

Representation Inheritance: A Safe Form of “White Box” Code Inheritance

Stephen H. Edwards, *Member, IEEE Computer Society*

Abstract—There are two approaches to using code inheritance for defining new component implementations in terms of existing implementations. Black box code inheritance allows subclasses to reuse superclass implementations as-is, without direct access to their internals. Alternatively, white box code inheritance allows subclasses to have direct access to superclass implementation details, which may be necessary for the efficiency of some subclass operations and to prevent unnecessary duplication of code.

Unfortunately, white box code inheritance violates the protection that encapsulation affords superclasses, opening up the possibility of a subclass interfering with the correct operation of its superclass' methods. *Representation inheritance* is proposed as a restricted form of white box code inheritance where subclasses have direct access to superclass implementation details, but are required to respect the representation invariant(s) and abstraction relation(s) of their ancestor(s). This preserves the protection that encapsulation provides, while allowing the freedom of access that component implementers sometimes desire.

Index Terms—Abstraction function, abstraction relation, behavioral subtype, inheritance, model-based specification, object-oriented, representation invariant, reuse, specialization, subclass.

1 INTRODUCTION

CONVENTIONAL wisdom about how best to use inheritance in object-oriented (OO) programming often centers around the reasoning problems of component clients, not implementers. Most solutions, e.g., adherence to the Liskov Substitutability Principle (LSP) [1], helpfully instruct component designers in the correct way to use specification inheritance. Unfortunately, these solutions do not address the code reuse problems that also affect class designers.

Specifically, code inheritance that allows a subclass to directly access the representation it inherits from its parent—which we might consider *white box* code inheritance—raises serious concerns about safety, correctness, and loss of locality when reasoning about implementations. In contrast, with *black box* code inheritance new features in a subclass are simply additions that are written in terms of the superclass' external client interface. Fig. 1 illustrates these two approaches, where a subclass of a basic list abstraction adds a **Reverse()** operation to the behavior it inherits from its parent. This paper addresses the utility of white box code inheritance as a practical mechanism for component implementers, describes the drawbacks it entails and their theoretical roots, and proposes *representation inheritance*—a safe variety of white box code inheritance that meets practical needs without raising the same concerns.

Section 2 explains why problems arise from white box code inheritance, and defines representation inheritance. Section 3 elaborates the discussion of the problems of white

box code inheritance through a simple example—a two-way list component. Section 4 then shows how representation inheritance can be applied in the example. Section 5 comments on methods of enforcing the restrictions imposed by representation inheritance, and finally Section 6 discusses relationships with previous work.

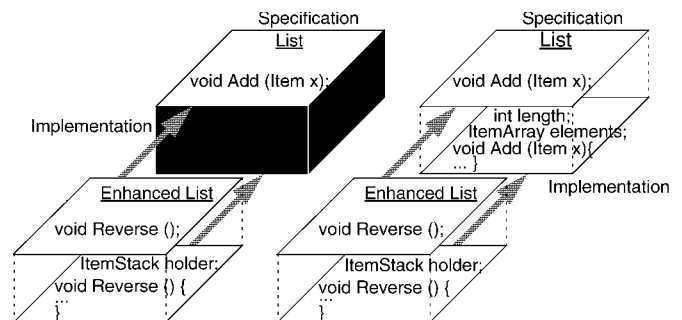


Fig. 1. Black box vs. white box inheritance.

2 THE PROBLEM

When defining a new subclass, an OO programmer often has the option of implementing some or all of a subclass' new features by directly manipulating the data members and/or using the internal operations inherited from its superclass,¹ which we call white box code inheritance. Unfortunately, a subclass implemented using white box code inheritance has a “back door” through the protection that encapsulation normally affords its parent. This leak opens the possibility of subclass code compromising the integrity of an encapsulated object's internal representation. As a result, one can no longer reason about the behavior of a particular method just by

• S.H. Edwards is with the Department of Computer Science, Virginia Polytechnic Institute and State University, Blacksburg, VA 24061.
E-mail: edwards@cs.vt.edu.

Manuscript received June 26, 1996; revised Nov. 11, 1996.

Recommended for acceptance by S.H. Zweben and M. Sitaraman.

For information on obtaining reprints of this article, please send e-mail to: transe@computer.org, and reference IEEECS Log Number S97026.

1. The problems and solutions we discuss may involve either single or multiple inheritance, but the descriptions in this paper are written in terms of single inheritance for simplicity.

looking at the class it is in—any present or future subclass has the potential to interact with it indirectly through the object's internal state in unforeseen ways.

For these reasons, many researchers and practitioners alike have advocated avoiding white box code inheritance completely. Why then would a programmer ever choose to use it? There are two reasons for using white box code inheritance in practical situations. The more commonly cited, but less compelling, reason is efficiency. In some circumstances, subclass operations implemented using black box techniques suffer large space or time performance penalties that could be avoided through the use of white-box code inheritance, as in the example component described in Section 3. When such a case arises, one might suggest simply reimplementing the new class independently, perhaps as a sibling rather than as a descendant of the chosen superclass.

Unfortunately, this leads to the less commonly cited but more compelling reason for using white box code inheritance—avoiding the extra testing and maintenance burden required by duplicating code. The “cut-and-paste”-style reuse involved in reimplementing a class separately in order to add a new method that requires direct access may save coding time, but provides no help for testing or maintenance—the two classes will require twice as much effort as the original class alone [2]. Ideally, one would instead like to consider the newly added code in a subclass to be independent of any inherited code for the purposes of testing and maintenance. While unrestricted white box code inheritance does not admit this possibility, representation inheritance does, as explained in Section 5.

For the remainder of this section, we turn our attention to the hole in class encapsulation that white box code inheritance opens. The difficulties that arise from this leak occur when a subclass either fails to respect its parent's representation invariant, or fails to respect its parent's abstraction relation.

