Adding external iterators to an existing Eiffel class library

Olivier ZENDRA, Dominique COLNET

E-mail: {zendra, colnet}@loria.fr
LORIA
UMR 7503
(INRIA - CNRS - University Henri Poincaré)
Campus Scientifique, BP 239,
54506 Vandœuvre-lès-Nancy Cedex
FRANCE

Abstract

This paper discusses common iteration schemes and highlights the interest of using explicit iterators. The advantages of external iterators are compared to those of internalized iterators.

The integration of an iterator class hierarchy to an existing library without modifying the latter is detailed. This integration brings an extra level of abstraction to the library, which thus becomes more flexible, more adapted to certain design patterns and hence can be used in a higher-level way. Such an integration is not only possible, but can even be done in an optimized way, taking into account the specific structure of the collection traversed.

A slight extension of existing class libraries can also be implemented that does not cause any compatibility problem and does not break existing code, but allows even further abstraction and makes it easier for the developer to use high-level, optimized, external iterators.

1 Introduction

Iterating over data structures and collections is unsurprisingly probably one of the most basic and common tasks of a developer. As basic as it seems, this task is not always as easy as it ought to be, especially when dealing with complex data collections and/or large libraries. As a consequence, the notion of iterators has emerged and evolved, for example in design patterns, in order to consider this technique — iteration — from a higher-level point of view.

From time to time, a flurry of discussions happens in some newsgroup or mailing-list, about the usefulness and/or efficiency of iterators, in general or in a specific implementation. Indeed, as simple as this issue may first appear, it does not always seem to be understood in all its aspects and implications. This paper is thus a report — which we intend to be useful to all developers and especially to libraries implementors — on our work designing and implementing external iterators for the pre-existing and already used library we provide with SmallEiffel, The GNU Eiffel compiler[1] [ZCC97, CZ99].

[1] `http://SmallEiffel.loria.fr`

This paper is organized as follows. Section 2 recalls the classical ways of iterating on data structures and where the usefulness of iterators appears. Section 3 explains how such iterators can be implemented and used without modifying exiting libraries. Section 4 focuses on a neat and optimized integration of iterators to existing libraries with only minor modifications to the latter. Finally, section 5 presents a overview of related implementations and section 6 concludes.

2 Classic iteration schemes without iterators

Without specific iterator mechanism, iterations on data structures may have different coding schemes according to the type of the data structure to be traversed. For example, an iteration on a STRING is different of an iteration on an ARRAY. Even more, when one has to traverse a DICTIONARY (or some kind of hash table), the iteration scheme becomes very different. In order to expose more clearly the problem, let us first present different iteration schemes. All the examples we present here use either standard Eiffel [Mey94a] classes or data structures from the SmallEiffel library (`SmallEiffel/lib_std`).

This first example is about the Eiffel STRING class and is thus not SmallEiffel specific. This classic iteration scheme is self-explanatory when one is aware that the first character of an Eiffel STRING is at index 1 :

```
string_iteration(my_string: STRING) is
    local
      i: INTEGER; element: CHARACTER;
    do
      from i := 1;    until i > my_string.count
      loop
        element := my_string.item(i);
        -- Iteration body itself.
        i := i + 1;
      end;
    end;
```

A second example deals with Eiffel ARRAYs. Since the very first index of an ARRAY is not always `1`, the iteration scheme must rely on two functions which indicate the `lower` bound and the `upper` bound. Here is an example to traverse an ARRAY of CHARACTERs :

```
array_iteration(my_array: ARRAY[CHARACTER]) is
    local
      i: INTEGER; element: CHARACTER;
    do
      from i := my_array.lower;    until i > my_array.upper
      loop
        element := my_array.item(i);
        -- Iteration body itself.
        i := i + 1;
      end;
    end;
```

The SmallEiffel library also provides a LINKED_LIST data structure. Such a linked data structure is internally very different from an ARRAY. But fortunately from the client point of view, a LINKED_LIST can be traversed exactly like an ARRAY :

```
linked_list_iteration(my_linked_list: LINKED_LIST[CHARACTER]) is
```

