Inheritance for Software Reuse: The Good, The Bad, and The Ugly

Murali Sitaraman and David Eichmann

Dept. of Statistics and Computer Science
West Virginia University
Morgantown, WV 26506
(murali, eichmann)@cs.wvu.wvnet.edu

Abstract

Inheritance is a powerful mechanism supported by object-oriented programming languages to facilitate modifications and extensions of reusable software components. This paper presents a taxonomy of the various purposes for which an inheritance mechanism can be used. While some uses of inheritance significantly enhance software reuse, some others are not as useful and in fact, may even be detrimental to reuse. The paper discusses several examples, and argues for a programming language design that is selective in its support for inheritance.

Keywords: extensions, implementation, inheritance, reusable software components, specification

1 Introduction

Inheritance has been widely recognized as an important mechanism for constructing new reusable software components from existing components [Liskov 87, Meyer 88]. This paper proposes a taxonomy for inheritance-based reuse. Some members of this taxonomy permit effective reuse and must be supported by object-oriented programming languages. However, there are other uses of inheritance that do not enhance reuse, and may even be detrimental to reuse. A language must, therefore, be selective in its support for inheritance.

2 A Framework for Discussion

We will use the "3C reference model" (for reusable software components) as the basis for our taxonomy in this paper [Edwards 90, Latour 90, Tracz 90b]. This model is the result of the discussions at the Reuse in Practice Workshop (July 1989) and the Workshop on Methods and Tools for

Reuse (June 1990). The 3C model associates three key ideas with reusable software components as summarized in [Weide 91]:

Concept An abstract (formal) specification explaining (precisely) what functionality is provided by a software piece, without saying how the functionality can be realized.

Content (for a concept) A piece of code that (precisely) describes the data structures and algorithms for implementing (in a formal, programming language) the concept.

Context A statement (precisely) explaining the environment (using formal notations) in which a concept or content is presented.

Several contents may implement the same concept. They will all be identical with respect to their functionality, but may be different with respect to their performance behaviors (e.g., space or time characteristics) To use a component, a client (user) needs to understand only its concept. The functional correctness of the client program depends only on this concept [Parnas 72]. The client will remain unaffected even if it switches from one content of the concept to another. These observations have important implications for modification and maintenance of software built from reusable components. We have used a similar model in our research to characterize the nature of a components industry that would evolve when current reuse efforts prove successful [Muralidharan 90b, Sitaraman 90, Weide 91].

3 A Classification of Uses of Inheritance

Inheritance can be used, in the above framework, to extend (or modify), and thus, reuse each aspect of a software component - concept, content, and context. This section presents a classification of such uses of inheritance. We restrict our attention in this paper to inheritance of concepts and contents alone. It is important to note that our classification has nothing to do with the actual inheritance mechanisms supported in object-oriented languages; it deals only with the possible uses of inheritance.

3.1 A Classification Scheme

The critical issues in inheritance mechanisms from a reuse perspective are who inherits, what is inherited, and what can be done with that which is inherited. We consider each of these issues in turn. This discussion supports both single and multiple inheritance.

(i) Who inherits and from whom

Specification inheritance occurs when parents are concepts. Implementation inheritance occurs when parents are contents. These definitions are similar in spirit to those found in [LaLonde 89]. The heir can be either a concept or a content for either specification or implementation inheritance. The only combination that is not meaningful (based on our definitions) is inheritance of a content by a concept.

(ii) What parts are inherited

We focus our attention here only on formally defined concepts and contents that implement these concepts. A formal concept for a data abstraction has two parts: the abstract model(s) that describes the type(s) provided by the concept, and the abstract specifications of the operations on the provided type(s). (When a concept provides only a procedural abstraction, only the second part is present.) The appendix describes an example concept - a formal specification of a stack data abstraction.

A content for a concept defining a data abstraction also has two parts: the representation(s) of the provided type(s), and the code for the provided operations.

An heir may selectively inherit only parts of a concept or content.

