The Iterator and Composite Patterns

Well-Managed Collections!

Breaking news…..

- The Objectville Diner and Pancake House have merged! **Menus** must be merged and have their separate identities! Owners agree on the implementation of the **MenuItems**.

```
public class MenuItem {
  String name;
  String description;
  boolean vegetarian;
  double price;

  public MenuItem(String name, String description,
                  boolean vegetarian, double price) {
    // code here
 }
  // set of getter methods to get access to the fields.
```

Menu Implementations - Pancake House

```
public class PancakeHouseMenu {
  ArrayList menuItems;
  public PancakeHouseMenu() {
    menuItems = new ArrayList();

    addItem("Regular Pancake Breakfast",
        "Pancakes with fried eggs, sausage", false, 2.99);
    addItem("Blueberry pancakes",
        "Pancakes made with fresh blueberries", true, 3.49);
    // other items
  }
  public void addItem(String name, String description,
                      boolean vegetarian, double price) {
    MenuItem menuItem = new MenuItem(name, description,
                                     vegetarian, price);
    menuItems.add(menuItem);
  }
  public ArrayList getMenuItems() {
    return menuItems;
  }
  // other methods
}
```

Uses an **ArrayList**, so the menu can be easily expanded.

To add a menuItem - create a **MenuItem** object, add it to the **ArrayList**

Dinner Menu Implementations

```
public class DinerMenu {
   static final int MAX_ITEMS = 6;
   int numberOfItems = 0;
   MenuItem[] menuItems;

   public DinerMenu() {
      menuItems = new MenuItem[MAX_ITEMS];
      addItem("Vegetarian BLT",
        "(Fakin') Bacon with lettuce & tomato on whole wheat", true, 2.99);
      // other menu items
   }
   public void addItem(String name, String description,
                       boolean vegetarian, double price) {
      MenuItem menuItem = new MenuItem(name, description, vegetarian, price);
      if (numberOfItems >= MAX_ITEMS) {
         System.err.println("Sorry, menu is full!");
      } else {
         menuItems[numberOfItems] = menuItem;
         numberOfItems = numberOfItems + 1;
      }
   }
   public MenuItem[] getMenuItems() {
      return menuItems;
   }
```

Uses an **array** of menu item

What's the problem with having two different menu representations?

- What would it take to implement the functionality that a Java-enabled `Waitress` may want:
  - `printMenu():` prints every item on the menu
  - `printBreakfastMenu()`: prints just the breakfast items
  - `printLunchMenu()`: prints just the lunch items
  - `printVegetarianMenu()`: prints all the vegetarian items
  - `isItemVegetarian(name)`: given the name of the item, returns true if vegetarian, false otherwise

Implementing the Waitress

The methods look the same, but return different types -- **ArrayList** versus **Array**

1. To print all the items on each menu:

```
PancakeHouseMenu pancakeHouseMenu = new PancakeHouseMenu();
ArrayList breakfastItems = pancakeHouseMenu.getMenuItems();
DinerMenu dinerMenu = new DinerMenu();
MenuItem[] lunchItems = dinerMenu.getMenuItems();
```

2. Print the items from the **PancakeHouseMenu** and the **DinerHouseMenu**
   - Need to loop over the **ArrayList** and **Array** respectively.

```
for (int j = 0; j < breakfastItems.size(); j++){
  MenuItem menuItem = (MenuItem)breakfastItems.get(j);
  System.out.println(menuItem.getName());
  // print out the description, vegetarian, and price
}
for (int j = 0; j < lunchItems.length; j++) {
  MenuItem menuItem = lunchItems[j]; }
```

3. Implementing the other methods is a variation on this theme.
   If another restaurant is added in, we would need three loops!

What now?

- Implementations can not be modified
  - Requires rewriting a lot of code in each respective menu
- Need: same interface for menus
  - `getMenuItems()` need to return a common object type
- How do we do that?

What now?

- One of the key principles is:
  "Encapsulate what varies".
  What varies here?
- **Iteration** caused by different collections of objects returned from the menus.
- Can we encapsulate this?
  1. To iterate through the breakfast items we use `size()` and `get()` methods on the `Arraylist`.
  2. To iterate through the lunch items we use the array `length` field and the array `subscript notation` on `MenuItem` array.

Simple Iterator

- What if we create an object, an **Iterator**, that encapsulates how we iterate through a collection of objects.

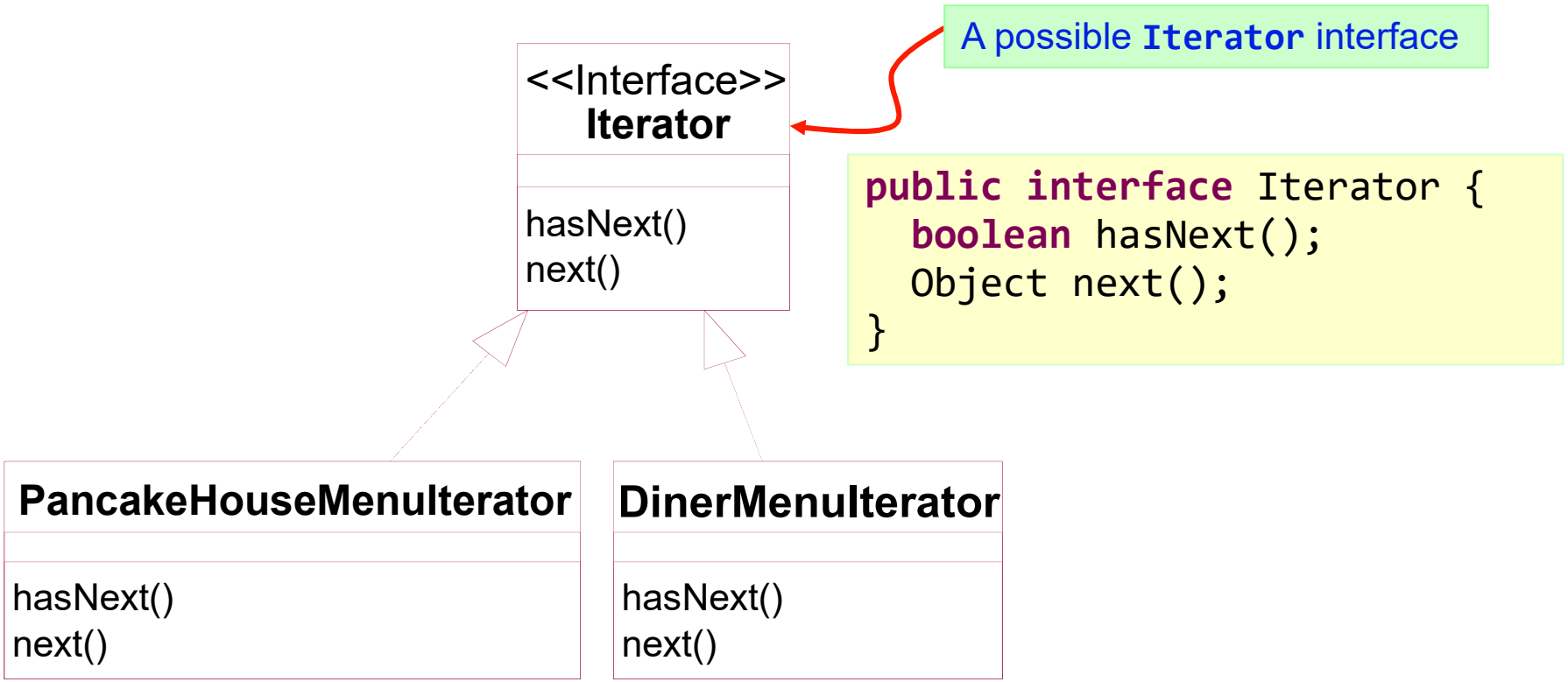
```
Iterator iterator = breakfastMenu.createIterator();
while (iterator.hasNext()) {
   MenuItem menuItem = (MenuItem) iterator.next();
}
```

- Similarly,

```
Iterator iterator = dinerMenu.createIterator();
while (iterator.hasNext()) {
   MenuItem menuItem = (MenuItem)iterator.next();
}
```

Meet the Iterator Pattern

- The **`Iterator`** Design Pattern relies on an interface called the Iterator interface.



Using the `Iterator` for the Diner Menu

```
public class DinerMenuIterator implements Iterator {
  MenuItem[] items;
  int position = 0;

  public DinerMenuIterator(MenuItem[] items) {
    this.items = items;
  }

  public Object next() {
    MenuItem menuItem = items[position];
    position = position + 1;
    return menuItem;
  }

  public boolean hasNext() {
    if (position >= items.length || items[position] == null) {
      return false;
    } else { return true; }
  }
}
```

position maintains the current position of the iteration over the array.

The constructor takes the array of menu items we are going to iterate over.

The **next()** method returns the next item in the array and increments the position.

The **hasNext()** method checks to see if we've seen all the elements of the array

Reworking the Diner Menu with Iterator

```
public class DinerMenu {
  static final int MAX_ITEMS = 6;
  int numberOfItems = 0;
  MenuItem[] menuItems;

  // constructor here

  // addItem here

  public MenuItem[] getMenuItems() {
    return menuItems;
  }

  public Iterator createIterator() {
    return new DinerMenuIterator(menuItems);
  }
  // other menu methods here
}
```

We're not going to need the **getMenuItems()** method anymore and in fact we don't want it because it exposes our internal implementation!

creates a **DinerMenuIterator** from the menuItems array and returns it to the client.