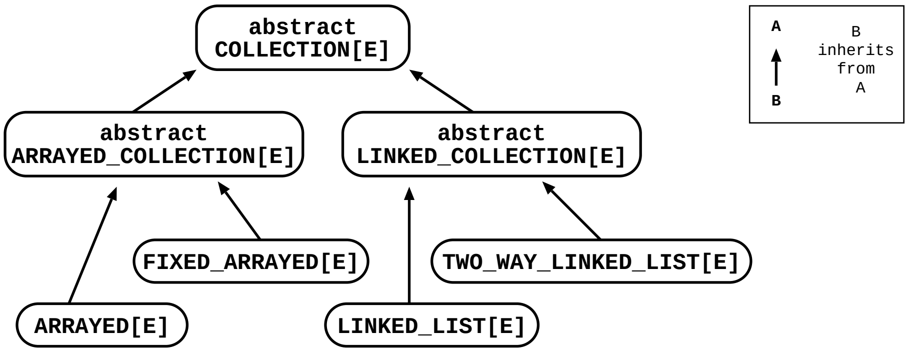

Figure 1. A part of the inheritance hierarchy from the SmallEiffel library.

```
    local
      i: INTEGER; element: CHARACTER;
    do
       from i := my_linked_list.lower;    until i > my_linked_list.upper
       loop
         element := my_linked_list.item(i);
         -- Iteration body itself.
         i := i + 1;
       end;
    end;
```

A closer look at the inheritance hierarchy of the COLLECTION cluster (figure 1) shows that the ARRAY[2] class and the LINKED_LIST class have a common abstract interface (the abstract, or `deferred` in the Eiffel terminology, COLLECTION class). As a consequence, any subclass of COLLECTION can be traversed the very same way. One can thus write an algorithm without knowing the concrete COLLECTION used:

```
collection_iteration(my_collection: COLLECTION[CHARACTER])  is
    local
      i: INTEGER; element: CHARACTER;
    do
       from i := my_collection.lower;    until i > my_collection.upper
       loop
         element := my_collection.item(i);
         -- Iteration body itself.
         i := i + 1;
       end;
    end;
```

Let us now consider a more complex problem, illustrated with the DICTIONARY class of the SmallEiffel library, which is an implementation of associative memory. Unlike COLLECTIONs, DICTIONARY indexes are not only INTEGERs, but any kind of hashable type. For example, one can use a DICTIONARY[CHARACTER,STRING] to associate an index of type STRING to a value of type CHARACTER. From the client point of view, it is important to be able to traverse all the elements in the dictionary, although no conceptual ordering exists on its elements. As an example, the DICTIONARY class is used by the SmallEiffel compiler to store

[2]Unlike C arrays, Eiffel ARRAYs are flexible. Thus, any possible operation on a LINKED_LIST can be implemented on an ARRAY. Obviously, adding an element in the middle of an ARRAY is more costly than on a LINKED_LIST

all the live routines of a class. At code generation time, the compiler has to produce code for each routine, and thus needs to traverse dictionaries. More generally, in many real-life applications, an associative memory data structure has to provide iteration facilities. As a simple example, an iteration is necessary to display the whole content of a DICTIONARY.

To fit within the COLLECTION iteration scheme, features `lower`, `upper` and `item` work exactly the same way on a DICTIONARY. The `item` function is used to access values and the `key` function is provided to access the corresponding keys (or indexes). Here is an example of a DICTIONARY traversal using both values and keys:

```
dictionary_iteration(my_dictionary: DICTIONARY[CHARACTER,STRING]) is
    local
      i: INTEGER; element: CHARACTER; key: STRING;
    do
       from i := my_dictionary.lower;    until i > my_dictionary.upper
       loop
          element := my_dictionary.item(i); --  To get the value.
          key := my_dictionary.key(i); --  To get the key.
          --  Iteration body itself.
          i := i + 1;
       end;
    end;
```

Because a DICTIONARY is not, strictly speaking, a kind of COLLECTION, there is not inheritance relationship between class DICTIONARY and class COLLECTION. As a consequence, even though the iteration scheme of a COLLECTION is exactly the same as the iteration scheme to traverse the values of a DICTIONARY, one can not share the traversal code. This fact leads us to introduce an external iterator mechanism in our library.

