Iterator and Composite

CSCI 4448/5448: Object-Oriented Analysis & Design

Lecture 23

Acknowledgement & Materials Copyright

- I’d like to start by acknowledging Dr. Ken Anderson
- Ken is a Professor and the Chair of the Department of Computer Science
- Ken taught OOAD on several occasions, and has graciously allowed me to use his copyrighted material for this instance of the class
- Although I will modify the materials to update and personalize this class, the original materials this class is based on are all copyrighted © Kenneth M. Anderson; the materials are used with his consent; and this use in no way challenges his copyright

Before we start: OO in Go & JavaScript

- Neither Go (Golang) or JavaScript (inc. TypeScript or Node.JS) are strictly OO languages…
- But they are clearly very useful and support some OO concepts
- And many students want to use them for the Semester Project (5/6/7), which is perfectly okay!
- A little searching will give you some good support in looking at how the languages can get some OO designs in place
- Go
  - OO Concepts https://www.toptal.com/go/golang-oop-tutorial
  - OO Patterns https://refactoring.guru/design-patterns/go
- Javascript
  - OO Concepts https://www.freecodecamp.org/news/how-javascript-implements-oop/
  - OO Patterns https://indepth.dev/posts/1495/js-design-patterns

Example in Head First

- We're up to Chapter 9 in Head First Design Patterns…
- The book looks at managing menus for restaurants
- There are two existing versions of sets of MenuItem objects
- A MenuItem is an object with a name, description, price, and a boolean for vegetarian items

```
m = new MenuItem(name, desc, veg, price);
```

Example - ArrayList

- One collection of MenuItems is in an ArrayList
- For ArrayList, the MenuItem object uses ArrayList.add() and .get() methods (along with .size() to know how many items there are) to insert and retrieve elements from the ArrayList

```
menuItems = new ArrayList<MenuItem>();
menuItems.add(menuItem);
for (int i=0; i<menuItems.size(); i++) {
        m = menuItems.get(i);
        System.out.println(m.getPrice());
}
```

Example - Array

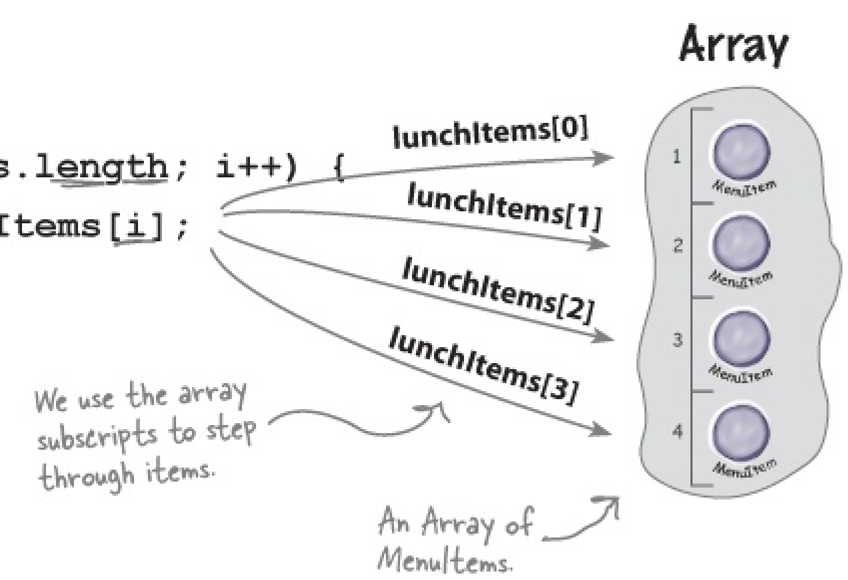
- One collection of MenuItem objects is in an Array
- For Array, the MenuItem object is assigned to slots in the Array, and the size of the Array is controlled in the code

```
MenuItem[] menuItems;
menuItems[i] = menuItem;
for (int i=0; i<menuItems.length; i++) {
    m = menuItems[i];
    System.out.println(m.getPrice());
}
```

