Iterator, Composite and Visitor

DEVELOPMENTOR

DEVELOPING PEOPLE WHO DEVELOP SOFTWARE

Iterator Pattern

- Provides a standard mechanism to allow access to aggregate objects
  - Provides navigation and access methods
  - Implemented in .NET with `IEnumerable<T>` and `IEnumerator<T>`
  - Consumed in C# with `foreach`
  - C# 2.0 yield `return` `keyword` allows for easier implementation

The Iterator Pattern provides a way to access the elements of an aggregate object without exposing its underlying representation

Collection classes

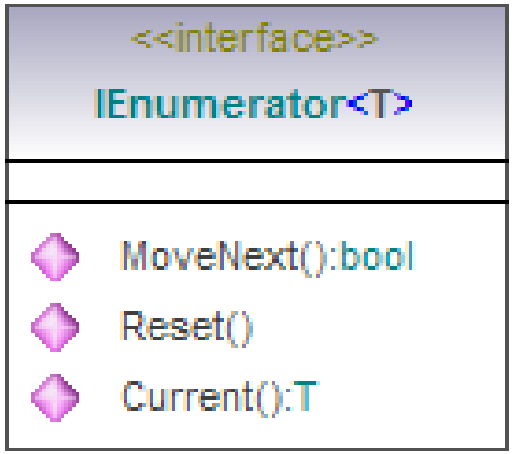
- Provide access to their data through an iterator
  - Don't want to know how implementation details of collection
  - Implement `IEnumerable`, `IEnumerable<T>` to signal intent
  - Use foreach to consume

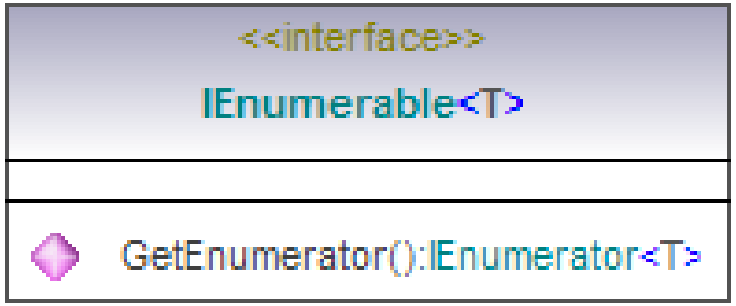
```
public class List<T> : IList<T>, ICollection<T>,
       IEnumerable<T>, IList, ICollection, IEnumerable
```

```
List<Entry> entries = new List<Entry>();
foreach(Entry e in entries)
{
    e.print();
}
```

Iterator Implementation

- Implement two interfaces to take part in iteration
  - `IEnumerable<T>` signals support for iteration and `foreach`
  - returns an implementation of `IEnumerator<T>`
  - `IEnumerator<T>` provides the iterator





Implementing Your Own Iterator

- May want to implement iterator for various reasons
  - May want to filter data you are iterating over
  - Concatenating multiple collections into a single iteration
- Two ways of implementing iterators:
  - Hard way is to implement IEnumerable/IEnumerator
  - Easy way is to use C# 2.0 yield return keyword

Example

- Blogging engine has Blog Categories
  - Each category has blog entries
  - Category class holds entries in a List<Entry> collection
  - Want to display entries from before a specific date

```
public class BlogCategory
{
    List<BlogEntry> entries = new List<BlogEntry>();
}
```

Implementing the Iterator

- Use `yield return` to implement an iterator
  - Method returns an `IEnumerable<T>`
  - `yield return` 'implements' the iterator

```
public IEnumerable<BlogEntry> GetBlogEntries()
{
    foreach (BlogEntry entry in entries)
    {
        yield return entry;
    }
}
```

Implementing the Filtered Iterator

- Pass a delegate to the iterator method
  - Can use built in Predicate<T> delegate
  - `public delegate bool Predicate<T> (T obj)`

```
public IEnumerable<BlogEntry>
   GetFilteredBlogEntries(Predicate<BlogEntry> filter)
{
    foreach (Blog entry in entries)
    {
        if ((filter == null) || filter(entry))
            yield return entry;
    }
}
```

