Design and Specification of Iterators Using the Swapping Paradigm

Bruce W. Weide[1]
Stephen H. Edwards[1]
Douglas E. Harms[2]
David A. Lamb[3]

Abstract — How should iterators be abstracted and encapsulated in modern imperative languages, e.g., Ada and C++? We consider the combined impact of several factors on this question: the need for a "common interface model" for user-defined iterator abstractions; the importance of formal methods in specifying such a model; and problems involved in modular correctness proofs of iterator implementations and clients. A series of iterator designs illustrates the advantages of the "swapping paradigm" over designs based on the traditional "copying paradigm." Specifically, swapping-based designs admit more efficient implementations, while offering straightforward formal specifications and the potential for modular reasoning about program behavior. The final proposed design schema defines a common interface model for iterators for a wide variety of generic collection abstractions in languages such as Ada and C++.

Index Terms — Formal specification, iterator, program verification, proof of correctness, swapping.

1 Department of Computer and Information Science, The Ohio State University, 2036 Neil Ave. Mall, Columbus, OH 43210. Telephone: 614-292-1517. Internet: weide@cis.ohio-state.edu, edwards@cis.ohio-state.edu.

2 Department of Mathematics and Computer Science, Muskingum College, New Concord, OH 43762. Internet: harms@vax.muskingum.edu.

3 Department of Computing and Information Science, Queen's University, Kingston, Ontario K7L 3N6, Canada. Internet: dalamb@qucis.queensu.ca.

1. Introduction

An *iterator* is an abstraction that supports sequential access to the individual items of a collection, without modifying the collection. While some "academic" languages (most notably Alphard [Shaw 77] and CLU [Liskov 77]) include special language constructs for iterators, and others have been proposed [Cameron 89], the most widely used modern imperative languages such as Ada and C++ offer no special support for iterators. In these languages and most others, iterators must be designed and encapsulated using the same mechanisms that are used for other user-defined abstractions: procedures, and packages, classes, or modules. This paper discusses why previously published iterator designs are unsatisfactory in several respects, and considers the combined impact of several recent advances on the potential for improvement.

One such development is the proposal by Harms and Weide [Harms 91, Weide 91] that swapping should replace copying as the primary data movement mechanism in imperative programs. This approach has several advantages for designing and implementing generic reusable software components, including improved efficiency and simplified modular reasoning about program behavior. The swapping paradigm is especially valuable when dealing with potentially large and complex composite data structures that represent collections of other items — just the situation in which iterators are normally used.

In another recent work, Edwards [Edwards 90] proposes that the swapping paradigm might be applied to the design and implementation of iterators. His work also addresses a serious problem facing software component designers, i.e., developing interface models that simplify component composition. Tracz [Tracz 89] discusses an Ada example involving what Edwards [Edwards 91] notices is an iterator. Edwards defines a *common interface model* as a convention, shared by designers of piece-part families and their potential clients, for how the "plugs" and "sockets" of plug-compatible software components are supposed to work. It includes not only parameter profiles of operations, but a shared understanding of the abstract behaviors expected of those operations.

A third recent development is the development of formal trace specifications for iterators by Lamb [Lamb 90], and by Pearce and Lamb [Pearce 92]. These papers clearly explain the need for, and difficulties in, formal specification of iterators. Two related aspects of this issue that must be faced when defining a common interface model are: How should the abstract behavior of an iterator be designed so that all relevant features can be formally specified, and how can one use this specification to reason about program behavior? Especially in a component-based system this reasoning must be modular, i.e., it must be possible to reason about the correctness of the iterator implementation independently of the client program, and vice versa. The crucial importance of, and difficulties with, modular verification of realistically large software systems are noted by Ernst, et al. [Ernst 91] and Hollingsworth [Hollingsworth 92], among others.

Previous work on iteration over the elements of a composite data structure, summarized nicely by Bishop [Bishop 90], has not considered together the importance of efficiency with respect to copying, the need for formal specification of a common interface model, and the importance of modular reasoning about correctness in the design of iterators. This paper therefore has the following related objectives:

(1) To show how to design an iterator in the swapping paradigm (so the iterator can be most efficiently implemented, i.e., without copying either items of the collection or the collection's representation data structure).

(2) To give an abstract model-oriented specification of an iterator for a particular abstract collection of items (so the iterator's abstract interface is clearly and unambiguously defined).

(3) To explain how this specification supports modular reasoning about the behavior of the iterator implementation and its clients (so programs involving iterators can be verified in a modular fashion).

(4) To demonstrate how the techniques that are used can be generalized to design iterators for virtually any abstract collection (so composability of components is promoted).

This paper is, in effect, a proposal for a common interface model for a large class of iterators. A superficial examination of this model suggests that it is not much different from previously published iterators. In fact, however, our designs resemble others' primarily in having similar names for the operations. The *behavior* of these operations — both in functionality and performance — is subtly and importantly different.

We begin in Section 2 with a brief review of the swapping paradigm and a notation for formal specification, and introduce a simple example that forms the basis for a development of a simple iterator: a FIFO queue abstract data type. We also discuss the basic problems with previous iterator designs. Section 3 explains, step-by-step, how to arrive at the design of an acceptable swapping-style iterator for this ADT. It addresses objectives 1–3 above for each candidate design along the way. Finally, Section 4 discusses various extensions and examples, and shows how the method used for the simple FIFO queue example can be generalized to give a schema for designing a class of iterators for other collection types. All iterators designed using these principles share a common interface model. Example code for two typical client operations is provided in Appendix A.

2. Background

This section discusses the features required of an acceptable iterator design, the rationale for limiting the discussion to user-defined abstractions (as opposed to built-in language constructs that support iterators), the relevant details of the swapping paradigm, and our approach to and notation for formal specification.

Throughout the discussion we refer to the "client" (respectively, "client program" or "client code") and to the "implementer" (respectively, "implementation"). The former is the programmer (respectively, program) that uses the abstract iterator concept. The latter is the programmer (respectively, program) that realizes the iterator abstraction in the form of an executable code.

2.1. Iterators

The simplest kind of iterator permits a client program to examine (i.e., to execute some piece of code for) each of the items of a collection without modifying the collection as a side-effect of iterating over it. The items are presented to the client in some natural order that is based on the collection abstraction. Examples include enumerating and accumulating information about the items in a set, printing all the items in a tree, and copying a FIFO queue. There is no natural order for iterating over the elements of a set (any order will do), but there are several useful presentation orders for trees and an obvious natural order for a FIFO queue.