We're returning the **Iterator** interface.
The client doesn't need to know how the menuItems are maintained in the **DinerMenu**, nor how the **DinerMenuIterator** is implemented.
It just needs to use the iterators to step through the items in the menu.

Fixing up the `Waitress` Code

```
public class Waitress {
  PancakeHouseMenu pancakeHouseMenu;
  DinerMenu dinerMenu;
  public Waitress(PancakeHouseMenu pancakeHouseMenu,
                  DinerMenu dinerMenu) {
    this.pancakeHouseMenu = pancakeHouseMenu;
    this.dinerMenu = dinerMenu;
  }

  public void printMenu() {
    Iterator dinerIterator = dinerMenu.createIterator();
    System.out.println("\nLunch");
    printMenu(dinerIterator);
    // similar set of statements for Breakfast menu
  }

  public void printMenu(Iterator iterator) {
    while (iterator.hasNext()) {
       MenuItem menuItem = (MenuItem) iterator.next();
       // print it out
    }
  }
}
```

Waitress takes the two objects as before

The **printMenu()** creates two iterators one for each menu

Back to one loop! Here is where the printing occurs.

What have we done?

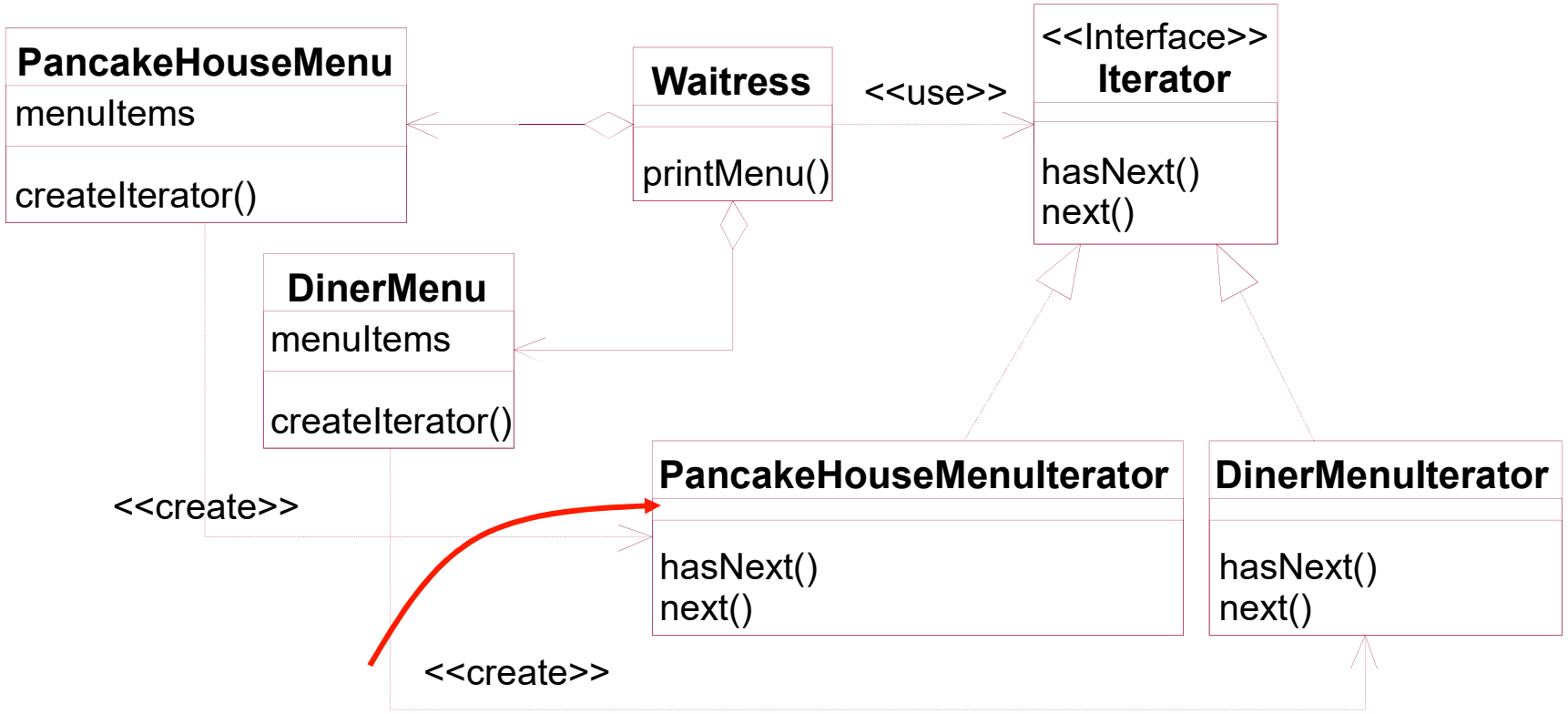
- The **Menus** are not well encapsulated; we can see the **Diner** is using an `ArrayList` and the **Pancake House** an `Array`.
- We need two loops to iterate through the `MenuItems`.
- The `Waitress` is bound to concrete classes (`MenuItem[]` and `ArrayList`)
- The `Waitress` is bound to two concrete `Menu` classes, despite their interfaces being almost identical.

- The `Menu` implementations are now encapsulated. The `Waitress` has no idea how the **Menus** hold their collection of menu items.
- All we need is a loop that polymorphically handles any collection of items as long as it implements the iterator.
- The `Waitress` now uses an interface (`Iterator`).
- The `Menu` interfaces are now exactly the same and uh, oh, we still don't have a common interface, which means the `Waitress` is still bound to two concrete `Menu` classes.

Bird's Eye View of Current Design

The waitress is still bound to two concrete Menu classes
Solution: define a common interface for two classes

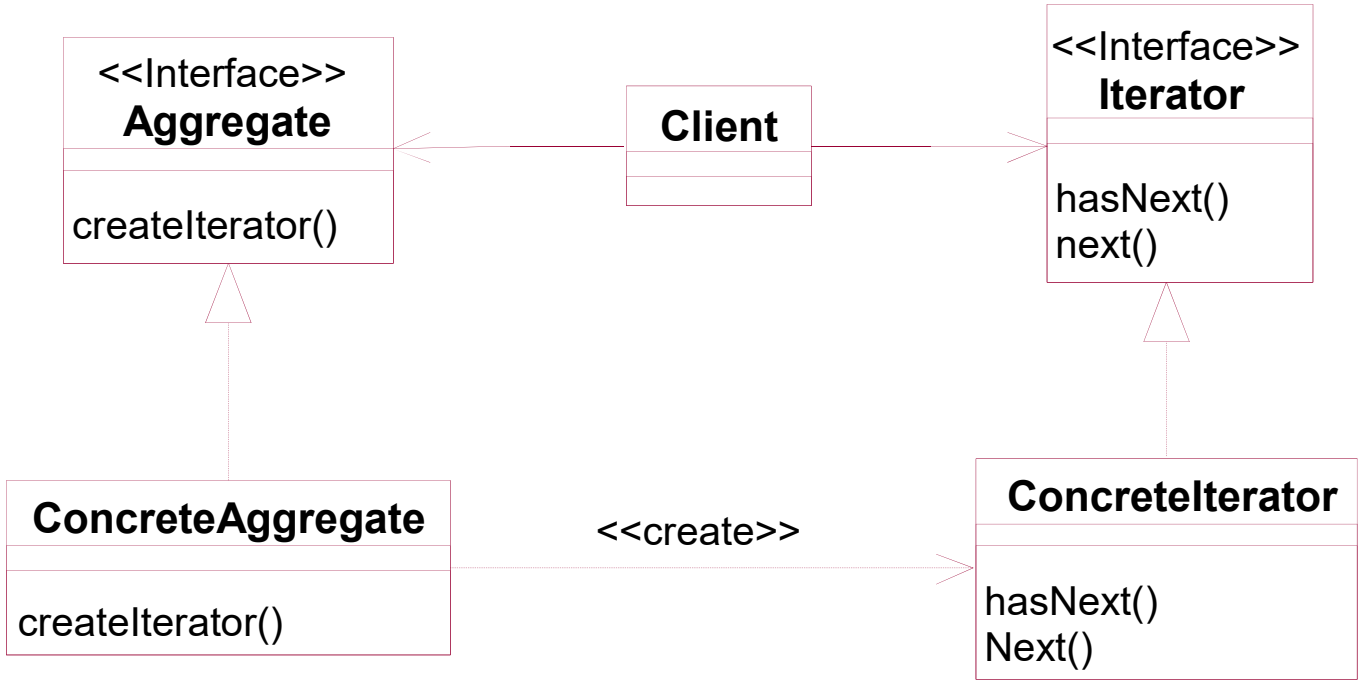
The **`Iterator`** allows the **`Waitress`** to be decoupled from the actual implementation of the concrete classes. She does not need to know if a **`Menu`** is implemented with an **`Array`** or **`ArrayList`**!
All she cares is that she can get an iterator to do her iterating.