Using the Filtered Iterator

- Can use `foreach`
  - pass a delegate to the iterator

```
Predicate<BlogEntry> test =
                          delegate(BlogEntryOrig entry){
    DateTime Jan012007 = new DateTime(2007, 1, 1);

    return (entry.Date().CompareTo(Jan012007) < 0);
};

foreach (BlogEntry entry in
                       cat.GetFilteredBlogEntries(test)){
    entry.Print();
}
```

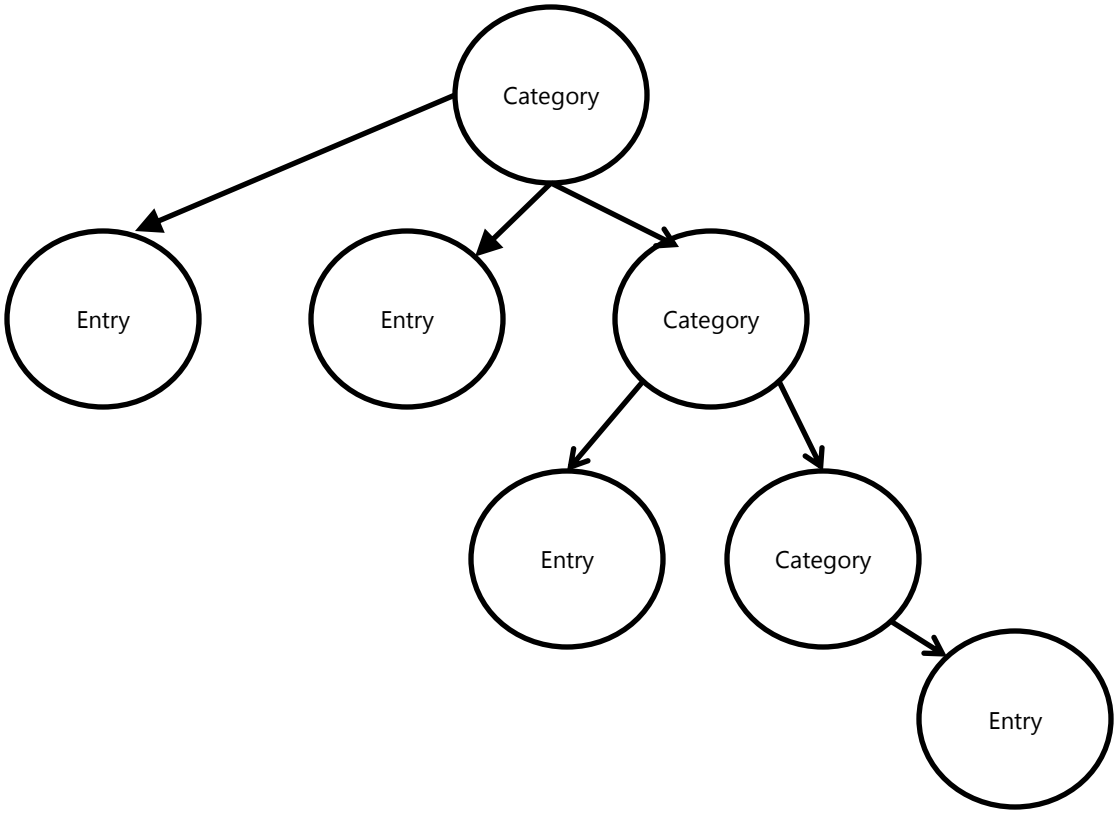
Composite Pattern

- Often want to iterate over hierarchical collections
  - Menus
  - File system
- Typically there are different types of items in the hierarchy
  - Menus and MenuItems
  - Directories and Files
- Processing hierarchies is easier if all items are the 'same'
  - Implement a common interface

The Composite Pattern allows you to compose objects into tree structures. Composite lets clients treat individual objects and composites uniformly