One can imagine a variety of more complex iterators and possible uses for them. For example, one might wish to be able to exit early from an iteration based on satisfaction of some condition; to have some control over the order of iteration or to leave it entirely unspecified; or to change the original collection while iterating over it. We begin by considering the simplest case described above, and comment on some of the more complex cases in Section 4.

A review of past work suggests there are two subtle aspects of even the simplest iterators:

(1) *Correctness* — It should not be permissible for a (correct) client program to iterate over a collection while interleaved operations on that collection might be changing it. We call this property *non-interference*.

(2) *Efficiency* — It should be possible for a client program to iterate over a collection without copying the data structure that represents the collection and without copying the individual items in the collection.[4]

Correctness — Recognition of the relationship between non-interference and the modular verification of correctness dates back to attempts to verify Alphard programs involving iterators [Shaw 77]. Programmers using one of Alphard's iterator constructs are advised to consider non-interference to be a "restriction on its use," but no formal proof obligation is raised during verification. Proof rules should permit local verification of an implementation and its client programs, but this cannot be achieved without an assurance of non-interference, either through restriction by language syntax or by the presence of a non-interference proof obligation [Ernst 91, Hollingsworth 92]. Alphard, like other languages with iterator constructs, offers neither.

In an attempt to deal with non-interference in user-defined iterator abstractions, Booch [Booch 87] and Bishop [Bishop 90] suggest classifying iterators into two categories, which Booch calls "active" and "passive." An active iterator is a module that exports an iterator type and associated operations and permits a client to build iteration loops with standard control constructs, e.g., while loops. The main difficulty with this approach is that the loop body may contain calls to operations that manipulate the collection over which iteration is being done; this is precisely the problem with Alphard's and other language-supplied iterator constructs. A passive iterator simply encapsulates the loop in a single procedure, which is parameterized by an action which would be the loop body in an iteration using an active iterator. The argument is that in this case there is no (obvious) way for a client to interleave operations that change the collection with those iterating over it, because the latter are encapsulated in the passive iterator procedure.

---

4 In the special case that copying a collection is the purpose of iterating over it, all copying should take place in the client code that is executed for each item. Copying should not be inherent in the iterator itself.

Unfortunately, passive iterators suffer from their own serious problems, discussed in detail by many authors [Booch 87, Bishop 90, Edwards 90]. From the standpoint of reusability they are far less flexible than active iterators. In the face of formal specification and the need for modular verification, the nature of the action procedure's effects and side-effects must be formally specified and proofs modularized. It is not clear how to do this. And a client still can violate non-interference by, for instance, declaring a collection to be global to the iterator's action procedure and interfering with the iteration by manipulating that collection surreptitiously. The *coup de grâce* for passive iterators is simply the observation that a passive iterator can be layered easily on top of an active one, but not vice versa.

Therefore, we follow the above-cited papers in concentrating on designs for active iterators. However, we insist that clients observe the non-interference property and be modularly verifiable, which necessarily makes our designs different from previous ones. That is, like Lamb [Lamb 90] we write our formal specification so non-interference *must* be observed by a correct client program. A proof obligation involving non-interference is raised in the client that can and must be discharged if the client program is to be considered correct.

By contrast, Booch [Booch 87] points out that his iterator designs are "relatively unprotected" from client abuse. Indeed nothing but self-discipline prevents a client from altering a collection during iteration over it. The same is true for Bishop's designs [Bishop 90]. Several methods for repairing this shortcoming are proposed by Edwards [Edwards 90], but like Booch and Bishop he does not deal explicitly with formal specification or the need for a framework for modular verification. These objectives drive many of our design decisions and account for the differences between Edwards' designs and the ones we propose here.

Efficiency — While non-interference has long been seen as a problem with iterators, Edwards [Edwards 90] was the first to recognize the inefficiency inherent in both published iterator abstractions and language constructs. All previously published designs for iterators (i.e., those before Edwards' papers [Edwards 90, Edwards 91]) include a function called, e.g., `Value_Of`. This returns to the client a copy of the next item from the collection.

The execution-time cost of such copying is troubling if the representations of the items in the collection are themselves large, complex data structures. As noted by Harms and Weide [Harms 91], the typical method of avoiding this expense — copying only a reference (pointer) to an item, as with the designs recommended by

Booch [Booch 87] and Bishop [Bishop 90] — creates even more serious problems from the standpoint of our objectives. It is a disaster for formal specification and it completely thwarts modular verification [Ernst 91, Hollingsworth 92]. From the practical standpoint, this makes understanding of and informal reasoning about program behavior much harder than it should be. Replacing copying by swapping is both efficient and amenable to tractable formal specification and modular proof rules, and hence to easier understanding of program behavior. This is the reason we prefer the swapping paradigm for our designs.

Another efficiency issue is noted by Edwards [Edwards 90] and Lamb [Lamb 90]. Achieving optimum performance of an iterator generally requires that the implementer of an iterator have access to the underlying representation of the collection. However, this is not essential solely to obtain the required functionality of an iterator, if the operations on the collection abstraction are sufficiently powerful [Edwards 90, Harms 91, Weide 91].

2.2. Language Features and User-Defined Iterator Abstractions

Alphard [Shaw 77] and CLU [Liskov 77] have built-in iterator constructs, and Cameron [Cameron 89] proposes some elegant variations. Here we concentrate on designing iterators as user-defined abstractions in languages that do not include special constructs to support iterators, and we do not further consider possible language support for our designs. There are three reasons for this. First, the practical successors to Alphard and CLU (e.g., Ada and C++) simply do not support iterators directly, so there is clearly a need for a design approach that does not rely on special language support. Second, even with language support one needs to define formally a common interface model for iterators if a high degree of composability of software components is to be expected [Edwards 91]. Finally, none of these language mechanisms satisfactorily addresses the problem of non-interference and the need for modular reasoning about program behavior, or the inefficiency of copying.

2.3. The Swapping Paradigm

The "swapping style" of software design [Harms 91, Weide 91] differs from the conventional "copying style" in using swapping (and the swap operator ":=:") to replace copying (and the standard ":=" operator). It is based on two observations about generic modules, e.g., Ada generic packages. First, items whose types are parameters to generic modules might have large data structures as their concrete representations. These items therefore might be expensive to copy. Second, an attempt to overcome the cost of copying the abstract values of such items by copying references to them inevitably leads to difficulties in establishing program correctness by modular reasoning. This in turn frustrates both the clients of an abstraction and maintainers of its implementations. Therefore, it is advantageous to design the abstract interface of such a generic component so an implementation can achieve data movement by swapping (exchanging) the abstract values of any two variables of the same type, rather than by copying abstract values or by copying references to abstract values.