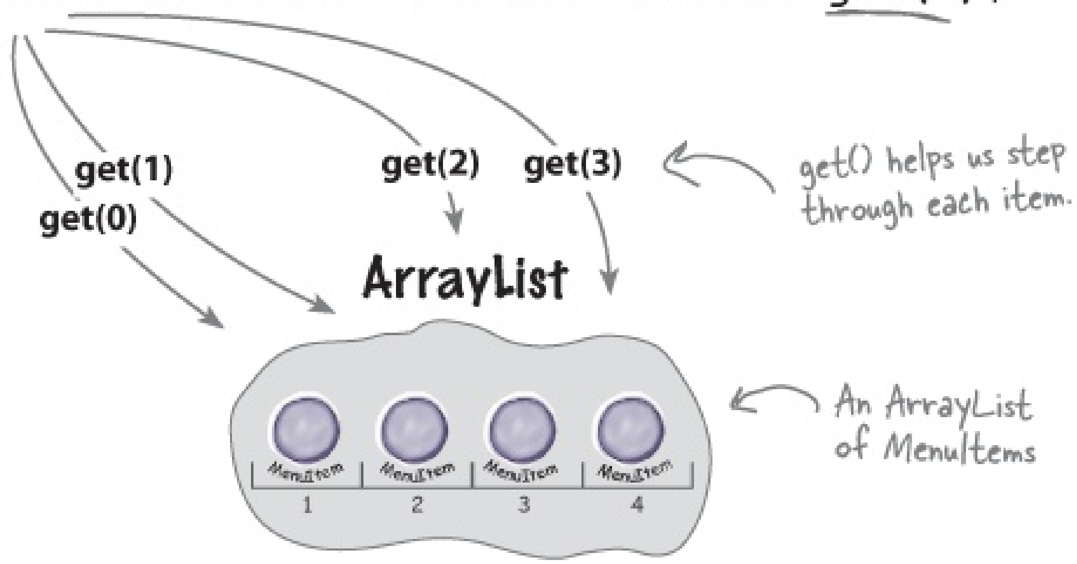
The Problem

- Someone trying to use the different MenuItem collections (ArrayList and Array) has to code them differently, even though the contained items are the same
- Ideally, we want to encapsulate the iterations

```
for (int i = 0; i < lunchItems.length; i++) {
    MenuItem menuItem = lunchItems[i];
}
```

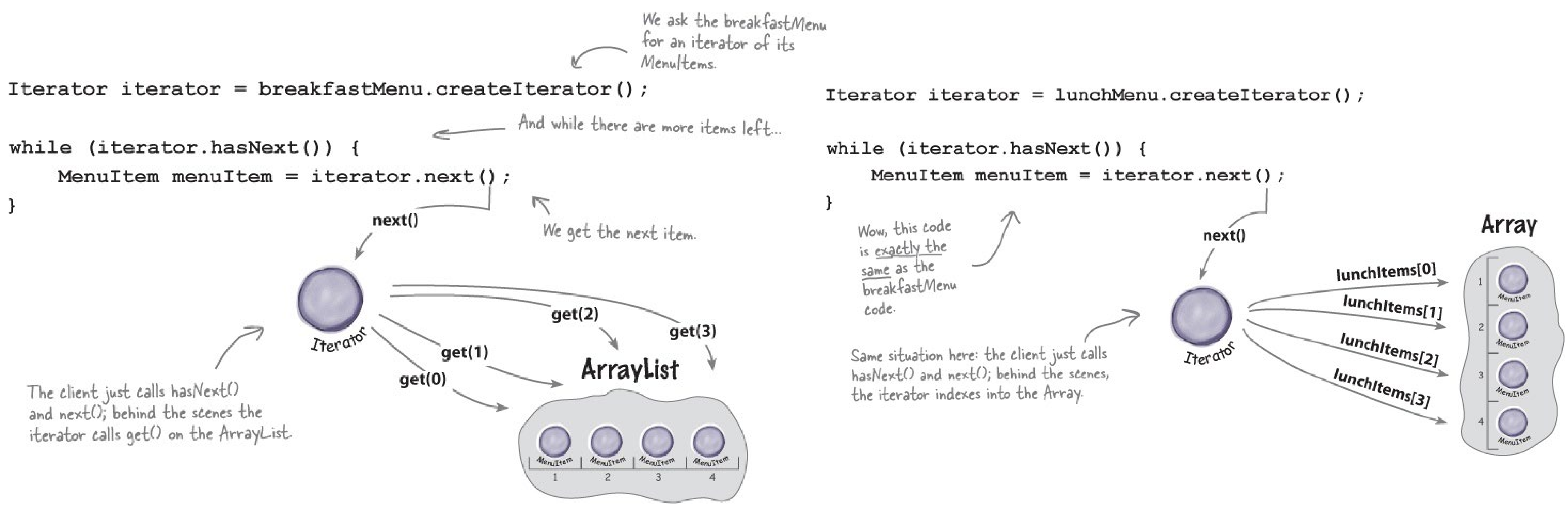



```
for (int i = 0; i < breakfastItems.size(); i++) {
    MenuItem menuItem = breakfastItems.get(i);
}
```



