

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

Stephen H. Edwards

Dept. of Computer and Information Science
The Ohio State University
395 Drees Lab
2015 Neil Avenue
Columbus, Ohio 43210-1277
Phone: (614) 292-5841
E-Mail: edwards@cis.ohio-state.edu
URL: <http://www.cis.ohio-state.edu/~edwards>

September, 1995

Copyright © 1995 by the author. All rights reserved.

Abstract

Inheritance as a programming language mechanism can be used to achieve several different goals, both in terms of expressing relationships between components and in terms of defining new components “by difference” from existing ones. For defining new component implementations in terms of existing implementations, there are several approaches to using “code inheritance.” 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.

Unfortunately, white box code inheritance violates the encapsulation protection afforded to superclasses, opening up the possibility for subclasses to interfere with the correct operation of 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 function(s) of their ancestor(s). This preserves the protection that encapsulation would have provided, while allowing the freedom of access that component implementers sometimes desire.

Keywords: inheritance, object oriented, representation invariant, abstraction function, reuse

1 Introduction

Inheritance as a programming language mechanism can be used to achieve several different goals, both in terms of expressing relationships between components and in terms of defining new components “by difference” from existing ones [LaL89]. Failing to differentiate the various uses of inheritance, however, can complicate life for both component clients and component implementers difficult [Edw93].

To address the problems of clients, the “Liskov Substitutability Principle” (LSP) has been proposed as the appropriate meaning for subtyping, and thus for publicly visible inheritance relationships, in modern object-oriented (OO) languages [LW94]. Informally, the LSP requires that for one class to be a “subtype” of another, it must respect all of the externally visible behavioral commitments of its superclass. This effectively addresses the client-side difficulties that inheritance can cause; it simplifies client reasoning by ensuring that one’s intuitions about a component’s behavior based on its supertype(s) are always correct with respect to the actual behavior of that component.

Unfortunately, the LSP does not solve the problems that inheritance can cause for component implementers. In contrast to using the inheritance mechanism to represent conceptual relationships between component specifications (for the benefit of clients), inheritance can also be used as a short hand way of defining component implementations by saying how they differ from existing component implementations available in the class library. The potential for leveraging inheritance for code reuse in this sense originally was advocated as one of the primary reuse benefits of OO techniques [Mey88, pp. 33–34, 270]. This is sometimes termed *code inheritance*, to distinguish it from *specification inheritance*, where subclasses are intended to inherit the behavioral specification of their superclass(es).

Using the (sole) inheritance mechanism in a particular language for code inheritance, however, is one common reason why the LSP is violated in practice. Furthermore, code inheritance that allows subclasses to directly access the representation they inherit from their parents—which we might consider “white box” code inheritance—raises serious concerns about safety, correctness, and loss of locality when reasoning about implementations. This paper addresses the need for code inheritance as a practical mechanism for component implementers, describes the drawbacks it entails and their theoretical roots, and proposes a *safe* variety of white box code inheritance that meets practical needs without raising the same concerns.

In Section 2, a simple example is introduced in C++ to illustrate the need for code inheritance techniques. Section 3 then explains the difference between specification inheritance, “black box” code inheritance, and “white box” code inheritance. Section 4 continues the discussion by presenting the technical reason for the lack of safety in white box code inheritance, illustrated by the component introduced in Section 2. A restricted form of white box inheritance called *representation inheritance* is presented in Section 5 that directly addresses these technical concerns.

2 An Example: A Two-Way List Component

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  l.left = empty_string and  
             l.right = empty_string
```

This formalization uses the notation of RESOLVE [W⁺94], although any convenient model-based specification notation could be used [Win90]. In this formal model of the type, the notion of the “fence” dividing the list of items into two halves is implicit—instead, 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 include:

Move_To_Start (1) Moves the fence to the beginning (left) of the list.

Move_To_Finish (1) Moves the fence to the end (right) of the list.

Advance (1) Moves the fence one position forward (right).

Retreat (1) Moves the fence one position backward (left).

Add_Right (1, x) Adds x to the list l right after (to the right of) the fence, and x returns with an initial value for its type.

Remove_Right (1, x) Removes the item immediately following (to the right of) the fence, and returns it in x.

At_Start (1) Returns true when the fence is at the far left end of the list.

At_Finish (1) Returns true when the fence is at the far right end of the list.

```

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

```

Figure 1: A C++ Two-Way List Class Template

In an object-oriented programming language such as C++, we might declare a class realizing this abstract concept as shown in Figure 1. The C++ `Two_Way_List` class template in defines a generic component that is parameterized by the type of items in the list. It is similar in several respects to Bertrand Meyer’s *BILINEAR* component [Mey94, pp. 141–146], although Meyer’s selection of primary operations and conceptual model differs in some details.

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 memory pointers. A technique that can help with this approach is the use of sentinel nodes. By placing sentinel nodes (or “dummy” nodes¹) at either end of the chain, all list operations

¹Joe Hollingsworth 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 [Hol94].