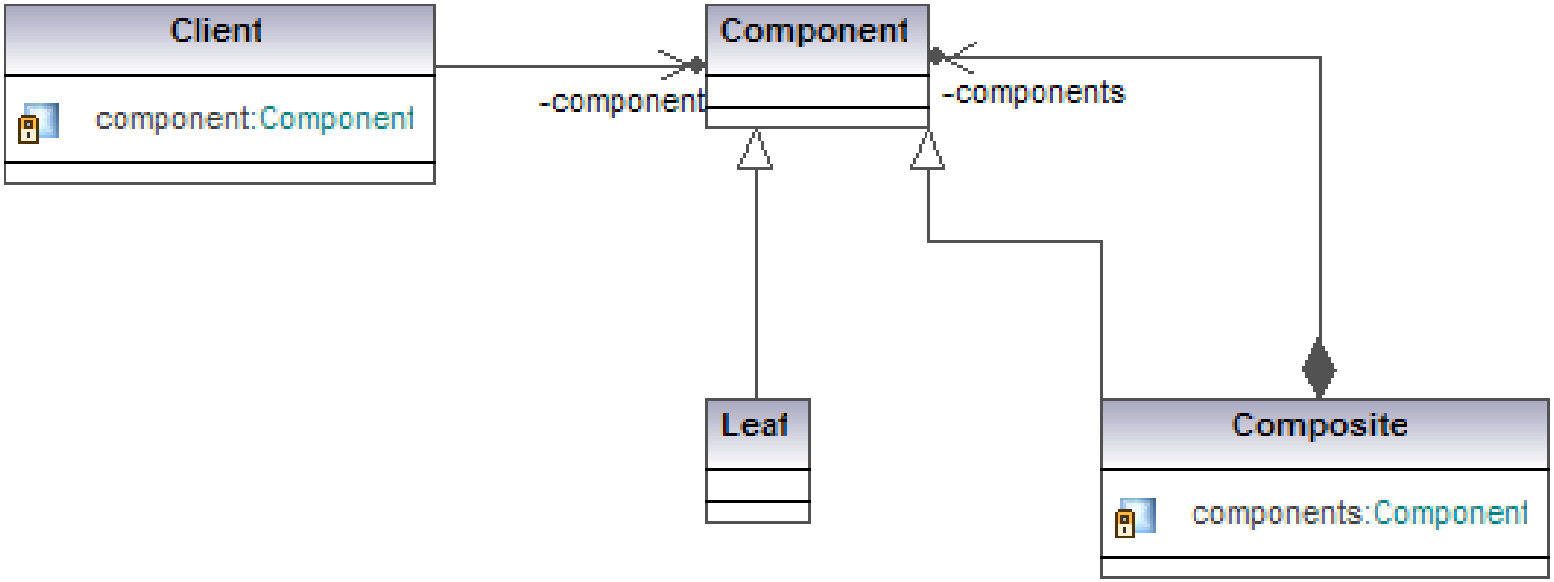
Example: Blog Categories

- Categories can be nested
  - Categories can contain entries or other categories
  - Categories can be nested arbitrarily deeply