The Solution – Make an Iterator Object

- Create an object that iterates any iterable item in a standard way



The Solution – Using an Iterator Object

```
public class Waitress {
    PancakeHouseMenu pancakeHouseMenu;
    DinerMenu dinerMenu;

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

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

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

    private void printMenu(Iterator iterator) {
        while (iterator.hasNext()) {
            MenuItem menuItem = iterator.next();
            System.out.print(menuItem.getName() + ", ");
            System.out.print(menuItem.getPrice() + " -- ");
            System.out.println(menuItem.getDescription());
        }
    }

    // other methods here
}
```

In the constructor the Waitress takes the two menus.

The printMenu() method now creates two iterators, one for each menu.

And then calls the overloaded printMenu() with each iterator.

Test if there are any more items.

Get the next item.

The overloaded printMenu() method uses the Iterator to step through the menu items and print them.

Use the item to get name, price, and description and print them.

Note that we're down to one loop.

Extending the Solution

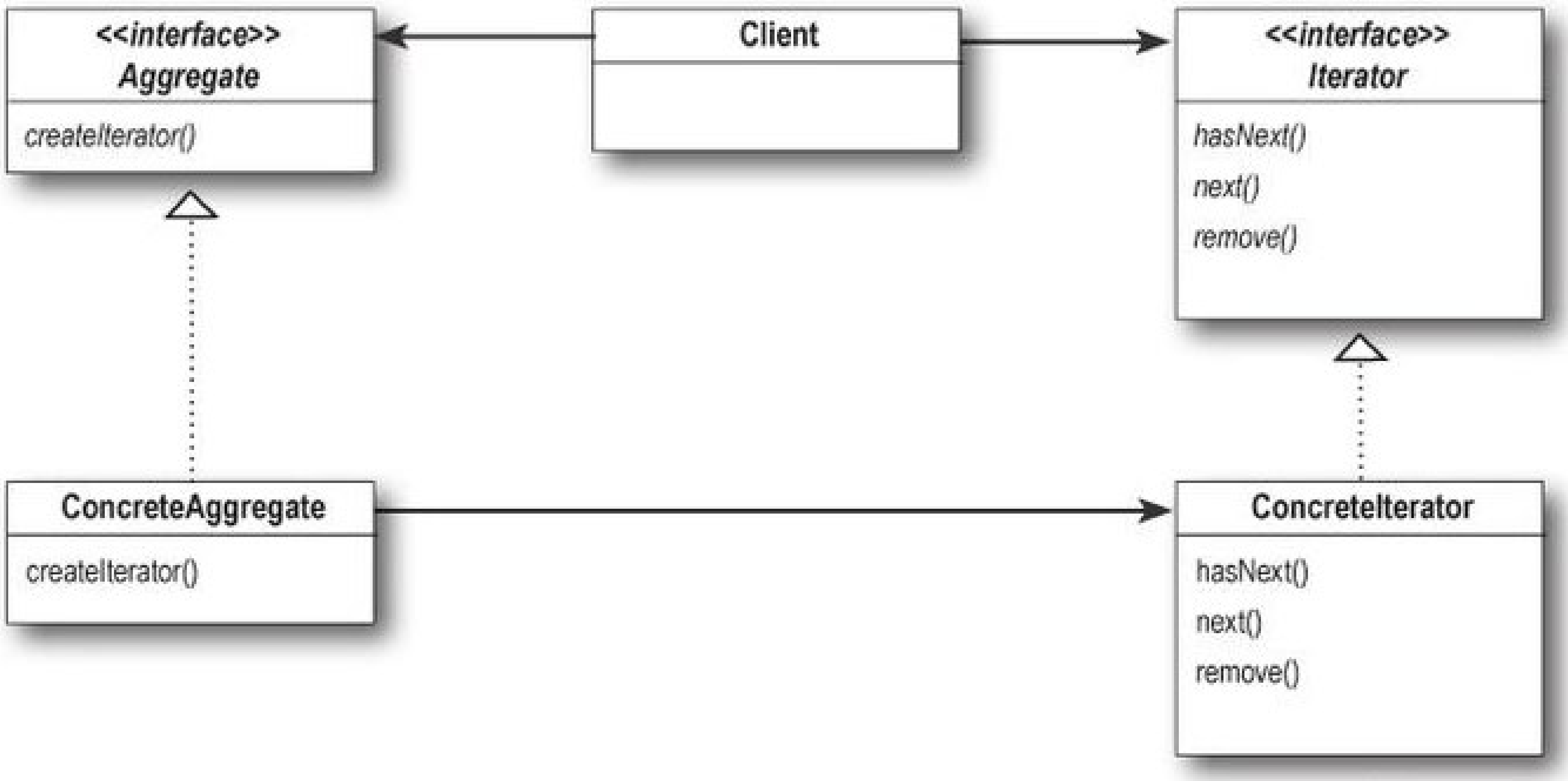
- Later, we find we have to add another set of MenuItem objects, this set represented by a HashTable
- Surprisingly, even though a HashTable is a fairly complex collection object, it supports iterator(), so we can pretty easily add this new collection in a similar fashion
- Note that HashTable “indirectly” supports Iterator
  - This is because HashTables actually have two collections: keys and values
  - You have to get the values before you can get the iterator for them