3 Using and implementing external iterators in Eiffel

The use of the *iterator pattern* as described in [GHJV94] solves the concerns expressed above. The key idea of this pattern (also known as *cursor*) is to take the responsibility for access and traversal out of the data structure and put it into an ITERATOR object. This way, traversing the same way any data structure or some aggregate object is very straightforward. Each iterator just has to keep track of its own traversal state and act exactly as a mediator between the user of the iterator and the traversed container.

Furthermore, as we will see later, we have shown that the iterator pattern can be added to the SmallEiffel library without any modification of the existing client code.

As described in [GHJV94], the abstract ITERATOR class is very simple (see its complete Eiffel source code in appendix A) and leads to a unique and general iteration scheme. As we are in an Eiffel context, the generic argument of the ITERATOR class represents the element type. For example, here is the unique iteration scheme with elements of type CHARACTER:

```
iterator_scheme(my_iterator: ITERATOR[CHARACTER]) is
    local
      i: INTEGER; element: CHARACTER;
    do
       from my_iterator.start;    until my_iterator.is_off
       loop
          element := my_iterator.item;
          --  Iteration body itself.
          my_iterator.next;
```

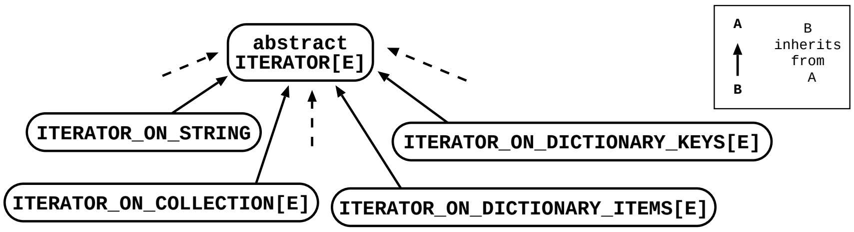

Figure 2. Inheritance hierarchy for the ITERATOR abstract class.

```
    end;
  end;
```

As one can see in the previous example, the aggregate object to be traversed is not known by the client. Furthermore, to force the client to use the iterator abstraction, there is no possible access to this object in the interface of ITERATOR (appendix A). Finally, once the iterator has been created and returned, there is no way to know about the traversed object.

Implementing ITERATOR relies on a simple inheritance mechanism (see figure 2). Class ITERATOR_ON_STRING is the implementation of ITERATOR to traverse a STRING, class ITERATOR_ON_COLLECTION traverses a COLLECTION, class ITERATOR_ON_DICTIONARY_ITEMS visits the items of a DICTIONARY, etc. As we will show later, an ITERATOR facility can be provided for any other kind of aggregate object by creating a new subclass of ITERATOR.

As the ITERATOR class is generic, the implementation of a concrete iterator using inheritance is not always as straightforward as the implementation of ITERATOR_ON_COLLECTION given in appendix B. For example, a look at the implementation of ITERATOR_ON_STRING (appendix C) shows how inheritance of a generic derivation can be used. The Eiffel type ITERATOR_ON_STRING which inherits ITERATOR[CHARACTER] is thus directly compatible to the latter.

The implementation of ITERATORs for a dictionary (appendices D and E) shows another aspect of the power of the Eiffel genericity mechanism. The type of the `dictionary` attribute is based both on a formal generic argument E and an actual type name HASHABLE.

When traversing items, using an ITERATOR_ON_DICTIONARY_ITEMS, the declaration type of the `dictionary` attribute is DICTIONARY[E,HASHABLE], where E is the formal generic argument corresponding to the item type and HASHABLE is the abstract class for all kinds of keys.

In the other case, when keys are traversed with ITERATOR_ON_DICTIONARY_KEYS the declaration type of the `dictionary` attribute is DICTIONARY[ANY,E], where E is the formal generic argument and ANY is the common abstract class for all kinds of items.

Moreover, aside abstraction, a great advantage of *external* iterators is that, since the traversal algorithm is held in the iterator, it is very easy to switch from one traversal policy to another for a given data structure just by using a different kind of iterator.