2.1 Respecting Representation Invariants

Internally, an object's methods interact indirectly with each other through the state variables the object encapsulates. Because this interaction is indirect, its success critically depends on assumptions about the meanings attributed to the variables and to changes in their values—assumptions shared by all the methods. A class' *representation invariant* captures exactly these assumptions [3, pp. 72-74].

As an example, consider a class that implements the abstraction of a “list of items.” As shown in Fig. 1, one way to implement such a class is to give it two internal state variables: an array of items called “**elements**,” and an integer called “**length**” recording how much of the array is in use. One might design the methods for this class so the value of **length** always refers to some valid index into the **elements** array—a representation invariant which all of the class methods would share.

Typical OOPs encourage one to encapsulate object state information within a class so that clients cannot violate assumptions that are critical to the correct functioning of the class' methods. However, subclasses may occasionally need direct access to a superclass' internal state (i.e., the specialization interface may provide a different view of the class

than the client interface). This access allows them to manipulate that representation in ways that can violate the representation invariant, introducing “bug-like” behavior in previously correct superclass methods.

To preclude the problems this unchecked freedom can introduce, we propose that:

If a subclass has direct access to the internal state of a superclass, it is likewise obliged to live by and uphold the common assumptions shared by all methods that have direct access to those internal details—e.g., the superclass' representation invariant.

2.2 Respecting Abstraction Relations

A class' client interface is often expressed at a different level of abstraction from its internal representation details (for example, a list described to the client as a mathematical string or sequence, but represented as a linked list of nodes). The correspondence between the internal state representation of an object and its intended conceptual value is expressed as an *abstraction function* [3, pp. 70-71] or, more generally, *abstraction relation* [4], [5].

Again as an example, consider our “list of items” abstraction in Fig. 1. One might assume that the items stored in the list are recorded in the **elements** array, while the **length** state variable records how much of the array is in use. Even so, there are still a variety of alternatives for representing the list's conceptual value in these variables. Which series of contiguous items in the **elements** array form the list: those before **length**, or those after? Does the **length** state variable indicate the index of the last item of the list, or does it refer to the first unused array index *after* the list? These choices are part of the abstraction relation for this class.

Subclasses that are intended to be behavioral subtypes of their superclasses must obey the Liskov Substitutability Principle, meaning that at the level of abstraction in the client interface, objects of the subclass must behave in a manner consistent with the superclass. To ensure that behavioral subtypes behave consistently, in addition to the LSP we propose that:

If a subclass is intended to be a behavioral subtype, yet has direct access to the representation of its superclass, it must live by and uphold the abstraction relation shared by all the methods that have direct access to those internal details. Behavioral substitutability (in the LSP sense) must also be established for any internal superclass methods that are overridden.

2.3 Representation Inheritance

Representation inheritance is a term for code inheritance where a subclass has white-box access to its parent's internals, and the subclass respects the parent's representation invariant. Because most modern OO programming languages (OOPs) provide only one inheritance mechanism, when behavioral subtyping is desired we will consider representation inheritance to encompass the requirement for a subclass to respect both superclass invariants and superclass abstraction relations.

Representation inheritance is built on lessons learned from model-based specification techniques [6], [7], which require one to explicitly state representation invariants and abstraction relations. This solution is notably different from

other proposed solutions, in that it does not involve partitioning a class into groups of interdependent methods that must be considered together when specializing the class. Lamping's work [8], [9], as well as that of Stata and Guttag [10], both indirectly address the difficulties of white-box reuse by grouping methods that depend on common assumptions, signaling to the specialized that these groups need to be examined or changed together. Here, we instead focus directly on the root of the problem—the shared (but often undocumented) assumptions upon which these methods depend. By capturing these assumptions in a representation invariant, it is possible to treat all inherited methods uniformly and independently, while simultaneously documenting exactly the assumptions about state maintenance upon which they depend.

3 AN EXAMPLE: A TWO-WAY LIST COMPONENT

3.1 The Two-Way List Abstraction

To ground the discussion of code inheritance, consider a class implementing the abstract notion of a “two-way list.” Conceptually, the value of a list object is simply a sequence of items that we can visualize as being arranged in a row from left to right. Without loss of generality, consider the left end of the row to be the front or head of the list, and the right end to be the back. As we advance down the list, we can imagine that there is also a “fence” separating the items we have already seen from those that lie ahead—it partitions the row by sitting between two items. This particular list component is “two-way” because we wish to be able to move either left or right in the sequence of items.

One simple formalization of this model of two-way lists is:

```
type Two_Way_List is modeled by (
  left : string of math [Item],
  right : string of math [Item]
)
exemplar 1
initialization
ensures 1. left = empty_string and
         1. right = empty_string
```

This formalization uses the notation of **RESOLVE** [11], although any convenient model-based specification notation could be used [6]. In this formal model of the type, the notion of the “fence” dividing the list of items into two halves is implicit: The value of a list is modeled as two separate “strings” (or sequences) that model the two parts of the row of items to the “left” and “right” of the fence.

With this model in mind, it is possible to decide on the basic operations for two-way lists. The basic operations provided for two-way list objects, as adapted from [7], include:

Move_To_Start (). Moves the fence to the beginning (left) of the list.

Move_To_Finish (). Moves the fence to the end (right) of the list.

Advance (). Moves the fence one position forward (right).

Retreat (). Moves the fence one position backward (left).

Add_Right (x). Adds *x* to the list right after (to the right of) the fence, and returns with *x* having an initial value for its type.

Remove_Right (x). Removes the item immediately following (to the right of) the fence, and returns it in *x*.

At_Start (). Returns true when the fence is at the far left end of the list.

At_Finish (). Returns true when the fence is at the far right end of the list.