- Java Collections include iterator() – Vector, LinkedList, Stack, PriorityQueue, etc.

Iterator Pattern

- Intent: Generally, decouple algorithms from the format of the containers (as possible), and allow for traversing a container's (or aggregate object's) elements
- Problem: Elements of an aggregate object need to be traversed without exposing underlying implementation
- Solution: Provide a separate iterator object that encapsulates access and traverse of an aggregate object, and allows traversal without exposing the aggregate structure
- Use: Clients create an iterator, and use it to loop through each member of an aggregate object's collection of objects

Iterator Pattern Structure



- ConcreteAggregate has a collection of objects and implements the code to return an Iterator object
- The ConcreteIterator manages the current position of iteration

Iterators in Java (java.util.Iterator)

```
List<String> list = new ArrayList<String>();
// add some strings
Iterator it = list.iterator();
while(it.hasNext()){
     String s = it.next();
}
```

Standard Java Iterator (for Iterable objects) also has methods:

- remove() = removes last object retrieved by next())
- forEachRemaining(action) = which performs an action on each remaining object

https://docs.oracle.com/javase/8/docs/api/java/util/Iterator.html

Iterators in Python

- Python has built in iterators in many collection classes
- In Python 3, Objects that support the __iter__ and __next__ dunder methods automatically work with for-in loops
- Internally, a for-in actually runs a simple while loop:
- The iteration object is retrieved by calling its __iter__ method
- After that, the loop repeatedly calls the iterator object's __next__ method to retrieve values from it

https://dbader.org/blog/python-iterators

```
# Sample built-in iterators

# Iterating over a list
print("List Iteration")
l = ["geeks", "for", "geeks"]
for i in l:
   print(i)

# Iterating over a tuple (immutable)
print("\nTuple Iteration")
t = ("geeks", "for", "geeks")
for i in t:
   print(i)

# Iterating over a String
print("\nString Iteration")
s = "Geeks"
for i in s :
   print(i)

# Iterating over Dictionary
print("\nDictionary Iteration")
d = dict()
d['xyz'] = 123
d['abc'] = 345
for i in d :
   print("%s  %d" %(i, d[i]))
```

https://www.geeksforgeeks.org/iterators-in-python/

Single Responsibility

- What if we allowed aggregates to handle both
  - Implementations and related operation for internal collections AND
  - Iteration methods

