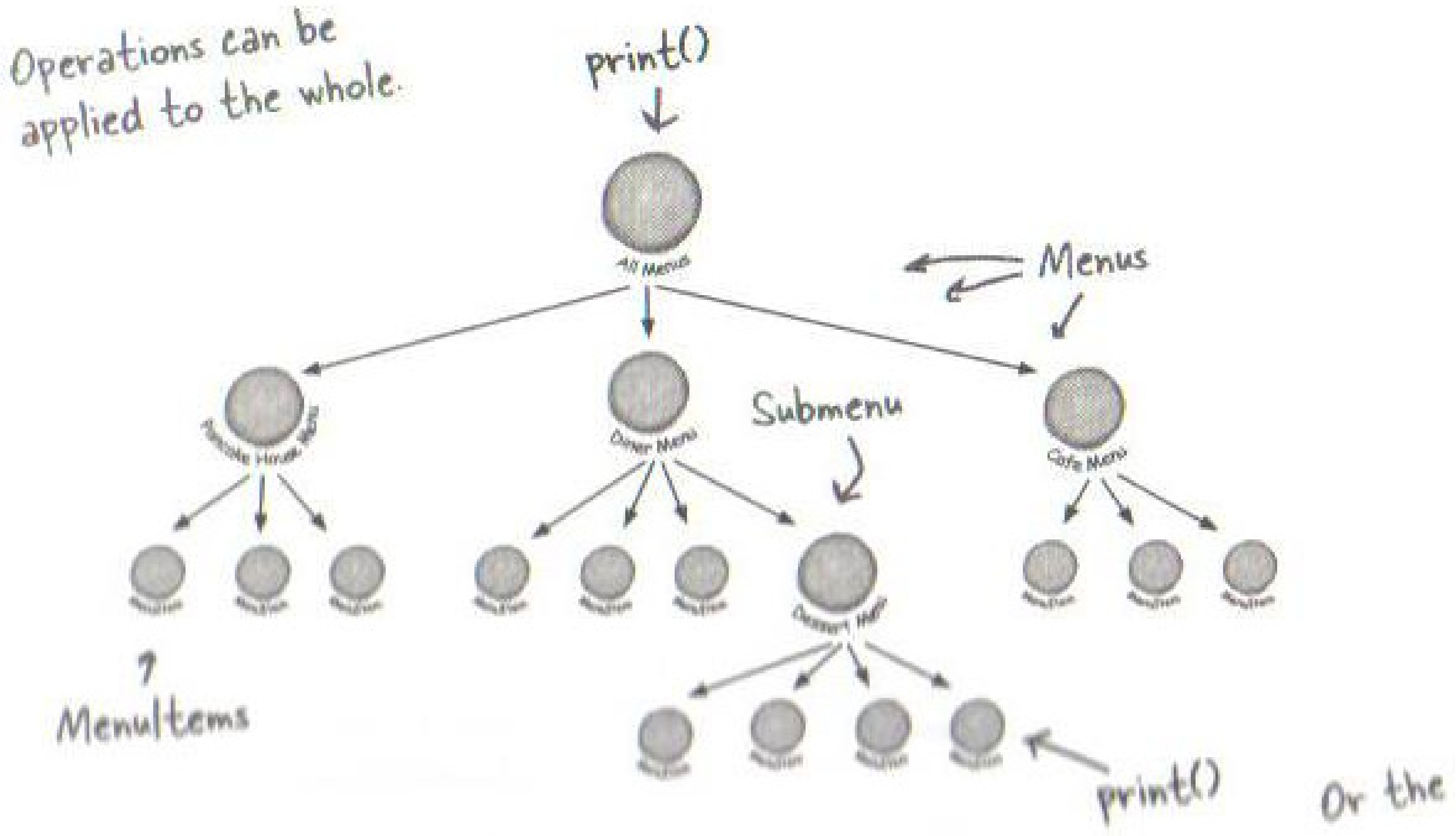
Here are the methods for manipulating the components. The components are MenuItem and Menu.

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.

Operations applied to whole or parts



Some observations

- The "print menu" method in the MenuComponent class is recursive.

- Now lets look at an alternative implementation which uses an iterator to iterate through composite classes
➔the composite iterator

MenuComponent

```
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) {
            The composite methods
            throw new UnsupportedOperationException();
    }
    public String getName() {
            throw new UnsupportedOperationException();
    }
…
```

MenuComponent

```
…
    public String getDescription() {
            throw new UnsupportedOperationException();
    }
    public double getPrice() {
            throw new UnsupportedOperationException();
    }
    public boolean isVegetarian() {
            throw new UnsupportedOperationException();
    }
    public void print() {
            throw new UnsupportedOperationException();
    }
}
```

MenuItem

```
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;
    }
…
```

MenuItem

```
…
    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());
    }
}
```

Menu

```
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);
    }
…
```

Menu

```
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());
        Iterator iterator = menuComponents.iterator();
        while (iterator.hasNext()) {
          MenuComponent menuComponent = (MenuComponent)iterator.next();
          menuComponent.print();
}
```

Related Patterns

- Chain of Responsibility – Component-Parent Link
- Decorator – When used with composite will usually have a common parent class. So decorators will need to support the component interface with operations like: Add, Remove, GetChild.
- Flyweight – Lets you share components, but they can no longer reference their parents.
- Iterator – Can be used to traverse composites.
- Visitor – Localizes operations and behavior that would otherwise be distributed across composite and leaf classes.

Homework: composite and iterator

- Using an iterator over a composite, let print all the menus, adding indentation. E.g.

```
PANCAKE HOUSE MENU
        K&B's Pancake Breakfast, Pancakes with scrambled eggs, and toast
        Regular Pancake Breakfast, Pancakes with fried eggs, sausage
        …
CAFE MENU
        Coffee Cake, Crumbly cake topped with cinnamon and walnuts
        …
DINER MENU
        Vegetarian BLT, (Fakin') Bacon with lettuce & tomato on whole wheat
        …
        DESSERT MENU
                Apple Pie, Apple pie with a flakey crust, topped with vanilla icecream
                Cheesecake, Creamy New York cheesecake, with a chocolate graham crust
```