The **`Iterator`** gives us a way to step through the elements of an aggregate without having the aggregate clutter its own interface with a bunch of methods to support traversal of its elements. It also allows the implementation of the **`iterator`** to live outside the aggregate --- in other words, we 've encapsulated the iteration.

The Iterator Pattern Defined

The **Iterator** Pattern provides a way to access the elements of an aggregate object sequentially without exposing its underlying implementation.



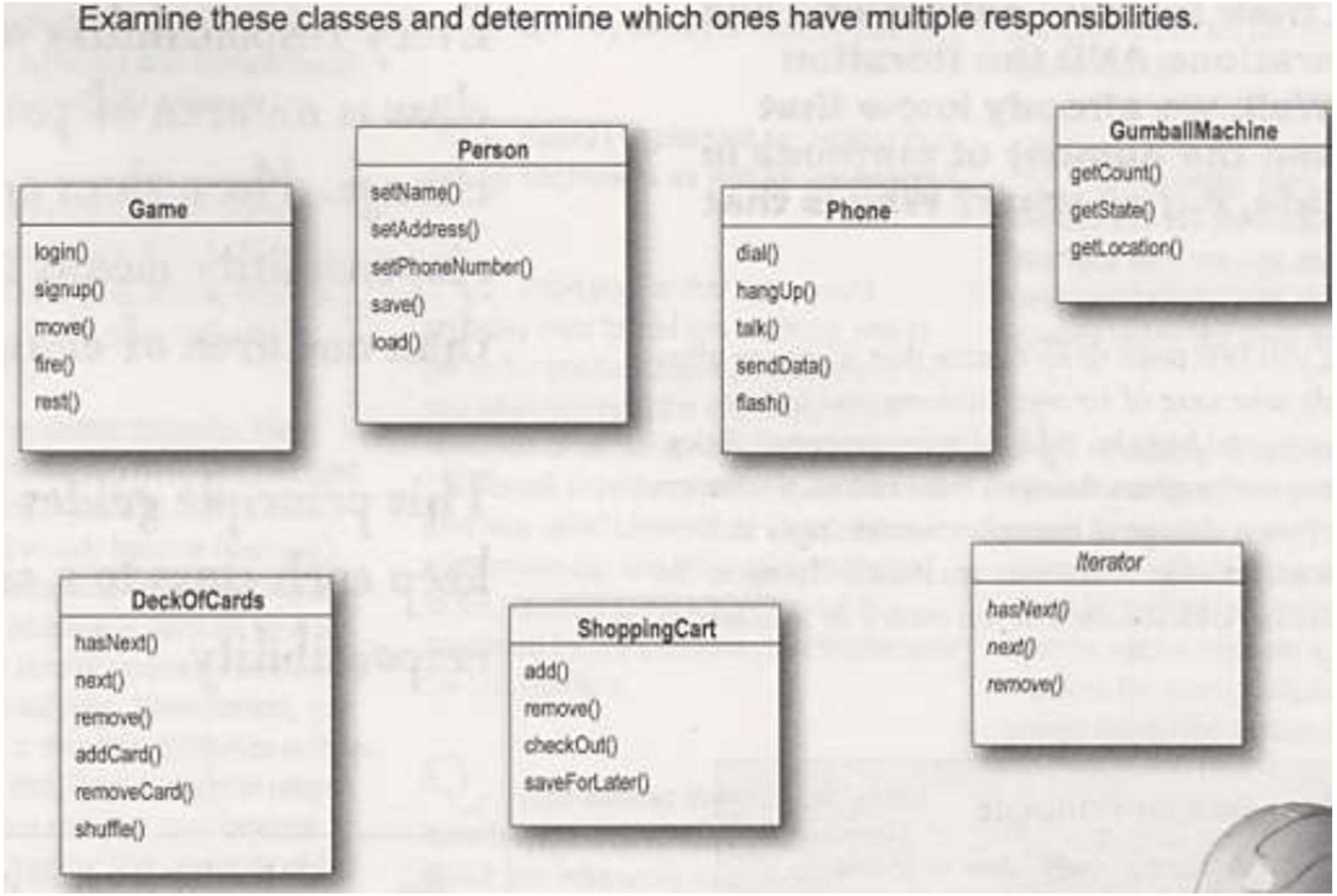
The **`Iterator`** places the task of traversal on the iterator object, not on the aggregate, which simplifies the aggregate interface and implementation, and places the responsibility where it should be.

Design Principle: Single Responsibility

A class should have only one reason to change.

- Every responsibility of a class is an area of potential change. More than one responsibility means more than one area of change.
- This principle guides us to keep each class to a single responsibility.
- We have studied the principle of single responsibility at the module level.
  What is it?

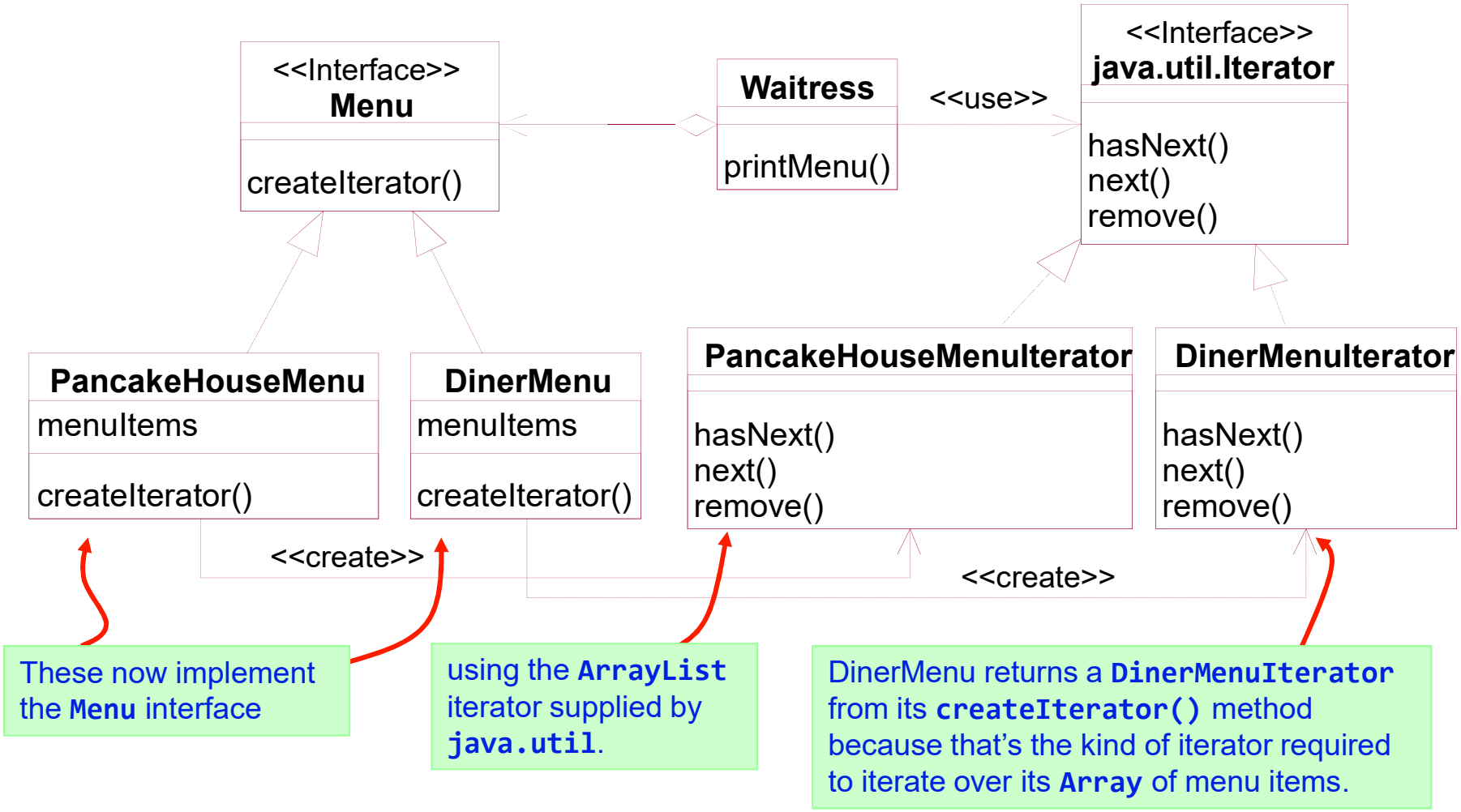
Brain Power


Examine these classes and determine which ones have multiple responsibilities.
Game
login()
signup()
move()
fire()
rest()
Person
setName()
setAddress()
setPhoneNumber()
save()
load()
Phone
dial()
hangUp()
talk()
sendData()
flash()
GumballMachine
getCount()
getState()
getLocation()
DeckOfCards
hasNext()
next()
remove()
addCard()
removeCard()
shuffle()
ShoppingCart
add()
remove()
checkOut()
saveForLater()
Iterator
hasNext()
next()
remove()

The `java.util.Iterator`

- The `java.util.Iterator` interface supports the `Iterator` pattern that we have discussed thus far.
- The Java Collections Framework – provides a set of classes and interfaces, including `ArrayList`, `Vector`, `LinkedList`, `Stack` and `PriorityQueue`.
- All of these classes implement the `Collection` interface, and provide a method `iterator()` to return an instance of the `java.util.iterator` to iterate over the collection.