In an object-oriented programming language such as C++, we might declare a class realizing this abstract concept as shown in Fig. 2. The C++ **Two_Way_List** class template in Fig. 2 defines a generic component that is parameterized by the type of item in the list. It is similar in several respects to Bertrand Meyer's **BILINEAR** [12, pp. 141–146] and **TWO_WAY_LIST** [12, pp. 154–155, 299–303] components, although Meyer's selection of primary operations and conceptual model differs in several details.

```
template <class Item>
class Two_Way_List
{
private:
    // Prevent assignment
    Two_Way_List& operator = (
        const Two_Way_List& rhs);
    // Prevent copy construction
    Two_Way_List (const Two_Way_List& l);

public:
    // The external interface

    Two_Way_List ();
    ~Two_Way_List ();

    void Move_To_Start ();
    void Move_To_Finish ();
    void Advance ();
    void Retreat ();
    void Add_Right (Item& item);
    void Remove_Right (Item& item);
    void Remove_Right (Item& item);

    Boolean At_Start ();
    Boolean At_Finish ();
};
```

Fig. 2. A C++ two-way list class template.

3.2 Implementing Two-Way List

Given this **Two_Way_List** declaration, we can now turn our attention to how one might implement a two-way list. For the purposes of this paper, the sample implementation will be presented in C++, although any other OO programming language could be used.

One obvious way to implement the **Two_Way_List** component appears in many data structures text books: Use a doubly-linked chain of nodes, where the links are implemented using pointers. A technique that can help with this approach is the use of sentinel nodes. By placing sentinel nodes (or “dummy” nodes)² at either end of the chain, all list operations can be handled uniformly—there are no special cases to handle when operating on either end of the chain.

2. Joe Hollingsworth, in a private communication, noted he regularly uses the approach of dual sentinel nodes when teaching linked representations to his CS1/CS2 students at Indiana University Southeast. Because of the subsequent notable reduction in bugs in student programs, he refers to such sentinels as “smart” nodes.

Given that the sequence of items contained within a **Two_Way_List** object will be held in a doubly-linked chain, only one question remains: What is the exact representation of a **Two_Way_List** object? One obvious choice is to represent a **Two_Way_List** by a pair of pointers: one to record the location of the head of the list, and another to record the location of the fence. Here, we arbitrarily choose for a two-way list object to have two data members, a **pre_front** pointer that points to the sentinel node at the front end of the chain, and a **pre_fence** pointer that points to the node containing the item immediately preceding the fence. Many other combinations would work just as well. This choice is elaborated in Figs. 3 and 4.

```
template <class Item>
class Two_Way_List
{
    // ...
    // The same external declarations as in
    // Figure 3
    // ...

private:
    // The representation of the class:
    struct TWL_Node          // A two-way list node
    {
        Item i;
        TWL_Node* next;      // forward pointer down
                           // the list
        TWL_Node* previous;  // backward pointer up
                           // the list
    };
    TWL_Node* pre-front;    // pointer to first
                           // sentinel
    TWL_Node* pre_fence;    // pointer to node just
                           // before the "fence"
};
```

Fig. 3. The representation of **Two_Way_List**.

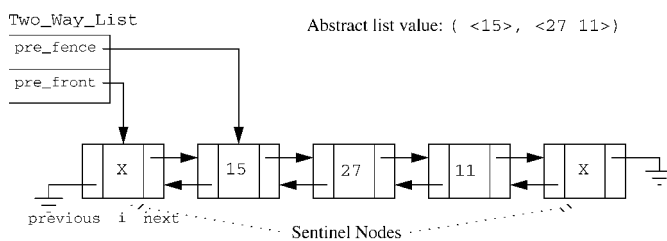


Fig. 4. A doubly-linked chain with sentinel nodes.

Fig. 3 shows the remainder of the **Two_Way_List** C++ class template declaration, including the declaration of the internal **TWL_Node** struct and of the data members holding the **pre_front** and **pre_fence** pointers. Fig. 4 then gives a pictorial representation of an actual **Two_Way_List** object where the items are integers. The sample list chosen has three items in the list (15, 27, and 11), with the fence currently between the first and second elements (after 15 and before 27). Fig. 4 also shows the corresponding abstract value of such a list in terms of the model defined above. Now that the representation choices have been made, providing code for the class methods is a straightforward process that is skipped here.

3.3 An Enhancement

Now that we have an example class component defined and implemented, we can turn our attention to a typical programming task: How can we extend this component with new operations that provide additional capabilities? For the purposes of this paper, we'll restrict ourselves to a simple extension: the addition of an operation called **Swap_Rights** that exchanges the tails (or right halves) of the two lists involved. Fig. 5 shows the **Enhanced_Two_Way_List** class template that adds the new method. Fig. 5 also shows a post-condition describing the behavior of the **Swap_Rights** operation in terms of the type's abstract model, using “#” to denote the value of an object before the method invocation.

Fig. 6 more concretely illustrates the effect of the **Swap_Rights** operation on two lists, where the items are integers. The first list has three elements, 15, 27, and 11, with its fence located between the first and second items. The second has four elements, 14, 87, 9, and 12, with the fence between the second and third items. Fig. 6 shows the effect of invoking the **Swap_Rights** method of the first list, passing the second list as the “rhs” argument to the operation. The sequences of items to the right of the fence in each list are exchanged.

The **Swap_Rights** operation is an interesting additional capability for two-way lists. Using the primary operations shown in Fig. 2, the only way to combine two lists, or separate one list into parts, is through a series of individual add and remove operations. The **Swap_Rights** operation is a useful building block that greatly simplifies the implementation of higher-level operations like concatenation, splitting, splicing, etc. Given the implementation for **Two_Way_List** based on sentinel nodes, what is the safest and most effective way of implementing the **Swap_Rights** operation?

```
template <class Item>
class Enhanced_Two_Way_List : public
Two_Way_List<Item>
{
public:
    void Swap_Rights (
        Enhanced_Two_Way_List& rhs);
    //ensures self =
    //      (#self.left, #rhs.right)
    //      and rhs =
    //      (#rhs.left, #self.right)
};
```

Fig. 5. The **Enhanced_Two_Way_List** class.