4 Integrating external iterators to existing libraries

As we have seen previously, ITERATORs can be added in a completely external way without any modification of the existing classes which are to be traversed. Because when creating an iterator it is tedious to explicitly select the appropriate implementation (IT-

ERATOR_ON_STRING, ITERATOR_ON_COLLECTION, etc.), some minor modifications are worth considering.

Indeed, it is possible to move the burden of the creation of the correct iterator from the client — the (application) developer — to the supplier — the library (developer) — by simply adding an iterator creation function in the aggregate to be traversed. This is easily done, by putting a generic version of this function in a common parent of traversable classes and defining it more specifically in concrete descendants. As an example, here is this function as it appears in the class COLLECTION:

```
get_new_iterator: ITERATOR_ON_COLLECTION[E]  is
    do
       !!Result.make(Current);
    end;
```

This way, only the library (implementor) has to worry about the type of the iterator on a specific class; the client only calls the iterator creation function of the class she wants to iterate on and gets a ready-to-use iterator which can be relied on without even knowing its exact type.

Note that the `get_new_iterator` routine above returns a new iterator each time it is called. As a consequence, since each iterator holds its own index (here `item_index`) on the aggregate structure, it is possible to have several iterations at the same time on the same data structure that operate in a completely independent and safe way.

A very important aspect of this technique is that it is fully compatible with legacy code without extra modifications. There is thus absolutely no impact on existing code and applications[3], which makes this technique to increase the expressiveness and ease-of-use of libraries quite attractive.

A second kind of modifications worth doing on an existing library pertains to optimizations. The iterator being created by the aggregate it traverses, it is possible to tie the two classes more tightly, taking advantage of the knowledge of the aggregate specific implementation to optimize its iterator. We will highlight the usefulness of such an optimization on an example involving several iterators used to traverse a same LINKED_LIST at the same time.

First, let us consider how LINKED_LIST list was implemented in SmallEiffel, independently of any iterator (see figure 3). Basically, a LINKED_LIST[E] is composed of cells (of type LINK[E]) holding both a value (`item`) and a reference to the next cell (`next`). Finding an element in the list is done by simply following the references from the first cell to the desired one. To access the element of rank n, the average cost is a function of n. Of course, in case there is an iteration (without iterator) on the n first elements of the list, simply doing this would obviously be unreasonable, since one would have to restart from the first element of the list for each access, the average cost being $1+2+3+...+n = n*((n+1)/2) = (n^2+n)/2$. Consequently, a caching mechanism was added to LINKED_LIST, in order to memorize the last cell accessed. This way, when the next item is accessed, for example in an iteration on the list, the only cost is dereferencing the `next` field of the current cell (and of course its `item` attribute as well), which has a constant cost of 1. Thus the cost of an iteration on the n first elements of a list would be only proportional to n, not n^2.

This works very well when there is only one iteration on the list, but not when two or more interleaved iterations are performed, because each iteration invalidates the cell memorized

[3]Unless the new name introduced for the iterator creation function is already used by another feature, which, in Eiffel, results in a name clash.

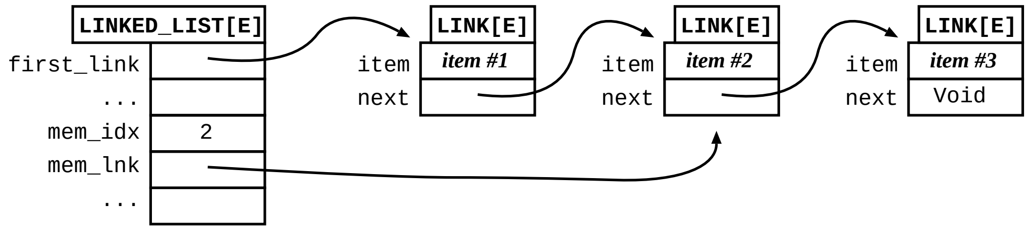

Figure 3. Implementation of a linked list with an access caching mechanism.