(iii) The mode of inheritance

An heir may inherit parts of a concept or a content for read only or for redefining purposes. When a heir redefines a part of its parent, the re-definition may or may not be "compatible" with its parent. The definition of compatibility depends on what is inherited; usually it involves restricting the domain of one or more inherited types.

3.2 Specification Inheritance - Inheritance of a Concept

A concept can be inherited by either another concept or by a content. (When multiple concepts are inherited, different concepts could be affected differently.)

3.2.1 Inheritance by a concept

First, we define what it means for an heir to compatibly redefine its parent's parts. The abstract model A of an heir is compatible with the corresponding model B of its parent, only if the parent concept is unaffected by substituting A for B. (For, example, the heir's model should satisfy the invariants in the parent concept.) An operation P in an heir is compatible with the corresponding operation Q in its parent, only if P's pre-condition is no stronger than Q's and P's post-condition is no weaker than Q's.

Because few object-oriented programming languages have included rigorous formal specifications, the issues raised by some of these combinations have not been explored in the community. In table 1, the meaningful combinations are marked with a •. For want of space, we discuss the meaning and relevance of only some of these combinations here.

(i) Read only - both abstract model(s) and operations

Table 1: Inheritance a concept by another concept

Mode	*None*	*Model*	*Operations*	*Both*
Read only				•
Read and compatible redefine			•	•
Read and incompatible redefine		•	•	•

This is probably the most common mode for specification-based extensions. For example, a basic stack concept may provide the operations push, pop, and is-empty. This concept may be extended to include, say, an operation to reverse a stack. The typical reason for extending a concept is either that the original concept is not sufficiently complete or that it is in the developmental stage. In [Sitaraman 91], we have argued for a reason to extend even well-designed concepts for building efficient implementations. Without the ability to inherit a concept, this is impossible to do. This use of inheritance can enhance reuse and programming languages must support this possibility.

(ii) Read all and compatibly redefine - operations

Sometimes, it may be essential to create a new concept by modifying the specifications of an existing concept. If the changes are compatible (according to the definitions of compatibility in this section) with the specifications in the original concept, then the new concept can be used wherever the original concept was being used. For example, a stack concept can inherit from a bounded stack concept, and relax the pre-condition on the push operation. Intuitively, an unbounded stack can be used wherever a bounded stack can be used.

(iii) Read all and incompatibly redefine - operations

If a stack concept is already defined, and someone extends it to be a bounded stack, this will be the case. In this case, the model of the stack has to be extended to include a bound. In addition, while the original stack will have no pre-condition for the push operation, the heir concept will have one. This is incompatible because the heir has a stronger pre-condition. Intuitively, a bounded stack cannot be used where an unbounded stack was previously used. If the abstract model of a type is redefined, the specifications of most, if not all, operations will have to be redefined. In this case, inheritance may result in some, but not in significant reuse.

3.2.2 Inheritance by a content

When a concept is inherited by a content, only few combinations are meaningful.

(i) Read only - both abstract model(s) and operations

This is the most normal case of concept inheritance by content. To implement a concept, a content must inherit it for read only purposes. Of course, more than one content may inherit the same concept in this mode, resulting in multiple implementations of a concept. This is an important use of inheritance [Meyer 88, Sitaraman 90], and is crucial for the evolution of a successful components industry.

Table 2: Inheritance of a concept by a content

Mode	*None*	*Model*	*Operations*	*Both*
Read only				•
Read and compatible redefine			•	
Read and incompatible redefine			•	

(ii) Read all and compatibly redefine - operations

Sometimes, an implementation of an operation may require fewer pre-conditions than stated in its specifications and ensure more post-conditions. In this case, the operation does more than what the specification of the operation needs it to do. For example, an operation may reclaim unused storage even if it is not explicitly stated in its specification.

(iii) Read all and incompatibly redefine - operations

This is an implementation where the code for some operations do not provide the behavior specified in the concept. In otherwords, this content does not correctly implement its concept, i.e., it is incorrect. Clearly, this is a bad use of inheritance.