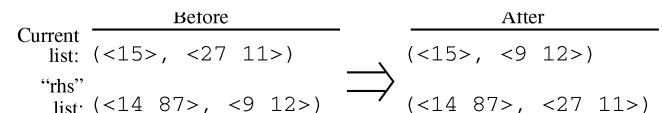


Fig. 6. The effect of **Swap_Rights**.

3.4 Implementing the Enhancement

Note that **Enhanced_Two_Way_List** can be implemented *without* access to the internals of its superclass. By treating the superclass as a black box, **Swap_Rights** could be implemented by moving each item from the right half of the first list over to the second list, one at a time. Then the items

from the right half of the second list have to be moved over to the first, one at a time. This will take time proportional to the number of items in the right halves of both lists.

Clearly, black box code reuse is safe, since subclasses have no more privileges than other clients when it comes to the internal representation of a superclass. One might even be tempted to claim that all code reuse should be achieved through black box methods. Unfortunately, the **Two_Way_List** example illustrates why implementers still turn to white box techniques for some problems.

Two_Way_List's doubly-linked chain representation lends itself to a much more efficient (and less complex!) implementation of **Swap_Rights**. It is only necessary to change two pointer values in each chain in order to exchange the right halves of the two lists, resulting in a constant-time implementation of the operation. Doing this requires access to the representation of the **Two_Way_List** superclass. Thus, even for well-designed components, white box code reuse is occasionally necessary to achieve algorithmic improvements in efficiency.

4 USING REPRESENTATION INHERITANCE

In order for **Enhanced_Two_Way_List** to access its superclass' representation safely, we use representation inheritance. With this approach, the author of a subclass such as **Enhanced_Two_Way_List** is *required* to obey the representation invariant(s) and respect the abstraction relation(s) of its superclass(es). In a language that has support for expressing representation invariants and abstraction relations, this requirement could be automatically enforced. Otherwise, it must be enforced by programming conventions and checked through code reviews and testing. Fortunately, well-defined representation invariants and abstraction relations should make the testing of new subclasses much easier—by verifying that subclass methods do in fact respect these superclass assumptions, the need for retesting of inherited methods or other nonlocal code artifacts is greatly reduced.

In the two-way list example, we can change the declaration of the **Two_Way_List** data members from **private** to **protected**, and write down the representation invariant and abstraction relation for its implementation (perhaps in structured comments, since C++ does not support formal descriptions of these assumptions). The author of the **Enhanced_Two_Way_List** class can then have direct access to the representation of list objects when implementing **Swap_Rights**, as long as the invariant and abstraction relation are respected—both must be respected, since **Enhanced_Two_Way_List** is intended to be a behavioral subtype of **Two_Way_List**. Here, "respected" means the following:

Assume that, before the method is called, the invariant holds on the two-way list object and the abstraction relation gives the correct conceptual value for it. The method then must ensure that upon its completion, the resulting two-way list also satisfies the invariant, and that the abstraction relation gives the correct conceptual value for the new list—one that appropriately reflects the conceptual changes the method was intended to make (i.e., one that conforms to the method's postcondition).

This is the essence of representation inheritance: the flexibility of white box code inheritance is achieved, without giving up the safety afforded by encapsulation of superclass representation information.

The implementation of **Two_Way_List** described in Section 3 relies on several conventions, which taken together form its representation invariant:

- 1) The **TWL_Nodes** within a **Two_Way_List** object are doubly-connected in a single chain.
- 2) The **pre_front** and **pre_fence** pointers refer to nodes within the same chain.
- 3) The unconnected pointers on the sentinel nodes are set to **NULL**.
- 4) The **pre_front** pointer always refers to the sentinel node at the beginning of the chain.

These conventions are stated informally here, but they could be formalized (with some effort). Some programming languages even provide syntactic slots for expressing representation invariants [4], [3].

In practice, a subclass with white box access to its inherited state variables could fail to maintain any one of these four properties. Fig. 7 gives one example of how an implementation of **Swap_Rights** could violate the first clause of the representation invariant. Fig. 7 depicts the representation of the two lists introduced in Fig. 6 in detail, both before and after the call to **Swap_Rights**. In this case, the implementation of **Swap_Rights** has only exchanged the "next" pointers in the two lists, and has failed to properly switch the corresponding "previous" pointers. Now, neither list's representation is a doubly-connected chain—the two chains are cross-connected. While this can rightly be considered a defect in the implementation of the **Swap_Rights** operation, note that many list operations will continue to operate correctly, and the effects may only be detected by a test suite that exercises the inherited operations interleaved with the new addition in a nontrivial way.

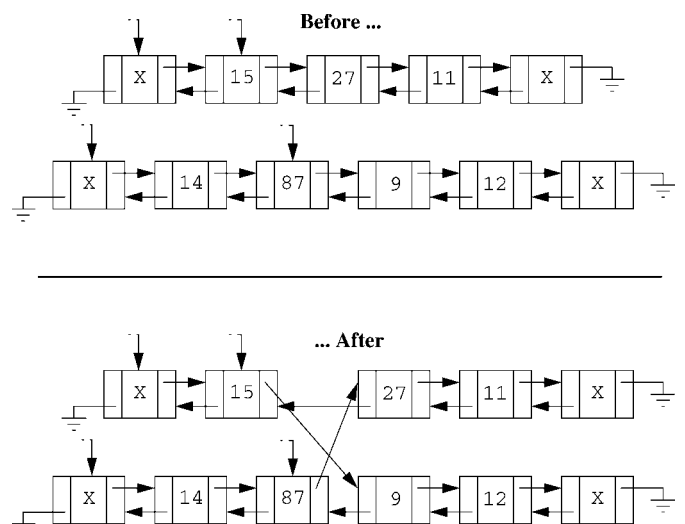


Fig. 7. Violating the representation invariant in **Swap_Rights**.