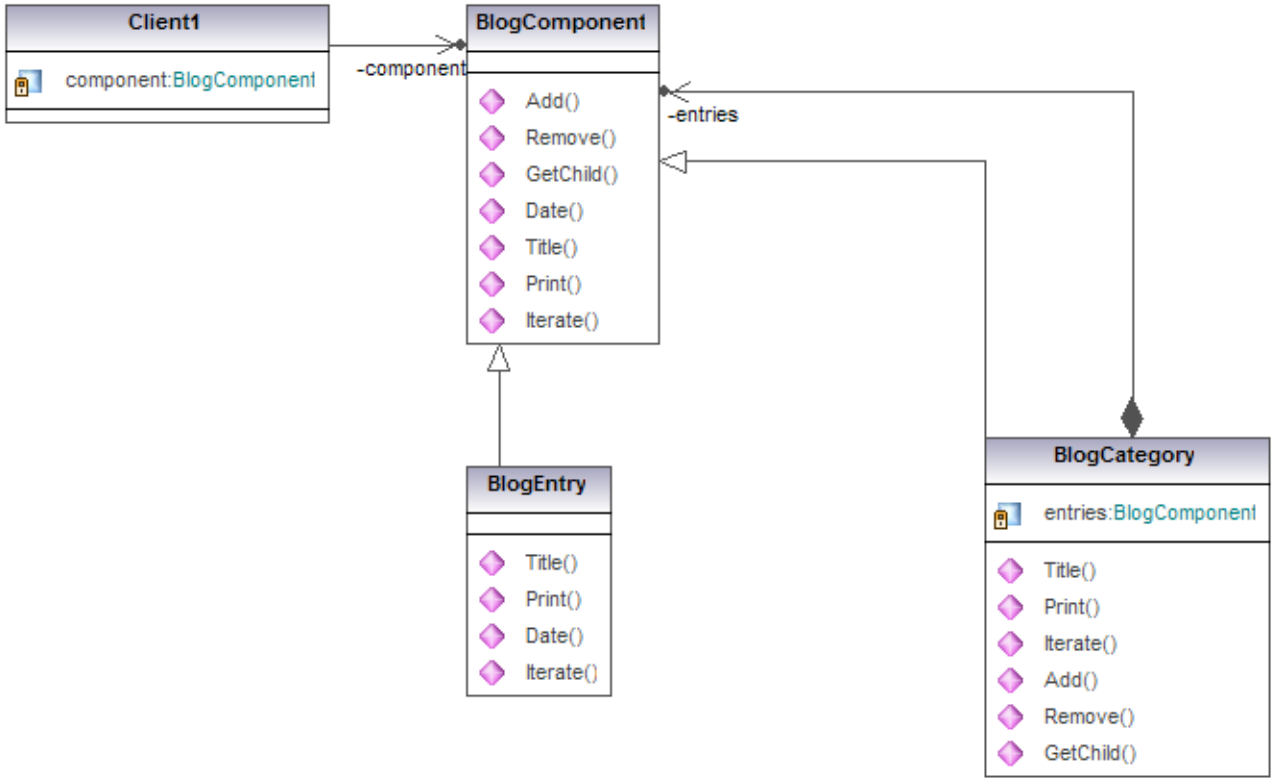
General Composite

- To implement the composite
  - Each node needs to derive from a common base class
  - Although derived class won't necessarily implement all the methods of the base



Blog Composite

- BlogComponent is the root class
  - BlogEntry and BlogCategory derive from this
  - BlogCategory holds a collection of BlogComponents



Common Base Class

- This is what the client codes against
  - Not all methods implemented by all deriving classes
  - Provide default implementations for operations
    - typically throw `NotSupportedException`

```
public abstract class BlogComponent
{
  // Blog Category operations
  public virtual void Add(BlogComponent blog){throw new NotSupportedException(); }
  public virtual void Remove(BlogComponent blog){throw new NotSupportedException();}

  // BlogEntry operations
  public virtual DateTime Date { get; set; }

  // operations on both
  public abstract string Title { set; get; }
  public abstract void Print();

  public abstract IEnumerable<BlogComponent> Children { get; }
}
```

Node Class

- This class can hold leaves or other Nodes
  - Implements methods necessary to be a node
  - Implements methods client can all on nodes and leaves

```
public class BlogCategory : BlogComponent
    {
         // other methods elided for clarity
         // implementations elided for clarity

        private List<BlogComponent> children = new List<BlogComponent>();

        public override void Add(BlogComponent blog) { ... }
        public override void Remove(BlogComponent blog) { ... }

        public override string Title { .. }
        public override void Print() { }

        public override IEnumerable<BlogComponent> Children
        {
            get { return children.OfType<BlogComponent>(); }
        }
    }
```

Leaf Class

- This class represents leaves
  - Implements methods necessary to be a leaf
  - Implements methods client can all on nodes and leaves

```
class BlogEntry : BlogComponent
{
   // other methods elided for clarity
   // implementations elided for clarity

   public override string Title
   { get; set; }

   public override void Print() {...}

   public override IEnumerable<BlogComponent> Children
   {
     get { yield break; }
   }
}
```

Client Usage

- Iterate over the children
  - Treat each entry in the collection as common base
  - regardless of whether it's a node or a leaf

```
BlogCategory cat = GetRootCategory();

// is it a leaf?
// is it a node?
// no, it's a component!
foreach (BlogComponent component in cat.Children)
{
    Console.WriteLine(component.Title);
}
```

