Iterator, Composite and Visitor

Estimated time for completion: 75 minutes

Overview:

There are many different kinds of collections, from lists to trees to stacks and queues. Often you will want to iterate over the collection. .NET provides a standard set of type-safe collections that provide standard iterators, however, there are times that an application needs to provide its own collections and provide iteration over those collections. In this lab you will explore three iteration patterns, iterator, composite and visitor.

Simple iterators are well catered for in .NET especially in .NET 2.0 with the 'yield return' keyword. It is straightforward to provide a method that can iterate over its classes data.

Hierarchies are more interesting. Hierarchies are collections that contain items where some of those items are themselves collections. In this case, if you need to provide a uniform iteration mechanism you can use the composite pattern. In the lab, you will implement the composite pattern to iterate over a hierarchy.

When iterating, you sometimes only want to get an item if a given condition holds. To do this you can use the strategy pattern and pass a filter to the iterator. You will (optionally) try this in the lab.

Both of the above are examples of 'internal' iteration, where the class themselves provide the iterator, all the user of the class has to do is to use foreach to iterate over the data. However there are occasions when either the collection does not provide an iterator, or the iteration does not do what the client needs. It is also possible that the iterator code is not type safe and that to do any work the client has to speculatively cast the item passed during the iteration before deciding what to do with that item. In this case it may be better to put the client in charge of the iteration, and for this you can use the visitor pattern

Goals:

- Understand how to implement iterators in .NET 2.0
- Write your own iterator
- Use the composite pattern to iterate over hierarchies
- Use strategies to provide behaviour to an iterator
- Use the visitor pattern to provide pull based iteration over collections

Lab Notes:

Part 1: Implementing an Iterator

In this part of the lab you will use the yield return keyword to implement an iterator.

1. Open the solution (Invoices.sln) and familiarize yourself with the project.
2. The initial solution has class several class definitions that define an invoice, these are LineItem, Order and Invoice. The idea is that an invoice contains orders, and that orders contain line items. However each order can also contain sub-orders nested within it.
3. The first part of the lab is to change invoices to support iteration. You will initially handle a simple example of iterating over the top level order and leave the processing of nested orders until the next part.
4. Open Invoice.cs and add a GetItems to the Invoice class. This method should return an IEnumerable<LineItem> and call the GetItems on the Order class
5. Change Main to call Invoice's GetItem method, and print out all the items
6. Build and run the code. You should see a list of line items produced on the console.

Part 2: Using the Composite Pattern

The invoice actually consists of orders that contain sub-orders and line items. The above code only iterates over the top level line items. You will now change the code to implement the composite pattern to iterate over nested orders.

1. Open the solution (Invoices.sln) and familiarize yourself with the project.
2. The basis of the composite pattern is to have a single interface that all members of the composite will implement. In this case the members of the composite are the LineItem and Order classes. Add an interface called IItem, this will be implemented by Order and LineItems
3. The next step is to change IItem to support Order functionality
4. Add a definition of GetItems to the IItem interface, this will be implemented by both Order and LineItem This should return something of type IEnumerable<IItem>
5. Change Order so that it implements IItem
6. Change LineItem so that it implements IItem, implement the GetItems method.
7. In Order change all references of IEnumerable<LineItem> to IEnumerable<IItem>
8. In Invoices change all references of IEnumerable<LineItem> to IEnumerable<IItem>
9. In Main change all references of IEnumerable<LineItem> to IEnumerable<IItem>
10. Build and run the code, it should work as before
11. Change IItem to support Item functionality
12. Add declarations of the Count, Description and UnitPrice properties to IItem
13. Add definitions of the Count, Description and UnitPrice properties to Order, these will be empty implementations.
14. Build and run the code it should still work.
15. Order is the class that will contain both lineitems and other orders. The Order class has to be changed to support nested orders
16. Look at the ProcessOrder method. This currently reads a new nested order from the .csv file, gets the order to process its own LineItems but the method does not add that order to the IItems collection.
17. Add a call to LineItems.Add(order) to add the new order to the IItems collection

```
private void ProcessOrder(StreamReader reader, string title)
```

```
{
    Order order = new Order();
    order.Title = title;
    string lineRead;
    while ((lineRead = reader.ReadLine()) != null)
    {
        string[] lineItemSplit = lineRead.Split(new char[] { ',' });
        if (lineItemSplit[0] == ORDER_END_IDENTIFIER)
            break;

        order.ProcessLine(reader, lineRead);
    }
    LineItems.Add(order);
}
```

