JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2006**

Vol. 5, No. 7, September-October 2006

Alternative Implementation of the C# Iterator Blocks

Horatiu Jula, Department of Computer Science, ETH Zurich, Switzerland

Abstract

In this paper, we propose an alternative implementation of the C# iterator blocks in terms of standard C# constructions. This implementation is based on syntactic sugar, so as the implementation described in the C# specification. Unlike the standard implementation, the syntactically transformed code that implements the iterator blocks is executed in a separate thread in a server-like manner by an enumerator object that handles access requests for its elements. The interface, represented by client routines such as *MoveNext()*, *Current*, *Dispose()* or *Reset()*, accesses the iterator in a thread-safe manner. The synchronization between these two layers is done using well known routines like *lock()*, *Pulse()* or *Wait()*.

1 INTRODUCTION

An *iterator block* is a block that yields an ordered sequence of values through *yield* statements. They are executed in a lazy manner, i.e. instead of returning an iterator object encapsulating all the elements yielded by the iterator block, an *enumerator object* (lazy iterator) is returned, whose execution can be outlined as follows:

- At each call of a *MoveNext()* method, it resumes its execution till its next interruption, which can be caused by an *yield* statement or by an exception occurred inside the iterator block.
- An *yield return exp* statement instantiates the current item with *exp* and then suspends the execution till the next call of *MoveNext()*, *Dispose()* or *Reset()*.
- An *yield break* statement triggers the termination of the iterator block's execution, but not before all the finally blocks associated with the enclosing try blocks are executed. If the statement is triggered by a *MoveNext()* call then, after the execution ends, *MoveNext()* returns false.
- A call of the *Dispose()* method resumes the execution and then, after all the finally blocks associated with the enclosing try blocks are executed, terminates the execution of the iterator block.
- The *Reset()* method first calls the *Dispose()* method, that causes a clean termination of the enumerator object's associated thread, then reinitializes the enumerator object, i.e. resets its parameters with their initial values and creates

Cite this article as follows: Horatiu Jula: "Alternative Implementation of the C# Iterator Blocks", in *Journal of Object Technology*, vol. 5, no. 7, September–October 2006, pp. 33-41
http://www.jot.fm/issues/issue_2006_09/article1

a new thread for its new execution. Initially, the execution of the enumerator object is suspended, awaiting a call of *MoveNext()*, *Dispose()* or *Reset()* , in order to be resumed.

Our implementation executes the syntactically transformed iterator block totally separated from the interfacing methods, i.e. in a new thread, using in a natural way common synchronization routines like *lock()*, *Pulse()* or *Wait()* , while the standard implementation executes everything in the *MoveNext()* method. Furthermore, the standard implementation transforms the iterator block in a *state machine*, i.e. into a switch-if-goto code, which necessitates a prohibitive amount of syntactic sugar when the iterator block's code becomes complicated, whereas our implementation requires a minimal amount of straightforward syntactical transformations for the iterator block. Moreover, our method allows a straightforward implementation of the *Reset()* method. This implementation's drawback is that, each new enumerator object handles its requests in a separate thread. Furthermore, the threads of the enumerator objects that didn't finish their execution remain alive in a suspended state, awaiting for a new request. But, this inconvenient can be easily surpassed by creating a pool in which the references to all the enumerator objects are kept, in order to cleanly terminate them at the end of the application, by invoking their *Dispose()* methods.

2 COMPARATIVE EXAMPLE

In this section, we present a comparative example of the syntactical transformations performed upon the iterator block in the standard implementation and those performed in our implementation proposal. In the standard implementation, the iterator block:

```
{
    for (int i = from; i >= to; i--) {
        if (i == to+ 1)
            yield break;
        yield return this.items[i];
    }
}
```

is transformed in a state-machine like code, enclosed by the enumerator object's *MoveNext()* method:

```
public bool MoveNext() {
    switch (__state) {
        case 1: goto __state1;
        case 2: goto __state2;
    }
    i = __from;
    __loop:
        if (i < __to)
            goto __state2;
        if (i == __to+ 1)
            goto __state2;
        __current = __this.items[i];
        __state = 1;
        return true;
    __state1:
```

```
        i--;
        goto __loop;
    __state2:
        __state = 2;
        return false;
}
```

whereas in our implementation proposal the iterator block is minimally transformed in a natural way, being enclosed by the *__run()* method of the iterator thread:

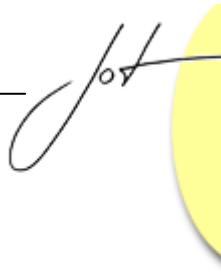
```
public void __run() {
    //enclosing lock and try-catch-finally blocks
    {
        //awaiting first request
        //guard to handle a possible dispose request
        for (int i = __from; i >= __to; i--) {
            if (i == __to+ 1)
                return;
            __return(__this.items[i]);
            //dispose request guard
        }
    }
}
```

The *__return(...)* method is explained in the fifth section.