```

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 immediately
                                   // before the "fence"
};

```

Figure 2: The Representation of `Two_Way_List`

can be handled uniformly—there are no “special cases” to handle when operating on the edges of the chain.

Given that the sequence of items contained within a `Two_Way_List` object will be held in a doubly-linked chain of `TWL_Node` objects, 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 end 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 list, and a `pre_fence` pointer that points to the `TWL_Node` containing the item immediately preceding the fence. Many other combinations would work just as well. This choice is elaborated in Figures 2 and 3.

Figure 2 shows the `Two_Way_List` C++ class template declaration, including the declaration of the data members holding the `pre_front` and `pre_fence` pointers. Figure 3 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). Figure 3 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.

Figure 3: A Doubly-Linked Chain With Sentinel Nodes

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. Figure 4 shows the `Enhanced_Two_Way_List` class template that adds the new method. Figure 4 also shows a postcondition 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.

The `Swap_Rights` operation is an interesting additional capability for two-way lists. Using the primary operations shown in Figure 1, 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, and so on. 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? The remainder of this paper answers this question.

```
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)
}
```

Figure 4: The `Enhanced_Two_Way_List` Class

3 “Black Box” Versus “White Box”

In most main-stream OO programming languages (OOPs) such as C++, Eiffel, and Smalltalk, a class is a single unit that combines both a syntactic interface declaration and a corresponding implementation. By convention, programmers may provide “empty” or null implementations for some or all of a class’ methods so that it can be treated more like the definition of an interface only—an *abstract* class—and many languages provide features to support this practice. However, the majority of OO languages provide a single inheritance mechanism through which a subclass inherits **both** the syntactic interface and the (partial or complete) implementation of its superclass(es).

Because this single language mechanism can be used to different ends, here we point out several stylized or idiomatic ways in which inheritance is used. The first is *specification inheritance*, where the programmer intends to convey meaningful information about the behavior of classes related by inheritance. Usually, the term specification inheritance is used when subclass B inherits from superclass A and obeys the same syntactic and behavioral interface (possibly with additional features). Thus, it is fair to say that “the specification of B inherits from the specification of A.”

Alternatively, one may wish to concentrate on creating the subclass such that “the *implementation* of B inherits from the implementation of A.” This is *code inheritance*. This can be carried out in two ways: by black box or white box code inheritance.

First, the author of B may want to reuse the implementation of the superclass A as is without changing it. New features in B are simply additions that are written in terms of A’s external interface. This is *black box* code inheritance, since the subclass need not see inside the representation of its superclass. Some languages provide special support for black box code inheritance; in C++, declaring A’s representation as **private** ensures that subclasses cannot access it directly, enforcing black box techniques. This same sort of code reuse can also be achieved through aggregation (where a B object contains an A object as a data member) without involving inheritance.

In Section 2, the data members of `Two_Way_List` were declared **private**. In C++, this will force the `Swap_Rights` operation in the `Enhanced_Two_Way_List` subclass to be implemented using calls to `Remove_Right` and `Add_Right`. This scenario exemplifies black box code inheritance.

Second, the author of B may want to reuse the implementation of superclass A while having direct access to it. The author would then have the option of implementing some or all of B’s new features by directly manipulating the data members or internal operations inherited from the superclass. This is *white box* code inheritance, since the subclass can see inside and directly manipulate the representation it inherits from its superclass. In Eiffel and Smalltalk, code inheritance is normally white box; in C++, white box inheritance can be achieved by declaring data members **protected** rather than **private** in superclasses.

In the `Two_Way_List` example, consider declaring the data members and `TWL_Node` structure as **protected** rather than **private**. In this case, `Swap_Rights` potentially could be implemented by directly manipulating the pointers of the two lists involved, rather than only in terms of the primary operations publicly visible in `Two_Way_List`.

Of course, since there usually is only one inheritance mechanism in a given OOP, com-

binations of the above goals can be achieved simultaneously. When `Enhanced_Two_Way_List` inherits from `Two_Way_List`, the intent is to capture **both** a specification inheritance relationship for the client's benefit, and a code inheritance relationship for the implementer's benefit. For code inheritance, which approach should be chosen for `Enhanced_Two_Way_List`?

4 The Root of the Problem

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.

Under black box restrictions, the code for `Swap_Rights` must move 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.

Of course, the 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 list in order to exchange their tails, resulting in a constant time implementation of the operation. Doing this requires access to the representation of the `Two_Way_List` superclass. So why not use white box code inheritance?

