Robust Iterators in ET++

Thomas Kofler

UBILAB Union Bank of Switzerland
Bahnhofsstr. 45
CH-8021 Zurich

e-mail: kofler@ZH010.ubs.ubs.arcom.ch

June 1992

Abstract

Container classes and iterators operating on them are a common feature of object-oriented class libaries. Most often, the question whether modifications of a container during an iteration should be allowed, is answered with no. This work, in contrast, justifies why it should be allowed and supported, at least in comprehensive C++ class libraries like ET++. It is further shown how the concept of a robust iterator can be reasonably defined and implemented for well-kown data structures. In this course, special attention is paid to hashing algorithms, in particular linear probing. Feasible and efficient solutions are described and evaluated.

Keywords: object-oriented programming, C++, ET++, class library, framework, container, collection, iterator, robust iterator, hashing, linear probing

1 Introduction

Many object-oriented class libraries have a container concept, and often also an iterator concept. ET++, a portable application framework written in C++, is no exception. ET++ provides so-called robust iterators that allow modification of the underlying container during an iteration in a consistent and well-defined way. Beside ET++, this outstanding feature is offered by the commercial C++ libraries MacApp 3.0 [Apple92], which has been recently released, and by Container 2 [Glocken90].

Up to version 2.2 of ET++, robust iterators have been limited to removals, but the version 3.0 being currently in preparation to be released support insertions as well. How containers and iterators are defined and implemented in both version is the main subject of this report. An answer why robust iterators are considered important is given. Efficiency is also addressed, because "collections are heavily used system-level classes ..." [Cox87:146].

ET++ is a single-rooted class library with the universal class called Object[1], and does not use multiple inheritance. More material on ET++ can be found in publications by its developpers [Gamma89, Gamma91, Weinand88, Weinand89, Weinand91]. Further work is referenced in the text.

This project started with an attempt to develop graph classes which are suitable to build a graph editor framework for ET++. Since the ET++ container classes provided robust iterators, the question arose whether robust iterators are also possible for graphs. In the ET++ container classes, the problem of simultaneous iteration and insertions was unsolved, however. Since I wanted to use some of the container classes as building blocks, I evaluated design and implementation of these classes and came up with an idea that also allows for insertions. So, the following goals were established:

- Refine the idea and develop an efficient solution for the ET++ container classes. As an important constraint, the existing client interfaces should not change whenever possible, and existing code should not be broken.

---

1 All classes derived from Object are called regular. A regular object is instance of a regular class.

- Implement the solution.
- Since the implementation would require considerable changes in the implementation of the container classes, repair wrong inheritance relationships and inconsistent interfaces. Also improve efficiency when possible.
- Test the collection classes thoroughly and check what impact the new classes have on client code. Compare the performance between the old classes and the new classes.

This work emphasizes on the conceptual part. In the last chapter, there is a short report on the other goals.

2 Overview on ET++ Containers

This chapter gives an overview on the container classes and related concepts found in ET++. Some terminology used in the sequel is also introduced. This chapter is intended for a reader not familiar with ET++.

2.1 General Approach

There are different general approaches for building container classes in a language, as C++, with static type checking. Heterogenous containers inhibit from static type checking, whereas type-specific and generic homogenous containers do not. Homogenous containers can be further divided into type-specific and generic containers.

A *heterogenous* container relies on some protocol which all element classes inherit from a common base class. In ET++, the operations of the container classes declare the element type as a pointer to the class Object, "the least common divisor". As a consequence, the more derived type of an element is lost and has to be recovered when retrieving an element. Furthermore, an ET++ container cannot deal neither with the basic types of C++ (like int, char, float) nor with classes not derived from Object. Examples for such non-regular classes are Point or Rectangle.

Another possibility is to build a *type-specific* container class whose interface declares the desired base type of the elements. Depending on the design of the container classes found in the library, the type-specific container reuses them by means of inheritance or composition. The library classes are especially designed for this purpose in order to minimize the additionally needed code. This approach is employed by the Container 2.0 library, and by the container classes in [Budd91], for instance.

When the language supports genericity, *generic* container classes take advantage of this feature. The element type is not specified, but a formal argument. When instantiating a generic class, the element type is passed as an actual argument. C++ calls its genericity construct *template*. ET++ does not use templates to stay independent from compiler versions.

2.1.1 Object Equality and Identity

Some of Object's operations are essential for the container classes. Determining equality of two objects is done by the virtual operation IsEqual. This concept is called *object equality*. In contrast, there is no operation for *object identity*: Two objects are considered identical if their addresses are equal. The virtual Compare operation returns an integer describing the ordering between the two involved objects. Object also introduces the Hash operation which returns an unsigned long integer. This value is used by containers whose implementation is based on a hashing algorithm. The class Date in example 2.1 overrides these three methods.

```
class Date : public Object {
public:
   Date(u_long aDayNumber) { dayNumber= aDayNumber; };
   u_long Hash() {...};                    // overridden
   bool IsEqual(Object *anObject);         // --- " ----
   int Compare(Object *anObject);          // --- " ----
   int DayOfWeek() {...};
   void PrintAsStringOn(ostream *file) {...};
protected:
   u_long dayNumber;
};
```

Ex. 2.1 The Example Class Date Overrides the Methods of Object's Comparison Protocol

Example 2.2a shows an implementation for IsEqual, example 2.2b one for Compare.

```
bool Date::IsEqual(Object *anObject)
{
   return   (anObject->IsKindOf(Date)) &&
            (this->DayNumber() == ((Date *) anObject)->DayNumber())
}
```

Ex. 2.2b An Example Implementation of IsEqual

```
int Date::Compare(Object *anObject)
{
   Date *aDate= Guard(anObject, Date);
   return (aDate->DayNumber()) - (this->DayNumber());
}
```

Ex. 2.2b An Example Implementation of Compare

2.1.2 Class Descriptors and Type-Safe Down Casts

Since C++ provides no type information at run time, ET++ defines a class descriptor concept. For each class, there is a descriptor which is an instance of the class Class. Every object can be asked for its class descriptor by the operation IsA. If two objects are of the same class, they refer to the same class descriptor. Further, it is possible to ask an object whether it is instance of a certain class by the operation IsKindOf.

```
Object *op= ...; Object *qp= ...;
if ( op->IsA() == qp->IsA() )
   ...;  // *op and *qp are members of the same class
if ( op->IsKindOf(qp->IsA()) )
   ...;  // *op is instance of *qp's class
```

Ex. 2.3 Using Class Descriptors

The Guard macro is used for type-safe down casts. It allows to safely recover a more derived type for a pointer value. Example 2.4 shows the usage and the semantics of the Guard construct. Type-safe down casts are particularly useful if the container classes, as in ET++, are heterogenous.

```
Object *op= new Date(666); Date *aDate;
aDate= Guard(op, Date);    // type-safe down cast using Guard

if ( op == 0 || (op != 0 && op->IsKindOf(Meta(Date))) )
   aDate= (Date *) op;
else
   Error("not a Date");    // run-time error
```

Ex. 2.4 Type-Safe Down Cast with Guard and Equivalent Explicit Formulation

2.2 Hierarchy and Functionality

The container classes of ET++ are very similar to those in the NIHCL[2] library [Gorlen90, Gorlen87]. Both C++ libraries were inspired by Smalltalk-80 [Goldberg83]. However, the hierarchy of version 2.2 is closer to Smalltalk's hierarchy than that of version 3.0. See also Appendix A for the container class hierarchies and Appendix B for the interface declarations of the abstract class Collection.

SeqCollection models sequenceable collections that have a well-defined sequence order. This implies that every element can be accessed by an index. OrdCollection is an array-based implementation, and ObjList is an implementation using doubly-linked lists. The list nodes are instances of the class ObjLink. The class SortedObjList sorts its elements by immediately inserting an element at the right place.

The class Set is a set-like collection. There cannot be two elements in a Set that are equal in the sense of object equality. Set's behaviour is implemented by linear probing, a variant of open-address hashing. The class IdSet (IdentitySet) explicitly uses object identity instead of object equality.

For the class Dictionary which is a Set of Associations in version 2.2, a corresponding variant IdDictionary (IdentityDictionary) is provided. In version 3.0, the client interface of Dictionary was slightly changed and is now derived from the newly introduced abstract class Container. See 6.1 for more details.

2.2.1 Inquiries

The inquiry operations provided by a collection consists of:

- determining the number of elements (Size)
- test for inclusion (Find and FindPtr),
- counting the number of occurrences (OccurrencesOf, OccurrencesOfPtr)
- internal iterators (Detect, Select, Collect)
- creation of external iterators (MakeIterator, MakeReversedIterator)

Some of the operations have two variants. The trailing 'Ptr' in the name indicates that the operation uses identity instead of equality. This distinction is important because there may be elements in a container which are equal but not identical.

2.2.2 Modifications

The Add operation inserts an object. An element is removed by either the Remove or the RemovePtr operation. Here again, Remove uses object equality to locate the element, and RemovePtr uses object identity. All modification operations indicate their success or failure with their return value. The Set's Add operation will fail, for instance, if the object to be inserted is

2 Formerly known as OOPS library.

already in the Set. A Remove invocation will fail, if the specified object is not element of the collection.

For sequencable collections, there are further operations like AddAt or RemoveAt. AddAt allows the insertion of an element at a specified index, and RemoveAt removes the element at a specified index.

2.2.3 Internal Iterators

The internal iterators Collect, Select and Detect are similar to those of the Smalltalk Collection classes. The actions to be executed for each element are specified in a function of appropriate type. This function which is the body of the iteration (*iteration body*) is passed when invoking one of the internal iteration operations.

How internal iterators are used is shown in example 2.5 for Collect. Collect returns a collection of the same type as the receiver: the iteration body is applied to each element of the receiver, and its value becomes an element of the result.

```
Object *Defer(Object *, Object *element, void *arguments)
{
   // the first argument is the receiver collection: ignored here
   int deferment= *(int *) arguments;     // no type check possible
   Date *aDate= Guard(element, Date);
   return new Date(aDate->DayNumber() + deferment);
}

Collection *meetings= ...;    // a Collection of Dates
int deferment= 7;
Collection *deferredMeetings= meetings->Collect(Defer, &deferment);
```

Ex. 2.5 The Internal Iterator Collect

2.2.4 External Iterators

External iterators are class objects of their own. The most important operation *yield* is declared as Object *Iterator::operator()(). On each invocation, it returns (or yields) another element of the collection. If there are no more elements, null is returned. An iterator can be reset by Reset, so another enumeration can be done.

```
void PrintAgendaOn(Collection *meetings, ostream *file)
{
   Iterator *next;
   Date *aDate;

   next= meetings->MakeIterator();      // creates a new iterator
   while ( aDate= Guard((*next)(), Date) )
      aDate->PrintAsStringOn(file);
   delete next;                         // iterator has to be deleted
}
```

Ex. 2.6 Using an External Iterator

2.2.5 More Convenient Iterators

The only purpose of the two constructs presented next is to provide some syntactic sugar. Despite their simplicity, they are important and heavily used.

2.2.5.1 The Iter Construct

There is no need to delete iterators explicitly when using the Iter construct. Especially functions and methods with several exit points profit from Iter. It lessens the potential of bugs and makes the code more compact. Example 2.7 shows code which is equivalent to example 2.6. Iter takes advantage of compiler-generated destruction of automatic variables, and of inlining.

```
void PrintAgendaOn(Collection *meetings, ostream *file)
{
   Iter next(meetings);
   Date *aDate;

   while ( aDate= Guard(next(), Date) )   // operator() is inline
      aDate->PrintAsStringOn(file);
   // destructor of Iter executes here and deletes the iterator object
}
```

Ex. 2.7 The Convenience Construct Iter

2.2.5.2 The ForEach construct

It is very common to invoke an operation on all elements of a container. Using an internal iterator is awkward in C++, and bulky code results. As a solution, there is the ForEach macro. Syntactically, its usage looks like an the invokation of an internal iterator, but the macro eventually uses an external iterator. The next example shows how ForEach is used:

```
void PrintAgendaOn(Collection *meetings, ostream *file)
{
   meetings->ForEach(Date,PrintAsStringOn)(file);
}
```

Ex. 2.8 The Convenience Construct ForEach

2.3 Comments on Internal and External Iterators

The notion of an external iterator emphasizes that the driving loop is external to the iterator and is executed in the context of the client code. Consequently, an internal iterator hosts and hides the driving loop. The context of the client code has to be explicitly built and passed in a language like C++ that does not allow for lexical closures[3]. In Smalltalk, this is a feature of the base language, so internal iterators are easy to use there.