3 THE ENUMERATOR OBJECT

The enumerator object contains the interfacing methods, the iterating thread, the iterator lock that assures thread-safe access to the iterator, the current item, the current state, and finally the attributes inherited from outside, i.e. the copies of the parameters of the method that encloses the iterator block, and the reference to *this* object. These, together with the creation and initialization of a new enumerator object are illustrated in the following code fragment:

```
      class Stack<T>: IEnumerable<T> {
        T[] items;
        ...
        public IEnumerator<T> GetEnumerator() {
          //sample iterator block
          for (int i = this.items.Length - 1; i >= 0; i--)
            yield return this.items[i];
        }
        public IEnumerator Iterate(int from, int to) {
          //the iterator block is replaced by a return of an
          //enumeration object
          return new __StackEnumerator(this, from, to);
        }

        class __StackEnumerator : //the class of the enumerator
object
          IEnumerator<T>, IEnumerator,
          IEnumerable<T>, IEnumerable, IDisposable {

          const int __before = 0; //the iteration is not started yet
          const int __inside = 1; //the last item is not reached yet
          const int __after = 2;  //the last item has been reached

          int __state; //the current
          T __current; //the current item to be returned
          Exception __exc; //the exception thrown by the iterator
block
          Stack<T> __this; //the 'this' object of the enclosing class
          //the initial values of the inherited parameters
          int __from_0;
          int __to_0;
          //the current values of the inherited parameters
          int __from;
          int __to;

          object __iterLock; //the iterator lock
          Thread __iterThread; //the iterating thread

          public __StackEnumerator(Stack<T> __this,int __from,int
__to){
            this.__this = __this;
            __from_0 = __from;
            __to_0 = __to;
            __iterLock = new object();
            __Iterators.__add(this); //the iterator is added to the
pool
            __init();
          }
          private void __init() { //initialization
            __state = __before;
            __from = __from_0;
            __to = __to_0;
            __exc = null;
            //relaunches the iterator block's execution
            __iterThread = new Thread(new ThreadStart(__run));
            __iterThread.Start();
          }
          ...
        }
      }
```



4 THE INTERFACING METHODS

In this section, we present the implementation of the interfacing methods, namely *MoveNext()*, *Current*, *Dispose()* and *Reset()*.

The property *Current* returns the current item of the iterator in a thread-safe manner, i.e. inside the iterator lock.

```
public T Current {
    get {
        lock (__iterLock) {
            return __current;
        }
    }
}
```

The *MoveNext()* method advances to the next item if there is any, otherwise it returns *false*. If an exception occurred in the iterator block, then it rethrows it.

```
public bool MoveNext() {
    lock(__iterLock) {
        if (__state == __before)
            __state = __inside;//starts the iteration
        if (__state == __inside) {
            //requests a move_next operation
            //resumes the iterator's execution
            Monitor.Pulse(__iterLock);
            //waits the effectuation of the operation
            Monitor.Wait(__iterLock);
            //if the execution triggers an exception,
            //then the exception is rethrown
            if (__exc != null) {
                  Exception __exc1 = __exc;
                  __exc = null;
                  throw __exc1;
            }
        }
        if (__state == __after)
            return false;
        return true;
    }
}
```

The *Dispose()* method terminates the iteration cycle, provoking the end of the iterator's execution and consequently the termination of its associated thread, by simply setting the iterator's state to *__after*.

```
public void Dispose() {
      lock(__iterLock) {
            if (__state == __after)
                  return;
            __state = __after;
            //requests a dispose operation
            Monitor.Pulse(__iterLock);
            //waits the effectuation of the operation

            Monitor.Wait(__iterLock);
      }
}
```

The *Reset()* method first disposes the enumerator object, the it reinitializes it.