- If we do, the class has two different possible reasons to change

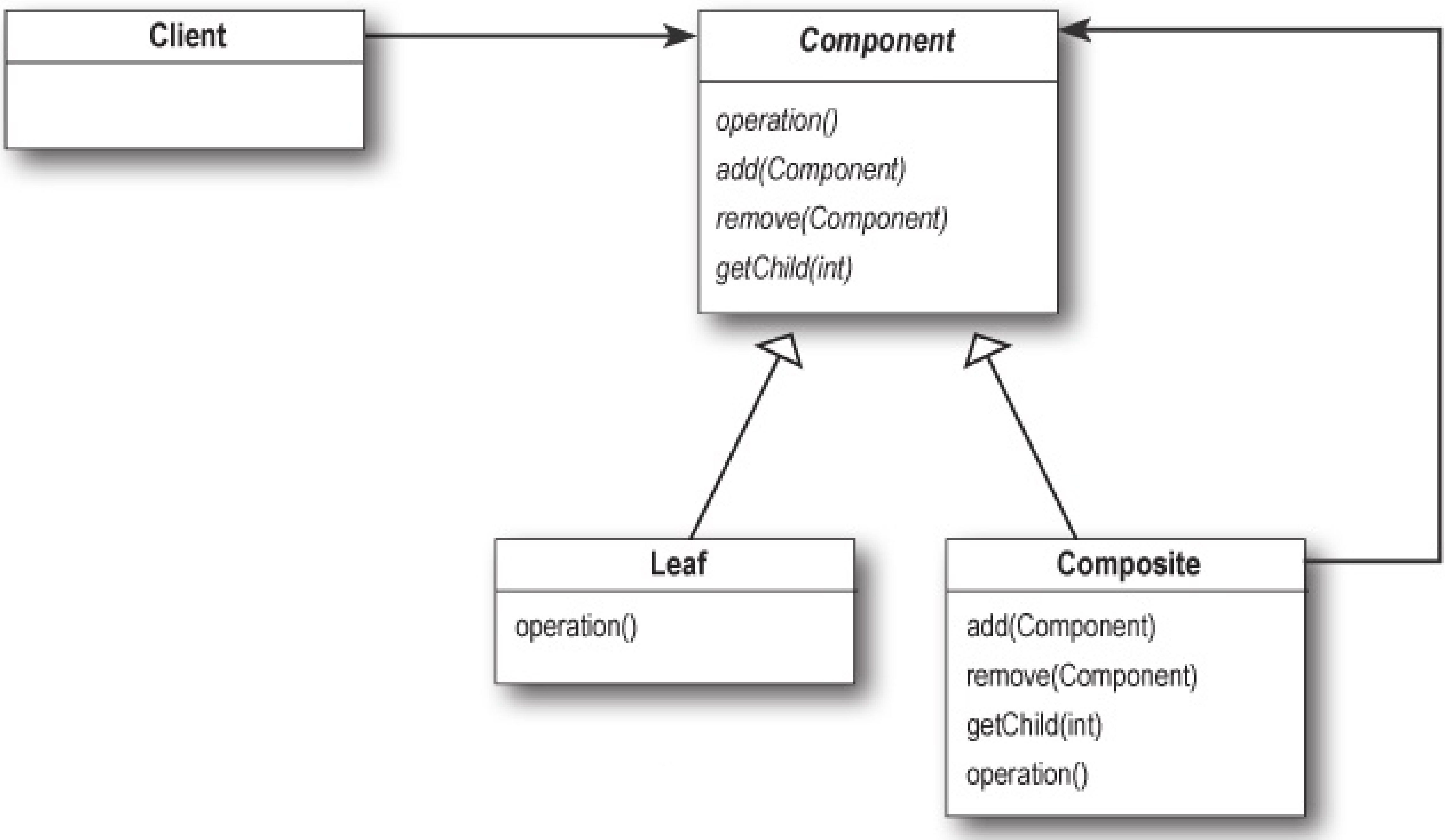
- The S in the SOLID principle is:
- The Single Responsibility Principle
  - Classes should have only one reason to change
  - Really, classes should be cohesive – supporting a single purpose

- Every responsibility we add is another area of potential change, if we have to modify code, problems may arise