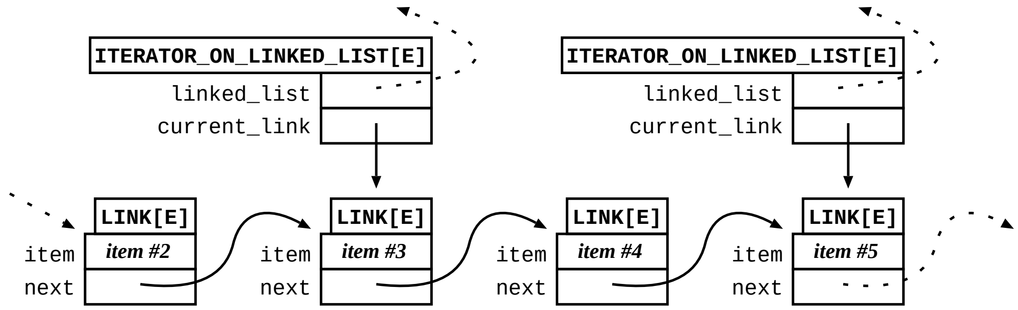

Figure 4. Duplicating the linked list access cache into each iterator objet.

at the previous one. In this case, it would be very impractical to add several caches in the list itself, because their number is not known in advance. It is however much neater and much more efficient to perform these interleaved iterations by using several iterators that can be easily optimized.

Indeed, the caching mechanism can be added to the iterator itself to directly "point to" the last cell accessed in the list (see figure 4). This way, each iterator can have its own private cache and any number of iterations can be performed independently on the list without hampering each others, thus allowing a much better scalability. In this case, the iteration is conceptually performed "inside" the iterator, instead of being performed in the LINKED_LIST. Appendix F shows the code for the ITERATOR_ON_LINKED_LIST and its internal cache, attribute `current_link`, used to reference the last cell accessed. Note that, unlike the previous iterator examples, the cache held by ITERATOR_ON_LINKED_LIST is not of type INTEGER, since it does not keep track of the index of the last element accessed. It is a LINK[E], that is directly the internal representation of a cell in a LINKED_LIST.

This requires another slight change in the aggregate class, since it now must expose part of its internal representation to the iterator. Thanks to the very flexible and powerful export rules of the Eiffel language, this can be easily and safely done. The code of class LINKED_LIST must be changed from:

```
feature {LINKED_LIST}
  first_link: LINK[E]; -- Void when empty or gives access to the first element.
```

to the following:

```
feature {LINKED_LIST,ITERATOR_ON_LINKED_LIST}
  first_link: LINK[E]; -- Void when empty or gives access to the first element.
```

The selective export mechanism of Eiffel made it possible not to make `first_link` public, which would completely break encapsulation and information hiding rules, but to simply export it to the only class that did need access to it, namely ITERATOR_ON_LINKED_LIST.

It thus appears easy to integrate iterators to an existing library, in an efficient way, and without breaking legacy code.

5 Other iterator implementations

The current ISE EiffelBase library [Eng99], based on [Mey94b], offers many ways to access data collections, among which are internalized iterators. Collections that can be iterated on — in a nutshell, various kinds of lists — inherit from CHAIN, which provides iteration routines as well as an internalized cursor, the latter inherited from CURSOR_STRUCTURE. Since the iterator is internalized, it is not possible to have several iterations using this mechanism at the same time on the same structure.

The EiffelGore [Gor96] library also uses internalized iterators, or cursors. Each traversable collection inherits the necessary features from common predefined ancestors, such as LIST_LINKED and LIST_ARRAY. Here, again, since the iterator is internal, it is not possible to have several iterations using this mechanism at the same time on the same structure.

The Gobo Eiffel Structure [Bez97] library strongly relies on iterators for traversal. The DS_CURSOR class is more or less equivalent of our ITERATOR class, but the traversed container is supposed to be accessible via the `container` feature. All traversable collections have to inherit DS_TRAVERSABLE, which basically adds cursor-awareness. Each descendant of DS_TRAVERSABLE provides its own specialized descendant of DS_CURSOR. But although these descendants offer iteration routines for the collection class they correspond to, there seems to be no such routine at the DS_CURSOR level, the most abstract one. Consequently, it seems impossible to iterate in an abstract way on collections, which in our opinion defeats most of the purpose of having iterators in the first place.