Harms and Weide propose detailed principles to help designers create good reusable software components in the swapping style [Harms 91, Weide 91]. For example, when designing operations on collection types such as a `Queue`, one should define insertion operations such as `Enqueue` to permit implementations that swap items into the structure. Similarly, it is good practice to define inspection or removal operations such as `Dequeue` to permit implementations that swap items out.

A particularly instructive example is an `Array` ADT. The (single) *primary* operation should take an `Array`, an index, and an item, and swap the indexed element with that item. Both fetch and store become *secondary* operations using this primitive. That is, they can be implemented with an insignificant performance penalty by layering on top of the primary swap-based operation if they are really needed — and in most clients they are not [Harms 91, Weide 91].

2.4. Formal Specification

The main example we use throughout the rest of this paper is a FIFO queue abstraction. The formal specification of the `Queue_Template` concept (in a dialect of RESOLVE, adapted from [Harms 91, Weide 91]) is shown in Figure 1.

```
concept Queue_Template
    parameters
        type Item
    interface
        type Queue is modeled by STRING (Item)
            exemplar q
            initially   q = EMPTY
        operation Enqueue (alters q: Queue, consumes x: Item)
            ensures     q = #q o #x
        operation Dequeue (alters q: Queue, produces x: Item)
            requires    q /= EMPTY
            ensures     x o q = #q
        operation Is_Empty (preserves q: Queue)
                returns empty: Boolean
            ensures     empty  iff  q = EMPTY
end Queue_Template
```

Figure 1: Formal Specification of FIFO Queue Template

`Queue_Template` is a generic abstract module, parameterized by an ADT `Item`. It exports an ADT `Queue` and primary operations to `Enqueue` and `Dequeue` `Items` and to test if a `Queue` `Is_Empty`. This is a model-oriented specification in which a `Queue` is modeled as a mathematical string of `Items`. The notation "a o b", where a and b are either strings or `Items`, denotes the string obtained by concatenating a and b. `EMPTY` is the empty string of `Items`. Initially, a variable of type `Queue` is empty, i.e., its model is the string `EMPTY`.

The notation used in **`ensures`** clauses (postconditions) is that a variable stands for the value of its mathematical model at the conclusion of the operation; the variable prefixed with "#" (pronounced "old") stands for the value of the variable's mathematical model at the start of that operation. The "#" prefix is not needed or used in **`requires`** clauses (preconditions), where all variables denote values at the start of the operation.

The parameter modes are used to simplify specification, and have nothing to do with the mechanism for passing parameters [Harms 91]. Mode **`alters`** means the argument replacing this formal parameter may be changed as a result of the call; how it is changed is stated explicitly in the postcondition, which generally relates that variable's new value to its old value and to the values of other formal parameters. Mode **preserves** means the argument's value at the conclusion of the operation is the same as at the start of the operation; for example, in operation `Is_Empty` there is no need to say explicitly in the postcondition that "`q = #q`". Mode **consumes** means the argument's value is changed to an initial value for its type; for example, consuming a variable of type `Queue` would make it `EMPTY`, while consuming an `Integer` would make it 0 (assuming the initial value for `Integers` is 0). Finally, mode **produces** means the argument's value may be changed by the call, but its value at the beginning of the call has no influence on the operation's behavior.

Lamb [Lamb 90], and Pearce and Lamb [Pearce 92], use trace specifications for iterators. In this paper we use model-oriented specifications like the one above. Model-oriented specifications seem well-suited to designs based on swapping, have seen relatively widespread use in practice (e.g., Larch and Z), and are rather easily understood even by those not intimately familiar with the wide variety of formal specification techniques currently in use [Wing 90]. They also have been used in proof systems for modular verification of implementations and clients [Ernst 91].

3. Development of an Iterator for a Queue

The goal of this section is to develop a design approach that applies to iterators for any type of collection of any type of item. Initially we create an iterator for the generic `Queue` type of Section 2, then generalize. The presentation is incremental. In each step we present a proposed design of the iterator and sample client code that uses it, then discuss it, critique it, and propose a new design, until the final design achieves the stated objectives.

The development proceeds as follows:

- *Design #1* — Attack problem (1) from Section 2.1, i.e., non-interference and modular verification of correctness. We define a companion type `Iterator` for type `Queue` with operations that support iteration over a `Queue`. The idea of this step is to make non-interference a non-issue and thereby permit modular correctness proofs. The chief problem with this design is that it is based on the copying paradigm and therefore is inherently inefficient. In fact, Design #1 may look like a "strawman" to some readers; after all, no one really designs iterators this way. But that is precisely the point: To enforce non-interference and achieve modularity of correctness proofs, designs based on the "copying paradigm" must sacrifice efficiency. Other "real" iterator designs attempt to achieve some degree of efficiency *at the expense of* assured non-interference and proof modularity. Design #1 illustrates that the trade-off might be made in the other direction. It also serves as the basis for better designs to follow.

- *Design #2* — Attack problem (2) from Section 2.1, i.e., efficiency with respect to copying. We revise Design #1 to use swapping. The purpose of this step is to permit an implementation of an iterator that still demands non-interference and supports modular verification, yet does not need to copy either the data structure that represents the `Queue` or any of the `Items` in it. The main problems with Design #2 are that it is cumbersome to write a loop invariant to demonstrate the correctness of a typical client program, and that some swapping-paradigm principles still are not completely observed.

- *Design #3* — We add some abstract state information to the `Iterator` type to remedy the verification problem above, and change the operations slightly to take advantage of it. The purpose of this step is to facilitate client correctness proofs and to achieve closer adherence to swapping-paradigm design principles. This design achieves all the stated objectives.

3.1. Design #1

First we define a companion type `Iterator` for the type `Queue`. This new type has its own operations that support iteration over a `Queue`. Typical client code involves two steps: Transfer the `Queue` value into an `Iterator` variable, then iterate over that variable (not over the original `Queue`).

An appropriate mathematical model of an `Iterator` is (like a `Queue`) a string of `Items`.[5] This string records the order in which the `Items` are to be processed during iteration. Here we choose this to be the order in which the `Items` would be `Dequeue`d from the original `Queue`. It is clear that other orderings can be specified easily by changing the postcondition of `Start_Iterator`, and that for some representations of type `Queue` other orderings can be implemented as easily as the natural order; see also Section 4. The specification for Design #1 is shown in Figure 2.

