Iterators Reconsidered

Jason O. Hallstrom, Scott M. Pike, and Nigamanth Sridhar
Computer and Information Science
Ohio State University
2015 Neil Ave
Columbus OH 43210-1277 USA
{hallstro,pike,nsridhar}@cis.ohio-state.edu

ABSTRACT

Software developers are eager to increase the scale of their software products at a rate proportional to the growth of computing resources. With memory, bandwidth, and computing power doubling roughly every eigtheen months, development approaches that are not based on compositional reasoning techniques can not be used to engineer the systems of tomorrow. The enormous scale of these projects far outstrips our ability to understand them using ad-hoc approaches.

Industry best practice recognizes the importance of component reuse, but the emphasis is weighted heavily on the reuse of component code, often times neglecting the need to reuse the effort that went into understanding the component's behavior. That is, any scalable software engineering discipline must provide mechanisms for reusing software components, as well as mechanisms for reusing the reasoning effort required to use those components.

This paper examines the Iterator pattern with regard to compositional reasoning. The approach, touted as industry best practice, is shown to provide ample opportunity for breaking the principles of encapsulation. These various hazards are briefly described, and several techniques for ensuring safe use of the pattern are explored.

1. GOD'S LAW

In March 2001, Silicon Valley hosted the flagship conference "ACM1: Beyond Cyberspace" [10]. With 14 plenary speakers forecasting the next *50 years* of computing, it was a veritable pep-rally for the knights-errant of technology. Like most crusades, there was a motivating belief behind which the movers and shakers could unite. In the case of ACM1, that belief was in Moore's Law: the unparalleled doubling of storage, bandwidth, and computing power roughly every 18 months. The inevitably smaller, faster, and cheaper computing devices of tomorrow were depicted as a holy grail within our reach. Astrophysicists, oceanographers, biogeneticists, and computer scientists alike hopped on the speculation bandwagon of promised technology. The future — so we were told — would inherit ubiquitous computing, intelligent mobile agents, synoptic knowledge of the oceans, and interplanetary exploration. But somehow a material question got drowned in the wake of enthusiasm: How would we ever engineer such systems?

Conspicuously understated was the so-called "God's Law" expressed by William Buxton [2] as the fact that human capacity for understanding is limited. Developing applications at the pace of Moore's Law will ultimately dwarf our raw abilities to manage their scale. It is well-known that problems of scale in software cannot be solved simply by adding more people [1]. To borrow a metaphor from Gerald Weinburg: you can't make a baby in one month by putting nine women on the job. The complexity of large-scale software simply outstrips the capacity of our unaided intellectual abilities. This asymmetry underscores the importance of **modular techniques that support local component certification and compositional reasoning reuse about system behavior and correctness**.

In principle, well-designed software components can help software engineers understand, and reason soundly about, the behavior of component-based software systems. Achieving sound reasoning, however, is more subtle than it might seem. The problem is tricky. Software components that are not designed to support modular reasoning can — and often do — exhibit unanticipated interactions when integrated into an overall system. Side-effects of component interference make it intractable to predict overall system behavior by composing local properties of the system's subcomponents [9]. Put otherwise, component-based development is not necessarily scalable development.

Reuse, of course, is central to the predictable construction of scalable systems. Fortunately, reuse is fashionable these days, even if component-based reasoning methods are not. Current practice is pervaded by reuse strategies including design patterns [5], software infrastructures (EJB, COM+, CORBA), and code libraries (java.util, C++ STL). Unfortunately, these approaches typically fall short of supporting abstraction, encapsulation, and information hiding — properties essential to component-based methods for modular composition and *reasoning*. As such, the landscape of the component revolution — as witnessed by current practice — is like quicksand: easy to get stuck in, and unable to support the intellectual burden of reasoning about system behavior. A sound foundation for scalable development depends on local, compositional reasoning methods to serve as footholds by which we may climb above the mounting complexity of

Fifth ICSE Workshop on Component-Based Software Engineering. Orlando FL. May 19–20 2002.

our ever-growing systems.

