

Iterator Pattern

Iterator

NEW Forum Discussions: NEW

What Design Pattern Should I Choose?

Design Principles and Patterns Iterator Motivation One of the most common data structures in software development is what is generic called a collection. A collection is just a grouping of some objects. They can have the same type or they can be all cast to a base type like object. A collection can be a list, an array, a tree and the examples can continue.

But what is more important is that a collection should provide a way to access its elements without exposing its internal structure. We should have a mechanism to traverse in the same way a list or an array. It doesn't matter how they are internally represented.

The idea of the iterator pattern is to take the responsibility of accessing and passing through the objects of the collection and put it in the iterator object. The iterator object will maintain the state of the iteration, keeping track of the current item and having a way of identifying what elements are next to be iterated.

Intent Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

The abstraction provided by the iterator pattern allows you to modify the collection implementation without making any changes outside of collection. It enables you to create a general purpose GUI component that will be able to iterate through any collection of the application.

Implementation

Applicability & Examples The iterator pattern allow us to:

- access contents of a collection without exposing its internal structure.
- support multiple simultaneous traversals of a collection.
- provide a uniform interface for traversing different collection.

Example 1: This example is using a collection of books and it uses an iterator to iterate through the collection. The main actors are:

- Iterator - This interface represent the AbstractIterator, defining the iterator
- BookIterator - This is the implementation of Iterator (implements the Iterator interface)
- IContainer - This is an interface defining the Aggregate
- BooksCollection - An implementation of the collection

Here is the code for the abstractions Iterator and IContainer:

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

```
interface IContainer
{
    public Iterator createIterator();
}
```

And here is the code for concrete classes for iterator and collection. Please note that the concrete iterator is a nested class. This way it can access all the members of the collection and it is encapsulated so other classes can not access the BookIterator. All the classes are not aware of BookIterator they use the Iterator:

```
class BooksCollection implements IContainer
{
    private String m_titles[] = {"Design Patterns", "1", "2", "3", "4"};

    public Iterator createIterator()
    {
        BookIterator result = new BookIterator();
        return result;
    }
}
```

```

private class BookIterator implements Iterator
{
    private int m_position;

    public boolean hasNext()
    {
        if (m_position < m_titles.length)
            return true;
        else
            return false;
    }
    public Object next()
    {
        if (this.hasNext())
            return m_titles[m_position++];
        else
            return null;
    }
}
}
}

```

Example 2: Java collection framework

Example 3: .NET collection framework

Specific problems and implementationIterator and multithreadingSeveral problems may appear when collections are added from different threads. First of all let's see which the basic steps when using an iterator are:

- Step one: the collection return a new iterator (using in our example the createIterator method). Usually this step is not affected when it is used in multithreading environments because it returns a new iterator object.
- Step two: The iterator is used for iterating through the objects. Since the iterators are different objects this step is not a problematic one in multithreading environments. It seems that the iterator does not raise special problems when a collection is used from different threads. Of course here we are talking about an "seems". To reformulate the iterator does not raise special problems when the collection used from different threads as long the collection is not changed.

Let's analyze each case:

- A new element is added to the collection (at the end). The iterator should be aware of the new size of the collection and to iterate till the end.
- A new element is added to the collection before the current element. In this case all the iterators of the collection should be aware of this. The same actions should occur when an element is removed from the collection. The iterators should be aware of the changes.

The main task when creating a multithreading iterator is to create a robust iterator (that allows insertions and deletions without affection transversal). Then the blocks which are changing or accessing resources changed by another thread have to be synchronized.

External vs. internal iterators.External Iterators - when the iteration is controlled by the collection object we say that we have an external Iterator.

In languages like .net on java it's very easy to create external iterators. In our classical implementation an external iterator is implemented. In the following example an external iterator is used: // using iterators for a clloection of String objects:

```

// using in a for loop
for (Iterator it = options.iterator(); it.hasNext(); ) {
    String name = (String)it.next();
    System.out.println(name);
}

```

```

// using in while loop
Iterator name = options.iterator();
while (name.hasNext()){
    System.out.println(name.next() );
}

```

```
// using in a for-each loop (syntax available from java 1.5 and above)
for (Object item : options)
    System.out.println(((String)item));
```

Internal Iterators - When the iterator controls it we have an internal iterator

On the other side implementing and using internal iterators is really difficult. When an internal iterator is used it means that the code is to be run is delegated to the aggregate object. For example in languages that offer support for this is easy to call internal iterators:

```
collection do: [:each | each doSomething] (Smalltalk)
```

The main idea is to pass the code to be executed to the collection. Then the collection will call internally the doSomething method on each of the components. In C++ it's possible to send the doMethod method as a pointer. In C# .NET or VB.NET it is possible to send the method as a delegate. In Java the Functor design pattern has to be used. The main idea is to create a base Interface with only one method (doSomething). Then the method will be implemented in a class which implements the interface and the class will be passed to the collection to iterate. For more details see the Functor design pattern. Who defines the traversal algorithm? The algorithm for traversing the aggregate can be implemented in the iterator or in the aggregate itself. When the traversal algorithm is defined in the aggregate, the iterator is used only to store the state of the iterator. This kind of iterator is called a cursor because it points to the current position in the aggregate.

The other option is to implement the traversal algorithm in the iterator. This option offers certain advantages and some disadvantages. For example it is easier to implement different algorithms to reuse the same iterators on different aggregates and to subclass the iterator in order to change its behavior. The main disadvantage is that the iterator will have to access internal members of the aggregate. In Java and .NET this can be done, without violating the encapsulation principle, by making the iterator an inner class of the aggregate class.

Robust Iterators- Can the aggregate be modified while a traversal is ongoing? An iterator that allows insertion and deletions without affecting the traversal and without making a copy of the aggregate is called a robust iterator. A robust iterator will make sure that when elements are added or removed from an aggregate during iteration; elements are not accessed twice or ignored.

Lets' say we don't need a robust iterator. If the aggregate can not be modified (because the iteration is started), it should be made explicitly, meaning that the client should be aware of it. We can just return a false value what an element is added to the collection stating that the operation has failed, or we can throw an exception.

An alternative solution is to add functions to change the aggregate in the iterator itself. For example we can add the following methods to our iterator: bool remove();

```
bool insertAfter();
```

```
bool insertBefore();
```

In the case when this solution is chosen the iterator handles the changes of the aggregator. In this case the operation to change the iteration should be added to the iterator interface or base class not to the implementation only, in order to have a general mechanism for the entire application. Mechanism provided by the programming language The iterator pattern can be implemented from scratch in Java or .NET, but there is already built-in support for Iterator Pattern (IEnumerator/IEnumerable in .NET and Iterator/Collection in JAVA). Hot Spot

- External vs. internal iterators - In languages like Java, C#, VB .NET, C++ is very easy to use external iterators.
- Who defines the traversal algorithm? - The aggregate can implement it or the iterator as well. Usually the algorithm is defined in the iterator.
- Robust Iterators - Can the aggregate be modified while a traversal is ongoing?
- Multithreading iterators - First of all multithreading iterators should be robust iterators. Second of all they should work in multithreading environments.