The problem is that white box code inheritance completely opens up the representation of the superclass to its descendants, effectively breaking the ancestor's encapsulation barrier. That encapsulation barrier was serving a purpose, however, one that is completely undermined by white box code inheritance. In addition to protecting the client from the clutter of unnecessary detail, encapsulation also protects the implementer by ensuring that assumptions the superclass' code relies on cannot be violated by client actions. White box code inheritance provides an opportunity for subclasses to make use of the superclass representation directly, but it also provides the opportunity for subclasses to *violate* assumptions upon which superclass methods rely.

In formalist terms, the assumptions that a class implementation depends on consist of two parts: the *representation invariant* and the *abstraction function* (or, more properly, *abstraction relation* [BHKW94, Lea89]). The representation invariant captures assumptions or programming conventions about the way information is recorded in the component's representation [LG86, pp. 72–74]. For example, the implementation of `Two_Way_List` described in Section 2 relies on several conventions:

1. The `TWL_Nodes` within a `Two_Way_List` 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 easily formalized. Some languages, such as RESOLVE [W⁺94], CLU [LG86], and Eiffel [Mey88], even provide syntactic slots for expressing representation invariants.

The abstraction function then relates representation values to the corresponding conceptual values they realize [LG86, pp. 70–71]. It captures the intentions the implementer had in mind 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 2 as follows:

1. The total 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 separated by 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 languages also provide syntactic slots for expressing abstraction functions or relations [BHKW94], [LG86].

The above statements of the representation invariant and the abstraction function are informal, but they capture critical information that is necessary for the correct functioning of the `Two_Way_List` methods. There are many other possible configurations of invariant and abstraction function 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 the class methods she provided critically depends on this choice (and on consistently following it).

Now the problem becomes clear: while white box inheritance gives subclass implementers direct access to superclass representations, it also allows them to manipulate that representation in ways that violate the representation invariant or are inconsistent with the abstraction function. This is the crux of the “lack of safety” with white box code inheritance. If a subclass implementer is unaware of the programming conventions encoded in the representation invariant and abstraction function, subclass methods could possibly leave a `Two_Way_List` object in a state violating these conventions. This would actually cause inherited methods to fail, even though those methods in isolation work perfectly.

Dewayne Perry and Gail Kaiser [PK90] describe requirements for adequately testing OO programs. They indicate that when subclasses are added to an inheritance hierarchy, it is necessary to retest not only the newly added methods in these subclasses, but also all of the inherited methods. They also indicate that clients of the superclasses need to be retested while using the subclasses. To most object-oriented programmers, this is counterintuitive and seems to fly in the face of conventional wisdom. If, however, one is working in a language where the inheritance mechanism normally allows white box code inheritance, 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.

5 A Safe Approach

The safety risks of white box code inheritance might once again tempt one to denounce white box techniques altogether, sticking religiously to black box methods. Surely a healthy amount of code reuse can still be achieved this way. Unfortunately, this strict approach is unsatisfying to many. The implementer of `Enhanced_Two_Way_List` can no longer use the “obvious” (and efficient) implementation of `Swap_Rights`, and clients who need the extra functionality must pay the price for our principles. Alternatively, by attacking the root of the problem perhaps we can have the best of both worlds.

Unrestricted white box code inheritance is dangerous because it subverts the protection that encapsulation affords the superclass—protection of its representation, ensuring that critical assumptions cannot be violated. If one could provide this same protection, while simultaneously giving subclasses direct access to the representation, the negative effects of code inheritance could be avoided.

Representation inheritance is a restricted form of white box code inheritance that does just that. The author of a subclass such as `Enhanced_Two_Way_List` is *required* to obey the representation invariant(s) and respect the abstraction function(s) of its superclass(es). In a language that has support for representation invariants and abstraction functions built-in, such as RESOLVE, this requirement can be enforced by the language. Otherwise, it must be enforced by convention and checked through code reviews and testing. Fortunately, well-defined representation invariants and abstraction functions 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 non-local 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 function described above (perhaps in structured comments, since C++ does not support them). 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 function are respected. Here, “respected” means the following:

Assume that, before the method is called, the invariant holds on the two-way list object and the abstraction function 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 function gives the correct conceptual value for the new list—one that appropriately reflects the conceptual changes the method was intended to make.