The abstraction relation then relates representation values to the corresponding conceptual values they realize. It

captures the intentions of the implementer about the “meaning” of the representation—how it encodes the conceptual state that clients reason about. Informally, the doubly-linked chain representation of two-way lists is related to the conceptual model described in Section 3 as follows:

- 1) The entire sequence of items in the list, as well as their order (i.e., `l.left * l.right`), is recorded by the contents and order of the `TWL_Nodes` in the (single) chain of the representation.
- 2) The separation between the “left” and “right” parts of the conceptual value (implicitly denoting the “fence”) is recorded by the `pre_fence` pointer. Specifically, the `pre_fence` pointer points to the `TWL_Node` containing the *last* item in the “left” portion of the list. The “right” portion of the list begins with the node in the chain immediately following the one pointed to by `pre_fence` (i.e., `pre_fence->next`).

Some programming languages also provide syntactic slots for expressing abstraction relations or functions [4], [3].

Continuing the running example, Fig. 8 gives one example of how an implementation of `Swap_Rights` could ignore the second clause of the abstraction relation. Here, the implementation of `Swap_Rights` has only exchanged the trailing halves of the two lists, beginning with the nodes pointed to by the “`pre_fence`” pointers. For the two sample lists under consideration, this behavior does not violate the representation invariant and will not cause the execution of any inherited methods to fail at a later point. Instead, there is a mismatch between the behavior described at the conceptual level and the actual representation. When viewed in the light of the abstraction relation described above, it is clear that `Swap_Rights` does not simply exchange everything to the right of the two fences—it also exchanges the item immediately to the left of the fence in each list. Without a description of the abstraction relation, however, the software engineer who wrote this version of `Swap_Rights` might never see the discrepancy.

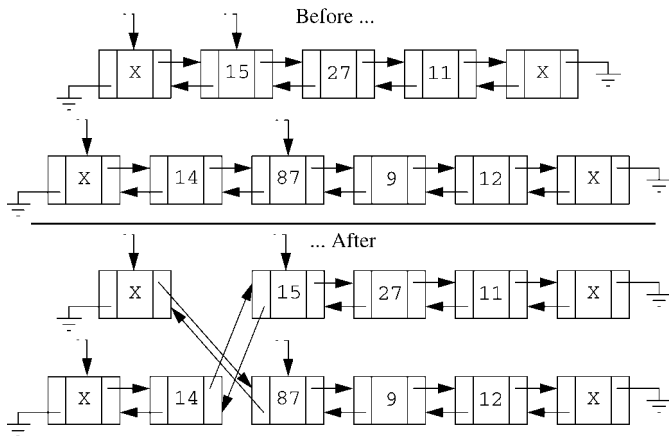


Fig. 8. Violating the abstraction relation in `Swap_Rights`.

The above statements of the representation invariant and the abstraction relation are informal, but they capture critical information necessary for the correct functioning

of the `Two_Way_List` methods. There are many other possible configurations of invariant and abstraction relation that could have been chosen (together with slight differences in the choices about the pointers and node structures used). While any of them may work well, the important point is that **one** choice was made in the implementation of the `Two_Way_List` class, and the implementer of that class used it consistently. The correct operation of `Two_Way_List`'s methods critically depends on this choice (and on consistently following it).

Fig. 9 shows an excerpt of the `Two_Way_List` class declaration with an informal version of the representation invariant and abstraction relation added in comments. Fig. 10 then shows the corresponding implementation of `Enhanced_Two_Way_List`'s `Swap_Rights` method, with comments marking the locations where critical assumptions about the inherited representation invariant and abstraction relation must be checked.

```
template <class Item>
class Two_Way_List
{
    // ...
    // The same external decls. As in Figure 3
    // ...

protected:
    // Allow representation inheritance
    // The representation of the class:
    struct TWL_Node { ... };
    TWL_Node* pre_front; // ptr. to 1st sentinel
    TWL_Node* pre_fence; // ptr. to node just
                        // before the "fence"

    /// Representation invariant:
    /// Let HEAD_SENTINEL and TAIL_SENT-
    /// INEL be the two sentinel TWL_Nodes for
    /// This list. Then:
    /// 1a. HEAD_SENTINEL.next-> ... -> next
    ///     == &TAIL_SENTINEL and
    /// 1b. TAIL_SENTINEL.previous-> ...
    ///     ->previous == &HEAD_SENTINEL
    /// 2. (pre_fence == &HEAD_SENTINEL or
    ///     pre_fence == &HEAD_SENTINEL.next->
    ///     ... ->next) and
    ///     pre_fence != &TAIL_SENTINEL
    /// 3. HEAD_SENTINEL.previous == NULL
    ///     and TAIL_SENTINEL.next ==
    ///     NULL (and nothing else is NULL)
    /// 4. pre_front == &HEAD_SENTINEL

    /// Abstraction relation:
    /// "left" == pre_front-> next-> item,
    ///           pre_front->next->next->item,
    ///           ...
    ///           pre_fence-> item
    /// "right" == pre_fence-> next->item,
    ///             pre_fence->next->item,
    ///             ...
    ///             TAIL_SENTINEL.previous
    ///             ->item
}
```

Fig. 9. The `Two_Way_List` representation invariant and abstraction relation.

```

template <class Item>
void Enhanced_Two_Way_List<Item>::
    Swap_Rights(Enhanced_Two_Way_List& rhs)
{
    /// assert(Two_Way_List-rep_invariant(self) ==
    ///      true);
    /// Assert(Two_Way_List.abs_relation(
    ///      (self.left, self.right),
    ///      (self.pre_front, self.pre_fence));

    TWL_Node* my_tail, rhs_tail;

    my_tail = pre_fence->next;
    rhs_tail = rhs.pre_fence->next;

    pre_fence->next = rhs_tail;
    rhs_tail->previous = pre_fence;

    rhs.pre_fence->next = my_tail;
    my_tail->previous = rhs.pre_fence;

    /// assert(Two_Way_List.rep_invariant(self) ==
    ///      true);
    /// assert(Two_Way_List.abs_relation(
    ///      (self.left, self.right),
    ///      (self.pre_front, self.pre_fence));

    /// My postcondition:
    /// assert ((self->left==self->left)    &&
    ///      (self->right !=rhs->right)    &&
    ///      (rhs->left = rhs->left)        &&
    ///      (rhs->right - #self->right));
}

```

Fig. 10. Implementing `Swap_Rights`.