Using `java.util.Iterator`



DinerMenuIterator implement java.util.Iterator

```
public class DinerMenuIterator implements Iterator {
   MenuItem[] list;
   int position = 0;
   public Object next() { // the same of above }
   public boolean hasNext() { // the same of above }
   public void remove() {
      if (position <= 0) {
         throw new IllegalStateException(
               "You can't remove an item until " +
               "you've done at least one next()");
      }
      if (list[position - 1] != null) {
         for (int i = position - 1; i < (list.length - 1); i++) {
            list[i] = list[i + 1];
         }
         list[list.length - 1] = null;
      }
   }
}
```

Iterator in Java 5

- **Iterators** and **Collections** in Java 5:
  - Added support for iterating over **Collections** so that you don't even have to ask for an iterator!
- Includes a new form of for statement -- **for/in**
  - Lets you iterate over a collection or an array without creating an iterator explicitly.

```
ArrayList items = new ArrayList();
items.add(new MenuItem("Pancakes",
                  "delicious pancakes", true, 1.59);
items.add(new MenuItem("Waffles", "yummy waffles", true, 1.99);
items.add(new MenuItem("Toast", "perfect toast", true, 0.59);

for (MenuItem item : items) {
   System.out.println("Breakfast item: " + item);
}
```

Is the Waitress ready for prime time?

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

   System.out.println("MENU\n-------");
   System.out.println("\nBREAKFAST");   printMenu(pancakeIterator);
   System.out.println("\nLUNCH");   printMenu(dinerIterator);
   System.out.println("\nDINNER");   printMenu(cafeIterator);
}

void printMenu(Iterator iterator) {
   // iterate over the collection and print
}
```

- Whats wrong with this?
  - We have done a good 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.
  - Ideas?

Packaging into an ArrayList

- We could package the menus up into an **`ArrayList`** and then get its **`iterator`** to iterate through each **`Menu`**.