Consume Tree as a flat list

- Add a FindAll method to the Component base class
- Builds upon Children property for depth first traversal

```
public abstract class BlogComponent
{
  // other methods elided for clarity
   public virtual IEnumerable<BlogComponent> FindAll()
   {
     yield return this;

     foreach (BlogComponent cpt in this.Children)
     {
       foreach (BlogComponent child in cpt.FindAll())
       {
         yield return child;
       }
     }
   }
}
```

return ourselves first

get each entry in our collection

iterate over that and yield return it

Safer Alternative to call stack based iteration

- Call stack recursion can be fragile for large composites
- Heap based stacks are less likely to run out of memory

```
public abstract class BlogComponent
{
  // other methods elided for clarity
   public virtual IEnumerable<BlogComponent> FindAll(){
     Stack<BlogComponent> cpts = new Stack<BlogComponent>();
     cpts.Push(this);

     while( cpts.Count > 0 ) {
        BlogComponent cpt = cpts.Pop();
        yield return cpt;

        foreach( BlogComponent childCpt in cpt.Children ) {
         cpts.Push(childCpt);
        }
      }
   }
}
```

Client Usage

- Using FindAll returns all BlogComponents as a flat list
- Composite structure can now change with out impacting client

```
BlogCategory cat = GetRootCategory();

// is it a leaf?
// is it a node?
// no, it's a component!
foreach (BlogComponent component in cat.FindAll())
{
    Console.WriteLine(component.Title);
}
```

LINQ usage

- Find all BlogEntries for 2009

```
BlogCategory cat = GetRootCategory();

 var entries = from blogEntry in cat.FindAll()
                                    .OfType<BlogEntry>()
               where blog.Date.Year == 2009
               select blogEntry;
```

Recap

- Composite allows the use of a consistent interface to traverse hierarchies
  - Nodes and leaves implement a common base interface
  - Nodes and leaves implement the appropriate methods
  - Client treats nodes and leaves as the same type
  - Client may need to check the node/leaf type
- XmlDocument uses this in the Framework
  - Everything in the XmlDocument is an XmlNode

Visitor Pattern

- Client may want to call extra methods nodes of a hierarchy
  - i.e. the methods do not exist on the nodes
  - these methods add extra capabilities
- Can Visit each node and make the method call
  - Visitor needs access to the nodes data

The Visitor Pattern lets you define an operation without changing the classes of the elements on which it operates.

Visitor Class

- Create a `Visitor` class
- Add `VisitXXX` methods to the class
  - These methods can be specific to the node type

```
public abstract class BlogVisitor
{
    public virtual void VisitBlogEntry(BlogEntry entry) { }
    public virtual void VisitBlogCategory(BlogCategory category){ }
}

public class SaveToTextFileVisitor : BlogVisitor
{
    public override void VisitBlogEntry(BlogEntry entry) {}

    public override void VisitBlogCategory(BlogCategory category) {}
}
```

Define an Accept method on the Nodes

- Each node that will be visited needs a `Accept` method
  - `Accept` calls the appropriate method on the `Visitor`