Booch [Booch87] introduces the term active instead of external, and passive instead of internal, respectively. Speaking of an active iterator has an other meaning in this work (see chapter 4.1).

External iterators are more flexible and powerful than internal iterators. There are algorithms that cannot be formulated by means of internal iterators. A good example is the two-way merge [Berztiss88]. Due to the somewhat cumbersome usage of internal iterators in C++, external iterators are the abstraction of choice.

The question about robust iteration arises for internal and external iterators. Internal iterators are best implemented by means of external iterators if external iterators are available. If they are robust, the internal iterators will automatically be robust as well.

---

3 Breuel [Breuel88] proposes, amongst others, to support lexical closures in C++.

3 Are Robust Iterators needed?

In the literature, many authors briefly address this question. Except for the developpers of ET++ [Weinand89] which coined the term, none of them states that there is a need for robust iterators. Sometimes, it is suggested to iterate through a copy of the container to be modified when the problem arises. MacApp 3.0 and Container 2 also provide robust iterators, but no argument is made. See chapter 6 for more.

In this chapter, an example taken from ET++ is presented first to illustrate the problem. Then, the problem is abstracted. A discussion follows where I argue that robust iterators are a necessity.

3.1 A Concrete Example Illustrating the Problem

In ET++, a running application is represented by a single global object (an instance of the class Application). As most important functions, the application object handles user events that cannot reasonably assigned to another event handling object in the system. Second, the application object maintains a collection of open documents. Each of them maintains a list of its associated windows.

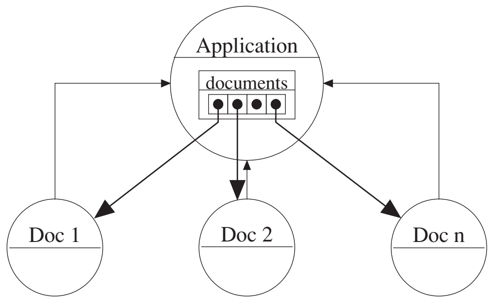

Fig. 3.1 Application, Document and Other Objects Form a Network

When the user selects "Quit", the application object invokes the Close operation of all documents. If the document is really closed it asks the application to remove itself. Example 3.1a and 3.1b describe the interaction in C++.

```
void Application::Quit()
{
   Document *doc;
   bool allClosed= true;
   Iter next(this->documents);

   while ( doc= (Document *) next() )
      allClosed= allClosed && doc->Close();
   if ( allClosed )
      this->Terminate();
}

Document *Application::RemoveDocument(Document *doc)
{
   return (Document *) documents->Remove(doc);
}
```

Ex. 3.1a Application's Quit and RemoveDocument Operation

Document's Close operation can also be invoked when only one document is to be closed (The invoking object is typically an associated window of the document in this case). The Close method as shown works for both cases. If there were no robust iterators, it would not work when invoked from the Quit operation. This is because the Close operation removes an element from the container documents while there is an iterator active on it.

```
bool Document::Close()
{
   bool canClose;

   canClose= true;
   if ( this->IsModified() )
      canClose= this->AskUser();
   if ( canClose )
      this->application->RemoveDocument(this);
   return canClose;
}
```

Ex. 3.1b Document's Close Operation

3.2 Networks of Interacting Objects

In an object-oriented application, many objects refer to each other. If the number of objects referred to by some object varies over time, the referring object (Application)[4] stores the references in some data structure which is essentially a container. The referring object often provides operations that affect the set of referenced objects (RemoveDocument), and the referring object often has to iterate through the set of references (Quit) and invokes some operation (Close) on a referenced object. This referenced object (Document), in turn, may invoke some operations that directly or indirectly modify the set of references in the referring object: Iteration and modification occur simultaneously!

This is typical for an object-oriented application. Its objects form a complicated network, and they interact in various ways. The example in 3.1 stands for a common arrangement: Passing an event to a set of event handlers eventually leads to the removal of the handler which reacted to the event. In ET++, this and similar situations occur in several places.

3.3 Advantages of Robust Iterators

The problem of simultaneous iteration and modification can be solved either implicitly by the provision of some robust iteration device, or ad-hoc where needed. All implicit variations discussed in this work can be used ad-hoc, of course. Coding one of them will require substantial effort until it works. Additionally, code duplication results when there are several places that need such special treatment. It is clear that ad-hoc solutions are least desired and should hence be avoided.

But first, a supplier of a framework has to find the actual and potential problems which the client is likely to meet because of simultaneous iteration and modification. When done systematically, this means to conduct a number of proves (even if this is accomplished in a rather informal way). Except for trivial cases, this is impossible in practice. After the supplier has found some or all potential problems, he must either solve or document them.

A framework is made to be customized. Even if some statement of correctness were possible, it had limited value if the critical parts are not hidden or protected from the client. Due to the design, it may be impossible to protect or to hide them. The framework may also be free of such problems until the client customizes it. Then, he or she has to find the reason and a solution.

---

4 In this paragraph, the names in parentheses refer to the example in 3.1.

An object-oriented framework like ET++ is large and comprises of complex interaction patterns. These patterns can also vary over time because the network formed by the interacting objects is often highly dynamic. The issue of robust iterators is therefore not trivial at all.

Experience with ET++ has shown that removals during iterations are common, but interestingly, insertions during iterations are relatively rare. Thus, iterators that are at least robust with respect to removals have the following essential advantages:

- The supplier of a framework with complex object interactions can save a lot of difficult reasoning during design and implementation.
- There is no incentive to design and code around the problem: both supplier and client can concentrate on more urgent things.
- Code duplication is avoided.
- A client less often faces the situation that an innocent looking piece of code does not work because it causes unforeseen simultaneous iteration and modification in the framework.

4 Robust Iterators - Definitions and Implementations

This chapter shows how robust iterators are defined and implemented in the old version and the new version for the data structures behind the classes OrdCollection, ObjList and Set. I call the approach used in version 2.2 the *place holder approach*, and the approach used in version 3.0 the *adjustment approach*.

Both the place holder and the adjustment approach rely on the fact that a container registers its active iterators. The need for registration in conjunction with the semantics of the Iterator interface lead to a particular state-transition model in version 2.2. This model has been retained for the new version, so it is presented before the detailed description of the two approaches. Finally, it is shown that copying the container or delaying operations is not safe.

4.1 Registering Robust Iterators and State-Transition Model

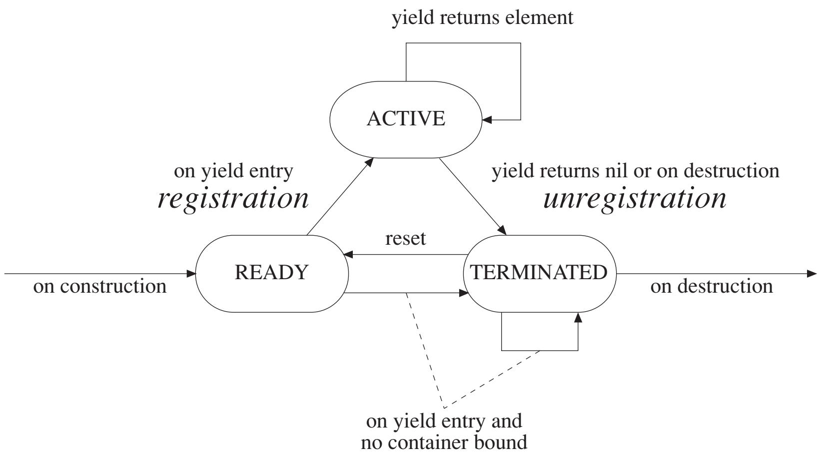

Fig. 4.1 State-Transition Model of Iterators

An iterator is either READY, ACTIVE or TERMINATED. After an iterator is created, it is READY. Upon the first yield request, it gets ACTIVE unless - as an exceptional case - the iterator is not *bound* to the container because the latter has been deleted meanwhile. During this transition, the registration at its underlying container takes place. The iterator remains active until it can no more yield another object. During the following transition to TERMINATED, the iterator is unregistered. See figure 4.1 for a graphical notation of this state-transition model. Note that an iterator can run through all three states at one yield request.

The state model, registration, and unregistration are totally transparent to the client. The registration/unregistration mechanism is realized at the abstract base classes Container[5] and Iterator, so all derivations inherit the mechanism.

Is such a complicated model needed? The answer is yes. The reason behind it has two important aspects. First, the yield operation should be applicable arbitrarily many times without returning an object once a yield operation returned null. So, at least two states are needed, namely ACTIVE and TERMINATED. As a second reason, the time of registration or unregistration should be independent of construction or destruction.

4.2 Preincrement and Postincrement Iteration

For each pass of an iteration, an iterator principally performs three actions:

a) It computes the subsequent value of the induction variable. For obvious reasons, it is called the *increment operation*.
c) It tests for the end of sequence (the *EOS operation*).
b) It computes the next element to be yielded (the *get operation*).

The results of the get and the EOS operation are returned by the yield operation. If the increment operation is executed before the get operation, the yield operation is preincrement. Analogously, the opposite is a postincrement yield operation.

Further, yielding can be defined as terminating the iterator either *early* or *late*. Early termination means that the iterator terminates during the yielding of the last element whereas late termination takes place when the container has no more elements to be yielded at that moment. An analogous distinction can be made for the initialization of an iterator, but this is irrelevant for the iterator concept presented here.

4.3 Version 2.2 - The Place Holder Approach

This approach is a variation of *lazy removal*[6] which is often used. If an element is to be removed and there are currently no active iterators, the element is removed such that the data structure is immediately updated. But if there are active iterators, only the pointer pointing to the element to be removed is assigned to an object acting as a place holder.

A collection determines whether there are active iterators by simply counting them. The class Collection maintains a variable that counts the active iterators. It is incremented on registration, and decremented on unregistration. If this counter eventually drops to zero, the collection cleans up its structure by removing the place holders and updating the data structure when necessary[7]. In order to avoid cleanups if there are no place holders, Collection also counts the number of place holders.

4.3.1 The Place Holder Approach for OrdCollection

The place holder approach works for OrdCollections, but has a disadvantage. Once a place holder has been set, all operations have to take that into account. Especially the At operation returning the element at the specified index must internally iterate until the number of elements (i.e. objects which are not place holders) becomes equal to the index. Thus, the advantage of an array has disappeared, and the run-time behaviour gets inacceptable if many At operations are invoked in this situation.

---

5 in version 2.2 by the class Collection

6 In the literature, it is often called lazy deletion, for instance in [Mathieu88], but this is obviously not precise. In C++, deletion has the meaning of destroying and deallocating an object.

7 For this purpose, Collection defines the hook method RemoveDeleted that is overridden by derived classes.

It is obvious that the place holder approach cannot provide robustness of iterators with respect to insertions. Whenever an insertion takes place at an index less or equal the index describing the iterator's current element, this index points to an array field already visited. For the example shown in figure 4.2, the iterator has already yielded object "D", but because of the insertion of "C" at index 2, "D" is incorrectly yielded a second time. As a consequence, it is considered an error to insert elements into an OrdCollection while it is being iterated over.