```
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) {
    // no code changes here
  }
}
```

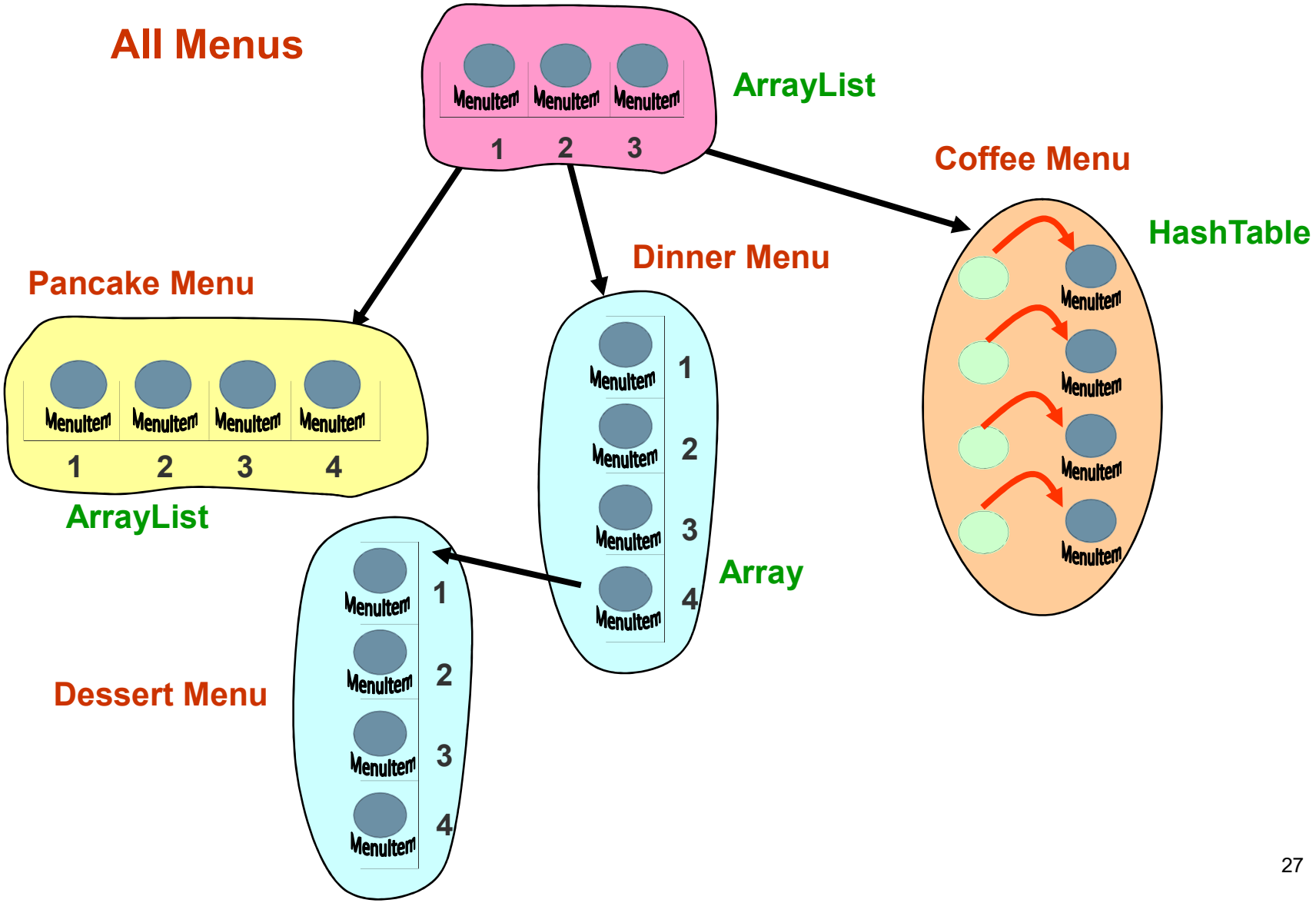
We do loose the name of the menus in this implementation.

Composite Pattern

Just when we thought it was safe …

- The `DinerMenu` wants to add a new `Dessert` "submenu"
- What does that mean?
  - We have to support not only multiple menus, but menus within menus.
- Solutions: we could make the `Dessert` menu an element of the `DinerMenu` collection, but that won't work as it is now implemented.
  - `DessertMenu` will need to be implemented as a collection
  - We can't actually assign a menu to a `MenuItem` array because the types are different.
- Time for a change!

The Desired Menu Structure

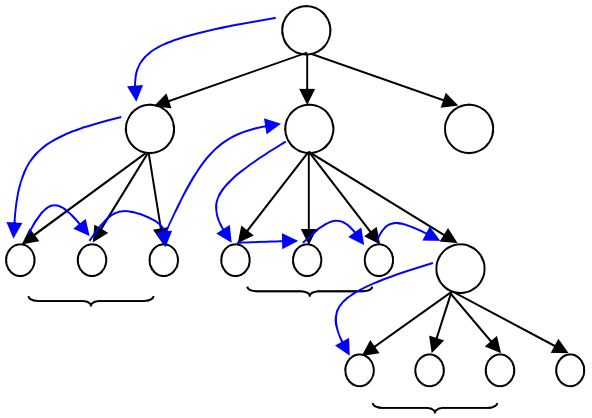


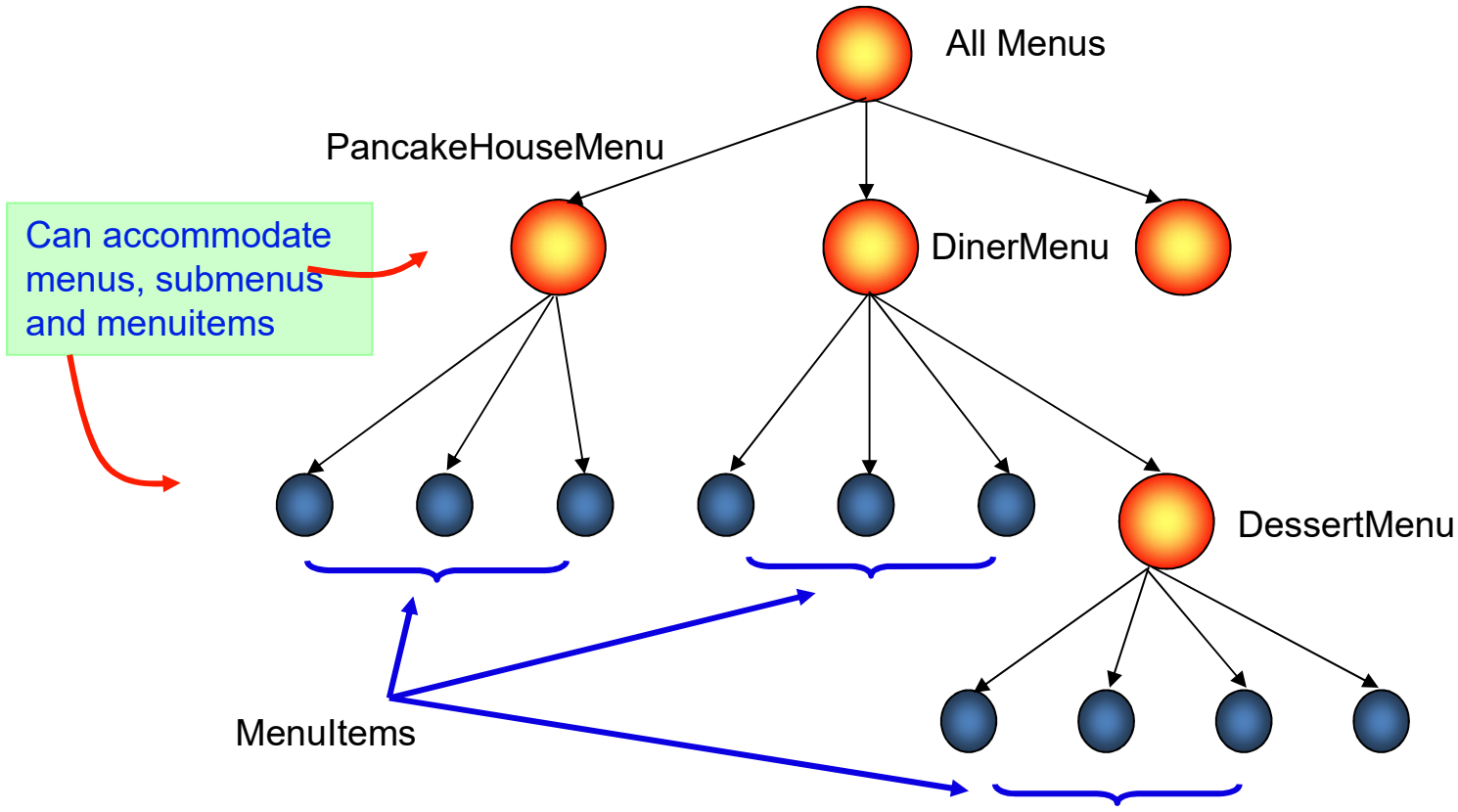
So what do we need?

- 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 need to be able 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 menu.

Possibilities

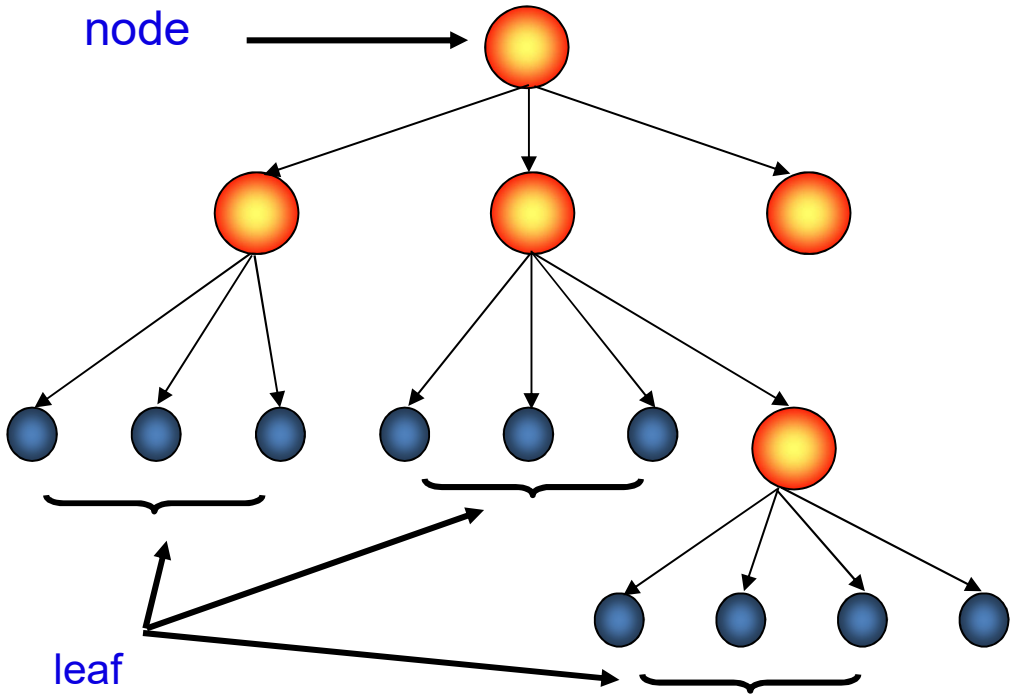
Menus and submenus sub-structure naturally fit into a tree-like structure





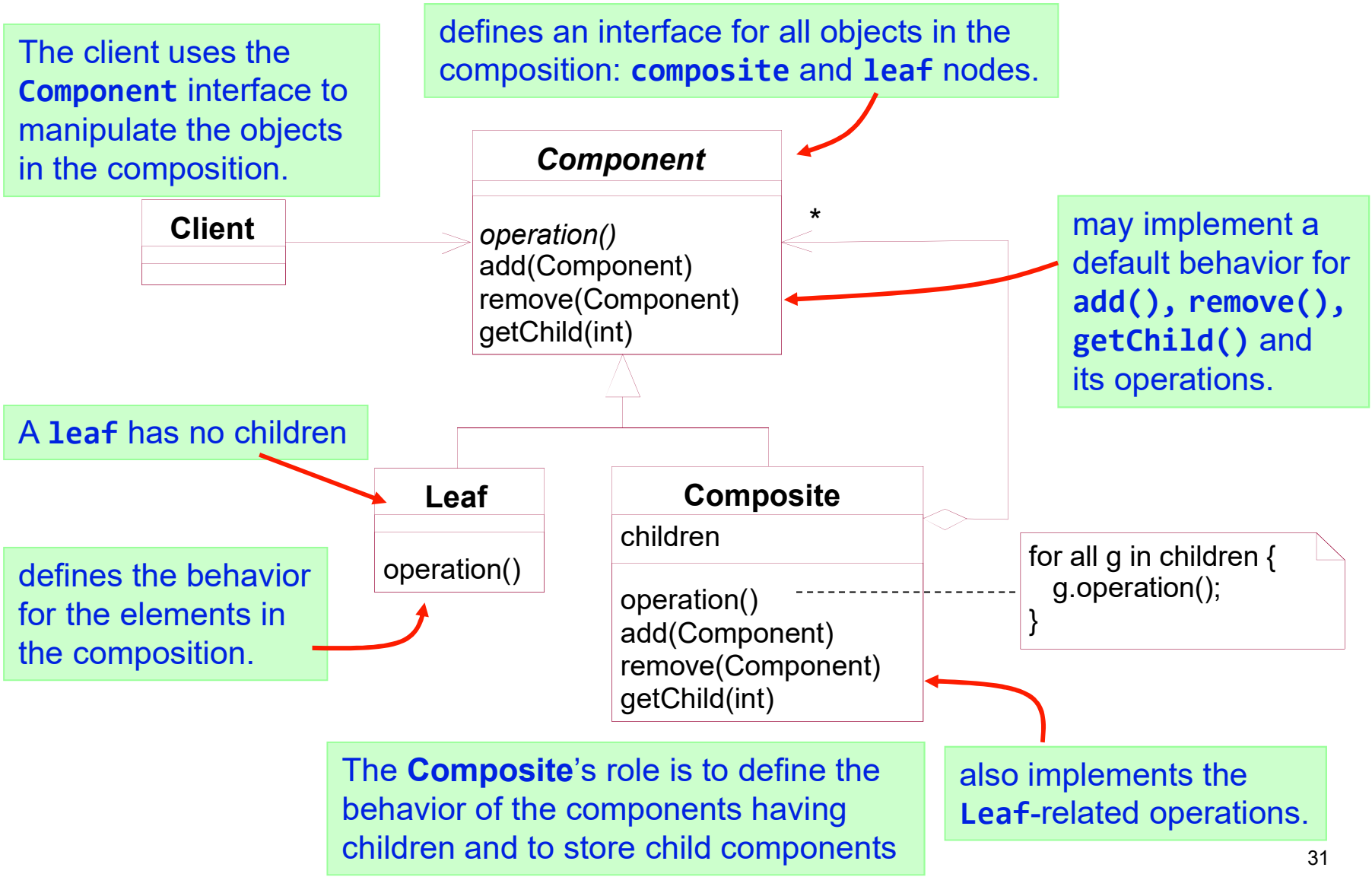
The Composite Pattern Defined

The **Composite Pattern** allows you to compose objects into tree structures to represent whole-part hierarchies. Composite lets clients treat individual objects and composition of objects uniformly.

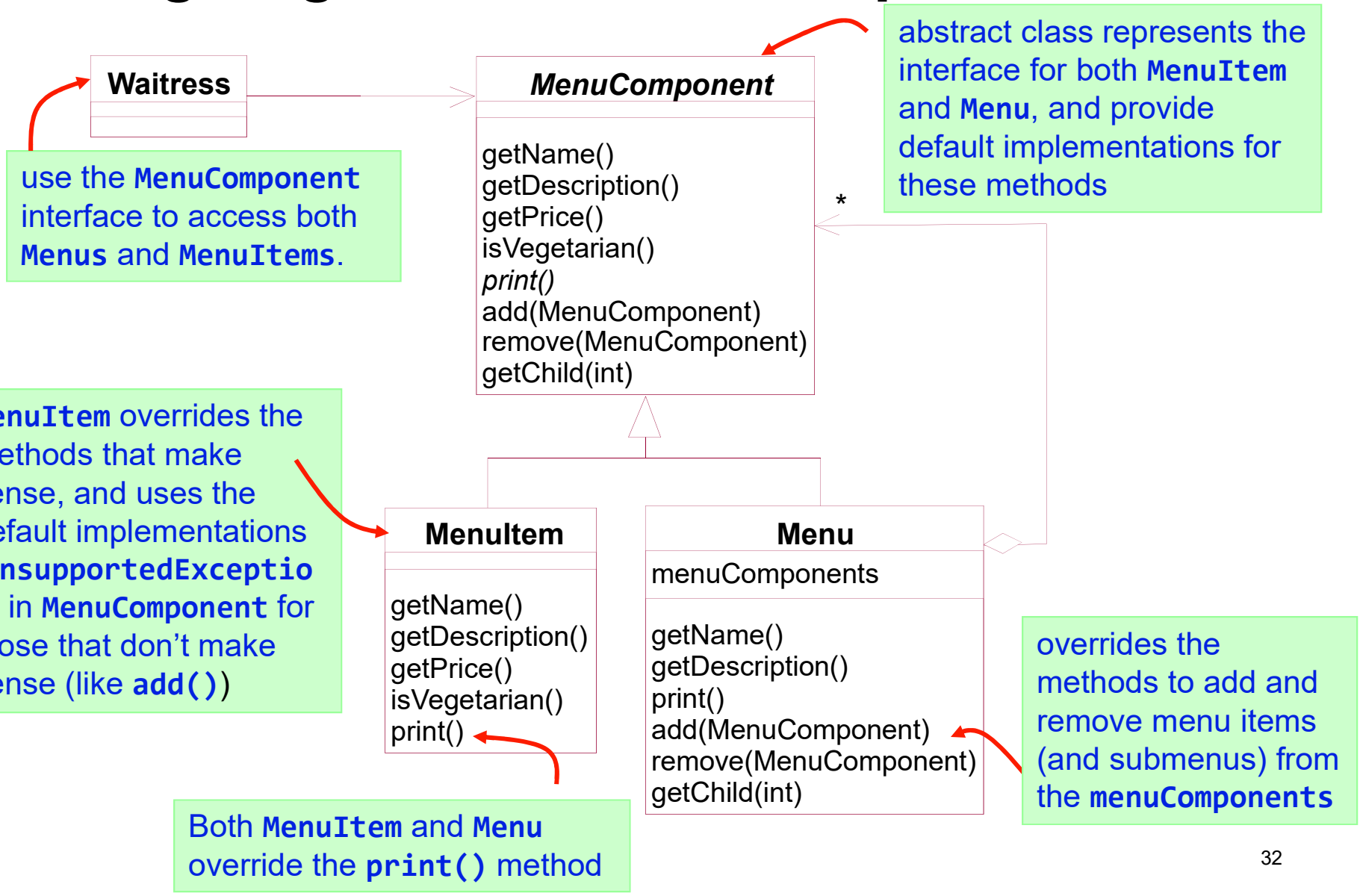


1. We can create arbitrarily complex trees.
2. We can treat them as a whole or as parts
3. Operations can be applied to the whole or the part.

Composite Pattern Structure



Designing Menus with Composite



Implementing the Menu Component