---

5 An `Iterator` is not "modeled by a `Queue`" because in our model-based specification framework an ADT's model is always a mathematical object, not another program object.

```
concept Queue_Iterator_Template enhances Queue_Template
    interface
        type Iterator is modeled by STRING (Item)
            exemplar i
            initially   i = EMPTY
        operation Start_Iterator (produces i: Iterator,
                preserves q: Queue)
            ensures     i = q
        operation Finish_Iterator (consumes i: Iterator)
            ensures     i = EMPTY
        operation Get_Next_Item (alters i: Iterator,
                produces x: Item)
            requires    i /= EMPTY
            ensures     x o i = #i
        operation Is_Empty (preserves i: Iterator)
                returns empty: Boolean
            ensures     empty  iff  i = EMPTY
end Queue_Iterator_Template
```

Figure 2: Queue Iterator Design #1

Discussion — Design #1 introduces a specification mechanism called *enhancement*. By saying that this new concept **enhances** the Queue_Template concept, we mean that Queue_Iterator_Template imports and exports all the types and operations of the Queue_Template, plus those explicitly specified in the new concept. Enhancement is a kind of specification or interface inheritance [LaLonde 89].

One possible implementation strategy for this enhancement is to *layer* the Iterator type and operations on top of the Queue_Template [Muralidharan 90, Harms 91, Weide 91] so the new code respects the Queue abstraction. Another is to *directly implement* all the exported types and operations from scratch. For instance, one might simply copy code for the Queue_Template and add code for the new type and operations so the new code has access to the concrete representation of a Queue. In subsequent discussions of efficiency we assume the traditional direct implementation, although layering is advantageous in virtually all other respects [Hollingsworth 91].

Here is a sample of client code for iteration using Design #1:

```
Start_Iterator (i, q)
while not Is_Empty (i) do
    Get_Next_Item (i, x)
    (* code to process x *)
end while
Finish_Iterator (i)
```

As is evident from the sample code, Design #1 achieves non-interference by defining it away. The original `Queue q` is completely separate from the `Iterator i`. The `Start_Iterator` operation protects `q` from being changed during iteration. If the code in the loop body of the sample code manipulates `q` there is no interference with the iteration. Similarly, changes to `x` in the code to process `x` do not influence either `q` or `i`. Therefore, it is acceptable for a client program to manipulate `q` inside a loop that is iterating over `i`, even if `i` was obtained from `q`.

Critique — Non-interference is assured here only at the cost of efficiency. Design #1 effectively forces an implementation of `Start_Iterator` to copy `Queue q` into `Iterator i`. The reason is that simply copying a reference to `q` or references to its `Items` creates aliases, and hence cannot preserve the independence of the abstract values of `q` and `i` [Harms 91, Hollingsworth 92]. It is impossible to prove that such an implementation is correct outside the context of a client program. The reason is that the client program might manipulate `q` or its `Items` through these aliases. The only way to create a modularly verifiable implementation of Design #1 is to copy `q` and to copy its `Items`.

A clever implementation can try to defer copying of `q` (but not its `Items`) so long as there are no calls to Enqueue or Dequeue on the original Queue `q` during an iteration over `i`. It can keep enough internal state as part of a `Queue` representation to recognize that, in the abstract view of these operations, it supposedly has been copied to an Iterator `i`. It can determine whether an iteration is in progress by monitoring whether the call to `Start_Iterator` has been matched by a bracketing call to `Finish_Iterator`. If a call to `Enqueue` or `Dequeue` occurs during an iteration, the copy of `q` can be made at that time. Supporting this implementation is the only real reason for a `Finish_Iterator` operation in Design #1. In the worst case, however, copying of `q` is still necessary.

We again note that nearly all previously published iterator designs do not copy the data structure representing the collection, but they *do* make copies of its `Items` in the course of iterating. In such designs, the counterpart of Get_Next_Item is a function that returns a copy of the next Item in the collection. Again, a modularly verifiable implementation may not make this copy cheaply by creating an alias to the Item. These problems are intrinsic to the copying paradigm [Harms 91, Weide 91].

3.2. Design #2

Design #1 can be changed to use the swapping paradigm. The reason for doing this is to permit an implementation that does not need to copy either the data structure that represents the Queue or any Items in it. Two key ideas make this approach workable.

The first is a change to Start_Iterator and Finish_Iterator. The former can be implemented by moving the original Queue into an Iterator, and the matching call to Finish_Iterator by moving it back. This design relieves the implementer from responsibility for copying the data structure that represents the Queue. "Moving" arbitrarily large data structures in this way can be accomplished in constant time with swapping [Harms 91].

The second idea is to define Get_Next_Item so its implementation does not need to return a copy of the Item to the client, but can swap it out. This is possible if the client is required to pass that Item back in the *next* call to Get_Next_Item. In this case the implementation can simply put the Item back into the Queue data structure, and swap out the next one to return to the client. The only real hurdle is to get the boundary conditions correct, so the first and last calls to Get_Next_Item are not special cases.

The mathematical model of an Iterator becomes an ordered triple: a string of Items (called future) serving the same purpose as the model in Design #1 , a single Item (called present) that records the Item value currently held by the client, and a string of Items (called collection) that records the value of the original Queue. (The last component is not really necessary for a Queue iterator because a Queue is modeled by a string, but we include it because it is important when the model of the underlying collection is another mathematical object; see Section 4.) The complete specification for Design #2 is shown in Figure 3.

```
concept Queue_Iterator_Template enhances Queue_Template
    interface
        type Iterator is modeled by tuple
            (
                future: STRING (Item)
                present: Item
                collection: STRING (Item)
            )
            exemplar i
            initially   i.future = EMPTY  and
                        Item.init (i.present)  and
                        i.collection = EMPTY
        operation Start_Iterator (produces i: Iterator,
                consumes q: Queue, produces x: Item)
            ensures     i.future = #q  and
                        i.present = x  and
                        i.collection = #q
        operation Finish_Iterator (consumes i: Iterator,
                produces q: Queue, consumes x: Item)
            requires    i.present = x
            ensures     q = #i.collection
        operation Get_Next_Item (alters i: Iterator,
                alters x: Item)
            requires    i.future /= EMPTY  and
                        i.present = x
            ensures     x o i.future = #i.future  and
                        i.present = x  and
                        i.collection = #i.collection
        operation Is_Empty (preserves i: Iterator)
                returns empty: Boolean
            ensures     empty  iff  i.future = EMPTY
end Queue_Iterator_Template
```

Figure 3: Queue Iterator Design #2

Discussion — Here is a sample of client code for iteration using Design #2:

```
Start_Iterator (i, q, x)
while not Is_Empty (i) do
     Get_Next_Item (i, x)
     (* code to process x without changing x *)
end while
Finish_Iterator (i, q, x)
```

Why is this specification so much more complex than Design #1? How does it permit the implementer to avoid copying the `Queue` data structure and its `Items`? How can a client check the preconditions of the `Get_Next_Item` and `Finish_Iterator` operations? We answer these and other questions below by considering how to implement the three main operations.

