Lesson 7 Iterator and Composite

The Iterator Pattern

1. Intent

   Provide a way to access the elements of an aggregate object (collection) sequentially without exposing its underlying representation

2. Motivation

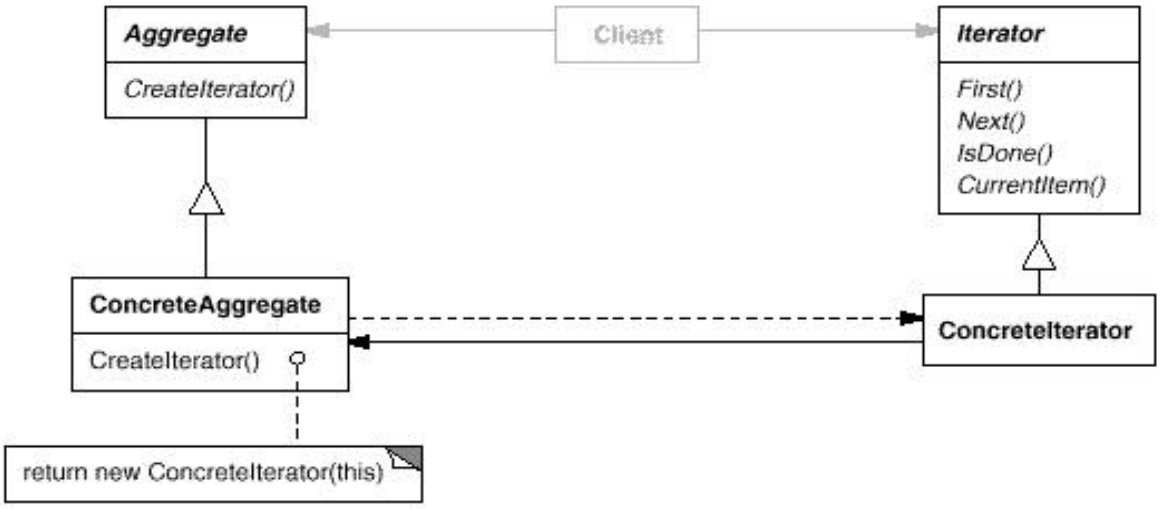
   a. An aggregate object such as a list or hash map should allow a way to traverse its elements without exposing its internal structure.
   b. It should allow different traversal methods depending on what the client needs (for example by using a functor).
   c. It should allow concurrent access by multiple threads.
   d. But we want to take the responsibility for access and traversal out of the aggregate object and put it into an Iterator object.

3. Applicability

   Use the Iterator pattern:

   a. To support traversals of aggregate objects without exposing their internal representation.
   b. To support multiple, concurrent traversals of aggregate objects.
   c. To provide a uniform interface for traversing different aggregate structures to support polymorphic iteration.

4. Structure



5. Participants

a. Iterator
   - defines an interface for accessing and traversing elements.

b. ConcreteIterator
   - implements the Iterator interface.
   - keeps track of the current position in the traversal of the aggregate.

c. Aggregate
   - defines an interface for creating an Iterator object.

d. ConcreteAggregate
   - implements the Iterator creation interface to return an instance of the proper ConcreteIterator.

6. Consequences

Simplifies the interface of the aggregate by not polluting it with traversal methods
Supports multiple, concurrent traversals
Supports variant traversal techniques

7. How to implement the Iterator pattern?

Aggregate (collection) interface

```
public interface Aggregate {
 public Iterator getIterator();
}
```

Iterator interface

```
public interface Iterator {
   public boolean hasNext();
   public Object next();
}
```

Concreate aggregate class with a nested concreate iterator class

```
public class NameRepository implements Aggregate {
         private String names[] = {"Rob" , "Jon" ,"Jul" , "Lor", "Pat",
"Ken"};

         //other methods of the NameRepository
         …

         @Override
         public Iterator getIterator() {
            return new NameIterator();
         }

         private class NameIterator implements Iterator {

            int index;

            @Override
            public boolean hasNext() {

               if(index < names.length){
                  return true;
```

```
            }
            return false;
        }

        @Override
        public Object next() {

            if(this.hasNext()){
                return names[index++];
            }
            return null;
        }
    }
}
```