```
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 abstract void print();
}
```

Implementing the 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;
   }
    public String getName() {
      return this.name;
   }
   public String getDescription() {
      return this.description;
   }
   public double getPrice() {
      return this.price;
   }
   public boolean isVegetarian() {
      return this.vegetarian;
   }
```

Implementing the `MenuItem` (cont)

```
    public void print() {
        System.out.print("  " + getName());
        if (isVegetarian()) {
            System.out.print("(v)");
        }
        System.out.println(", " + getPrice());
        System.out.println("     -- " + getDescription());
    }
}
```

Implementing the Menu Composite

```
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 void print() {
      System.out.print("\n" + getName());
      System.out.println(", " + getDescription());
      System.out.println("---------------------");
      Iterator iterator = menuComponents.iterator();
      while (iterator.hasNext()) {
         MenuComponent component = (MenuComponent) iterator.next();
         component.print();
      }
   }
}
```

Implementing the Menu Composite

```
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) {
      return menuComponents.get(i);
   }
   public void print() {
      System.out.print("\n" + getName());
      System.out.println(", " + getDescription());
      System.out.println("---------------------");
      for (MenuComponent component : menuComponents) {
         component.print();
      }
   }
}
```

An Observation - Violating SRP?

- First we say: One Class, One Responsibility.
- Composite is a pattern with two responsibilities in one class: manages a hierarchy and performs operations related to MenuItems.
- In the **Composite pattern** we trade the Single Responsibility design principle for transparency.
  - The client can treat both composites and leaf nodes uniformly
- We lose: safety because the client may try to do something inappropriate or meaningless.
  - We could take the design in the other direction and separate out the responsibilities into interfaces.
  - We would loose transparency in this case and our code would have to use conditionals and the `instanceOf` operator.

Flashback to the `Iterator` – A `Composite Iterator`

- To implement a **Composite Iterator** we add the `createIterator()` method in every component.
  - We add a `createIterator()` method to the abstract `MenuComponent`. Means that calling `createIterator()` on a composite should apply to all children of the composite.
- Implementations in `Menu` and `MenuItem`:

```
public class Menu extends MenuComponent {
  // other code does not change
  public Iterator createIterator() {
    return new CompositeIterator(menuComponents.iterator());
  }
}

public class MenuItem extends MenuComponent {
  // other code does not change
  public Iterator createIterator() {
    return new NullIterator();
  }
}
```

`CompositeIterator` knows how to iterate over any composite.

`NullIterator` coming up….

The Composite Iterator - A Serious Iterator

Its job is to iterate over the **MenuItems** in the component, and of all the child **Menus** (and child child Menus, …) are included

```
public class CompositeIterator implements Iterator {
  Stack stack = new Stack();
  public CompositeIterator(Iterator iterator) {
    stack.push(iterator);
  }

  public Object next() {
    if (hasNext()) {
      Iterator iterator = (Iterator) stack.peek();
      MenuComponent component =
             (MenuComponent) iterator.next();
      if (component instanceof Menu) {
        stack.push(component.createIterator());
      }
      return component;
    } else { return null; }
  }
```

implementing the **java.util.Iterator** interface

The iterator of the top level composite we're going to iterate over is passed in. We throw that in a stack data structure.

make sure there is one by calling **hasNext()**

If there is a next element, we get the current iterator off the stack and get its next element.

If that element is a menu, we have another composite that needs to be included in the iteration, so we throw it on the stack.
In either case, we return the component.

The Composite Iterator (con't)

```
public boolean hasNext() {
    if (stack.empty()) { return false; }
    else {
      Iterator iterator = (Iterator) stack.peek();
      if (!iterator.hasNext()) {
        stack.pop();
        return hasNext();
      } else { return true; }
    }
  }

  public void remove() {
    throw new UnsupportedOperationException();
  }
}
```

To see if there is a next element, we check to see if the **`stack`** is empty;

Otherwise, we get the **`iterator`** off the top of the **`stack`** and see if it has a next element. If it does not we pop it off the stack and call **`hasNext()`** recursively.

We are not supporting remove, just traversal.

The Null Iterator

- A **`MenuItem`** has nothing to iterate over. So how do we handle the **`createIterator()`** implementation?
  - Choice 1: return **`null`**, and have conditional code in the client to check if a **`null`** was returned or not.
  - Choice 2: Return an iterator that always returns a **`false`** when **`hasNext()`** is called. Create an iterator that is a no-op.

```
public class NullIterator implements Iterator {
   public Object next() {
      return null;
   }
   public boolean hasNext() {
      return false;
   }
   public void remove() {
      throw new UnsupportedOperationException();
   }
}
```

Give me the vegetarian menu…

```
public class Waitress {
  MenuComponent allMenus;
  public Waitress(MenuComponent allMenus) {
    this.allMenus = allMenus;
  }
  public void printMenu() {
    allMenus.print();
  }
  public void printVegetarianMenu() {
    Iterator iterator = allMenus.createIterator();
    System.out.println("\nVEGETARIAN MENU\n----");
    while (iterator.hasNext()) {
      MenuComponent menuComponent = (MenuComponent)iterator.next();
      try {
        if (menuComponent.isVegetarian()) {
          menuComponent.print();
        }
      } catch (UnsupportedOperationException e) {}
    }
  }
}
```

takes the **allMenu**'s composite and gets its iterator. This will be our **CompositeIterator**.

print() is only called on **MenuItems**, never on composites. **Can you see why?**

We implemented **isVegetarian()** on the **Menus** to always throw an exception. If that happens we catch the exception, but continue with the iteration….

Exceptions and Program Logic….

