Iterators: Opportunities Lost, Lessons Learned.

John Beidler

Computing Sciences Dept.
University of Scranton
Scranton, PA 18510
Tel: (717) 941-7446
Email: beidler@cs.uofs.edu

Abstract

Iterators, as defined by Booch, have restricted the potential that is available through iterators to encourage software reusability. Iterators present a rich opportunity for to improve the reusability of generics, but this opportunity requires some trade offs between making iterators general and keeping them usable. It also requires the development of educational material on how to make the best use of iterators.

Keywords: Reuse, data structure components, iterators, generics

Workshop Goals: Designing reuseable components and tools, teaching reuse.

Working Groups: Low level(intermediate level) reuse, reuse education

1 Background

Booch's book **Software Components with Ada**, established a standard for the encapsulation of objects in Ada packages. Normally, the operations on objects are classified into two categories, constructors and selectors. However, Booch expanded upon this classification by including a third category, iterators. An iterator is defined by Booch as "an operation that permits all parts of an object to be visited". Throughout his book Booch addresses the development of iterators over various objects, but due to the nature and goals of the book, the construction and potential uses of iterators in Ada packages are never fully developed.

We take a slightly different view of iterators. Instead of viewing iterators as a third category of operations on objects, we view iterators as an intermediate building block, a reusable tool, that may be used to fabricate constructors and selectors.

Unfortunately, Booch's book established a precedent on how iterators are viewed by the Ada community. This precedent is continued with another set of commercially available components, the GRACE components from EVB Software Engineering. As such, both the Booch and GRACE components contain at most one iterator per data structure. In fact, there are several possible iterators for most data structure. For example, both the Booch and GRACE components supply only one iterator for binary trees. The iterator is a depth first iterator. It is easy to demonstrate that there are at least 14 standard variations of depth first iterators and several other families of binary tree iterators.

In several papers we built upon the concept presented by Booch and demonstrates that some care must be taken to build iterators in order to make them more versatile to potential users of the packages that contains them. The result of providing more useful iterators a potentially greater amount of low and intermediate level software reuse. Iterators also encourages the decomposition of certain algorithms in a standardized fashion leading to improved software readability.

The full use of iterators stand as a lost opportunity to the Ada83 community. Perhaps it is time to revisit this subject and, perhaps, learn some lessons about the encapsulation of reusable components and tools that may provide insight into reuse at other levels.

Further, Ada9X presents us with new opportunities. In particular, many of the new features in Ada9X provide new approaches to encapsulation that allow us, under many circumstances, to avoid the use of generics and limited private types, two features that are at the heart of safe encapsulation in Ada83. How will these new features simplify the client's use of encapsulated components and tools?

2 Position

We propose that Booch's view of iterators should be abandoned for two reasons. First, the view, by being too narrow, discourages the study of iterator, hence a lost opportunity that might provide some insight into low level and intermediate level reuse issues. Specifically, we feel that a study of the issues surrounding the encapsulation of iterators may provide some insight into the general issues of encapsulating reusable software.

Second, as various generic components, and data structure components in particular, are redesigned to take full advantage of Ada9x, we should review the features that are available in the language to properly support better and more efficient reusability. The potential richness of the study of how we should encapsulate iterators and provide other means of access and manipulating objects within a structure might provide us with additional insight into other, higher level, reuse.

A case worth analyzing is the issue of appropriate encapsulation of a sufficient collection of binary tree iterators that will assist clients when they reuse a data structure. We have studied four families of binary tree iterators:

- Depth First Iterators
- Breadth First Iterators
- Binary Search Iterators
- Priority Queue Directed Iterators

An analysis of these families of iterators provides insight into the trade-offs that must be made between generality and usability of an encapsulated iterator. If an iterator is encapsulated as a generic procedure, within a generic package, what guidelines might be established regarding the use of the formal instantiation parameters, the formal parameters of the instantiating procedure(s), and the iterator's formal parameters?

An equally important issue is the issue of training software developers to properly use iterators. It has been our experience that clients might have to reorganize their (low level) design, or at least restructure parts of algorithms, to make use of an encapsulated iterator. However, with consistent iterator design guidelines, we may be able to easily educate software developers regarding how to redesign algorithms to take advantage of encapsulated iterators. For the last two years we have experimented with teaching about the use of iterators, but with poor success, in an advanced data structures course. We have llearned that good iterator design is not enough, it must be combined with appropriate education in how to design with reuse in mind. We plan to do include a balanced presentation that addresses both side, designing reusable software and exploiting reusable software, this in the Fall as part of an advanced data structure course.

In Ada 83, we have adopted the following design guidelines for the construction of iterators:

- If the iterator allows the instantiating procedure to have an iterator control, frequently called the **Continue** parameter, the control should be passed as an **in out** parameter and given an initial value by the iterator.
- A pass through parameter must be included to allow the client to pass information between instantiating procedures and the client procedure that uses the instantiated iterator.
- Are the positions and use of parameters consistent? Consistent within the encapsulation of an iterator? Consistent across iterators within other packages?
- Clients must be provided with guidelines regarding the safe and efficient instantiation of generic procedures that encapsulate iterators.

2.1 Comparison

As we move to Ada9x (or C++), how do we build upon the lessons learned about iterators and take full advantage of the new opportunities available in Ada9x? Even if the issues of iterators are redressed in the commercially available component sets, can we expect other advances as well? Through an ARPA Software Engineering and Ada Education Grant we build a set of iterators in

Ada83 that address the issue of iterators, as well as several other issues that have been overlooked in the popular commercial component sets.

Another issue not addressed by the commercial components is the issue of "**in place**" access to objects within a structured object. For example, typical access to an object within a structured object is through copy and modify functions and procedures. Yet "in place" access is easy to supply, and provide a efficient and safe means of changing the value of an object within an object.

To create an analogy (perhaps a bit extreme), how does a dentist fix a cavity in a tooth? Does he remove the tooth, fix it, then put it back? Of course not, he does not remove the tooth, he fixes it (changes its value) while leaving the tooth where it belongs. If Booch and EVB had a data structure called "gums" that contained objects called "tooth", they would pull the tooth out, fix it, and replace it, which is not an efficient way to proceed, rather than providing "in place" access to the tooth.

2.2 Ada9X and Iterators

The object oriented features in Ada9X provide us with an opportunity to experiment with the encapsulation of reusable components without the use of generics and limited private types. Many current reuse problems center on misunderstandings by clients of the need for limited private types. **Finalization** in Ada9X provides a safe way of using private types in Ada9X, when similar safety was accomplished in Ada83 only through the use of limited private types.

Clients also faced problems with generic instantiation in Ada83. Specifically, the specific placement of instantiations caused various problems with a number of Ada compilers. With the judicious use of **tagged types** and **Access parameters**, various data structure components may be encapsulated in Ada9X in a non-generic package! Further, instead of using generic procedures with the package to encapsulate iterators, iterators may be encapsulated as non generic procedures, using a tagged type that may be polymorphed to pass client information to and from the client's procedure that is controlled by the iterator. The client's procedure may be passed to the iterator as an Access type.

3 Biography

John Beidler is a Professor of Computer Science at the University of Scranton. He served there as department chair for over fifteen years before stepping down as chair to spend more time on research. He currently serves as Director of the Masters Program in Software Engineering. His research interests are in programming support environments, software reusability, data structures and algorithms, and computational complexity.