5 ENFORCING OBLIGATIONS

Representation inheritance relies on a subclass living up to the commitments made by its superclass(es). Thus, the safety afforded by representation inheritance is only as strong as the guarantee we have that the subclass will indeed fulfill its obligations. Further, it is clear that this safety is only as strong as the “tightness” of the representation invariants and abstraction functions documented for each class. Failure to capture all the restrictions that superclass methods rely on can still allow too much freedom to subclasses. With regard to gauging safety, there are three basic approaches to establishing the degree to which subclasses obey their representation inheritance restrictions: formal verification, run-time checking, and testing.

5.1 Formal Verification

In theory, formal verification support is needed to provide complete automatic enforcement of representation invariants and abstraction relations. This necessity is brought about by the fact that some representation invariants or abstraction relations may not be computable, implying conformance may not be checkable by a computer through either run-time checks or testing. This should not be surprising, since conformance to a behavioral specification may be just as difficult to check, depending on the specification notation used. Further, regardless of the enforcement technique used, checking adherence to superclass abstraction relations requires that one have behavioral specifications for both super- and subclasses—something lacking in most present software.

In practice, however, few practitioners are willing to proceed with formal verification at present. Instead, code reviews and testing seem to provide acceptably high confidence levels for conformance with behavioral specifications, so we turn our attention to the natural analogues for enforcing representation inheritance restrictions.

5.2 Run-Time Checking

Without the resources for formal verification, many practitioners feel that run-time checking of representation invariants is critical to enforcement. Eiffel is one language which uses run-time checks consistently to provide some level of enforcement for programming obligations [13].

For example, in the `Two_Way_List` component described in Section 3, one could write a protected method that would operationally check the class’ representation invariant. This method could then be called directly (perhaps using the standard `assert()` macro) in appropriate places (both in `Two_Way_List` methods and methods of its descendant classes) to ensure that the invariant is being maintained. This would certainly provide some degree of confidence that the necessary obligations imposed on subclasses were being observed.

This is certainly a viable approach to representation inheritance enforcement in many cases. It is important to note that it is not always possible to provide run-time checks (i.e., when the representation invariant is not computable). Further, some run-time checks might be considered prohibitively expensive to consider leaving in place in fielded software. As a result, one might ask the question of whether the same degree of confidence could be obtained without requiring the overhead of run-time checking.

5.3 Enforcement through Testing

The primary strategy for using testing to enforce representation inheritance restrictions is to:

- 1) Use run-time checking to operationally test representation invariants at all necessary points during testing.
- 2) Expand white box testing techniques to generate test cases that stress these run-time checks.

This approach utilizes the best features of run-time checking without requiring run-time checks in fielded code. Further, if test case generation takes into account representation invariants, run-time checks during testing may even provide a higher level of confidence the obligations are observed than run-time checks alone.

Simply put, for testing purposes, every class should have a method that operationally checks the representation invariant on its internal state. If all classes export such an operation, it is a simple matter to write “defensive” wrappers for every class that check invariants on entry and exit to every method. By providing such defensive wrappers around superclasses during testing, run-time checking can be systematically inserted where ever it is desired. Generic programming provides an effective way to insert or remove such defensive wrappers without modifying subclasses or their inheritance links [14], [11].

To fully exploit such run-time checks during testing, it is necessary to consider representation invariants when generating test cases. Effectively, the obligation to maintain a

representation invariant becomes an extra part of the behavioral specification of each method (hidden from the eventual clients of a class), and thus is subject to the same test case generation techniques as are used for gauging conventional behavioral conformance.

Once a subclass has been fully tested with run-time checks in place, those checks can be safely removed. Any future subclasses cannot affect code they might inherit, as long as those subclasses obey their representation inheritance obligations. This allows superclasses and subclasses to be validated independently through testing.

5.4 Testing without Representation Inheritance

In **light** of the previous discussion, it is also worth considering the requirements for testing when the restrictions of representation inheritance are not observed or enforced. Perry and Kaiser [15] describe requirements for adequately testing OO programs. They indicate that when subclasses are added to an inheritance hierarchy, not only must one test the newly added methods in these subclasses, one must also *retest* all of the inherited methods. They also indicate that clients of the superclasses need to be retested while using the subclasses. Component regression testing guidelines built on these requirements have also been described by Skublics et al. [16, p. 85].

To most object-oriented programmers, however, this testing advice is counterintuitive and seems to fly in the face of conventional wisdom. Instead of simply testing the newly added code, one must test all methods in every class. While code reuse may have saved some time during the coding phase of development, according to Perry and Kaiser's recommendations, it saves absolutely no effort in testing. From the testing viewpoint, it is almost as if no inheritance had occurred at all—the testing effort required is the same as if all of the superclass code were reproduced from scratch in the new component.

If one is working in a language where the inheritance mechanism normally allows white box code inheritance, such as Smalltalk or Eiffel, then the technical reasons for Perry and Kaiser's recommendation start to make more sense. When a subclass can cause code outside of itself to fail, testing in the context of newly added subclasses becomes a much more involved process.

From this, we can also infer that white box code inheritance provides little if any savings in maintenance effort. Changing code in one class method could conceivably have adverse affects in arbitrarily distant ancestor or descendant classes. Thus, to make maintenance changes or enhancements in one class, the entire root-to-leaf branch of the inheritance hierarchy it lives in must be understood, and then retested after the change—classes fail to provide the fire walls of modularity that programmers expect.