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.

6 Relation to Previous Work

As mentioned in the introduction, in spirit representation inheritance is related to much previous work on specification inheritance. The work of Liskov and Wing in defining the subtype relation so that it preserves behavioral abstraction typifies this work [LW93, LW94]. In a similar vein, Gary Leavens describes a foundation for the modular verification of OO software built around interpreting inheritance as a behavioral abstraction [Lea89, LW95]. These approaches address the client-side reasoning issues posed by inheritance mechanisms, however, and do not directly address code inheritance.

The safety problems with white box code reuse have been described by S. Muralidharan and Bruce Weide [MW90]. They note the efficiency concerns that make white box techniques desirable, but concentrate on clearly delineating the disadvantages that come with breaking encapsulation. Muralidharan and Weide propose no solutions to the problem.

Perhaps the most well-known work that attempts to address the problems discussed here is Bertrand Meyer’s Eiffel [Mey88]. There are several critical differences between Eiffel and the ideas described in this paper, however, which highlight the contributions of representation inheritance. 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 most preserve when they complete.
- It ensures that subclasses inherit preconditions, postconditions, and invariants—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 facility for capturing the corresponding component specifications in the language [Mey88, p. 59]. As a result, the mechanisms in the language only support capturing information relevant to the implementation, and other details such as abstraction functions are not addressed. In practice, however, programmers try to capture as much of the intended specification as is reasonable within the assertions the language does support, resulting in conflicting goals.

Eiffel’s **invariant** assertions typify this conflict; they must serve double-duty:

1. They should capture the *abstract* invariant [LG86, 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 [Mey94] rather than private state variables, turning them into abstract invariants.

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. 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 functions (or simulation relations [LW95]) will not be captured. In the `Two_Way_List` example, the assumption that the `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 ignore the issue. While Eiffel's inheritance rules attempt to achieve the same goal as representation inheritance in spirit, they fail for these reasons.

7 Conclusions

Conventional wisdom about how to best use inheritance in OO programming often centers around the reasoning problems of component clients, not implementers. Most solutions, like adherence to the Liskov Substitutability Principle (LSP), helpfully instruct component designers in the correct way to use specification inheritance. Unfortunately, these solutions do not address the code reuse problems with which class designers are simultaneously grappling.

Typically, a combination of specification and code inheritance, co-mingled via a single language mechanism, is used to define 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 classes depend on are described in terms of representation invariants and abstraction functions (or 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.

Acknowledgements

The author gratefully acknowledges financial support from the National Science Foundation (grant number CCR-9311702) and the Advanced Research Projects Agency (contract number 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

- [BHKW94] Paolo Bucci, Joseph E. Hollingsworth, Joan Krone, and Bruce W. Weide. Implementing components in RESOLVE. *ACM SIGSOFT Software Engineering Notes*, 19(4):40–52, October 1994.
- [Edw93] Stephen H. Edwards. Inheritance: One mechanism, many conflicting uses. In Larry Latour, editor, *Proceedings of the Sixth Annual Workshop on Software Reuse*, November 1993.
- [Hol94] Joseph Hollingsworth. Indiana University, personal communication, 1994.
- [LaL89] Wilf R. LaLonde. Designing families of data types using exemplars. *ACM Transactions on Programming Languages and Systems*, 11(2):212–248, April 1989.
- [Lea89] Gary T. Leavens. *Verifying Object-Oriented Programs That Use Subtypes*. PhD thesis, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, February 1989.
- [LG86] Barbara Liskov and John Guttag. *Abstraction and Specification in Program Development*. The MIT Electrical Engineering and Computer Science Series. MIT Press, Cambridge, MA, 1986.
- [LW93] Barbara H. Liskov and Jeannette M. Wing. A new definition of the subtype relation. In *ECOOP'93—Object-Oriented Programming, 7th European Conference*, volume 707 of *Lecture Notes in Computer Science*, pages 118–141, New York, NY, 1993. Springer-Verlag.
- [LW94] Barbara H. Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16:1811–1841, November 1994.
- [LW95] Gary T. Leavens and William E. Weihl. Specification and verification of object-oriented programs using supertype abstraction. *Acta Informatica*, to appear 1995.

- [Mey88] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, New York, NY, 1988.
- [Mey94] Bertrand Meyer. *Reusable Software: The Base Object-Oriented Component Libraries*. Prentice Hall International, Hertfordshire, UK, 1994.
- [MW90] S. Muralidharan and Bruce W. Weide. Should data abstraction be violated to enhance software reuse? In *Proceedings of the 8th Annual National Conference on Ada Technology*, pages 515–524, Atlanta, GA, March 1990. ANCOST, Inc.
- [PK90] Dewayne E. Perry and Gail E. Kaiser. Adequate testing and object-oriented programming. *Journal of Object-Oriented Programming*, 2(5):13–19, January/February 1990.
- [W⁺94] Bruce W. Weide et al. Special feature: Component-based software using RESOLVE. *ACM SIGSOFT Software Engineering Notes*, 19(4):21–67, October 1994.
- [Win90] Jeannette M. Wing. A specifier’s introduction to formal methods. *IEEE Computer*, 23(9):8–24, September 1990.