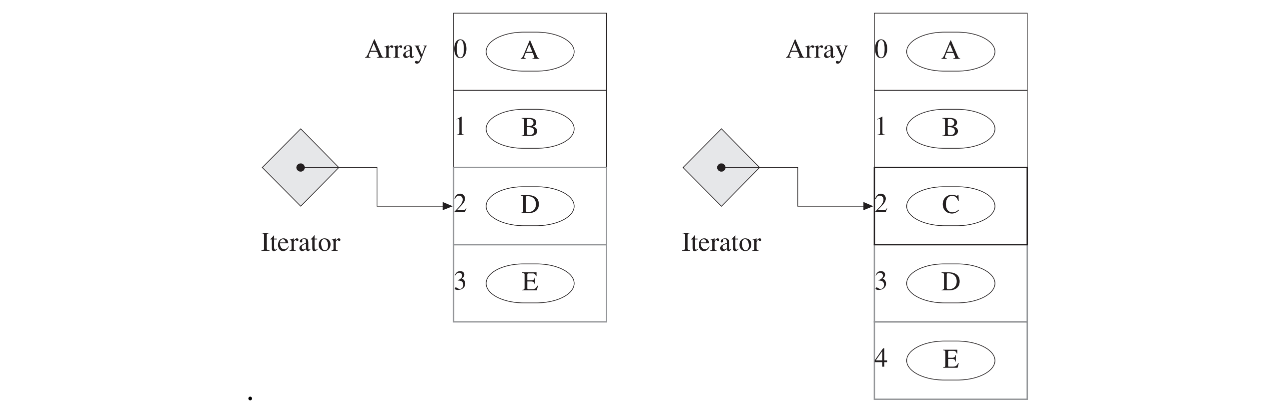

Fig. 4.2 OrdCollection: Iterating and Insertions

4.3.2 The Place Holder Approach for ObjList

ObjList is implemented as a doubly-linked list. The list nodes are instances of the regular class ObjLink derived from Object. The list nodes can be instances of a class derived from ObjLink, and may be endowed therefore with additional functionality by the client.

Since there was no semantic model for version 2.2, the behaviour of ObjList's iterator was erroneously not considered robust with respect to insertions. But insertions in a doubly-linked list do not pose problems for robust iteration: An iterator may be implemented such that insertions do not require any actions on the side of the iterator, no matter whether pre- or postincrement iteration is prescribed.

The problem lied in the discrepancy between implementation and expected semantics. The iterator was implemented in postincrement style, but it was expected to have preincrement semantics. For example, an iterator just having yielded the object "B" (see figure 4.3) will miss object "C" that has been inserted after the access to "B" although "C" comes after "B" in the collection's sequence.

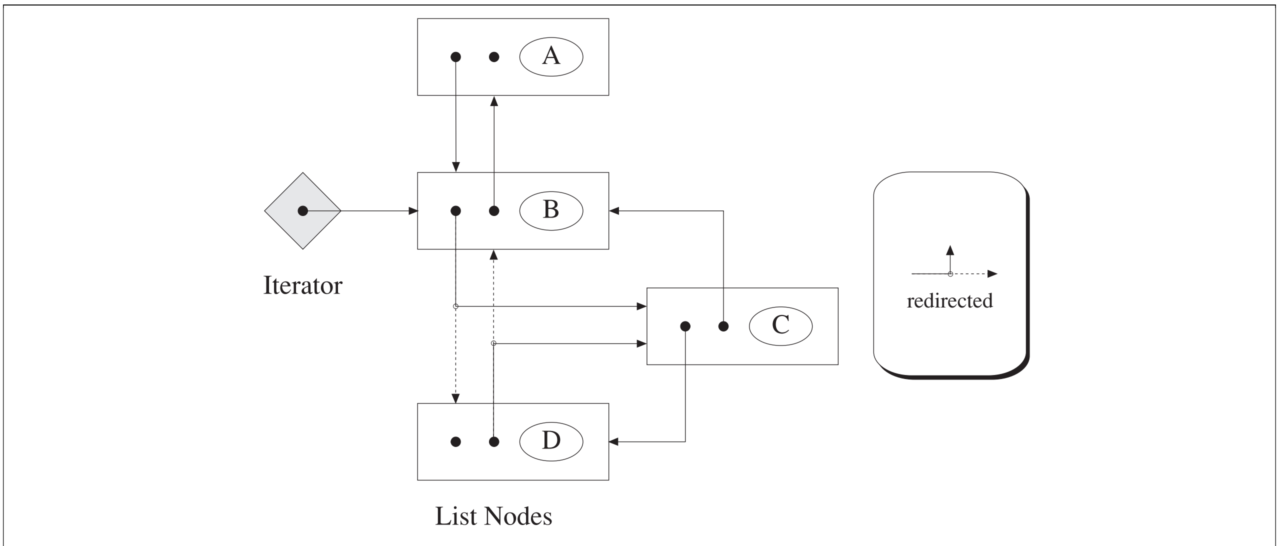

Fig. 4.3 Iterating over a Doubly-Linked List

4.3.3 The Place Holder Approach for Set

The class Set implements its behaviour using open-address hashing with linear probing. Why linear probing was chosen is explained in 4.4.4.1.

For all open-address hashing schemes, a *table rehash* is needed if the number of elements exceeds some boundary which is a constant fraction[8] of the table size. This is done by allocating a new table of a greater or smaller size. After that, all elements are inserted into that table. Finally, the old table is deallocated.

Growing the hash table is essential because the performance dramatically decreases if the load factor approaches one, i.e. the table becomes nearly full. Shrinking the table is also important to avoid wasting memory. Of course, the table rehashing is transparent to the client.

When there are no active iterators, removals and insertions are immediately done, and the resulting number of elements is checked against the upper and lower bounds of the table size during these operations. When there are active iterators, the remove operation replaces the element to be removed by a place holder. This is necessary because otherwise, the fixing of the collision chain would possibly relocate elements that are currently pointed to by an iterator.

Insertions are considered illegal if there are active iterators. But as long the table is not rehashed, insertions would be possible. It could not be guaranteed, however, whether a newly inserted element would be yielded by an iterator. But if the table is rehashed, every element is going to be stored in another slot. Thus, the sequences of elements in the old and the new table are totally different. This would have the effect that some elements are yielded twice, and other elements are not yielded at all. This is unacceptable, and insertions are trapped if iterators are active.

The intertwining of iteration and growth behaviour is undesirable. Iterators that never terminate do prevent from ever cleaning up the table or doing a table rehash. The most likely reason for such cases are active iterators that the client forgets to delete. If the Iter class is consequently used, iterators will be virtually never forgotten because they are automatically deleted, but the client is free to not use the Iter class. The implementation even created a place holder object for each removed element, so the waste of memory could be considerable.

4.3.4 Analysis

Eventually, for every removed element, a new place holder was created. This is unneccesary, hence a waste of memory and cpu time[9], because for ObjList and OrdCollection, sharing one such object is enough. For the Set class, it might seem necessary to store the hash value of the object that the place holder stands for, but the hash value of the removed element is no more needed. It would be needed only if a place holder is to be rehashed but linear probing makes that unnecessary. In other open-address hashing schemes, the collision chain cannot be fixed anyway, so sharing the place holder object is also possible for open-address hashing in general.

In general, using place holders tends to clutter up the code, because all procedures accessing the representation must take place holders into account. Most often, it is also difficult to avoid runtime overhead due to the contamination[10]. For example, accessing an element in an array by index is no more efficient as soon as place holders are present. Sorting data structures pose the problem to have a place holder to compare as the element it has been replaced for. When the sorting algorithm copes with place holders, it may get extremely complicated.

---

8 This the maximal load factor.

9 Place holders are instances of DeletedObject.

10 There exist algorithms where place holders may improve efficiency, however. See [Mathieu88] for example. Place holders may increase efficiency when their costs are smaller than the gains obtained by avoiding or postponing the reorganization of the data structure.

4.4 Version 3.0 - The Adjustment Approach

The adjustment approach is based on the idea that the controlling variables of an iterator can be adjusted when necessary. Adjustment allows for insertions during an iteration, and also allows for *immediate removals*, in constrast to lazy removals as applied in the place holder approach.

4.4.1 A Semantic Model of Robust Iteration

In version 2.2, there was no abstract model that prescribed the behaviour of robust iterators. As long as no insertions are allowed during iteration, such a model is not that important. But as soon as insertions are allowed, a model is needed such that all iterators behave in the same way. Otherwise, the client has to distuingish between slightly different behaviours. This hampers substitutability.

4.4.1.1 Formal Description

The semantic model describes the relationship between the sequence order and the effects of insertion and removal operations. The *sequence order* is the order of the objects as yielded by an iterator.

The sequence order is described by a set A of *sequence numbers* α_i and a bijective mapping ϕ between these numbers and the elements c_i of the container C. The elements of A are in the open interval (0, 1] of the real numbers. The image of 1, i.e. $\phi(1)$, can be seen as an special object which is interpreted as end of sequence. Initially, A is { 1 }. There is another set D of used sequence numbers δ_j being subset of the interval (0, 1). D's elements have been mapped to elements of C which have been removed meanwhile.

If an object p is inserted into C, it receives a sequence number π that is neither in A or D. If there is an element c with sequence number γ in C that immediately follows p, then π is less than γ but greater than all sequence numbers in A or D which are less than γ. Accordingly, if p immediately follows c, π is smaller than the successor of γ in A, but greater than all sequence numbers in A or D which are smaller than the successor of γ. If p becomes the last element of the sequence, π is greater than all numbers in A or D.

To say it in other words: The assignment of a sequence number π is always relative to the sequence number γ in A that will follow π after the insertion of p, and π is the "closest" number in the half left to γ. Figure 4.4a illustrates the assignment of sequence numbers with an example.

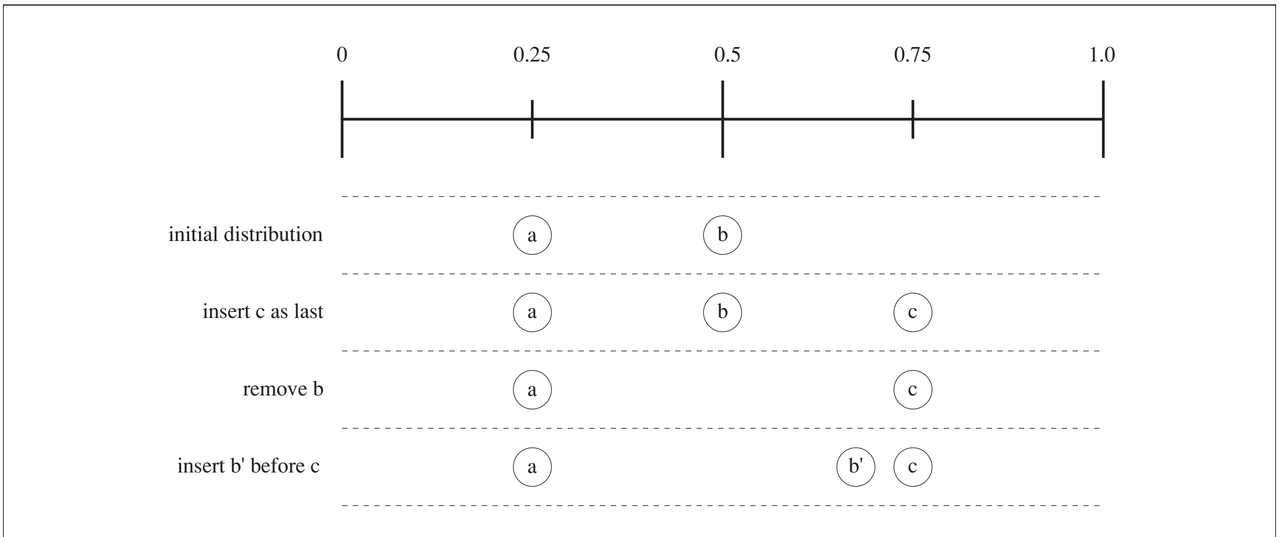

Fig. 4.4a Example for Assignment of Sequence Numbers

Initially, an iterator's current sequence number is 0. Upon a yield request, the element of C is returned whose sequence number is the least number in A being greater than the iterator's current sequence number. Afterwards, the iterator's current sequence number becomes the sequence number of the yielded element. Figure 4.4b shows the resulting sequence for two iterators. The modifications of the container are the same as in figure 4.4a.

Note that the iterator's current sequence number does not need to be in A when a yield operation is invoked. In this case, the most recently yielded element has been removed meanwhile. If there is no more sequence number less than one, then there are no more elements, and the iterator terminates.