One common misconception is that we appear to know how to reason about small systems, so therefore we should be able to apply the same techniques directly to large systems. In his Notes on Structured Programming [4], Dijkstra exposes the fallacy of this argument:

> Apparently we are too much trained to disregard differences in scale, to treat them as "gradual differences that are not essential". We tell ourselves that what we can do once, we can also do twice and by induction we fool ourselves into believing that we can do it as many times as needed, but this is just not true! A factor of a thousand is already far beyond our powers of imagination!

Put otherwise, the techniques that allow reasoning about a program with ten lines of code cannot necessarily be applied directly to a program with ten thousand lines of code. Tractable reasoning about large systems requires techniques for leveraging our reasoning about smaller, locally certifiable components into a compositional certification of the overall system resulting from component integration. Code reuse and reasoning reuse must go hand in hand. Differences in the scale of software systems cannot be disregarded if we want to continue along the path to building larger and larger systems. We have to learn to embrace reuse in more dimensions than just code reuse; reasoning reuse must be equally as important if we want to overcome the complexities of scale.

The remainder of this paper sketches a silhouette of Iterators as a composition mechanism that supports code reuse without supporting reasoning reuse. Iterators are touted as a so-called "best practice" in software engineering, both as a codified design pattern [5] and as part of the ANSI C++ Standard Template Library [8]. As such, Iterators are representative of mainstream currents of code reuse in present-day software practice. Section 2 presents a brief overview of iterators and their uses. In Section 3, we show how iterators can thwart compositional reasoning by breaking encapsuation. We present some desiderata for safe use of iterators in Section 4, and summarize our conclusions in Section 5.

2. ITERATORS: A BEST PRACTICE?

The Iterator patter [5] describes a technique for exposing individual elements of a container class without exposing the underlying container representation. Container classes supporting the pattern provide one or more factory methods used to generate the appropriate *iterator*, depending on how the container elements will be traversed. A binary tree container, for example, might provide one factory method for creating pre-order iterators, another for creating post-order iterators, and a third for creating in-order iterators. The iterators encapsulate the state required to perform each traversal; multiple tree traversals can therefore be performed simultaneously.

Iterators come in two basic flavors, depending on the access-control tastes of the client. If the iterator client does not need to control the iteration progress, an *internal iterator* is appropriate. Iterators of this type will typically provide only a single iterate(...) method, which, when invoked with a particular operation, will apply the operation to each element of the container. An iterator of this type might be used to increment every integer in a Set container. In this case the iteration is under the control of the iterator; the client is only concerned with applying a particular operation to the elements of a container.

Not all clients are so passive. The control-hungry lot desire a richer interface for manipulating the behavior of the iterator. *External iterators* satisfy this desire by providing methods for navigating the elements under the iterator's control. Typical interface methods include: moveFirst(), moveNext(), currentItem(), and isDone(), all with their obvious meanings. Given an algorithm that works on linear inputs, an external iterator can be viewed as a composition mechanism – a mechanism for grafting an algorithm onto an arbitrary container class. In Java, for example, iterators are the *glue* that connect sorting algorithms to binary trees, sets, hash tables, etc.

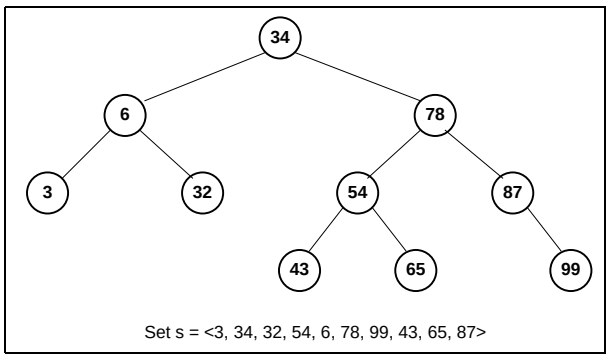
What is the itch that iterators are trying to scratch? Simply put, they seek to decouple algorithms from containers. This is acheived by sequentially exposing the elements of an arbitrary container through a *standard interface*, which can be presupposed by various algorithms. In the C++ Standard Template Library, iterators are considered a generalization of pointers that can enable clients to traverse an arbitrary container component in the same way that they would traverse an ordinary C array [8]. As such, decoupling is effectively realized by the level of indirection introduced by pointers. As we shall illustrate, however, the exposure of external aliasing is also the root of breaking the encapsulation barriers of containers. Based on the observed usage of iterators in practice, the benefits of iterators are well understood by practitioners. It is unfortunate that the hazards of their usage are not.