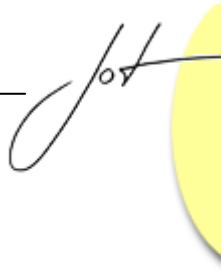
```
public void Reset() {
      lock(__iterLock) {
            this.Dispose();
            this.__init();
      }
}
```

5 THE SYNTACTICALLY TRANSFORMED ITERATOR BLOCK

The iterator block is executed by the __run() method inside the iterator thread. The iterator block's code suffers just minor modifications. The *yield return x* statement is replaced with a *__return(x)* call and the *yield break* statement is replaced with a *return* statement. The rest is just added to the code. The code is enclosed by the iterator lock due to mutual exclusion considerations, and also by a try-catch-finally block, in order to handle the possible exceptions thrown from the iterator block. Guards that handle the possible *dispose* requests are also added to the code.

```
{
    for (int i = from; i >= to; i--) {
        if (i == to+ 1)
            yield break;
        if (new Random((int)DateTime.Now.Ticks).Next(4) == 2)
            throw new Exception("exception thrown");
        yield return this.items[i];
    }
}
```

is syntactically modified as follows:



```
private void __run() {
   lock(__iterLock) {
      try {
            Monitor.Wait(__iterLock); //awaits the first request
            if (__state == __after) //dispose request guard
                  return;
            for (int i = __from; i >= __to; i--) {

                  if (i == __to+ 1)
                        return; //yield break; --> return;
                  if (new Random((int)DateTime.Now.Ticks).Next(4) ==
2)
                        throw new Exception("exception thrown");
                  //yield return x; --> __return(x);
                  __return(__this.items[i]);

                  //a dispose request might follow
                  if (__state == __after)
                        return;
            }
      }
      catch (Exception e) {
            //exception saved, in order to be rethrown by MoveNext()
            __exc = e;
      }
      finally {
            __state = __after;
            //announces the end of the execution
            Monitor.Pulse(__iterLock);
      }
   }
}
```

The *__return(T x)* method sets the current item to *x*, then signalizes the end of the move_next operation and suspends the thread's execution , waiting for a new request.

```
            private void __return(T x) {
                  __current = x;
                  //signalizes the end of move_next
                  Monitor.Pulse(__iterLock);
                  //waits for the next request
                  Monitor.Wait(__iterLock);
            }
```

6 THE POOL OF ENUMERATOR OBJECTS

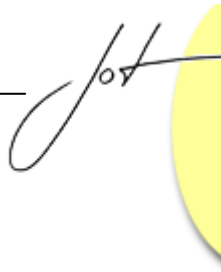
In order to cleanly terminate at the end of application all the remaining suspended enumerator objects, one collects the references to all the enumerator objects, by adding them to the pool by the time of their creation, i.e. in the constructor, by invoking *__Iterators.__add(this)*. One cleanly kills the residual enumerator objects by invoking *__Iterators.__killAll()*. The class *__Iterators*, together with these two static methods, are illustrated in the following code:

```
class __Iterators
{
      private static ArrayList __iterators;

      static __Iterators() {
            __iterators = new ArrayList();
      }

      public static void __add(IDisposable __t) {
            __iterators.Add(__t);
      }

      public static void __killAll() {
            for (int __i = 0; __i < __iterators.Count; __i++)
                  ((IDisposable)__iterators[__i]).Dispose();
      }
}
```

__Iterators.__killAll() may be invoked automatically or manually anywhere in the code. For instance, in the case of a graphical application one may insert automatically the call to *__Iterators.__killAll()* inside the *Dispose()* method of the main form.

7 CONCLUSION

The drawback of this implementation is that, each new enumerator object handles its requests in a separate thread. Moreover, the threads of the residual enumerator objects stay alive in a suspended state, awaiting new requests. But, this inconvenient can be easily surpassed by creating a pool with the references to all the enumerator objects, in order to dispose them all at the end of the application. One can easily notice that, if the calls to *MoveNext()* are dense enough (occur very quickly one after another, e.g. in a loop), then the items' retrieval is substantially slower than in the case of the standard implementation, because of the fast alternation of context switches and suspend-resume cycles, caused by the wait-notify synchronization protocol used in our approach. But, we believe that this is an extreme situation, i.e. one doesn't intend in so many situations to retrieve immediately 1000000 items from the enumerator object. This method's advantages are the easiness and the naturalness of the iterator block's syntactical transformation and the implementation of the *Reset()* method, unsupported in the standard implementation.



REFERENCES

[C#Spec2004] C# 2.0 Language Specification, MSDN 2004.

[ECMAC#2005] ECMA-334 C# Language Specification, 3rd Edition, June 2005.

About the author

Horatiu Jula research assistant at ETH Zurich, in the department of Computer Science, institute of Computer Systems, Formal Methods group. My current research interests are concerning model checking of software programs. I can be contacted at julah@inf.ethz.ch.