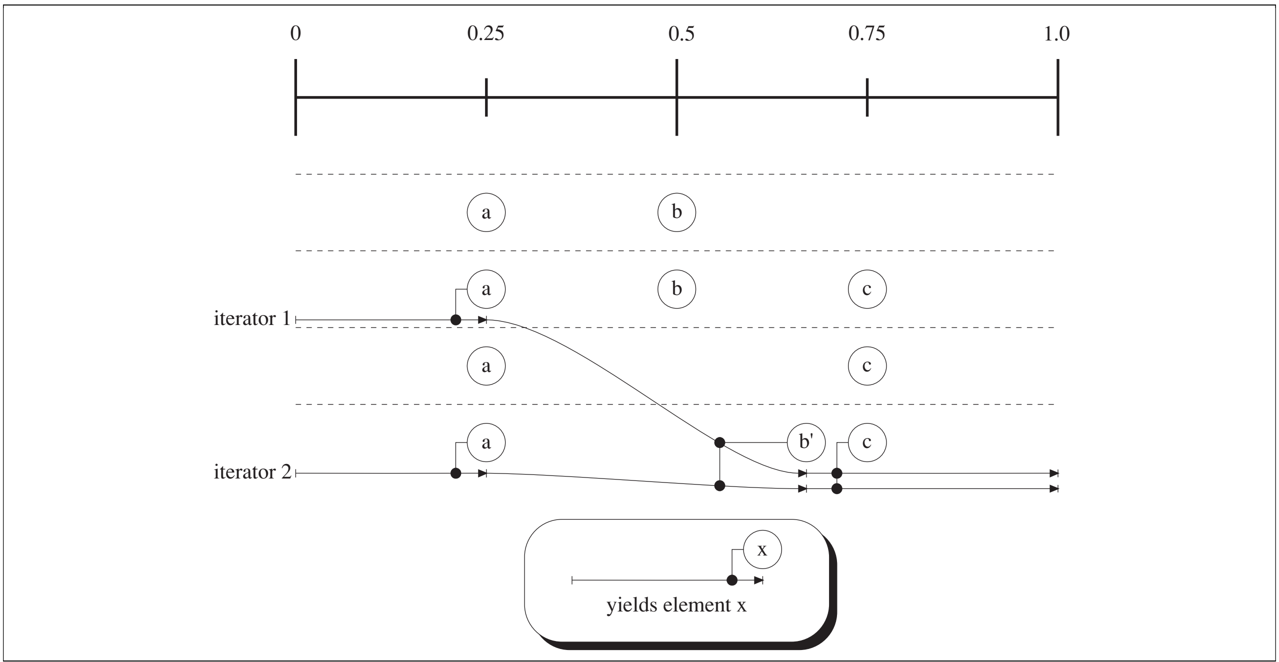

Fig 4.4b Two Iterators and their Sequence Order

Because unordered collections like Sets do not have either an explicit or implicit sequence order, an artificial one is defined. An element is always inserted such that its sequence number is greater than all other numbers in A or D. Such a sequence order reflects the time of insertions. Weaker definitions would also be sufficient, however.

4.4.1.2 Design Considerations

This semantic model clearly prescribes preincrement yielding. This is not visible at the interface of the iterator class, however, until the semantics of yielding is viewed in conjunction with simultaneous modifications of the underlying container. To decide on preincrement yielding is somewhat arbitrary. There are two arguments against preincrement yielding. First, programmers usually code loops in postincrement style, so postincrement yielding seems to be more natural. Second, preincrement yielding means for linked data structures like linked lists that, when removing an element which is currently referred to by an iterator, the iterator must refer to the predecessing element afterwards. When there is no backward reference, this can be expensive.

Moreover, the semantic model prescribes late termination. This is an somewhat arbitrary decision as well. Late termination is more powerful than early termination and could be exploited by the conscious programmer, although this is not particularly important because robust iteration has been devised for other reasons.

Both variants, however, may lead to infinite loops when the programmer is not aware of what may happen in complex situations. Contrary to the situation as discussed in chapter 3, this is not a problem in practice. I decided to use late termination.

Note that the yield operation is the only relevant operation of an iterator. This abstraction of the flow of control is atomic. The atomicity of the yield operation is not only a prerequisite for the kind of robust iterators proposed, but also for the abstractness of the construct itself. See also section 5.3 on the Eiffel library.

4.4.2 The Adjustement Approach for OrdCollection

An iterator operating on an OrdCollection uses an index to describe its current position. If an element is inserted at an index less or equal to the iterator's current position, all the elements after that element are shifted up by one. In order to be consistent, the iterator's index must be incremented by one. Analogously, removals are treated. If the index of the element to be removed is less or equal to the iterator's index, the latter must be decremented by one.

4.4.3 The Adjustment Approach for ObjList

Some remarks have already been made in 4.3.2. An iterator on a doubly-linked list stores a pointer to the current element's list node. Like all other container iterators in version 3.0, it operates in preincrement fashion. Using an head node[11] which terminates the list at both ends makes that easy. The pointer describing the iterator's position must be updated only if it points to the node of the element about to be removed. In all other cases, nothing must be done.

4.4.4 The Adjustment Approach for Set

The class Set was required to use a hashing algorithm for its implementation, but also to provide robust iteration without any restrictions. So, section 4.4.4.1 gives an overview on existing hashing schemes, and shows why linear probing was chosen again. An independent *sequence list* is then needed to guarantee a stable sequence order. 4.4.4.2 and 4.4.4.3 present the algorithms as far it concerns robust iteration, and 4.4.4.4 presents different possibilities for the sequence list.

4.4.4.1 Linear Probing versus Other Hash Algorithms

Two categories of hashing schemes exist. They differ in how collisions are handled. *Open-addressing* schemes compute alternative hash addresses, whereas *chaining schemes* link all elements which hash to a particular address in a chain. The class of chaining schemes can be further divided into the subcategories of direct chaining (which is supposed to include separate chaining) and coalesced hashing. The class of open-address hashing schemes includes, amongst others, linear probing, quadratic probing and double-hashing. A compilation covering all the mentioned schemes and more is found in [Gonnet91].

"Comparisons between hashing algorithms in different classes are often difficult, (...), because each class has its own assumptions, storage requirements, and tradeoffs." [Vitter87:89]. In order to choose an appropriate hashing scheme, the following points were to be considered. First, an implementation supporting robust iterators must be feasible and reasonably efficient. Second, it must be possible to grow or shrink the hash table to an arbitrary size by rehashing the whole table. Finally, the scheme must avoid contamination even in the worst cases. Thus, the hashing scheme must allow for immediate removals.

Linear probing is the only open-addressing scheme that allows for immediate removals [Knuth73b]. It is therefore the only candidate within its category[12].

Direct chaining has some advantages. It is simple to implement, and it allows for immediate removals. However, performance characteristics are not convincing when comparing it with other schemes on the base of gross memory consumption.

---

[11] See also Knuth [Knuth73a].

[12] As a further argument, linear probing can be very easily abstracted because it operates on an array-like data structure. My abstract algorithm for the robust hash table needs only four primitives to be implemented in a concrete derivation.

Coalesced hashing [Vitter87] needs no dynamic allocation of memory, but uses for each slot an extra link into the table. Coalesced hashing has excellent performance characteristics. Immediate removals that preserve randomness are possible, but require additional information to be stored in each slot except for the case of standard coalesced hashing which uses no cellar [Vitter87:114]. Standard coalesced hashing is clearly inferior to the variants with cellar. For the comparison to follow, I now assume that a (fictivous) coalesced hashing scheme with cellar exists that does not need extra links to allow for immediate removals.

Comparing *VISCH* (which is the best variant of coalesced hashing) and linear probing for the number elements growing to infinity shows that the average number of comparisons for linear probing is always smaller than 1.5 times the corresponding number of VISCH with the load factor doubled. A load factor of some α means that the hash table for linear probing occupying the same amount of memory has a load factor of $\alpha/2$, because in my case, both link and key are pointers and have the same size. Since I assumed that VISCH need no extra links, linear probing is here the better algorithm. It also seems to be simpler to implement than variants of coalesced hashing.

It is worth to note that many class libaries comparable with ET++ also use linear probing. This is the case for the NIHCL library, and Smalltalk [ParcPlace89, ParcPlace90]. Unfortunately, their designers did not document their considerations on this particular question.

4.4.4.2 Adjustments during an Insertion

Insertions that cause no growth do not need special attention. The element is simply inserted into the hash table and into the sequence list. Every active iterator will yield that element unless it has been removed meanwhile.

If the table grows during an insertion, all active iterators are notified to remember the element they currently point to. In terms of the semantic model, this is the element most recently yielded. It is guaranteed that there is such an element, because an iterator only gets active upon the first yield request. After the table growth is done, all active iterators may determine their position in the new hash table by searching the remembered element.

4.4.4.3 Adjustments during a Removal

When using an integrated sequence list, immediately fixing the collision chain requires to distuinguish between the three following cases:

a) An iterator points to the element about to be removed
b) An iterator points to an element going to be relocated during the fixing of the collision chain.
c) An iterator points to an element whose position is not changed.

There are three variants to handle these cases.

Variant A - Remembering Two Elements

All active iterators are notified to remember the current element before the operation actually takes place. Unlike for insertions, an active iterator also remembers the element previous to the current one. If the current element is the element about to be removed, the previous element is used to determine the iterator's position after the operation has taken place. If the search for the remembered element fails, then this element has been removed, and the previous element is searched. It is possible that the current element is the first element, so there is no previous element. This means that the iterator's position will be set to the initial position where no element is stored. This variant also works, when the element next to the current element is remembered instead of the previous one.

Variant B - Remembering One Element

The notification of an active iterator include passing it either the element about to be removed, or a value (e.g. an index or a pointer) that refers to the element's location in the data structure. By means of this value, the operation handling the notification can decide whether to remember the current or the previous element. Once the removal is done, the iterator is notified again. It then determines its possibly new position by searching the remembered element.

Variant C - Immediate Adjustment

In this variant, an active iterator does not remember any element. When the element to be removed is actually removed, all iterators are notified so each of them can determine whether it should set its position to the previous element. This is done by passing the position of the element to be removed, so an iterator can compare it with its own position. Afterwards, the collision chain will be fixed. For each element to be relocated, all active iterators are notified again. This time, they receive the old and the new position of the relocated element.

Comparison of the Variants

For all variants, the overall costs of the adjustment are proportional to the numbers of active iterators. Thus, it is sufficient to discuss the case of one active iterator.

In variant A, the cost of the adjustment is that of a succesful search if the iterator does not refer to the element about to be removed. If the iterator refers to the element to be removed, however, an unsuccesful search and a succesful search are done. Although it might seem a statistically rare case, unfortunately, it is not. A common application of robust iterators is to remove the element which the iterator has just yielded, as shown in example 4.1.

```
void CancelMondays(Collection *meetings)
{
   Iter next(meetings);
   Date *aDate;

   while ( aDate= Guard(next(), Date) ) {
      if ( aDate->DayOfWeek() == cMonday )
         meetings->Remove(aDate);
   }
}
```

Ex. 4.1 A Common Application of Robust Iteration

Variant B is clearly better than A because it always needs only one search. Variant B and C impose comparable costs. Since the operations "remember the current element" and "reset position" are also applicable for insertions, variant B is the best choice. It is more efficient than A, and is simpler to implement than C.

Note that the overhead of all variants depends on the costs of searching, and therefore on the current load factor. Thus, the maximal load factor also determines the smallest upper bound for the average costs of an adjustment.

4.4.4.4 Variations of the Sequence List

This section shows first the solution I decided to implement. Other variants using a separaated sequence list are presented and discussed afterwards. Note that the list nodes may be allocated from an array whose capacity is determined by the maximal number of elements in the hash table.

Integrated Doubly-Linked List

The figure 4.5 shows how the hash table and the sequence list as a doubly-linked list is organized. Each entry of the table consists of the pointer to the object and link to the previous (prv) and to the next slot (nxt). In order to keep the figure simple, the pointer to the objects and the objects themselves are denoted with "A", "B", "C" etc. The rightmost slot of the table does not really belong to the table, but it is the head node of the list. All lists used have an head node since it makes the code for insertion and removal of elements simpler and thus more efficient.