3.3 Implementation Inheritance - Inheritance of a Content

A content can be inherited only by another content. The concept of the parent and the heir may or may not be the same. Just as in the case of a concept, a content may be inherited in three different modes. A content redefines a representation compatibly only if the heir's representation when used in the place of the parent's representation leaves the parent content unaffected. A compatible redefinition of an operation does not violate the specification of the operation in the parent content's concept. Content inheritance may also be selective. (When multiple contents are inherited, different contents could be affected differently.)

(i) Read only - both representation(s) and operations

Apparently, this use of content inheritance is to permit an heir take advantage of the otherwise hidden details of another content. For a well-designed component, providing "sufficiently complete" functionality, all essential details of the content may be accessed by calling the operations in its concept. This use of inheritance helps in avoid a few procedure calls, but clearly violates the principle of information hiding. This can lead to serious pitfalls, including poor developmental independence and maintainability [Muralidharan 90a, Raj 90]. This may, however, be a useful way of keeping track of different versions of the same content.

(ii) Read all and compatibly redefine - operations

This case of content inheritance probably is most useful to keep track of the different versions of an evolving content.

(iii) Read all and compatibly redefine - both rep. and operations

Table 3: Inheritance of a content by a content

Mode	*None*	*Rep.*	*Operations*	*Both*
Read only		•		•
Read and compatible redefine	•	•	•	•
Read and incompatible redefine	•		•	•

Sometimes, when a new concept is created by compatibly redefining an existing concept, it may be possible to create a content for the new concept by compatibly redefining a content of the original concept. The new content, in this case, will also be a content for the original concept.

Incompatible redefinitions may be useful in some rare cases. It must be noted, however, that all uses of content inheritance suffer from certain basic problems because their violate information hiding.

4 Discussion

Object-oriented programming languages typically support one mechanism for inheritance that is useful for various purposes. While this is important, we believe the mechanism should be discriminatory and allow only certain uses. We have shown that most uses of specification inheritance are useful and some uses of implementation inheritance may not be desirable. The components of a library that would evolve from discriminatory uses of inheritance will facilitate construction of software systems that are reliable, modifiable, and maintainable.

The work presented here can be formalized, and extended to compare inheritance mechanisms in various languages and the forms of uses that are supported. Also, it is important to identify interesting examples for the various classes, thereby leading to a better understanding of the usefulness of these classes. The present scheme should also be enhanced to account for context inheritance.

References

[Edwards 90] Edwards, S., "The 3C Model of Reusable Software Components," *Third Annual Workshop: Methods and Tools for Reuse,* Syracuse, 1990.

[LaLonde 89] LaLonde, W. R., "Designing Families of Data Types Using Exemplars," *ACM Transactions on Programming Languages and Systems* 11, 2, April 1989, pp. 212-248.

[Latour 90] Latour, L., T. Wheeler, and W. Frakes, "Descriptive and Predictive Aspects of the 3Cs Model: SETA1 working group summary," *Third Annual Workshop: Methods and Tools for Reuse,* Syracuse, 1990.

[Liskov 87] Liskov, B., "Data Abstraction and Hierarchy," *Addendum to the Procs. of OOPSLA 1987,* Orlando, FL, pp. 17-34.

[Meyer 88] Meyer, B., *Object-Oriented Software Construction,* Prentice-Hall, Englewood Cliffs, NJ, 1988.

[Muralidharan 90a] Muralidharan, S., and B. W. Weide, "Should Data Abstraction Be Violated to Enhance Software Reuse?," *Proc. 8th Annual National Conf. on Ada Technology,* ANCOST, Inc., Atlanta, GA, Mar. 1990, 515-524.

[Muralidharan 90b] Muralidharan, S., and B. W. Weide, "Reusable Software Components = Formal Specifications + Object Code: Some Implications," *3rd Annual Workshop: Methods and Tools for Reuse,* Syracuse Univ. CASE Center, Syracuse, NY, July 1990.

[Parnas 72] Parnas, D. L., "On the Criteria to be Used in Decomposing Systems into Modules," *Communications of the ACM* 15, 12, December 1972, 1053-1058.