Figure 4 traces an example of the effects of the above sample client code segment. It shows both the abstract models of `i`, `q`, and `x` (to illustrate the abstract behavior), and possible concrete representations of `i` and `q` (to support performance claims). In this case `q` is a `Queue` of `Integers`.[6] The figure shows a typical `Queue` representation, which is a pointer to a record containing two fields: `f` points to the front node of the queue and `r` to the rear. The representation of an `Iterator` is identical except that there is an additional field in the representation record: `p` points to the node whose `Item` is presently held by the client (if any). These concrete representations are only illustrative. Many others are possible and would achieve the claimed performance.

In the top row of Figure 4, just before execution of the sample client code begins, `i` and `x` might have any values. For example, `i` might have an initial value for type `Iterator` and `x` might be 17, as illustrated. In the second row, after the call to `Start_Iterator`, `x` again might have any value, say 0. The initial value of `x` is immaterial, and the specification does not say what `Start_Iterator (q, i, x)` returns for `x` except that it remembers that value in `i.present` (the second component of the triple that denotes `i`'s abstract value). The next three rows of the figure show the situation after the three calls to `Get_Next_Item` that occur in the case that the original `q` is modeled by the three-element string ‹ 9 6 90 ›. The value of `x` after the call to `Finish_Iterator` is certainly 0, because the specification says that operation consumes `x`; i.e., it leaves `x` with an initial value of type `Integer`, which we have assumed to be 0.

6 This makes it easy to understand the operation of the iterator, but it also makes the example too simple to illustrate the importance of not copying an `Item` — which may be a far more complex type than `Integer`! We opted for ease of understanding in choosing the example.

One aspect of Figure 4 may seem mysterious: Why are there top-level pointers to the records representing an `Iterator` and a `Queue`? These pointers are not strictly necessary in order to achieve the claimed performance; swapping of `Iterators` and `Queues` still would require only constant time even without this extra level of indirection. However, it is important for implementing swapping in a uniformly bounded time, and for code-sharing among instances of generics, as noted in [Harms 91].

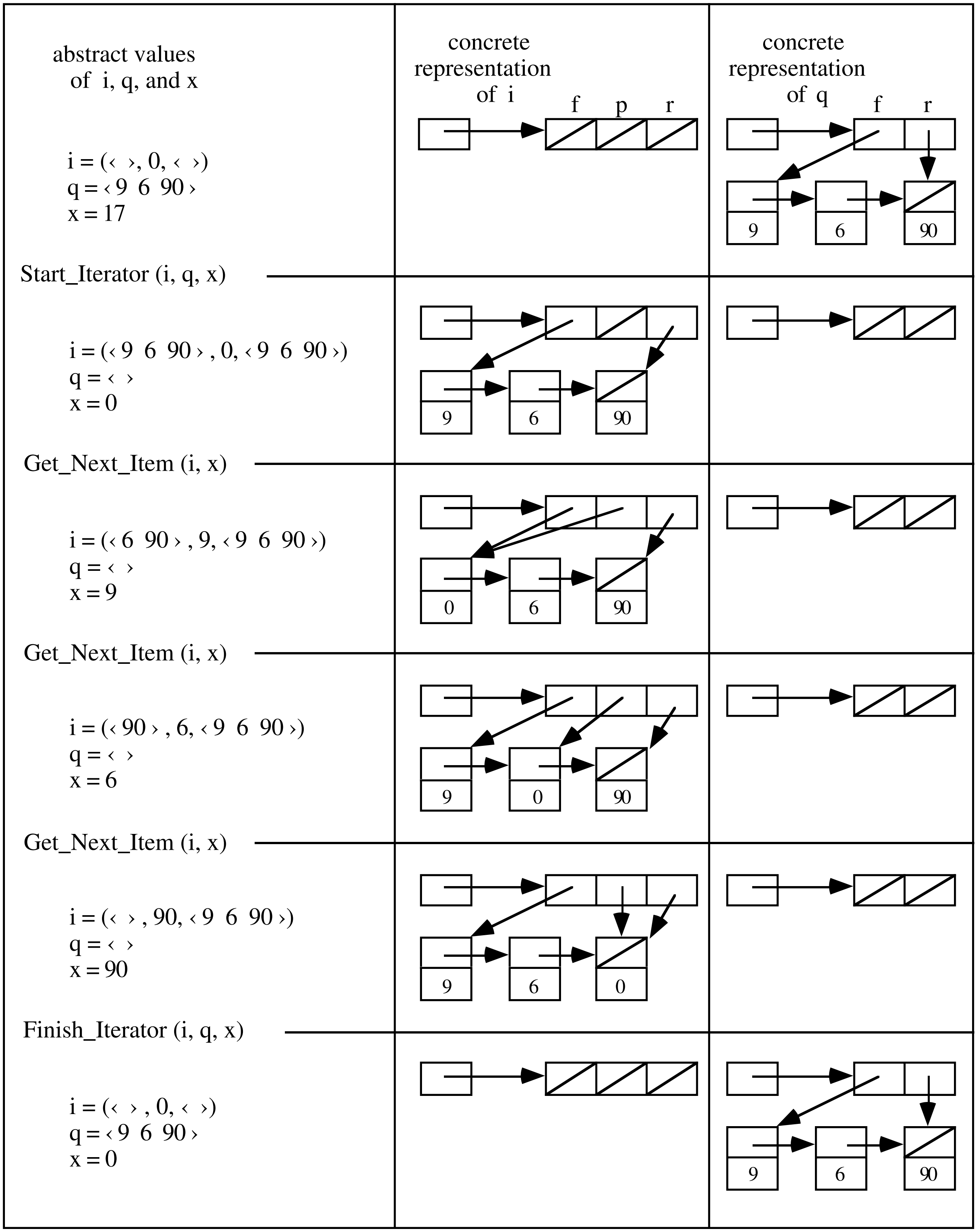

Figure 4: Sample Execution for Design #2

In the abstract explanation of `Start_Iterator`, the original value of `q` is remembered in `i.future`, from which `Items` subsequently are to be dispensed to the client by `Get_Next_Item`. An implementation of `Start_Iterator` in Design #2 need not copy the original `Queue` data structure in order to achieve this effect. It can acquire the original value of `q` by swapping. `Start_Iterator` is designed to consume `q` in order to support this implementation.