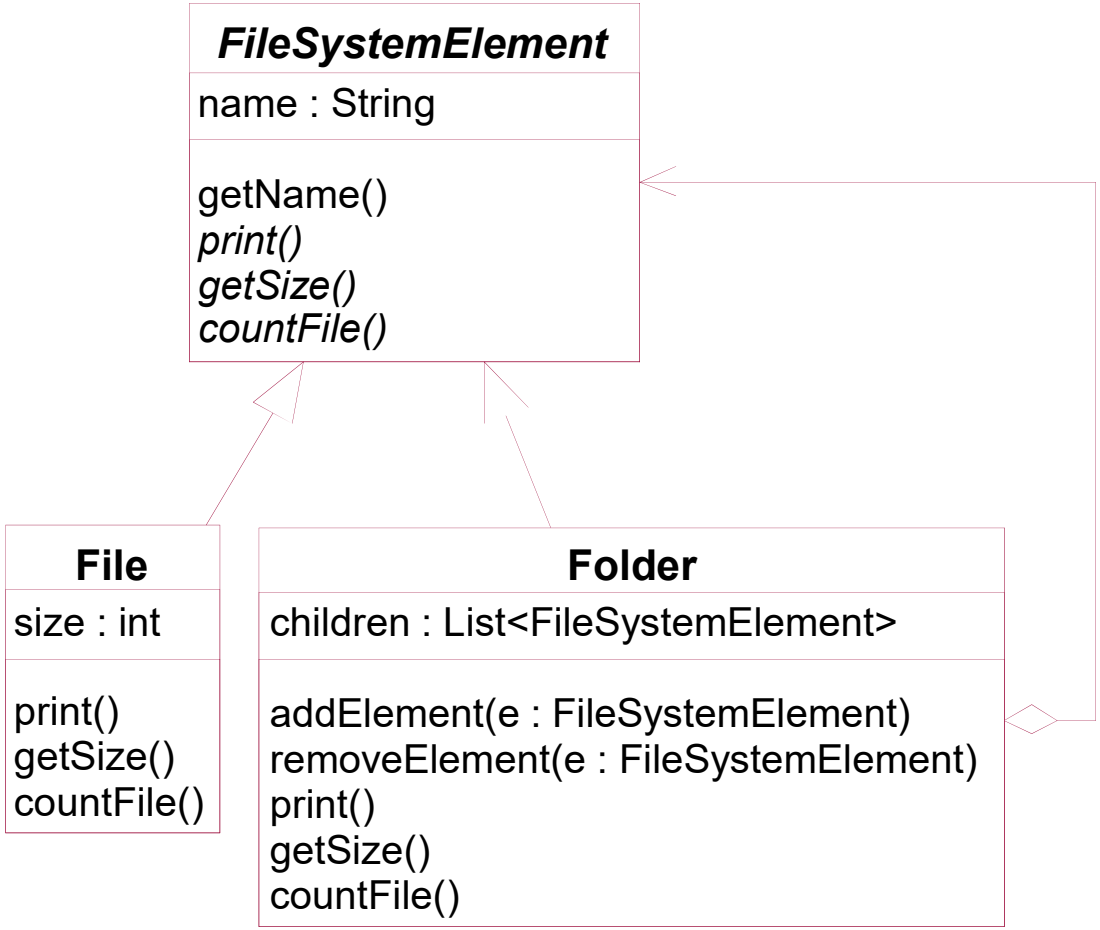
- In general, using ***exceptions*** to implement program logic is a big no, no! Yet that's exactly what we are doing with the `isVegetarian()` method.
- We could have checked the runtime type of the `Menu` component with `instanceOf` to make sure it's a `MenuItem` before making the call to `isVegetarian()`.
  But in the process we would lose transparency.
- We could also change `isVegetarian()` in the `Menus` to return a `false`. This provides a simple solution and we keep our transparency.
- We choose to go for clarity. We want to communicate that for the `Menus` this really is an unsupported operation.
  It also allows for someone to come along and implement a reasonable `isVegetarian()` method for `Menu` and have it work with existing code.

Other Composite Pattern Examples

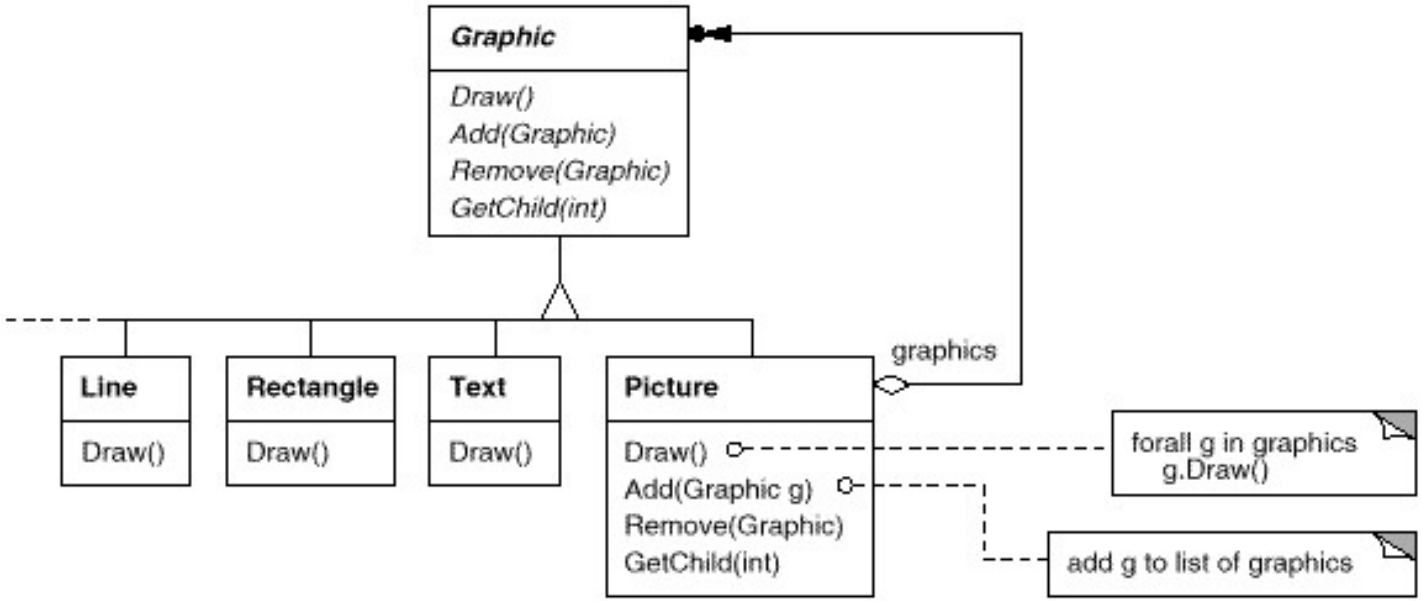
- File System
- Graphic hierarchy
- Arithmetic Expression Tree
- List Structure
- Tree Structure
- XML Document

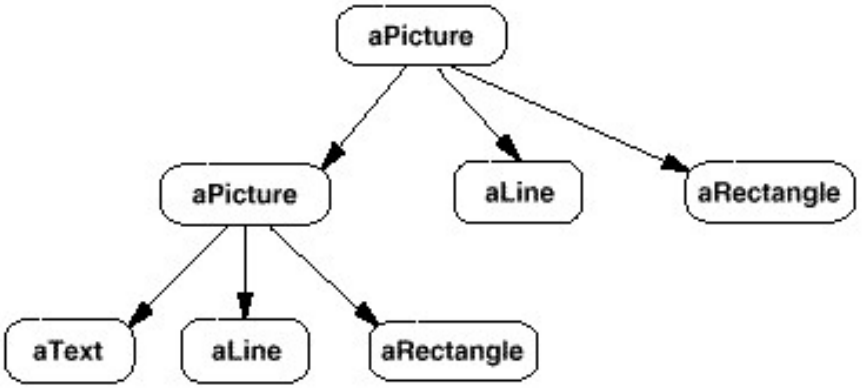
File System

- File và folder là các thành phần của một hệ thống quản lý tập tin.
  - File biểu diễn dữ liệu cần lưu trữ.
  - Folder là ngăn chứa các tập tin. Trong mỗi folder có thể chứa các folder con khác.