I favour this solution because no indirection occurs, and because immediate removals are possible without additional costs. The gross memory consumption is greater than in the linked-list variations. I decided to implement this solution.

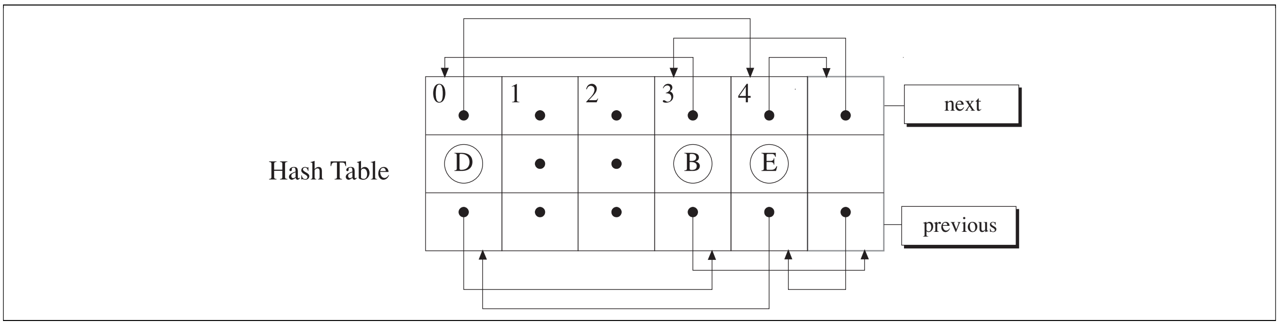

Fig. 4.5 Integrated Doubly Linked Sequence List

Arrayed Sequence List

Defining a sequence order with an separate array is straightforward and relatively simple, but has a serious problem: If both insertion and removal frequencies are high, the whole structure has to be frequently rebuilt in order to eliminate the holes in the sequence list array. Due to this external fragmentation effect, this organization is quite unattractive.

The potential advantages of that organization is the modest memory consumption if removals are rare. The indirection costs cpu time, however. I found for my particular environment that the indirection nearly doubles the time to access an element.

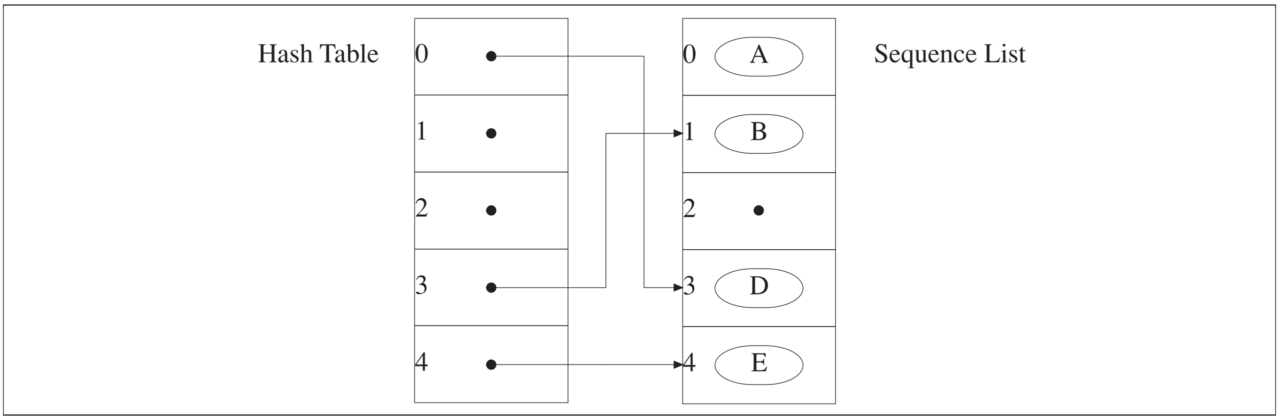

Fig. 4.6 Arrayed Sequence List

Single-Linked List

The first variant for separate lists is the single-linked sequence list. If an element is removed, its list node will not immediately be removed, but the pointer to the element is set to null. It is not possible to immediately remove the list node. If the number of list nodes pointing to null has reached a certain level, the positions of all iterators are forwarded to a list node not pointing to null. Then the list is traversed right from the beginning, and all list nodes pointing to null will be removed.

Although this variant uses the least amount of memory, it makes not much sense, since high modificational activity will require quite frequent traversing of the list. As for the arrayed sequence list, the indirection doubles the time to access an element.

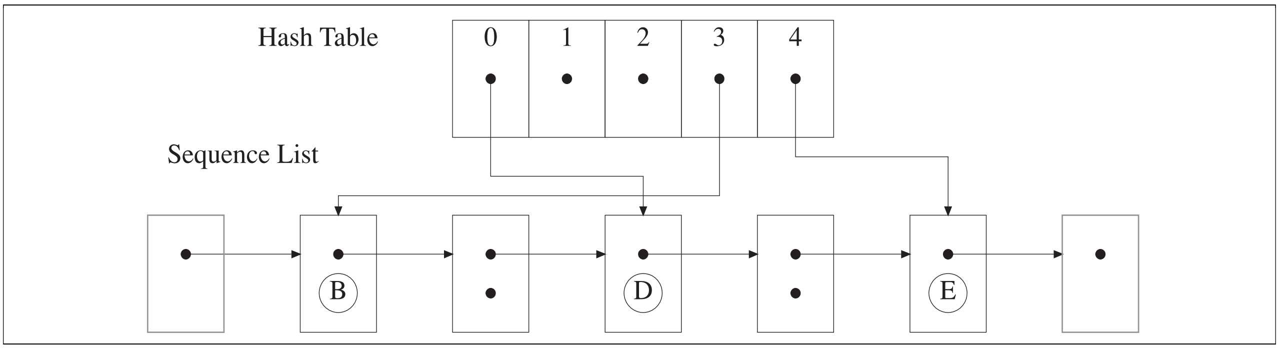

Fig. 4.7 Singe-Linked List

Shifted Single-Linked List

This variant is similar to the single-linked list. As major difference, the pointer to the object is not stored in the list node pointed to by the slot entry but in the next node. This structure could also be called a single-linked circular list with list elements shifted forward by one. Thus, a list node can be removed from the list without going through the whole list until its previous node is found.

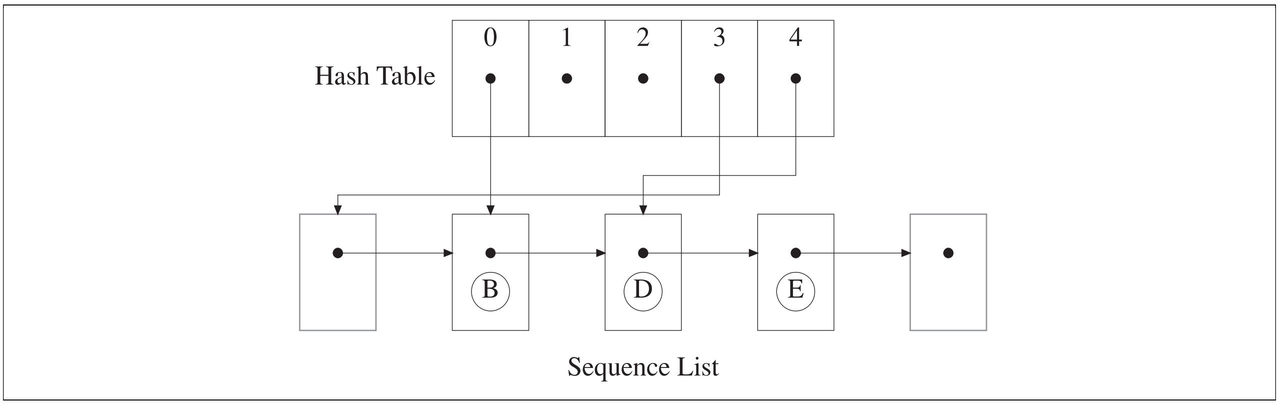

Fig. 4.8 Shifted Singe-Linked List

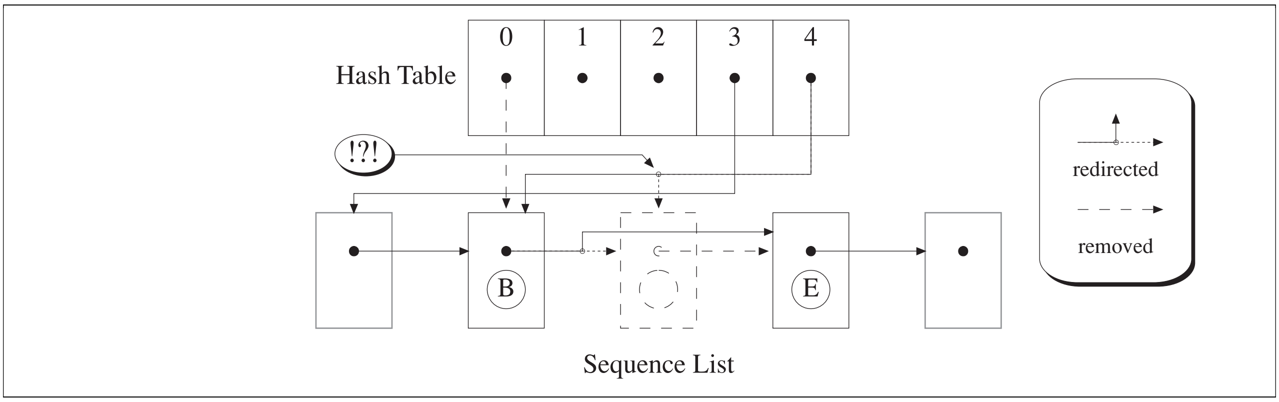

Fig. 4.9 Problem with Shifted Single-Linked List

But there is a problem with this structure that makes it relatively unattractive. Figure 4.9 shows a situation where the object "D" shall be removed. Since the reference "to E" must point to the node previously referred to by "to D", the hash table must be searched for "E" to update the reference "to E", too. This means that every removal requires an additional succesful search operation. This is the first, less important reason, because variant A also needs one additional succesful search at least. The more important argument not to choose the shifted single-linked list is the double indirection. Access times are more than doubled here. Moreover, this solution is relatively difficult to implement.

Separate Doubly-Linked List

Here, the removal of list elements makes no problems since the references to a list node can be updated without any problems. Obviously, this data structure uses the largest amount of memory per element. Again, the indirection is costly. As an advantage, the separate doubly-linked list is relatively simple to implement. It can be also easily combined with ObjList.

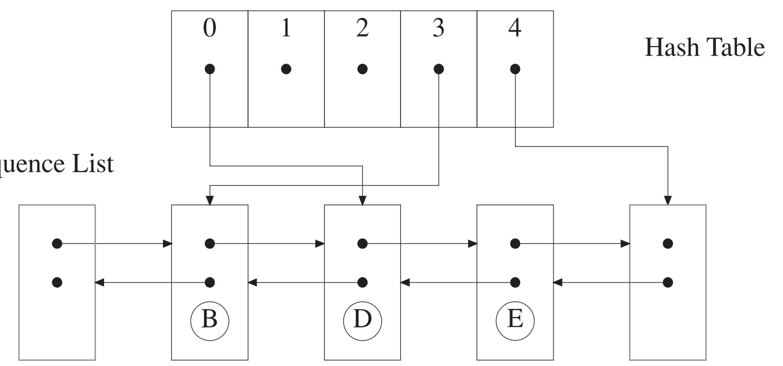

Fig. 4.10 Separate Doubly-Linked List

4.4.4.5 Implementation Details of Registration and Adjustment

The adjustment approach implies that a container must be able to access all its active iterators. Therefore, it maintains a doubly linked list whose nodes are the active iterators themselves. Registering an iterator means inserting it into the list, and unregistering an iterator means removing it from the list. With this kind of organization, the overhead is kept to a minimum.