3. NON-COMPOSITIONAL REASONING

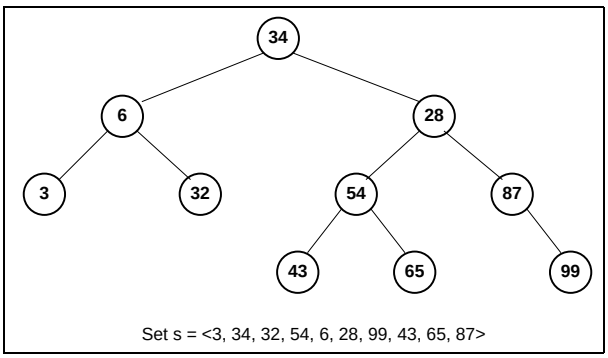
The iterator approach has potential to break encapsulation by exposing aliases into the internal representation of a container component. Thirty years of research in information hiding dictates that this practice undermines sound component-based design and development [7]. The reason is simple: breaking encapsulation exposes both the component *and its clients* to interference side-effects that can thwart modular reasoning about component behavior [6].

For example, consider a Set component implemented as a binary search tree (BST). The correctness of Set operations like Add and Remove depends on the binary search tree property being satisfied as a representation invariant. Any algorithm using an iterator to traverse the Set, however, has access to the key values of items contained in the underlying representation. Changing any key value can potentially violate the binary search tree property, so that entire subtrees become "lost" as unretrievable items from the Set. This is illustrated in Figure 1. Changing just one key value (from 78 to 28) in the Set presented in Figure 1(a) violates the invariant required for correct tree operations in Figure 1(b).

As another example, consider an iterator that increments every integer in a Set by one. Such an operation on the Set may be "safe" for BST-implementations, but it has disastrous consequences for the correctness of hashing implementations: the resulting values may no longer be located in their correct hash buckets! This is illustrated in Figure 2, where integers are hashed into buckets based on their value mod 3. Figure 2(a) shows the state of the representation prior to incrementing each integer via iteration. Figure 2(b) shows the representation after the iteration. Note that the

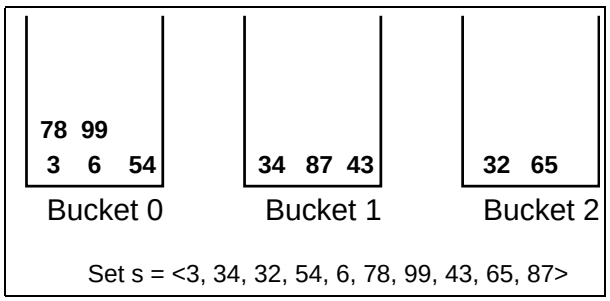


(a) Set s represented as a binary search tree

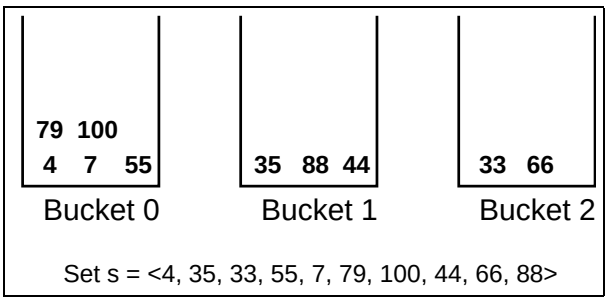


(b) Set s with key value 78 changed to 28

Figure 1: Binary Search Tree



(a) Set s represented as a hash table



(b) Set s after iterator incremented all values by 1

Figure 2: Hash representation

representation invariant is broken now, since none of the integers in the resulting Set is in its correct hash bucket.