Lab 7-1

1. Suppose the name repository in the above example uses a 2-dimensional array to store the names. Names can be dynamically added or removed from it. When you remove a name, you simply replace the name with a "-". (You do not need to implement the add/remove methods though). Rewrite the NameIterator class that implements the same Iterator interface. But make sure that a "-" is never returned by the next() method.
2. (Do not do this one.)Implement the polymorphic iterator we discussed in class. Note: you must use a Factory Method to create different iterators. You can use the 2 name repository classes as your concrete aggregates.
3. (Do not do this one.)Design a data structure that allows a client to quickly search by either people's first name or last name. Then provide 2 iterators for the data structure - FirstnameIterator and LastnameIterator.
   Hint: Think about using rosters. Most rosters are in alphabetical order of either first name or last name. Suppose we know someone by first name only. But if a roster (with at least hundreds of names on it) is ordered by last name, it will be hard to find the person by first name. Let's solve the problem by:

1) Implementing a data structure (to store names) that gives a good performance for searching both by first and by last name.
2) Based on your data structure, provide 2 iterators that traverse through all names by first and by last name.
3) (Optional) Provide some metrics on the performance to compare with your classmates' implementations. Use a same 500-name roster for testing.

The Composite Pattern

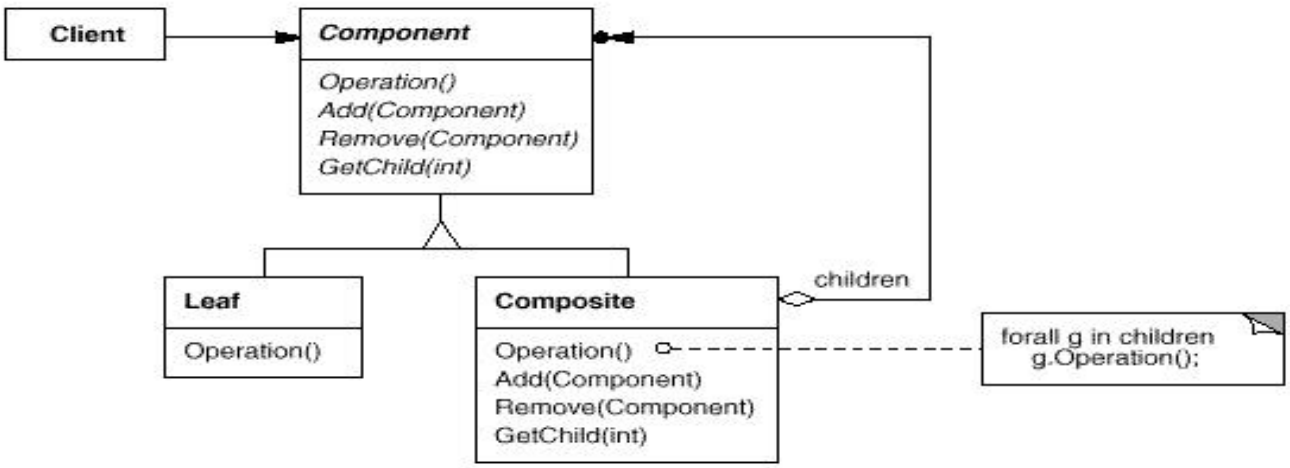
1. Intent

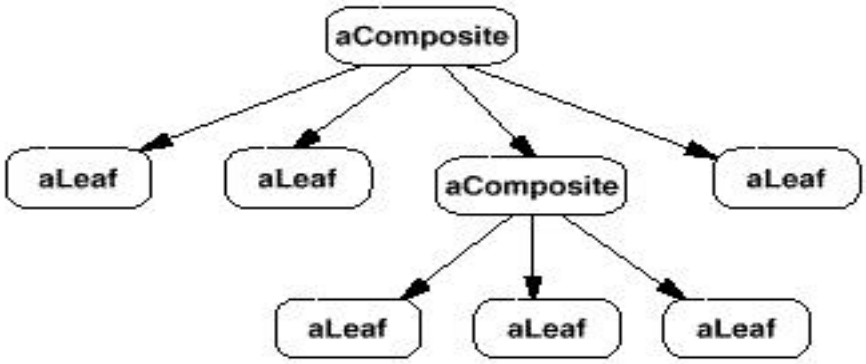
   Compose objects into tree structures to represent part-whole or parent-child hierarchies.
   Composite lets clients treat individual objects and compositions of objects uniformly.

2. Motivation

   Many times you need to model a system that deals with relationships between objects that are of the same type but on different levels (for example in whole-part, parent-child, or supervisor-employee, relationships.) and in your system the number of levels is unknown until runtime.