Methods which have to adjust the active iterators use a special-purpose iterator, much like the ForEach construct described in 2.2.5. For each active iterator, the notification operation is invoked, so that the iterator may adjust its variables when needed. Example 4.2 illustrates this for the class ObjList.

```
void ObjList::RemoveLink(ObjLink *link)
{
   if ( this->HasIterators() )
      ForEachIterDo(ObjListIter, OnRemove(link));
   link->previous->next= link->next;
   link->next->previous= link->previous;
}

void ObjListIter::OnRemove(ObjLink *link)
{
   // this method is protected
   if ( this->currentlink == link )
      this->currentlink= this->currentlink->previous;
}
```

Ex. 4.2 Notifying an Iterator for Adjustment

4.5 Copying and Delaying Approaches

This section briefly discusses two other approaches for providing robust iterators. Both are considered as not suitable in the context of C++ because the language lacks of automatic garbage collection.

The idea behind *copying* approaches is to provide the iterator with a private, shallow copy of the container. Since there is no automatic garbage collection, the iterator possibly yields pointers that are no more valid. Such an invalid pointer referred to an object that has been deleted meanwhile. This is disastrous. In a language with automatic garbage collection (e.g. Smalltalk), copying the container is suitable as long the incurring overhead is small, i.e. the container has only few elements.

As a refinement which would improve run-time efficiency, the copying is done only when needed. It is triggered by the first modification that affects active iterators. A further optimization would consist of an mechanism that allows for sharing of copies where possible.

Another idea is to *delay* modifications. It is done by keeping track of the operation requests until they can be really executed on the data structure. However, it suffers from the same problem as copying the container when no automatic garbage collection is present. Furthermore, it is difficult to define semantics that can be implemented in a reasonably effficient way[13]. This is especially true if an arbitrary number of iterators on the same container must be possible.

For a system without automatic garbage collection, copying and delaying are safe only if none the removed elements are deleted. Since this again requires some prove by the programmer, the arguments of chapter 3 apply here accordingly.

5 Other Systems and Libraries

This chapter takes a closer look at other object-oriented libraries. It reports on how these libraries treat the problem of simultaneous iteration and modification. The non-object-oriented languages SETL and CLU which provide internal iterators are also investigated. All code examples refer to the example class Date introduced in chapter 2.

More general discussions addressing design and implementation of containers in C++ can be found in a paper by Lea [Lea88] and also in [Budd91]. The latter also includes a chapter on the container classes of Smalltalk-80.

5.1 C++ Class Libraries

Since ET++ is written in C++, other object-oriented libraries written for this language deserve special interest.

5.1.1 NIHCL and Borland

The NIHCL library [Gorlen87, Gorlen90] is a C++ library that also provides a rich set of container classes. They are conceptually very close to those of ET++. An interesting difference concerns the interface of the external iterator. In NIHCL, the class Iterator provides two operations:

```
virtual Object *Iterator::operator++();    // yield operator
virtual Object *Iterator::operator()();    // get current element
```

Ex. 5.1 Interface of Iterator in NIHCL

The yield operator causes the iterator to remember the element most recently returned. This element can be asked for by operator(). The yield operator indicates the end of iteration by returning null. How an iterator is used, is shown in example 5.2a.

---

[13] Delaying and copying approaches can be said to implement simple transaction mechanisms.

```
void PrintAgendaOn(Collection *meetings, ostream *file)
{
Date *aDate;
Iterator it(*meetings);

while ( aDate= (Date *) it++ )
   aDate->PrintAsStringOn(file);
```

Ex. 5.2a Using an External Iterator in NIHCL

An alternative, but equivalent formulation of the iteration loop is shown in example 5.2b.

```
while ( it++ )
   ((Date *) it())->PrintAsStringOn(file);
```

Ex. 5.2b An Alternative Formulation Using operator++ and operator()

Iterators in NIHCL are not robust in any way. Although an iterator lets its collection do all the work, a collection does not even count its iterators. Counting the active iterators is very inexepensive and allows to issue an error message in critical situations, so the client would be informed at least.

The authors suggest to copy the collection as a remedy [Gorlen90:155]: "It is a dangerous practice to add objects or to remove objects from a container while iterating through the container, because the order of the objects in the container may change. ... One correct way of programming this problem is to iterate through a copy of the container while modifying the original, ..."

Note that it only takes a few lines for the operator(), but its presence prohibits the later provision of robust removals.

For the library of container and iterator classes supplied as a supplement to Borland's C++ compiler [Borland91], all statements made for NIHCL apply accordingly.

5.1.2 Container 2.0

Except for ET++ and MacApp 3.0, this C++ library is the only container class library I currently know which offers robust iterators. Container 2.0 [Glocken90] offers a Set class based on a hashing algorithm. This section shows how the library implements robust iterators for hash tables.

Like in ET++, this library also uses an adjustment approach. The hashing scheme employed is direct chaining. The nodes of the doubly-linked list are dynamically allocated, and a new node is always appended to the chaining list. A simplified example is shown in figure 5.1.

The crucial assumption is now that the hash table is never rehashed, so the bucket which an element is hashed to never changes, and the sequence order of the chaining lists never changes, too. Thus, a robust iterator is to be adjusted to the previous[14] list node only if the node currently pointed to by the iterator is removed.

An iterator increments or adjusts its position by asking a list node for the next node. Since the last node of a chaining list points to null, every list node must additionally refer to the hash table. The hash table is needed to locate the bucket the last node belongs to, and then to locate the first node in the next non-empty bucket.

14 or to the next node node when postincrement iteration is prescribed

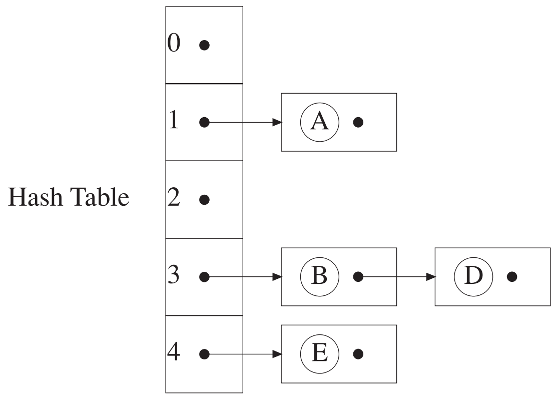

Fig. 5.1 Direct Chaining

5.1.1.1 Discussion

This implementation is not optimal. First, there is an absolutely unnecessary overhead by referencing the hash table in every list node. Second, an iteration over the whole table costs as many hash address computations as there are non-empty buckets.

The list nodes are assigned the responsibility to determine the next node because this saves a little bit of code: all containers in the library are composed of abstract nodes that can be asked for their previous and next nodes. Thus, there is only one implementation needed for the iterator's base functionality.

Even if this approach is considered as reasonable, the implementation can still considerably be improved. The list nodes at the end of the chaining list simply have to point to the first list node of the next bucket[15]. For each bucket, a counter stores the number of list nodes currently associated to this bucket. Thus, the pointer to the hash table in every node gets superflous, and iterating through the table no longer costs any computations of hash adresses[16]. Note that the requirement of never rehashing the table is still valid.

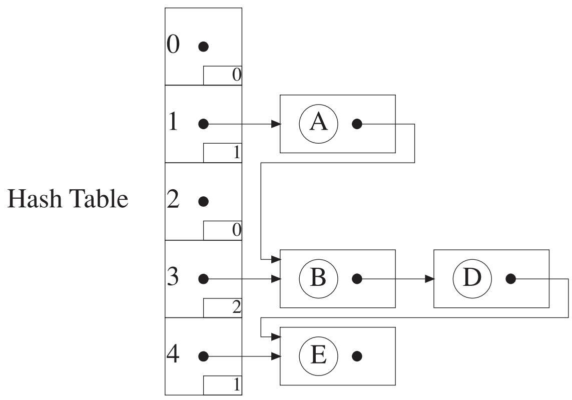

Fig. 5.2 Improving Direct Chaining in Container 2.0

15 This is true for singly- and doubly-linked lists. The implementation as found in the library uses doubly-linked lists, because a "navigator" object offering the iteration functionality can also yield an element preceding the current.

16 It is assumed that efficient iteration has highest priority.

5.2 Smalltalk

In Smalltalk-80, internal iterators are used more often than Streams which correspond more or less to external iterators in my terminology. However, Smalltalk streams base on a slightly different approach.

Smalltalk-80 does not offer something like robust iterators. The problem exists, but it is obviously not viewed as very important. "The programmer has to be aware of that problem and has to avoid such situations", says Ralph Johnson [Johnson91]. Lalonde and Pugh write in [Lalonde90:384]: "Users should not be attempting to modify collections while in the process of sequencing through their elements - the results can be unpredictable because the bounds of the loop are generally computed at the start of the loop - not every time through."

When the problem arises, Smalltalk programmers often copy the collection to be iterated over. The main reason why copying the container is not feasible for C++ libraries is the lack of automatic garbage collection. This issue is discussed in 3.3.

5.3 Eiffel

The Eiffel library as described in [Meyer88] offers containers which have no iterators. What comes close to an iterator is the cursor concept found in the list classes[17]. The cursor concept as realized in the Eiffel library allows only limited robust iteration. This section presents the concept and its implications in great detail. The reader is invited to compare this section with chapter 3. Note that the examples are coded in C++.

5.3.1 The Cursor Approach

A cursor is part of a list and stores a current position. Removals and insertions are done relative to the cursor. Moving the cursor to a position by the operations Go(int) or Search(Object *) allows to specify where the object is to be inserted. Operations relating to the cursor (and hence to modifying operations) are tabulated in example 5.3.

```
void List::Insert_Right(<ObjectType> *anObject);
   // inserts anObject right to the cursor, but does not move the cursor
void List::Delete();
   // removes the element pointed to by the cursor and moves the cursor
   // to the next element in the list

void List::First();     // sets cursor set to first element
void List::Forth();     // sets cursor set to next element in the list
bool List::Offright();  // is cursor after the last element?
void List::Mark();      // pushes cursor onto an internal stack
void List::Return();    // pops cursor from the internal stack
```

Ex. 5.3 Eiffel List Operations (denoted in C++)

Since there is no operation that corresponds to the atomic yield operation (operator()) of an ET++ iterator, an iteration through a list must always invoke three operations: the EOS test (Offright in example 5.4), the get operation (Value) and an increment operation (Forth).

---

17 The library also offers trees which are considered to be nested lists.

```
void PrintAgendaOn(List<Date> *meetings, ostream *file)
{
   Date *aDate;

   meetings->First();
   while ( ! meetings->Offright() );
      aDate= meetings->Value();     // no down-cast - List is generic
      aDate->PrintAsStringOn(file);
      meetings->Forth();
   }
}
```

Ex. 5.4 Iterating through a Eiffel List (Code in C++)

An Eiffel list easily allows for simultaneous iteration and removals, as shown in example 6.5. Since the library also offers single-linked lists, a removal moves the cursor to the next list node. This is post-increment behaviour (see 4.2 for more).

```
void CancelMondays(List<Date> *meetings)
{
   Date *aDate;

   meetings->First();
   while ( ! meetings->Offright() ) {
      aDate= meetings->Value();
      if ( aDate->DayOfWeek() == cMonday )
         meetings->Delete();           // moves cursor to next element
   }
}
```

Ex. 5.5 Simultaneous Iterating and Removing for a Eiffel List (Code in C++)

5.3.2 The Problem of The Cursor Approach

As long as there is only one active iteration, everything works fine. But two simultaneous iterations pose problems because a list has only one cursor. An Eiffel list now allows for pushing the cursor on an internal stack by the public operation Mark and for popping it from the stack by Return. Consider the code in example 6.6 for the date/meetings example used throughout this paper: If there is a meeting on wednesday, cancel all meetings on thursday earlier than the meeting at wednesday. It is not assumed that the list is sorted on the dates.

```
CorrectMeetings(List<Date> *meetings)
{
   Date *ad, *bd;

   meetings->First();
   while ( ! meetings->Offright() ) {
      ad= meetings->Value();
      if ( ad->DayOfWeek() == cThursday ) {
         meetings->Mark();
         meetings->First();
         while ( ! meetings->Offright() ) {
            bd= meetings->Value();
            if ( bd->DayOfWeek() == cWednesday && bd->Compare(ad) < 0 )
               meetings->Delete();
            else
               meetings->Forth();
         }
         meetings->Return();
      }
      meetings->Forth();
   }
}
```

Ex. 5.6 Problem for an Eiffel List: Two Iterations and Modifications (Code in C++)

The problem is that the definition of List forbids modifications if there are pushed cursors. This is stated as a pre-condition (called require clause in Eiffel) for the Return operation. The example is hence illegal according to the definition of List. Since the violation of a pre-condition itself has no effect in Eiffel, the example would not crash. The code even works here because the pushed cursor is not affected by the removals. This is not the case when an element is removed that is currently referenced to by a pushed cursor. Note that the "correctness" of the code was to be manually derived.