By enabling external access to internal structure, iterators prevent reasoning about the correctness of these Set implementations in isolation; that is, they can not be validated *modularly* via local certifiability. As a composition mechanism, iterators facilitate code reuse by splicing algorithms onto arbitrary containers, but they open the door to aliasing side-effects that undermine reasoning reuse about the components under composition.

In Section 1, we claimed that the way to reason about large systems is to compose the reasoning arguments of the smaller constituent components in the system. However, as is evidenced by the foregoing example, iterators could potentially invalidate all of the reasoning arguments that have been built up until this stage. This would take us back to the point where we would have to restart from scratch in building a sound argument for why the system works. In effect, a single iterator in a system could potentially ruin all chances of modular reasoning in an otherwise sound system.

Practitioners may claim that although iterators are potentially dangerous, they can still be used safely. Such was the prevailing wisdom of best practices when goto statements were the talk of the town. The analogy here is more than skin-deep. Among its other ills, the goto statement could break procedural abstraction by enabling control to jump into unrestricted segments of code. Similarly, iterators can break component abstraction by enabling clients to jump into unrestricted segments of data. Of course, we never really got rid of jumps; we simply hid them from programmers by using compilers to enforce their safe introduction into executable code. So too, we do not wish to get rid of component traversals; we simply need to encapsulate safe mechanisms for accessing their data, lest we suffer the same fate as the goto cowboys: namely, intractable (i.e., non-compositional) reasoning about program behavior [3].

4. ITERATORS AND DATA PROTECTION

If encapsulation is the cement that holds modular verification techniques together, aliasing is the jack-hammer. Components which leak aliases into their internal representation typically cannot be verified in isolation; the Set component implemented as a binary search tree is an representative example. If the iterator used to traverse the elements of the Set returns aliases to the individual container elements, a single update of any element may violate the BST invariant. Thus, even if the Set implementation is correct in isolation, the iterator interface makes it vulernable to tampering when composed with client algorithms. The upshot is that modular verification efforts can be compromised.

Readers accustomed to programming in languages like Java might argue that neither the Set nor the iterator have

violated principles of encapsulation. After all, the implementation has not exposed aliases to its internal structure, only to the elements stored within that structure. With the exception of the scalar types, Java does not provide support for value-type variables — everything is a reference. Returning references to objects within a container class is common practice. How has encapsulation been violated?

While it may not be possible for a Set client to gain knowledge of its internal represenation, it is possible for a client to modify elements of the container without going through the container interface. Depending on the container representation, some updates may be safe, while others may violate the representation invariant. That is to say, the correctness of the Set implementation depends on the context in which it is used. Any attempts at proving the correctness of such a component must explicitly consider how the component is being used on a case by case basis. One small step for code reuse, one giant leap (backwards) for reasoning reuse.

Without abandoning the Iterator pattern altogether, there are some techniques which can be used to ensure the scalability of the approach. These techniques rely on a notion complementary to information hiding, namely, *information protection*. Iterators providing information protection guarantee that any element exposed through the iterator can not be modified without going through the associated container's interface. With this additional proof obligation in place, it is impossible for a client to violate the representation invariant. The iterator and the corresponding container class can now be verified independently of their client-side context of use.

One obvious approach to achieve information protection is to design iterators so that they yield values rather than references. If a client modifies a value returned by an iterator, no harm done, the original value is still preserved within the container. This approach may be viable for simple data types like integers or characters, but may be prohibitively expensive for larger, more complex objects. So how can an iterator provide efficient access to (potentially large) elements of a container class, while still achieving information protection?

When a container class is designed to hold items of a particular type, it is sometimes possible to return references to the container elements without violating information protection. In particular, when a container is designed to hold immutable objects, it is always safe to expose aliases to the contained elements. By definition, the value of an immutable object can not be modified once the object has been created.

Note that the immutable object approach is only applicable if the container is designed with knowledge of the components that it will hold. Because the correctness proof will be able to explicitly consider the various contained types, the immutability obligation can be dispatched. This does, however, impose a subtle constraint on other classes: derived classes of an immutable base class must themselves be immutable. If this were not the case, derived class objects could be placed in the container and modified without the container's knowledge, whereby reasoning is set back to square one.