3. Structure





4. Participants

a. Component
- declares the interface for objects in the composition.
- implements default behavior for the interface common to all classes, as appropriate.
- declares an interface for accessing and managing its child components.
- (optional) defines an interface for accessing a component's parent in the recursive structure, and implements it if that's appropriate.

b. Leaf
- represents leaf objects in the composition. A leaf has no children.
- defines behavior for primitive objects in the composition.

c. Composite
- defines behavior for components having children.
- stores child components.
- implements child-related operations in the Component interface.

d. Client
   - manipulates objects in the composition through the Component interface.

5. Applicability

Use the Composite pattern when

a. You want to represent part-whole or parent-child hierarchies of objects
b. You want clients to be able to ignore the difference between compositions of objects and individual objects. Clients will treat all objects in the composite structure uniformly.

6. Implementation of Composite Pattern

a. Component abstract class

```
public abstract class Component {
      private Collection<Component> list = new ArrayList<Component>();
      protected String title;
      public abstract void print();
      public void addItem(Component item){

            list.add(item);

     }

}
```

b. Composite is a component that can contain other components

```
public class Composite extends Component{

      public Composite(String title) {

            super(title);

      }
```

```
    public void print() {

          System.out.println( "Composite name=" + title );

          for (Component item : list){

                item.print();

          }

    }

}
```

c. Leaf is a component that does not contain others

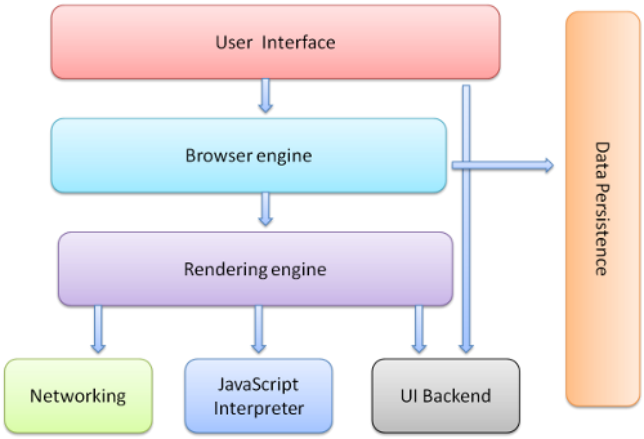
```
    public class Leaf extends Component {
          private String number;

          public Leaf(String number, String title) {
                super(title);
                this.number = number;
          }
          //for addItem() method, print a message "cannot add child"
          public void print() {
                System.out.println("Leaf [isbn=" + number + ", title=" +
    title + "]");
          }

    }
```

d. Client

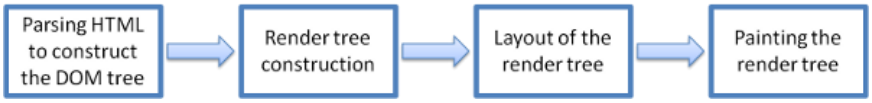
```
    public class Client {

          public static void main(String[] args) {
                Component root = new Composite("root");
                Component leaf1 = new Leaf("1", "leaf1");
                Component comp = new Composite("composite");
                Component leaf2 = new Leaf("2", "leaf2");
                comp.add(leaf2);
                root.add(leaf1);
                root.add(comp);
          }
    }
```

Lab 7-2

Below is a high-level view of the browser architecture. We are going to implement a "render tree" for the rendering engine. A render tree is basically a data structure that stores all visual elements in an html document with the original relations, dimensions, stylings information kept (for example, parent-child/sibling relations, height, width, color, styles, etc.).



The rendering engine processes/renders information in the following sequence. The "render tree" is created in the second "box" in the diagram.



For the lab, you will implement the render tree based on a given html file. Then provide a paint() method for the tree. The following is an example html file you can use.

```
<HTML>
<HEAD>
<TITLE>Your Title Here</TITLE>
</HEAD>
<BODY>
<CENTER><IMG SRC="clouds.jpg" > </CENTER>
<a href="http://somegreatsite.com">Link Name</a>
<H1>This is a Header</H1>
<H2>This is a Medium Header</H2>
<B>This is a new paragraph!</B>
<B><I>This is a new sentence without a paragraph break, in bold italics.</I></B>
</BODY>
</HTML>
```

So you will create a tree-structure that stores all the elements in the html file. Then call the paint() method from the client.