Composite Pattern

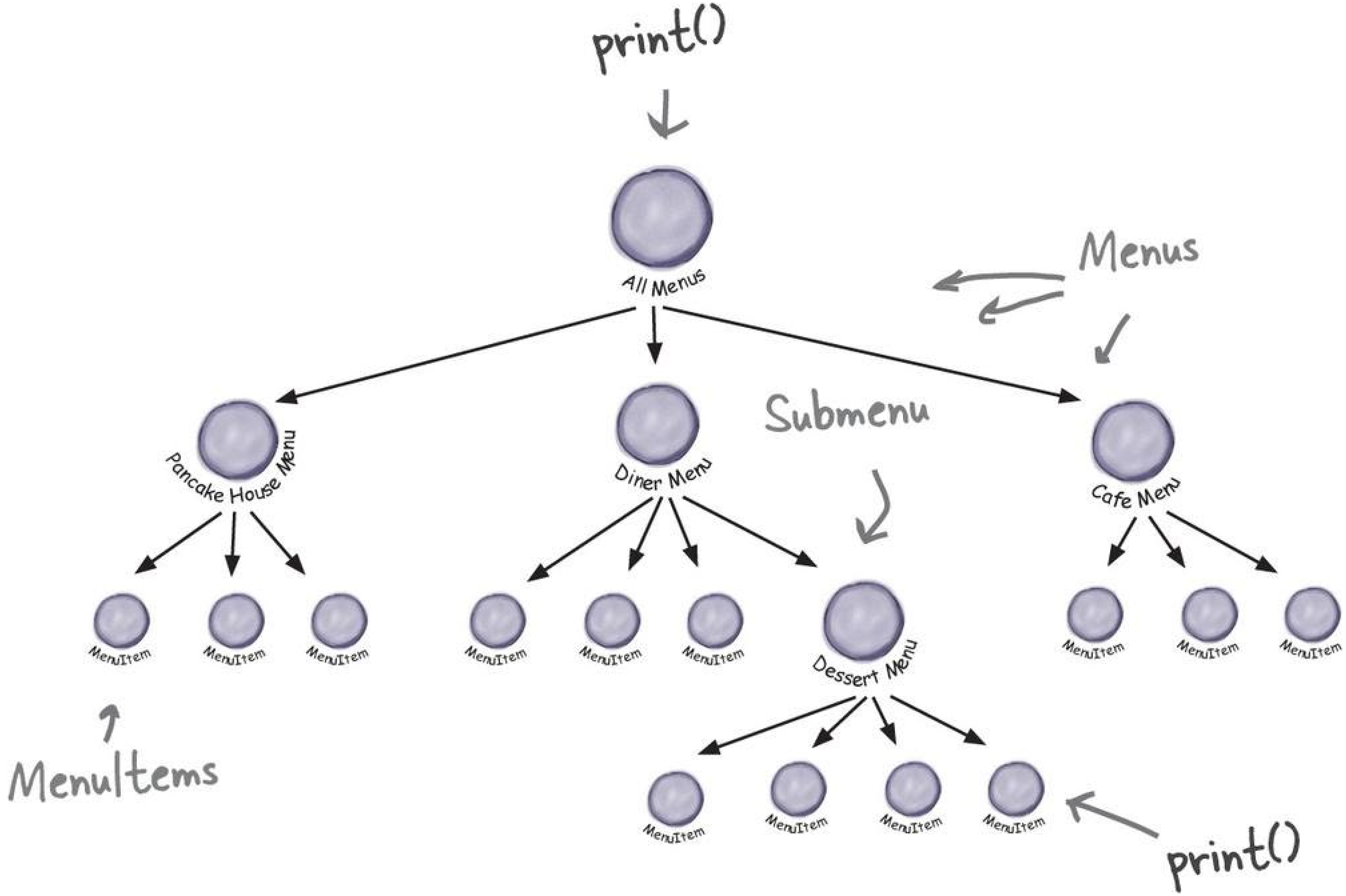
- Intent: Allows composing objects into tree structures, treating individual and composed objects the same way
- Problem: Represent a part-whole hierarchy that allows uniform treatment of parts or whole object structures
- Solution: Provide one interface for both leaf (i.e. part) and composite (whole) objects
- Use: Client creates a composite object which can have 1 or more child objects (either leaf or other composite objects). The composite object provides methods for adding, removing, and getting child objects.

Composite Pattern Structure



Using Composite

- Operations can be applied to the whole or the parts



Composites in Java

```java
import java.util.ArrayList;
import java.util.List;
public class Employee {
   private String name;
   private String dept;
   private int salary;
   private List<Employee> subordinates;
   // constructor
   public Employee(String name,String dept, int sal) {
      this.name = name;
      this.dept = dept;
      this.salary = sal;
      subordinates = new ArrayList<Employee>();
   }
   public void add(Employee e) { subordinates.add(e); }
   public void remove(Employee e) { subordinates.remove(e); }
   public List<Employee> getSubordinates(){ return subordinates;}
   public String toString(){
      return ("Name : " + name + ", dept : " + dept + ", salary :" + salary+" ]");
   }
}
```