The Pylon library [Arn97] uses a kind of external iterators which is similar to ours. All traversable collections inherit, directly or not, from P_TRAVERSABLE, which basically provides an equivalent to our `get_new_iterator` routine. Of course, specialized descendants also provide specialized iterators wherever needed. Pylon's are read-write iterators, that is they encapsulate all the operations that can be done on the collection they iterate on. Ours, on the contrary are for the time being read-only, that is limited to accessing the items.

Pylon's iterators also try to address the problem of changes to the collection structure (item removal, for example) while iterators are active on it. Two mechanisms are used (protecting and locking) which affect either the whole structure or only the item to which an iterator points. This system has thus made the decision to trade some efficiency for some more safety. Conversely, our iterator library focuses on efficiency, especially because we do not want iterators to have a negative impact on performance even when they are not used. Another effect of these opposite choices is that creating new traversable classes and/or iterators, and using them, is much simpler in our library than with Pylon iterators. Indeed, with Pylon, one has to check the whole source code of a collection class for which an iterator is developed, in order to be sure the protection mechanisms are correctly implemented. In our system, changes are much more limited. This difference probably corresponds to the fact the Pylon was designed with iterators from the beginning, whereas we *added* them to an existing library.

Our COLLECTION class is inspired by its Smalltalk [GR83] counterpart, which leads us to a similar iteration scheme. Of course, in Smalltalk, iterations are supported directly by the language keyword syntax.

In Java [JGS96], the collection framework features powerful iterators, based on `Enumerations` and the inner class language mechanism. The latter allows a collection to provide its own iterator without exposing its representation.

6 Conclusion

We have shown how iterators could be very easily added in a completely external way to pre-existing libraries (section 3). Although the examples we used are taken from the SmallEiffel library, the very same conclusions apply to other libraries written in Eiffel. Such additions increase the level of abstraction on data collections in a significant way and can be done with no modification of the existing libraries.

In order to have a safer and faster behavior, it is possible to integrate more closely the iterators and the collections they work on (section 4). This can be easily and efficiently done thanks to some mechanisms of the Eiffel language, especially the powerful selective export.

As the ITERATOR abstract class we presented is very simple, a developer who wants to write portable code among various Eiffel libraries may consider using such a similar abstraction. Furthermore, the same abstraction may be applied to some more broader definitions of iteration, for example to traverse files.

Of course, some tradeoffs had to be found, especially considering the problem of modifications on objects were iterations are taking place. We have chosen to focus on a simple and efficient solution for iterators; other works have made different choices (section 5). As a consequence, we think that finding a solution that would address both the efficiency and the security problem is optimal ways still has to be found and represents future work.

A Source code of the abstract ITERATOR class

```
deferred class ITERATOR[E]
-- The iterator pattern at work: this abstract class defines a
-- traversal interface for any kind of aggregate data structure.
feature
  start is
        -- Positions the iterator to the first object in the
        -- aggregate to be traversed.
      deferred end;
  item: E is
        -- Returns the object at the current position in the sequence.
      require
        not is_off
      deferred end;
  next is
        -- Positions the iterator to the next object in the sequence.
      require
        not is_off
      deferred end;
  is_off: BOOLEAN is
        -- Returns true when there are no more objects in the sequence.
      deferred end;
```

```
end
```

B Implementation of ITERATOR for COLLECTIONs

```
class ITERATOR_ON_COLLECTION[E] inherit ITERATOR[E];
creation make
feature {NONE}
  collection: COLLECTION[E]; -- The collection to traverse.
  item_index: INTEGER; -- Memorizes the current item position.
feature
  make(c: COLLECTION[E]) is
      require
        c /= Void
      do
        collection := c;
        item_index := collection.lower;
      end;
  start is
      do item_index := collection.lower; end;
  item: E is
      do Result := collection.item(item_index); end;
  next is
      do item_index := item_index + 1; end;
  is_off: BOOLEAN is
      do Result := not collection.valid_index(item_index); end;
end
```

C Implementation of ITERATOR for STRINGs

```
class ITERATOR_ON_STRING inherit ITERATOR[CHARACTER];
creation make
feature {NONE}
  string: STRING; -- The string to traverse.
  item_index: INTEGER; -- Memorizes the current character position.
feature
  make(s: STRING) is
      require
        s /= Void
      do
        string := s;
        item_index := 1;
      end;
  start is
      do item_index := 1; end;
  item: CHARACTER is
      do Result := string.item(item_index); end;
  next is
      do item_index := item_index + 1; end;
  is_off: BOOLEAN is
      do Result :=  item_index > string.count; end;
end
```

