**Oregon Institute of Technology**

ExternalCompositeIteratorVisitor Lab Winter Term   
Due in lab Feb 23

### Overview

**Goals:**

* Write an External Iterator that clients can use to traverse the hierarchical data structure built in previous lab.
* Use the Visitor pattern concepts to implement both a company visitor and an employee visitor.

The Iterator built in part one was used internally by the Composite class, hence the label “Internal Iterator.” In this lab we will implement an “External Iterator” that provides the ability for client code to traverse the composite data structure.

Through class discussion, we know that, in order to not pollute the Component participant, we only added methods essential to both the Composite and Leaf participants. Now that we have an External Iterator, the client code can perform its own operations as it traverses the data structure. Note however, that the client code will probably need to speculatively cast each component returned to do any real work.

Our code would have considerably more value to our clients if it provided frequently used operations and removed the responsibility for coding operations, iterating, and speculative casting. Enter the Visitor Pattern.

In addition, by using the Visitor pattern, it becomes easy to add operations that a particular client may ask for without changing the existing code base. Sound familiar—closed for modification and open for extension.

### Part One Steps—Implementing an “External Iterator”

### Start with the code you wrote at the end of last lab. There is not a lot of code to write to implement an External Iterator once the Inner Iterator and Composite patterns are in place. However, the code is a bit “intense” so please take time to understand it. Refer to the ExternalCompositeIterator demo for assistance.

### Add a class named CompositeIterator and derive it from AbstractIterator. The CompositeIterator will, of course, need to override the methods of AbstractIterator. In addition, it will need a private Stack data member that holds AbstractIterator types. You may use the .NET Stack class, or simply build your own.

### The c’tor will receive a component type as its only parameter. The c’tor will assign the internal reference to the component parameter. It will then push an instance of the component’s iterator onto the stack. Lastly, it needs to set the internal “current” data member to 0;

### The First() method delegates to the First() method of the top iterator on the Stack.

### The GetCurrent() method is simple as well, it delegates to the GetCurrent() method of the top iterator on the Stack.

### MoveNext() is a bit more involved. If the top iterator on the Stack points to a composite, it needs to create and push that composite’s iterator (for further iteration) onto the stack. Otherwise, it delegates to the MoveNext() method of the top iterator on the Stack.

### Now for the most complicated—the IsDone() method.

### Check to see if the Stack is empty...if so, IsDone() should return true.

### “Else” If the iterator at the top of the stack “IsDone()”:

### Pop the iterator at the top of the stack

### If the Stack still isn’t empty—call the MoveNext() method of the (new) top iterator on the Stack and then recursively call IsDone(). “Else” If the iterator at the top of the stack is not done, return false.

### In program.cs, demonstrate that your external composite iterator works by printing out the names of the composite and leaf nodes.

### Part Two Steps—Implement the Visitor Pattern.

1. Now for some fun!! Refer to the ExternalCompositeIteratorVisitor demo. Create an IVisitor.cs file. The IVisitor interface will declare two methods named Visit(). One of the Visit() methods will take a CompanyComposite paramenter, the other will take an Employee parameter. Visit (Employee employee) will perform an operation on every Employee node that it visits and Visit(CompanyComposite composite) will perform an operation on every CompanyComposite node that it visits.
2. Create a concrete CompanyCompositeVisitor class that derives from the IVisitor interface and takes a component type as a parameter for its c’tor
3. In the c’tor polymorphically call the component type’s Accept(IVisitor visitor) method. (Note, we haven’t refactored the CompanyComponent participant yet so this won’t compile.)
4. In the Visit(CompanyComposite) method—output to the console that we are at a companycomposite node. In an actual CompanyCompositeVisitor, this is where the new operation would be coded.
5. In the Visit(Employee) method—do nothing, we are only interested in CompanyComposites.
6. Now we need to refactor the CompanyComponent particpant to be accessible to the visitor—add an abstract Accept method that takes an IVisitor as a parameter. Both the leaf class and composite class will need to override this method.
7. Implement Accept(IVisitor visitor) in the CompanyComposite class—using the Visitor passed in, call Visit(this) and then use internal iterator logic to call Accept(visitor) on the next nodes.
8. Implement Accept(IVisitor visitor) in the Leaf class—using the Visitor passed in, call Visit(this). Because this is a leaf node, there is no traversal code (unless you want to use a null Iterator).
9. Using the above steps, build a Visitor that ouputs something interesting to the console everytime it visits a Leaf node and does nothing when visiting a composite node. Call it an EmployeeVisitor.
10. Use your existing hiearchical data structure where Companies contain companies that ultimately contain Employees and demonstrate that both of your Visitors work.

|  |
| --- |
|  |