Improved compiler support for type safety could prove useful here. Compilers designed to determine whether a given class is immutable could enforce the constraints discussed above, thereby extending the applicability of the foregoing approach to generic container classes. If a particular container class is tagged as element-immutable, the compiler can prevent the developer from adding mutable objects to the container. The generic container and iterator pair can now be modularly verified because the compiler guarantees that the container will never hold mutable objects.

Another approach to achieve information protection is to implement the iterator as a layered extension of the original container component. The iterator has access to the internals of the original aggregate *only* through the aggregate's published interface. This kind of *black-box reuse* restricts the scope of possible changes the iterator can make to the state of the aggregate. So long as the published interface allows only safe access to the aggregate, the iterator cannot render the representation invalid by enabling surreptitious changes.

Implementing the currentItem() method can be achieved by actually removing an item from the aggregate and returning it to the client. The moveNext() operation would then put the item back into the aggregate. With this implementation, however, care needs to be taken that items in the aggregate are not skipping or duplicated. To avoid such errors, an auxiliary aggregate object can be used to hold the processed items, and then swapped with the original aggregate at the end of the iteration.

The only safe way of using internal iterators is if the operation that the internal iterator applies on the items in the aggregate is read-only. The operation should not modify any of the items in the aggregate. Further, no state must be retained after the interation has been completed. For instance, the client must not be able to store references to items in the aggregate, since such stored references also lead to the same problems cited in the examples in Section 3.

5. CONCLUSION

As computing power grows, it is safe to say that software will follow suit with ever larger and more complex systems. The good news is that industry has incorporated reuse composition techniques into many mainstream development practices. The bad news is that code reuse — by itself — is an insufficient basis for scalable development. Without compositional reasoning techniques supporting the predictable assembly of components, the intellectual effort required to understand complex systems becomes intractable.

It is well-recognized that sound software engineering disciplines should provide composition mechanisms for code reuse. With promises of drastically reduced development costs, industry has been certainly eager to pick up on this concept. The iterator pattern is but one such example of present-day best practices. As a composition mechanism, iterators support mix-and-match reuse between container components and traveral algorithms. The gain in code reuse is appreciable, but the drawbacks with respect to undermining modular verification cannot be overlooked. While the size of software products has grown, the promise of drastic cost reductions have not yet been realized.

Reusable software components have come at a cost. Surprisingly the purchase price is not the issue, but rather the exhorbitant cost of understanding how reusable components interact with one another when composed, both with each other, as well as with the hosting application. The first phase of the component revolution targeted the cost of writing lines of code. The next phase should target the cost of understanding what is written.

6. REFERENCES

[1] F. P. Brooks, Jr. *The Mythical Man-Month*. Addison-Wesley Publishing Co., Reading, Mass., 1975.

[2] W. Buxton. Less is more (more or less). In P. J. Denning, editor, *The Invisible Future: the seamless integration of technology into everyday life*, pages 145–179. McGrawHill, 2001.

[3] E. W. Dijkstra. Go To statement considered harmful. *Communications of the ACM*, 11(3):147–148, 1968.

[4] E. W. Dijkstra. Notes on structured programming. In O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, editors, *Structured Programming*. Academic Press, London, 1972.

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison Wesley, 1995.

[6] S. S. Owicki and D. Gries. An axiomatic proof technique for parallel programs i. *Acta Informatica*, 6:319–340, 1976.

[7] D. L. Parnas and D. P. Siewiorek. Use of the concept of transparency in the design of hierarchically structured systems. *Communications of the ACM*, 18(7):401–408, 1975.

[8] A. A. Stepanov and M. Lee. The Standard Template Library. Technical Report X3J16/94-0095, WG21/N0482, 1994.

[9] B. W. Weide and J. E. Hollingsworth. Scalability of reuse technology to large systems requires local certifiability. In *Proc. of the 5th Annual Workshop on Software Reuse*, Palo Alto, CA, October 1992.

[10] www.acm.org/acm1. ACM1: Beyond cyberspace, March 2001.