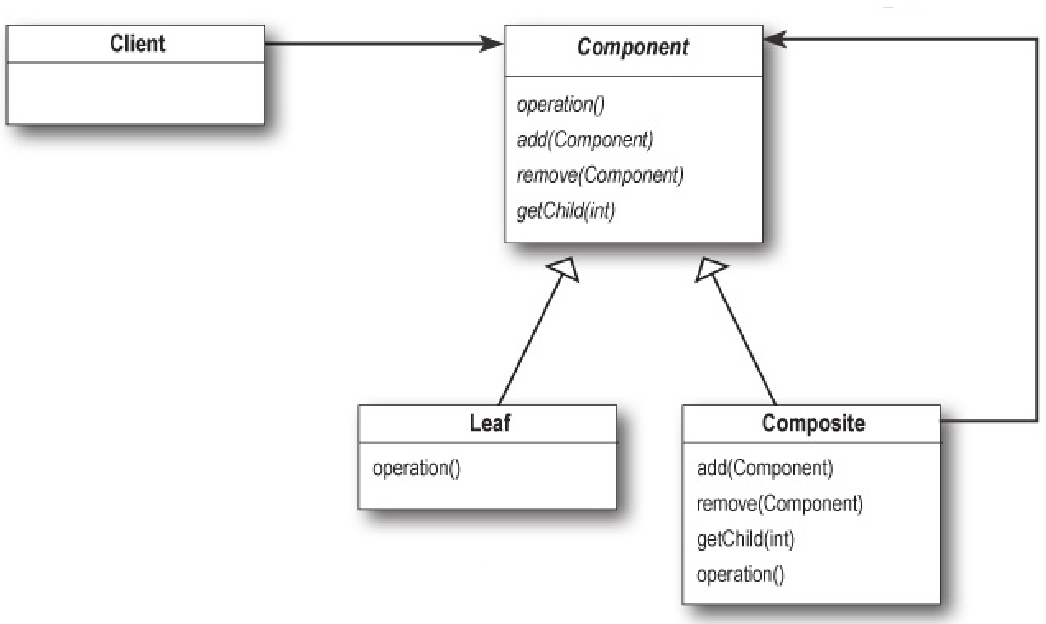
Composites in Java

```
public class CompositePatternDemo {
   public static void main(String[] args) {
      Employee CEO = new Employee("John","CEO", 30000)
      Employee headSales = new Employee("Robert","Head Sales", 20000);
      Employee headM;arketing = new Employee("Michel","Head Marketing", 20000);
      Employee clerk1 = new Employee("Laura","Marketing", 10000);
      Employee clerk2 = new Employee("Bob","Marketing", 10000);
      Employee salesExecutive1 = new Employee("Richard","Sales", 10000);
      CEO.add(headSales);
      CEO.add(headMarketing);
      headSales.add(salesExecutive1);
      headMarketing.add(clerk1);
      headMarketing.add(clerk2);
      //print all employees of the organization
      System.out.println(CEO);
      for (Employee headEmployee : CEO.getSubordinates()) {
         System.out.println(headEmployee);
         for (Employee employee : headEmployee.getSubordinates()) {
            System.out.println(employee);
         }
      }
   }
}
```

https://www.tutorialspoint.com/design_pattern/composite_pattern.htm

Composites in Python

- Abstract class requires components to have an operation
- Composite objects control the logic for adding and discarding elements from a set object
- Both composites and leaf objects have an operation to define