On first reading it might appear that `Start_Iterator` should have to copy `q` in order to satisfy the postcondition clause "`i.collection = #q`." This also is not the case. It is critical to recognize that `i.collection` is part of the *abstract* state of an `Iterator`. There is no implication that the *concrete* implementation of an `Iterator` must explicitly represent `i.collection`, and indeed none of the other operations demands that `i.collection` be represented explicitly, as explained below. This "correspondence" trick (which is routine in model-oriented specifications of complex abstractions) spares the implementer of `Start_Iterator` from having to copy the data structure that represents `#q`.

`Start_Iterator` also produces an `Item x`, whose value may be anything upon return. The postcondition clause "`i.present = x`" means that the `Item` value returned to the client is remembered as part of the `Iterator`'s state. As above, note that `i.present` is only part of the abstract state of an `Iterator` and need not be represented explicitly unless some operation's implementation calls for that; none does in this case.

Similarly, `Get_Next_Item` need not copy an `Item`. Its precondition "`i.present = x`" requires that the client pass in as `x` an `Item` equal to the one most recently returned by `Start_Iterator` or `Get_Next_Item`. The implementation can merely put this value back into the `Queue` data structure (in the node referenced by field `p` in Figure 4) and return the next `Item` by swapping it out of the structure. Again, there is no need for copying because the `Item` returned must be passed to the next call to `Get_Next_Item`, and so on.

When iteration is completed the client calls `Finish_Iterator`. This operation's precondition requires that the client give back the one outstanding `Item` (whose value is `i.present`), at which point the implementation has the entire data structure and all the `Items` in the original `Queue`. It simply swaps this with parameter `q` to achieve the stated postcondition.

One point worth noting is that no code — either in the client or in the implementation — *checks* the clause "`i.present = x`" at the beginning of a call to `Get_Next_Item` or `Finish_Iterator`. In fact, because there is no operation that reveals the value of `i.present`, a client could not write such code without copying `Items`; nor could an implementer of `Queue_Iterator_Template`. Thus the only means for a client to be sure that no preconditions are violated is to be able to *prove* that the code to process `x` does not change `x`.

Without the precondition on `Get_Next_Item` and `Finish_Iterator` no such proof obligation would be raised in an arbitrary client program. While it then might be possible to verify a particular use of the swapping-based implementation, there would be no way to separate a proof of correctness of the implementation from that of the client program. Therefore, we could not prove the correctness of this implementation in a modular fashion, and we could not declare the swapping-based implementation of `Queue_Iterator_Template` to be correct out of the context of a particular client. The feasibility of such a modular correctness proof was one of the primary objectives of our design.

Critique — While Design #2 has a more complex specification than Design #1, its swapping-based implementation is straightforward and efficient. However, experience using the specification of Design #2 in modular correctness proofs suggests some minor changes. Most importantly, with Design #2 it is cumbersome to show in the sample client program that the code to process `x` actually is executed for every item in the original `Queue q`. The proof relies on a loop invariant that keeps track of the `Items` that have been processed and relates them to the `Items` in `i.future` and `i.collection`. It is possible to introduce an adjunct variable for each loop to keep track of the processed `Items`, but it is more convenient to include support for this in the specification. This and other minor modifications are discussed in the next section.

3.3. Design #3

In Figure 5 we add two other fields to the mathematical model of an `Iterator`: a string of `Items` (called `past`) that records the `Items` that have been returned to the client through `Get_Next_Item`, and an `Item` (called `primer`) that records the `Item` that the client passed into the first call to `Start_Iterator`. We also add a precondition to `Start_Iterator` to guarantee that the `Iterator i` satisfies its initial condition, and modify the postcondition of `Finish_Iterator` so `i.primer` is returned in `x`. The reasons for these changes are discussed below.

```
concept Queue_Iterator_Template enhances Queue_Template
    interface
        type Iterator is modeled by tuple
            (
                past: STRING (Item)
                present: Item
                future: STRING (Item)
                collection: STRING (Item)
                primer: Item
            )
            exemplar i
            initially   i.past = EMPTY  and
                        Item.init (i.present)  and
                        i.future = EMPTY  and
                        i.collection = EMPTY  and
                        Item.init (i.primer)
        operation Start_Iterator (alters i: Iterator,
                consumes q: Queue, consumes x: Item)
            requires    Iterator.init (i)
            ensures     i.past = EMPTY  and
                        i.present = x  and
                        i.future = #q  and
                        i.collection = #q  and
                        i.primer = #x
        operation Finish_Iterator (consumes i: Iterator,
                produces q: Queue, alters x: Item)
            requires    i.present = x
            ensures     q = #i.collection  and
                        x = #i.primer
        operation Get_Next_Item (alters i: Iterator,
                alters x: Item)
            requires    i.future /= EMPTY  and
                        i.present = x
            ensures     i.past = #i.past ∘ x  and
                        i.present = x   and
                        x ∘ i.future = #i.future  and
                        i.collection = #i.collection  and
                        i.primer = #i.primer
        operation Is_Empty (preserves i: Iterator)
                returns empty: Boolean
            ensures     empty  iff  i.future = EMPTY
end Queue_Iterator_Template
```

Figure 5: Queue Iterator Design #3

Discussion — Here is a sample of client code for iteration using Design #3 (see Appendix A for complete client examples):

```
Start_Iterator (i, q, x)
maintaining     i.past ∘ i.future = #i.past ∘ #i.future  and
                i.present = x  and
                i.collection = #i.collection  and
                i.primer = #i.primer
while not Is_Empty (i) do
    Get_Next_Item (i, x)
    (* code to process x without changing x *)
end while
Finish_Iterator (i, q, x)
```

In this sample code we include the loop invariant in a **`maintaining`** clause, which may be considered an extra syntactic slot in the while-loop construct. The notation means that at the beginning of each iteration of the loop, the concatenation of `i.past` and `i.future` equals their concatenation just before the loop is first encountered; that `i.present` equals `x`; and that `i.collection` and `i.primer` equal their values just before the loop is first encountered.