[Raj 90] Raj, R. K., "Code Inheritance Considered Harmful," *3rd Annual Workshop: Methods and Tools for Reuse,* Syracuse Univ. CASE Center, Syracuse, NY, July 1990.

[Sitaraman 91] Sitaraman, M. and D. Eichmann, *Building and Using Efficient Extensions: An Approach Based on Inheritance,* TR 91-01-02, Dept. of Stat. and Comp. Science, West Virginia University, Morgantown, WV 26506.

[Sitaraman 90] Sitaraman, M., *Mechanisms and Methods for Performance Tuning of Reusable Software Components,* Ph. D. Dissertation, Dept. of Comp. and Info. Science, Ohio State Univ., Columbus, OH, July 1990.

[Tracz 90a] Tracz, W., "Where Does Reuse Start?," *ACM SIGSOFT Software Engineering Notes* 15, 2, pp. 42-46.

[Tracz 90b] Tracz, W., "The Three Cons of Software Reuse," *Third Annual Workshop: Methods and Tools for Reuse,* Syracuse, 1990.

[Weide 91] Weide, B. W., W. Ogden, and S. H. Zweben, "Reusable Software Components," *Advances in Computers,* M. C. Yovits, eds., Academic Press, New York, NY, 1991.

[Weide 86] Weide, B. W., *Design and Specification of Abstract Data Types Using OWL,* OSU- CISRC-TR-86-01, Dept. of Comp. and Info. Science, Ohio State Univ., Columbus, OH, March 1986.

[Wing 90] Wing, J. M., "A Specifier's Introduction to Formal Methods," *IEEE Computer* 23, 9, September 1990, pp. 8-24.

5 Appendix: An Example Concept

Figure 1 shows a concept for a Stack component explained using a model-based specification. For our purposes, it does not matter which specific specification language and/or programming language is used in explaining concepts and contents. The concepts could use any of the formal methods described in [Wing 90]. We have chosen a dialect of RESOLVE [Weide 91].

Here, the type Stack is modeled as a mathematical STRING of Items and the operations are formally specified using mathematical string functions EMPTY and POST. Each operation has been explained using two clauses: a requires clause that states what must be true of the arguments

```
concept Stack_Template (type Item)
        type Stack is modeled by STRING (Item)
                initially  for all s: Stack, s = EMPTY

        operation Push(s: Stack, x: Item)
                ensures  s = POST(s, x) and Item.Init (x)

        operation Pop(s: Stack, x: Item)
                requires s /= EMPTY
                ensures  #s = POST (s, x)

        operation Is_Empty(s: Stack) return Boolean
                ensures  Is_Empty iff s = EMPTY
end Stack_Template
```

Figure 1: Formal Specification of a Stack Abstraction

passed to the operation and an ensures clause that states what will be true of the parameters at the completion of the operation. In the ensures clause, the notation "#x" for a parameter x denotes its incoming value and RxS denotes its value when the operation returns. (In the requires clause, the variables always denote the incoming values.) The specification of Push, for example, states that the value of the returned stack (s) is its incoming value (#s) with the incoming value of x (#x) appended to the end. The returned value of x is an initial value of the type Item.

6 About the Authors

Murali Sitaraman is an Assistant Professor of Computer Science at the West Virginia University. He holds a Ph. D. in Computer Science from The Ohio State University and M. E. (distinction) from the Indian Institute of Science. His current research interests are in data structures and algorithms, programming languages, software reuse, verification and validation, and some aspects of distributed computing. His Internet address is murali@cs.wvu.wvnet.edu.

David Eichmann is currently an Assistant Professor of Computer Science at West Virginia University and heads the Software Reuse Repository Lab (SoRReL). He received his doctorate in computer science from The University of Iowa, and taught in Seattle University's Master's in Software Engineering Program before joining WVU. His research interests focus on software reuse systems, particularly in the representation and retrieval of life cycle artifacts, and on database systems, particularly in type systems for databases. SoRReL is currently supported in part by NASA's RBSE project (previously known as AdaNet).