5.3.3 Discussion

The cursor approach has the following problems:

- An iteration pass is required to explicitly invoke all three fundamental operations as described in 4.2. This flexibility is not needed in general. In contrary, it leads to bulky code compared to an iterator with an atomic yield operation.
- Pushing and popping cursors is error-prone. Since an operation during an iteration can trigger another nested iteration on the same container, such programming mistakes may be hard to find.
- The iteration facility is robust only when there are no pushed cursors. It requires a prove of correctness.

The adjustment approach as proposed in this paper is amenable to the Eiffel cursor concept. To provide an atomic counterpart to the yield operation of an ET++ iterator would be easy, too. Last but not least, exploiting the compiler-generated destruction of automatic variables as done in the Iter construct would be helpful for popping cursors.

However, the major flaw of the cursor concept cannot be alleviated. As long as an iteration device is part of its container, programming nested or interleaved iterations is tedious and difficult to understand for the reader. The cursor concept is hence of limited value.

5.4 Objective-C and Dee

In the Objective-C library, there is no attempt to offer a solution to the problem. Cox writes in [Cox87:227-228]: "For effiency reasons, many collection classes assume the collection will not be changed while enumeration is occuring. (...) A safer implementation would be to pass the sequence a copy of the array, but this decreases performance to the point that the way enumeration is implemented across diverse collections could not be kept hidden from performance-conscious programmers."

Grogono who presented an object-oriented language called *Dee* considers the modification of an container while being iterated on as poor programming practice, hence denies the need of robust iterators. There are even no attempts to detect such cases [Grogono91:11-12].

5.5 Non-Objected-Oriented Languages

CLU

The designers of CLU, Liskov and Guttag, devote a quite interesting discussion on this issue (see [Liskow86:127-128]), but they conclude that it is not possible to provide robust iteration. As an answer to the question, they state that there is usually no need for robust iteration. Finally, they recommend that the client programmer should avoid modifications during iteration.

SETL

SETL [Schwartz75] is a very high level language dealing with sets and tuples. SETL offers iterators which are very similar to iterators in CLU. In contrast to CLU, iterators are naturally robust in SETL because the language has strict value semantics.

6 Implementing, Testing, and Evaluating the Improved Container Classes

This chapter highlights the most important details about the implementation of the improved containers as found in version 3.0 of ET++. This includes testing. Finally, benchmark results are reported.

6.1 Implementation

The implementation of the new iterators required substantial changes, so I decided to re-implement all concerned container classes from scratch. At this occasion, these classes were also improved in other respects: The protocols are now more consistent, and due to the extensive testing, the classes can be also said to be more reliable. Nevertheless, I tried to maintain compatibility to the container classes of version 2.2 as much as possible. When integrating the reworked container classes into the ET++ core library, only a few changes in the client code were necessary.

In version 3.0, a new class Container was introduced. As the most important reason, the inheritance relation between Set and Dictionary was to be eliminated: Dictionary should not be a subclass of Set (and therefore a Collection) because the relevant protocol of Dictionary is not consistent with the definition of Collection's protocol. As a further advantage, the Dictionary of version 3.0 does not use Associations, so the client is freed from dealing with Association objects. Finally, the new approach is more efficient in terms of memory usage and speed. The class Container defines the protocol common to Collections and Dictionaries and also allows to localize the implementation of the iterator registration mechanism.

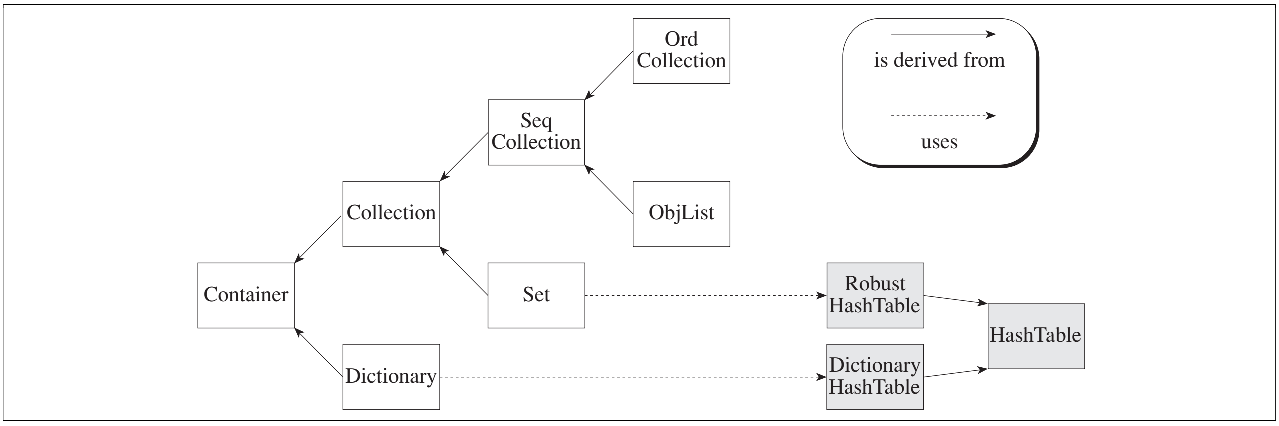

Fig. 6.1 Data Type and Data Structure Hierarchies (Version 3.0)

In the course of eliminating the inheritance relationship between the classes Set and Dictionary, separate hashing classes were built to factor out the common implementation. This distinction between data structure classes (HashTable etc.) and data type classes (Set, Dictionary) which use data structure classes is an organization principle widely agreed on. See figure 6.1 for the relationships between the resulting hierarchies. The amount of code is shown in figure 6.2.

Class	.C file	.h file	Total
Iterator	148	79	227
Container	259	168	427
Collection	195	60	255
SeqCollection	264	84	348
OrdCollection	465	109	574
ObjList	493	125	618
SortedOList	85	41	126
Set	281	84	365
Dictionary	295	113	408
HashTable	372	103	475
RobustHashTable	160	61	221
DictionaryHashTable	199	50	249
Total	3216	1077	4293

Fig. 6.2 Code Size of the Container Classes (Version 3.0) in Lines Of Code

The class OrdCollection now uses a so-called gap. When a sequence of modifying operations exposes some locality of reference, less data must be moved around.

As an add-on, the class SkipList has been built to implement the skip list algorithm [Pugh90]. Skiplist is derived from SeqCollection and supports that interface, except for insertions at a specific position. The class Skiplist provides robust iterators based on the adjustment approach.

6.2 Testing

This section sketches how the container classes were tested. Since testing is not the very subject of this paper, most details and many interesting issues in this context are omitted.

The container classes were thouroughly tested using a set of test classes whose hierarchy parallels the container classes' hierarchy. For each container subtype, there is a corresponding testing class dealing with the pecularities of that subtype.

```
Object *TCollection::TestAdd(Collection *cp, Object *anElement)
{
   Enter("TestAdd");    // for tracing the flow of control w/o a debugger
   Object *notAccepted;
   int oldSize;

   oldSize= cp->Size();
   notAccepted= this->Add(cp, anElement);
   if ( notAccepted != 0 )
      Error(1, "Add must not fail", anElement);
   if ( oldSize + 1 != cp->Size() )
      Error(2, "size incorrect", anElement);
   return notAccepted;
}
```

Ex. 6.1a Checking for Postconditions of the Add operation

This approach has advantages. First, the class construct is used to compensate for the lacking support of modules. Second, code that checks for the violation of postconditions and class invariants can be implemented as methods in the corresponding testing class. A derived classes inherits these methods and can easily use them. A method checking for a class invariant can also be overridden in a derived class. The overidden method first invokes the inherited method and then checks for the stronger postcondition. Code examples 6.1a and 6.1b show how checking for the postconditions of operation Add could have been implemented. The true code for TestAdd is acctually more complicated, however.

```
void TSeqCollection::TestAdd(Collection *cp, Object *anElement)
{
   Enter("TestAdd");
   SeqCollection *sp;
   Object *notAccepted;
   int oldLastIndex, lastIndex;

   sp= Guard(cp, SeqCollection);
   oldLastIndex= sp->LastIndex();
   notAccepted= Collection::TestAdd(cp, anElement);
   lastIndex= sp->LastIndex();
   if ( oldLastIndex + 1 != lastIndex() )
      Error(1, "lastIndex incorrect");
   if ( sp->At(lastIndex) != anElement )
      Error(2, "wrong element at last index");
   return notAccepted;
}
```

Ex. 6.1b Checking for the Stronger Postconditions of the Derived Class SeqCollection

Some principles were followed while developping the testing classes. The first principle says that test cases are not input from an external data source, e.g. a file. Feeding test cases from an external data source would considerably complicate the testing code because it involves parsing and conversion of the data representing the test cases. The testing classes generate them, and only report on differences between predicted and eventual results. As a second principle, the code under test must not be modified because this requires some organizational efforts and may introduce new programming errors. Third, I decided not to employ any white-box-testing. Conducting a test suite relying only on black-box-testing is more valuable because the test suite is completely independent of the implementation. This is especially important for regression tests.

However, black-box-testing cannot detect methods that are never used. Several methods of SeqCollection are implemented such that they are perfectly valid and usable, for instance. But both its derived classes OrdCollection and ObjList override them for efficiency. These methods of SeqCollection are therefore never executed. They can be called dead methods.

It is not very difficult to find dead methods by inspecting the code, but testing them is more troublesome. Temporary modifications of the code are ruled out. Testing dead methods using the scope operator :: introduces implementation-dependencies in the testing code, so the only clean solution is derive an additional, concrete[18] class that implements the primitive methods only. Since this is not free, I did not test dead methods.

Is the presence of dead methods an indicator for a doubtful implementation? This can be answered with yes or no. There is a conflict between desirable properties of an implementation. On one side, there should be no code which is never executed. Thus, the methods in question should be treated as though they were abstract. On the other side, they provide a default implementation and hence allow the client to derive a new class with a minimal effort. Moreover, a dead method documents the semantics of the operation in terms of other operations of the class.

It is not surprising that the total size of the testing software consists of about 1500 LOCs. This reflects the well-kown fact that serious testing takes a considerable fraction of the total work. Some of the code could be reused for the benchmark programs.

6.3 Benchmarks

The developped benchmarks were built to compare the performance between version 2.2 and 3.0 on one side. On the other side, the benchmarks also had to show that the concept of robust iteration as proposed is feasible in terms of the incurring run-time overhead.

The benchmarks showed that the costs of adjusting one iterator is always smaller than the costs of the operation causing the adjustment. As the most important result, this is even true for the proposed approach (Variant A) for linear probing.

6.3.1 Comparison between version 2.2 and 3.0

As the most interesting difference, the new implementation of the Dictionary class is three times faster on modifications than the old one. There are two reasons which equally contribute to the speed gain:

1. Since the new implementation does not use Association objects, but stores the keys and the values in an array with two columns, only one virtual function call instead of two is now needed to compute the hash value of a key or test two keys for equalness.
2. Inserting a key-value pair required the dynamic allocation of an Association object, and removing a key-value pair required the dynamic deallocation of its Association object.

For the other classes, the differences are small and not worth reporting.

6.3.2 Feasability of the Adjustment Approach

OrdCollection

For the class OrdCollection, inserting an new object such that the gap does not need to be moved is the fastest operation. The measurements show that the relative overhead incurring by one active iterator is 50% for this case. Note that this is the worst relative overhead.