With representation inheritance, the methods inherited from a superclass cannot fail because of defects introduced in subclass method implementations. As a result, changes in a subclass do not require the retesting of inherited code. The virtues of modularity and encapsulation are thus preserved at class boundaries.

6 RELATION TO PREVIOUS WORK

As mentioned in the introduction, representation inheritance is related in spirit to various work on specification inheritance. Liskov and Wing's definition of the subtype relation so that it preserves behavioral abstraction typifies this work [17], [1]. In a similar vein, Leavens and Weihl describe a foundation for the modular verification of OO software built around interpreting inheritance as a behavioral abstraction [5]. These approaches only address the client-side reasoning issues posed by inheritance mechanisms, however, and do not directly address code inheritance.

The notions of representation invariant and abstraction relation (or function) [3] are also taken directly from past work on formal specification and formal verification. Both have been used in languages and methods centered on model-based program specification, including **RESOLVE** [4]. Leavens and Weihl [5] provides a complete formal treatment of representation invariants and abstraction relations in an object-oriented context, and they are naturally extended to other model-based specification approaches, such as those surveyed in Lano and Haughton [18]. This paper gives a more pragmatic presentation in order to familiarize practitioners with the uses of the more theoretical development of representation invariants and abstraction relations presented elsewhere.

The safety problems with white box code reuse have been described by Muralidharan and Weide [19]. They note the efficiency concerns that make white box techniques desirable, but concentrate on clearly delineating the disadvantages that come with breaking encapsulation. They propose no solutions to the problem. The **RESOLVE** programming language [11] does provide the necessary support for representation invariants and abstraction relations, however, and there are plans to add representation inheritance to the language.

As mentioned in Section 2, Lamping [8] also has examined the risks associated with subclass access during specialization. His work is type-system oriented, however, where the solution proposed here derives from model-theoretic specification techniques. Lamping suggests partitioning classes into groups of methods which share assumptions, as documentation for use by programmers writing specializations. He does not specifically address capturing the assumptions themselves, however.

Stata and Guttag [10] have also explored grouping methods for this purpose. Their work is more closely related, since it is also specification-based. They further propose that instance variables can be partitioned along with the methods, splitting a superclass into "modular" chunks that can be treated independently. While this does allow subfacets of an object to be specialized independently, it fails to capture the critical assumptions about module state upon which the methods depend. Stata and Guttag go on to require that if any method in such a group is overridden by a subclass, then all methods in that group must be, which is necessary for safety. Here, we instead explicitly capture the conventions about how state variables are maintained, we do not require methods to be grouped, and we allow any method to be overridden individually.

Perhaps the best-known work that attempts to address the problems discussed here is Meyer's Eiffel [13]. There are several critical differences between Eiffel and the ideas described in this paper, however, which highlight the contributions of the present paper. At first glance, Eiffel appears to have all of the machinery necessary to capture both specification inheritance and representation inheritance built into the language:

- It supports preconditions and postconditions for describing method behaviors.
- It supports **invariant** assertions to capture properties that methods must preserve when they complete.
- It ensures that each subclass inherits the preconditions, postconditions, and invariants of its superclass(es)—descendants must live up to the obligations of their ancestors.

Unfortunately, under practical usage these mechanisms are not enough to ensure the safety that representation inheritance provides.

Classes in Eiffel represent component *implementations*, and there is no linguistic facility for capturing the corresponding component specifications [13, p. 59]. As a result, the mechanisms in the language only support capturing information relevant to the implementation, and other details such as abstraction relations are not addressed.

As a result, Eiffel's **invariant** assertions must serve double-duty:

- 1) Programmers try to use them to capture the *abstract invariant* [3, p. 92], which defines client-visible constraints on an object's conceptual value.
- 2) They should also capture the *representation invariant*, which defines constraints on an object's internal state that is invisible to clients.

Of course, assertions that deal with the hidden state of objects are not helpful for client understanding, so it is common to see Eiffel invariants phrased in terms of publicly visible accessor functions [12] rather than private state variables. As a result, Eiffel's **invariant** assertions become abstract invariants in practice.

This tendency is exemplified by Meyer's version of *TWO_WAY_LIST* [12, pp. 154–155, 299–303]. The Eiffel version has its invariant phrased in terms of publicly visible accessors, and simply constrains client-visible properties, like the relationships between the number of items in the list, the position of the fence, and the values of predicates similar to *At_Start()* and *At_Finish()*. None of the representation-level constraints shown in Fig. 9 are captured.

In addition, the computational nature of Eiffel's assertion mechanism prevents some invariants from being expressed because they are not computable, and discourages programmers from writing down others that are expensive to check. For example, consider a component that implements an associative mapping using a hash table with sorted buckets. The fact that the buckets are maintained in sorted order, and that every key in the mapping is unique, are invariant properties of this implementation. Unfortunately, it is expensive to check these properties at run-time, perhaps prohibitively so. As a result, facets of the component's representation invariant may be ignored by component designers when writing Eiffel assertions.

Finally, the lack of separate specifications in Eiffel ensures that abstraction relations will not be captured. In the *Two_Way_List* example, the assumption that *pre_fence* points to the node *before* the first item in the right half of the conceptual value of the list cannot be captured in an Eiffel **invariant** clause. As a result, subclass methods could violate this assumption, perhaps by leaving a particular list so that the *pre_fence* pointed to the node holding the first item in the right half of the list. This error could potentially cause other methods to fail, or simply have the incorrect behavioral result from the client's point of view. Either way, however, Eiffel assertions cannot address the issue.

While Eiffel's inheritance rules attempt to achieve the same goal as representation inheritance in spirit, in practice none of the assumptions recorded for the *Two_Way_List* example in Fig. 9 would have been captured or checked in a typical Eiffel version of the component. Indeed, none are for the most similar components in Meyer's library. Eiffel fails to provide the safety of representation inheritance for this reason.