Clearly this invariant is true at the start of the first iteration. It is easy to show it is true for an arbitrary iteration if and only if the code to process `x` does not change `x`. With the addition of the `past` field to an `Iterator`'s abstract state, it also is easy to show that all `Items` in the original `Queue q`, and only those `Items`, are processed by the loop.

The other changes in Design #3 support a general principle of the swapping paradigm: There are advantages in simplified reasoning about program behavior and in the performance of storage management activities if temporary variables in a program act as *catalyst* variables [Harms 91]. A catalyst variable is one that is necessary to carry out a computation but experiences no (net) change in value from the beginning to the end of that computation. In the expected use of `Queue_Iterator_Template` we would like to make sure the local variables `i` and `x` are catalysts. Notice that this need not be true for Design #2; `i` and `x` might start out with any values whatsoever before `Start_Iterator`, and their values after `Finish_Iterator` might be different.

In Design #3 we therefore require that `Iterator i` be an initial value for its type before the call to `Start_Iterator`. `Finish_Iterator` consumes `i`, leaving it again as an initial value for its type. Also in Design #3, we record the value of x that is passed to Start_Iterator and restore that value, i.primer, in Finish_Iterator. Now both i and x act as catalyst variables.

4. Variations and Extensions

There are several variations and extensions of this approach to iterators that merit discussion. We briefly mention a few of them here. The reader is referred to [Edwards 90] for a more detailed treatment.

4.1. Early Exit from Iteration

A client program that exits from an iteration loop before the Iterator is empty poses no particular problems; see Appendix A for an example. The main reason for not layering Queue_Iterator_Template on top of Queue_Template, however, is efficiency in this special case. If all the Queue_Template operations take constant time, then all the layered operations take constant time except Finish_Iterator. In the case of an early exit from an iteration, Finish_Iterator takes time proportional to the number of Items remaining in the Iterator. A direct implementation in which the new operations have access to the underlying Queue representation (as in Figure 4) achieves constant-time performance for all operations.

4.2. Different Orders of Iteration

It is easy to modify the specification of Design #3 to define an Iterator type that presents the Items to the client in a different order. We simply define a (mathematical) function:

$$\pi : \text{STRING (Item)} \rightarrow \text{STRING (Item)}$$

that yields the appropriate permutation of the Items. Then we change the **ensures** clause of Start_Iterator as underlined:

```
operation Start_Iterator (alters i: Iterator,
        consumes q: Queue, consumes x: Item)
    requires    Iterator.init (i)
    ensures     i.past = EMPTY  and
                i.present = x  and
                i.future = π(#q)  and
                i.collection = #q  and
                i.primer = #x
```

If π is a particular permutation, then the client knows the order in which the `Items` should be expected. However, if π is only known *to be* a permutation (i.e., to have the appropriate mathematical properties), then the implementer has complete freedom to present the `Items` from the `Iterator` in any convenient order. This variation can lead to improved efficiency in some circumstances, and is most appropriate when there is no "natural" order of iteration, as with sets and multisets [Pearce 92].

4.3. Other Collections

To specify `Iterators` for collections that are not modeled as mathematical strings, we can use the same approach as suggested above to get different orders of iteration. For example, suppose we want to define an `Iterator` type for a `Set` abstraction, and our program type `Set` is modeled by a mathematical set of `Items`. First we define a (mathematical) function:

```
f : SET (Item) → STRING (Item)
```

that yields the desired ordering of the `Items`. This function has the property that every `Item` in its argument (a set) occurs exactly once in the string the function produces. As with the different orders of iteration, this function might be explicit or it might simply be known to have this property. In either case we can write the specification of Figure 6, where changes from Figure 5 (not name changes from "`Queue`" to "`Set`," but the substantive ones) are underlined.

```
concept Set_Iterator_Template enhances Set_Template
    interface
        type Iterator is modeled by tuple
            (
                past: STRING (Item)
                present: Item
                future: STRING (Item)
                collection: SET (Item)
                primer: Item
            )
            exemplar i
            initially   i.past = EMPTY  and
                        Item.init (i.present)  and
                        i.future = EMPTY  and
                        i.collection = NULL  and
                        Item.init (i.primer)
        operation Start_Iterator (alters i: Iterator,
                consumes s: Set, consumes x: Item)
            requires    Iterator.init (i)
            ensures     i.past = EMPTY  and
                        i.present = x  and
                        i.future = f(#s)  and
                        i.collection = #s  and
                        i.primer = #x
        operation Finish_Iterator (consumes i: Iterator,
                produces s: Set, alters x: Item)
            requires    i.present = x
            ensures     s = #i.collection  and
                        x = #i.primer
        operation Get_Next_Item (alters i: Iterator,
                alters x: Item)
            requires    i.future /= EMPTY  and
                        i.present = x
            ensures     i.past = #i.past ∘ x  and
                        i.present = x   and
                        x ∘ i.future = #i.future  and
                        i.collection = #i.collection  and
                        i.primer = #i.primer
        operation Is_Empty (preserves i: Iterator)
                returns empty: Boolean
            ensures     empty  iff  i.future = EMPTY
end Set_Iterator_Template
```

Figure 6: Set Iterator Design

The `Iterator` type in Figure 6 is still modeled so that `past` and `future` are strings of `Items`. However, `collection` is modeled by the same mathematical type as the underlying collection over which we are iterating, in this case a set. The reasons for remembering the original collection in the `Iterator`'s model are that it is easy to understand, and that we need to assume nothing about f in order to specify `Finish_Iterator` (e.g., that we know its inverse).

5. Conclusions

Previously published iterator designs, particularly for use with modern imperative languages such as Ada or C++, are unsatisfactory along several dimensions. The iterator design that was incrementally developed in Section 3 addresses the deficiencies of prior approaches in the following specific ways:

- It is designed to support efficient implementations: neither the implementer nor the client needs to copy the data structure representing the collection or any of the individual items inside the collection.

- Its abstract behavior (including the non-interference property) is formally specified.

- Its implementations and clients can be verified independently (modularly, in the sense of [Ernst 91]).

- It can be considered a schema that can be modified in an obvious way to produce iterator abstractions for other collections, so that all iterator abstractions in a system can share a common interface model.