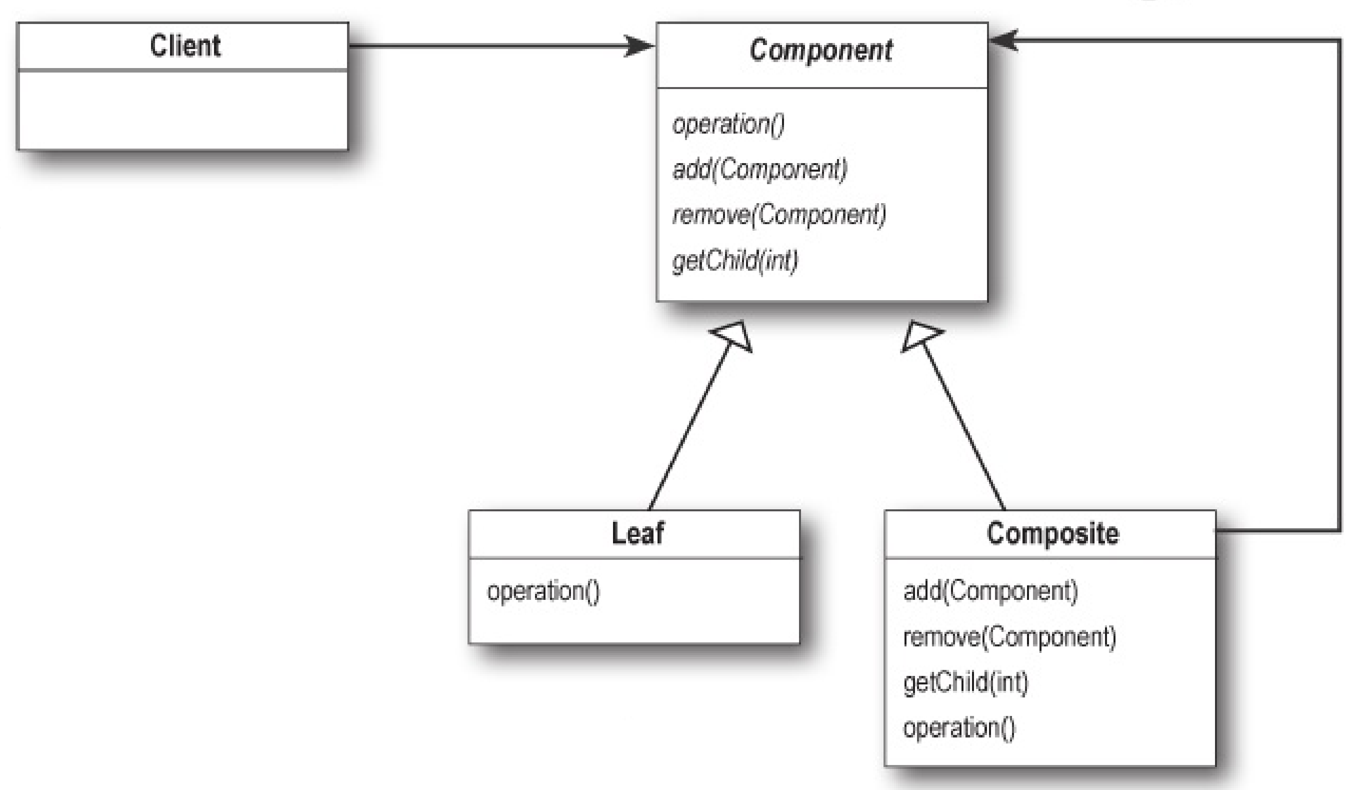
- https://sourcemaking.com/design_patterns/composite/python/1

```
class Component(metaclass=abc.ABCMeta):
    @abc.abstractmethod
    def operation(self):
        pass

class Composite(Component):
    # Components with children
    def __init__(self):
        self._children = set()

    def operation(self):
        for child in self._children:
            child.operation()

    def add(self, component):
        self._children.add(component)

    def remove(self, component):
        self._children.discard(component)

class Leaf(Component):
    # Components w/o children
    def operation(self):
        pass

def main():
    leaf = Leaf()
    composite = Composite()
    composite.add(leaf)
    composite.operation()
```

Broken Principle?

- We just went on about Single Responsibility…
- This pattern handles component operations AND…
  It handles the logic for managing hierarchies!?!

- It's true – we're trading the Single Responsibility principle for transparency
- Being able to use one interface to deal with component and leaf objects – makes the difference between those two things transparent to the client
- Patterns are guidelines, not rules



Key Points - Iterator

- An Iterator allows access to an aggregate's elements without exposing its internal structure
  - Can hide (abstract) the complexity of a data structure from clients (encapsulation of implementation and data)
- An Iterator takes the job of iterating over an aggregate and encapsulates it in another object
  - Example: a Tree data structure with a Depth-first Iterator and a Breadth-first Iterator
- 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 you to use polymorphism when writing code that makes use of the items of the aggregate
- Can reduce duplication of iteration code
- Can be a bit of overkill for an app with just simple collections

Key Points - Composite

- We should strive to assign only one responsibility to each class
- The Composite Pattern provides a structure to hold both 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 some design tradeoffs in implementing Composite. You need to balance transparency and safety with your needs.
- Be careful to use a tree structure only when that is what you need…

- Note: The Head First book example in Chapter 9 combines the Java Iterator to traverse a Composite tree to create a walk through menu items looking for Vegetarian options – take a look