7 CONCLUSIONS

Class designers have a choice between black box and white box techniques when they specialize existing classes. While it is always best in principle to use black box code inheritance, there are practical situations where programmers really desire more freedom of access to information encapsulated within superclasses. When these situations arise, white box code inheritance is appropriate.

Unrestricted white box code inheritance is clearly unsafe, however. By breaking the encapsulation of superclasses, it allows subclass implementers to violate assumptions upon which superclass methods depend. This can mean that subclasses actually introduce errors that are only observed through execution of inherited methods, making it impossible to reason about class correctness locally, and seriously complicating the requirements for adequate testing of software.

If the assumptions that classes depend on are described in terms of representation invariants and abstraction relations, then it is possible to address the shortcomings of white box reuse. Representation inheritance is a controlled form of white box code inheritance in which subclasses must respect the representation assumptions of their ancestors. By doing so, subclasses ensure that superclass code assumptions are protected, while simultaneously enjoying the benefits of direct access to superclass state representations. This gives desirable freedom to subclass implementers, while preserving the safety and locality considerations for which all programmers strive.

ACKNOWLEDGMENTS

The author gratefully acknowledges financial support from the National Science Foundation under Grant No. CCR-9311702 and the Advanced Research Projects Agency under Contract No. F30602-93-C-0243 (monitored by the USAF Materiel Command, Rome Laboratories, ARPA order number A714). Bruce Weide also deserves special thanks for

suggesting the ideas that later developed into the notion of representation inheritance presented here.

REFERENCES

- [1] B.H. Liskow and J.M. Wing, "A Behavioral Notion of Subtyping," *ACM Trans. Programming Languages and Systems*, vol. 16, pp. 1,811–1,841, Nov. 1994.
- [2] S.H. Edwards, "An Approach for Constructing Reusable Software Components in Ada," IDA Paper P-2378, Inst. For Defense Analyses, Alexandria, Virginia, Sept. 1990.
- [3] B. Liskov and J. Guttag, "Abstraction and Specification in Program Development," *The MIT Electrical Engineering and Computer Science Series*, Cambridge, Mass.: MIT Press, 1986.
- [4] P. Bucci, J.E. Hollingsworth, J. Krone, and B.W. Weide, "Implementing Components in RESOLVE," *ACM SIGSOFT Software Eng. Notes*, vol. 19, pp. 50–52, Oct. 1994.
- [5] G.T. Leavens and W.E. Weihl, "Specification and Verification of Object-Oriented Programs Using Supertype Abstraction," *Acta Informatica*, vol. 32, no. 8, pp. 705–778, 1995.
- [6] J.M. Wing, "A Specifier's Introduction to Formal Methods," *Computer*, vol. 23, no. pp. 8–24, Sept. 1990.
- [7] M. Sitaraman, L.R. Welch, and D.E. Harms, "On Specification of Reusable Software Components," *Int'l J. Software Eng. and Knowledge Eng.*, vol. 3, no. 2, pp. 207–229, 1993.
- [8] J. Lamping, "Typing the Specialization Interface," *Proc. Conf. OOPSLA'93*, pp. 201–214, ACM, Oct. 1993.
- [9] J. Lamping and M. Abadi, "Methods as Assertions," *ECOOP'94—Object-Oriented Programming, Proc. Eighth European Conf.*, Lecture Notes in Computer Science, vol. 821, pp. 60–80. New York: Springer-Verlag, 1994.
- [10] R. Stata and J.V. Guttag, "Modular Reasoning in the Presence of Subclassing," *Proc. Conf. OOPSLA'95*, pp. 200–213. New York: ACM, 1995.
- [11] M. Sitaraman and B.W. Weide, eds., "Special Feature: Component-Based Software Using RESOLVE," *ACM SIGSOFT Software Eng. Notes*, vol. 19, pp. 21–67, Oct. 1994.
- [12] B. Meyer, *Reusable Software: The Base Object-Oriented Component Libraries*. Hertfordshire, UK: Prentice Hall Int'l, 1994.
- [13] B. Meyer, *Object-Oriented Software Construction*. New York: Prentice Hall, 1988.
- [14] J. Hollingsworth, "Software Component Design-for-Reuse: A Language Independent Discipline Applied to Ada," PhD thesis, Dept. of Computer and Information Science, Ohio State Univ., Columbus, 1992.
- [15] D.E. Perry and G.E. Kaiser, "Adequate Testing and Object-Oriented Programming," *J. Object-Oriented Programming*, vol. 2, pp. 13–19, Jan./Feb. 1990.
- [16] S. Skublics, E.J. Klimas, and D.A. Thomas, *Smalltalk with Style*. Englewood Cliffs, N.J.: Prentice Hall, 1996.
- [17] B.H. Liskov and J.M. Wing, "A New Definition of the Subtype Relation," *ECOOP'93—Object-Oriented Programming, Proc. Seventh European Conf.*, Lecture Notes in Computer Science, vol. 707, pp. 118–141, New York: Springer-Verlag, 1993.
- [18] K. Lano and H. Haughton, eds., *Object-Oriented Specification Case Studies*. Englewood Cliffs, N.J.: Prentice Hall, 1993.
- [19] S. Muralidharan and B.W. Weide, "Should Data Abstraction be Violated to Enhance Software Resuse?" *Proc. Eighth Ann. Nat'l Conf. Ada Technology*, Atlanta, pp. 515–524, ANCOST, Inc., Mar. 1990.



Stephen H. Edwards received the BSEE degree from the California Institute of Technology, Pasadena, and the MS and PhD degrees in computer and information science from the Ohio State University, Columbus. He is currently a visiting assistant professor in the Department of Computer Science at the Virginia Polytechnic Institute and State University. He also maintains a close relationship with the Reusable Software Research Group at the Ohio State University. His research interests include software engineering and reuse, the use of formal methods in programming languages, and information retrieval technology. Dr. Edwards is a member of the IEEE Computer Society, ACM, and Upsilon Pi Epsilon.