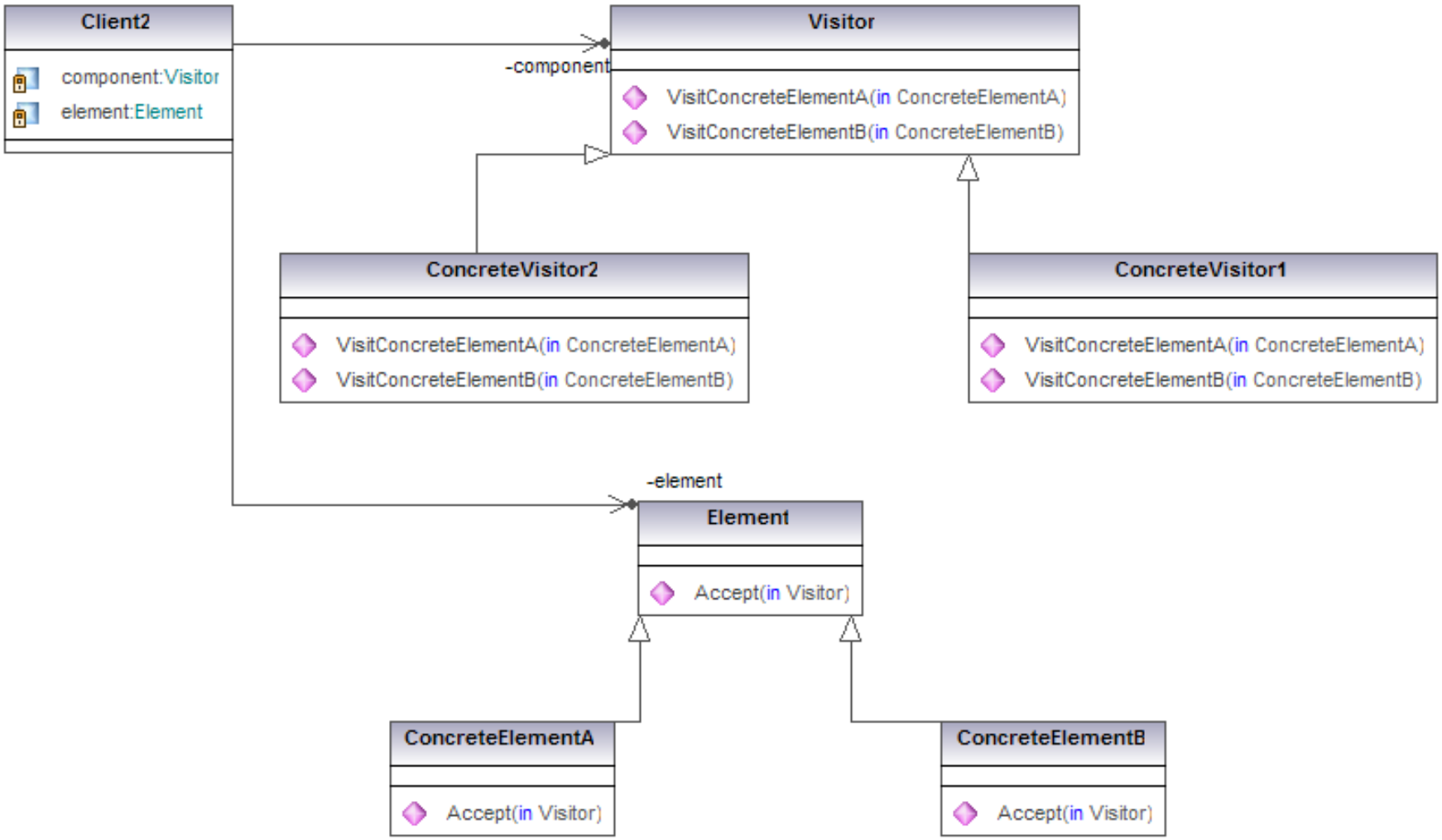
```
public abstract class BlogComponent{
    public abstract void Accept(BlogVisitor visitor);
}

public class BlogEntry : BlogComponent{
    public override void Accept(BlogVisitor visitor){
        visitor.VisitBlogEntry(this);
    }
}

public class BlogCategory : BlogComponent {
    public override void Accept(BlogVisitor visitor) {
        visitor.VisitBlogCategory(this);
        foreach (BlogComponent component in entries) {
            component.Accept(visitor);
        }
    }
}
```

internal iteration to visit children

Visitor


Client2
component:Visitor
element:Element
-component
Visitor
VisitConcreteElementA(in ConcreteElementA)
VisitConcreteElementB(in ConcreteElementB)
ConcreteVisitor2
VisitConcreteElementA(in ConcreteElementA)
VisitConcreteElementB(in ConcreteElementB)
ConcreteVisitor1
VisitConcreteElementA(in ConcreteElementA)
VisitConcreteElementB(in ConcreteElementB)
-element
Element
Accept(in Visitor)
ConcreteElementA
Accept(in Visitor)
ConcreteElementB
Accept(in Visitor)

Implementation Issues - Iteration

- Who is responsible for Iteration
  - Iteration can be provided by the hierarchy
  - Iteration can also be provided by an external iterator
  - Don't put the iteration in the visitor
    - Each concrete visitor will then need to provide the code

Implementation Notes - Encapsulation

- Encapsulation
  - Visitor may need access to object's state
  - This may break encapsulation

Implementation Notes – Double Dispatch

- For Single Dispatch Languages
  - Method call depends on the name of the method and the type calling it
    - e.g. `entry.Print()`
- In Double Dispatch
  - depends on the name of the method and the type of two objects in the call
  - Accept is a double dispatch method
  - It relies on the type of the caller (the Visitor) and the type on which Accept is being called

Summary

- Use Iterator to iterate over a collection of nodes
- Use composite when you want to provide uniform iteration
- Use Visitor to layer behaviour onto nodes in a hierarchy