Because of these advantages, the final iterator design presented in Section 3.3 (Figure 5) should be considered as a baseline proposal for a common interface model for iterator abstractions. This baseline supports sequential access to the individual items of a collection without allowing the collection to be modified, and is robust enough to be generalized to any container structure where such iterations are feasible. Furthermore, it can be generalized to more complex iteration schemes as described in Section 4. As a result, this new design shows how iterators can be abstracted and encapsulated in modern imperative languages to best support modular programming, as well as modular reasoning about program behavior.

Acknowledgments

We are indebted to Wayne Heym, Joe Hollingsworth, Bill Ogden, Murali Sitaraman, and Stu Zweben for many helpful comments. The first two authors also acknowledge the support of the National Science Foundation, CCR-9111892.

References

[Bishop 90] Bishop, J.M., "The effect of data abstraction on loop programming techniques," *IEEE TSE 16*, 4 (Apr. 1990), 389-402.

[Booch 87] Booch, G., *Software Components with Ada*, Benjamin Cummings, 1987.

[Cameron 89] Cameron, R.D., "Efficient high-level iteration with accumulators," *ACM TOPLAS 11*, 2 (Apr. 1989), 194-211.

[Edwards 90] Edwards, S., *An Approach for Constructing Reusable Software Components in Ada*, IDA paper P-2378, Institute for Defense Analyses, Alexandria, VA, Sept. 1990.

[Edwards 91] Edwards, S.H., "Common interface models for components are necessary to support composability," *Proc. 4th Ann. Workshop on Softw. Reuse*, SPC, Herndon, VA, Nov. 1991.

[Ernst 91] Ernst, G.W., Hookway, R.J., Menegay, J.A., and Ogden, W.F., "Modular verification of Ada generics," *Comp. Lang. 16*, 3/4 (1991), 259-280.

[Harms 91] Harms, D.E., and Weide, B.W., "Copying and swapping: influences on the design of reusable software components," *IEEE TSE 17*, 5 (May 1991), 424-435.

[Hollingsworth 91] Hollingsworth, J.E., Weide, B.W., and Zweben, S.H., "Confessions of some used program clients," *Proc. 4th Ann. Workshop on Softw. Reuse*, SPC, Herndon, VA, Nov. 1991.

[Hollingsworth 92] Hollingsworth, J.E., *Software Component Design-for-Reuse: A Language-Independent Discipline Applied to Ada*, Ph.D. diss., Dept. of Comp. and Inf. Sci, Ohio State Univ., Columbus, Aug. 1992.

[LaLonde 89] LaLonde, W.R., “Designing families of data types using exemplars,” *ACM Trans. on Prog. Lang. and Syst. 11*, 2 (1989), 212-248.

[Lamb 90] Lamb, D.A., “Specification of iterators,” *IEEE TSE 16*, 12 (Dec. 1990), 1352-1360.

[Liskov 77] Liskov, B, Snyder, A., Atkinson, R., and Schaffert, C., “Abstraction mechanisms in CLU,” *CACM 20*, 8 (Aug. 1977), 564-576.

[Muralidharan 90] Muralidharan, S., and Weide, B.W., “Should data abstraction be violated to enhance software reuse?” *Proc. 8th Ann. Natl. Conf. on Ada Tech.,* ANCOST, Inc., Atlanta, GA, Feb. 1990, pp. 515-524.

[Pearce 92] Pearce, T.W., and Lamb, D.A., “The property vector specification of a multiset iterator,” *Proc. 14th Intl. Conf. on Softw. Eng.*, ACM/IEEE, May 1992.

[Shaw 77] Shaw, M., Wulf, W.A., and London, R.L., “Abstraction and verification in Alphard: defining and specifying iteration and generators,” *CACM 20*, 8 (Aug. 1977), 553-564.

[Tracz 89] Tracz, W., “Parameterization: a case study,” *Ada Letters 9*, 4 (May/June 1989), 92-102.

[Weide 91] Weide, B.W., Ogden, W.F., and Zweben, S.H., “Reusable Software Components,” in *Advances in Computers, vol. 33*, M.C. Yovits, ed., Academic Press, 1991, pp. 1-65.

[Wing 90] Wing, J.M. “A specifier’s introduction to formal methods,” *Computer 23,* 9 (Sept. 1990), 8-24.

Appendix A: Client Examples for Design #3

Here is a simple example client for Design #3: An operation to copy a `Queue` using an `Iterator`:

```
operation Copy (preserves q1: Queue, produces q2: Queue)
    ensures     q2 = q1
begin
    variable
        garbage: Queue, i: Iterator, x1: Item, x2: Item
    q2 :=: garbage
    Start_Iterator (i, q1, x1)
    maintaining     i.past o i.future = #i.past o #i.future  and
                    i.present = x1  and
                    i.collection = #i.collection  and
                    i.primer = #i.primer  and
                    q2 = i.past
    while not Is_Empty (i) do
        Get_Next_Item (i, x1)
        Copy_Item (x1, x2)
        Enqueue (q2, x2)
    end while
    Finish_Iterator (i, q1, x1)
end Copy
```

The next example illustrates simultaneous iteration over two collections, and a possible early exit from an iteration loop: An operation to determine whether two Queues are equal.

```
operation Are_Equal (preserves q1: Queue, preserves q2: Queue)
        returns equal: Boolean
    ensures     equal  iff  q2 = q1
begin
    variable
        i1: Iterator, i2: Iterator, x1: Item, x2: Item
    equal := true
    Start_Iterator (i1, q1, x1)
    Start_Iterator (i2, q2, x2)
    maintaining     i1.past ∘ i1.future = #i1.past ∘ #i1.future  and
                    i1.present = x1  and
                    i1.collection = #i1.collection  and
                    i1.primer = #i1.primer  and
                    i2.past ∘ i2.future = #i2.past ∘ #i2.future  and
                    i2.present = x2  and
                    i2.collection = #i2.collection  and
                    i2.primer = #i2.primer  and
                    equal = (i1.past = i2.past)
    while equal and not Is_Empty (i1) and not Is_Empty (i2) do
        Get_Next_Item (i1, x1)
        Get_Next_Item (i2, x2)
        equal := Are_Equal_Items (x1, x2)
    end  while
    if equal and (not Is_Empty (i1) or not Is_Empty (i2)) then
        equal := false
    end  if
    Finish_Iterator (i1, q1, x1)
    Finish_Iterator (i2, q2, x2)
end Are_Equal
```