D Implementation of ITERATOR for DICTIONARY items.

```
class ITERATOR_ON_DICTIONARY_ITEMS[E] inherit ITERATOR[E];
creation make
feature {NONE}
  dictionary: DICTIONARY[E,HASHABLE]; -- The dictionary to traverse.
  item_index: INTEGER; -- Memorizes the current item position.
feature
  make(d: DICTIONARY[E,HASHABLE]) is
      require
        d /= Void
      do
        dictionary := d;
        item_index := 1;
      end;
  start is
      do item_index := 1; end;
  item: E is
      do Result := dictionary.item(item_index); end;
  next is
      do item_index := item_index + 1; end;
  is_off: BOOLEAN is
      do Result := item_index > dictionary.count; end;
end
```

E Implementation of ITERATOR for DICTIONARY keys.

```
class ITERATOR_ON_DICTIONARY_KEYS[E] inherit ITERATOR[E];
creation make
feature {NONE}
  dictionary: DICTIONARY[ANY,E]; -- The dictionary to traverse.
  item_index: INTEGER; -- Memorizes the current key position.
feature
  make(d: DICTIONARY[ANY,E]) is
      require
        d /= Void
      do
        dictionary := d;
        item_index := 1;
      end;
  start is
      do item_index := 1; end;
  item: E is
      do Result := dictionary.key(item_index); end;
  next is
      do item_index := item_index + 1; end;
  is_off: BOOLEAN is
      do Result := item_index > dictionary.count; end;
end
```

F Implementation of ITERATOR for LINKED_LISTS.

```
class ITERATOR_ON_LINKED_LIST[E] inherit ITERATOR[E];
creation make
feature {NONE}
  linked_list: LINKED_LIST[E]; -- The list to traverse.
  current_link: LINK[E]; -- Memorize the current item position.
feature
  make(ll: LINKED_LIST[E]) is
      require
        ll /= Void
      do
        linked_list := ll;
        current_link := linked_list.first_link;
      ensure
        linked_list = ll
      end;
  start is
      do current_link := linked_list.first_link; end;
  is_off: BOOLEAN is
      do Result := current_link = Void; end;
  item: E is
      do Result := current_link.item; end;
  next is
      do current_link := current_link.next; end;
end
```

References

[Arn97] Franck Arnaud. Pylon: a foundation library. URL: http://www.altsoft.demon.co.uk/doc/pylon.html, 1997.

[Bez97] Eric Bezault. Gobo Eiffel Structure Library. URL: http://gobosoft.com/eiffel/gobo/structure/index.html, 1997.

[CZ99] Dominique Colnet and Olivier Zendra. Optimizations of Eiffel programs: SmallEiffel, The GNU Eiffel Compiler. In *29th conference on Technology of Object-Oriented Languages and Systems (TOOLS Europe'99)*, pages 341–350. IEEE Computer Society, June 1999.

[Eng99] Interactive Software Engineering. EiffelBase library. URL: http://eiffel.com/products/base/classes/base.html, 1999.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison Wesley, 1994.

[Gor96] Jacob Gore. *Object Structures : Building Object-Oriented Software Components With Eiffel.* Eiffel in Practice Series. Addison Wesley, 1996.

[GR83] A. Goldberg and D. Robson. *Smalltalk-80, the Language and its Implementation.* Addison Wesley, 1983.

[JGS96] Bill Joy James Gosling and Guy Steele. *The Java Language Specification.* Addison Wesley, 1996.

[Mey94a] Bertrand Meyer. *Eiffel, The Language.* Prentice Hall, 1994.

[Mey94b] Bertrand Meyer. *Reusable Software : The Base Object-Oriented Component Libraries.* Prentice Hall, August 1994.

[ZCC97] Olivier Zendra, Dominique Colnet, and Suzanne Collin. Efficient Dynamic Dispatch without Virtual Function Tables. The SmallEiffel Compiler. In *12th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'97)*, volume 32, pages 125–141, 1997.