18. Build and run the code. It works after a fashion. You see all the line items, but where there are orders in the collection you see a line printed which gives the title of the orders collection but not the line items within that collection, not quite what you want.
19. This is because the iteration code needs to be changed to support iteration of nested Orders.
20. Currently the code returns each line IItem it finds, but the code needs to iterate over the items as any one may now be an order.
21. Change the GetItems method of Order so that it first yield returns itself, then iterates over each item in the collection.
22. For each item in the collection it should iterate over that items children yield returning them.

```
public IEnumerable<IItem> GetItems()
{
    yield return this;

    foreach (IItem item in LineItems)
    {
        foreach (IItem childItem in item.GetItems())
        {
            yield return childItem;
        }
    }
}
```

23. You now need to implement GetItems in the LineItem class. This should simply yield return this;
24. Build and run the code. It should now be working. Listing the line items and the nested orders.

Part 3: Using a filter

There are occasions when the caller of the iterator does not want to get all the items in the collection, you would like to filter them. You will amend the iterator so that you can pass a filter

to determine which items are returned during iteration. .NET 2.0 already has a delegate defined for this behaviour called Predicate<T>

1. Open the solution (invoices.sln) and familiarize yourself with the project.
2. In IItem change the definition of GetItems so that it takes something of type Predicate<IItem> filter. Change the corresponding definitions in Order and LineItem.
3. Where etItems is called in Invoice.cs pass null. Where GetItems is called in Order.cs used the passed filter parameter.

```
public IEnumerable<IItem> GetItems(Predicate<IItem> filter)
{
    yield return this;

    foreach (IItem item in LineItems)
    {
        foreach (IItem childItem in item.GetItems(filter))
        {
            yield return childItem;
        }
    }
}
```

4. Build and test the code. It should work as before
5. You now need to change the code to check to see if a filter has been passed and if so, call it. If the filter returns true then execute the iterator, if not, do no execute the iterator

---

Part 4: Using the Visitor Pattern

1. Open the solution (InvoiceVisitor.sln) and familiarize yourself with the project.
2. In this lab you have an invoice that can consist of different line item types. The Invoice has LineItem, DiscountLineItem and RefundLineItem types. Each of these types has different fields and will require different processing.
3. Currently Program has two functions that process the different line items. If you look at these functions you will see that the functions work in a similar way. They both iterate over the collection, then cast the returned IItem to the appropriate type. This makes the two methods fragile. To fix this you will create a visitor
4. The first thing you have to do is to define the visitor types
    a. First define a visitor interface, call it IItemVisitor. Add one method to the interface for each type you will visit

```
public interface IItemVisitor
{
    void VisitLineItem(LineItem lineItem);
    void VisitRefundLineItem(RefundLineItem refundLineItem);
    void VisitDiscountLineItem(DiscountLineItem discountLineItem);
}
```

    b. Now you need to define a class to implement the visitor behaviour.

    c. Create a class called AccumulatorVisitor and have it implement IItemVisitor
    d. Add a property of type double called Amount
    e. Copy the code from the GetTotalPayable method is Program.cs and put it in the appropriate method in AccumultatorVisitor
5. Now that you have implemented the visitor you need to change the line items to accept the visitor
6. Add a definition of void Accept(IItemVisitor visitor); to IItem and implement this in each class. For example the RefundLineItem will look like this

```
public void Accept(IItemVisitor visitor)
{
    visitor.VisitRefundLineItem(this);
}
```

7. Check that the code builds.
8. In the Program class change the code that calculates the amount with code that uses the visitor.
9. To do this, iterate over the collection calling each node'sAccept method passing a reference to a AccumulatorVisitor.
10. Build and run the code, it should produce the same output as before.
11. As a final optional step move the PrintReport code into a visitor