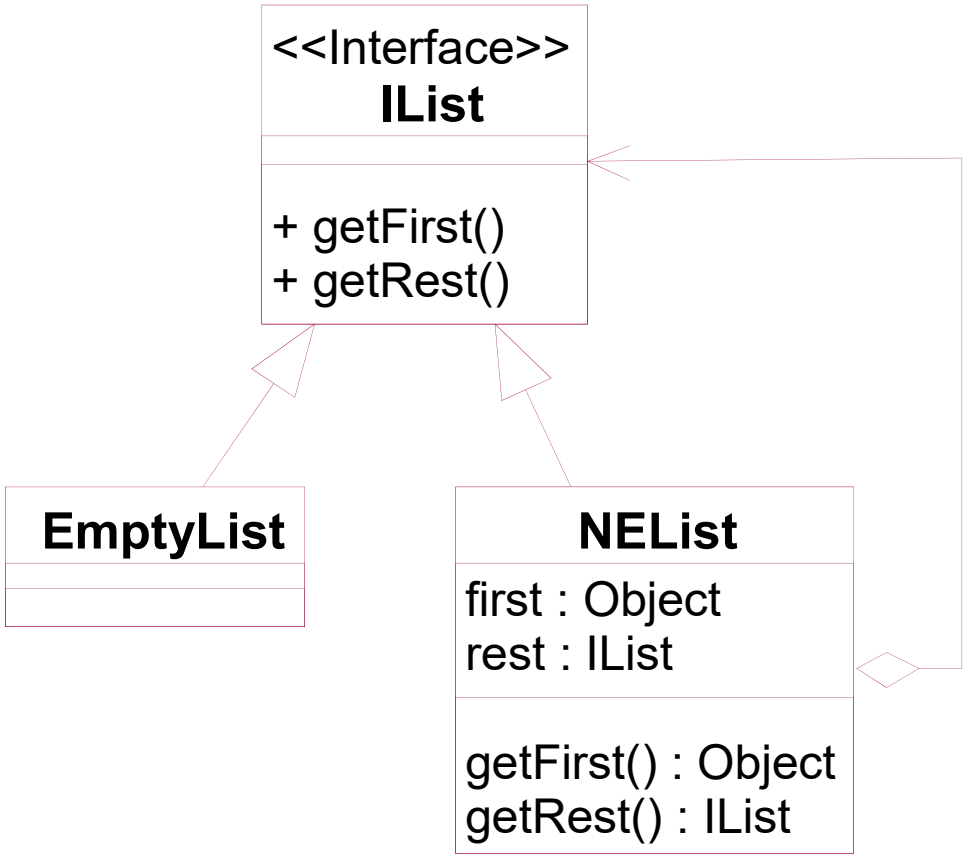


Graphic hierarchy



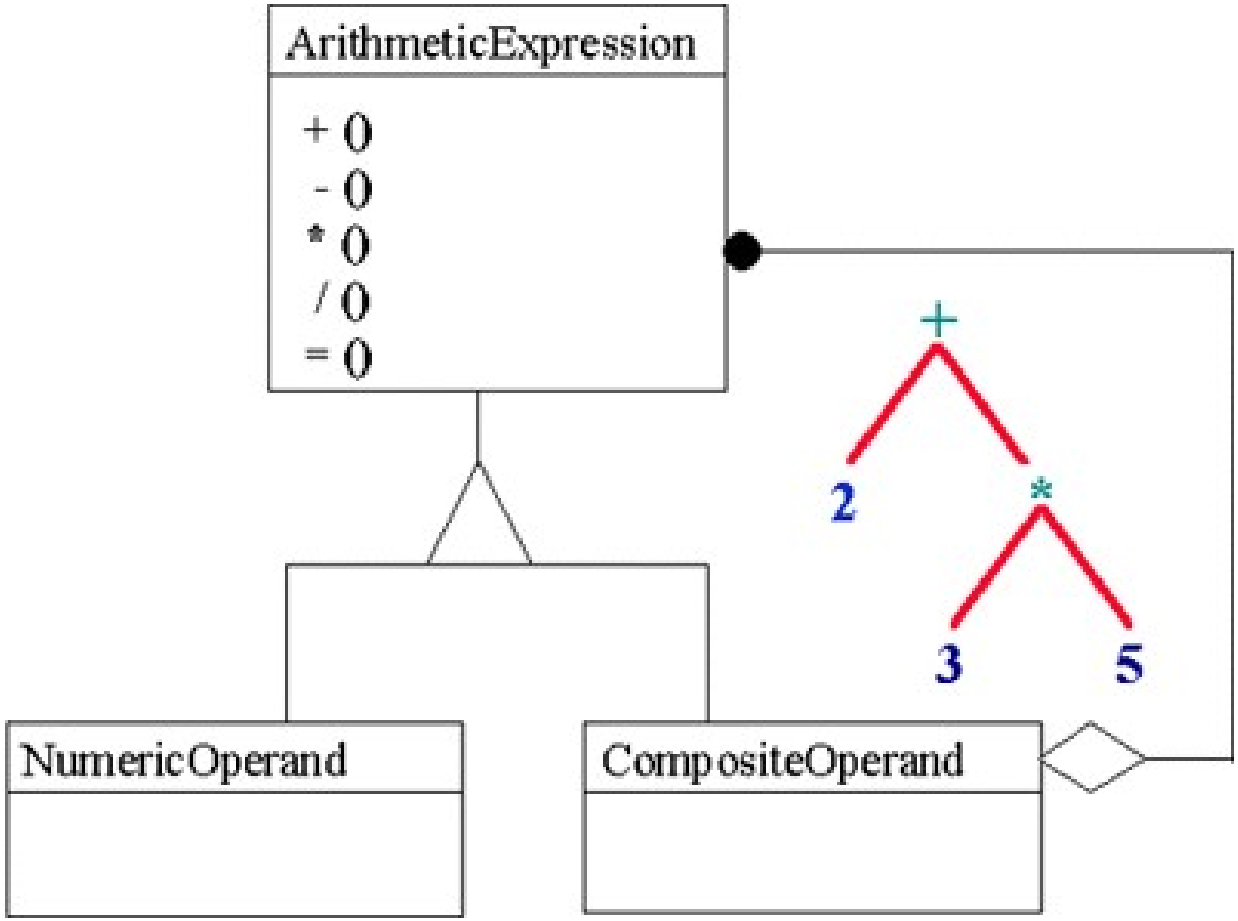


List Structure



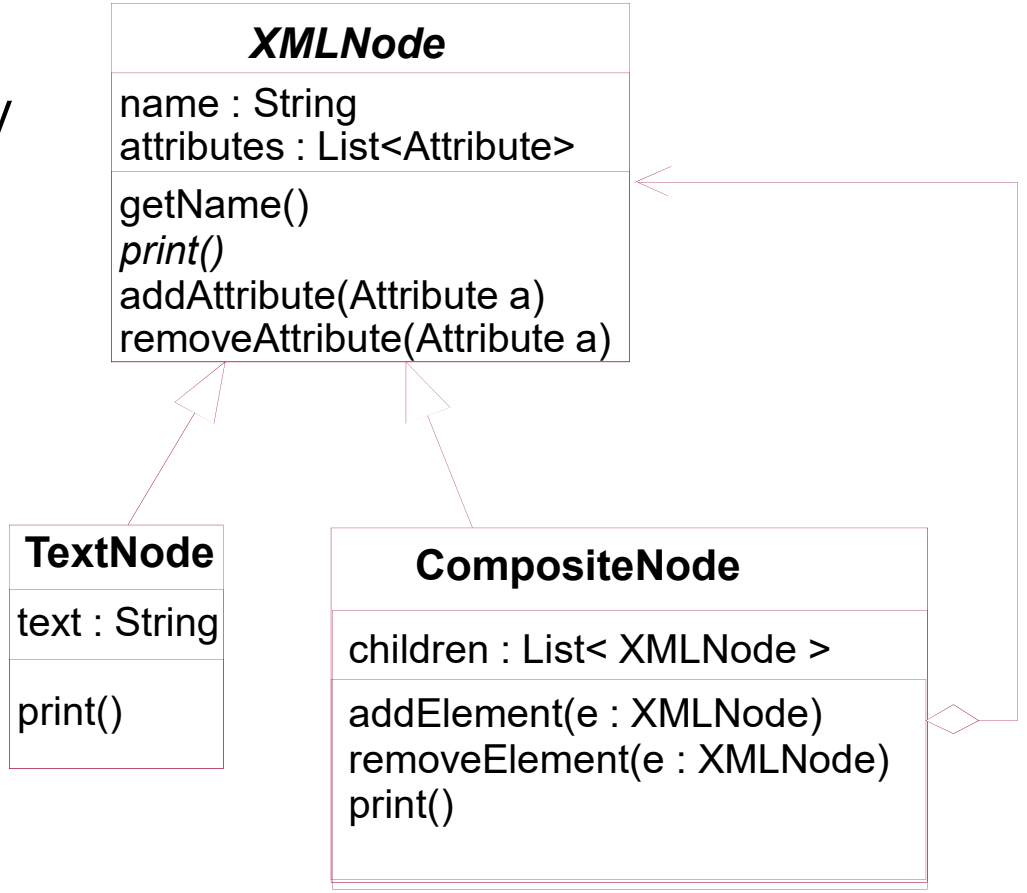
Binary Tree

Arithmetic Expression hierarchy



XML Document

- Tài liệu XML là một tài liệu văn bản có cấu trúc dạng cây phân cấp các thành phần. Mỗi thành phần có tên (**name**) và có danh sách các thuộc tính (**attributes**). Có 2 loại thành phần:
  - Thành phần văn bản (**TextNode**) chứa nội dung văn bản (**text**), và
  - thành phần tổ hợp (**CompositeNode**) có thể chứa một danh sách các thành phần khác.
- Một tài liệu XML sẽ bắt đầu từ một thành phần gọi là gốc.



Summary (1/2)

- An iterator allow access to an aggregate's elements without exposing its internal structure.
- An iterator takes the job of iterating over an aggregate and encapsulates it in another object.
- When using an iterator, we relieve the aggregate of the responsibility of supporting operations for traversing its data.
- An iterator provides a common interface for traversing the items of an aggregate, allowing us to use polymorphism when writing code that makes use of the items of the aggregate.
- We should strive to assign only one responsibility to each class.

Summary (2/2)

- The Composite Pattern provides a structure to hold both the 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 the Composite. You need to balance transparency and safety with your needs.