ObjList

The nodes of the doubly-linked list are allocated and deallocated by the standard memory allocator used almost everywhere in the ET++ library. Since the deallocation of a node takes so much time in comparison to the iterator adjustment, the relative overhead is vanishing to zero.

---

[18] An abstract class cannot be tested because it cannot be instantiated.

Set

The most critical class with respect to the efficiency of robust iterators is the class Set which uses linear probing as described. The interesting operation is the removal of an element. The time it takes to remove all elements without an iterator was compared with the time it takes to remove all elements using an iterator[19]. The experiment was arranged such that the elements are removed in the same order for both cases. The measured relative overhead was always less than 100% and approves the theoretical argument to follow.

The dominant operation that determines time complexity for an internal hashing scheme is the test for equality. Time complexity is hence measured by the average number of tests taken by either a succesful search ($\mu(\alpha)$) or an unsuccesful search ($\tau(\alpha)$). Time complexities are functions of the load factor α. No distinction is made between the testing for a free slot and the very testing for equality.

For linear probing, removing an element means that each element in the corresponding collision chains is either tested for equality or for possibly hashing to the emptied slot. Assuming that both operations approximatively have the same costs, the average complexity of a removal is equal to the average complexity $\tau(\alpha)$ of the unsuccesful search. When removing an element which an iterator is currently referring to, the adjustment is accomplished by searching either the previous or next element in the sequence order. Since this element is always found, the average complexity of an adjustment is equal to the average complexity $\mu(\alpha)$ of a succesful search. Thus, the relative average overhead $\upsilon(\alpha)$ of a removal with adjustment can be written as follows:

$$\upsilon(\alpha) = \frac{\mu(\alpha)}{\tau(\alpha)}.$$

The formulas for $\mu(\alpha)$ and $\tau(\alpha)$ are taken from the "Handbook of Algorithms and Data Structures" [Gonnet91] and describe the asymptotic behaviour for the number of elements growing to infinity. The graph of the function $\upsilon(\alpha)$ is shown in figure 6.3.

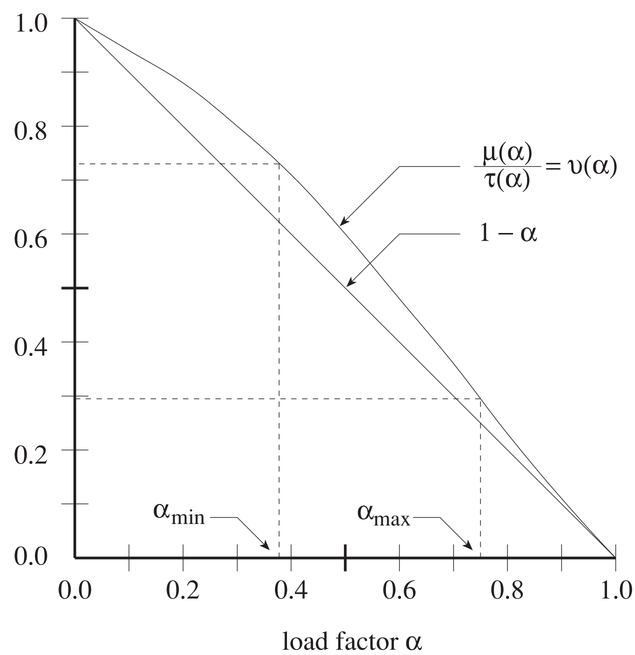

Fig. 6.3 Relative Overhead $\upsilon(\alpha)$ for a Removal With Adjustment

19 This is the situation of example 4.1.

Acknowledgements

I am indebted to Erich Gamma and Andre Weinand for many valuable hints and interesting discussions. Thanks go the other staff of the UBILAB, the Informatics Laboratory of the Union Bank of Switzerland, where this work was done. Last but not least, thanks especially go to Martin Wyser, also at UBILAB, for reading and commenting the manuscript.

Appendix A: Excerpts from the ET++ Class Hierarchy

Version 2.2

- **Iterator**
  - (subclasses not shown)
- **Object**
  - Assoc
  - **Collection**
    - Bag
    - ObjArray
    - RunArray
      - TextRunArray
    - **SeqCollection**
      - OrdCollection
      - ObjList
        - SortedObjList
    - Set
      - Dictionary
        - IdDictionary
      - IdSet
  - ObjLink

Fig. A1 The ET++ Container Hierarchy, Version 2.2

(Abstract classes shown in bold face)

Version 3.0

- **Iterator**
  - (subclasses not shown)
- **Object**
  - **Container**
    - **Collection**
      - **SeqCollection**
        - ObjArray
        - OrdCollection
        - ObjList
          - SortedObjList
      - Set
        - IdSet
    - Dictionary
      - IdDictionary
  - ObjLink
  - RunArray

Fig. A2 The ET++ Container Hierarchy, version 3.0

(Abstract classes shown in bold face)

Appendix B: Interface of Collection (Version 2.2)

```
// The following two declarations are global.

typedef Object *(*ObjectFunc)(Object *, Object *anObject, void *);
typedef bool (*BoolFunc)(Object *, Object *, void *);

class Collection : public Object {
public:
   int Size();
   bool IsEmpty();

   virtual Object *Find(Object *toFind);
   virtual Object *FindPtr(Object *toFind);
   virtual int OccurencesOf(Object *anObject);
   virtual int OccurencesOfPtr(Object *anObject);
   bool Contains(Object *anObject);
   bool ContainsPtr(Object *anObject);

   virtual Collection *Collect(ObjectFunc body, void *arguments= 0);
   virtual Collection *Select(BoolFunc body, void *arguments= 0);
   virtual Object *Detect(BoolFunc body, void *arguments= 0);

   virtual Iterator *MakeIterator();

   virtual Object *Add(Object *anElement);
   virtual Object *Remove(Object *toRemove);
   virtual Object *RemovePtr(Object *toRemove);
   virtual void Empty(int aSizeHint);
};

class Iterator {
public:

   virtual Object *operator()(); // yield operator
   virtual void Reset();
};
```

Ex. B Interface of Collection and Iterator, Version 2.2[20]

[20] For version 3.0, the methods in bold face are still introduced by Collection, whereas the others are now defined by Container.

References

[Aho83] Alfred V. Aho, John E. Hopcroft, Jeffrey D. Ullman: Data Structures and Algorithms. Addison-Wesley, Reading (Massachusetts), 1983.

[Apple92] Apple Computer, Inc.: Programmer's Guide to MacApp. Cupertino (California), 1992.

[Berztiss88] Alfs Berztiss: Programming with Generators. In: Software-Practice and Experience, Vol. 18(1), 1988.

[Booch87] Grady Booch: Software Components with Ada: Structures, Tools and Subsystems. Benjamin/Cummings, Menlo Park (California), 1987.

[Breuel88] Thomas M. Breuel: Lexical Closures for C++. In: USENIX Proceedings, C++ Conference (Denver, CO, Oct. 17-21, 1988), USENIX Association, Berkeley (California), 1988, pp. 293-304.

[Borland91] Borland International, Inc.: Borland C++ Version 2.0, User's Guide. Scotts Valley (California), 1991.

[Budd91] Timothy Budd: Object-Oriented Programming. Addison-Wesley, Reading (Massachusetts), 1991.

[Cox87] Brad J. Cox: Object-Oriented Programming - An Evolutionary Approach. Addison-Wesley, Reading (Massachusetts), 1986 (Reprinted with Corrections 1987).

[Gamma89] Erich Gamma, André Weinand, Rudolf Marty: Integration of a Programming Environment into ET++ - a Case Study. In: ECOOP 89, Proceedings of the Third European Conference on Object-Oriented Programming (Nottingham, UK), S. Cook (ed), Cambridge University Press, Cambridge, 1989, pp. 283-297.

[Gamma91] Erich Gamma: Objektorientiertes Software Engineering - Klassenbibliotheken, Designtechniken, Werkzeugunterstützung. PhD Thesis, University of Zurich (Switzerland), 1991.

[Glocken90] Glockenspiel Limited: Container Release 2.0 - Class Reference. Version 2.0, 1990.

[Goldberg83] Adele Goldberg, David Robson: Smalltalk-80 - The Language and its Implementation. Addison-Wesley, Reading (Massachusetts), 1983 (Reprinted with Corrections 1985).

[Gonnet91] Gaston H. Gonnet, Ricardo Baeza-Yates: Handbook of Algorithms and Data Structures - In Pascal and C. Addison-Wesley, Workingham (England), 1991 (2nd revised Edition, 1st Edition 1984).

[Gorlen87] Keith E. Gorlen: An Object-Oriented Class Library for C++ Programs. In: USENIX Proceedings and Additional Papers C++ Workshop (Santa Fe, New Mexico, Nov. 9-10, 1987), Berkeley (California), 1987.

[Gorlen90] Keith E. Gorlen, Sanford M. Orlow, Perry S. Plexico: Data Abstraction and Object-Oriented Programming in C++. Wiley, Chichester (England), 1990.

[Grogono91] Peter Grogono: Issues in the Design of an Object-Oriented Programming Language. In: Structured Programming, Vol. 12, 1991, pp. 1-15.

[Johnson91] Ralph E. Johnson, University of Illinois, in a personal communication.

[Knuth73a] Donald E. Knuth: The Art of Programming, 2nd edition, Volume 1/Fundamental Algorithms. Addison-Wesley, Reading (Massachusetts), 1973, 1968.

[Knuth73b] Donald E. Knuth: The Art of Programming, Volume 3/Sorting and Searching. Addison-Wesley, Reading (Massachusetts), 1973.

[Lalonde90] Wilf R. Lalonde, John R. Pugh: Inside Smalltalk, Volume I. Prentice Hall, Englewood Cliffs (New Yersey), 1990.

[Lea88] Douglas Lea: libg++, The GNU C++ Library. In: USENIX Proceedings C++ Conference (Denver, Colorado, Oct 17-21 1988), Berkeley (California), 1988, pp. 243-256.

[Liskov86] Barbara Liskov, John Guttag: Abstraction and Specification in Program Development. The MIT Press, Cambridge (Massachusetts), 1986.

[Meyer88] Betrand Meyer: Object-Oriented Software Construction. Prentice-Hall, New York, 1988.

[Mathieu88] Claire M. Mathieu, Jeffrey Scott Vitter: Maximum Queue Size and Hashing with Lazy Deletion. Rapports de Recherche #851, Institut National de Recherche en Informatique et en Automatique (INRIA), Rocquencourt (France), 1988.

[ParcPlace89] ParcPlace Systems, Inc.: ObjectWorks\Smalltalk Version 2.5 Reference Guide. Mountain View (California), 1989.

[ParcPlace90] ParcPlace Systems, Inc.: ObjectWorks\Smalltalk Release 4 User's Guide. Mountain View (California), 1990.

[Pugh90] William Pugh: Skiplists - A Probabilistic Alternative to Balanced Trees. In: Communications of the ACM, Vol. 33, No. 6, June 1990.

[Schwartz75] Jacob T. Schwartz: On Programming - An Interim Report on the SETL Project, Part I and II. Courant Institute of Mathematical Siences, New York University, New York (New York), revised 1975 (Part I has been first published in 1973).

[Vitter87] Jeffrey Scott Vitter, Wen-Chin Chen: Design and Analysis of Coalesced Hashing. Oxford University Press, New York (New York), 1987.

[Weinand88] André Weinand, Erich Gamma, Rudolf Marty: ET++ - an Object-Oriented Application Framework in C++. In: OOPSLA'88 Conference Proceedings (September 256-30, San Diego, CA), published as Special Issue of SIGPLAN Notices, Vol. 23, No. 11, November 1988, pp. 168-182.

[Weinand89] André Weinand, Erich Gamma, Rudolf Marty: Design and Implementation of ET++, a Seamless Object-Oriented Application Framework. In: Structured Programming, Vol. 10, No. 2, June 1989, pp. 63-87

[Weinand91] André Weinand: Objektorientierter Entwurf und Implementierung portabler Fensterumgebungen am Beispiel des Application-Frameworks ET++. PhD Thesis, University of Zurich (Switzerland), 1991.